USB-to-I2C
Ultra

I²C/SMBus and SPI Control DLL

User's Manual

Date: July 1, 2018

Microsoft Visual Basic, Visual C++, Visual C#, and Windows are registered trademarks of Microsoft Corporation.

Delphi and C++ Builder are trademarks of Embarcadero Technologies, Inc.

Other brand names are trademarks or registered trademarks of their respective owners.

Questions or comments regarding this document should be emailed to: support@i2ctools.com.

Suggestions for enhancements can be emailed to: support@i2ctools.com.

Table of Contents

I²C Protocol

General Characteristics

The I²C protocol allows data to be transferred between devices using two open-drain (or open-collector) bi-directional lines. One line is the serial clock (SCL) and the other is the serial data (SDA). The bus master generates the Start conditions, the clock signals on SCL, as well as the Stop condition. An acknowledge (ACK) is transmitted by the receiving device on the bus after each byte is sent.

Bit Transfer

Data on SDA must be stable while SCL is high. The state of SDA when SCL is high determines the logic level of the transmitted data bit.

Start and Stop Conditions

Within the procedure of the I²C bus, unique situations arise which are defined as START and STOP conditions. A HIGH to LOW transition on the SDA line while SCL is HIGH is one such unique case. This situation indicates a START condition. A LOW to HIGH transition on the SDA line while SCL is HIGH defines a STOP condition. The master always generates START and STOP conditions. The bus is considered to be busy after the START condition. The bus is considered to be free again a certain time after the STOP condition.
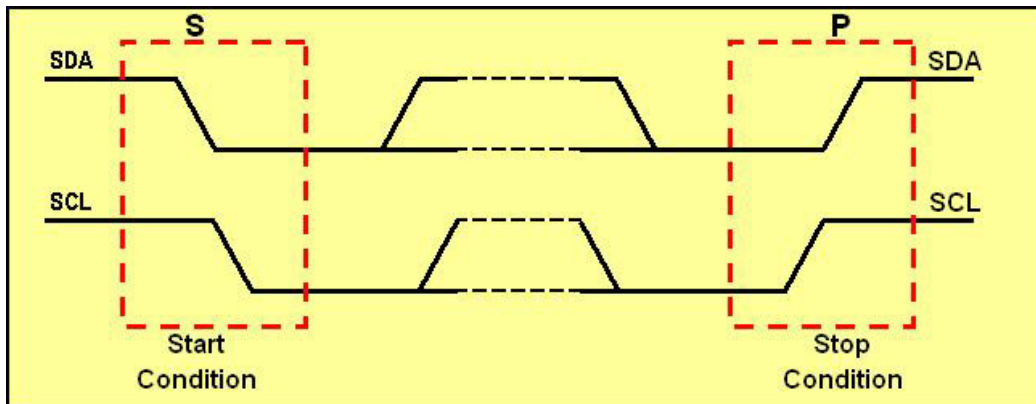
I²C Address

The first seven bits of an I²C transmission, after a Start condition, make up the slave address. The eighth bit (or the least significant bit) is the R/W bit that determines the direction of the message.
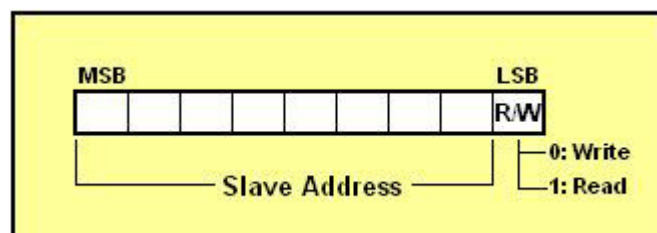
A '0' in the least significant position of the first byte means that the master will WRITE information to the selected slave. A '1' in this position means that the master will READ information from the slave.

When an I^2C address is sent, each device in a system compares the first seven bits after the START condition with its own address. If they match, the device considers itself addressed by the master as a slave-receiver or slave-transmitter, depending on the R/W bit.

Subaddress

When an I^2C device contains more than one register, the various registers are generally accessed using a subaddress that is sent following the device address (see the I2CWriteArray and I2CReadArray sections below). The subaddress acts like a pointer to the register that needs to be accessed.

Data Transfer

Every byte on the SDA line must be 8 bits long. The number of bytes that can be transmitted per transfer is unrestricted. Each byte must also be followed by an acknowledge bit. Data is transferred with the most significant bit first. If a receiver can't receive another complete byte of data until it has performed some other function, it can hold the clock line SCL low to force the transmitter into a wait state.

Acknowledge

The Acknowledge related clock pulse is generated by the master (USB-to-I2C Ultra is always the bus master). The transmitter releases the SDA line during the acknowledge clock pulse. The receiver must pull down the SDA line during the acknowledge clock pulse so that it remains stable low during the high period of the clock pulse.

The master-receiver signals the end of a read by not acknowledging the last byte it requires.

I²C Bus Documentation

The complete I^2C Bus specification can be found on the NXP Semiconductors website.

Minimum System Configuration

- ✓ PC with 8MB RAM or better
- ✓ 64-bit versions of Windows XP, Vista, Windows 7, Windows 8, and Windows 10
- ✓ 20 MB of free HDD space
- ✓ Internet access for software download
- ✓ USB port

USB-to-I2C Ultra Contents

- USB-to-I2C Ultra installation file
- USB-to-I2C Ultra Hardware
- USB-to-I2C Level Shifting Hardware

Files installed for USB-to-I2C Ultra

- USBtoI2Cpro.dll - this is the actual DLL file you will link to your application. The installation process places this file in the appropriate Windows\SysWOW64 folder
- USB-to-I2C Ultra DLL User's Manual (this document)
- Visual C++, Delphi, C++ Builder, and Visual Basic example files
- USBtoI2C.exe application
- USB-to-SPI.exe application
- USB-to-I2C Software User's Manual
- USB-to-I2C - Getting Started
- USB-to-I2C Hardware User's Manual
- Software license agreement (license.txt)
- Registration Form (RegFile.txt)

Location of DLL

The USBtoI2C_Ultra.dll is placed in the Windows\SysWOW64 directory during installation.

Testing the Installation

After USB-to-I2C Ultra has been installed on your hard disk, the installation of the driver can be tested with the included USB-to-I2C application. The hardware should be inserted into an available USB port, and then the USB-to-I2C application can be started. If the installation was successful, you will see the "Hardware Detected" message on the screen.
Note that you <u>must have Administrator privileges,</u> or the USB drivers will not be loaded correctly. After the software has successfully been installed, normal user privileges can be restored.

Exported Functions using the stdcall convention

Most programming languages, such as Visual C++, Visual C#, Delphi, C++ Builder, Visual Basic, and LabView can use the **stdcall** calling convention. The stdcall convention passes the parameters to the functions in the DLL from right to left and it is up to the called functions (in this case, the functions in USBtoI2C_Ultra.dll) to clean up the stack.

General Control Functions

Enable3VOutputPower

The USB-to-I2C hardware contains a 3.3V output which can be used to supply power to a low-power target. The USB-to-I2C hardware enumerates as a low power device which means that it draws less than 100mA. The combined current consumption of the target system and the USB-to-I2C hardware should therefore consume less than 100mA in order to meet this requirement.
The function takes a Boolean false ('0') to disable the output while a true (non-zero value) will enable the output. The function returns true if enabled and false if disabled.
The default value is false (disabled).

```
C/C++:    short int Enable3VOutputPower(short int State);
Delphi:   Enable3VoutputPower(State: LongBool): LongBool;
VB:       Enable3VOutputPower(ByVal State As Boolean) As Boolean
```

Enable5VOutputPower

The USB-to-I2C hardware contains a 5V output which can be used to supply power to a low-power target. The USB-to-I2C hardware enumerates as a low power device which means that it draws less than 100mA. The combined current consumption of the target system and the USB-to-I2C hardware should therefore consume less than 100mA in order to meet this requirement.
The function takes a Boolean false ('0') to disable the output while a true (non-zero value) will enable the output. The function returns true if enabled and false if disabled.
The default value is false (disabled).

```
C/C++:    short int Enable5VOutputPower(short int State);
Delphi:   Enable5VoutputPower(State: LongBool): LongBool;
VB:       Enable5VOutputPower(ByVal State As Boolean) As Boolean
```

GetFirmwareRevision

The firmware revision can be found by using the GetFirmwareRevision function. The revision is returned in BCD format. For example, a value of 0x12 would correspond to firmware version 1.2.

```
C/C++:    uchar GetFirmwareRevision(void);
Delphi:   GetFirmwareRevision: byte;
VB:       GetFirmwareRevision() As Byte
```

GetHardwareInfo

This function takes a pointer to an array of bytes which will be loaded with three values:
1. I2C frequency (note that the value returned is the I2C frequency divided by 2)
2. 3.3V Power Output State (0 = 'On' and 1='Off')
3. 5V Power Output State (0 = 'On' and 1='Off')

C/C++: `int GetHardwareInfo(uchar *HardwareData);`
Delphi: `GetHardwareInfo(var HardwareData: byte): integer;`
VB: `GetHardwareInfo(ByRef HardwareData As Byte) As Long`

GetNumberOfDevices

The GetNumberOfDevices function returns the number of USB-to-I2C adapters currently enumerated on the user's PC.

C/C++: `int GetNumberOfDevices(void);`
Delphi: `GetNumberOfDevices: integer;`
VB: `GetNumberOfDevices() As Long`

SelectBySerialNumber

USB-to-I2C allows you to communicate with a specific adapter based on its serial number. Just use the SelectBySerialNumber function to specify the serial number (labeled on every USB-to-I2C adapter) you would like to communicate with and then all communications will continue with this adapter until a new serial number is specified. This is a simple way to identify the USB adapter when multiple USB-to-I2C adapters are present in your system.
The function returns a '0' if the serial number is not found and a '1' if the serial number has been found.

C/C++: `int SelectBySerialNumber(int SerialNumber);`
Delphi: `SelectBySerialNumber(SerialNumber: integer): integer;`
VB: `SelectBySerialNumber(ByVal SerialNumber As Long) As Long`

GetSerialNumbers

This function takes a pointer to an array which will be loaded with the serial numbers of the USB-to-I2C adapters attached to the user's system. The function returns the number of USB-to-I2C adapters attached.

C/C++: `int GetSerialNumbers(int *SerialNumbers);`
Delphi: `GetSerialNumbers (var SerialNumbers: integer): integer;`
VB: `GetSerialNumbers (ByRef SerialNumbers As Long) As Long`

Get_DLL_Version

This function returns the version of the USBtoI2C_Ultra.dll file. Normally, it will not be necessary to call this function unless you need to know the specific version of the DLL is installed. We have found this function to be useful for development environments that do not automatically load the DLL into memory until a function is called (for example, console applications).

C/C++: `int Get_DLL_Version(void);`

Delphi: `Get_DLL_Version: `**`integer;`**
VB: `Get_DLL_Version () `**`As Long`**

ShutdownProcedure

This function should be called when the application using the DLL is closed.

C/C++:   **`void`** `ShutdownProcedure(`**`void`**`);`
Delphi: `ShutdownProcedure;`
VB: `ShutdownProcedure()`

IO Functions

There are three sources of general purpose IO available on the USB-to-I2C hardware:
1. There is a dedicated IN and a dedicated OUT pin. These are always available, even when both the I2C and SPI buses are in use
2. The I2C pins may be used as open-drain IO. The I2C functions must be disabled for these pins to be used as IO.
3. The SPI pins may be used as quasi-bidirectional IO. The SPI functions must be disabled for these pins to be used as IO.

Read_IO

This function returns the state of the general-purpose input (IN) port on the USB-to-I2C hardware. The function returns a '1' for a high logic level and a '0' for a low logic level.

```
C/C++:    int Read_IO(void);
Delphi:   Read_IO: integer;
VB:       Read_IO() As Long
```

Write_IO

This function writes a value to the general-purpose output (OUT) port on the USB-to-I2C hardware. The function takes a '1' for a high logic level and a '0' for a low logic level.

```
C/C++:    int Write_IO(int OutputState);
Delphi:   Write_IO(OutputState: integer): integer;
VB:       Write_IO(ByVal OutputState As Long) As Long
```

Disable_I2C

The Disable_I2C function disables the I2C functionality on the USB-to-I2C hardware. The I2C function is enabled on hardware after reset so this function must be called before the I2C pins (SCL and SDA) can be used as GPIO. After the function has been called, the pins are configured as open-drain IO. If the communication to the hardware was successful, the function returns a '1' while a '0' is returned if unsuccessful.

```
C/C++:    int Disable_I2C (void);
Delphi:   Disable_I2C: integer;
VB:       Disable_I2C () As Long
```

Enable_I2C

The Enable_I2C function re-enables the I2C functionality on the USB-to-I2C hardware. The I2C functionality is disabled after the Disable_I2C function is called so this function can be called to once again enable the I2C interface.
If the communication to the hardware was successful, the function returns a '1' while a '0' is returned if unsuccessful.

```
C/C++:    int Enable_I2C(void);
Delphi:   Enable_I2C: integer;
VB:       Enable_I2C () As Long
```

GPIO_I2C_Write

The GPIO_I2C_Write function writes two bits of IO data to the I2C pins. Disable_I2C must be called before using these pins as GPIO. If the communication to the hardware was successful, the function returns a '1' while a '0' is returned if unsuccessful.
The data byte will be sent to the I2C interface in the following format:

x	x	x	x	x	x	SDA	SCL

C/C++:   **int** GPIO_I2C_Write (**uchar** GPIOData);
Delphi: GPIO_I2C_Write (GPIOData : **byte**): **integer**;
VB: GPIO_I2C_Write (**ByVal** GPIOData **As Byte**) **As Long**

GPIO_I2C_Read

The GPIO_I2C_Read function reads one byte of data from the I2C pins. Disable_I2C must be called before using these pins as GPIO.
The data will read 0xFF if the read is unsuccessful.
The data byte will be returned from the I2C interface in the following format:

0	0	0	0	0	0	SDA	SCL

C/C++:   **uchar** GPIO_I2C_Read (**void**);
Delphi: GPIO_I2C_Read: **byte**;
VB: GPIO_I2C_Read () **As Byte**

Disable_SPI

The Disable_SPI function disables the SPI functionality on the USB-to-I2C hardware. The SPI function is enabled on hardware reset so this function must be called before the SPI pins SCLK, MISO, MOSI, and SSN can be used as GPIO. After the function has been called, the pins are configured as quasi bidirectional. If the communication to the hardware was successful, the function returns a '1' while a '0' is returned if unsuccessful.

C/C++:   **int** Disable_SPI (**void**);
Delphi: Disable_SPI: **integer**;
VB: Disable_SPI () **As Long**

Enable_SPI

The Enable_SPI function re-enables the SPI functionality on the USB-to-I2C hardware. The SPI functionality is disabled after the Disable_SPI function is called so this function can be called to enable SPI communications once again. If the communication to the hardware was successful, the function returns a '1' while a '0' is returned if unsuccessful.

C/C++:   **int** Enable_SPI (**void**);
Delphi: Enable_SPI: **integer**;
VB: Enable_SPI () **As Long**

Configure_SPI_GPIO

The Configure_SPI_GPIO function allows the user to change the direction (input or output) of the SPI pins used as GPIO. A "1" configures the pin as an output while a "0" configures the pin as an input. The default direction for all four SPI pins is input.

Note that this function is only available in hardware version 3 and later.

Important note: when using the hardware level shifter, this function should not be used. The MOSI, SSN, and SCLK are output only, while the MISO is an input only.

X	X	X	X	MOSI	MISO	SSN	SCLK

C/C++: `uchar Configure_SPI_GPIO (uchar GPIODirection);`
Delphi: `Configure_SPI_GPIO (GPIODirection : byte): byte;`
VB: `Configure_SPI_GPIO (ByVal GPIODirection As Byte) As Byte`

GPIO_SPI_Write

The GPIO_SPI_Write function writes four bits of data to the SPI pins. Disable_SPI must be called before using these pins as GPIO. If the communication to the hardware was successful, the function returns a '1' while a '0' is returned if unsuccessful.

Important note: when using the hardware level shifter, only the MOSI, SSN, and SCLK should be used as outputs.

The data byte will be sent to the SPI interface in the following format:

X	X	X	X	MOSI	MISO	SSN	SCLK

C/C++: `int GPIO_SPI_Write (uchar GPIOData);`
Delphi: `GPIO_SPI_Write (GPIOData : byte): integer;`
VB: `GPIO_SPI_Write (ByVal GPIOData As Byte) As Long`

GPIO_SPI_Read

The GPIO_SPI_Read function reads one byte of data from the SPI pins. Disable_SPI must be called before using these pins as GPIO. An unsuccessful read will return 0xFF.

Important note: when using the hardware level shifter, only the MISO pin can be configured as an input.

The data byte will be returned from the SPI interface in the following format:

0	0	0	0	MOSI	MISO	SSN	SCLK

C/C++: `uchar GPIO_SPI_Read (void);`
Delphi: `GPIO_SPI_Read: byte;`
VB: `GPIO_SPI_Read () As Byte`

I2C Specific Functions

USB-to-I2C Ultra provides a large number of I2C specific functions to easily provide communications with any i2c device. In reality, we only needed to provide two functions: I2CWrite and I2CRead, but in order to ease integration as well as to improve the performance of USB-to-I2C Ultra, we have included a total of eleven i2c message related functions. These have been designed to help you integrate i2c into your application as well as to give superior performance.

GetI2CFrequency

This function takes no arguments and returns the current I^{2}C clock frequency.

```
C/C++:    int GetI2CFrequency(void);
Delphi:   GetI2CFrequency: integer;
VB:       GetI2CFrequency() As Long
```

GetI2CIgnoreAck

This function returns the state of IgnoreAckFlag. See the I2CIgnoreAck function for details.

```
C/C++:    int GetI2CIgnoreAck(void);
Delphi:   GetI2CIgnoreAck: integer;
VB:       GetI2CIgnoreAck () As Long
```

I2CIgnoreAck

The I2CIgnoreAck function is used to signal the USBtoI2C hardware to complete all messages even if there was a Nack (no acknowledge) during the message. Normally, the hardware will end the message and send a Stop condition after a Nack was encountered.
When the IgnoreAckFlag is '1', all the I2C functions (I2CWriteByte, I2CRead, etc.) will return the standard error codes if there was a normal transmission, however, if there was a Nack, the error code returned will be 0x0B to signal that the message was sent but there was a Nack. Note that this function requires USB-to-I2C firmware revision 1.9 or later.

```
C/C++:    int I2CIgnoreAck (int IgnoreAckFlag);
Delphi:   I2CIgnoreAck (IgnoreAckFlag: integer): integer;
VB:       I2CIgnoreAck (ByVal IgnoreAckFlag As Long) As Long
```
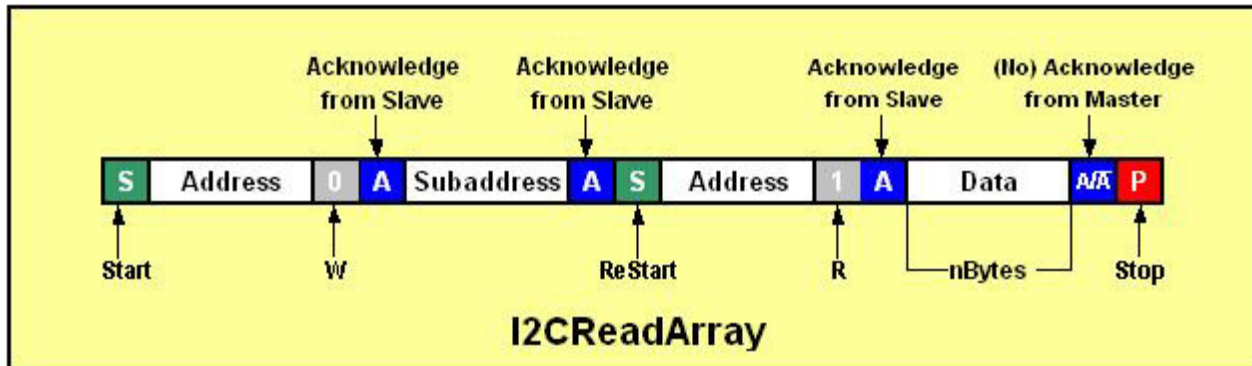
I2CReadArray

The I2CReadArray function takes four arguments: the device (slave) address, the device subaddress, the number of bytes to read, and a pointer to an element within an array of bytes. It is the calling program's responsibility to allocate the correct memory space for the array. The function ensures that the least significant bit of the address is appropriate ('1' or '0' depending on Write or Read) before it is sent to the target device.
The maximum number of data bytes (nBytes) to be read during the I2CReadArray function is currently set to a maximum of 256 bytes.

```
C/C++:    uchar I2CReadArray(uchar address, uchar subaddress, short int nBytes,
          uchar *ReadData);
Delphi:   I2CReadArray(address,subaddress:byte;nBytes:word; var ReadData:byte):byte;
VB:       I2CReadArray(ByVal address, ByVal subaddress As Byte, ByVal nBytes
```

```
As Integer, ByRef ReadData As Byte) As Byte
```


I2CReadArray

I2CReadArrayDB

This function will generally be used for reading an EEPROM of 32kbits or larger that require two subaddress bytes. The I2CReadArrayDB function takes five arguments: the device (slave) address, the device's high subaddress, low subaddress, the number of bytes to read, and a pointer to an element within an array of bytes. It is the calling program's responsibility to allocate the correct memory space for the array. The function ensures that the least significant bit of the address is appropriate ('1' or '0' depending on Write or Read) before it is sent to the target device.

The maximum number of data bytes (nBytes) to be read during the I2CReadArrayDB function is currently set to a maximum of 256 bytes.

C/C++:   `uchar I2CReadArrayDB(uchar address, uchar saHigh, uchar saLow,short int nBytes, uchar *ReadData);`

Delphi: `I2CReadArrayDB(address,subaddressHigh,subaddressLow:byte;nBytes:word; var ReadData:byte):byte;`

VB: `I2CReadArrayDB(ByVal address, ByVal saHigh As Byte, ByVal saLow As Byte, ByVal nBytes As Integer, ByRef ReadData As Byte) As Byte`
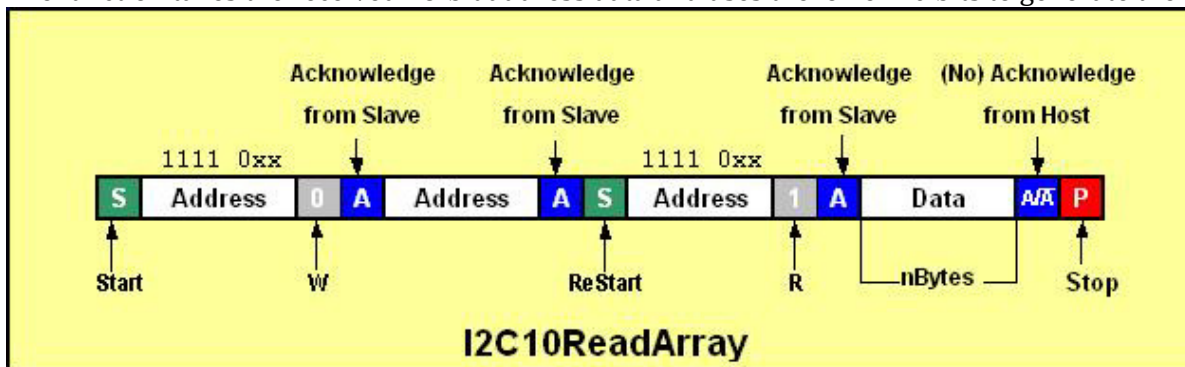
I2C10ReadArray

The I2C10ReadArray function (read an array with 10-bit device addressing) is similar to the I2CReadArray function; however, it uses 10-bit I^2C addressing. The I^2C specification states that the 10-bit address has the following format:

First byte: 1111 0xx + R/W bit

Second byte: xxxx xxxx; where x = the 10 bits of address

The function takes the received 16-bit address data and uses the lower 10 bits to generate the

I2C10ReadArray

proper 10-bit I^{2}C compliant format. A subaddress is also sent after the second byte of the address (not shown in diagram below).

The maximum number of data bytes to be read during the I2C10ReadArray function is currently set to a maximum of 256 bytes.

C/C++: `uchar I2C10ReadArray(short int address, uchar subaddress, short int nBytes, uchar *ReadData);`

Delphi: `I2C10ReadArray(address: word, subaddress: byte; nBytes: word; var ReadData: byte): byte;`

VB: `I2C10ReadArray(ByVal address As Integer, ByVal subaddress As Byte, ByVal nBytes As Integer, ByRef ReadData As Byte) As Byte`

I2CRead

The I2CRead function (read an array with no subaddress) is similar to the I2CReadArray function; however, it does not perform the write to a subaddress before the read is transmitted.

This function takes four arguments: the device address, the number of bytes to read, a pointer to an element within an array of bytes where the bytes will be stored, and finally, a variable indicating whether a Stop condition will be sent at the end of the transmission. It is the calling program's responsibility to allocate the correct memory space for the array. The function ensures that the least significant bit of the address is set to a '1' before it is sent to the target device.

The maximum number of data bytes that can be read during the I2CRead function is currently
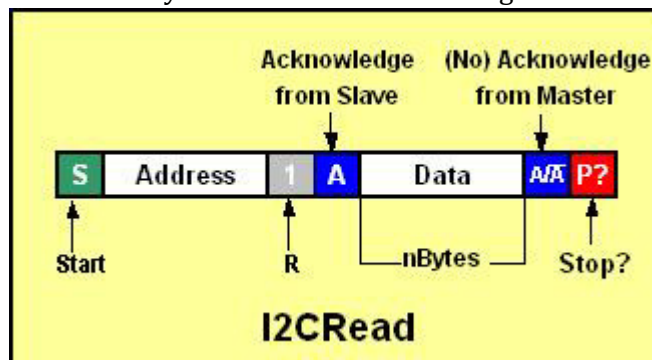

set to a maximum of 500 bytes.

C/C++: `uchar I2CRead(uchar address, short int nBytes, uchar *ReadData, short int SendStop);`

Delphi: `I2CRead(address: byte; nBytes: word; var ReadData: byte; SendStop: LongBool): byte;`

VB: `I2CRead(ByVal address As Byte, ByVal nBytes As Integer, ByRef ReadData As Byte, ByVar SendStop As Boolean) As Byte`

I2CReceiveByte

The I2CReceiveByte function reads one byte from an I^{2}C Bus/SMBus device. The function takes the device address and a pointer to a memory location used to store the data byte. I2CReceiveByte returns any error condition it encounters. The function ensures that the least significant bit of the address is a '1' before it is sent to the target device.
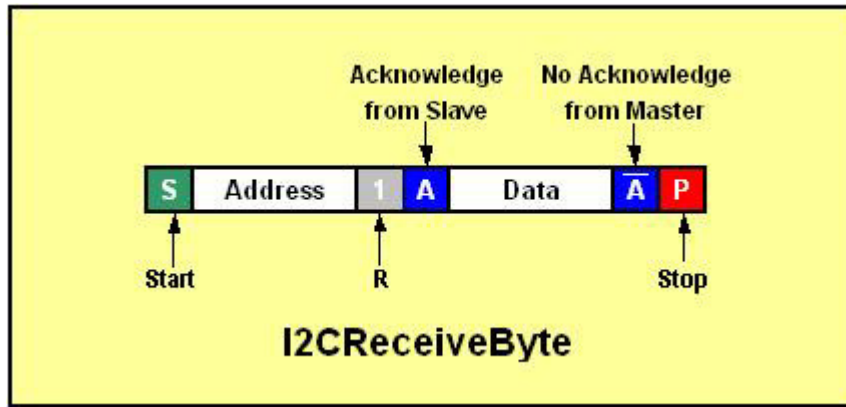
C/C++: `uchar I2CReceiveByte (uchar address, uchar *ReadData);`
Delphi: `I2CReceiveByte (address: byte; var ReadData: byte): byte;`
VB: `I2CReceiveByte (ByVal address As Byte,ByRef ReadData As Byte) As Byte`

I2CReadByte

The I2CReadByte function reads one byte from an I²C Bus/SMBus device. The function takes the device address, the subaddress and a pointer to a memory location to store the data byte. I2CReadByte returns any error condition it encounters. The function ensures that the least significant bit of the address is a '1' before it is sent to the target device.
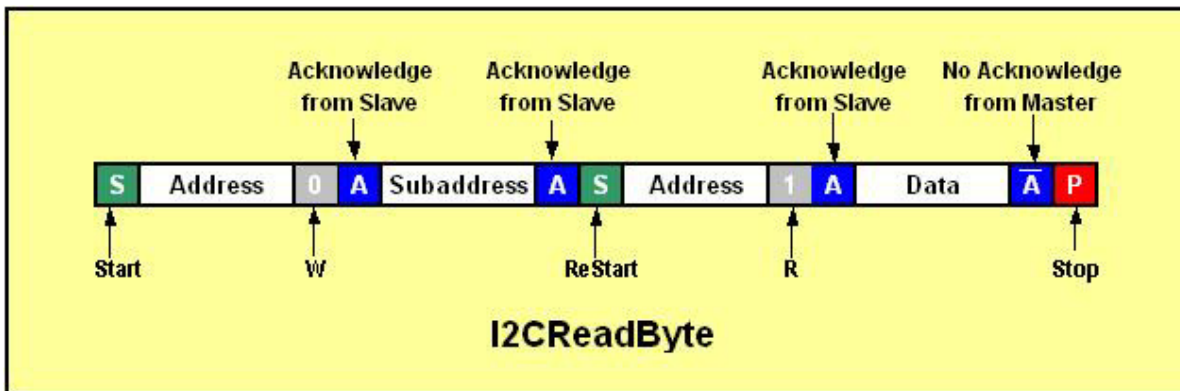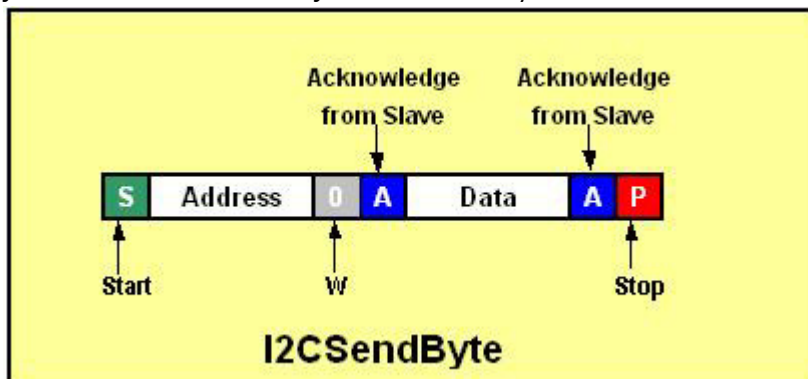

C/C++: `uchar I2CReadByte (uchar address, uchar subaddress, uchar *ReadData);`
Delphi: `I2CReadByte (address, subaddress: byte; var ReadData: byte): byte;`
VB: `I2CReadByte (ByVal address As Byte, ByVal subaddress As Byte,`
`ByRef ReadData As Byte) As Byte`

I2CSendByte

The I2CSendByte function writes one byte to an I²C Bus/SMBus device. The function takes the

device address and one data byte. I2CSendByte returns any error condition it encounters. The function ensures that the least significant bit of the address is a '0' before it is sent to the target device.

C/C++:	**uchar** I2CSendByte (**uchar** address, **uchar** DataByte);
Delphi:	I2CSendByte (address: **byte**; DataByte: **byte**): **byte**;
VB:	I2CSendByte (**ByVal** address **As Byte, ByVal** DataByte **As Byte**) **As Byte**

I2CWrite

The I2Cwrite function is a generic i2c write function. It takes four parameters: device address, number of bytes to be sent, and a pointer to an element within an array of bytes, followed by a Boolean value used to indicate if a Stop condition is required at the end of the transmission. The function ensures that the least significant bit of the address is a '0' before it is sent to the target device. The function returns any error condition it encounters.
The DLL is currently set to write up to a maximum of 500 bytes in one I2CWrite transmission.
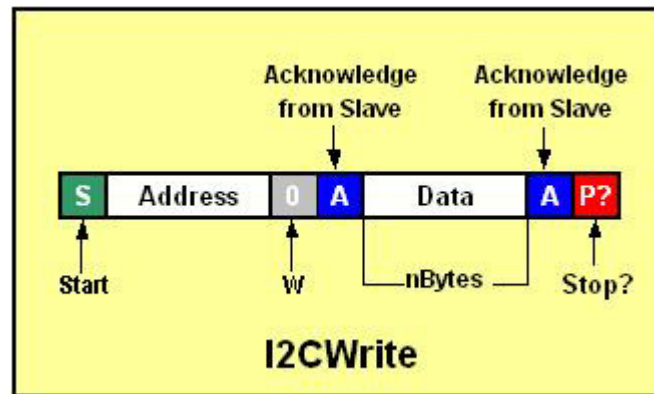
C/C++:	**uchar** I2CWrite (**uchar** address, **short int** nBytes, **uchar ***WriteData, **short int** StopCondition);
Delphi:	I2CWrite (address: **byte**; nBytes: **word**; **var** WriteData: **byte**; StopCondition: **LongBool): Byte;**
VB:	I2CWrite (**ByVal** address **As Byte, ByVal** nBytes **As Integer, ByRef** WriteData **As Byte,** StopCondition **As Bool) As Byte**

I2CWriteAddress

The I2CWriteAddress function sends an I2C address on the bus. It takes no parameters.
Only a write of the address is possible with this function; reads are not allowed.

C/C++:	**uchar** I2CWriteAddress (**uchar** address);
Delphi:	I2CWriteAddress (address: **byte): Byte;**
VB:	I2CWriteAddress (**ByVal** address **As Byte) As Byte**

I2CWriteArray

The I2CWriteArray takes four parameters: device address, device subaddress, number of bytes to be sent, and a pointer to an element within an array of bytes. The function ensures that the lsb of the address is a '0' before it is sent to the target device. The function returns any error condition it encounters.
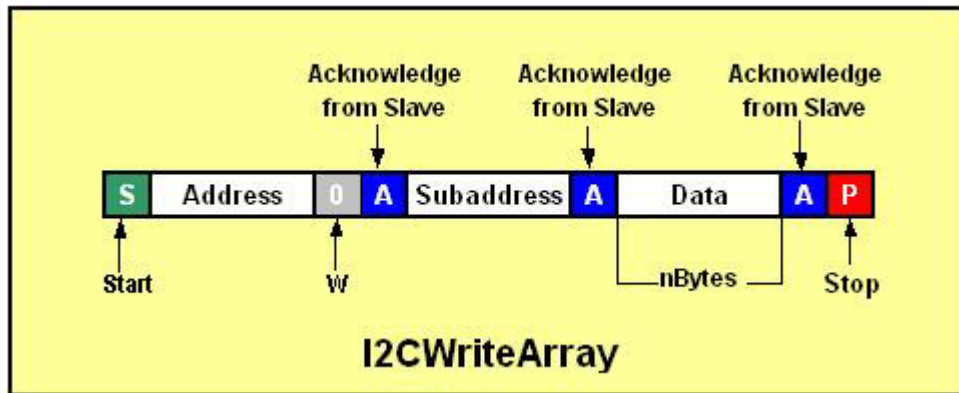
The DLL is currently set to write up to a maximum of 500 data bytes in one I2CWriteArray transmission.

C/C++:   **uchar** I2CWriteArray (**uchar** address, **uchar** subaddress, **short int** nBytes, **uchar ***WriteData);

Delphi: I2CWriteArray (address, subaddress: **byte**; nBytes: **word**; **var** WriteData: **byte**): **byte**;

VB: I2CWriteArray (**ByVal** address **As Byte**, **ByVal** subaddress **As Byte**, **ByVal** nBytes **As Integer**, **ByRef** WriteData **As Byte**) **As Byte**

I2CWriteArrayDB

The I2CWriteArrayDB is generally used to write to EEPROMs larger than 32kbits that require two subaddresses to define a memory location within the device. The I2CWriteArrayDB takes five parameters: device address, device high and low subaddresses, number of bytes to be sent, and a pointer to an element within an array of bytes. The function ensures that the lsb of the address is a '0' before it is sent to the target device. The function returns any error condition it encounters.

The DLL is currently set to write up to a maximum of 500 data bytes in one I2CWriteArrayDB transmission.

C/C++:   **uchar** I2CWriteArrayDB (**uchar** address, **uchar** saHigh, **uchar** saLow, **short int** nBytes, **uchar ***WriteData);

Delphi: I2CWriteArrayDB (address, saHigh, saLow: **byte**; nBytes: **word**; **var** WriteData: **byte**): **byte**;

VB: I2CWriteArrayDB (**ByVal** address **As Byte**, **ByVal** saHigh **As Byte**, **ByVal** saHigh **As Byte**, **ByVal** nBytes **As Integer**, **ByRef** WriteData **As Byte**) **As Byte**
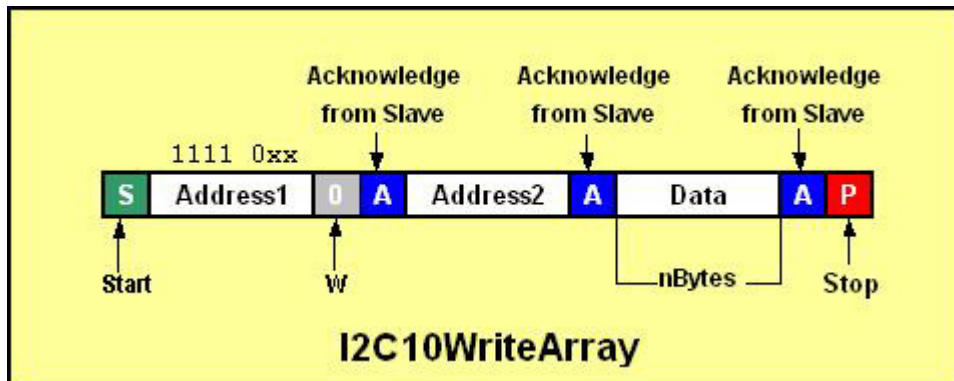
I2C10WriteArray

The I2C10WriteArray function (write an array with 10-bit device addressing) is similar to the I2CWriteArray function; however, it uses 10-bit I^{2}C addressing. The I^{2}C specification states that the 10-bit address has the following format:

First byte: 1111 0xx + R/W bit

Second byte: xxxx xxxx; where x = the 10 bits of address

The function takes the received 16-bit address data and uses the lower 10 bits to generate the proper 10-bit I^{2}C compliant format.

The DLL is currently set to write up to a maximum of 256 data bytes (nBytes) in one I2C10WriteArray transmission.

A subaddress is also sent after the second byte of the address (not shown in diagram), followed by the data.

C/C++:   **uchar** I2C10WriteArray (**short int** address, **uchar** subaddress, **short int** nBytes, **uchar ***WriteData);

Delphi: I2C10WriteArray (address, subaddress: **byte**; nBytes: **word**; **var** WriteData: **byte**): **byte**;

VB: I2C10WriteArray (**ByVal** address **As Integer**, **ByVal** subaddress **As Byte**, **ByVal** nBytes **As Integer**, **ByRef** WriteData **As Byte**) **As Byte**
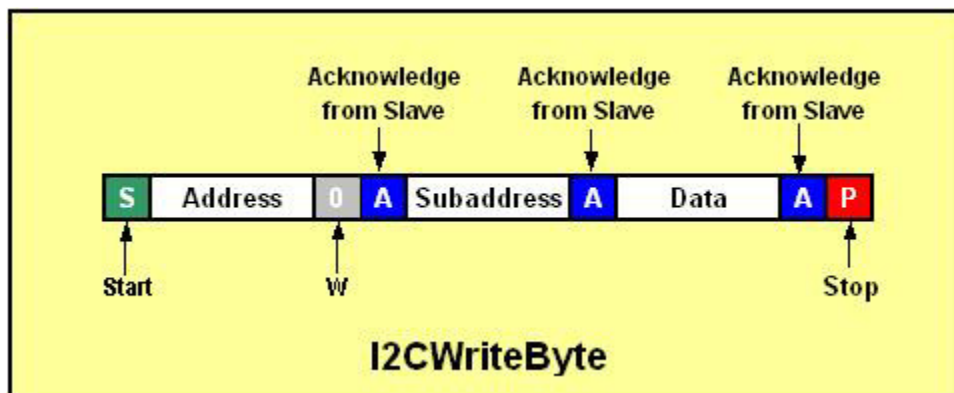
I2CWriteByte

The I2CWriteByte function writes one data byte to an I^{2}C bus device. The function takes three parameters: the device address, the subaddress, and a single data byte and returns any error condition it encounters. The function ensures that the lsb of the address is a '0' before it is sent to the target device.

C/C++:   **uchar** I2CWriteByte (**uchar** address, **uchar** subaddress, **uchar** Data);

Delphi: I2CWriteByte (address, subaddress, Data: **byte**): **byte**;

VB: I2CWriteByte (**ByVal** address **As Byte**, **ByVal** subaddress **As Byte**, **ByVal** Data **As Byte**) **As Byte**

I2CWriteRepWrite

The I2CwriteRepWrite function writes two i2c messages separated by a Restart condition. Each i2c message is defined by three parameters: the address, the number of bytes in the message, and a pointer to the data to be sent. The function ensures that the lsb of the addresses are '0' before they are sent to the target device.
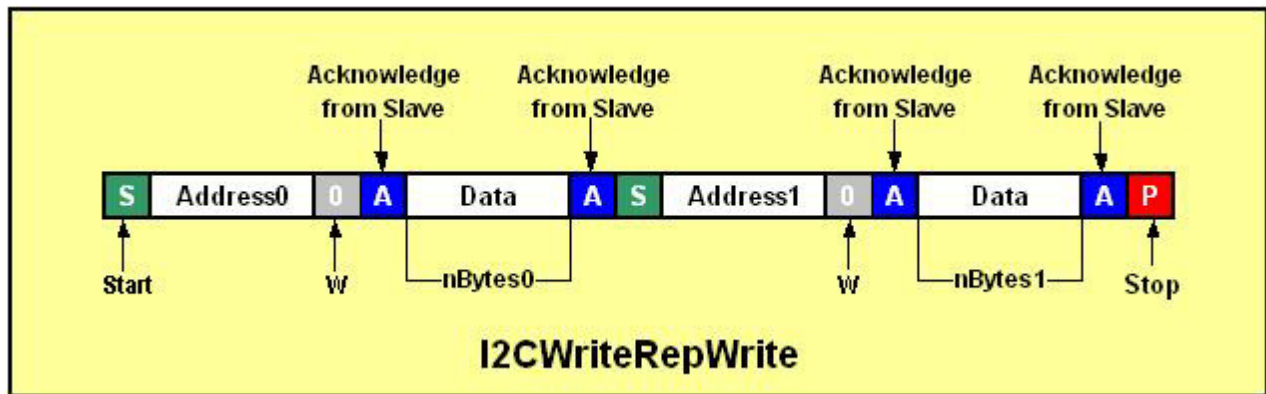
The maximum number of data bytes (nBytes0 + nBytes1) that can be sent in the two messages is set at 500 bytes.

This function is useful for a device such as a large EEPROM which requires an address and two subaddresses to define where to write the data.

C/C++:   **uchar** I2CWriteRepWrite (**uchar** address0, **short int** nBytes0, **uchar ***WriteData0, **uchar** address1, **short int** nBytes1, **uchar ***WriteData1);

Delphi: I2CWriteRepWrite (address0: **byte**; nBytes0: **short int**; **var** WriteData0: **byte**; address1: **byte**; nBytes1: **short int**; **var** WriteData1: **byte**): **byte**;

VB: I2CWriteRepWrite (**ByVal** address0 **As Byte, ByVal** nBytes0 **As Integer, ByRef** WriteData0 **As Byte, ByVal** address1 **As Byte, ByVal** nBytes1 **As Integer, ByRef** WriteData1 **As Byte) As Byte**

I2CWriteRepRead

The I2CWriteRepRead function writes an I2C message followed by an I2C read message. The two messages are separated by a Restart condition.

Each message is defined by two parameters: the number of bytes in the message, and a pointer to the data to be sent/received. The address is the same for both the write and read. The lsb of the address byte is not important since the software will set it high for a read and low for a write.

The maximum number of data bytes that can be written and read is 256 bytes each.

This function is useful for a device such as a large EEPROM or flash memory which requires an address and three subaddresses to define which memory address to read.

Using this function will be faster than using an I2C_Write followed by an I2C_Read function.

C/C++:   **uchar** I2CWriteRepRead (**uchar** address, **short int** nBytesWrite, **uchar ***WriteData, **short int** nBytesRead, **uchar ***DataRead);

Delphi: I2CWriteRepRead (address: **byte**; nBytesWrite: **word**; **var** WriteData: **byte**; nBytesRead: **word**; **var** DataRead: **byte**): **byte**;

VB: I2CWriteRepRead (**ByVal** address **As Byte, ByVal** nBytesWrite **As Integer, ByRef** WriteData **As Byte, ByVal** nBytesRead **As Integer, ByRef** DataRead **As Byte) As Byte**

SetI2CFrequency

This function sets the I^{2}C clock frequency to the value (in thousands of Hz) passed by the user's program. The frequency must be a positive integer. If a frequency is selected which is above the maximum frequency, the DLL will set the frequency to the maximum I^{2}C frequency that the USB-to-I2C Hardware is capable of achieving, currently 1000 kHz. The Hardware is not capable of achieving every discrete frequency, so it will choose the closest available frequency. The actual frequency used by the USB-to-I2C hardware will be returned by the function. The minimum frequency is 24 kHz so if a value is requested which is lower than this; the USB-to-I2C adapter will be set to the minimum value.

C/C++: `int SetI2CFrequency (int frequency);`
Delphi: `SetI2CFrequency (frequency: integer): integer;`
VB: `SetI2Cfrequency (ByVal frequency As Long) As Long`

SetI2CFrequency_DC

This function allows the user to set the frequency and the duty cycle of I^{2}C clock. While the SetI2CFrequency (see above) function attempts to get close to 50% duty cycle, there are times when you may want to use different times for the clock high and low times.
The frequency is determined by the following equation:
$f_{SCL} = 12 \times 10^6 / (SCLhigh + SCLlow)$
Example:
$f_{SCL} = 12 \times 10^6 / (25 + 15) = 300$ kHz

The minimum value for SCLhigh or SCLlow is 4.
The maximum value for SCLhigh or SCLlow is 255.
The actual frequency used by the USB-to-I2C hardware will be returned by the function.

C/C++: `int SetI2CFrequency_DC (int SCLhigh, int SCLlow);`
Delphi: `SetI2CFrequency_DC (SCLhigh, SCLlow: integer): integer;`
VB: `SetI2CFrequency_DC (ByVal SCLhigh As Long, ByVal SCLlow As Long) As Long`

SPI Specific Functions

SPI_Configure

The SPI_Configure function takes one byte that defines the configuration of the SPI master. The two low order bits M1 and M0 define the SPI mode.

```
C/C++:    uchar SPI_Configure (uchar SPI_Mode);
Delphi:   SPI_Configure (SPI_Mode: byte): byte;
VB:       SPI_Configure (ByVal SPI_Mode As Byte) As Byte
```
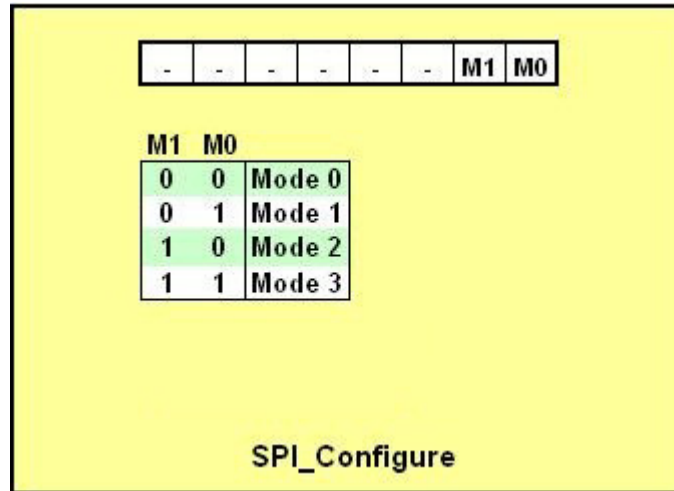

SPI_SetFrequency

The SPI_SetFrequency function takes a value that defines the frequency in kHz and configures the USB-to-I2C adapter to transmit at an appropriate frequency. The USB-to-I2C Ultra adapter has the ability to transmit at several specific frequencies. The hardware will be programmed to transmit at the following clock speeds:

```
C/C++:    int SPI_SetFrequency (int frequency);
Delphi:   SPI_SetFrequency (frequency: integer): integer;
VB:       SPI_SetFrequency (ByVal frequency As Long) As Long
```

Programmed Value (kHz)	Actual Clock Frequency (kHz)
0 to 250	250
251 to 999	500
1000 to 1999	1000
2000 to 3999	2000
4000 to 5999	4000
6000 to 7199	6000
7200 to 8999	7200
9000 to 11999	9000
12000 to 16000	12000

SPI_Generic

The SPI_Generic function writes the number of bytes defined by nBytes with values defined in an array WriteData. As the data is written, the USB-to-I2C adapter also reads the data into the array defined by pointer ReadData. The number of bytes sent and received will equal nBytes. The maximum value of nBytes is 500 bytes.
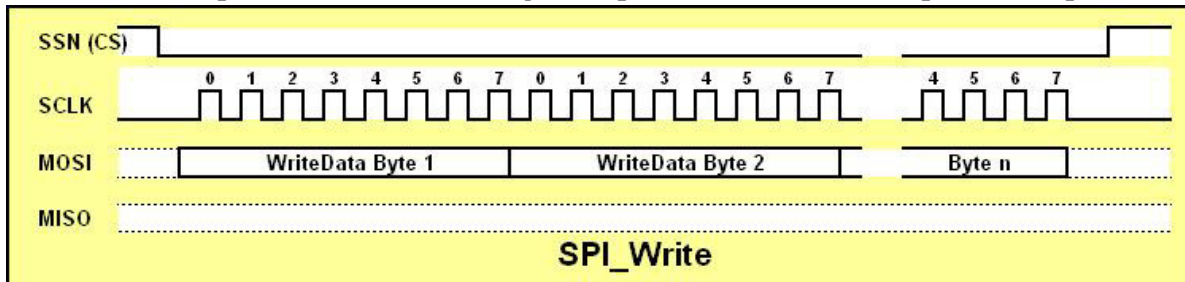
C/C++:   `uchar SPI_Generic (short int nBytes, uchar *WriteData, uchar *ReadData);`
Delphi: `SPI_Generic (nBytes: word; var WriteData: byte; var ReadData: byte): byte;`
VB: `SPI_Generic (ByVal nBytes As Integer, ByRef WriteData As Byte,`
             `ByRef ReadData As Byte) As Byte`
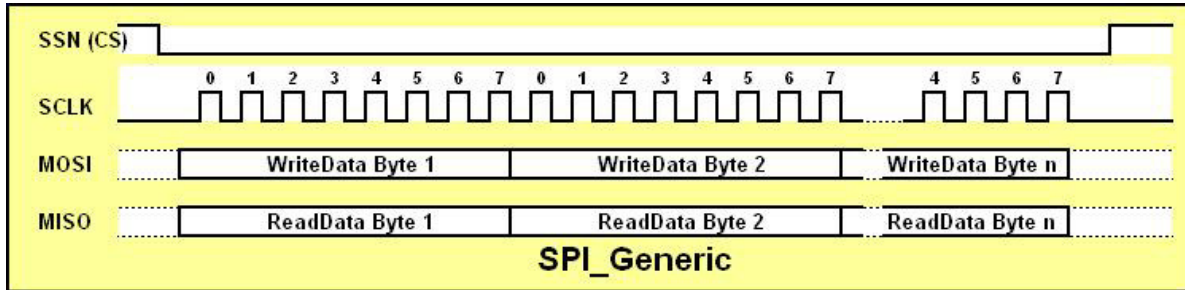
SPI_Write

The SPI_Write function writes the number of bytes defined by nBytes with values defined by pointer WriteData. The returned value will indicate if the transmission was successful. The maximum value of nBytes is 500 bytes.

C/C++:   `uchar SPI_Write (short int nBytes, uchar * WriteData);`
Delphi: `SPI_Write (nBytes: word; var WriteData: byte): byte;`
VB: `SPI_Write (ByVal nBytes As Integer, ByRef WriteData As Byte) As Byte`


SPI_WriteRead

The SPI_WriteRead function writes an array of data to an SPI slave device and reads an array from it. The nBytes0 value defines the number of bytes written while nBytes1 defines the number of bytes read. WriteData defines a pointer to the array of data to be written while ReadData points to an array where the nBytes1 bytes of data will be returned by the device. The maximum size of nBytes0 and nBytes1 is 500 bytes (each).

C/C++:   `uchar SPI_WriteRead (short int nBytesWrite, short int nBytesRead,`
             `uchar * WriteData, uchar * ReadData);`
Delphi: `SPI_WriteRead (nBytesWrite, nBytesRead: word; var WriteData: byte;`
             `var ReadData: byte): byte;`

VB: SPI_WriteRead (**ByVal** nBytesWrite **As Integer, ByVal** nBytesRead **As Integer,**
**ByRef** WriteData **As Byte, ByRef** ReadData **As Byte**) **As Byte**
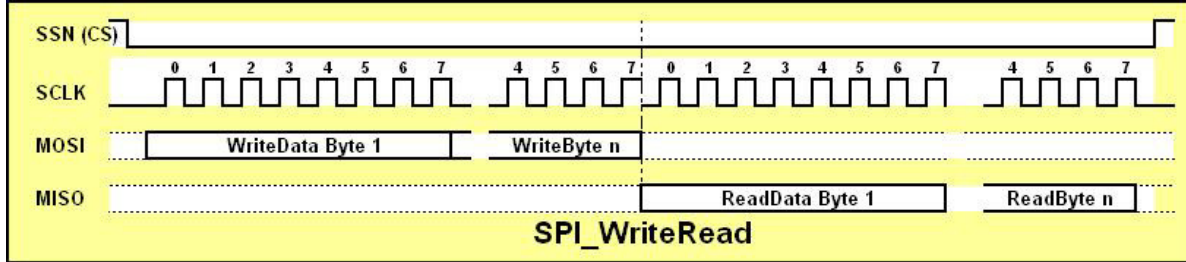
SPI_WriteWithOC

The SPI_WriteWithOC function allows you to send an OpCode during one CS cycle and then data in another CS cycle. This can be useful for devices such as EEPROMs which require the user to enable a write before the write cycle is initiated. The first byte in the WriteData array will be sent in the first CS cycle while the remaining bytes will be sent in the second CS cycle. The maximum number of bytes which can be sent in one packet is 500 bytes.

C/C++:   **uchar** SPI_WriteWithOC (**short int** nBytes, **uchar *** WriteData);
Delphi: SPI_WriteWithOC (nBytes: **word; var** WriteData: **byte**): **byte;**
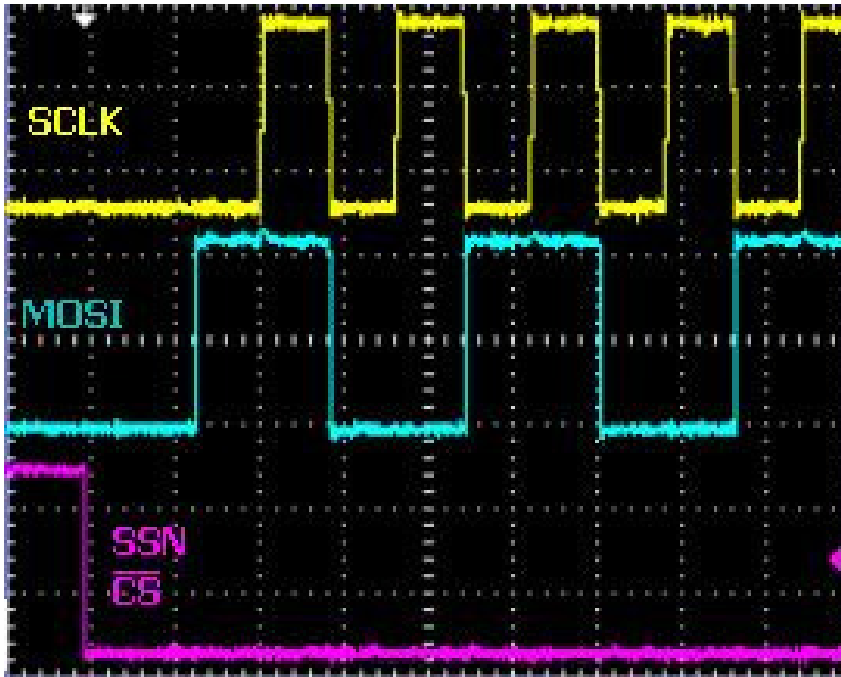**VB:**   SPI_WriteWithOC (**ByVal** nBytes **As Integer, ByRef** WriteData **As Byte**) **As Byte**
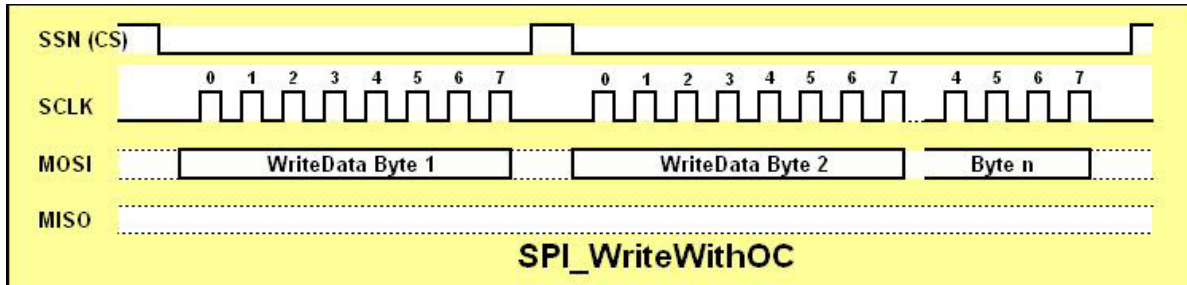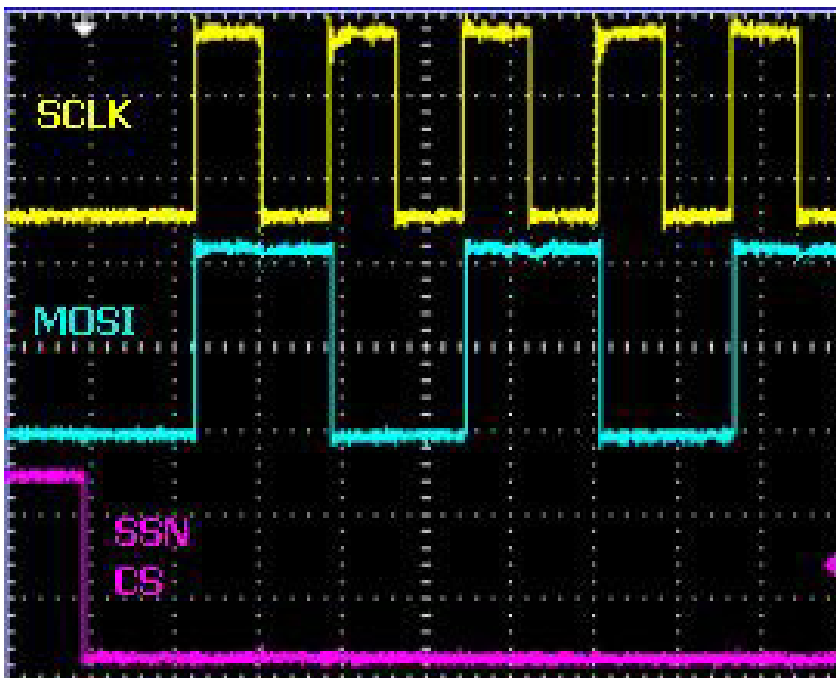
APPENDIX A

SPI Modes of Operation

SPI is defined by both a clock polarity and phase. Here are the four possible modes of operation, all are support by the USB-to-I2C DLL.
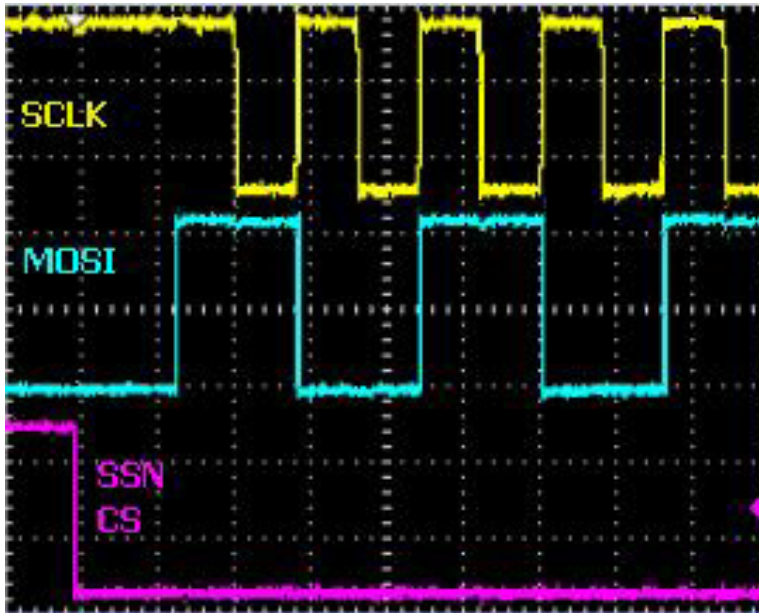

SPI MODE 0

Mode 0 operation is characterized by the clock (SCLK) starting at a low level. The data is sampled on the leading edge of the clock.
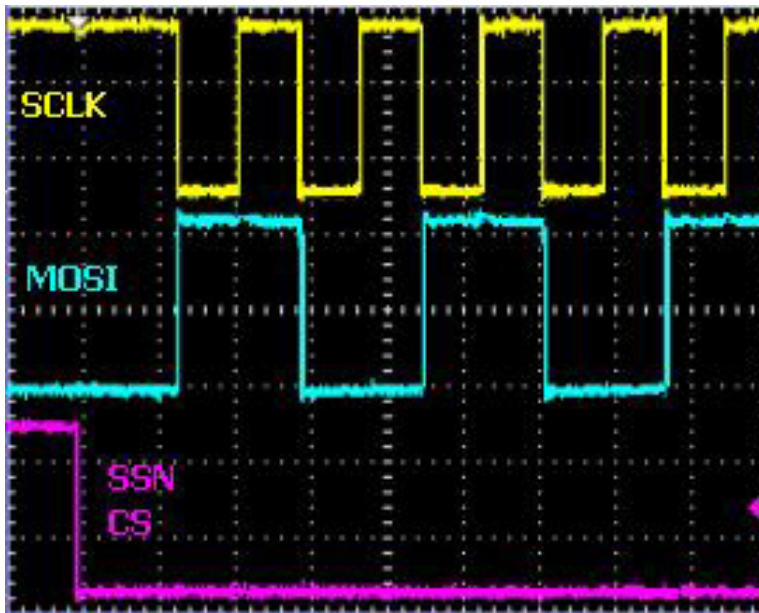

SPI MODE 1

Mode 1 operation is characterized by the clock (SCLK) starting at a low level. The data is sampled on the falling edge of the clock.

SPI MODE 2

Mode 2 operation is characterized by the clock (SCLK) starting at a high level. The data is sampled on the falling edge of the clock.

SPI MODE 3

Mode 3 operation is characterized by the clock (SCLK) starting at a high level. The data is sampled on the rising edge of the clock.

Error Codes

The following error codes are returned by the various functions in USBtoI2C_Ultra.dll:

0x00: No error
0x01: Address not Acknowledged
0x02: Data not Acknowledged
0x07: Arbitration lost
0x08: I2C Time Out
0x09: I2C Time Out with no START condition. This could occur if the bus
      was very busy or if the bus was held in a state that did not allow
      the START to be generated. Check to ensure you have pull-up
      resistors on the bus.
0x0A: Transmission aborted
0x0B: Message sent but a Nack was encountered
0x80: Unsupported function (make sure you have the latest firmware)
0xFF: Hardware not detected or USB error