

# MCP2221 DLL User Manual

---

## Contents

Document Revision History.....	6
Which DLL version to choose? .....	7
DLL Requirements.....	7
API Overview.....	8
DLL Structure:.....	8
DLL Initialization: .....	8
Function List (Unmanaged): .....	9
Function List (Managed):.....	12
DllConstants: .....	14
DLL Error Codes.....	15
Detailed API Function List .....	17
How to read detailed function description: .....	17
DllInit .....	18

DllInitCustom.....	18
GetUsbVid .....	18
GetUsbPid.....	19
SetUsbVidPid .....	19
GetUsbPowerSource .....	20
SetUsbPowerSource .....	20
GetUsbRemoteWakeupEnable.....	20
SetUsbRemoteWakeupEnable .....	21
GetUsbCurrentRequirement .....	21
SetUsbCurrentRequirement.....	22
GetUsbStringManufacturer.....	22
SetUsbStringManufacturer .....	23
GetUsbStringDescriptor .....	23
SetUsbStringDescriptor .....	24
GetSerialNumber.....	25
SetSerialNumber .....	25
GetFactorySerialNumber.....	26
GetSerialNumberEnumerationEnable.....	26

SetSerialNumberEnumerationEnable .....	27
GetFirmwareVersion .....	27
GetHardwareVersion.....	28
GetConnectionStatus .....	29
GetFlashProtectionState .....	29
SetFlashProtectionOff .....	29
SetFlashPermanentLock .....	30
SetFlashPasswordProtection .....	30
GetInterruptPinMode.....	31
SetInterruptPinMode .....	31
GetClockPinDividerValue .....	32
GetClockPinDutyCycle .....	32
SetClockPinConfiguration.....	32
GetGpConfiguration .....	33
GetGpPinDirection .....	35
SetGpPinDirection .....	35
GetDacVoltageReference .....	35
SetDacVoltageReference.....	36

GetDacValue.....	36
SetDacValue .....	37
GetAdcVoltageReference .....	37
SetAdcVoltageReference.....	38
GetInitialPinValueLedUartTx .....	38
SetInitialPinValueLedUartTx.....	39
GetInitialPinValueLedUartRx.....	39
SetInitialPinValueLedUartRx .....	40
GetInitialPinValueLedI2c .....	40
SetInitialPinValueLedI2c.....	40
GetInitialPinValueSspnd .....	41
SetInitialPinValueSspnd.....	41
GetInitialPinValueUsbcfg.....	42
SetInitialPinValueUsbcfg .....	42
ResetDevice .....	42
GetAdcData .....	43
ClearInterruptPinState .....	44
ReadGpioPinValue.....	44

WriteGpioPinValue.....	44
EnterAccessPassword.....	45
WriteI2cData .....	45
ReadI2cData .....	46
SmbWriteBlock.....	47
SmbReadBlock.....	48
StopI2cDataTransfer.....	48
GetDevCount.....	49
GetSelectedDevNumber.....	49
GetSelectedDevInfo .....	50

## Document Revision History

Version	Release Date	Description
V1.0	05/08/2014	Initial release
V1.1	06/23/2014	Text added for clarification of “GetClockPinDividerValue” function and modified DLL requirements for the managed DLL change.

## Which DLL version to choose?

The DLL comes in two different versions: managed and unmanaged. The managed DLL utilizes the Microsoft .NET framework when the unmanaged does not. To get help on which version to use, follow the guidelines below:

Scenario:	Which version to use:
You are planning to use the DLL with a .NET application	Managed
You are looking for the most simple way to interface with this DLL	Managed
You are using Visual Studio IDE	Managed
You do not want your application to require the .NET framework	Unmanaged
You are using programming tools/languages like Python, C++, LabVIEW, etc.	Unmanaged

## DLL Requirements

A breakdown of the requirements is shown below:

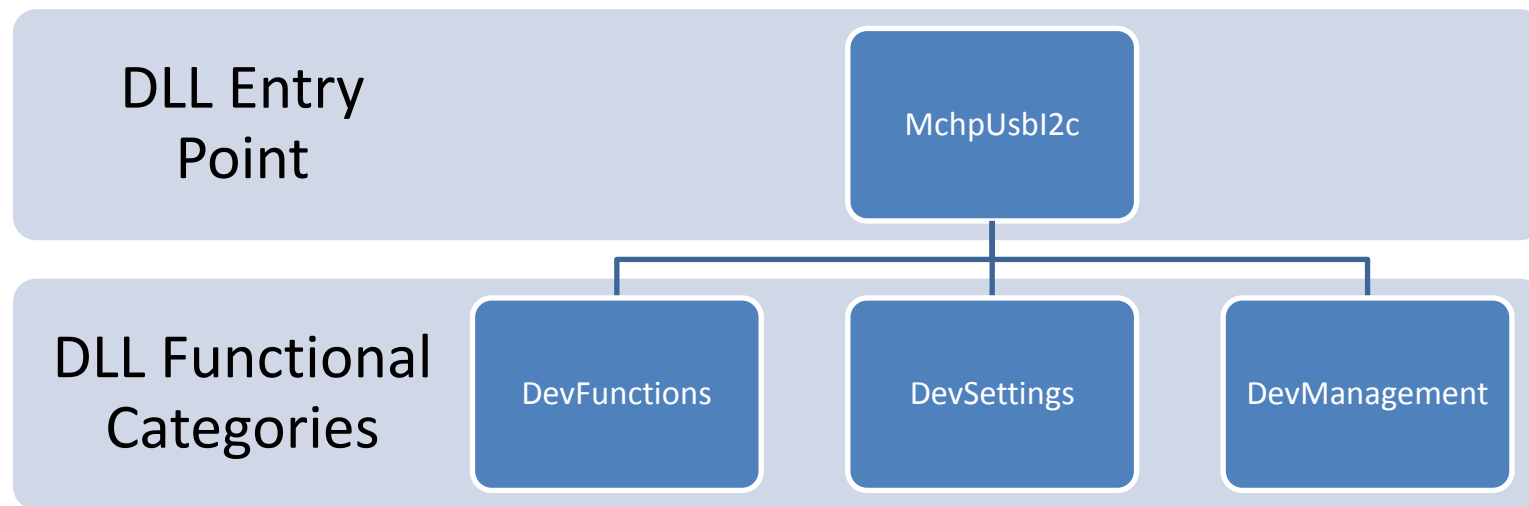
DLL Version:	Requirements:
Managed (.NET4 version)	<ol style="list-style-type: none"><li>1. .NET framework (v4 Client profile or higher)</li><li>2. Microsoft Visual C++ 2010 Redistributable Package (x86) (<b>OR</b> msvcp100.dll and msucr100.dll files in MCP2221 DLL directory)</li></ol>
Managed (.NET2 version)	<ol style="list-style-type: none"><li>1. .NET framework (v2 or V3.5)</li><li>2. Microsoft Visual C++ 2008 Redistributable Package (x86) (<b>OR</b> msvcp90.dll, msucr90.dll, and msucm90.dll files in MCP2221 DLL directory)</li></ol>
Unmanaged	<ol style="list-style-type: none"><li>1. Microsoft Visual C++ 2013 Redistributable Package (x86) (<b>OR</b> msvcp120.dll and msucr120.dll files in MCP2221 DLL directory)</li></ol>

## API Overview

The two versions of the DLL have nearly the same API structure, but there are some subtle differences for a handful of functions. Refer to the individual function description to ensure the appropriate API is utilized for the version of the DLL chosen.

### DLL Structure:

Both the managed and unmanaged DLL's have the same structure, however, this structure is more apparent in the managed DLL. The reason for this has to do with how a .NET DLL exposes the classes that make up the DLL. In contrast, an unmanaged DLL exposes the individual functions, thus hiding the classes containing these functions.



### DLL Initialization:

One aspect that the DLL's differ is in the way they are initialized. The managed DLL will access the DLL through the "entry point" class named "MchpUsbI2c", where the unmanaged DLL will call a specific function to initialize the DLL. See the table below for the specifics:



DLL Version:	Initialization steps:
Unmanaged	Call "DllInit" to initialize the DLL with default MCP2221 VID/PID. Or Call "DllInitCustom" and pass in the VID/PID to be used.
Managed	Create new instance of MchpUsbI2c class and pass in the VID/PID as parameters. <i>Ex. MCP2221.MchpUsbI2c usbI2c = new MchpUsbI2c(vid, pid);</i>

### Function List (Unmanaged):

```

//          DLL Initialization          //
void          DllInit();
void          DllInitCustom(UINT32 vid, UINT32 pid);

//          USB          //
INT64         DevSettings::GetUsbVid();
INT64         DevSettings::GetUsbPid();
int           DevSettings::SetUsbVidPid(UINT32 newVid, UINT32 newPid);
int           DevSettings::GetUsbPowerSource();
int           DevSettings::SetUsbPowerSource(int newPowerSource);
int           DevSettings::GetUsbRemoteWakeupEnable();
int           DevSettings::SetUsbRemoteWakeupEnable(bool remoteWakeupEnable);
int           DevSettings::GetUsbCurrentRequirement();
int           DevSettings::SetUsbCurrentRequirement(int newCurentValue);
int           DevSettings::GetUsbStringManufacturer(wchar_t * manString);
int           DevSettings::SetUsbStringManufacturer(wchar_t * manString);
int           DevSettings::GetUsbStringDescriptor(wchar_t *);
int           DevSettings::SetUsbStringDescriptor(wchar_t *);
int           DevSettings::GetSerialNumber(wchar_t *);
int           DevSettings::SetSerialNumber(wchar_t *);
int           DevSettings::GetFactorySerialNumber(wchar_t *);

```

```

int      DevSettings::GetSerialNumberEnumerationEnable();
int      DevSettings::SetSerialNumberEnumerationEnable(bool enableSerialNumberEnumeration);
int      DevSettings::GetFirmwareVersion(char*);
int      DevSettings::GetHardwareVersion(char*);
bool     DevSettings::GetConnectionStatus();

//          FLASH ACCESS CONTROL          //
int      DevSettings::GetFlashProtectionState();
int      DevSettings::SetFlashProtectionOff();
int      DevSettings::SetFlashPermanentLock();
int      DevSettings::SetFlashPasswordProtection(char*);

//          Interrupt pin and GP clock      //
int      DevSettings::GetInterruptPinMode(int whichToGet);
int      DevSettings::SetInterruptPinMode(int whichToSet, int interruptPinModeToSet);
int      DevSettings::GetClockPinDividerValue(int whichToGet);
int      DevSettings::GetClockPinDutyCycle(int whichToGet);
int      DevSettings::SetClockPinConfiguration(int whichToSet, int clockDividerValueToSet);

//          GPIO Configuration              //
int      DevSettings::GetGpPinDirection(int whichToGet, BYTE pinNumber);
int      DevSettings::SetGpPinDirection(int whichToSet, BYTE pinNumber, BYTE directionToSet);
int      DevSettings::GetGpPinConfiguration(int whichToGet, BYTE* gpPinDesignations, BYTE* gpPinDirections, BYTE*
                                           gpPinOutputLatches);
int      DevSettings::SetGpPinConfiguration(int whichToSet, BYTE* gpPinDesignations, BYTE* gpPinDirections, BYTE*
                                           gpPinOutputLatches);

//          DAC & ADC                      //
int      DevSettings::GetDacVoltageReference();
int      DevSettings::SetDacVoltageReference(int whichToSet, int vRefValue);
int      DevSettings::GetDacValue();
int      DevSettings::SetDacValue(int whichToSet, int dacValue);
int      DevSettings::GetAdcVoltageReference();
int      DevSettings::SetAdcVoltageReference(int whichToSet, int vRefValue);

```

```

//          Initial Pin Values          //
int         DevSettings::GetInitialPinValueLedUartTx();
int         DevSettings::SetInitialPinValueLedUartTx(int newInitialPinValue);
int         DevSettings::GetInitialPinValueLedUartRx();
int         DevSettings::SetInitialPinValueLedUartRx(int newInitialPinValue);
int         DevSettings::GetInitialPinValueLedI2c();
int         DevSettings::SetInitialPinValueLedI2c(int newInitialPinValue);
int         DevSettings::GetInitialPinValueSspnd();
int         DevSettings::SetInitialPinValueSspnd(int newInitialPinValue);
int         DevSettings::GetInitialPinValueUsbcfg();
int         DevSettings::SetInitialPinValueUsbcfg(int newInitialPinValue);

//          Device Operations/Functions          //
int         DevFunctions::ResetDevice();
int         DevFunctions::GetAdcData(WORD* adcData);
int         DevFunctions::ClearInterruptPinState();
int         DevFunctions::ReadGpioPinValue(BYTE pinNumber);
int         DevFunctions::WriteGpioPinValue(BYTE pinNumber, BYTE pinValue);
int         DevFunctions::EnterAccessPassword(char* accessPassword);
//          I2C OPERATIONS          //
int         DevFunctions::WriteI2cData(BYTE i2cAddress, BYTE* i2cDataToSend, UINT32 numberOfBytesToWrite, UINT32
                                     i2cBusSpeed);
int         DevFunctions::ReadI2cData(BYTE i2cAddress, BYTE* i2cDataReceived, UINT32 numberOfBytesToRead, UINT32
                                     i2cBusSpeed);
int         DevFunctions::StopI2cDataTransfer();
//          SMBus OPERATIONS          //
int         DevFunctions::SmbWriteBlock(BYTE smbAddress, BYTE* smbDataToSend, UINT32 numberOfBytesToWrite, UINT32
                                     smbSpeed, BYTE usesPEC);
int         DevFunctions::SmbReadBlock(BYTE smbAddress, BYTE* smbDataToRead, UINT32 numberOfBytesToRead, UINT32
                                     smbSpeed, BYTE usesPEC, BYTE readRegIndex);

//          Multiple device management          //
int         DevManagement::GetDeviceCount()
int         DevManagement::GetSelectedDevNumber()
int         DevManagement::GetSelectedDevInfo(char* devInformation)

```

```
int          DevManagement::SelectDev(int devNum)
```

#### Function List (Managed):

```
        //          USB          //
INT64    DevSettings_M::GetUsbVid();
INT64    DevSettings_M::GetUsbPid();
int      DevSettings_M::SetUsbVidPid(UINT32 newVid, UINT32 newPid);
int      DevSettings_M::GetUsbPowerSource();
int      DevSettings_M::SetUsbPowerSource(int newPowerSource);
int      DevSettings_M::GetUsbRemoteWakeupEnable();
int      DevSettings_M::SetUsbRemoteWakeupEnable(bool remoteWakeupEnable);
int      DevSettings_M::GetUsbCurrentRequirement();
int      DevSettings_M::SetUsbCurrentRequirement(int newCurentValue);
String^  DevSettings_M::GetUsbStringManufacturer();
int      DevSettings_M::SetUsbStringManufacturer(String^ manString);
String^  DevSettings_M::GetUsbStringDescriptor();
int      DevSettings_M::SetUsbStringDescriptor(String^ usbString);
String^  DevSettings_M::GetSerialNumber();
int      DevSettings_M::SetSerialNumber(String^ serialNumber);
String^  DevSettings_M::GetFactorySerialNumber();
int      DevSettings_M::GetSerialNumberEnumerationEnable();
int      DevSettings_M::SetSerialNumberEnumerationEnable(bool enableSerialNumberEnumeration);
String^  DevSettings_M::GetFirmwareVersion();
String^  DevSettings_M::GetHardwareVersion();
bool     DevSettings_M::GetConnectionStatus();

        //          FLASH ACCESS CONTROL          //
int      DevSettings_M::GetFlashProtectionState();
int      DevSettings_M::SetFlashProtectionOff();
int      DevSettings_M::SetFlashPermanentLock();
int      DevSettings_M::SetFlashPasswordProtection(String^ password);

        //          Interrupt pin and GP clock          //
int      DevSettings_M::GetInterruptPinMode(int whichToGet);
```

```

int      DevSettings_M::SetInterruptPinMode(int whichToSet, int interruptPinModeToSet);
int      DevSettings_M::GetClockPinDividerValue(int whichToGet);
int      DevSettings_M::GetClockPinDutyCycle(int whichToGet);
int      DevSettings_M::SetClockPinConfiguration(int whichToSet, int clockDividerValueToSet);

//          GPIO Configuration          //
int      DevSettings_M::GetGpPinDirection(int whichToGet, BYTE pinNumber);
int      DevSettings_M::SetGpPinDirection(int whichToSet, BYTE pinNumber, BYTE directionToSet);
int      DevSettings_M::GetGpPinConfiguration(int whichToGet, array<System::Byte>^ gpPinDesignations,
        array<System::Byte>^ gpPinDirections, array<System::Byte>^ gpPinOutputLatches);
int      DevSettings_M::SetGpPinConfiguration(int whichToSet, array<System::Byte>^ gpPinDesignations,
        array<System::Byte>^ gpPinDirections, array<System::Byte>^ gpPinOutputLatches);

//          DAC & ADC          //
int      DevSettings_M::GetDacVoltageReference();
int      DevSettings_M::SetDacVoltageReference(int whichToSet, int vRefValue);
int      DevSettings_M::GetDacValue();
int      DevSettings_M::SetDacValue(int whichToSet, int dacValue);
int      DevSettings_M::GetAdcVoltageReference();
int      DevSettings_M::SetAdcVoltageReference(int whichToSet, int vRefValue);

//          Initial Pin Values          //
int      DevSettings_M::GetInitialPinValueLedUartTx();
int      DevSettings_M::SetInitialPinValueLedUartTx(int newInitialPinValue);
int      DevSettings_M::GetInitialPinValueLedUartRx();
int      DevSettings_M::SetInitialPinValueLedUartRx(int newInitialPinValue);
int      DevSettings_M::GetInitialPinValueLedI2c();
int      DevSettings_M::SetInitialPinValueLedI2c(int newInitialPinValue);
int      DevSettings_M::GetInitialPinValueSspnd();
int      DevSettings_M::SetInitialPinValueSspnd(int newInitialPinValue);
int      DevSettings_M::GetInitialPinValueUsbcfg();
int      DevSettings_M::SetInitialPinValueUsbcfg(int newInitialPinValue);

//          Device Operations/Functions          //

```

```

int      DevFunctions_M::ResetDevice();
int      DevFunctions_M::GetAdcData(array<WORD>^ adcData);
int      DevFunctions_M::ClearInterruptPinState();
int      DevFunctions_M::ReadGpioPinValue(BYTE pinNumber);
int      DevFunctions_M::WriteGpioPinValue(BYTE pinNumber, BYTE pinValue);
int      DevFunctions_M::EnterAccessPassword(String^ accessPassword);

//          I2C OPERATIONS          //
int      DevFunctions_M::WriteI2cData(BYTE i2cAddress, array<System::Byte>^ i2cDataToSend, UINT32
        numberOfBytesToWrite, UINT32 i2cBusSpeed);
int      DevFunctions_M::ReadI2cData(BYTE i2cAddress, array<System::Byte>^ i2cDataReceived, UINT32
        numberOfBytesToRead, UINT32 i2cBusSpeed);
int      DevFunctions_M::StopI2cDataTransfer();

//          SMBus OPERATIONS          //
int      DevFunctions_M::SmbWriteBlock(BYTE smbAddress, array<System::Byte>^ smbDataToSend, UINT32
        numberOfBytesToWrite, UINT32 smbSpeed, BYTE usesPEC);
int      DevFunctions_M::SmbReadBlock(BYTE smbAddress, array<System::Byte>^ smbDataToRead, UINT32
        numberOfBytesToRead, UINT32 smbSpeed, BYTE usesPEC, BYTE readRegIndex);

//          Multiple device management          //
int      DevManagement_M::GetDeviceCount()
int      DevManagement_M::GetSelectedDevNumber()
String^  DevManagement_M::GetSelectedDevInfo()
int      DevManagement_M::SelectDev(int devNum)

```

### DllConstants:

```

//The two constants below are the same - user chooses which is more intuitive
static const int OFF = 0;
static const int DISABLED = 0;
//The two constants below are the same - user chooses which is more intuitive
static const int ON = 1;
static const int ENABLED = 1;
//Constants to be used for all whichTo(Get/Set) variables in DLL functions
static const int CURRENT_SETTINGS_ONLY = 0;
static const int PWRUP_DEFAULTS_ONLY = 1;
static const int BOTH = 2;

```

## DLL Error Codes

Nearly every function within the DLL will return an error code should something go wrong in the operation. Use this table to decipher what went wrong and what action to take in order to resolve the error.

Error Code	Error Description	Details and Recommended Resolution
<b>3</b>	Command not allowed	The flash is either locked or password protected. If password protected, send correct access password to unlock flash for editing.
<b>0</b>	No error (Success)	N/A
<b>-1</b>	Board not found	Check connection and device enumeration
<b>-2</b>	Wrong device ID	Ensure DLL was initialized properly
<b>-3</b>	Reading the device failed	Ensure DLL was initialized properly
<b>-4</b>	Device write failed	
<b>-5</b>	Device read failed	
<b>-10</b>	GP pin not configured as GPIO	Configure GP pin as GPIO and try operation again.
<b>-11</b>	I2C Slave data NACK received	
<b>-12</b>	Wrong PEC	
<b>-13</b>	Flash locked	
<b>-14</b>	Password attempt limit reached	
<b>-15</b>	Invalid state	
<b>-16</b>	Invalid data length	
<b>-17</b>	Error copying memory	

<b>-18</b>	Timeout	
<b>-19</b>	I2C send error	
<b>-20</b>	Error setting I2C address	
<b>-21</b>	Error setting I2C speed	
<b>-22</b>	Invalid I2C status	
<b>-23</b>	Address NACK received	
<b>-201</b>	Invalid parameter given (1 <sup>st</sup> parameter)	1 <sup>st</sup> parameter was invalid, verify parameters
<b>-202</b>	Invalid parameter given (2 <sup>nd</sup> parameter)	2 <sup>nd</sup> parameter was invalid, verify parameters
<b>-203</b>	Invalid parameter given (3 <sup>rd</sup> parameter)	3 <sup>rd</sup> parameter was invalid, verify parameters
<b>-204</b>	Invalid parameter given (4 <sup>th</sup> parameter)	4 <sup>th</sup> parameter was invalid, verify parameters
<b>-205</b>	Invalid parameter given (5 <sup>th</sup> parameter)	5 <sup>th</sup> parameter was invalid, verify parameters
<b>-206</b>	Invalid parameter given (6 <sup>th</sup> parameter)	6 <sup>th</sup> parameter was invalid, verify parameters
<b>-207</b>	Invalid parameter given (7 <sup>th</sup> parameter)	7 <sup>th</sup> parameter was invalid, verify parameters
<b>-208</b>	Invalid parameter given (8 <sup>th</sup> parameter)	8 <sup>th</sup> parameter was invalid, verify parameters
<b>-209</b>	Invalid parameter given (9 <sup>th</sup> parameter)	9 <sup>th</sup> parameter was invalid, verify parameters



## Detailed API Function List

Below you will find detailed descriptions of all the functions found in the DLL. Unless otherwise stated, the function is identical when used in either the managed or unmanaged DLL's.

### How to read detailed function description:

Each function will be described in the same format. The example below will help

Example: *int ExtractCharacter (char[] arg1, int arg2, char arg2)*

#### FunctionName

##### Purpose:

This function is an example of how function descriptions are structured.

##### Parameters:

[Input] arg1 (char[]) - String sent as input to function

[Input] arg2 (int) - Index of the string to extract character from

[Output] arg3 (char) - Character that was extracted from the given string and index number

##### Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

##### Notes:

None

## DllInit

### Purpose:

Sets the Vendor and Product ID used for the project. THIS MUST BE DONE IN ORDER TO BEGIN USING THE DLL!

### Parameters:

none

### Returns:

none

### Notes:

Call this function (when using UNMANAGED DLL only) before any other function calls! This will initialize the DLL using the default Microchip VID/PID values.

---

## DllInitCustom

### Purpose:

Sets the Vendor and Product ID used for the project.

### Parameters:

#### Inputs:

vid - Assigned by USB IF ([www.usb.org](http://www.usb.org))

pid - Assigned by the Vendor ID Holder

### Returns:

int - Contains error code. 0 = successful. Other = failed

### Notes:

Call this function (when using UNMANAGED DLL only) before any other function calls! This will initialize the DLL using the specified VID/PID values.

---

## GetUsbVid

Purpose:

Get the USB vendor ID of the device

Parameters:

none

Returns:

INT64 - If successful, returns value of USB VID. A value less than 0 indicates an error.

Notes:

none

---

## GetUsbPid

Purpose:

Get the USB product ID of the device

Parameters:

none

Returns:

INT64 - If successful, returns value of USB PID. A value less than 0 indicates an error.

Notes:

none

---

## SetUsbVidPid

Purpose:

Set the VID and PID of the part.

Parameters:

Inputs:

newVid (UINT32) - VID value to set

newPid (UINT32) - PID value to set

Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

Notes:

The new VID/PID values will not take effect until the device is power cycled.

---

## GetUsbPowerSource

### Purpose:

Gets the USB power source (host or self).

### Parameters:

none

### Returns:

int - If successful, returns value of device power source (0 = bus-powered, 1 = Self-powered). A value less than 0 indicates an error.

### Notes:

If response is 0, device is powered by host. If response is 1, device is self-powered.

---

## SetUsbPowerSource

### Purpose:

Sets the USB power source (bus-powered or self-powered).

### Parameters:

#### Inputs:

newPowerSource (int) - Value to indicate the power source. (0 = bus-powered, 1 = self-powered)

### Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

### Notes:

Send a value of 0 to indicate device is bus-powered or a value of 1 if device is self-powered.

---

## GetUsbRemoteWakeupEnable

### Purpose:

Get the enable state for the remote-wakeup feature

Parameters:

none

Returns:

int - If successful, returns 0 if disabled or 1 if enabled. A value less than 0 indicates an error.

Notes:

none.

---

## SetUsbRemoteWakeupEnable

Purpose:

Set the device to be remote wakeup capable or not

Parameters:

Inputs:

remoteWakeupEnable (bool) - Enable this feature (true) or disable this feature (false)

Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

Notes:

none

---

## GetUsbCurrentRequirement

Purpose:

Gets the amount of current (mA) that the device will request from the USB bus

Parameters:

none

Returns:

int - If successful, returns value of USB current requested (in mA). A value less than 0 indicates an error.

Notes:

The value returned has the units of mA.

---

## SetUsbCurrentRequirement

### Purpose:

Sets the amount of current (mA) that the device will request from the USB bus

### Parameters:

#### Inputs:

newCurrentValue (int) - New value to set for the USB current (in mA)

### Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

### Notes:

The input into this function must be the desired value in mA.

---

## GetUsbStringManufacturer

### Purpose:

Get the manufacturer USB string from the device.

### MANAGED API:

#### Parameters:

##### Inputs:

none

#### Returns:

String^ - Unicode manufacturer USB string output from the device

### UNMANAGED API:

#### Parameters:

##### Inputs:

none

##### Outputs:

usbStringManufacturer (wchar\_t \*) - Unicode manufacturer USB string output from the device

#### Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

### Notes:

The string returned can be a maximum length of 30 characters.

---

## SetUsbStringManufacturer

### Purpose:

Set the manufacturer USB string for the device.

### MANAGED API:

#### Parameters:

##### Inputs:

String^ - Unicode manufacturer USB string output from the device

##### Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

### UNMANAGED API:

#### Parameters:

##### Inputs:

usbStringManufacturer (wchar\_t \*) - Unicode manufacturer USB string output from the device

##### Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

### Parameters:

#### Inputs:

##### Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

### Notes:

The string can have a maximum length of 30 characters. The PC operating system will see the change on the next device connection.

---

## GetUsbStringDescriptor

### Purpose:

Get the USB string descriptor from the device.

### MANAGED API:

#### Parameters:

none  
Returns:  
        usbStringManufacturer (String^) - Unicode descriptor USB string output from the device  
UNMANAGED API:  
Parameters:  
    Inputs:  
        none  
    Outputs:  
        usbStringManufacturer (wchar\_t \*) - Unicode manufacturer USB string output from the device  
Returns:  
    int - If successful, returns 0. A value less than 0 indicates an error.  
Notes:  
    The string returned can be a maximum length of 30 characters.

---

## SetUsbStringDescriptor

Purpose:  
    Set the manufacturer USB string for the device.  
MANAGED API:  
Parameters:  
    Inputs:  
        usbStringManufacturer (String^) - Unicode manufacturer USB string output from the device  
Returns:  
    int - If successful, returns 0. A value less than 0 indicates an error.  
UNMANAGED API:  
Parameters:  
    Inputs:  
        usbStringManufacturer (wchar\_t \*) - Unicode manufacturer USB string output from the device  
Returns:  
    int - If successful, returns 0. A value less than 0 indicates an error.  
Notes:  
    The string can have a maximum length of 30 characters. The PC operating system will see the change on the next device connection.



---

## GetSerialNumber

### Purpose:

Get the device serial number.

### MANAGED API:

#### Parameters:

none

#### Returns:

String^ - If successful, Unicode serial number string is returned. Otherwise blank string is returned.

### UNMANAGED API:

#### Parameters:

##### Inputs:

none

##### Outputs:

serialNumber (wchar\_t\*) - Unicode serial number string

#### Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

### Notes:

This value can be different than the factory serial number, which cannot be changed.

---

## SetSerialNumber

### Purpose:

Set the device serial number.

### MANAGED API:

#### Parameters:

##### Inputs:

serialNumber (String^) - Unicode serial number string to set.

#### Returns:

If successful, returns 0. A value less than 0 indicates an error.

UNMANAGED API:

Parameters:

Inputs:

serialNumber (wchar\_t\*) - String that holds the serial number

Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

Notes:

None

---

## GetFactorySerialNumber

Purpose:

Get the device factory serial number.

MANAGED API:

Parameters:

none

Returns:

String^ - If successful, Unicode factory serial number string is returned. Otherwise blank string is returned.

UNMANAGED API:

Parameters:

Inputs:

none

Outputs:

serialNumber (wchar\_t\*) - Unicode serial number string

Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

Notes:

This value is set at the factory during manufacturing - it can't be changed.

---

## GetSerialNumberEnumerationEnable

**Purpose:**

Get the current enable setting for whether or not the device will use its serial number during CDC enumeration.

**Parameters:**

none

**Returns:**

int - Returns 0 if serial number enumeration is disabled or 1 if it is enabled. A value less than 0 indicates an error.

**Notes:**

none

---

## **SetSerialNumberEnumerationEnable**

**Purpose:**

Set the enable setting for whether or not the device will use its serial number during CDC enumeration.

**Parameters:**

**Inputs:**

enableSerialNumberEnumeration (bool) - Whether or not to show serial number to OS during enumeration.

**Returns:**

int - If successful, returns 0. A value less than 0 indicates an error.

**Notes:**

true = the device WILL provide its serial number to the OS during CDC enumeration

false = the device WILL NOT provide its serial number to the OS during CDC enumeration

---

## **GetFirmwareVersion**

**Purpose:**

Get the device firmware version information.

**MANAGED API:**

**Parameters:**

none

**Returns:**

String^ - If successful, returns firmware version string. If error, blank string is returned.

UNMANAGED API:

Parameters:

Outputs:

firmwareVersion (char \*) - Character array of length 3 to hold the firmware version string ([major].[minor])

Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

Notes:

none

---

## GetHardwareVersion

Purpose:

Get the device hardware version information.

MANAGED API:

Parameters:

none

Returns:

String^ - If successful, returns hardware version string. If error, blank string is returned.

UNMANAGED API:

Parameters:

Outputs:

hardwareVersion (char \*) - Character array of length 2 to hold the hardware revision string ([major][minor])

Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

Notes:

none

---

## GetConnectionStatus

**Purpose:**

Retrieve the connection status of the device.

**Parameters:**

none

**Returns:**

bool - If connected, returns true. Else, returns false.

**Notes:**

This function will return the connection status of the selected device.

---

## GetFlashProtectionState

**Purpose:**

Get the state of flash protection for the device

**Parameters:**

none

**Returns:**

int - If successful, returns protection state (see below). A value less than 0 indicates an error.

**Notes:**

0 = unsecured

1 = password protection is enabled

2 = permanently locked (cannot be undone)

---

## SetFlashProtectionOff

**Purpose:**

Disable flash password protection for the device

**Parameters:**

none

**Returns:**

int - If successful, returns 0. A value less than 0 indicates an error.

Notes:

Device must be unlocked for this function to complete properly. Use the "SendAccessPassword()" function first to unlock the device.

---

## SetFlashPermanentLock

Purpose:

Permanently lock the device flash settings -- this action CAN'T be undone.

Parameters:

none

Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

Notes:

!!! WARNING !!! -- USE THIS FUNCTION WITH GREAT CAUTION. THE CHIP FLASH SETTINGS (boot-up defaults) CANNOT BE CONFIGURED AFTER THIS FUNCTION HAS BEEN INVOKED!!

---

## SetFlashPasswordProtection

Purpose:

Enable the password protection with the supplied password

MANAGED API:

Parameters:

Inputs:

accessPassword (String^) - Password to use for flash protection

Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

UNMANAGED API:

Parameters:

Inputs:

accessPassword (char\*) - Password to use for flash protection

Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

Notes:

The password MUST be 8 ASCII characters in length.

---

## GetInterruptPinMode

Purpose:

Get the interrupt mode for the interrupt pin.

Parameters:

Inputs:

whichToGet (int) - Current setting = 0, Power-up default = 1

Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

Notes:

none

---

## SetInterruptPinMode

Purpose:

Set the interrupt mode for the interrupt pin

Parameters:

Inputs:

whichToSet (int) - Current setting = 0, Power-up default = 1, Both = 2

interruptPinModeToSet (int) - Valid options: 1 = rising edges, 2 = falling edges, 3 = both

Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

Notes:

none

---

## GetClockPinDividerValue

### Purpose:

Get the clock pin divider value, which ranges between 1 and 31.

### Parameters:

#### Inputs:

whichToGet (int) - Current setting = 0, Power-up default = 1

### Returns:

int - If successful, returns value of clock divider (ranges from 1 - 31). A value less than 0 indicates an error.

### Notes:

This value is the exponent in the clock divider calculation:  $(2^n)$  (Ex. Value 2 = clock divider is 4, 3 = 8, etc). See the part datasheet for more details.

---

## GetClockPinDutyCycle

### Purpose:

Get the clock pin duty cycle.

### Parameters:

#### Inputs:

whichToGet (int) - Current setting = 0, Power-up default = 1

### Returns:

int - If successful, returns value of clock duty cycle (ranges from 0 - 3). A value less than 0 indicates an error.

### Notes:

3 = 75%, 2 = 50%, 1 = 25%, 0 = 0%

---

## SetClockPinConfiguration

### Purpose:

Set the clock pin divider value and duty cycle.



Parameters:

Inputs:

whichToSet (int) - Current setting = 0, Power-up default = 1  
clockDividerValueToSet (int) - Value to use for the clock divider.  
dutyCycleToSet (int) - Value indicating the clock duty cycle: 3 = 75%, 2 = 50%, 1 = 25%, 0 = 0%

Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

Notes:

Possible value for clock divider ranges from 1 to 7.  
Possible values for clock duty cycle are 0 to 3.

---

## GetGpConfiguration

Purpose:

Get the device GP pin configuration, which includes the pin function designation, direction, and latch value.

MANAGED API:

Parameters:

Inputs:

whichToGet (int) - Use constants defined in this class. Current setting = 0, Power-up default = 1

Outputs:

gpPinDesignations (array<System::Byte>^) - An array (length of 4) specifying each pin designation  
gpPinDirections (array<System::Byte>^) - An array (length of 4) specifying each GPIO pin direction  
gpPinValues (array<System::Byte>^) - An array (length of 4) specifying each GPIO pin value

Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

UNMANAGED API:

Parameters:

Inputs:

whichToGet (int) - Use constants defined in this class. Current setting = 0, Power-up default = 1

Outputs:

gpPinDesignations (BYTE \*) - An array (length of 4) specifying each pin designation  
gpPinDirections (BYTE \*) - An array (length of 4) specifying each GPIO pin direction  
gpPinValues (BYTE \*) - An array (length of 4) specifying each GPIO pin value

Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

Notes:

Possible pin designations: 0 = GPIO, 1 = dedicated function, 2 = alternate 1, 3 = alternate 2, 4 = alternate 3

GPIO Pin directions: 0 = output, 1 = input

GPIO Pin Output Latches: 0 = logical low, any other value = logical high

---

## SetGpPinConfiguration

Purpose:

Set up the device GP pins with the desired function designation, direction, and latch values.

MANAGED API:

Parameters:

Inputs:

whichToSet (int) - Use constants defined in this class. Current setting = 0, Power-up default = 1

gpPinDesignations (array<System::Byte>^) - An array (length of 4) specifying each pin designation

gpPinDirections (array<System::Byte>^) - An array (length of 4) specifying each GPIO pin direction

gpPinValues (array<System::Byte>^) - An array (length of 4) specifying each GPIO pin value

Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

UNMANAGED API:

Parameters:

Inputs:

whichToSet (int) - Use constants defined in this class. Current setting = 0, Power-up default = 1

gpPinDesignations (BYTE \*) - An array (length of 4) specifying each pin designation

gpPinDirections (BYTE \*) - An array (length of 4) specifying each GPIO pin direction

gpPinValues (BYTE \*) - An array (length of 4) specifying each GPIO pin value

Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

Notes:

Possible pin designations: 0 = GPIO, 1 = dedicated function, 2 = alternate 1, 3 = alternate 2, 4 = alternate 3

GPIO Pin directions: 0 = output, 1 = input

GPIO Pin Output Latches: 0 = logical low, any other value = logical high

---

## GetGpPinDirection

### Purpose:

Get the current direction of the specified pin.

### Parameters:

#### Inputs:

whichToGet (int) - Current setting = 0, Power-up default = 1  
pinNumber (BYTE) - The pin number of the pin to get the value of.

### Returns:

int - If successful, returns pin direction (0 = output, 1 = input). A value less than 0 indicates an error.

### Notes:

The direction (input/output) only matters if the pin is designated as a GPIO.

---

## SetGpPinDirection

### Purpose:

Set the current direction of the specified pin.

### Parameters:

#### Inputs:

whichToSet (int) - Current setting = 0, Power-up default = 1, Both = 2  
pinNumber (BYTE) - The pin number of the pin to get the value of (0, 1, 2, or 3 are valid values)  
pinDirection (BYTE) - The pin direction to set on the specified pin (0 = output or 1 = input).

### Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

### Notes:

The direction (input/output) only matters if the pin is designated as a GPIO.

---

## GetDacVoltageReference

**Purpose:**

Get the voltage reference value for the DAC.

**Parameters:**

none

**Returns:**

int - If successful, returns one of the values below. A negative value indicates an error.

0 = Vdd (default)

1 = 1.024V

2 = 2.048V

3 = 4.096V

**Notes:**

none

---

## SetDacVoltageReference

**Purpose:**

Set the voltage reference value for the DAC.

**Parameters:**

**Inputs:**

whichToSet (int) - Current setting = 0, Power-up default = 1, Both = 2

vRefValue (int) - This value indicates to what VRef should be set. Use key below to set desired value:

0 = Vdd (default)

1 = 1.024V

2 = 2.048V

3 = 4.096V

**Returns:**

int - If successful, returns 0. A value less than 0 indicates an error.

**Notes:**

none

---

## GetDacValue

**Purpose:**

Get the DAC value.

**Parameters:**

none

**Returns:**

int - If successful, returns value of the DAC (ranges between 0 and 31). A negative value indicates an error.

**Notes:**

none

---

## SetDacValue

**Purpose:**

Set the DAC value.

**Parameters:**

**Inputs:**

whichToSet (int) - Current setting = 0, Power-up default = 1, Both = 2

dacValue (int) - This value can range between 0 and 31.

**Returns:**

int - If successful, returns 0. A value less than 0 indicates an error.

**Notes:**

none

---

## GetAdcVoltageReference

**Purpose:**

Get the voltage reference value for the ADC.

**Parameters:**

none

**Returns:**

int - If successful, returns 0. A value less than 0 indicates an error.

0 = Vdd (default)

1 = 1.024V  
2 = 2.048V  
3 = 4.096V

Notes:

none

---

## SetAdcVoltageReference

Purpose:

Set the voltage reference value for the ADC.

Parameters:

Inputs:

whichToSet (int) - Current setting = 0, Power-up default = 1, Both = 2

vRefValue (int) - This value indicates to what VRef should be set. Use key below to set desired value:

0 = Vdd (default)

1 = 1.024V

2 = 2.048V

3 = 4.096V

Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

Notes:

none

---

## GetInitialPinValueLedUartTx

Purpose:

Get the initial pin value for LEDUARTTX pin.

Parameters:

none

Returns:

int - If successful, returns initial value of LEDUARTTX pin. A value less than 0 indicates an error.

**Notes:**

This value represents the logic level signaled when no UART TX transmission takes place. When the UART TX (of the MCP2221) is sending data, the LEDUARTTX pin will take the negated value of this bit.

---

## **SetInitialPinValueLedUartTx**

**Purpose:**

Set the initial pin value for LEDUARTTX pin.

**Parameters:**

**Inputs:**

newInitialPinValue (int) - New initial pin value to set

**Returns:**

int - If successful, returns 0. A value less than 0 indicates an error.

**Notes:**

This value represents the logic level signaled when no UART TX transmission takes place. When the UART TX (of the MCP2221) is sending data, the LEDUARTTX pin will take the negated value of this bit.

---

## **GetInitialPinValueLedUartRx**

**Purpose:**

Get the initial pin value for LEDUARTRX pin.

**Parameters:**

none

**Returns:**

int - If successful, returns initial value of LEDUARTRX pin. A value less than 0 indicates an error.

**Notes:**

This value represents the logic level signaled when no UART RX activity takes places. When the UART RX (of the MCP2221) is receiving data, the LEDUARTRX pin will take the negated value of this bit.

## SetInitialPinValueLedUartRx

### Purpose:

Set the initial pin value for LEDUARTX pin.

### Parameters:

#### Inputs:

newInitialPinValue (int) - New initial pin value to set

### Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

### Notes:

This value represents the logic level signaled when no UART RX activity takes places. When the UART RX (of the MCP2221) is receiving data, the LEDUARTX pin will take the negated value of this bit.

---

## GetInitialPinValueLedI2c

### Purpose:

Get the initial pin value for LEDI2C pin.

### Parameters:

none

### Returns:

int - If successful, returns initial value of LEDI2C pin. A value less than 0 indicates an error.

### Notes:

This value represents the logic level signaled when no I2C traffic occurs. When I2C traffic is active, the LEDI2C pin (if enabled) will take the negated value of this bit.

---

## SetInitialPinValueLedI2c

### Purpose:

Set the initial pin value for LEDI2C pin.

### Parameters:

#### Inputs:



`newInitialPinValue (int)` - New initial pin value to set

Returns:

`int` - If successful, returns 0. A value less than 0 indicates an error.

Notes:

This value represents the logic level signaled when no I2C traffic occurs. When I2C traffic is active, the LEDI2C pin (if enabled) will take the negated value of this bit.

---

## GetInitialPinValueSspnd

Purpose:

Get the initial pin value for SSPND pin.

Parameters:

none

Returns:

`int` - If successful, returns initial value of SSPND pin. A value less than 0 indicates an error.

Notes:

This value represents the logic level signaled when the chip is not in suspend mode. Upon entering suspend mode, the SSPND pin (if enabled) will take the negated value of this bit.

---

## SetInitialPinValueSspnd

Purpose:

Set the initial pin value for SSPND pin.

Parameters:

Inputs:

`newInitialPinValue (int)` - New initial pin value to set

Returns:

`int` - If successful, returns 0. A value less than 0 indicates an error.

Notes:

This value represents the logic level signaled when the chip is not in suspend mode. Upon entering suspend mode, the SSPND pin (if enabled) will take the negated value of this bit.

---

## GetInitialPinValueUsbcfg

### Purpose:

Get the initial pin value for USBCFG pin.

### Parameters:

none

### Returns:

int - If successful, returns initial value of USBCFG pin. A value less than 0 indicates an error.

### Notes:

This value represents the logic level signaled when the chip is not USB configured. When the chip will be USB configured, the USBCFG pin (if enabled) will take the negated value of this bit.

---

## SetInitialPinValueUsbcfg

### Purpose:

Set the initial pin value for USBCFG pin.

### Parameters:

#### Inputs:

newInitialPinValue (int) - New initial pin value to set

### Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

### Notes:

This value represents the logic level signaled when the chip is not USB configured. When the chip will be USB configured, the USBCFG pin (if enabled) will take the negated value of this bit.

---

## ResetDevice

### Purpose:

Perform a device reset.

Parameters:

none

Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

Notes:

none

---

## GetAdcData

Purpose:

Get the ADC data for all 3 channels.

MANAGED API:

Parameters:

Inputs:

none

Outputs:

array<unsigned short>^ adcData - Array of at least 3 16-bit values.

Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

UNMANAGED API:

Parameters:

Inputs:

none

Outputs:

adcData (WORD\*) - Array of at least 3 16-bit values.

Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

Notes:

The value at array index 0 is the value of ADC1, the value at index 1 is the value of ADC2, and so on.

## ClearInterruptPinState

**Purpose:**

Clear the current state of the interrupt pin.

**Parameters:**

none

**Returns:**

int - If successful, returns 0. A value less than 0 indicates an error.

**Notes:**

none

---

## ReadGpioPinValue

**Purpose:**

Get the current pin value of the specified pin.

**Parameters:**

**Inputs:**

pinNumber (BYTE) - The pin number of the pin to get the value of.

**Returns:**

int - If successful, returns pin value. A value less than 0 indicates an error.

**Notes:**

none

---

## WriteGpioPinValue

**Purpose:**

Set the specified pin value to the specified pin value.

**Parameters:**

**Inputs:**

pinNumber (BYTE) - The pin number of the pin to get the value of (0, 1, 2, or 3 are valid values)

pinValue (BYTE) - The pin value to set on the specified pin (0 or 1).

Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

Notes:

none

---

## EnterAccessPassword

Purpose:

Send the access password to the chip. The proper password unlocks the device for modifying boot-up settings.

MANAGED API:

Parameters:

Inputs:

accessPassword (String^) - Password string to send to chip.

Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

UNMANAGED API:

Parameters:

Inputs:

accessPassword (char\*) - Password string to send to chip.

Returns:

int - If successful, returns 0. Value of 3 is returned if command wasn't allowed (see below for details)

Notes:

A valid password must be sent within first 3 attempts. Otherwise device will block future attempts and return the value 3 (until reset).

---

## WriteI2cData

Purpose:

Write I2C data to the specified address.

MANAGED API:

Parameters:

[Input] i2cAddress (BYTE) - The I2C slave address of the device from which we wish to receive the data  
[Output] dataToSend (array<System::Byte>^) - Array of bytes holding data to send  
[Input] transferLength (UINT32) - The number of bytes we want to read from the I2C Slave chip  
[Input] i2cBusSpeed (UINT32) - The I2C communication speed

Returns:

int - If successful, returns 0. A value less than 0 indicates an error (write failed).

UNMANAGED API:

Parameters:

[Input] i2cAddress (BYTE) - The I2C slave address of the device to which we wish to send the I2C data  
[Output] dataToSend (BYTE\*) - Array of bytes holding data to send  
[Input] transferLength (UINT32) - The length of dataToSend array  
[Input] i2cBusSpeed (UINT32) - The I2C communication speed

Returns:

int - If successful, returns 0. A value less than 0 indicates an error (write failed).

Notes:

none

---

## ReadI2cData

Purpose:

Read data from the I2C device at the specified address.

MANAGED API:

Parameters:

[Input] i2cAddress (BYTE) - The I2C slave address of the device from which we wish to receive the data  
[Output] i2cDataReceived (array<System::Byte>^) - The data that was read from the I2C Slave chip  
[Input] numberOfBytesToRead (UINT32) - The number of bytes we want to read from the I2C Slave chip  
[Input] i2cBusSpeed (UINT32) - The I2C communication speed

Returns:

int - If successful, returns 0. A value less than 0 indicates an error (read failed).

UNMANAGED API:

Parameters:

[Input] i2cAddress (BYTE) - The I2C slave address of the device from which we wish to receive the data

[Output] i2cDataReceived (BYTE\*) - The data that was read from the I2C Slave chip  
[Input] numberOfBytesToRead (UINT32) - The number of bytes we want to read from the I2C Slave chip  
[Input] i2cBusSpeed (UINT32) - The I2C communication speed

Returns:

int - If successful, returns 0. A value less than 0 indicates an error (read failed).

Notes:

none

---

## SmbWriteBlock

Purpose:

Send the SMB write block command and the given user data.

MANAGED API:

Parameters:

[Input] smbAddress (BYTE) - the I2C/SMB address of the slave we want to write data to  
[Output] smbDataToSend (array<System::Byte>^) - data to send to the I2C/SMB device  
[Input] numberOfBytesToWrite (UINT32) - data transfer length  
[Input] smbSpeed (UINT32) - the communication speed used  
[Input] usesPEC (BYTE) - use PEC or not

Returns:

int - If successful, returns 0. A value less than 0 indicates an error (write failed).

UNMANAGED API:

Parameters:

[Input] smbAddress (BYTE) - the I2C/SMB address of the slave we want to write data to  
[Output] smbDataToSend (BYTE\*) - data to send to the I2C/SMB device  
[Input] numberOfBytesToWrite (UINT32) - data transfer length  
[Input] smbSpeed (UINT32) - the communication speed used  
[Input] usesPEC (BYTE) - use PEC or not

Returns:

int - If successful, returns 0. A value less than 0 indicates an error (write failed).

Notes:

none

---

## SmbReadBlock

### Purpose:

Send the SMB read block command and get the data

### MANAGED API:

#### Parameters:

[Input] smbAddress (BYTE) - The I2C/SMB address of the slave we want to read data from  
[Output] smbDataToRead (array<System::Byte>^) - Data to read from the I2C/SMB device  
[Input] numberOfBytesToRead (UINT32) - Number of bytes to read  
[Input] smbSpeed (UINT32) - The communication speed used  
[Input] usesPEC (BYTE) - Use PEC or not  
[Input] readRegIndex (UINT32) - the register index (as per SMB specs) we will use to read data from

#### Returns:

int - If successful, returns 0. A value less than 0 indicates an error (read failed).

### UNMANAGED API:

#### Parameters:

[Input] smbAddress (BYTE) - The I2C/SMB address of the slave we want to read data from  
[Output] smbDataToRead (BYTE\*) - Data to read from the I2C/SMB device  
[Input] numberOfBytesToRead (UINT32) - Number of bytes to read  
[Input] smbSpeed (UINT32) - The communication speed used  
[Input] usesPEC (BYTE) - Use PEC or not  
[Input] readRegIndex (UINT32) - the register index (as per SMB specs) we will use to read data from

#### Returns:

int - If successful, returns 0. A value less than 0 indicates an error (read failed).

### Notes:

none

---

## StopI2cDataTransfer

### Purpose:

Stop any current I2C data transfers.



Parameters:

none

Returns:

int - If successful, returns 0. A value less than 0 indicates an error.

Notes:

none

---

## GetDevCount

Purpose:

Gets the total number of attached devices

Parameters:

none

Returns:

int - The total number of attached devices.

Notes:

\*\*\*IMPORTANT: You MUST use the GetConnectionStatus() function prior to calling this function since the device count is refreshed by doing so.\*\*\*

---

## GetSelectedDevNumber

Purpose:

Gets the unique index number that indicates which MCP2221 device is selected.

Parameters:

none

Returns:

int - The unique index number identifying the selected MCP2221 device.

Notes:

Numbering of devices starts with 0. Hence, the first device will be indicated by the number 0 and the 2<sup>nd</sup> device by 1 and so on.

---

## GetSelectedDevInfo

### Purpose:

Get the information for the currently selected device

### MANAGED API:

#### Parameters:

none

#### Returns:

String^ - String that gives information regarding the currently selected device

### UNMANAGED API:

#### Parameters:

##### Inputs:

none

##### Outputs:

outputString (char\*) - String that gives information regarding the currently selected device

#### Returns:

int - If successful, returns 0. A value less than 0 indicates an error (read failed).

### Notes:

In order to uniquely identify each device from another, you can use this string and/or the device serial number. This string is retrieved from the operating system and contains device path information.

---

## SelectDev

### Purpose:

Select the device to which the DLL will communicate

### Parameters:

#### Inputs:

devNum (int) - The device number to be selected

### Returns:

int - Contains error code. 0 = successful. Other = failed

### Notes:

Numbering of devices starts with 0. Hence, the first device will be indicated by the number 0 and the 2<sup>nd</sup> device by 1 and so on. Be sure to get the total count of devices available before using this function in order to be sure that you are selecting a valid device number (max valid number is devCount-1). \*\*\*IMPORTANT: You MUST use

the GetConnectionStatus() function after switching your selected device to allow proper operation and manipulation of that device.\*\*

---