UIO48 Digital I/O
Windows Device Driver Package

1 Introduction

1.1 Many of WinSystems' full-featured, high-performance single board computers (SBC) offer integrated Digital I/O. All digital I/O solutions utilize the WS16C48 Universal I/O Controller ASIC, which provides the various input/output and interrupt configurations.

1.2 The WS16C48 ASIC supports up to 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 UIO48 driver package is designed for and has been verified with 32-bit and 64-bit versions of Microsoft Windows 7, 8, and 10.

1.4 The UIO48 driver is for use with the following WinSystems, Inc. products. The number of I/O and interrupt lines provided varies for each product. These values will affect the size of port and bit parameters in the functions detailed below.

Product	I/O Lines	Interrupt Lines
EPX-C380	48	24
EBC-C384	48	24
PXM-C388	24	24
PPM-C407	24	24
PPM-C412	24	24
EBC-C413	48	24
EPX-C414	48	24
PCM-C418	24	24
PPM-LX800-G	16	16

2 Installation

2.1 Before loading the Windows operating system, verify that the Digital I/O settings are correct using the BIOS setup utility.

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

a. uio48.sys – Windows device driver
b. uio48.inf – Windows installation file
c. uio48.cat – Windows catalog file
d. WdfCoinstaller01011.dll – Windows co-installer
e. uio48DLL.dll – Windows DLL
f. uio48DLL.lib – Windows library file
g. uio48DLL.h – driver include file
h. GPIO-Poll-App.cpp – Windows application source
i. GPIO-Test-App.cpp – Windows application source
j. GPIO-Poll-App.exe – Windows application
k. GPIO-Test-App.exe – Windows application
l. 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 *uio48.inf*. The Windows installer will copy the *uio48.sys* driver file to the appropriate directory in the Windows installation.

2.4 In Device Manager, the UIO48 Device will appear under the System Devices item. The desired hardware configuration can be selected under the Resources tab of the UIO48 Device Properties window. A reboot may be required after resource selection is complete.

2.5 The included console applications, *GPIO-Test-App.exe* and *GPIO-Poll-App.exe,* can be used to verify driver installation and functionality. Usage of these programs is described later in this document. The WinSystems loopback cable (part number TCBL-DIO24-002-02A) is required to run these applications.

3 Driver Overview and Architecture

3.1 The file *uio48.sys* is a Windows Driver Foundation kernel-mode (KMDF) driver which facilitates access to the underlying hardware.

3.2 The file *uio48DLL.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 the WS16C48 ASIC. Data is passed to and from the driver utilizing input and output buffers.

4 Driver Usage

4.1 The *uio48DLL.h* file included in the driver distribution contains the function definitions to be used by an application to communicate with the uio48 driver. This file is included in Appendix A: uio48DLL.h.

4.1 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 uio48 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.

```
if (InitializeSession())
       printf("Error opening device.\n");
```

4.3 Once the driver is initialized, the other functions can be used to control the UIO48. Following is a description and sample code for each function. For all functions, if a zero is returned, the function was successful. Otherwise there was an error and the function did not complete.

4.3.1 int ResetDevice(void)

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

```
if (ResetDevice())
       printf("Error resetting device.\n");
```

4.3.2 int SetIoMask(unsigned int *portState)

Configures the mask for all ports provided in the memory location provided by the input 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 mask[3]; // 24 I/O = 3 ports

mask[0] = 0xFF; // all bits output
mask[1] = 0x00; // all bits input
mask[2] = 0xF0; // upper nibble output, lower nibble input

if (SetIoMask(mask))
       printf("Error configuring port masks.\n");
```

4.3.3 int GetIoMask(unsigned int *portState)

Retrieves the mask for all 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 mask[2]; // 16 I/O = 2 ports

if (GetIoMask(mask))
       printf("Error reading port masks.\n");
```

4.3.4 int ReadAllPorts(unsigned int *readValueArray)

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

```
unsigned int read_value[6]; // 48 I/O = 6 ports
```

```
if (ReadAllPorts(read_value))
       printf("Error reading all ports.\n");
```

4.3.5 int ReadPort(int port, unsigned int *readValue)

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

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

if (ReadPort(port, &read_value))
       printf("Error reading port %d.\n", port);
else
       printf("Port %d = 0x%0x\n", port, read_value);
```

4.3.6 int ReadBit(int bit, unsigned int *bitValue)

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

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

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

4.3.7 int SetBit(int bit)

Sets the selected bit.

```
int bit = 24;

if (SetBit(bit))
       printf("Error setting bit %d.\n", bit);
```

4.3.8 int ClearBit(int bit)

Clears the selected bit.

```
int bit = 0;

if (ClearBit(bit))
       printf("Error clearing bit %d.\n", bit);
```

4.3.9 int WritePort(int port, unsigned int writeValue)

Writes the value in the parameter *writeValue* to the selected port.

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

if (WritePort(port, write_value))
```

```
printf("Error writing to port %d.\n", port);
```

4.3.10 int WriteBit(int bit, unsigned int bitValue)

Writes the value specified by the parameter *bitValue* (0 or 1) to the selected bit.

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

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

4.3.11 int EnableInterrupt(int bit, int edge)

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

```
int bit = 0;

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

4.3.12 int DisableInterrupt(int bit)

This function disables interrupts for the selected bit.

```
int bit = 23;

if (DisableInterrupt(bit))
	printf("Error enabling interrupts for bit %d.\n", bit);
else
	printf("Interrupts disabled for bit %d.\n", bit);
```

4.3.13 int GetInterrupt(unsigned int *irqArray)

This function retrieves the interrupt status for all 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 bit. After being read, the interrupt status on all interruptible bits is reset to zero.

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

if (GetInterrupt(irq))
	printf("Error retrieving interrupts.\n");
```

4.3.14 int WaitForInterrupt(unsigned int *irqArray, unsigned long timeout)

This function forces the driver to wait for an interrupt on any 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. The pending operation is canceled.

If the session is terminated before an interrupt occurs, the IO request will be automatically terminated.

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

if (WaitForInterrupt(irq, timeout))
       printf("Error waiting for interrupts.\n");
```

4.3.15 int LockPort(int port)

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

```
int port = 5;  // 48 I/O = 6 Ports

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

4.3.16 int UnlockPort(int port)

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

```
int port = 0;  // 24 I/O = 3 Ports

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

4.3.17 int CloseSession(void)

This function is used to disable the uio48 device and close the driver when complete. If a zero is returned, the timer is disabled and the driver is closed. Otherwise there was an error and the driver is still open.

```
if (CloseSession())
       printf("Error closing driver.\n");
```

4.4 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.

4.4.1 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.

4.4.2 ACCESS_ERROR (2)

This error indicates that the driver has tried to write to a bit defined as an input. It is also generated if the driver attempts to enable interrupts for a bit defined as an output. Verify that the port mask has been set as desired.

4.4.3 INVALID_HANDLE (3)

This error indicates that the driver has not initialized or closed. The driver attempts to obtain a handle to the UIO48 device, and this error indicates that the handle was not obtained. Verify that the driver has loaded successfully.

4.4.4 INVALID_PARAMETER (4)

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

4.4.5 TIMEOUT_ERROR (5)

This error indicates that the WaitForInterrupt function has exceeded the provided timeout value before an interrupt occurred.

5 Sample Applications

There are two sample Windows console applications provided in the driver package, *GPIO-Test-App* and *GPIO-Poll-App*. The source code for the applications are provided in Appendix B: GPIO-Test-App.cpp and Appendix C: GPIO-Poll-App.cpp.

5.1 GPIO-Test-App

The *GPIO-Test-App* sample application is a simple program that illustrates how to set and clear single output points and write values to ports. The program utilizes the loopback cable to walk a one in bits 9 through 20 which are configured as outputs. Bits 1-8 and bits 21-24 are configured as outputs and should reflect the bit pattern on the output bits.

5.2 GPIO-Poll-App

The *GPIO-Poll-App* sample application uses two separate threads where one thread is configured to wait for interrupts. Using the same bit configuration as the *GPIO-Test-App* application, bits 1-8 are configured for rising edge interrupts and bits 21-24 are configured for falling edge interrupts. The output bits are toggled and the appropriate interrupts are verified.

Appendix A: uio48DLL.h

```
//*****************************************************************************
//
//     Copyright 2017 by WinSystems Inc.
//
//*****************************************************************************
//
//     Name    : uio48DLL.h
//
//     Project : UIO48 Windows DLL
//
//     Author  : Paul DeMetrotion
//
//*****************************************************************************
//
//       Date        Rev                   Description
//     --------    -------     -----------------------------------------
//     05/04/17      1.0           Original Release of DLL
//
//*****************************************************************************

#ifndef _UIO48_DLL_H_
#define _UIO48_DLL_H_

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

extern "C"
{
    DECLDIR int InitializeSession();
    DECLDIR int ResetDevice();
    DECLDIR int SetIoMask(unsigned int *portState);
    DECLDIR int GetIoMask(unsigned int *portState);
    DECLDIR int ReadAllPorts(unsigned int *readValueArray);
    DECLDIR int ReadPort(int port, unsigned int *readValue);
    DECLDIR int ReadBit(int bit, unsigned int *bitValue);
    DECLDIR int SetBit(int bit);
    DECLDIR int ClearBit(int bit);
    DECLDIR int WritePort(int port, unsigned int writeValue);
    DECLDIR int WriteBit(int bit, unsigned int bitValue);
    DECLDIR int EnableInterrupt(int bit, int edge);
    DECLDIR int DisableInterrupt(int bit);
    DECLDIR int GetInterrupt(unsigned int *irqArray);
    DECLDIR int WaitForInterrupt(unsigned int *irqArray, unsigned int timeout);
    DECLDIR int LockPort(int port);
    DECLDIR int UnlockPort(int port);
    DECLDIR int CloseSession();
}

typedef enum {
    SUCCESS = 0,
    DRIVER_ERROR,
    ACCESS_ERROR,
```

```
    INVALID_HANDLE,
    INVALID_PARAMETER,
    TIMEOUT_ERROR
} ErrorCodes;

#endif
```

Appendix B: GPIO-Test-App.cpp

```
// GPIO-Test-App.cpp : Defines the entry point for the console application.
// 0 input 1 output
//
// UIO MAPPING WITH LOOPBACK
//
// INPUTS   0    1    2    3    4    5    6    7    20    21   22   23
//          |    |    |    |    |    |    |    |    |     |    |    |
//          |    |    |    |    |    |    |    |    |     |    |    |
//          |    |    |    |    |    |    |    |    |     |    |    |
// OUTPUTS  8    9    10   11   12   13   14   15   16    17   18   19

#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include "..\uio48DLL.h"
#include <conio.h>
#include <Windows.h>
#include <iostream>
#include <bitset>
#include <process.h>


#define PORTCNT 3
#define PORTSZ 8
#define UIOCNT 24

//loopback plug connects nibbles of third port, and then connects the other two to each
other.

BOOL quitFlag = FALSE;

//Reads and prints inputs (Inputs should be port 0, and the second half of port 2)
void readInputsThread(void *)
{
	unsigned int reading = 0;
	unsigned int readnib = 0;
	unsigned int readbit = 0;
	while (!quitFlag)
	{
		readnib = 0;
		if (ReadPort(0,&reading))
			std::cout << "Failed to read\n";

		for (int j = 20; j < 24; ++j) //reads second nibble of port 2
		{
			if (ReadBit(j, &readbit))
				std::cout << "Failed to read\n";
			if (readbit)
				readnib = readnib | (readbit << j-20);
		}

		std::bitset<8> binreadbyte(reading);
		std::bitset<4> binreadnib(readnib);
		std::cout << "UIO 8-1 + 24-21: " << binreadbyte << " + "<<binreadnib<<"\n";
		Sleep(1000); //1 second sleep
```

```
	}
	_endthread();
}

int _tmain(int argc, _TCHAR* argv[])
{
	int errCode = 0;

	//Setting up
	unsigned int testread = 1;
	unsigned int portTest[3];
	unsigned int maskTest[3];

	portTest[0] = 0x00;		//setting up vars for iomask (Port 0 Input, Port
1 Output, Port 2[1-4] output, Port2[5-8] input)
	portTest[1] = 0xFF;
	portTest[2] = 0x0F;
	try{
		if (errCode = InitializeSession()) //get handle to driver
		{
			std::cout << "Failed to initialize session.\n";
			throw errCode;
		}
		else
			std::cout << "Driver Initialized.\n";

		if (errCode = ResetDevice()) //make sure in clean state after
initialization
		{
			std::cout << "Failed to reset ports.\n";
			throw errCode;
		}

		if (errCode = SetIoMask(portTest))
		{
			std::cout << "Failed to set mask.\n";
			throw errCode;
		}

		if (errCode = GetIoMask(maskTest))
		{
			std::cout << "Failed to get mask.\n";
			throw errCode;
		}

		for (int i = 0; i < 3; ++i) //ensure get and set are functioning as
expected
		{
			if (portTest[i] != maskTest[i])
			{
				printf("Mask does not match what it was set to. Exiting . .
.\n");
				return 1;
			}
		}

		_beginthread(readInputsThread, 0, NULL); //kicks off thread that reads and
prints inputs
```

```
		while (1)
		{
			for (int i = 8; i < 20; ++i) //each loop does a set bit and then
clears it after one second
			{

				if (errCode = SetBit(i))
				{
					std::cout << "Failed to set bit : " << i << "\n";
					throw errCode;
				}
				Sleep(1000); //sleep one second
				if (errCode = ClearBit(i))
				{
					std::cout << "Failed to clear bit: " << i << "\n";
					throw errCode;
				}
			}
			std::cout << "\n**********\nBit walk complete. Press q to quit, or
any other key to restart walk\n**********\n";
			_kbhit();

			if (_getch() == 'q')
			{
				quitFlag = TRUE;
				if (errCode = ResetDevice())
					{
						std::cout << "Failed to reset device.\n";
						throw errCode;
					}

					if (errCode = CloseSession())
					{
						std::cout << "Failed to close Driver.\n";
						throw errCode;
					}
					std::cout << "UIO closed. Exiting\n";
					break;
			}
		}
	}
	catch (int err)
	{
		switch (err)
		{
		case DRIVER_ERROR:
		{
			std::cout << "Failed to communicate with driver\n";
			break;
		}
		case ACCESS_ERROR:
		{
			std::cout << "Access Error\n";
			break;
		}
```

```
		case INVALID_HANDLE:
		{
			std::cout << "Failed to obtain handle to driver\n";
			break;
		}
		case INVALID_PARAMETER:
		{
			std::cout << "Bad parameter\n";
			break;
		}
		default:
			std::cout << "Unkown error!\n";
		}
		system("pause");
		return err;
	}

	return 0;
}
```

Appendix C: GPIO-Poll-App.cpp

```
// GPIO-Poll-App.cpp : Defines the entry point for the console application.
//
// 0 input 1 output
//
// UIO MAPPING WITH LOOPBACK
//
// INPUTS   0    1    2    3    4    5    6    7    20    21   22   23
//          |    |    |    |    |    |    |    |    |     |    |    |
//          |    |    |    |    |    |    |    |    |     |    |    |
//          |    |    |    |    |    |    |    |    |     |    |    |
// OUTPUTS  8    9    10   11   12   13   14   15   16    17   18   19

#include <tchar.h>
#include <stdio.h>
#include <stdlib.h>
#include "..\uio48DLL.h"
#include <conio.h>
#include <Windows.h>
#include <iostream>
#include <bitset>
#include <process.h>


//loopback plug connects nibbles of third port, and then connects the other two to each
other.

#define FALLING_EDGE 0
#define RISING_EDGE 1
#define PORTCNT 3
#define PORTSZ 8
#define UIOCNT 24

//Waits for interrupts, and prints which line they are detected on. Inputs should be port
0, and the second half of port 2.
void readInputsThread(void* threadData)
{
	BOOL* pquitFlag = (BOOL*)threadData;
	int errcode = 0;
	unsigned int timeout = 50000; //5 seconds
	unsigned int irqArr[PORTCNT];

	if(GetInterrupt(irqArr))
		std::cout << "Failed to get interrupts\n";

	std::cout << "Should see interrupts occur on all of port 0 (bits[0-7]).\n";

	for (int i = 0; i < UIOCNT; ++i)
	{
		if (((irqArr[i / PORTSZ]) & (1 << (i % PORTSZ))) > 0)
			std::cout << "Interrupt occured on bit : " << i << "\n";
	}

	while (!(*pquitFlag))
	{
		errcode = 0;
```

```
		if (errcode = WaitForInterrupt(irqArr, timeout))
		{
			std::cout << "Failed to wait for interrupt with error code : " <<
errcode << "\n";
			continue;
		}

		for (int i = 0; i < UIOCNT; ++i)
		{
			if (((irqArr[i / PORTSZ]) & (1 << (i % PORTSZ))) > 0)
				std::cout << "Interrupt occured on bit :"<< i <<"\n";
		}

		Sleep(100);
	}
	_endthread();
}

int _tmain(int argc, _TCHAR* argv[])
{
	int errCode = 0;
	BOOL quitFlag = FALSE;
	//Setting up
	unsigned int testread = 1;
	unsigned int irqArr[PORTCNT];
	unsigned int portTest[PORTCNT];
	unsigned int maskTest[PORTCNT];
	unsigned int testTimeout = 5000;
	portTest[0] = 0x00;                                  //setting up iomask (Port 0 Input,
Port 1 Output, Port 2[1-4] output, Port 2[5-8] input)
	portTest[1] = 0xFF;
	portTest[2] = 0x0F;

	try
    {
			quitFlag = FALSE;
			if (errCode = InitializeSession())
			{
				std::cout << "Failed to initialize session.\n";
				throw errCode;
			}
			else
				std::cout << "Driver Initialized.\n";

			if (errCode = ResetDevice()) //make sure in clean state after
initialization
			{
				std::cout << "Failed to reset ports.\n";
				throw errCode;
			}

			if (errCode = SetIoMask(portTest))
			{
				std::cout << "Failed to set mask.\n";
				throw errCode;
			}

			if (errCode = GetIoMask(maskTest))
```

```
		{
			std::cout << "Failed to get mask.\n";
			throw errCode;
		}

		for (int i = 0; i < PORTCNT; ++i) //ensure get and set are
functioning as expected
		{
			if (portTest[i] != maskTest[i])
			{
				printf("Mask does not match what it was set to. Exiting
. . .\n");
				return 1;
			}
		}

		//set first port to have rising edge interrupts
		for (int i = 0; i < PORTSZ; ++i){
			if (errCode = EnableInterrupt(i, RISING_EDGE))
			{
				std::cout << "Failed to enable interrupt.\n";
				throw errCode;
			}
		}

		//set second half of third port to have falling edge interrupts
		for (int i = 20; i < 24; ++i)
		{
			if (errCode = EnableInterrupt(i, FALLING_EDGE))
			{
				std::cout << "Failed to enable interrupt.\n";
				throw errCode;
			}
		}

		std::cout << "Testing Wait for Interrupt's timeout with " <<
testTimeout << "  milliseconds\n";

		if (WaitForInterrupt(irqArr, testTimeout) == TIMEOUT_ERROR)
		{
			std::cout << "WaitForInterrupt successfully threw a timeout
error\n";
		}
		else{
			std::cout << "Timeout failed to work correctly\n";
		}

		if (errCode = WritePort(1, 0xFF)) //make interrupts occur on all of
port 0 by making rising edge occur on all of port 1 (should see after get interrupt)
		{
			std::cout << "Failed to write port\n";
			throw errCode;
		}

		_beginthread(readInputsThread, 0, (void*)&quitFlag); //kicks off
thread that reads and prints inputs
```

```
                if (errCode = WritePort(1, 0x00)) //Setting outputs back to 0, so
rising edge interrupts trigger
                {
                        std::cout << "Failed to write port\n";
                        throw errCode;
                }

                Sleep(1000);

                while (1)
                {
                        //loops through all outputs
                        for (int i = 8; i < 20; i++) //each loop sets a bit and then
clears it after one second
                        {
                                if (errCode = WriteBit(i, 1)) //set
                                {
                                        std::cout << "Failed to set bit : " << i <<
"\n";
                                        throw errCode;
                                }
                                else
                                        std::cout << "Bit " << i << " set\n";

                                Sleep(1000); //sleep one second

                                if (errCode = WriteBit(i, 0)) //clear
                                {
                                        std::cout << "Failed to clear bit: " << i <<
"\n";
                                        throw errCode;
                                }
                                else
                                        std::cout << "Bit " << i << " cleared\n";

                                Sleep(1000);
                        }

                        std::cout << "\n**********\nInterrupt walk complete. Press q
to quit, or any other key to restart walk\n**********\n";

                        _kbhit();
                        if (_getch() == 'q')
                        {
                                quitFlag = TRUE;
                                if (errCode = ResetDevice())
                                {
                                        std::cout << "Failed to reset device.\n";
                                        throw errCode;
                                }

                                if (errCode = CloseSession())
                                {
                                        std::cout << "Failed to close Driver.\n";
                                        throw errCode;
                                }
                        std::cout << "UIO closed. Exiting\n";
                        break;
```

```
                        }
                }
        }
        catch (int err)
        {
                switch (err)
                {
                case DRIVER_ERROR:
                {
                        std::cout << "Failed to communicate with driver\n";
                        break;
                }
                case ACCESS_ERROR:
                {
                        std::cout << "Access Error\n";
                        break;
                }
                case INVALID_HANDLE:
                {
                        std::cout << "Failed to obtain handle to driver\n";
                        break;
                }
                case INVALID_PARAMETER:
                {
                        std::cout << "Bad parameter\n";
                        break;
                }
                default:
                        std::cout << "Unkown error!\n";
                }
                system("pause");
                return err;
        }
        system("pause");

        return 0;
}
```