PCM-MIO-G-1 Windows Device Driver Package

1 Introduction

1.1 The WinSystems PCM-MIO-G-1 device is a versatile, PC/104 analog input, analog output, and digital I/O board designed for high accuracy and high channel count analog and digital I/O. The board is based upon Linear Technologies' state of the art precision converters and voltage references which require no external calibration.

1.2 The WS16C48 ASIC provides 48 digital I/O lines addressed through six contiguous registers. Each I/O line is individually programmable for input, output, or output with read back operation. The ASIC supports up to 24 event sense lines which can sense a positive or negative transition on the input. These can be used to generate a system interrupt request.

1.3 The LTC-1859 provides two 8-channel, 16-bit Analog-to-Digital (A/D) converters with sample-and-hold-circuit support. Input ranges supported are: 0-5V, 0-10V, ±5V and ±10V. The data sheet is found at http://www.analog.com/media/en/technical-documentation/data-sheets/185789fb.pdf.

1.4 The LTC-2704 provides two 4-channel, 12-bit Digital-to-Analog (D/A) converters. Output ranges supported are: 0-5V, 0-10V, ±5V or ±10V, +/-2.5V, and -2.5V to 7.5V. The data sheet is found at http://www.analog.com/media/en/technical-documentation/data-sheets/2704fd.pdf.

1.5 For more detailed explanations of the PCM-MIO-G-1 software requirements, refer to the PCM-MIO Product Manual found at https://www.winsystems.com/wp-content/uploads/product-manuals/pcm-mio-g-1-pm.pdf.

1.6 The PCM-MIO-G-1 driver package is designed for and has been verified with 32-bit and 64-bit versions of Microsoft WES 7 and Windows 10.

2 Installation

2.1 Before installing the device, verify that the board jumpers are configured for the desired I/O base address. In the BIOS set-up menus, verify that the selected resources are not used by other devices.

2.2 The driver, support files, and console applications are supplied in a zip file. The following files are included:

- pcmmio.sys – Windows device driver
- pcmmio.inf – Windows installation file
- pcmmio.cat – Windows catalog file
- WdfCoinstaller01011.dll – Windows co-installer
- pcmmioDLL.dll – Windows DLL
- pcmmioDLL.lib – Windows library file
- pcmmioDLL.h – driver include file
- flash.cpp – Windows application source
- poll.cpp – Windows application source
- getvolt.cpp – Windows application source
- getvolt_irq.cpp – Windows application source
- getall.cpp – Windows application source
- adcBuff.cpp – Windows application source
- adcRepeat.cpp – Windows application source
- setvolt.cpp – Windows application source
- setvolt_irq.cpp – Windows application source
- dacBuff.cpp – Windows application source
- flash.exe – Windows application
- poll.exe – Windows application
- getvolt.exe – Windows application
- getvolt_irq.exe – Windows application
- getall.exe – Windows application
- adcBuff.exe – Windows application
- adcRepeat.exe – Windows application
- setvolt.exe – Windows application
- setvolt_irq.exe – Windows application
- dacBuff.exe – Windows application
- vcredist_x86 or vcredist_x64 - Microsoft Visual C++ Redistributable

2.3 Installation is accomplished via the 'Add legacy hardware' selection found in the Action menu of the Windows Device Manager. Navigate to the drive and folder containing the driver files and select *pcmmio.inf*. The Windows installer will copy the *pcmmio.sys* driver file to the appropriate directory in the Windows installation.

2.4 If multiple boards are stacked, the driver must be loaded for each device. Each instance of the driver should be configured to match the jumper configuration. The I/O range is 32 sequential bytes.

2.5 In Device Manager, the PCM-MIO Device(s) will appear under the System Devices item. The desired hardware configuration can be selected under the Resources tab of the PCM-MIO Device Properties window. A reboot may be required after resource selection is complete.

2.6 The included console applications can be used to verify driver installation and functionality. Usage of the programs is described later in this document.

3 Driver Overview and Architecture

3.1 The file *pcmmio.sys* is a Windows Driver Foundation kernel-mode (KMDF) driver which facilitates access to the underlying hardware.

3.2 The file *pcmmioDLL.dll* is a Windows Dynamic-Link Library which provides more user-friendly functions to access the device.

3.3 The driver utilizes the I/O Control (IOCTL) Request framework to control the register set of each PCM-MIO device. Data is passed to and from the driver utilizing input and output buffers.

4 Driver Usage

4.1 The *pcmmioDLL.h* file included in the driver distribution contains the function definitions to be used by an application to communicate with the pcmmio driver. This file is included in Appendix A: pcmmioDLL.h.

4.2 An application calls the function *InitializeSession* to open the driver. This is required before any of the other functions can be called. The following example opens the WinSystems *pcmmio* driver. If a zero is returned, then the driver has been successfully initialized. Any other returned value indicates that an error has occurred and the device is unusable.

```
#define DEVICE 1  // access device 1

if (InitializeSession(DEVICE))
      printf(“Error opening device\n”);
```

4.3 Once the driver is initialized, the other functions can be used to control the PCM-MIO. Following is a description and sample code for each function. Each function requires a *device* parameter which selects the desired PCM-MIO to access.

All functions below will follow the same return value model. If a zero is returned, the function was successful. Otherwise there was an error and the function did not complete. Specific error codes are defined later in this document.

4.4 DIO Functions

4.4.1 int DioResetDevice(unsigned int device)

This function resets the WS16C48 ASIC to a known state. All bits are defined as outputs at state zero and all interrupts are disabled. Any locked port is unlocked.

```
if (DioResetDevice(2)) // reset device 2
      printf(“Error resetting device\n”);
```

4.4.2 int DioSetIoMask(unsigned int device, unsigned int *portState)

Configures the mask for all digital I/O (DIO) ports provided in the memory location *portState*. A mask bit defined as a zero is an input, and a mask bit defined as a one is an output.

```
unsigned int dev = 1;
unsigned int mask[6]; // 48 I/O = 6 Ports

mask[0] = 0xFF; // all bits output
mask[1] = 0x00; // all bits input
mask[2] = 0xF0; // upper nibble output, lower nibble input
mask[3] = 0xFF; // all bits output
mask[4] = 0xAA; // alternating input/output bits
mask[5] = 0x55; // alternating input/output bits

if (DioSetIoMask(dev, mask))
      printf(“Error configuring port masks\n”);
```

4.4.3 int DioGetIoMask(unsigned int device, unsigned int *portState)

Retrieves the mask for all DIO ports and stores them in the memory locations provided by the array parameter *portState*. A mask bit defined as a zero is an input, and a mask bit defined as a one is an output.

```
unsigned int dev = 3;
unsigned int mask[6]; // 48 I/O = 6 Ports

if (DioGetIoMask(dev, mask))
      printf(“Error reading port masks\n”);
```

4.4.4 int DioReadAllPorts(unsigned int device, unsigned int *readValueArray)

Reads the current value of all available DIO ports and stores them in the memory locations provided by the array parameter *readValueArray*.

```
unsigned int dev = 2;
unsigned int read_value[6]; // 48 I/O = 6 Ports

if (DioReadAllPorts(dev, read_value))
      printf(“Error reading all ports\n”);
```

4.4.5 int DioReadPort(unsigned int device, int port, unsigned int *readValue)

Reads the current value of the selected DIO port and stores it in the memory location provided by the parameter *readValue*.

```
unsigned int dev = 1;
unsigned int read_value;
int port = 1;

if (DioReadPort(dev, port, &read_value))
      printf(“Error reading port %d\n”, dev, port);
else
      printf(“Port %d = 0x%02x\n”, dev, port, read_value);
```

4.4.6 int DioReadBit(unsigned int device, int bit, unsigned int *bitValue)

Reads the current value (0 or 1) of the selected DIO bit and stores it in the memory location provided by the parameter *bitValue*.

```
unsigned int dev = 1;
unsigned int bit_value;
int bit = 40;

if (DioReadBit(dev, bit, &bit_value))
      printf("Error reading bit %d\n", dev, bit);
else
      printf("Bit %d = %d\n", dev, port, bit_value);
```

4.4.7 int DioSetBit(unsigned int device, int bit)

Sets the selected DIO bit. If the bit is part of a locked port, an ACCESS_ERROR value is returned.

```
unsigned int dev = 4;
int bit = 24;

if (DioSetBit(dev, bit))
      printf("Error setting bit %d\n", dev, bit);
```

4.4.8 int DioClearBit(unsigned int device, int bit)

Clears the selected DIO bit. If the bit is part of a locked port, an ACCESS_ERROR value is returned.

```
unsigned int dev = 1;
int bit = 0;

if (DioClearBit(dev, bit))
      printf("Error clearing bit %d\n", dev, bit);
```

4.4.9 int DioWritePort(unsigned int device, int port, unsigned int writeValue)

Writes the value in the parameter *writeValue* to the selected DIO port. If the port is locked, an ACCESS_ERROR value is returned.

```
unsigned int dev = 2;
unsigned int write_value = 0x55;
int port = 4;

if (DioWritePort(dev, port, write_value))
      printf("Error writing to port %d\n", dev, port);
```

4.4.10 int DioWriteBit(unsigned int device, int bit, unsigned int bitValue)

Writes the value specified by the parameter *bitValue* (0 or 1) to the selected DIO bit. If the bit is part of a locked port, an ACCESS_ERROR value is returned.

```
unsigned int dev = 1;
int bit = 32;
unsigned int bit_value = 1;

if (DioWriteBit(dev, bit, bit_value))
      printf("Error writing bit %d\n", dev, bit);
```

4.4.11 int DioEnableInterrupt(unsigned int device, int bit, int edge)

This function enables interrupts for the selected DIO bit. The edge parameter selects a rising or falling edge trigger for the interrupt. An enumerated value is provided which defines valid values for the parameter *edge* (FALLING_EDGE = 0 and RISING_EDGE = 1).

```
unsigned int dev = 2;
int bit = 0;

if (DioEnableInterrupt(dev, bit, RISING_EDGE))
      printf("Error enabling interrupts for bit %d\n", device, bit);
else
      printf("Interrupts enabled for bit %d\n", dev, bit);
```

4.4.12 int DioDisableInterrupt(unsigned int device, int bit)

This function disables interrupts for the selected DIO bit.

```
unsigned int dev = 2;
int bit = 23;

if (DioDisableInterrupt(dev, bit))
      printf("Error disabling interrupts for bit %d\n", device, bit);
else
      printf("Interrupts disabled for bit %d\n", dev, bit);
```

4.4.13 int DioGetInterrupt(unsigned int device, unsigned int *irqArray)

This function retrieves the interrupt status for all 24 DIO bits and stores them in the memory locations provided by the parameter *irqArray*. Any bit that is set indicates that an interrupt has occurred on that DIO bit. After being read, the interrupt status on all interruptible bits is reset to zero.

```
unsigned int dev = 4;
unsigned int irq[3]; // 24 IRQ = 3 Ports

if (DioGetInterrupt(dev, irq))
      printf("Error retrieving interrupts for device %d\n", dev);
```

4.4.14 int DioWaitForInterrupt(unsigned int device, unsigned int *irqArray, unsigned long timeout)

This function forces the driver to wait for an interrupt on any DIO bit that has been enabled for interrupts. If an interrupt already exists on a bit, the function will act like the GetInterrupt and immediately return and store the interrupt status in the memory locations provided by the parameter *irqArray*.

If no interrupts are present, the function will wait until an interrupt does occur. This function will not stop the execution of driver functions in another thread.

The *timeout* parameter provides a safeguard against system lockup. The value entered provides a time out in milliseconds. If no interrupt occurs before the time-out expires, the pending operation is canceled. To disable the timeout feature, use INFINITE as the timeout parameter.

```
unsigned int dev = 1;
unsigned int irq[3];  // 24 IRQ = 3 Ports
long timeout = 0x1000;  // timeout = 4096 ms

if (DioWaitForInterrupt(dev, irq, timeout))
      printf("Error waiting for interrupts from device %d\n", dev);
```

4.4.15 int DioLockPort(unsigned int device, int port)

Locks the selected DIO port which prevents writing to any bits in that port. The register can still be read.

```
unsigned int dev = 1;
int port = 5;

if (DioLockPort(dev, port))
      printf("Error locking port %d\n", dev, port);
else
      printf("Port %d locked for writing\n", dev, port);
```

4.4.16 int DioUnlockPort(unsigned int device, int port)

Unlocks the selected DIO port which enables writing to any bits in that port.

```
unsigned int dev = 1;
int port = 0;

if (DioUnlockPort(dev, port))
      printf("Error unlocking port %d\n", dev, port);
else
      printf("Port %d unlocked\n", dev, port);
```

4.5 ADC Functions

The PCM-MIO uses two Linear Tech LTC-1859 8-channel A/D converters. Each device is independently software configurable to support the listed input modes and ranges. The devices use a full-duplex serial interface which transmits and receives data simultaneously. An 8-bit command is shifted into the ADC interface to configure it for the next conversion. At the same time, the data from the previous conversion is shifted out of device. Consequently, the conversion result is delayed by one conversion from the command word. Consecutive conversions of the same channel are required to obtain the current voltage measurement. Most of the functions defined include this functionality. For example, the function AdcGetChannelValue will return the current value on the specified channel.

4.5.1 int AdcSetChannelMode(unsigned int device, unsigned int channel, int mode, int duplex, int range)

Configures the input range of the selected ADC channel. The mode variable configures each pair of channels as two single ended inputs or a single differential input. The valid selections are ADC_SINGLE_ENDED and ADC_DIFFERENTIAL. The duplex variable configures the channel for a unipolar or bipolar conversion. The valid selections are ADC_UNIPOLAR and ADC_BIPOLAR. The range variable determines the input span for the conversion. The valid selections are ADC_TOP_5V and ADC_TOP_10V.

```
 // configure channel 4 for single ended ±10V
unsigned int dev = 1;
unsigned int ch = 4;

 dllReturn = AdcSetChannelMode(dev, ch, ADC_SINGLE_ENDED, ADC_BIPOLAR,
 ADC_TOP_10V);

 if (dllReturn)
 {
     printf("Error configuring ADC channel %d\n", ch);
     exit(dllReturn);
 }
 else
     printf("ADC channel %d configured\n", ch);
```

4.5.2 int AdcGetChannelValue(unsigned int device, unsigned int channel, unsigned short *value)

Measures and returns the 16-bit value for the specified ADC channel and stores it in the memory location provided by the parameter *value*.

```
// read ADC value on channel 3
unsigned int dev = 1;
unsigned int ch = 3;
unsigned short adcValue;

dllReturn = AdcGetChannelValue(dev, ch, &adcValue);

if (dllReturn)
{
    printf("Error reading ADC value\n");
    exit(dllReturn);
}
else
    printf("ADC channel %d value is %04x ... ", ch, adcValue);
```

4.5.3 int AdcGetChannelVoltage(unsigned int device, unsigned int channel, float *voltage)

Measures and returns the voltage for the specified ADC channel and stores it in the memory location provided by the parameter *voltage*.

```
// read ADC voltage on channel 2
unsigned int dev = 1;
unsigned int ch = 2;
unsigned short adcVoltage;

dllReturn = AdcGetChannelVoltage(dev, ch, &adcVoltage);

if (dllReturn)
{
    printf("Error reading ADC voltage\n");
    exit(dllReturn);
}
else
    printf("ADC channel %d voltage is %.4f ... ", ch, adcVoltage);
```

4.5.4 int AdcGetAllChannelValues(unsigned int device, unsigned short *valBuf)

Measures and returns the value for all sixteen ADC channels and stores them in the memory array provided by the parameter *valBuf*.

```
// read ADC voltage on all channels
 unsigned int dev = 1;
unsigned short adcValue[16];

dllReturn = AdcGetAllChannelValues(dev, adcValue);

if (dllReturn)
{
    printf("Error reading all ADC values\n");
    exit(dllReturn);
}
else
    for (int ch = 0; ch < 16; ch++)
       printf("ADC channel %d voltage is %04x ... ", ch, adcValue[ch]);
```

4.5.5 int AdcGetAllChannelVoltages(unsigned int device, float *voltBuf)

Measures and returns the voltage for all sixteen ADC channels and stores them in the memory array provided by the parameter *voltBuf*.

```
// read ADC voltage on all channels
 unsigned int dev = 1;
float adcVoltage[16];

dllReturn = AdcGetAllChannelVoltages(dev, adcVoltage);

if (dllReturn)
{
    printf("Error reading all ADC voltages\n");
    exit(dllReturn);
}
else
    for (int ch = 0; ch < 16; ch++)
       printf("ADC channel %d voltage is %.4f ... ", ch, adcVoltage[ch]);
```

4.5.6 int AdcAutoGetChannelVoltage(unsigned int device, unsigned int channel, float *voltage)

This function selects the ADC mode based on the voltage input on the specified channel. This provides the most accurate voltage measurement. Once the mode is set, it measures and returns the voltage in the memory location provided by the parameter *voltage*. This function eliminates the need to set the channel mode before doing a read of a voltage.

```
// auto read ADC voltage on channel 7
unsigned int dev = 1;
unsigned int ch = 7;
float adcVoltage;

dllReturn = AdcAutoGetChannelVoltage(dev, ch, adcVoltage);

if (dllReturn)
{
    printf("Error reading ADC voltage\n");
    exit(dllReturn);
}
else
    printf("ADC channel %d voltage is %.4f ... ", ch, adcVoltage);
```

4.5.7 int AdcConvertSingleChannelRepeated(unsigned int device, unsigned int channel, int count, unsigned short *valBuf)

This function measures the same ADC channel the number of times specified in the variable *count*. Each measurement value is stored in the memory array provided by the parameter *valBuf*. The user must ensure that the size of the array provides enough space for the entire series of measurements.

```
// read channel 15 ADC value 16x
unsigned int dev = 1;
unsigned int ch = 15;
unsigned short buffer[16];
int cnt = sizeof(buffer) / sizeof(unsigned short);
float adcVoltage;

dllReturn = AdcConvertSingleChannelRepeated(dev, ch, cnt, buffer);

if (dllReturn)
{
    printf("Error reading ADC channel 15\n");
    exit(dllReturn);
}
else
    for (int i = 0; i < cnt; i++)
       printf("ADC channel %d voltage is %04x ... ", ch, buffer[i]);
```

4.5.8 int AdcBufferedChannelConversions(unsigned int device, unsigned int *chanBuf, unsigned short *outBuf)

This function measures a series of ADC channels and returns the value for each measurement. An array of channels to be measured is provided in the variable *chanBuf*. This array must be terminated with a 0xFF value. A second array is provided to store the measured values with the variable *outBuf*. To guarantee enough storage, this array must be at least one less than the size of the *chanBuf* array.

```
// provide arrays for function
 unsigned int dev = 1;
unsigned int chanBuf[] = { 1, 5, 7, 3, 8, 10, 11, 12, 8, 14, 0xFF };
unsigned short outBuf[sizeof(chanBuf) - 1];

// buffered channel conversions
dllReturn = AdcBufferedChannelConversions(dev, chanBuf, outBuf);

if (dllReturn)
{
    printf("Error reading series of channels\n");
    exit(dllReturn);
}
else
    for (int i = 0; i < (sizeof(chanBuf) / sizeof(unsigned int)) - 1; i++)
        printf("ADC Channel %d value read is %04x\n", chanBuf[i], outBuf[i]);
```

4.5.9 int AdcConvertToVolts(unsigned int device, unsigned int channel, int value, float *voltage)

This function converts a 16-bit measured value to a voltage. This function can be used with other functions that return a 16-bit value. The mode of the specified channel determines the conversion factors used. For the previous example, we can convert all the returned values using the following code.

```
// provide arrays for function
 unsigned int dev = 1;
unsigned int chanBuf[] = { 1, 5, 7, 3, 10, 11, 12, 14, 0xFF };
unsigned short outBuf[sizeof(chanBuf) - 1];
float temp;

// buffered channel conversions
dllReturn = AdcBufferedChannelConversions(dev, chanBuf, outBuf);

if (dllReturn)
{
    printf("Error reading series of channels\n");
    exit(dllReturn);
}
else
    for (int i = 0; i < (sizeof(chanBuf) / sizeof(unsigned int)) - 1; i++)
    {
        AdcConvertToVolts(dev, chanBuf[i], buffer[i], &temp);
        printf("ADC Channel %d voltage read is %.5f\n", chanBuf[i], temp);
     }
```

4.5.10 int AdcEnableInterrupt(unsigned int device, unsigned int channel)

This function is used to enable interrupts for a specific ADC device. An ADC generates an interrupt when a channel conversion is complete. This eliminates the need for the code to poll the ready bit to indicate when the conversion is complete.

If interrupts are enabled for a specific channel, then interrupts are enabled for all channels on that device. ADC device 1 supports channels 0 to 7 and ADC device 2 supports channels 8 to 15.

```
// selecting channel 12 will enable interrupts for channels 8-15
unsigned int dev = 1;
unsigned int ch = 12;

dllReturn = AdcEnableInterrupt(dev, ch);

if (dllReturn)
{
    printf("Error enabling interrupts for ADC2\n");
    exit(dllReturn);
}
else
    printf("Interrupts enabled for ADC2\n");
```

4.5.11 int AdcDisableInterrupt(unsigned int device, unsigned int channel)

This function is used to disable interrupts for a specific ADC device. If interrupts are disabled for a specific channel, then interrupts are disabled for all channels on that device. ADC device 1 supports channels 0 to 3 and ADS device 2 supports channels 4 to 7.

```
// selecting channel 2 will disable interrupts for channels 0-7
unsigned int dev = 1;
unsigned int ch = 2;

dllReturn = AdcDisableInterrupt(dev, ch);

if (dllReturn)
{
    printf("Error disabling interrupts for ADC1\n");
    exit(dllReturn);
}
else
    printf("Interrupts disabled for ADC1\n");
```

4.5.12 int AdcWaitForConversion(unsigned int device, unsigned int channel, unsigned short *value, unsigned long timeout)

This function forces a process to wait for a conversion to complete on a specific ADC channel that has been enabled for interrupts. The function will wait until an interrupt does occur and then complete the conversion by returning the measured value on that channel. This function will not stop the execution of driver functions in another thread.

The *timeout* parameter provides a safeguard against system lockup. The value entered provides a time out in milliseconds. If no interrupt occurs before the time-out expires, the pending operation is canceled. To disable the timeout feature, use INFINITE as the timeout parameter.

```
// wait for interrupt on channel 1
 unsigned int dev = 1;
unsigned int ch = 1;
unsigned short chValue;
unsigned long *timeout = 0x1000; // 4096 msec

// put process to sleep until conversion completes
dllReturn = AdcWaitForConversion(dev, ch, &chValue, timeout);

if (dllReturn) {
    printf("Error waiting for conversion on channel %d\n", ch);
    exit(dllReturn);
}
else
    printf("ADC Channel %d value is %04x\n", channel, chValue);
```

4.5.13 int AdcWriteCommand(unsigned int device, unsigned int channel, unsigned int adcCommand)

This function allows the user to program the ADC Command Register. This register provides the Input Data Word for one of the ADC devices. The variable *adcCommand* defines the signal type and input range for a specific ADC channel. This 8-bit field is defined as follows. More information for each bit field can be found in the LTC1859 Data Sheet.

				INPUT RANGE			
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
SGL / DIFF	ODD SIGN	SELECT 1	SELECT 0	UNI	GAIN	NAP	SLEEP
MUX CHANNEL SELECTION						**NOT SUPPORTED**	

```
// channel 8 is channel 0 on ADC2
 unsigned int dev = 1;
unsigned int ch = 8;
unsigned int cmd = 0x84; // single ended, positive, ch 0, -10V to +10V

dllReturn = AdcWriteCommand(dev, ch, cmd);

if (dllReturn) {
    printf("Error writing command to channel %d\n", ch);
    exit(dllReturn);
}
else
    printf("Wrote command %02x to ADC Channel %d\n", cmd, ch);
```

4.5.14 int AdcWaitForReady(unsigned int device, unsigned int channel)

This function allows the user to read the ADC Status Register and determine if the Data Ready bit has been set. If the function returns a zero, then the ready bit has been set and the data register contains valid data. If the function returns a value of TIMEOUT_ERROR (5), the ready bit was never set.

```
// wait for ready on channel 4
unsigned int dev = 1;
unsigned int ch = 4;

dllReturn = AdcWaitForReady(dev, ch);

if (!dllReturn)
    printf("Ready returned for channel %d\n", ch);
else if (dllReturn == TIMEOUT_ERROR)
    printf("Ready not returned for channel %d\n", ch);
else {
    printf("Error waiting for ready on channel %d\n", ch);
    exit(dllReturn);
}
```

4.5.15 int AdcReadData(unsigned int device, unsigned int channel, unsigned short *adcData)

This function allows the user to read the ADC Data Registers. The 16-bit value is returned in the variable *adcData*. This should only be used if the AdcWaitForReady function completes successfully.

```
unsigned int dev = 1;
unsigned int ch = 12;
unsigned short data;

dllReturn = AdcReadData(dev, ch, &data);

if (dllReturn)
{
    printf("Error reading data from channel %d\n", ch);
    exit(dllReturn);
}
else
    printf("ADC channel %d value is %04x ... ", ch, data);
```

4.5.16 int AdcStartConversion(unsigned int device, unsigned int channel)

This function combines the AdcWriteCommand and AdcWaitForReady functions into a single call. It performs a complete conversion cycle except for reading the output data. This function can be used as a dummy conversion to align the data output to the current time.

```
unsigned int dev = 1;
unsigned int ch = 15;

dllReturn = AdcStartConversion(dev, ch);

if (dllReturn)
{
    printf("Error performing a conversion from channel %d\n", ch);
    exit(dllReturn);
}
else
```

```
        printf("ADC Conversion complete on channel %d\n", ch);
```

4.6 DAC Functions

The PCM-MIO contains two Linear Tech LTC-2704 Digital-to-Analog Converter (DAC) devices. Each device is a 4-channel converter with software selectable output span.

4.6.1 int DacSetChannelVoltage(unsigned int device, unsigned int channel, float voltage)

This function programs the specified channel to the desired voltage level. The optimal DAC output range is selected according to the desired voltage. The following example code will step the voltage on channel 6 from -10V to +10V by one volt in two-second intervals.

```
 unsigned int dev = 1;
unsigned int ch = 6;
float voltage;

// step through voltages from -10V to 10V
for (voltage = -10.0; voltage <= 10.0; voltage++) {
    dllReturn = DacSetChannelVoltage(dev, ch, voltage);

    if (dllReturn)
    {
        printf("Error setting DAC channel %d voltage\n", ch);
        exit(dllReturn);
    }
    else
    {
        printf("DAC Channel %d voltage set to %.5f\n", ch, voltage);
        Sleep(2000); // sleep for two seconds
    }
}
```

4.6.2 int DacSetChannelOutput(unsigned int device, unsigned int channel, unsigned short code)

This function programs the specified channel to the desired 16-bit value. The actual voltage will depend on the current span setting for that channel. No span adjustments are made unlike the DacSetChannelVoltage function.

```
 unsigned int dev = 1;
unsigned int ch = 1;
unsigned short code = 0x4000;

dllReturn = DacSetChannelValue(dev, ch, code);

if (dllReturn)
{
    printf("Error setting DAC channel %d value\n", ch);
    exit(dllReturn);
}
else
    printf("DAC Channel %d value set to %04x\n", ch, code);
```

4.6.3 int DacSetChannelSpan(unsigned int device, unsigned int channel, unsigned short span)

This function programs the span value for the specified channel. An enumerated value called *DacSpan* provides all valid span values.

DacSpan	Voltage Range
DAC_SPAN_UNI5	Unipolar 0V to 5V
DAC_SPAN_UNI10	Unipolar 0V to 10V
DAC_SPAN_BI5	Bipolar -5V to 5V
DAC_SPAN_BI10	Bipolar -10V to 10V
DAC_SPAN_BI2	Bipolar -2.5V to 2.5V
DAC_SPAN_BI7	Bipolar -2.5V to 7.5V

```
// set channel 0 span to bipolar -5V to 5V
unsigned int dev = 1;
unsigned int ch = 0;

dllReturn = DacSetChannelSpan(dev, ch, DAC_SPAN_BI5);

if (dllReturn)
{
    printf("Error setting DAC span on channel %d\n", ch);
    exit(dllReturn);
}
```

4.6.4 int DacBufferedVoltage(unsigned int device, unsigned short *chanBuf, float *voltBuf)

This function programs a series of DAC channels to a corresponding series of voltages. An array of channels to be measured is provided in the variable *chanBuf*. This array must be terminated with a 0xFF value. A second array provides the voltage to be programmed on the corresponding channel in the variable *voltBuf*. This array must be one less than the size of the *chanBuf* array.

```
unsigned int dev = 1;
unsigned short chanBuf[] = { 0, 1, 2, 3, 4, 5, 6, 7, 0xff };
float voltBuf[] = { -8.8, -5.5, -3.3, -1.8, 1.1, 2.2, 4.4, 9.9 };

dllReturn = DacBufferedVoltage(dev, chanBuf, voltBuf);

if (dllReturn)
{
    printf("Error programming DAC channels\n");
    exit(dllReturn);
}
```

4.6.5 int DacEnableInterrupt(unsigned int device, unsigned int channel)

This function is used to enable interrupts for a specific DAC device. A DAC generates an interrupt when a channel program cycle is complete. This eliminates the need for the code to poll the ready bit to indicate when the program process is complete.

If interrupts are enabled for a specific channel, then interrupts are enabled for all channels on that device. DAC device 1 supports channels 0 to 3 and DAC device 2 supports channels 4 to 7.

```
// selecting channel 7 will enable interrupts for channels 4-7
unsigned int dev = 1;
unsigned int ch = 7;

dllReturn = DacEnableInterrupt(dev, ch);

if (dllReturn)
{
    printf("Error enabling interrupts for DAC2\n");
    exit(dllReturn);
}
else
    printf("Interrupts enabled for DAC2\n");
```

4.6.6 int DacDisableInterrupt(unsigned int device, unsigned int channel)

This function is used to disable interrupts for a specific DAC device. If interrupts are disabled for a specific channel, then interrupts are disabled for all channels on that device. DAC device 1 supports channels 0 to 3 and DAC device 2 supports channels 4 to 7.

```
// selecting channel 1 will diable interrupts for channels 0-3
unsigned int dev = 1;
unsigned int ch = 1;

dllReturn = DacDisableInterrupt(dev, ch);

if (dllReturn)
{
    printf("Error disabling interrupts for DAC1\n");
    exit(dllReturn);
}
else
    printf("Interrupts disabled for DAC1\n");
```

4.6.7 int DacWaitForUpdate(unsigned int device, unsigned int channel, unsigned short code, unsigned long timeout)

This function forces a process to wait for a programming cycle to complete on a specific DAC channel that has been enabled for interrupts. The function will wait until an interrupt does occur indicating that a voltage has been set on that channel. This function will not stop the execution of driver functions in another thread.

The timeout parameter provides a safeguard against system lockup. The value entered provides a time out in milliseconds. If no interrupt occurs before the time-out expires, the pending operation is canceled. To disable the timeout feature, use INFINITE as the timeout parameter.

```
// wait for interrupt on channel 7
```

```
unsigned int dev = 1;
unsigned int ch = 7;
unsigned short chValue = 0xAB00;
unsigned long *timeout = 0x1000; // 4096 msec

// put process to sleep until program cycle completes
dllReturn = DacWaitForConversion(dev, ch, chValue, timeout);

if (dllReturn) {
    printf("Error waiting for program of channel %d\n", ch);
    exit(dllReturn);
}
else
    printf("DAC Channel %d value set to %04x\n", ch, chValue);
```

4.6.8 int DacWriteCommand(unsigned int device, unsigned int channel, unsigned int dacCommand)

This function allows the user to program the DAC Command Register. Each DAC contains a command register used to configure the span and load the data. The command word consists of a 4-bit command and a 4-bit address, as shown. The variable *dacCommand* contains the value to be written. More information for each bit field can be found in the LTC2704 Data Sheet. An enumerated value called *DacControl* provides all valid command values.

7	6	5	4	3	2	1	0
C3	C2	C1	C0	A3	A2	A1	A0

```
// write command to channel 3
unsigned int dev = 1;
unsigned int ch = 3;

dllReturn = DacWriteCommand(dev, ch, DAC_CMD_WR_UPDATE_SPAN);

if (dllReturn) {
    printf("Error writing command to channel %d\n", ch);
    exit(dllReturn);
}
```

4.6.9 int DacWriteData(unsigned int device, unsigned int channel, unsigned int dacData)

This function allows the user to program the 16-bit DAC Data Register. The data register sets the value of the DAC voltage based on the span of the same channel. The variable dacData contains the value to be written.

```
// write data to channel 5
unsigned int dev = 1;
unsigned int ch = 5;
unsigned short chValue = 0x7788;

dllReturn = DacWriteData(dev, ch, chValue);

if (dllReturn) {
    printf("Error writing data to channel %d\n", ch);
    exit(dllReturn);
}
```

4.6.10 int DacReadData(unsigned int device, unsigned int channel, unsigned int *dacData)

This function allows the user to read the DAC Data Registers. The 16-bit value is returned in the variable *dacData*. The Readback Enable bit is set before the read. This should only be used if the DacWaitForReady function completes successfully.

```
    unsigned int dev = 1;
    unsigned int ch = 2;
    unsigned short dacValue;

    dllReturn = DacReadData(dev, ch, & dacValue);

    if (dllReturn)
    {
        printf("Error reading data from channel %d\n", ch);
        exit(dllReturn);
    }
    else
        printf("DAC channel %d value is %04x ... ", ch, dacValue);
```

4.6.11 int DacWaitForReady(unsigned int device, unsigned int channel)

This function allows the user to read the DAC Status Register and determine if the Data Ready bit has been set. If the function returns a zero, then the ready bit has been set and the data register contains valid data. If the function returns a value of TIMEOUT_ERROR (5), the ready bit was never set.

```
    // wait for ready on channel 4
    unsigned int dev = 1;
    unsigned int ch = 4;

    dllReturn = DacWaitForReady(dev, ch);

    if (!dllReturn)
        printf("Ready returned for channel %d\n", ch);
    else if (dllReturn == TIMEOUT_ERROR)
        printf("Ready not returned for channel %d\n", ch);
    else {
        printf("Error waiting for ready on channel %d\n", ch);
        exit(dllReturn);
    }
```

4.7 Generic Functions

4.7.1 int CloseSession(unsigned int device)

This function is used to disable the pcmmio device and close the driver when complete. If a zero is returned, the driver is closed. Otherwise there was an error and the driver is still open.

```
unsigned int dev = 1;

if (CloseSession(dev))
      printf(“Error closing driver\n”);
```

4.8 Every function returns a zero or a positive integer value indicating success or failure. If a zero is returned, the function has completed successfully. If a failure occurs, the specific value returned provides more clarity as to the failure mechanism.

4.8.1 DRIVER_ERROR (1)

This error indicates that some function within the driver has failed. This error indicates that one of the IOCTL calls within the driver itself has not completed successfully. Using Windows Device Manager, verify that the driver is loaded and has no resource conflicts.

4.8.2 ACCESS_ERROR (2)

This error indicates the following conditions are present:

- The driver has tried to write to a DIO bit defined as an input
- The driver attempts to enable interrupts for a DIO bit defined as an output
- The driver attempts to write to a locked DIO port
- The driver attempts to change a bit in a locked DIO port
- The driver is unable to start a DioWaitForInterrupt, AdcWaitForConversion, or DacWaitForUpdate call

4.8.3 INVALID_HANDLE (3)

This error indicates that the driver has not initialized or closed. The driver attempts to obtain a handle to the PCM-MIO device, and this error indicates that the handle was not obtained. Verify that the driver has loaded successfully.

4.8.4 INVALID_PARAMETER (4)

This error indicates that one of the parameters in a DLL function is out of bounds.

4.8.5 TIMEOUT_ERROR (5)

This error indicates the following conditions are present:

- Either the DioWaitForInterrupt, AdcWaitForConversion, or DacWaitForUpdate function has exceeded the provided timeout value before an interrupt occurred
- An ADC conversion failed due to the ready bit never being set
- A DAC program cycle failed due to the ready bit never being set
- The function AdcWaitForReady or DacWaitForReady never had the ready bit set

5 Sample Applications

The driver package provides sample Windows console applications for most of the functions provided in the driver package. The source code and an executable file is provided for each application.

5.1 DIO Applications

5.1.1 flash

The *flash* sample application sets and clears each bit in sequential order. All bits are configured as outputs, and then each bit is toggled with a 200 msec delay. If any error occurs during execution, the error is reported and the application is terminated.

5.1.2 poll

The *poll* sample application tests DIO interrupts. The device is opened and all bits of the first three ports are configured as inputs. Interrupts are enabled for each input bit alternating between rising and falling edge. A separate thread is created that calls the DioWaitForInterrupt function. If any of the input bits are toggled with the proper edge polarity, the specific interrupt is displayed on the console. Any keystroke will terminate the program and report the total number of interrupts.

5.2 ADC Applications

5.2.1 getvolt

The *getvolt* sample application configures the selected channel for ±10V bi-polar and reads and displays the voltage on that channel using the AdcGetChannelVoltage function. The channel number is provided as a command line argument. If any error occurs during execution, the error is reported and the application is terminated.

5.2.2 getvolt_irq

The *getvolt_irq* sample application uses interrupts to read the voltage on each channel sequentially. All channels are configured for ±10V bi-polar and both ADC devices are enabled for interrupts. Two separate threads are created for each ADC device that calls the AdcWaitForConversion function for each channel sequentially every 50 msec. If successful, the voltage read is displayed. Any keystroke will terminate the program and report the total number of interrupts.

5.2.3 getall

The *getall* sample application configures all channels for ±10V bi-polar and uses the AdcGetAllChannelVoltages function to read and display the voltage on all channels. If any error occurs during execution, the error is reported and the application is terminated.

5.2.4 adcBuff

The *adcBuff* sample application configures all channels for ±10V bi-polar and uses the AdcBufferedChannelConversions function to read and display the voltage for the channel sequence provide by the channel array variable. The AdcConvertToVolts function is also demonstrated. If any error occurs during execution, the error is reported and the application is terminated.

5.2.5 adcRepeat

The *adcRepeat* sample application configures the selected channel ±10V bi-polar and uses the AdcConvertSingleChannelRepeated function to read and display the voltage on that channel for the number of times specified in the function arguments. The channel number is provided as a command line argument. The AdcConvertToVolts function is also demonstrated. If any error occurs during execution, the error is reported and the application is terminated.

5.3 DAC Applications

5.3.1 setvolt

The *setvolt* sample application programs the selected channel from -10V to 10V in one volt steps every 2 seconds using the DacSetChannelVoltage function. The channel number is provided as a command line argument. If any error occurs during execution, the error is reported and the application is terminated.

5.3.2 setvolt_irq

The *setvolt_irq* sample application uses interrupts to continuously program a value on a single channel for each device. All channels are configured for a span of ±10V bi-polar and both DAC devices are enabled for interrupts. Two separate threads are created for each DAC device that calls the DacWaitForUpdate function for the specified channel every 50 msec. The value is incremented by a set value after each programming step. If successful, the programmed value is displayed. Any keystroke will terminate the program and report the total number of interrupts.

5.3.3 dacBuff

The *dacBuff* sample application uses the DacBufferedVoltage function to program the channel sequence with the voltage sequence provided by the array variables. The function will optimize the span for each channel based on the desired voltage. If any error occurs during execution, the error is reported and the application is terminated.

Appendix A: pcmmioDLL.h

```
//****************************************************************************
//
//     Copyright 2018 by WinSystems Inc.
//
//****************************************************************************
//
//     Name    : pcmmioDLL.h
//
//     Project : PCM-MIO Windows DLL
//
//     Author  : Paul DeMetrotion
//
//****************************************************************************
//
//       Date        Rev                   Description
//     --------    -------     ---------------------------------------------
//     03/09/18      1.0         Original Release of DLL
//
//****************************************************************************

#ifndef _PCMMIO_DLL_H_
  #define _PCMMIO_DLL_H_

#if defined DLL_EXPORT
  #define DECLDIR __declspec(dllexport)
#else
  #define DECLDIR __declspec(dllimport)
#endif  // DLL_EXPORT

extern "C"
{
    DECLDIR int InitializeSession(unsigned int device);
    DECLDIR int CloseSession(unsigned int device);

    // DIO functions
    DECLDIR int DioResetDevice(unsigned int device);
    DECLDIR int DioSetIoMask(unsigned int device, unsigned int *portState);
    DECLDIR int DioGetIoMask(unsigned int device, unsigned int *portState);
    DECLDIR int DioReadAllPorts(unsigned int device, unsigned int *readValueArray);
    DECLDIR int DioReadPort(unsigned int device, int port, unsigned int *readValue);
    DECLDIR int DioReadBit(unsigned int device, int bit, unsigned int *bitValue);
    DECLDIR int DioSetBit(unsigned int device, int bit);
    DECLDIR int DioClearBit(unsigned int device, int bit);
    DECLDIR int DioWritePort(unsigned int device, int port, unsigned int writeValue);
    DECLDIR int DioWriteBit(unsigned int device, int bit, unsigned int bitValue);
    DECLDIR int DioEnableInterrupt(unsigned int device, int bit, int edge);
    DECLDIR int DioDisableInterrupt(unsigned int device, int bit);
    DECLDIR int DioGetInterrupt(unsigned int device, unsigned int *irqArray);
    DECLDIR int DioWaitForInterrupt(unsigned int device, unsigned int *irqArray, unsigned
long timeout);
    DECLDIR int DioLockPort(unsigned int device, int port);
    DECLDIR int DioUnlockPort(unsigned int device, int port);

    // ADC functions
    DECLDIR int AdcSetChannelMode(unsigned int device, unsigned int channel, int mode,
int duplex, int range);
    DECLDIR int AdcGetChannelValue(unsigned int device, unsigned int channel, unsigned
short *value);
    DECLDIR int AdcGetChannelVoltage(unsigned int device, unsigned int channel, float
*voltage);
    DECLDIR int AdcGetAllChannelValues(unsigned int device, unsigned short *valBuf);
    DECLDIR int AdcGetAllChannelVoltages(unsigned int device, float *voltBuf);
    DECLDIR int AdcAutoGetChannelVoltage(unsigned int device, unsigned int channel, float
*voltage);
    DECLDIR int AdcConvertSingleChannelRepeated(unsigned int device, unsigned int
channel, int count, unsigned short *valBuf);
    DECLDIR int AdcBufferedChannelConversions(unsigned int device, unsigned int *chanBuf,
unsigned short *outBuf);
    DECLDIR int AdcConvertToVolts(unsigned int device, unsigned int channel, int value,
float *voltage);
    DECLDIR int AdcEnableInterrupt(unsigned int device, unsigned int channel);
    DECLDIR int AdcDisableInterrupt(unsigned int device, unsigned int channel);
    DECLDIR int AdcWaitForConversion(unsigned int device, unsigned int channel, unsigned
short *value, unsigned long timeout);
    DECLDIR int AdcWriteCommand(unsigned int device, unsigned int channel, unsigned int
adcCommand);
    DECLDIR int AdcWaitForReady(unsigned int device, unsigned int channel);
    DECLDIR int AdcReadData(unsigned int device, unsigned int channel, unsigned short
*adcData);
    DECLDIR int AdcStartConversion(unsigned int device, unsigned int channel);

    // DAC functions
    DECLDIR int DacSetChannelVoltage(unsigned int device, unsigned int channel, float
voltage);
    DECLDIR int DacSetChannelOutput(unsigned int device, unsigned int channel, unsigned
short code);
    DECLDIR int DacSetChannelSpan(unsigned int device, unsigned int channel, unsigned
short span);
    DECLDIR int DacBufferedVoltage(unsigned int device, unsigned short *chanBuf, float
*voltBuf);
    DECLDIR int DacEnableInterrupt(unsigned int device, unsigned int channel);
    DECLDIR int DacDisableInterrupt(unsigned int device, unsigned int channel);
    DECLDIR int DacWaitForUpdate(unsigned int device, unsigned int channel, unsigned
short code, unsigned long timeout);
    DECLDIR int DacWriteCommand(unsigned int device, unsigned int channel, unsigned int
dacCommand);
    DECLDIR int DacWriteData(unsigned int device, unsigned int channel, unsigned int
dacData);
    DECLDIR int DacReadData(unsigned int device, unsigned int channel, unsigned int
*dacData);
    DECLDIR int DacWaitForReady(unsigned int device, unsigned int channel);
}

typedef enum {
    SUCCESS = 0,
    DRIVER_ERROR,
    ACCESS_ERROR,
    INVALID_HANDLE,
    INVALID_PARAMETER,
    TIMEOUT_ERROR
} ErrorCodes;
```

```
typedef enum {
    FALLING_EDGE = 0,
    RISING_EDGE
} IrqEdge;

typedef enum {
    ADC_DIFFERENTIAL = 0,
    ADC_SINGLE_ENDED
} AdcMode;

typedef enum {
    ADC_CH0_SELECT = 0,
    ADC_CH2_SELECT,
    ADC_CH4_SELECT,
    ADC_CH6_SELECT,
    ADC_CH1_SELECT,
    ADC_CH3_SELECT,
    ADC_CH5_SELECT,
    ADC_CH7_SELECT
} AdcChSelect;

typedef enum {
    ADC_BIPOLAR = 0,
    ADC_UNIPOLAR
} AdcDuplex;

typedef enum {
    ADC_TOP_5V = 0,
    ADC_TOP_10V
} AdcRange;

typedef enum {
    DAC_A = 0,
    DAC_B = 2,
    DAC_C = 4,
    DAC_D = 6,
    DAC_ALL = 15,
} DacAddress;

typedef enum {
    DAC_SPAN_UNI5 = 0,
    DAC_SPAN_UNI10,
    DAC_SPAN_BI5,
    DAC_SPAN_BI10,
    DAC_SPAN_BI2,
    DAC_SPAN_BI7
} DacSpan;

typedef enum {
    DAC_CMD_WR_B1_SPAN = 2,
    DAC_CMD_WR_B1_CODE,
    DAC_CMD_UPDATE,
    DAC_CMD_UPDATE_ALL,
    DAC_CMD_WR_UPDATE_SPAN,
    DAC_CMD_WR_UPDATE_CODE,
    DAC_CMD_WR_SPAN_UPDATE_ALL,
    DAC_CMD_WR_CODE_UPDATE_ALL,
    DAC_CMD_RD_B1_SPAN,
    DAC_CMD_RD_B1_CODE,
    DAC_CMD_RD_B2_SPAN,
    DAC_CMD_RD_B2_CODE,
    DAC_CMD_SLEEP,
    DAC_CMD_NOP
} DacControl;

#endif  // _PCMMIO_DLL_H_
```