

WinSystems

Windows XP Embedded UIO 48/96 Digital I/O
Driver Documentation

Supporting the
PCM-UIO48A and PCM-UIO96A Digital I/O
Devices

Table of Contents

1	Introduction	3
1.1	Hardware Requirements.....	3
1.2	Host System Requirements	3
1.3	Driver Deliverables	3
2	Driver Integration	3
2.1	Driver Installation	3
2.2	Driver Files	3
2.3	Adding the Driver to your Configuration.....	3
3	Using the Driver	3
3.1	IOCTL_WIO_SET_IO_MASK.....	3
3.2	IOCTL_WIO_GET_IO_MASK.....	3
3.3	IOCTL_WIO_CLEAR_IO_MASK.....	3
3.4	IOCTL_WIO_READ_BIT.....	3
3.5	IOCTL_WIO_WRITE_BIT.....	3
3.6	IOCTL_WIO_SET_BIT	3
3.7	IOCTL_WIO_CLR_BIT	3
3.8	IOCTL_WIO_READ_BYTE	3
3.9	IOCTL_WIO_WRITE_BYTE	3
3.10	IOCTL_WIO_ENAB_INT.....	3
3.11	IOCTL_WIO_DISAB_INT.....	3
3.12	IOCTL_WIO_WAIT_ON_INT.....	3
3.13	IOCTL_WIO_GET_NUM_IO_POINTS.....	3
3.14	IOCTL_WIO_INT_PERMITTED.....	3
4	Appendix A: How To Modify The Component Registry Settings.....	3
4.1	Modifying the I/O Base Address and IRQ Values	3
4.2	WinSystems PCM-UIO48A Digital I/O	3
4.3	WinSystems PCM-UIO96A Digital I/O	3
4.4	Modifying the Registry Settings for Multiple Device Configurations	3
5	Appendix B: Ioctl.h.....	3

1 Introduction

The WinSystems PCM-UIO48A and PCM-UIO96A are highly versatile PC/104 input/output modules providing 48 and 96 lines of digital I/O, respectively. The provided driver is a Windows XP Embedded driver that has been tested using the Windows XP Embedded Operating System Service Pack 1.

1.1 Hardware Requirements

- This driver has been designed to work with any x86 processor supported by Windows XP Embedded.

1.2 Host System Requirements

Your development machine should contain the following:

- Microsoft Windows Embedded Studio including Target Designer and Component Database Manager.

1.3 Driver Deliverables

WSUIO48_96XP.zip

This zip file contains the WinSystems PCM-UIO48A/PCM-UIO96A Driver and support files for the Windows XP Embedded Operating System.

WSUIO48_96XPSource.zip

This zip file contains the source files for the WinSystems PCM-UIO48A/PCM-UIO96A Driver for the Windows XP Embedded Operating System.

2 Driver Integration

2.1 Driver Installation

Step 1 Unzip the WSUIO48_96XP.zip file into a folder on the computer containing the Microsoft Windows Embedded Studio development tools.

Step 2 Import the WinSystems PCM-UIO48A Digital I/O and WinSystems PCM-UIO96A Digital I/O components into the Component Database. Run the Component Database Manager program and on the **Database** tab, select **Import....** The Import SLD dialog will then appear. Click the **...** button next to the **SLD file:** edit box and browse to the folder you unzipped the WSUIO48_96XP.zip file in step 1 and select the “WSUIO48_96XP.sld” file and click **Open**. You should not need to modify the default entry for the **Repository root:** field. Make sure that you have the **Copy repository files to repository root** checkbox checked so that the driver files will be copied into the Windows XP Embedded Component Database repository. Click **Import** to begin the import process. When the import process is complete click **Close** on the Import SLD dialog and then click **Close** on the Microsoft Component Database Manager dialog to close the Component Database Manager program.

2.2 Driver Files

- WSUIO48_96XP.sld

This is the WinSystems UIO48/96 Digital I/O Driver Windows XP Embedded component file.

- WSUIO48_96XP.inf

This is the WinSystems UIO48/96 Digital I/O Driver Windows XP installation file. The component file, WSUIO48_96XP.sld, is based on this file.

- WSUIO48_96XP.cat

This is a dummy catalog file. You may submit this driver to the Microsoft Windows Hardware Quality Lab (WHQL) for WHQL certification.

- WSUIO48_96XP.sys

This is the retail build of the WinSystems UIO48/96 Digital I/O Driver.

- Ioctl.h

This file contains structure definitions and IOCTL definitions to be used by an application to communicate with the WinSystems UIO48/96 Digital I/O Driver. This file is included in this document in Appendix B: Ioctl.h.

2.3 Adding the Driver to your Configuration

The following steps will guide you in integrating the WinSystems UIO48/96 Digital I/O Driver to your configuration.

Step 1 Open or create the project for which the UIO 48/96 Digital I/O Driver is to be integrated.

Step 2 Right-click on the WinSystems PCM-UIO48A Digital I/O or WinSystems PCM-UIO96A Digital I/O component, located in the Hardware\Devices\Non-Plug and Play Devices node of the component tree view, depending upon which device is in your hardware configuration.

Step 3 Both of these components depend upon the Class Installer – Non-Plug and Play Drivers component. The Class Installer – Non-Plug and Play Drivers component should automatically be added to the configuration when you perform a dependency check. If you prefer to add this component manually you must change the minimum visibility to a value of 100 before you can locate the component in the Software\System\SystemServices\Base node of the component tree view.

Step 4 The WinSystems PCM-UIO48A Digital I/O and WinSystems PCM-UIO96A Digital I/O components must be configured to the specific settings of the hardware resources. To configure these components the registry settings of these components need to be modified. Verify that you are running Target Designer in “Expert” mode and that the **Resources** menu item in the **View** menu is checked. Select the WinSystems PCM-UIO48A Digital I/O or the WinSystems PCM-UIO96A Digital I/O component in your configuration,

expand its tree node and select **Registry Data**. The registry settings for the driver will then be shown in the Details Pane. Several of these settings will need to be modified to match the configuration of your hardware. Please consult Appendix A: How To Modify The Component Registry Settings for the instructions on modifying the registry settings to correspond to your hardware configuration.

3 Using the Driver

The IOCTL.S.H file included in the driver distribution files contains structure definitions and IOCTL definitions to be used by an application to communicate with the WinSystems UIO48/96 Digital I/O Driver.

An application uses the Windows API function *CreateFile* to open the driver. The following example opens the first instance of the WinSystems UIO48/96 Digital I/O Driver.

```
HANDLE hWIOHandle = CreateFile( _T("\\\\.\\WSUIO48_96XP_0"),
                                GENERIC_READ | GENERIC_WRITE,
                                0,
                                NULL,
                                OPEN_EXISTING,
                                FILE_FLAG_OVERLAPPED,
                                NULL);
```

The following example opens a second instance of the WinSystems UIO48/96 Digital I/O Driver.

```
HANDLE hWIOHandle = CreateFile( _T("\\\\.\\WSUIO48_96XP_1"),
                                GENERIC_READ | GENERIC_WRITE,
                                0,
                                NULL,
                                OPEN_EXISTING,
                                FILE_FLAG_OVERLAPPED,
                                NULL);
```

Note: You must use the FILE_FLAG_OVERLAPPED file attribute flag in the call to *CreateFile*.

Once a handle to the driver has been obtained by calling *CreateFile*, the handle can be used in calls to the Windows API function *DeviceIoControl* to access the driver's features.

```
DWORD dwNumIOPoints;
DWORD dwActualOut;
DeviceIoControl( hWIOHandle,
                 IOCTL_WIO_GET_NUM_IO_POINTS,
                 NULL,
                 0,
                 (PVOID) &dwNumIOPoints,
                 sizeof(DWORD),
                 &dwActualOut,
                 NULL);
```

When finished using the WinSystems UIO48/96 Digital I/O Driver a call should be made to the Windows API function *CloseHandle*.

```
CloseHandle(hWIOHandle);
```

The following is a description of each of the driver IOCTL functions that can be called via the Windows API function *DeviceIoControl*. Information as to the *DeviceIoControl* parameter definitions can be found in the *IOCTLS.H* file. This file is included in this document in Appendix B: *Ioctls.h*.

3.1 IOCTL_WIO_SET_IO_MASK

Sets the I/O mask definition. The driver can operate with or without an I/O mask definition. If an I/O mask definition is defined, the driver will prevent the application from writing to I/O points that are defined as inputs. If no I/O mask definition is defined, the driver treats all I/O points as outputs and allows the application to write to an I/O point even if it is logically an input.

3.2 IOCTL_WIO_GET_IO_MASK

Gets the I/O mask definition. The driver can operate with or without an I/O mask definition. If an I/O mask definition is defined, the driver will prevent the application from writing to I/O points that are defined as inputs. If no I/O mask definition is defined, the driver treats all I/O points as outputs and allows the application to write to an I/O point even if it is logically an input.

3.3 IOCTL_WIO_CLEAR_IO_MASK

Clears the I/O mask definition. The driver can operate with or without an I/O mask definition. If an I/O mask definition is defined, the driver will prevent the application from writing to I/O points that are defined as inputs. If no I/O mask definition is defined, the driver treats all I/O points as outputs and allows the application to write to an I/O point even if it is logically an input.

3.4 IOCTL_WIO_READ_BIT

Reads the state of the specified I/O point.

3.5 IOCTL_WIO_WRITE_BIT

Sets the state of the specified I/O point to the specified value. If an I/O mask definition is set via a call to the function *IOCTL_WIO_SET_IO_MASK* and the I/O point is defined as an input, this function returns with an error and a *GetLastError* of *STATUS_ACCESS_VIOLATION*.

3.6 IOCTL_WIO_SET_BIT

Sets the state of the specified I/O point to a Low Impedance state. If an I/O mask definition is set via a call to the function *IOCTL_WIO_SET_IO_MASK* and the I/O point is defined as an input, this function returns with an error and a *GetLastError* of *STATUS_ACCESS_VIOLATION*.

3.7 IOCTL_WIO_CLR_BIT

Sets the state of the specified I/O point to a High Impedance state. If an I/O mask definition is set via a call to the function *IOCTL_WIO_SET_IO_MASK* and the I/O point is defined as an input, this function returns with an error and a *GetLastError* of *STATUS_ACCESS_VIOLATION*.

3.8 IOCTL_WIO_READ_BYTE

Gets the state of all I/O points in the specified port.

3.9 IOCTL_WIO_WRITE_BYTE

Sets the state of all I/O points in the specified port. If an I/O mask definition is set via a call to the function `IOCTL_WIO_SET_IO_MASK` and any of the I/O points in the specified port are defined as inputs, this function returns with an error and a *GetLastError* of `STATUS_ACCESS_VIOLATION`.

3.10 IOCTL_WIO_ENAB_INT

Enables an interrupt for the specified I/O point at the specified polarity. If the driver is operating in a polled mode, `IOCTL_WIO_ENAB_INT` has no effect and returns with an error and a *GetLastError* of `STATUS_INVALID_DEVICE_REQUEST`.

3.11 IOCTL_WIO_DISAB_INT

Disables the interrupt for the specified I/O point. If the driver is operating in a polled mode, `IOCTL_WIO_DISAB_INT` has no effect and returns with an error and a *GetLastError* of `STATUS_INVALID_DEVICE_REQUEST`.

3.12 IOCTL_WIO_WAIT_ON_INT

Waits for an interrupt to occur on any of the I/O points that have interrupts enabled via previous calls to `IOCTL_WIO_ENAB_INT`. When calling this function, execution will NOT return to the caller until an interrupt on the device occurs. If the driver is operating in a polled mode, `IOCTL_WIO_WAIT_ON_INT` returns immediately with an error and a *GetLastError* of `STATUS_INVALID_DEVICE_REQUEST`.

3.13 IOCTL_WIO_GET_NUM_IO_POINTS

Gets the number of I/O points the driver is controlling.

3.14 IOCTL_WIO_INT_PERMITTED

Call to determine if interrupts are permitted in the driver. The driver may be configured to handle interrupts or operate in a polled only mode. This function allows the application to determine if the driver can support interrupts, or can only operate in a polled mode.

4 Appendix A: How To Modify The Component Registry Settings

4.1 Modifying the I/O Base Address and IRQ Values

The I/O Base Address and IRQ values are embedded in the data of the HKEY_LOCAL_MACHINE\System\CurrentControlSet\Enum\Root\LegacyDriver\0000\LogConf\ForcedConfig entry. These values must be modified to match your hardware configuration.

Select the WinSystems PCM-UIO48A Digital I/O or WinSystems PCM-UIO96A Digital I/O component to modify from the configuration tree view. Expand the node and select **Registry Data**. Select the HKEY_LOCAL_MACHINE\System\CurrentControlSet\Enum\Root\LegacyDriver\0000\LogConf\ForcedConfig entry. Right-click and select **Properties** to display the Registry Resource Properties dialog. The binary value of this registry entry is shown in the **Value:** field of the dialog. To edit a value, position the cursor over the left-most character of the byte to modify and begin to enter the new value. To remove a value, or several values, highlight the value(s) to remove and click the **Delete** button.

4.2 WinSystems PCM-UIO48A Digital I/O

Below is the default configuration for the PCM-UIO48A Digital I/O. This default configuration sets an I/O Base Address of 0x0240 and IRQ 5.

```
01 00 00 00 FF FF FF FF
00 00 00 00 00 00 00 00
02 00 00 00 01 00 05 00
40 02 00 00 00 00 00 001
10 00 00 00 02 01 01 00
05 00 00 002 05 00 00 003
FF FF FF FF
```

The shaded section is the interrupt configuration section. If your PCM-UIO48A device is not configured for an interrupt this section must be removed.

¹ This field contains the I/O Base Address of the device. The size of this field is 64 bits (eight bytes) in length. The PCM-UIO48A device has a valid range of 0x0010 – 0x0FF0. Therefore, you should only modify the first two bytes of this field to match your hardware configuration.

² This field contains the IRQ Level. The size of this field is 32 bits (four bytes) in length. The PCM-UIO48A device supports the following IRQ values: 2, 3, 4, 5, 6, 7, 10, 11, 12, 14 and 15. Therefore, you should only modify the first byte of this field to match your hardware configuration. Please note that the field is represented in hexadecimal, therefore, if assigning an IRQ value of 10, 11, 12, 14 or 15 the value entered would be 0A, 0B, 0C, 0E or 0F respectively.

³ This field contains the IRQ Vector. The size of this field is 32 bits (four bytes) in length. This field should match the value entered for the IRQ Level field.

4.3 WinSystems PCM-UIO96A Digital I/O

Below is the default configuration for the PCM-UIO96A Digital I/O. This default configuration sets an I/O Base Address of 0x0240, IRQ 5 and IRQ7.

```
01 00 00 00 FF FF FF FF
00 00 00 00 00 00 00 00
03 00 00 00 01 00 05 00
40 02 00 00 00 00 00 001
20 00 00 00 02 01 01 00
05 00 00 002 05 00 00 003
FF FF FF FF 02 01 01 00
07 00 00 004 07 00 00 005
FF FF FF FF
```

The shaded sections are the interrupt configuration sections. If your PCM-UIO96A device is not configured for an interrupt these sections must be removed. If your PCM-UIO96A device is configured for a single “shared” interrupt the second interrupt section must be removed.

¹ This field contains the I/O Base Address of the device. The size of this field is 64 bits (eight bytes) in length. The PCM-UIO98A device has a valid range of 0x0020 – 0x01FE0. Therefore, you should only modify the first two bytes of this field to match your hardware configuration.

² This field contains the IRQ Level for the 24 I/O points at the base address. The size of this field is 32 bits (four bytes) in length. The PCM-UIO96A device supports the following IRQ values: 2, 3, 4, 5, 6, 7, 10, 11, 12, 14 and 15. Therefore, you should only modify the first byte of this field to match your hardware configuration. Please note that the field is represented in hexadecimal, therefore, if assigning an IRQ value of 10, 11, 12, 14 or 15 the value entered would be 0A, 0B, 0C, 0E or 0F respectively.

³ This field contains the IRQ Vector for the 24 I/O points at the base address. The size of this field is 32 bits (four bytes) in length. This field should match the value entered for the IRQ Level field for the 24 I/O points at the base address.

⁴ This field contains the IRQ Level for the 24 I/O points at the base address plus 16 bytes. The size of this field is 32 bits (four bytes) in length. The PCM-UIO96A device supports the following IRQ values: 2, 3, 4, 5, 6, 7, 10, 11, 12, 14 and 15. Therefore, you should only modify the first byte of this field to match your hardware configuration. Please note that the field is represented in hexadecimal, therefore, if assigning an IRQ value of 10, 11, 12, 14 or 15 the value entered would be 0A, 0B, 0C, 0E or 0F respectively.

⁵ This field contains the IRQ Vector for the 24 I/O points at the base address plus 16 bytes. The size of this field is 32 bits (four bytes) in length. This field should match the value entered for the IRQ Level field for the 23 I/O points at the base address plus 16 bytes.

4.4 Modifying the Registry Settings for Multiple Device Configurations

If your hardware configuration contains multiple WinSystems PCM-UIO48A/PCM-UIO96A devices additional registry settings must be modified. There must be a component in your XP Embedded configuration for each device in your hardware configuration. Both the WinSystems PCM-UIO48A and the PCM-UIO96A devices use the same driver. Therefore, whatever your hardware configuration consists of, if you have more than one of these devices in your hardware configuration you must follow these instructions to configure the registry settings of the components in your XP Embedded configuration.

Along with each step are the changes that would need to be made for an example where a second instance of the WinSystems PCM-UIO48/96 Digital I/O Driver added to the OS configuration. The changes are shown in this font.

Step 1 For each additional instance of the driver you must modify the registry key
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Class\
{8ECC055D-047F-11D1-A537-0000F8753ED1}\0000 to match the number
of the instance.

Changes made to the second instance of the driver:

```
HKEY_LOCAL_MACHINE\...\{ 8ECC055D-047F-11D1-A537-0000F8753ED1 }\0001\DriverDate
HKEY_LOCAL_MACHINE\...\{ 8ECC055D-047F-11D1-A537-0000F8753ED1 }\0001\DriverDateData
HKEY_LOCAL_MACHINE\...\{ 8ECC055D-047F-11D1-A537-0000F8753ED1 }\0001\DriverDesc
HKEY_LOCAL_MACHINE\...\{ 8ECC055D-047F-11D1-A537-0000F8753ED1 }\0001\DriverVersion
HKEY_LOCAL_MACHINE\...\{ 8ECC055D-047F-11D1-A537-0000F8753ED1 }\0001\InfPath
HKEY_LOCAL_MACHINE\...\{ 8ECC055D-047F-11D1-A537-0000F8753ED1 }\0001\InfSection
HKEY_LOCAL_MACHINE\...\{ 8ECC055D-047F-11D1-A537-0000F8753ED1 }\0001\InfSectionExt
HKEY_LOCAL_MACHINE\...\{ 8ECC055D-047F-11D1-A537-0000F8753ED1 }\0001\MatchingDeviceId
HKEY_LOCAL_MACHINE\...\{ 8ECC055D-047F-11D1-A537-0000F8753ED1 }\0001\ProviderName
```

Step 2 For each additional instance of the driver you must modify the registry key
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Enum\Root\
LEGACYDRIVER\0000 to match the number of the instance.

Changes made to the second instance of the driver:

```
HKEY_LOCAL_MACHINE\... \Enum\Root\LEGACYDRIVER\0001\Capabilities
HKEY_LOCAL_MACHINE\... \Enum\Root\LEGACYDRIVER\0001\Class
HKEY_LOCAL_MACHINE\... \Enum\Root\LEGACYDRIVER\0001\ClassGUID
HKEY_LOCAL_MACHINE\... \Enum\Root\LEGACYDRIVER\0001\ConfigFlags
HKEY_LOCAL_MACHINE\... \Enum\Root\LEGACYDRIVER\0001\Control\ActiveService
HKEY_LOCAL_MACHINE\... \Enum\Root\LEGACYDRIVER\0001\Control\FilteredConfigVector
HKEY_LOCAL_MACHINE\... \Enum\Root\LEGACYDRIVER\0001\DeviceDesc
HKEY_LOCAL_MACHINE\... \Enum\Root\LEGACYDRIVER\0001\DeviceParameters\
HKEY_LOCAL_MACHINE\... \Enum\Root\LEGACYDRIVER\0001\Driver
HKEY_LOCAL_MACHINE\... \Enum\Root\LEGACYDRIVER\0001\FriendlyName
HKEY_LOCAL_MACHINE\... \Enum\Root\LEGACYDRIVER\0001\HardwareID
HKEY_LOCAL_MACHINE\... \Enum\Root\LEGACYDRIVER\0001\LogConf\BasicConfigVector
HKEY_LOCAL_MACHINE\... \Enum\Root\LEGACYDRIVER\0001\LogConf\ForcedConfig
HKEY_LOCAL_MACHINE\... \Enum\Root\LEGACYDRIVER\0001\Mfg
HKEY_LOCAL_MACHINE\... \Enum\Root\LEGACYDRIVER\0001\Service
```

Step 3 For the first instance and for each additional instance you must modify the “Count” and “NextInstance” fields of the HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\WSUIO48_96XP\Enum registry key. The “Count” field contains the number of instances of the driver. The “NextInstance” field contains the zero-based number of the next instance. The OS updates these fields when the user installs another instance of the driver. Target Designer does not modify these fields when multiple instances of the driver are added to the configuration. Therefore, the designer of the OS must modify these values to correspond to the actual hardware and driver configuration.

Changes made to the first instance of the driver:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\WSUIO48_96XP\Enum\Count

Value = 2

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\WSUIO48_96XP\Enum\NextInstance

Value = 2

Changes made to the second instance of the driver:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\WSUIO48_96XP\Enum\Count

Value = 2

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\WSUIO48_96XP\Enum\NextInstance

Value = 2

Step 4 For each additional instance you must modify the instance pointer to point to the correct driver configuration instance. This is done by modifying the “0” field of the HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\WSUIO48_96XP\Enum registry key to reflect the instance and to point to the correct driver instance under the HKEY_LOCAL_MACHINE\System\CurrentControlSet\Enum\Root\LEGACYDRIVER key. Since Target Designer does not allow the addition of registry entries to a component, the “0” field will be set in the registry by the first instance and each additional instance will set its instance and instance pointer.

Changes made to the second instance of the driver:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\WSUIO48_96XP\Enum\1

Value = ROOT\LEGACYDRIVER\0001

5 Appendix B: Ioctl.h

```
// IOCTLS.H -- IOCTL code definitions for WSUIO48_96XP driver
// Copyright (c) winSystems. All rights reserved.

#ifndef IOCTLS_H
#define IOCTLS_H

#ifndef CTL_CODE
#pragma message("CTL_CODE undefined. Include winioctl.h or wdm.h")
#endif

// Definition of the maximum number of UIO chips that can be
// contained in the device.
#define MAX_DEVICES 2

// Maximum number of IO ports.
#define MAX_IO_PORTS 6

// Maximum number of IO points per device.
#define MAX_IO_POINTS 48

// Maximum number of bits in a byte.
#define MAX_BITS_BYTE 8

// Maximum number of interrupt ports.
#define MAX_INT_PORTS 3

// Maximum number of interrupt IO points per device.
#define MAX_INT_POINTS 24

typedef unsigned char BYTE;
typedef unsigned char* PBYTE;
typedef unsigned long DWORD;
typedef unsigned long* PDWORD;

// Structure to use when writing a value to an
// I/O point or port using IOCTL_WIO_WRITE_BIT
// and IOCTL_WIO_WRITE_BYTE.
typedef struct _UIOWriteValue
{
    union
    {
        // Bit number to access.
        // Valid Range:
        //     UIO-48 configuration 1-48
        //     UIO-96 configuration 1-96
        DWORD dwBitNumber;

        // Port number to access.
        // Valid Range:
        //     UIO-48 configuration 1-6
        //     UIO-96 configuration 1-12
        DWORD dwPortNumber;
    };

    // Value to write, 1 or 0.
    BYTE byValue;
} UIOWRITE_VALUE, *PUIOWRITE_VALUE;

// Structure to use when configuring interrupts
// using IOCTL_WIO_ENAB_INT.
typedef struct _UIOConfigInterrupt
{
    union
    {
        // Bit number to access.
        // Valid Range:
        //     UIO-48 configuration 1-48
        //     UIO-96 configuration 1-96
        DWORD dwBitNumber;
```

```

        // Port number to access.
        // Valid Range:
        //     UIO-48 configuration    1-6
        //     UIO-96 configuration    1-12
        DWORD dwPortNumber;
};

// Polarity of the interrupt.
// 1 = Enable rising-edge interrupt.
// 0 = Enable falling-edge interrupt.
BYTE byPolarity;

} UIO_CONFIG_INTERRUPT, *PUIO_CONFIG_INTERRUPT;

// Define the type of device.
// This parameter can be no bigger than a WORD value. The values used by
// Microsoft are in the range of 0-32767 and the values between
// 32768-65535 are reserved for use by OEMs and IHVs.
#define WIO_DEV_TYPE 32768

// *****
//
// IOCTL_WIO_SET_IO_MASK
//
// Device IO control routine to set the I/O mask definition.
//
// dwCode    IOCTL_WIO_SET_IO_MASK
// cbin      [IN] Size of input data buffer.
//           Length of the I/O mask definition.
//           There must be 6 bytes for a UIO-48 config.
//           There must be 12 bytes for a UIO-96 config.
// cbout     Ignored
// pCopyBuffer [IN] Pointer to an unsigned character array
//                holding the I/O mask definition.
//                A 0 represents the I/O point as an input.
//                A 1 represents the I/O point as an output.
//                There must be 6 bytes for a UIO-48 config.
//                There must be 12 bytes for a UIO-96 config.
//
//                Byte# 0 = bits 7 - 0
//                      1 = bits 15 - 8
//                      2 = bits 23 - 16
//                      3 = bits 31 - 24
//                      4 = bits 39 - 32
//                      5 = bits 47 - 40
//
//                Sample mask definition:
//                Bit# 76543210
//                Mask 10101010
//
//                | | | | | | | | Input
//                | | | | | | | | Output
//                | | | | | | | | Input
//                | | | | | | | | Output
//                | | | | | | | | Input
//                | | | | | | | | Output
//                | | | | | | | | Input
//                | | | | | | | | Output
//
// info      [OUT] Number of output bytes put into the copy buffer.
//
#define IOCTL_WIO_SET_IO_MASK\
CTL_CODE(WIO_DEV_TYPE, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)

// *****
//
// IOCTL_WIO_GET_IO_MASK
//
// Device IO control routine to get the I/O mask definition.
//
// dwCode    IOCTL_WIO_GET_IO_MASK
// cbin      Ignored

```

```

// cbout      [IN] Size of output data buffer.
//             Length of the I/O mask definition.
//             There must be 6 bytes for a UIO-48 config.
//             There must be 12 bytes for a UIO-96 config.
// pCopyBuffer [OUT] Pointer to an unsigned character array
//             to receive the I/O mask definition.
//             A 0 represents the I/O point as an input.
//             A 1 represents the I/O point as an output.
//             There must be 6 bytes for a UIO-48 config.
//             There must be 12 bytes for a UIO-96 config.
//
//             Byte# 0 = bits 7 - 0
//                   1 = bits 15 - 8
//                   2 = bits 23 - 16
//                   3 = bits 31 - 24
//                   4 = bits 39 - 32
//                   5 = bits 47 - 40
//
//             Sample mask definition:
//             Bit# 76543210
//             Mask 10101010
//             | | | | | | | |
//             | | | | | | | | Input
//             | | | | | | | | Output
//             | | | | | | | | Input
//             | | | | | | | | Output
//             | | | | | | | | Input
//             | | | | | | | | Output
//             | | | | | | | | Input
//             | | | | | | | | Output
//
// info        [OUT] Number of output bytes put into the copy buffer.
//
#define IOCTL_WIO_GET_IO_MASK\
    CTL_CODE(WIO_DEV_TYPE, 0x801, METHOD_BUFFERED, FILE_ANY_ACCESS)
// *****
//
// IOCTL_WIO_CLEAR_IO_MASK
//
// Device IO control routine to clear the I/O mask definition.
//
// code        IOCTL_WIO_CLEAR_IO_MASK
// cbin        Ignored
// cbout       Ignored
// pCopyBuffer Ignored
// info        [OUT] Number of output bytes put into the copy buffer.
//
#define IOCTL_WIO_CLEAR_IO_MASK\
    CTL_CODE(WIO_DEV_TYPE, 0x802, METHOD_BUFFERED, FILE_ANY_ACCESS)
// *****
//
// IOCTL_WIO_READ_BIT
//
// Device IO control routine to get the state of the I/O pin
// specified.
//
// code        IOCTL_WIO_READ_BIT
// cbin        [IN] Size of input data buffer, should be
//             sizeof(DWORD).
// cbout       [IN] Size of output data buffer, should be
//             sizeof(BYTE).
// pCopyBuffer [IN/OUT]
//             [IN] Pointer a DWORD containing the bit number
//             to read.
//             valid Range:
//                 UIO-48 configuration 1-48
//                 UIO-96 configuration 1-96
//             [OUT] Pointer to a BYTE containing the state
//             of the I/O pin.
//             A 0 indicates pin is in a High Impedance state.
//             A 1 indicates pin is in a Low Impedance state.

```

```

// info      [OUT] Number of output bytes put into the copy buffer.
//
#define IOCTL_WIO_READ_BIT\
    CTL_CODE(WIO_DEV_TYPE, 0x803, METHOD_BUFFERED, FILE_ANY_ACCESS)

// *****
//
// IOCTL_WIO_WRITE_BIT
//
// Device IO control routine to set the state of the I/O pin
// specified.
//
// code      IOCTL_WIO_WRITE_BIT
// cbin      [IN] Size of input data buffer, should be
//           sizeof(UIO_WRITE_VALUE).
// cbout     Ignored
// pCopyBuffer [IN] Pointer a UIO_WRITE_VALUE structure.
//
//           UIO_WRITE_VALUE structure
//           dwBitNumber The bit number to write value to.
//           Valid Range:
//           UIO-48 configuration 1-48
//           UIO-96 configuration 1-96
//           byvalue      value to write to bit.
//           A 0 causes the pin to go to a High Impedance state.
//           A 1 causes the pin to go to a Low Impedance state.
//
// info      [OUT] Number of output bytes put into the copy buffer.
//
#define IOCTL_WIO_WRITE_BIT\
    CTL_CODE(WIO_DEV_TYPE, 0x804, METHOD_BUFFERED, FILE_ANY_ACCESS)

// *****
//
// IOCTL_WIO_SET_BIT
//
// Device IO control routine to set the state of the I/O pin
// specified to a Low Impedance state.
//
// code      IOCTL_WIO_SET_BIT
// cbin      [IN] Size of input data buffer, should be
//           sizeof(DWORD).
// cbout     Ignored
// pCopyBuffer [IN] Pointer a DWORD containing the bit number
//           to set.
//           Valid Range:
//           UIO-48 configuration 1-48
//           UIO-96 configuration 1-96
//
// info      [OUT] Number of output bytes put into the copy buffer.
//
#define IOCTL_WIO_SET_BIT\
    CTL_CODE(WIO_DEV_TYPE, 0x805, METHOD_BUFFERED, FILE_ANY_ACCESS)

// *****
//
// IOCTL_WIO_CLR_BIT
//
// Device IO control routine to set the state of the I/O pin
// specified to a High Impedance state.
//
// code      IOCTL_WIO_CLR_BIT
// cbin      [IN] Size of input data buffer, should be
//           sizeof(DWORD).
// cbout     Ignored
// pCopyBuffer [IN] Pointer a DWORD containing the bit number
//           to clear.
//           Valid Range:
//           UIO-48 configuration 1-48
//           UIO-96 configuration 1-96
//
// info      [OUT] Number of output bytes put into the copy buffer.
//
#define IOCTL_WIO_CLR_BIT\

```

```

CTL_CODE(WIO_DEV_TYPE, 0x806, METHOD_BUFFERED, FILE_ANY_ACCESS)

// *****
//
// IOCTL_WIO_READ_BYTE
//
// Device IO control routine to get the state of all I/O pins
// in the specified port.
//
// code      IOCTL_WIO_READ_BYTE
// cbin      [IN] Size of input data buffer, should be
//           sizeof(DWORD).
// cbout     [IN] Size of output data buffer, should be
//           sizeof(BYTE).
// pCopyBuffer [IN/OUT] Pointer a DWORD containing the port number
//           to read.
//           valid Range:
//               UIO-48 configuration    1-6
//               UIO-96 configuration    1-12
// [OUT] Pointer to a BYTE containing the state
// of the I/O port.
// info      [OUT] Number of output bytes put into the copy buffer.
//
#define IOCTL_WIO_READ_BYTE\
CTL_CODE(WIO_DEV_TYPE, 0x807, METHOD_BUFFERED, FILE_ANY_ACCESS)

// *****
//
// IOCTL_WIO_WRITE_BYTE
//
// Device IO control routine to set the state of all I/O pins
// in the specified port.
//
// code      IOCTL_WIO_WRITE_BYTE
// cbin      [IN] Size of input data buffer, should be
//           sizeof(UIO_WRITE_VALUE).
// cbout     Ignored
// pCopyBuffer [IN] Pointer a UIO_WRITE_VALUE structure.
//           UIO_WRITE_VALUE structure
//           dwPortNumber The port number to write value to.
//           valid Range:
//               UIO-48 configuration    1-6
//               UIO-96 configuration    1-12
//           byValue      value to write to port.
//
// info      [OUT] Number of output bytes put into the copy buffer.
//
#define IOCTL_WIO_WRITE_BYTE\
CTL_CODE(WIO_DEV_TYPE, 0x808, METHOD_BUFFERED, FILE_ANY_ACCESS)

// *****
//
// IOCTL_WIO_ENAB_INT
//
// Device IO control routine to enable an interrupt for the
// specified I/O pin at a specified polarity.
//
// code      IOCTL_WIO_ENAB_INT
// cbin      [IN] Size of input data buffer, should be
//           sizeof(UIO_CONFIG_INTERRUPT).
// cbout     Ignored
// pCopyBuffer [IN] Pointer a UIO_CONFIG_INTERRUPT structure.
//           UIO_CONFIG_INTERRUPT structure
//           dwBitNumber The bit number to enable interrupts on.
//           valid Range:
//               UIO-48 configuration    1-24
//               UIO-96 configuration    1-48
//           byPolarity The polarity of the interrupt.

```



```

//          Non-zero enables rising-edge interrupt.
//          Zero enables falling-edge interrupt.
//
// info      [OUT] Number of output bytes put into the copy buffer.
//
#define IOCTL_WIO_ENAB_INT\
CTL_CODE(WIO_DEV_TYPE, 0x809, METHOD_BUFFERED, FILE_ANY_ACCESS)

// *****
//
// IOCTL_WIO_DISAB_INT
//
// Device IO control routine to disable an interrupt for the
// specified I/O pin.
//
// code      IOCTL_WIO_DISAB_INT
// cbin      [IN] Size of input data buffer, should be
//           sizeof(DWORD).
// cbout     Ignored
// pCopyBuffer [IN] Pointer a DWORD containing the bit number
//           to disable interrupts on.
//           valid Range:
//               UIO-48 configuration    1-24
//               UIO-96 configuration    1-48
// info      [OUT] Number of output bytes put into the copy buffer.
//
#define IOCTL_WIO_DISAB_INT\
CTL_CODE(WIO_DEV_TYPE, 0x80A, METHOD_BUFFERED, FILE_ANY_ACCESS)

// *****
//
// IOCTL_WIO_WAIT_ON_INT
//
// Device IO control routine to wait for an interrupt for
// any bits that have been specified on previous calls to
// IOCTL_WIO_ENAB_INT.
//
// code      IOCTL_WIO_WAIT_ON_INT
// cbin      Ignored.
// cbout     [IN] Size of output data buffer.
//           There must be 3 bytes for a UIO-48 config.
//           There must be 6 bytes for a UIO-96 config.
// pCopyBuffer [OUT] Pointer to an unsigned character array
//           to receive the I/O bits that have triggered
//           an interrupt.
//           There must be 3 bytes for a UIO-48 config.
//           There must be 6 bytes for a UIO-96 config.
//
// In a UIO-48 device there are 48 I/O points.
// The UIO-48 has the ability to monitor 24 I/O
// points for both rising and falling digital
// edge transitions and issue an interrupt. The
// first 24 I/O points are the points that can
// be configured to issue an interrupt upon the
// rising or falling digital edge transition.
//
// In a UIO-96 device there are 96 I/O points.
// The UIO-96 has the ability to monitor 48 I/O
// points for both rising and falling digital
// edge transitions and issue an interrupt. The
// UIO-96 device contains two winSystems'
// WS16C48 ASICs. Therefore, the 48 I/O points
// that may be configured for interrupts
// span across the two chips. The 48 I/O points
// that may be configured for interrupts
// correspond to I/O bits 1 - 24 and bits 49 - 72.
//
// Below is a chart that details the format of
// the buffer pointed to by pBufOut.
//
//
// Byte#  0 = Int. bits 8 - 1 I/O bits 8 - 1
//         1 = Int. bits 16 - 9 I/O bits 16 - 9

```

```

//          2 = Int. bits 24 - 17 I/O bits 24 - 17
//          3 = Int. bits 32 - 25 I/O bits 56 - 49
//          4 = Int. bits 40 - 33 I/O bits 64 - 57
//          5 = Int. bits 48 - 41 I/O bits 72 - 65
//
//      Sample return value:
//      Bit# 76543210
//      value 10101010
//      | | | | | | | |
//      | | | | | | | | No Interrupt
//      | | | | | | | | Interrupt
//      | | | | | | | | No Interrupt
//      | | | | | | | | Interrupt
//      | | | | | | | | No Interrupt
//      | | | | | | | | Interrupt
//      | | | | | | | | No Input
//      | | | | | | | | Interrupt
//
// info      [OUT] Number of output bytes put into the copy buffer.
//
#define IOCTL_WIO_WAIT_ON_INT\
CTL_CODE(FILE_DEVICE_UNKNOWN, 0x80B, METHOD_BUFFERED, FILE_ANY_ACCESS)
// *****
//
// IOCTL_WIO_GET_NUM_IO_POINTS
//
// Device IO control routine to get the number of I/O points
// in the configuration.
//
// code      IOCTL_WIO_GET_NUM_IO_POINTS
// cbin      Ignored
// cbout     [IN] Size of output data buffer, should be
//           sizeof(DWORD).
// pCopyBuffer [OUT] Pointer a DWORD to receive the number
//           of I/O points in the configuration.
// info      [OUT] Number of output bytes put into the copy buffer.
//
#define IOCTL_WIO_GET_NUM_IO_POINTS\
CTL_CODE(WIO_DEV_TYPE, 0x80C, METHOD_BUFFERED, FILE_ANY_ACCESS)
// *****
//
// IOCTL_WIO_INT_PERMITTED
//
// Device IO control routine to determine if interrupts are
// permitted in the driver.
//
// code      IOCTL_WIO_INT_PERMITTED
// cbin      Ignored
// cbout     [IN] Size of output data buffer, should be
//           sizeof(DWORD).
// pCopyBuffer [OUT] Pointer a DWORD to receive a value
//           of 1 if interrupts are permitted, 0 if
//           interrupts are not permitted.
// info      [OUT] Number of output bytes put into the copy buffer.
//
#define IOCTL_WIO_INT_PERMITTED\
CTL_CODE(WIO_DEV_TYPE, 0x80D, METHOD_BUFFERED, FILE_ANY_ACCESS)
//
#endif

```