

# Unmanaged SimpleIO DLL Documentation

---

## Contents

Unmanaged vs. Managed DLL and Their Requirements.....	3
How to Use Visual Studio to Find Which DLL Your Application Needs .....	3
Sample Code for an Unmanaged Application (C++).....	5
Simple IO API.....	7
Summary: .....	7
1. InitMCP2200 .....	8
2. IsConnected .....	8
3. ConfigureMCP2200 .....	9
4. fnRxLED .....	10
5. fnTxLED .....	11
6. fnHardwareFlowControl .....	12
7. fnULoad.....	13
8. fnSuspend .....	14
9. fnSetBaudRate .....	14
10. ConfigureIO .....	15
11. ConfigureIoDefaultOutput .....	16
12. SetPin .....	17
13. ClearPin .....	17
14. ReadPin .....	18

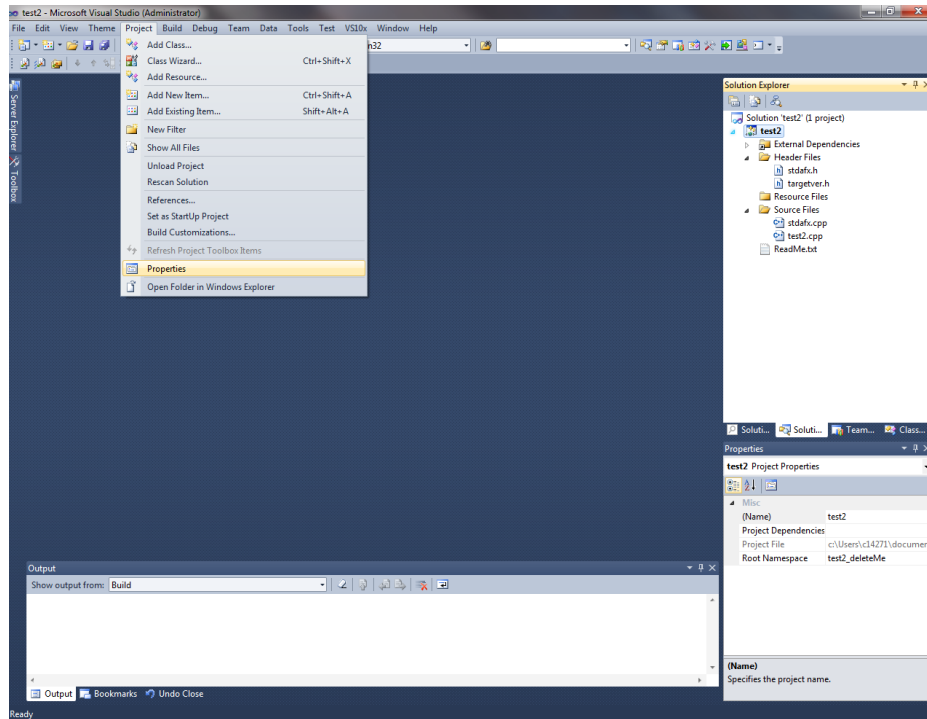
15.	ReadPinValue .....	19
16.	WritePort .....	20
17.	ReadPort .....	20
18.	ReadPortValue .....	21
19.	SelectDevice .....	22
20.	GetSelectedDevice .....	23
21.	GetNoOfDevices.....	24
22.	GetDeviceInfo .....	24
23.	GetSelectedDeviceInfo.....	25
24.	ReadEEPROM .....	26
25.	WriteEEPROM .....	27

## Unmanaged vs. Managed DLL and Their Requirements

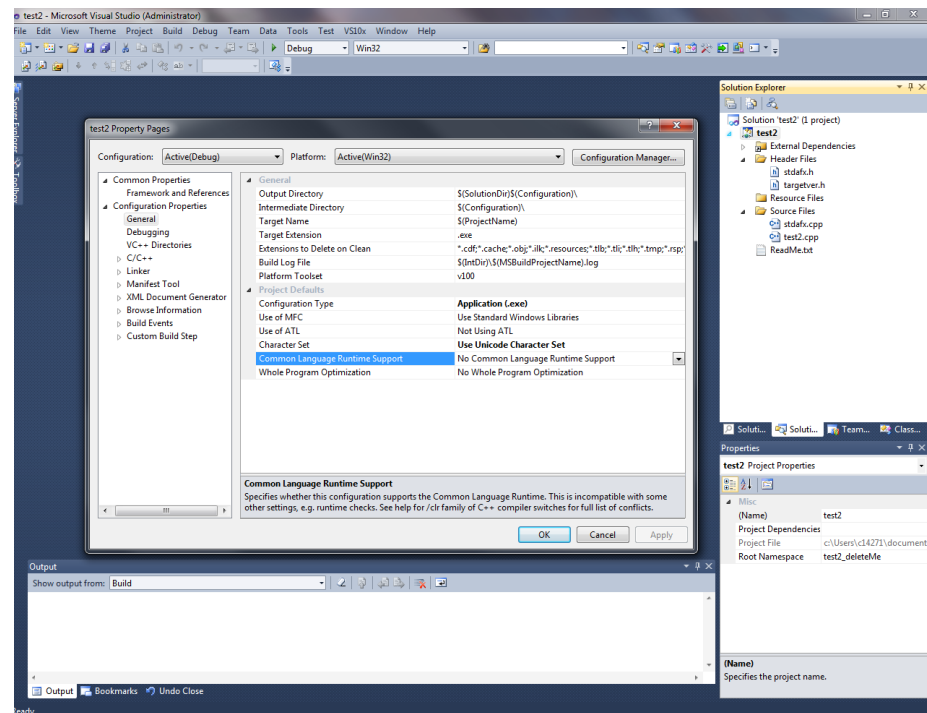
The SimpleIO DLL comes in two different forms, managed or unmanaged. **If the SimpleIO file has a “-UM” appended to its name, it is **unmanaged** and if a “-M” is appended, the DLL is **managed**.** Older versions of this DLL that are named without any suffix are **managed** versions. The managed DLL requires that the **.NET framework (v2.0 or higher)** to be installed in order to work. The unmanaged version does not have this requirement, but it does require the **Microsoft Visual C++ 2008 Redistributable Package** to be installed. However, accessing the managed version of the DLL is a more simple process than accessing the unmanaged version. Before choosing one of these versions, it is important to know which version is the appropriate one to use for your project. In general, an unmanaged application uses the unmanaged DLL and vice-versa. This document will cover the use of the SimpleIO-UM.dll library file. If you are unsure which version you need, you can use the following section below to determine this.

## How to Use Visual Studio to Find Which DLL Your Application Needs

If you are using Microsoft Visual Studio as your development tool, chances are that your program is managed by the .NET Framework. And likewise, if your development tool is not Visual Studio, it is likely your project is unmanaged. Either way, it is wise to be certain. To find out if your Visual Studio project is managed or unmanaged, left-click on the project in the “Solution Explorer” window and then click the “Project” menu item and select the “Properties” option.



Once this is done, expand the “Configuration Properties” menu and then click on the general category. The “Common Language Runtime Support” option should be set to “No Common Language Runtime Support” if your project is unmanaged. If it is set to any other option then your project is managed. You should see something similar to what is shown below.



## Sample Code for an Unmanaged Application (C++)

```
/*
 *          Microchip End User's License Agreement
 *
 * MICROCHIP SOFTWARE NOTICE AND DISCLAIMER: You may use this software, and any
 * derivatives created by any person or entity by or on your behalf, exclusively
 * with Microchip's products. Microchip and its licensors retain all ownership
 * and intellectual property rights in the accompanying software and in all
 * derivatives hereto.
 *
 * This software and any accompanying information is for suggestion only.
 * It does not modify Microchip's standard warranty for its products. You agree
 * that you are solely responsible for testing the software and determining its
 * suitability. Microchip has no obligation to modify, test, certify, or support
 * the software.
 *
 * THIS SOFTWARE IS SUPPLIED BY MICROCHIP "AS IS". NO WARRANTIES, WHETHER EXPRESS,
 * IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF
 * NON-INFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO
 * THIS SOFTWARE, ITS INTERACTION WITH MICROCHIP'S PRODUCTS, COMBINATION WITH ANY
 * OTHER PRODUCTS, OR USE IN ANY APPLICATION.
 *
 * IN NO EVENT, WILL MICROCHIP BE LIABLE, WHETHER IN CONTRACT, WARRANTY, TORT
 * (INCLUDING NEGLIGENCE OR BREACH OF STATUTORY DUTY), STRICT LIABILITY, INDEMNITY,
 * CONTRIBUTION, OR OTHERWISE, FOR ANY INDIRECT, SPECIAL, PUNITIVE, EXEMPLARY,
 * INCIDENTAL OR CONSEQUENTIAL LOSS, DAMAGE, FOR COST OR EXPENSE OF ANY KIND
 * WHATSOEVER RELATED TO THE SOFTWARE, HOWSOEVER CAUSED, EVEN IF MICROCHIP HAS
 * BEEN ADVISED OF THE POSSIBILITY OR THE DAMAGES ARE FORESEEABLE. TO THE FULLEST
 * EXTENT ALLOWABLE BY LAW, MICROCHIP'S TOTAL LIABILITY ON ALL CLAIMS IN ANY WAY
 * RELATED TO THIS SOFTWARE WILL NOT EXCEED THE AMOUNT OF FEES, IF ANY, THAT YOU
 * HAVE PAID DIRECTLY TO MICROCHIP FOR THIS SOFTWARE.
 *
 * MICROCHIP PROVIDES THIS SOFTWARE CONDITIONALLY UPON YOUR ACCEPTANCE OF THESE TERMS
 *
 */

#include "StdAfx.h"
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <string>
#include <Windows.h>           //Must include this header in program.

using namespace std;

int main( int argc, char * argv[] )
{
    //Variables
    int userSelection = 0;           //User input when requested for main menu
    static unsigned int mcp2200_VID = 0x04D8;    //VID for MCP2200
}
```

```

static unsigned int mcp2200_PID = 0x00DF; //PID for MCP2200
bool connectedStatus = false; //Connection status of MCP2200
unsigned int result = 0; //Result of function calls with uint returned

//STEP 1: Get handle to DLL- Path and name (Ex. C:\\SimpleIO-UM.dll) or just name
// if in working directory (put this in the quotes)
HINSTANCE DLL_handle = LoadLibrary(TEXT("SimpleIO-UM.dll"));
//Print result of LoadLibrary call
if( DLL_handle == NULL ) //If it is null, LoadLibrary call failed
{
    DWORD error = GetLastError();
    cout << "Loading of the DLL failed\n";
    cout << "The error was: " << error << "\n\n";
    return -1;
}
else
{
    cout << "DLL has been loaded\n\n";
}

//STEP 2: Get pointer to the function in the DLL
FARPROC lpfnGetProcessID0 = GetProcAddress(HMODULE (DLL_handle), "InitMCP2200");
FARPROC lpfnGetProcessID1 = GetProcAddress(HMODULE (DLL_handle), "IsConnected");

//STEP 3: Define the Function in the DLL for reuse.
// (Prototyping the DLL's function) Use "stdcall" calling convention
typedef void (__stdcall * pICFUNC0)(unsigned int, unsigned int);
typedef bool (__stdcall * pICFUNC1)();
//Same step as above.
pICFUNC0 DLL_InitMCP2200 = pICFUNC0(lpfnGetProcessID0);
pICFUNC1 DLL_IsConnected = pICFUNC1(lpfnGetProcessID1);

//STEP 4: Call the DLL function through the prototype name given in step 3
//Initialize the MCP2200 - NOTE: Must be plugged in when program is ran
DLL_InitMCP2200(mcp2200_VID, mcp2200_PID);
cout << "The MCP2200 was successfully initialized.\n";
//Check connection status.
connectedStatus = DLL_IsConnected();
if(connectedStatus == true)
    cout << "The device is CONNECTED";
else
    cout << "The device is DISCONNECTED";
cout << "\n\n";

//STEP 5: Release the DLL
FreeLibrary(DLL_handle);

return 0;
}

```

# Simple IO API

## Summary:

```
void __stdcall SimpleIOClass::InitMCP2200(unsigned int VendorID,
                                         unsigned int ProductID)

bool __stdcall SimpleIOClass::IsConnected()

bool __stdcall SimpleIOClass::ConfigureMCP2200(unsigned char IOMap,
                                              unsigned long BaudRateParam,
                                              unsigned int RxLEDMode,
                                              unsigned int TxLEDMode,
                                              bool FLOW,
                                              bool ULOAD,
                                              bool SSPND)

bool __stdcall SimpleIOClass::SetPin(unsigned int pin)
bool __stdcall SimpleIOClass::ClearPin(unsigned int pin)
int __stdcall SimpleIOClass::ReadPinValue(unsigned int pin)
bool __stdcall SimpleIOClass::ReadPin(unsigned int pin,
                                       unsigned int *returnValue)

bool __stdcall SimpleIOClass::WritePort(unsigned int portValue)
bool __stdcall SimpleIOClass::ReadPort(unsigned int *returnValue)
int __stdcall SimpleIOClass::ReadPortValue()
int __stdcall SimpleIOClass::SelectDevice(unsigned int uiDeviceNo)
int __stdcall SimpleIOClass::GetSelectedDevice()
unsigned int __stdcall SimpleIOClass::GetNoOfDevices()
void __stdcall SimpleIOClass::GetDeviceInfo(unsigned int uiDeviceNo,
                                           LPSTR strOutput)

void __stdcall SimpleIOClass::GetSelectedDeviceInfo(LPSTR strOutput)
int __stdcall SimpleIOClass::ReadEEPROM(unsigned int uiEEPAddress)
int __stdcall SimpleIOClass::WriteEEPROM(unsigned int uiEEPAddress,
                                         unsigned char ucValue)
```

While ConfigureMCP2200 configures the device with one call, it may also be configured one parameter at a time:

```
bool __stdcall SimpleIOClass::fnRxLED(unsigned int mode)
bool __stdcall SimpleIOClass::fnTxLED(unsigned int mode)
bool __stdcall SimpleIOClass::fnHardwareFlowControl(unsigned int onOff)
bool __stdcall SimpleIOClass::fnULoad(unsigned int onOff)
bool __stdcall SimpleIOClass::fnSuspend(unsigned int onOff)
bool __stdcall SimpleIOClass::fnSetBaudRate(unsigned long BaudRateParam)
bool __stdcall SimpleIOClass::ConfigureIO(unsigned char IOMap)
bool __stdcall SimpleIOClass::ConfigureIoDefaultOutput(unsigned char ucIoMap,
                                                         unsigned char ucDefValue)
```

## Constants:

```
const unsigned int OFF = 0;
const unsigned int ON = 1;
const unsigned int TOGGLE = 3;
const unsigned int BLINKSLOW = 4;
const unsigned int BLINKFAST = 5;
```

---

## 1. InitMCP2200

### Function:

```
void __stdcall SimpleIOClass::InitMCP2200 (unsigned int VendorID,  
                                           unsigned int ProductID)
```

### Summary:

Configures the Simple IO class for a specific Vendor and product ID.

### Description:

Sets the Vendor and Product ID used for the project.

### Precondition:

None

### Parameters:

Vendor ID - Assigned by USB IF ([www.usb.org](http://www.usb.org))  
Product ID - Assigned by the Vendor ID Holder

### Returns:

none

### Example:

```
InitMCP2200 (0x4D8, 0x00DF);
```

### Remarks:

Call this function before any other calls to set the Vendor and Product IDs.

---

## 2. IsConnected

### Function:

```
bool __stdcall SimpleIOClass::IsConnected()
```

### Summary:

Checks with the OS to see if the current VID/PID device is connected

### Description:

Checks if a MCP2200 is connected to the computer and if so it returns true, otherwise the result will be false

### Precondition:



VID & PID must have been previously set via a call to InitMCP2200(VID, PID)

**Parameters:**

none

**Returns:**

true if the device is connected to the host.  
false if the device is not connected to the host.

**Example:**

```
<code>
    unsigned int rv;
    if (IsConnected ())
    {
        lblStatusBar->Text = "Device connected";
    }
    else
        lblStatusBar->Text = "Device Disconnected";
</code>
```

**Remarks:**

No actual communication with the end device is conducted. The function inquiries with the OS to see if the specified VID/PID has enumerated.

---

### 3. ConfigureMCP2200

**Function:**

```
bool __stdcall SimpleIOClass::ConfigureMCP2200 (unsigned char IOMap,
                                                unsigned long BaudRateParam,
                                                unsigned int RxLEDMODE,
                                                int TxLEDMODE,
                                                bool FLOW,
                                                bool ULOAD,
                                                bool SSPND)
```

**Summary:**

Configures the device.

**Description:**

Sets the default GPIO designation, baudrate, TX/RX Led modes, flow control

**Precondition:**

The Vendor and Product ID must have been specified by SimpleIOInit.

Parameters:

IOMap - A byte which represents the input/output state of the pins  
(each bit may be either a 1 for input, and 0 for output).

BaudRateParam - the default communication baudrate

RxLEDMode - can take one of the constant values (OFF, ON, TOGGLE,  
BLINKSLOW, BLINKFAST) in order to define the behavior of  
the RX Led

```
OFF = 0;
ON = 1;
TOGGLE = 3;
BLINKSLOW = 4;
BLINKFAST = 5;
```

TxLEDMode - can take one of the defined values (OFF, ON, TOGGLE,  
BLINKSLOW, BLINKFAST) in order to define the behavior of  
the TX Led

FLOW - This parameter establishes the default flow control method  
(false - no HW flow control, true - RTS/CTS flow control)

ULOAD - This parameter establishes if the pin is configured as USBCFG  
status.

SSPND - This parameter establishes if the pin is configured as SSPND  
status.

Returns:

Function returns true if the transmission is successful and returns  
false if the transmission fails.

Example:

```
if (ConfigureMCP2200(0x43, 9600, BLINKSLOW, BLINKFAST, false, false,
                    false) == SUCCESS)
    lblStatusBar->Text = "Success";
else
    lblStatusBar->Text = "Invalid command "
```

Remarks:

None

---

## 4. fnRxLED

Function:

```
bool __stdcall SimpleIOClass::fnRxLED (unsigned int mode)
```

Summary:

Configures the Rx LED mode. Rx LED configuration will be stored in NVRAM.

Description:

Sets the Rx Led mode to one of the possible values and it also sets the remaining of the relevant parameters (GPIO designation, baudrate, flow control, Tx Led) with the default values as they're assigned either at the call to the `ConfigureMCP2200()` or with the default values read back from the device itself

Precondition:

The Vendor and Product ID must have been specified by `InitMCP2200()`.

Parameters:

mode (constant): OFF, TOGGLE, BLINKSLOW, BLINKFAST

Returns:

returns False if the transmission fails.

Example:

```
if (fnRxLED (BLINKFAST) == SUCCESS)
    lblStatusBar->Text = "Success";
else
    lblStatusBar->Text = "Invalid command " + GetLastError();
```

Remarks:

Error code is returned in `GetLastError()`

---

## 5. fnTxLED

Function:

```
bool __stdcall SimpleIOClass::fnTxLED (unsigned int onOff,
                                       unsigned int mode)
```

Summary:

Configures the Tx LED mode. Tx LED configuration will be stored NVRAM.

Description:

Sets the Tx Led mode to one of the possible values and it also sets the remaining of the relevant parameters (GPIO designation, baudrate, flow control, Tx Led) with the default values as they're assigned either at

the call to the `ConfigureMCP2200()` or with the default values read back from the device itself

**Precondition:**

The Vendor and Product ID must have been specified by `InitMCP2200()`.

**Parameters:**

mode (constant): OFF, TOGGLE, BLINKSLOW, BLINKFAST

**Returns:**

Function returns true if the transmission is successful returns False if the transmission fails.

**Example:**

```
if (fnTxLED (BLINKSLOW) == SUCCESS)
    lblStatusBar->Text = "Success";
else
    lblStatusBar->Text = "Invalid command " + GetLastError();
```

**Remarks:**

Error code is returned in `GetLastError()`

---

## 6. fnHardwareFlowControl

**Function:**

```
bool __stdcall SimpleIOClass::fnHardwareFlowControl (
    unsigned int onOff)
```

**Summary:**

Configures the flow control of the MCP2200. The flow control configuration will be stored in NVRAM

**Description:**

Sets the flow control to HW flow control (RTS/CTS) or No flow control

**Precondition:**

The Vendor and Product ID must have been specified by `InitMCP2200()`

**Parameters:**

onOff -        1 - if Hw flow control needed  
              0 - if No flow control needed

**Returns:**

Function returns true if the transmission is successful returns False if the transmission fails.

Example:

```
if (fnHardwareFlowControl(1) == SUCCESS)
    lblStatusBar->Text = "Success";
else
    lblStatusBar->Text = "Invalid command " + GetLastError();
```

Remarks:

Error code is returned in GetLastError()

---

## 7. fnULoad

Function:

```
bool __stdcall SimpleIOClass::fnULoad(unsigned int onOff)
```

Summary:

Configures the GP1 pin of the MCP2200 to show the status of the USB configuration

Description:

When the GP1 is designated to show the USB configuration status, the pin will start low (during power-up or after reset) and it will go high after the MCP2200 is successfully configured by the host

Precondition:

The Vendor and Product ID must have been specified by InitMCP2200()

Parameters:

onOff -	1 - GP1 will reflect the USB configuration status
	0 - GP1 will not reflect the USB configuration status (can be used as GPIO)

Returns:

Function returns true if the transmission is successful returns False if the transmission fails.

Example:

```
if (fnULoad(1) == SUCCESS)
    lblStatusBar->Text = "Success";
else
    lblStatusBar->Text = "Invalid command " + GetLastError();
```

Remarks:

Error code is returned in GetLastError()

---

## 8. fnSuspend

### Function:

```
bool __stdcall SimpleIOClass::fnSuspend(unsigned int onOff)
```

### Summary:

Configures the GP0 pin of the MCP2200 to show the status of Suspend/Resume USB states

### Description:

When the GP0 is designated to show the USB Suspend/Resume states, the pin will go low when the Suspend state is issued or will go high when the Resume state is on

### Precondition:

The Vendor and Product ID must have been specified by InitMCP2200()

### Parameters:

onOff -        1 - GP0 will reflect the USB Suspend/Resume states  
              0 - GP0 will not reflect the USB Suspend/Resume states (can be used as GPIO)

### Returns:

Function returns true if the transmission is successful returns False if the transmission fails.

### Example:

```
if (fnSuspend(1) == SUCCESS)
    lblStatusBar->Text = "Success";
else
    lblStatusBar->Text = "Invalid command " + GetLastError();
```

### Remarks:

Error code is returned in GetLastError()

---

## 9. fnSetBaudRate

### Function:

```
bool __stdcall SimpleIOClass::fnSetBaudRate (
                                         unsigned long BaudRateParam)
```

### Summary:

Configures the device's default baudrate. The baudrate value will be stored in NVRAM.

**Description:**

Sets the desired baudrate and it will store it into device's NVRAM.

**Precondition:**

The Vendor and Product ID must have been specified by InitMCP2200.

**Parameters:**

BaudRateParam - the desired baudrate value

**Returns:**

Function returns true if the transmission is successful and returns false if the transmission fails.

**Example:**

```
if (fnSetBaudRate(9600) == SUCCESS)
    lblStatusBar->Text = "Success";
else
    lblStatusBar->Text = "Invalid command " + GetLastError();
```

**Remarks:**

Error code is returned in GetLastError()

---

## 10.ConfigureIO

**Function:**

```
bool __stdcall SimpleIOClass::ConfigureIO (unsigned char IOMap)
```

**Summary:**

Configures the GPIO pins for Digital Input, Digital Output

**Description:**

GPIO Pins can be configured as Digital Input, Digital Output

**Precondition:**

The Vendor and Product ID must have been specified by InitMCP2200.

**Parameters:**

IOMap - a byte which represents a bitmap of the GPIO configuration  
a bit set to '1' will be a digital input  
a bit set to '0' will be a digital output  
MSB - - - - - LSB  
GP7 GP6 GP5 GP4 GP3 GP2 GP1 GP0

**Returns:**

Function returns true if the transmission is successful and returns false if the transmission fails.

Example:

```
if (ConfigureIO(0xA5) == SUCCESS)
    lblStatusBar->Text = "Success";
else
    lblStatusBar->Text = "Invalid command " + GetLastError();
```

Remarks:

Error code is returned in GetLastError()

---

## 11.ConfigureIoDefaultOutput

Function:

```
bool __stdcall SimpleIOClass::ConfigureIoDefaultOutput(
    unsigned char ucIoMap,
    unsigned char ucDefValue)
```

Summary:

Configures the IO pins for Digital Input, Digital Output and also the default output latch value

Description:

IO Pins can be configured as Digital Input, Digital Output  
The default output latch value is received as a parameter

Precondition:

The Vendor and Product ID must have been specified by InitMCP2200.

Parameters:

ucIoMap - a byte containing a bit-map used to set the GPIOs as either input or output

1 - GPIO configured as input

0 - GPIO configured as output

MSB    -    -    -    -    -    -    LSB

GP7   GP6   GP5   GP4   GP3   GP2   GP1   GP0

ucDefValue - the default value that will be loaded to the output latch  
(effect only on the pins configured as outputs)

Returns:

Function returns true if the transmission is successful and returns false if the transmission fails.

Example:

```
if (ConfigureIoDefaultOutput(IoMap, DefValue) == SUCCESS)
```



```
        lblStatusBar->Text = "Success";  
    else  
        lblStatusBar->Text = "Invalid command " + GetLastError();
```

Remarks:

Error code is returned in GetLastError()

---

## 12.SetPin

Function:

```
bool __stdcall SimpleIOClass::SetPin(unsigned int pin)
```

Summary:

Sets the specified pin.

Description:

Sets the specified pin to logic '1'.

Precondition:

Must have previously been configured as an output via a ConfigureIO or ConfigureIoDefaultOutput call.

Parameters:

pin - The pin number to set (0-7)

Returns:

Function returns true if the transmission is successful and returns false if the transmission fails.

Example:

```
if (SetPin (2))  
    lblStatusBar->Text = "Success";  
else  
    lblStatusBar->Text = "Invalid command " + GetLastError();
```

Remarks:

Error code is returned in GetLastError()

---

## 13.ClearPin

Function:

```
bool __stdcall SimpleIOClass::ClearPin(unsigned int pin)
```

**Summary:**

Clears the specified pin.

**Description:**

Clears the specified pin to logic '0'.

**Precondition:**

Must have previously been configured as an output via a ConfigureIO or ConfigureIoDefaultOutput call.

**Parameters:**

pin - The pin number to set (0-7)

**Returns:**

Function returns true if the transmission is successful returns False if the transmission fails.

**Example:**

```
if (ClearPin (2))
    lblStatusBar->Text = "Success";
else
    lblStatusBar->Text = "Invalid command " + GetLastError();
```

**Remarks:**

Error code is returned in GetLastError()

---

## 14.ReadPin

**Function:**

```
bool __stdcall SimpleIOClass::ReadPin(    unsigned int pin,
                                         unsigned int *returnvalue)
```

**Summary:**

Reads the specified pin.

**Description:**

Reads the specified pin and returns the value in returnvalue. If the pin has been configured as Digital Input, the return value will be either 0 or 1.

**Precondition:**

Must have previously been configured as an input via a ConfigureIO or ConfigureIoDefaultOutput call.

**Parameters:**

pin - The pin number to set (0-7)  
returnvalue - the value read on the pin (0 or 1)

**Returns:**

Function returns true if the transmission is successful  
returns False if the transmission fails.

**Example:**

```
unsigned int rv;  
if (ReadPin (0, &rv))  
    lblStatusBar->Text = "Success";  
else  
    lblStatusBar->Text = "Invalid command " + GetLastError();
```

**Remarks:**

Error code is returned in GetLastError()

---

## 15.ReadPinValue

**Function:**

```
int __stdcall SimpleIOClass::ReadPinValue(unsigned int pin)
```

**Summary:**

Reads the specified pin.

**Description:**

Reads the specified pin and returns the value as the return value. If the pin has been configured as Digital Input, the return value will be either 0 or 1.  
if an error occurs, the function will return a value of 0x8000

**Precondition:**

Must have previously been configured as an input via a ConfigureIO or ConfigureIoDefaultOutput call.

**Parameters:**

pin - The pin number to set (0-7)

**Returns:**

Function returns the read value of the pin and returns a value of 0x8000 if an error occurs.

**Example:**

```
unsigned int rv;  
if (ReadPinValue(0) != 0x8000)  
    lblStatusBar->Text = "Success";
```

```
else
    lblStatusBar->Text = "Invalid command " + GetLastError();
```

Remarks:

Error code is returned in GetLastError().

---

## 16. WritePort

Function:

```
bool __stdcall SimpleIOClass::WritePort(unsigned int portValue)
```

Summary:

Writes a value to the GPIO port.

Description:

Writes the GPIO port. This provides a means to write all pins at once instead of one-at-a-time.

Precondition:

Must have previously been configured as an input via a ConfigureIO or ConfigureIoDefaultOutput call.

Parameters:

portValue - Byte value to set on the port.

Returns:

Function returns true if the transmission is successful returns False if the transmission fails.

Example:

```
if (WritePort (0x5A))
    lblStatusBar->Text = "Success";
else
    lblStatusBar->Text = "Invalid command " + GetLastError();
```

Remarks:

Pins configured for output returns the current state of the port.  
Pins configured as input read as one.

---

## 17. ReadPort

Function:

```
bool __stdcall SimpleIOClass::ReadPort(unsigned int *returnvalue)
```

**Summary:**

Reads the GPIO port as digital input.

**Description:**

Reads the GPIO port and returns the value in returnvalue. This provides a means to read all pins at once instead of one-at-a-time.

**Precondition:**

Must have previously been configured as an input via a ConfigureIO or ConfigureIoDefaultOutput call.

**Parameters:**

pin - The pin number to set (0-7)  
returnvalue - the value read on the pin (0 or 1)

**Returns:**

Function returns true if the read is successful  
returns False if there the transmission fails.

**Example:**

```
unsigned int rv;  
if (ReadPort (&rv))  
    lblStatusBar->Text = "Success";  
else  
    lblStatusBar->Text = "Invalid command " + GetLastError();
```

**Remarks:**

Pins configured for output returns the current state of the port.  
Pins configured as input read as one.

---

## 18.ReadPortValue

**Function:**

```
int __stdcall SimpleIOClass::ReadPortValue()
```

**Summary:**

Reads the GPIO port as digital input.

**Description:**

Reads the GPIO port and returns the value of the port. This provides a mean to read all pins at once instead of one-at-a time. In case of an error the returned value will be 0x8000

**Precondition:**

Must have previously been configured as an input via a `ConfigureIO` or `ConfigureIoDefaultOutput` call.

**Parameters:**

None

**Returns:**

Function returns true if the read is successful  
returns False if the transmission fails.

**Example:**

```
int rv;
rv =ReadPortValue()
if (rv != 0x8000)
    lblStatusBar->Text = "Success";
else
    lblStatusBar->Text = "Invalid command " + GetLastError();
```

**Remarks:**

Pins configured for output returns the current state of the port.  
Pins configured as input read as one.

---

## 19.SelectDevice

**Function:**

```
int __stdcall SimpleIOClass::SelectDevice(unsigned int uiDeviceNo)
```

**Summary:**

Selects one of the active devices in the system

**Description:**

The function is used to select one of the detected devices in the system as the "active device"

**Precondition:**

At least one call to the `InitMCP2200()` is needed in order to have the DLL searching for the compatible devices. Also, in order to have the current number of devices in the system, call the `IsConnected()` function in order to update the number of connected devices available

**Parameters:**

`uiDeviceNo` - the ID of the device we want to select (can have a value between 0 and (number of devices - 1))

**Returns:**

Function returns 0 in case of selection success, otherwise it will return

E\_WRONG\_DEVICE\_ID (-1) - for a device ID that is out of range  
E\_INACTIVE\_DEVICE (-2) - for an inactive device

Example:

```
<code>
    int iResult;
    iResult = SelectDevice(1)           //Assuming 2 devices are connected
    if (iResult == 0)
    {
        lblStatusBar->Text = "Success";
    }
    else
        lblStatusBar->Text = "Error selecting device";
</code>
```

Remarks:

Call the IsConnected() prior to the call of this function  
in order to have the most recent number of devices that are present in  
the system.

---

## 20. GetSelectedDevice

Function:

```
int __stdcall SimpleIOClass::GetSelectedDevice(void)
```

Summary:

Returns the ID of the selected device

Description:

The function returns the ID of the current selected device.

Precondition:

At least one call to the InitMCP2200() is needed in order to have the DLL  
searching for the compatible devices.

Parameters:

None

Returns:

Function returns the ID of the current selected device. Its value can  
range from 0 to (number of devices - 1)

Example:

```
<code>
    lblStatusBar->Text = GetSelectedDevice();
</code>
```





**Summary:**

Returns the pathname for one of the connected devices

**Description:**

The function will return the pathname for the given device id

**Precondition:**

At least one call to the InitMCP2200() is needed in order to have the DLL searching for the compatible devices

**Parameters:**

uiDeviceNo - The device ID for which we need the path information  
Can have a value between 0 and (number of devices - 1)

**Returns:**

Function returns a string containing the pathname of the given device id. In the case the given ID is out of range, the function will return the "Device Index Error" string. In the case the device for which we need to have the pathname is not connected anymore, the return string will be "Device Not Connected".

**Example:**

```
<code>
char * result = new char[256];
GetDeviceInfo(0, result);
    //Output comes through parameter "result"
string strValue(result);
    lblStatusBar->Text = strValue;
</code>
```

**Remarks:**

None

---

## 23. GetSelectedDeviceInfo

**Function:**

```
void __stdcall SimpleIOClass::GetSelectedDeviceInfo(LPSTR strOutput)
```

**Summary:**

Returns the selected device pathname through the parameter "strOutput".

**Description:**

The function returns a string containing the unique pathname of the selected device.

**Precondition:**

At least one call to the InitMCP2200() is needed in order to have the DLL searching for the compatible devices.

**Parameters:**

None

**Returns:**

Function returns a string containing the unique pathname of the selected device.

**Example:**

```
<code>
char * result = new char[256];
GetSelectedDeviceInfo(result);
//Output comes through parameter "result"
string strValue(result);
lblStatusBar->Text = strValue;
</code>
```

**Remarks:**

The default selected device is the first one that the DLL finds. If the user wants to retrieve other device's pathname (assuming more than one device is present in the system), a call to `SelectDevice(deviceNo)` is needed.

---

## 24. ReadEEPROM

**Function:**

```
int __stdcall SimpleIOClass::ReadEEPROM(unsigned int uiEEPAddress)
```

**Summary:**

Reads a byte from the chip's EEPROM.

**Description:**

Reads a byte from the EEPROM at the given address.

**Precondition:**

At least one call to the `InitMCP2200()` is needed in order to have the DLL searching for the compatible devices.

**Parameters:**

`uiEEPAddress` - The EEPROM address location we need to write to (must be from 0 to 255incl.).

**Returns:**

Function returns any positive value as being the EEPROM's location value  
`E_WRONG_ADDRESS` (-3) - in case the given EEPROM address is out of range  
`E_CANNOT_SEND_DATA` (-4) in case the function cannot send the command to the device.

**Example:**

```
<code>
int iRetValue = ReadEEPROM(0x01, 0xAB);

if (iRetValue >= 0)
{
    lblStatusBar->Text = "Success";
}
```

```
        else
            lblStatusBar->Text = "Error reading to EEPROM " + GetLastError();
</code>
```

Remarks:  
None

---

## 25. WriteEEPROM

### Function:

```
int __stdcall SimpleIOClass::WriteEEPROM(unsigned int uiEEPAddress,
                                         unsigned char ucValue)
```

### Summary:

Writes a byte into the chip's EEPROM.

### Description:

Writes a byte at the given address into the internal 256 bytes EEPROM.

### Precondition:

At least one call to the InitMCP2200() is needed in order to have the DLL searching for the compatible devices.

### Parameters:

uiEEPAddress - The EEPROM address location we need to write to (must be from 0 to 255incl.)  
ucValue - the byte value we need to write to the given location

### Returns:

The function returns 0 if the write command was successfully sent to the Device.  
E\_WRONG\_ADDRESS (-3) - in case the given EEPROM address is out of range  
E\_CANNOT\_SEND\_DATA (-4) - in case the function cannot send the command to the device

### Example:

```
<code>
int iRetVal = WriteEEPROM(0x01, 0xAB);

    if (iRetVal == 0)
    {
        lblStatusBar->Text = "Success";
    }
    else
        lblStatusBar->Text = "Error writing to EEPROM " + GetLastError();
</code>
```

### Remarks:

The function will send the write EEPROM command but it has no confirmation whether the EEPROM location was actually written. In order to verify the correctness of EEPROM write, the user can issue a ReadEEPROM() and check if the returned value matched the written one.

