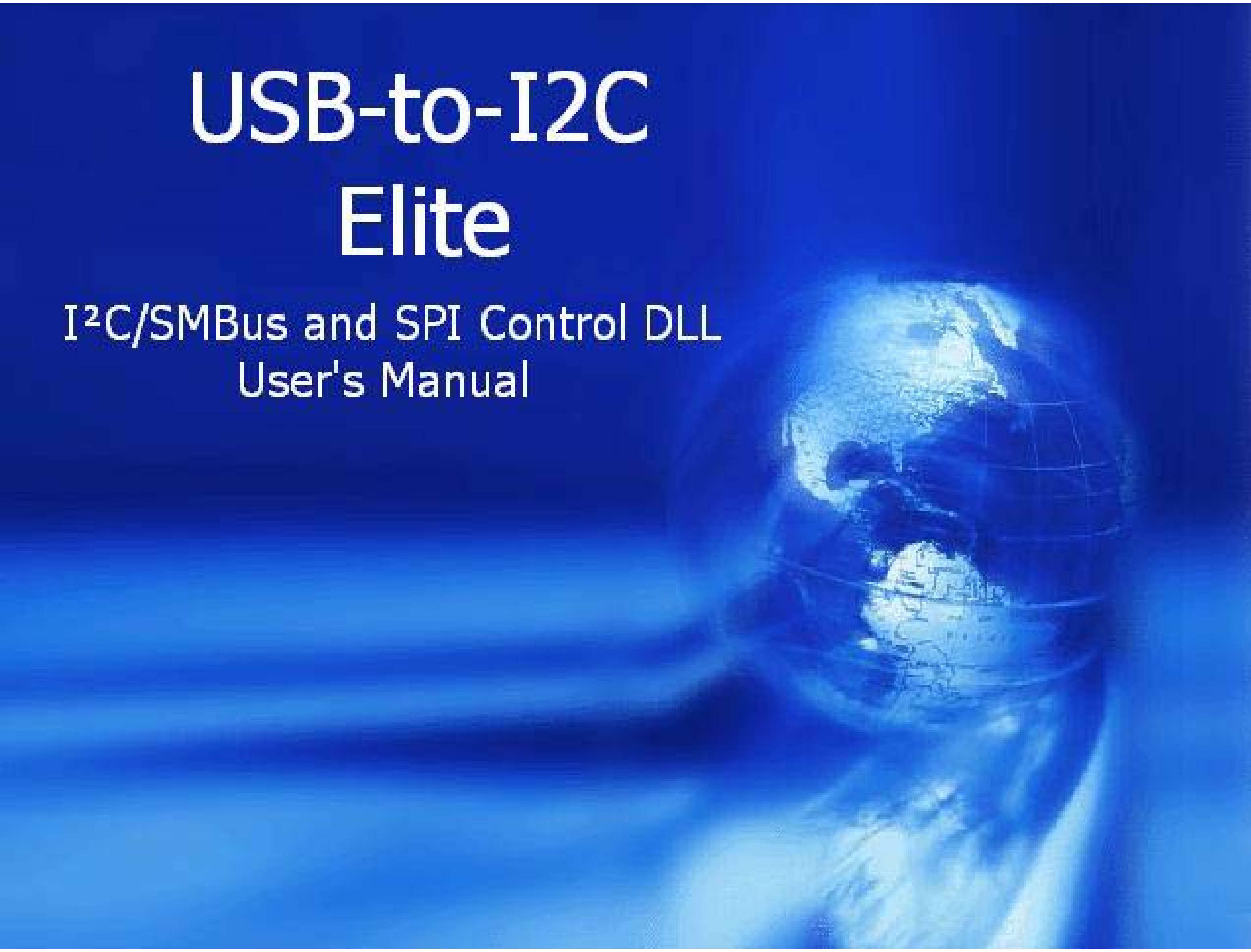


USB-to-I2C Elite

I²C/SMBus and SPI Control DLL User's Manual

Date: February 3, 2013

Questions, comments and suggestions for enhancements to this document can be emailed to:
support@i2ctools.com.

February 2013

Table of Contents

I²C Protocol

General Characteristics

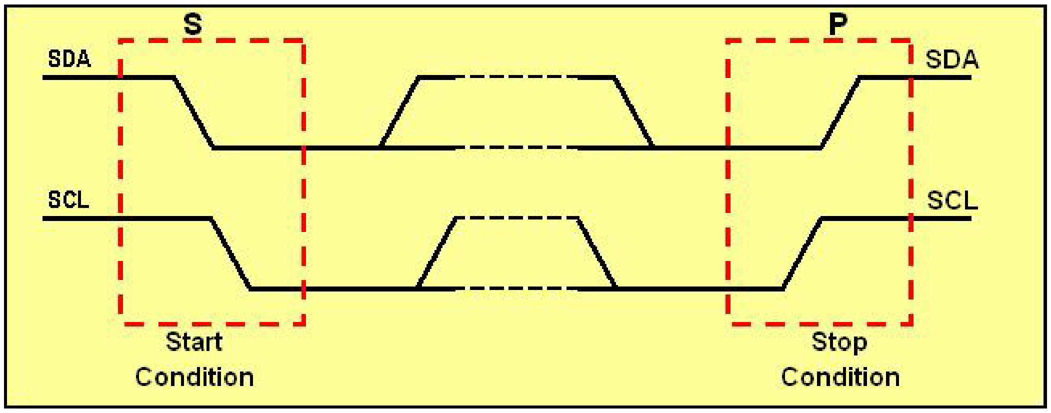
The I²C protocol allows data to be transferred between devices using two open-drain (or open-collector) bi-directional lines. One line is the serial clock (SCL) and the other is the serial data (SDA). The bus master generates the Start conditions, the clock signals on SCL, as well as the Stop condition.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

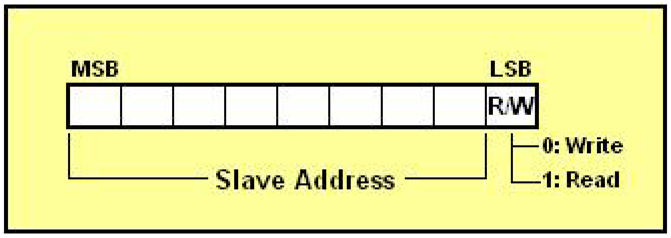
Start and Stop Conditions

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 I2C_WriteArray and I2C_ReadArray 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 Elite 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.

I^2C Bus Documentation

The complete I^2C Bus specification (UM10204) can be found at
http://www.standardics.nxp.com/literature/i2c/

Minimum System Configuration

- ✓ PC with a Pentium and 8MB RAM or better
- ✓ 32-bit or 64-bit versions of Windows XP, Vista, Windows 7, or Windows 8
- ✓ 20 MB of free HDD space
- ✓ CD ROM drive (used for installation only)
- ✓ USB port (either 1.1 or 2.0 compatible)

USB-to-I2C Elite Contents

- USB-to-I2C Elite Getting Started manual
- USB-to-I2C Elite installation CD ROM
- USB-to-I2C Elite Hardware
- USB cable
- 18-pin split cable for connection to user target system

Files installed for USB-to-I2C Elite

- USBtoI2C32.dll - this is the actual dll file you will link to your application. The installation process places this file in the appropriate Windows\System folder
- USB-to-I2C Elite DLL User's Manual (this document)
- DLL example files
- USBtoI2C Elite application
- USB-to-SPI Elite application
- USB-to-I2C Elite Software User's Manual
- USB-to-I2C Elite - Getting Started
- USB-to-I2C Elite Hardware User's Manual
- Software license agreement (license.txt)
- Registration Form (RegFile.txt)

Location of DLL

The USBtoI2C32.dll is placed in the Windows\System32 directory during installation.

Testing the Installation

After USB-to-I2C Elite 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 when installing the software, you must have Administrator privileges 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++, Delphi, C++ Builder, and Visual Basic, 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 USBtoI2C32.dll) to clean up the stack.

General Control Functions

Enable3VOutputPower

The USB-to-I2C Elite hardware contains a 3.3V output that 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 Elite 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 that 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 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 that 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 Elite 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 Elite 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 Elite 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 Elite adapters attached to the user's system. The function returns the number of USB-to-I2C Elite 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 USBtoI2C32.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).
The function returns an integer value that represents the DLL version. For example, the integer value 1234 is version 1.2.3.4.

```
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

GPIO_Configure

This function allows you to configure the IO ports as either inputs or outputs. A '0' will configure a pin as an input while a '1' configures the IO as an output.
IO0, IO1, and IO3 are fully configurable as input or output.
IO2 is an open-drain input/output pin.
Note that IO4 and IO5 can be used as either a general-purpose output or an SPI Slave Select output; therefore, these two IOs are not configurable. No configuration is needed to decide if they are general-purpose outputs or SSN outputs. The user just needs to be aware that if used as a Slave Select, data should not be written to this pin via software using the IO functions.
The default value for the GPIO_Configure register is 0x07.

x	x	x	x	IO3	IO2	IO1	IO0

C/C++: `byte GPIO_Configure(uchar PortConfiguation);`

Delphi: `GPIO_Configure (PortConfiguation: byte): byte;`

VB6: `GPIO_Configure(ByVal PortConfiguation As Byte) As Byte`

GPIO_IN

This function returns the state of the general purpose I/O port pins on the USB-to-I2C Elite hardware. The function returns a '1' for a high logic level and a '0' for a low logic level.
Even though IO4 and IO5 are output only, the state of the pins can be read back.

x	x	IO5	IO4	IO3	IO2	IO1	IO0

The input voltage should meet the following characteristics:

Vil LOW-state input voltage	**0.8V (max)**
Vih HIGH-state input voltage	**2.0V (min)**

Note that the microcontroller is a 3.3V device but pins configured as inputs or open-drain are 5V tolerant.

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

Delphi: `GPIO_IN: integer;`

VB6: `GPIO_IN() As Long`

GPIO_OUT

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

x	x	IO5	IO4	IO3	IO2	IO1	IO0

If using IO4 and IO5 for SPI Slave Selects, you must write a ‘1’ to these pins.
The Output Port has the following characteristics:
V_{oh} = 2.9V (typical @ 4 mA)
V_{ol} = 0.4V (typical @ 4mA)

C/C++: `int GPIO_OUT(int OutputState);`

Delphi: `GPIO_OUT(OutputState: integer): integer;`

VB6: `GPIO_OUT(ByVal OutputState As Long) As Long`

I2C Master Specific Functions

USB-to-I2C Elite 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: I2C_Write and I2C_Read, but in order to ease integration as well as to improve the performance of USB-to-I2C Elite, we have included a total of eleven i2c message related functions. These have been designed to help you easily integrate
I2C into your application.

I2C_GetFrequency

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

```
C/C++:  int I2C_GetFrequency(void)
Delphi: I2C_GetFrequency: integer;
VB6:    I2C_GetFrequency() As Long
```

I2C_BusRecovery

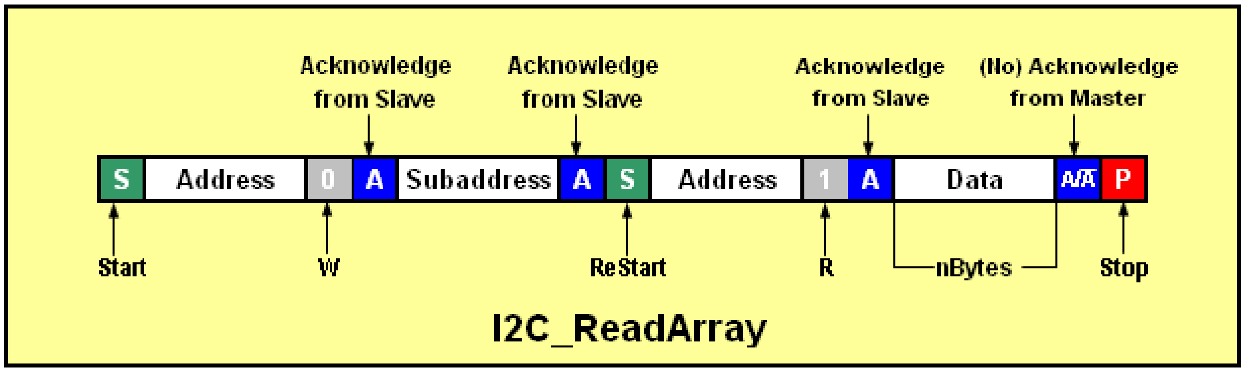
In the event that an I2C Bus is stuck low, usually due to noise on the bus, it may be possible to recover from this condition by sending nine clock pulses followed by a Stop condition.
The clock speed is 10 kHz and not adjustable by the user.
The I2C_BusRecovery function takes no arguments and returns 0 for success and a non-zero value for failure.

```
C/C++:  int I2C_BusRecovery(void);
Delphi: I2C_BusRecovery: integer;
VB6:    I2C_BusRecovery() As Long
```

I2C_ReadArray

The I2C_ReadArray 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 I2C_ReadArray function is currently set to a maximum of 4096 bytes.

```
C/C++:  uchar I2C_ReadArray(uchar address, uchar subaddress, short int nBytes, uchar *ReadData);
Delphi: I2C_ReadArray(address, subaddress: byte; nBytes: word; var ReadData: byte): byte;
VB6:    I2C_ReadArray(ByVal address, ByVal subaddress As Byte, ByVal nBytes As Integer, ByRef
        ReadData As Byte) As Byte
```



I2C_ReadArrayDB

This function will generally be used for reading an eeprom of 32 Kbits or larger and that require a two byte subaddress. The I2C_ReadArrayDB 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 I2C_ReadArrayDB function is currently set to a maximum of 4096 data bytes.

C/C++:
```
uchar I2C_ReadArrayDB(uchar address, uchar SubaddressHigh, uchar SubaddressLow, short
int nBytes, uchar *ReadData);
```

Delphi:
```
I2C_ReadArrayDB(address, subaddressHigh, subaddressLow: byte; nBytes: word; var
ReadData: byte): byte;
```

VB6:
```
I2C_ReadArrayDB(ByVal address, ByVal saHigh As Byte, ByVal saLow As Byte, ByVal
nBytes As Integer, ByRef ReadData As Byte) As Byte
```

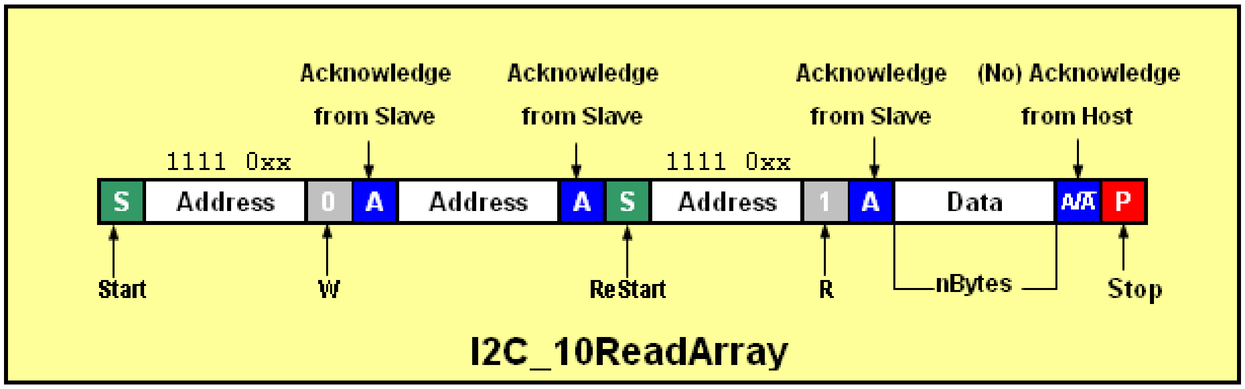
I2C_10ReadArray

The I2C_10ReadArray function (read an array with 10-bit device addressing) is similar to the I2C_ReadArray 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 proper 10-bit I^2C 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 I2C_10ReadArray function is currently set to a maximum of 4096 data bytes.

C/C++:
```
uchar I2C_10ReadArray(short int address, uchar subaddress, short int nBytes, uchar
*ReadData)
```

Delphi:
```
I2C_10ReadArray(address: word; subaddress: byte; nBytes: word; var ReadData:byte):byte;
```

VB6: `I2C_10ReadArray(ByVal address As Integer, ByVal subaddress As Byte, ByVal nBytes As Integer, ByRef ReadData As Byte) As Byte`



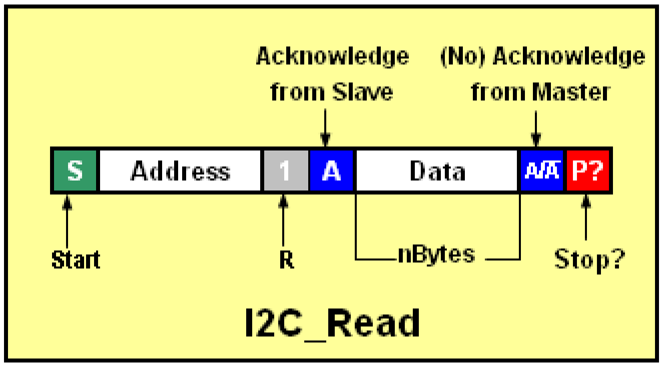
I2C_Read

The I2C_Read function (read an array with no subaddress) is similar to the I2C_ReadArray 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 I2C_Read function is currently set to a maximum of 4096 data bytes.

C/C++: `uchar I2C_Read(uchar address, short int nBytes, uchar *ReadData, short int SendStop)`

Delphi: `I2C_Read(address: byte; nBytes: word; var ReadData: byte; SendStop: LongBool): byte;`

VB6: `I2C_Read(ByVal address As Byte, ByVal nBytes As Integer, ByRef ReadData As Byte, ByVar SendStop As Boolean) As Byte`



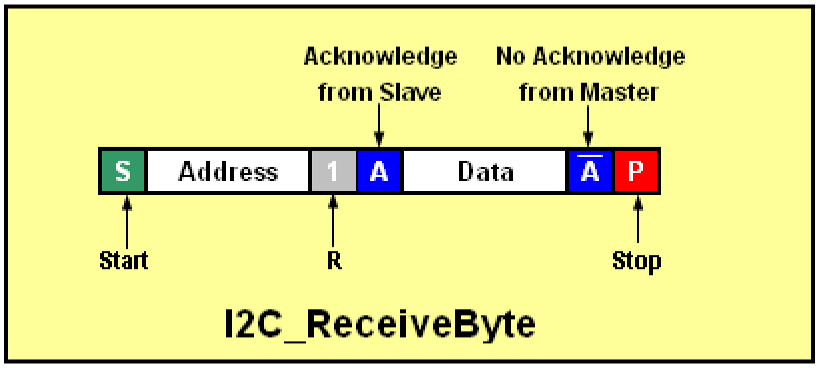
I2C_ReceiveByte

The I2C_ReceiveByte 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. I2C_ReceiveByte 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 I2C_ReceiveByte (uchar address, uchar *ReadData)`

Delphi: `I2C_ReceiveByte (address: byte; var ReadData: byte): byte;`

VB6: `I2C_ReceiveByte (ByVal address As Byte, ByRef ReadData As Byte) As Byte`



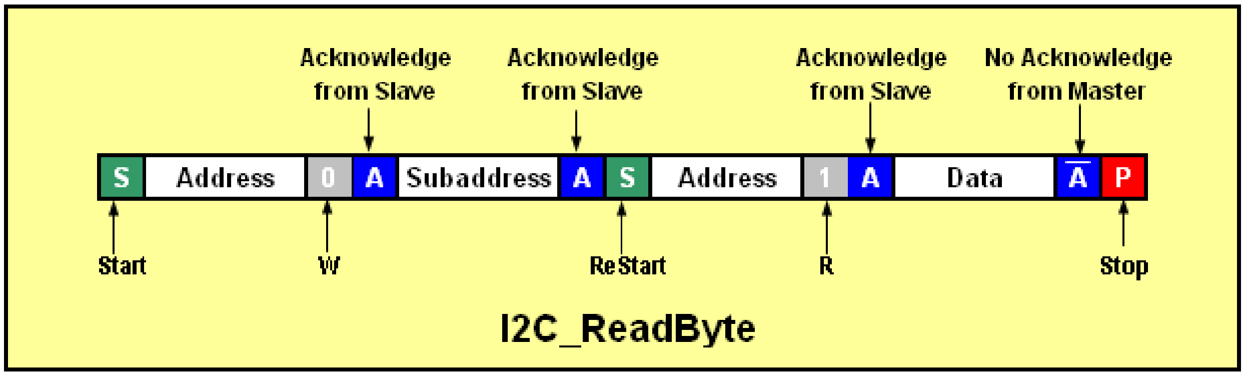
I2C_ReadByte

The I2C_ReadByte function reads one byte from an I^{2}C Bus/SMBus device. The function takes the device address, the subaddress and a pointer to a memory location to store the data byte. I2C_ReadByte 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 I2C_ReadByte (uchar address, uchar subaddress, uchar *ReadData);`

Delphi: `I2C_ReadByte (address, subaddress: byte; var ReadData: byte): byte;`

VB6: `I2C_ReadByte (ByVal address As Byte, ByVal subaddress As Byte, ByRef ReadData As Byte) As Byte`



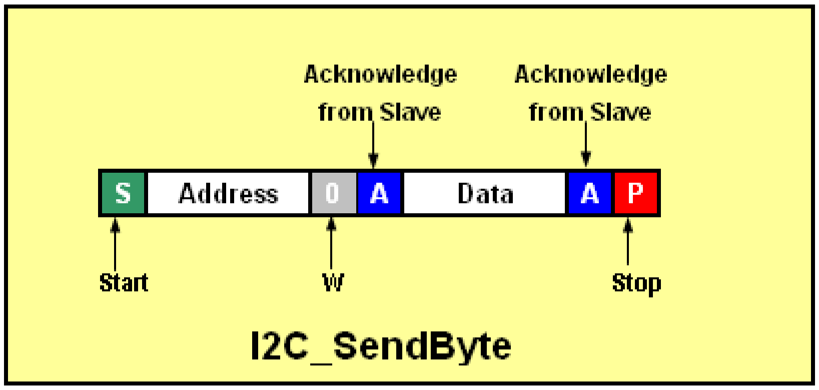
I2C_SendByte

The I2C_SendByte function writes one byte to an I^{2}C Bus/SMBus device. The function takes the device address and one data byte. I2C_SendByte 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 I2C_SendByte (uchar address, uchar DataByte);`

Delphi: `I2C_SendByte (address: byte; DataByte: byte): byte;`

VB6: `I2C_SendByte (ByVal address As Byte, ByVal DataByte As Byte) As Byte`



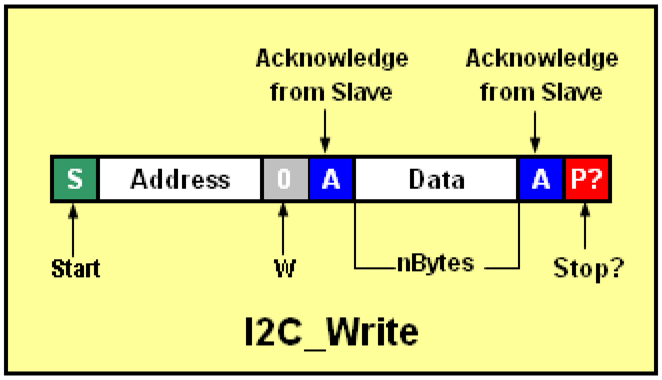
I2C_Write

The I2C_Write 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 4096 bytes in one I2C_Write transmission.

C/C++: `uchar I2C_Write (uchar address, short int nBytes, uchar *WriteData, short int Stop);`

Delphi: `I2C_Write (address: byte; nBytes: word; var WriteData: byte; Stop: LongBool): Byte;`

VB6: `I2C_Write (ByVal address As Byte, ByVal nBytes As Integer, ByRef WriteData As Byte, Stop As Bool) As Byte`



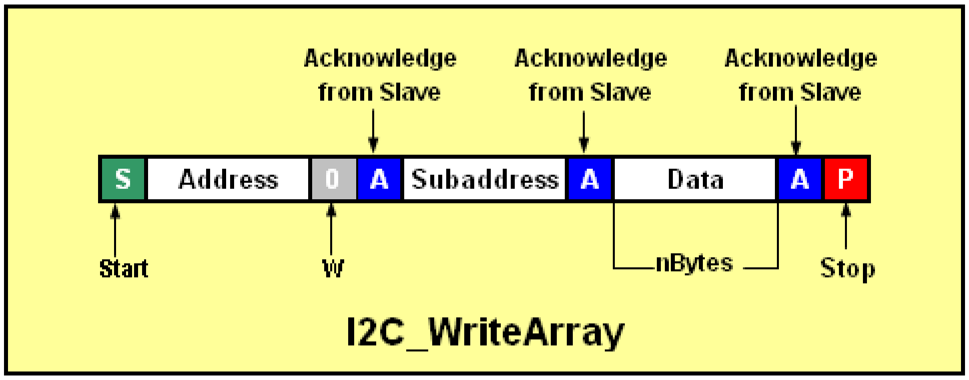
I2C_WriteArray

The I2C_WriteArray 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 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 4096 data bytes in one I2C_WriteArray transmission.

C/C++: `uchar I2C_WriteArray (uchar address, uchar subaddress, short int nBytes, uchar *WriteData);`

Delphi: `I2C_WriteArray (address, subaddress: byte; nBytes: word; var WriteData: byte): byte;`

VB6: `I2C_WriteArray (ByVal address As Byte, ByVal subaddress As Byte, ByVal nBytes As Integer, ByRef WriteData As Byte) As Byte`



I2C_WriteArrayDB

The I2C_WriteArrayDB is generally used to write to eeproms larger than 32k bits and devices that require a two-byte subaddress to define a memory location within the device. The I2C_WriteArrayDB 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 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 4096 data bytes in one I2C_WriteArrayDB transmission.

C/C++: `uchar I2C_WriteArrayDB (uchar address, uchar saHigh, uchar saLow, short int nBytes, uchar *WriteData);`

Delphi: `I2C_WriteArrayDB (address, saHigh, saLow: byte; nBytes: word; var WriteData: byte): byte;`

VB6: `I2C_WriteArrayDB (ByVal address As Byte, ByVal saHigh As Byte, ByVal saHigh As Byte, ByVal nBytes As Integer, ByRef WriteData As Byte) As Byte`

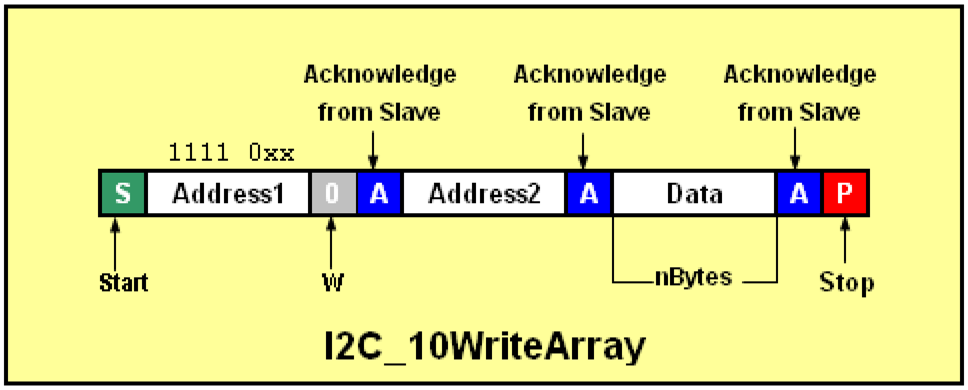
I2C_10WriteArray

The I2C_10WriteArray function (write an array with 10-bit device addressing) is similar to the I2C_WriteArray 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 4096 data bytes (nBytes) in one I2C_10WriteArray transmission.
A subaddress is also sent after the second byte of the address (not shown in diagram below), followed by the data.

C/C++: `uchar I2C_10WriteArray (short int address, uchar subaddress, short int nBytes, uchar *WriteData);`

Delphi: `I2C_10WriteArray (address, subaddress: byte; nBytes: word; var WriteData: byte): byte;`

VB6: `I2C_10WriteArray (ByVal address As Integer, ByVal subaddress As Byte, ByVal nBytes As Integer, ByRef WriteData As Byte) As Byte`



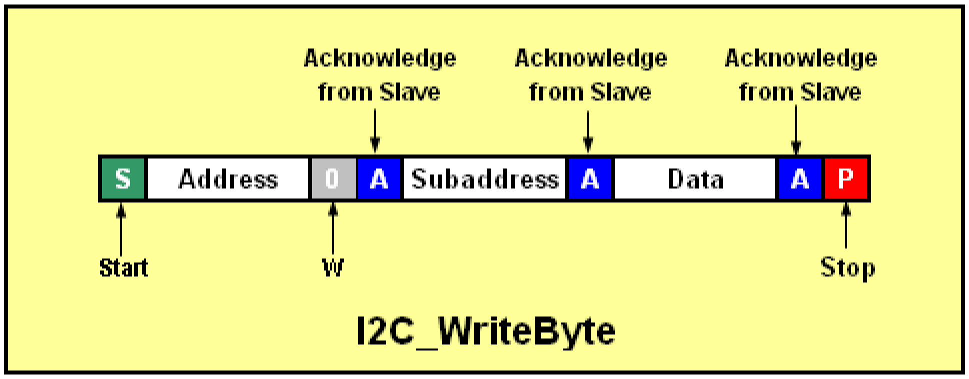
I2C_WriteByte

The I2C_WriteByte 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 least significant bit of the address is a ‘0’ before it is sent to the target device.

C/C++: `uchar I2C_WriteByte (uchar address, uchar subaddress, uchar Data);`

Delphi: `I2C_WriteByte (address, subaddress, Data: byte): byte;`

VB6: `I2C_WriteByte (ByVal address As Byte, ByVal subaddress As Byte, ByVal Data As Byte) As Byte`



I2C_WriteRepWrite

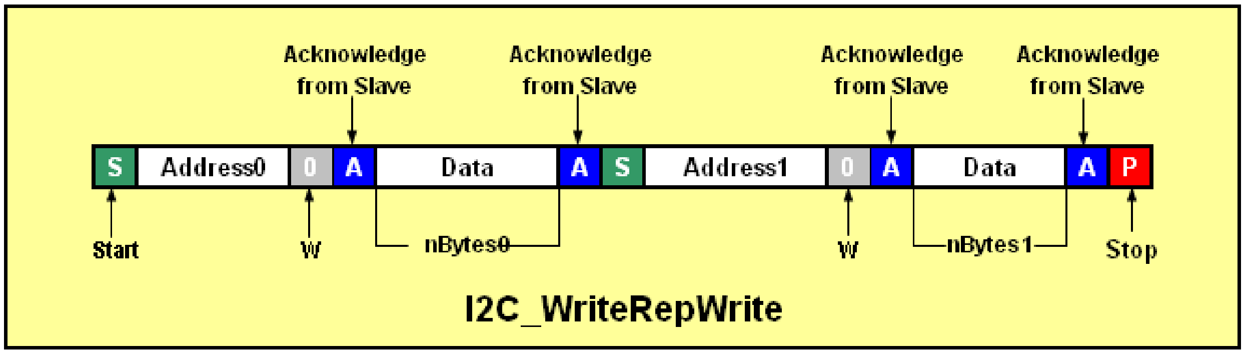
The I2C_WriteRepWrite function writes two i2c messages separated by a Restart condition. Each i2c message is defined by three parameters: the address, the number of data bytes in the message, and a pointer to the data to be sent. The function ensures that the least significant bit of the addresses are ‘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 4096 data bytes.

C/C++: `uchar I2C_WriteRepWrite (uchar address0, short int nBytes0, uchar * WriteData0, uchar address1, short int nBytes1, uchar * WriteData1);`

Delphi: `I2C_WriteRepWrite (address0: byte; nBytes0: word; var WriteData0: byte; address1: byte;`

```
nBytes1: word; var WriteData1: byte): byte;
```

VB6:
```
I2C_WriteRepWrite (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
```



I2C_SetFrequency

This function sets the I^{2}C clock frequency to the value (in 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 Elite Hardware is capable of achieving, currently 3000 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 Elite hardware will be returned by the function. The minimum frequency is 500 Hz so if a value is requested which is lower than this; the USB-to-I2C adapter will be set to the minimum value. If an error occurs when sending the data to the hardware, a value of 0 will be returned.

The user should be aware of the timeout built into the USB-to-I2C Elite hardware that terminates any I2C message that requires over two seconds to complete. For example, if using 500 Hz, the maximum number of bytes that can be transmitted is less than 900 bytes before this timeout will occur.

The USB-to-I2C Elite hardware is capable of operating at much higher frequencies than the 400 kHz fast mode defined in the I2C specification. At low bus capacitance, it may be possible to successfully operate at high frequencies, but the user should be aware that the actual maximum frequency they are able to achieve will be application specific.

C/C++:
```
int I2C_SetFrequency (int frequency);
```

Delphi:
```
I2C_SetFrequency (frequency: integer): integer;
```

VB6:
```
I2C_Setfrequency (ByVal frequency As Long) As Long
```

I2C Slave Specific Functions

The following functions have been provided to allow the USB-to-I2C Elite hardware to emulate an I2C slave device. When enabled, it will behave like a 256 byte static RAM. You can write up to 256 bytes of data and read them back.

I2C_SetToSlaveMode

The USB-to-I2C Elite hardware will not respond as an I2C slave until this function is called. When operating as a slave, the master mode is disabled.

```
C/C++:   uchar I2C_SetToSlaveMode(uchar I2Caddress);
Delphi:  I2C_SetToSlaveMode(I2Caddress: byte): byte;
VB6:     I2C_SetToSlaveMode (ByVal I2Caddress As Byte) As Byte
```

I2C_SetToMasterMode

The USB-to-I2C Elite hardware operates as an I2C master at start-up. This function only needs to be called if it is currently running in slave mode, and you would like to begin operating as an I2C master again.

```
C/C++:   uchar I2C_SetToMasterMode (void);
Delphi:  I2C_SetToMasterMode: byte;
VB6:     I2C_SetToMasterMode () As Byte
```

I2C_SlaveReadBuffer

The contents of the slave buffer can be read back using the I2C_SlaveReadData function. The user provides the number of bytes to return. The data returned always starts with the first byte in the slave device data buffer.
The maximum number of bytes that will be returned is 256. Note that you can read back up to 256 data bytes even if you previously defined the slave buffer size to a value less than 256 bytes.

```
C/C++:   uchar I2C_SlaveReadBuffer (short int nBytes, uchar *ReadData);
Delphi:  I2C_SlaveReadBuffer (nBytes: word; var ReadData: byte): byte;
VB6:     I2C_SlaveReadBuffer (ByVal nBytes As Integer, ByRef ReadData As Byte) As Byte
```

I2C_SlaveSetSize

The default slave size is 256 bytes. The user can change the size of the slave buffer to any value between 1 and 256. Setting the slave size determines the behavior of the slave when data is being sent to it. If the number of bytes sent during a write is greater than the size of the slave buffer, the incoming data will overwrite what was previously received. For example, if the slave size is set to 16 bytes, and 18 data bytes are written to the slave, two data bytes will be overwritten.

```
C/C++:   uchar I2C_SlaveSize (short int nBytes);
```

```
Delphi: I2C_SlaveSize (nBytes: word): byte;
VB6:    I2C_SlaveSize (ByVal nBytes As Integer) As Byte
```

I2C_SlaveIdle

The slave functionality can temporarily be disabled by using the I2C_SlaveIdle function. The hardware is still operating as a slave, but it will not respond to its address.

```
C/C++:  uchar I2C_SlaveIdle (void);
Delphi: I2C_SlaveIdle: byte;
VB6:    I2C_SlaveIdle () As Byte
```

I2C_SlaveResume

A slave that has been idled may be restarted by using the I2C_SlaveResume function.

```
C/C++:  uchar I2C_SlaveResume (void);
Delphi: I2C_SlaveResume: byte;
VB6:    I2C_SlaveResume () As Byte
```

I2C_SlaveFillBuffer

The I2C_SlaveFillBuffer function can be used to initialize the slave buffer with a specific value. The value will fill the entire 256 data bytes in the slave buffer. This function is useful to initialize or "erase" the contents of the buffer.

```
C/C++:  uchar I2C_SlaveFillBuffer (uchar FillValue);
Delphi: I2C_SlaveFillBuffer (FillValue: byte): byte;
VB6:    I2C_SlaveFillBuffer (ByVal FillValue As Byte) As Byte
```

MSA Specific Functions

MSA is a system for testing transceiver modules associated to different multi-source agreement standards, comprising a transceiver testing device and an interface connection unit. The transceiver testing device comprises a processor for performing at least one testing operation on an MSA transceiver module associated to a certain multi-source agreement standard. The MSA protocol is designed to be implemented over an I^2C data link. Refer to the I2C REFERENCE DOCUMENT FOR 300 PIN MSA 10G and 40G TRANSPONDER available from several sources on the World Wide Web.
USB-to-I2C Elite has incorporated functions to implement the MSA standard.

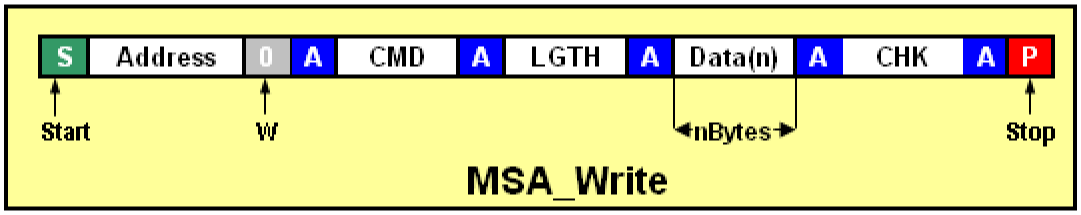
MSA_Write

The MSA_Write function sends an array of data to a device compliant to the MSA standard. The function requires the user provide an address and a pointer to an array of data to be transmitted.
The array should be structured with the data in the following sequence:
1. A CMD byte which contains the code of the command to be executed by the module.
2. A LGTH byte which contains the length of the command parameters field (may be 0, and up to 18 bytes in the MSA standard, however a user may use up to 255 bytes in the MSA_Write function);
3. An optional command parameters field containing the Data
4. A CHK byte, which contains the message check byte. Note that the CHK byte is not included in the LGTH field.
The array to write to the target device is therefore a minimum of three bytes (CMD, LGTH, and CHK) and a maximum of twenty one bytes according to the MSA standard.
The returned value is the Error Code of the function (see Error Codes at the end of this document).

C/C++: `uchar MSA_Write (uchar address, uchar *WriteData);`

Delphi: `MSA_Write I2C_Write (address: byte; var WriteData: byte): Byte;`

VB6: `MSA_Write (ByVal address As Byte, ByRef WriteData As Byte) As Byte`



MSA_Read

The MSA_Read function receives an array of data from a device compliant to the MSA standard. The function requires the user provide an address and a pointer to an array where the data will be stored.
The device should respond with data in the following format:
1. A STS byte which reflects the status byte of the module, giving information about the last received command completion;
2. A LGTH byte which contains the length of the command parameters field (may be 0, and up to 18 bytes in the MSA standard, however a user may use up to 255 bytes in the MSA_Read function);

3. An optional command parameters field containing the Data. The MSA_Read function reads this LGTH and will continue reading the number of data bytes found in this field;
4. A CHK byte, which contains the message check byte. Note that the CHK byte is not included in the LGTH field.
The array read from the target device is therefore a minimum of three bytes (STS, LGTH, and CHK) and a maximum of twenty one bytes according to the MSA standard. Note that the MSA_Read function allows LGTH to be a value from 0 to 255 bytes.
The returned value is the Error Code of the function (see Error Codes at the end of this document).

C/C++:
```
uchar MSA_Read (uchar address, uchar *ReadData);
```

Delphi:
```
MSA_Read (address: byte; var ReadData: byte): Byte;
```

VB6:
```
MSA_Read (ByVal address As Byte, ByRef ReadData As Byte) As Byte
```

MSA_WriteRead

The MSA_WriteRead function sends an MSA_Write function followed by an MSA_Read function (defined above). The user must specify an array of data to write structured as defined in the MSA_Write function. The MSA_Read function will use the same address as the write function.
A 5us (approximately) delay is inserted automatically between the write and read to comply with the I^{2}C specification. An additional delay time can be added. The 16-bit Delay value is measured in tens of microseconds. Therefore, the delay between the write and read will be a minimum of 0.005 microseconds and a maximum of 0.655 seconds (5 to 655355 microseconds).

C/C++:
```
uchar MSA_WriteRead (uchar address, uchar *WriteData, short int Delay,
uchar *ReadData);
```

Delphi:
```
MSA_WriteRead I2C_Write (address: byte; var WriteData: byte; Delay: word;
var ReadData: byte): byte;
```

VB6:
```
MSA_WriteRead (ByVal address As Byte, ByRef WriteData As Byte, ByVal Delay As int,
ByRef ReadData As Byte) As Byte
```

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 define the SPI mode. The valid modes are 0, 1, 2, and 3.

```
C/C++:  uchar SPI_Configure (uchar SPI_Mode);
Delphi: SPI_Configure (SPI_Mode: byte): byte;
VB6:    SPI_Configure (ByVal SPI_Mode As Byte) As Byte
```

SPI_GetConfiguration

The SPI_Configuration function returns the frequency of the SPI interface and places the SPI Mode in the location specified by SPI_Mode.

```
C/C++:  integer SPI_GetConfiguration (uchar *SPI_Mode);
Delphi: SPI_GetConfiguration (var SPI_Mode: byte): integer;
VB6:    SPI_GetConfiguration (ByRef SPI_Mode As Byte) As Long
```

SPI_SetFrequency

The SPI_SetFrequency function takes a value that defines the frequency in Hz and configures the USB-to-I2C Elite adapter to transmit at an appropriate frequency.
The valid frequencies range is from 5 kHz to 15 MHz.
Note that not all integer values within the frequency range can be achieved. The value returned by the function reflects the actual frequency that will be used by the hardware.

```
C/C++:  int SPI_SetFrequency (int frequency);
Delphi: SPI_SetFrequency (frequency: integer): integer;
VB6:    SPI_SetFrequency (ByVal frequency As Long) As Long
```

SPI_Generic

The SPI_Generic function writes the number of bytes defined by nByte with values defined in array WriteData. As the data is written, the USB-to-I2C Elite adapter also reads the data into the array defined by pointer ReadData. The number of bytes sent and received will equal nByte.
The parameter SSN defines the Slave Select output that will be active during the SPI communications. Valid values for SSN are 0, 1, and 2.

SS	Active Slave Select Output
0	SSN
1	IO5/SSN1

2	IO4/SSN2

The maximum value of nByte is 4096 bytes.

C/C++: `uchar SPI_Generic (short int nByte, int SS, uchar *WriteData, uchar *ReadData);`

Delphi: `SPI_Generic (nByte: word; SS: integer; var WriteData: byte;var ReadData: byte): byte;`

VB6:
```
SPI_Generic (ByVal nByte As Integer, ByVal SS As Long, ByRef WriteData As Byte,
ByRef ReadData As Byte) As Byte
```

SPI_ByteToggle

The SPI_ByteToggle is very similar to the SPI_Generic function but it toggles the Slave Select output between bytes.
This function writes the number of bytes defined by nByte with values defined in array WriteData. As the data is written, the USB-to-I2C Elite adapter also reads the data into the array defined by pointer ReadData. The number of bytes sent and received will equal nByte.
The Slave Select output toggles between each byte.
The parameter SSN defines the Slave Select output that will be active during the SPI communications. Valid values for SSN are 0, 1, and 2.

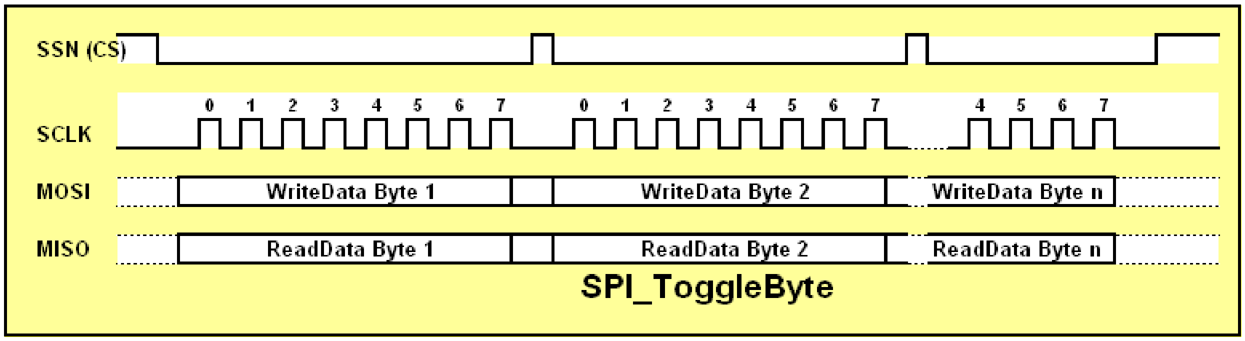
SS	Active Slave Select Output
0	SSN
1	IO5/SSN1
2	IO4/SSN2

The maximum value of nBytes is 4096 bytes.

C/C++: `uchar SPI_ByteToggle (short int nByte, int SS, uchar *WriteData, uchar *ReadData);`

Delphi: `SPI_ByteToggle (nByte: word; SS: integer; var WriteData: byte;var ReadData: byte): byte;`

VB6:
```
SPI_ByteToggle (ByVal nByte As Integer, ByVal SS As Long, ByRef WriteData As Byte,
ByRef ReadData As Byte) As Byte
```

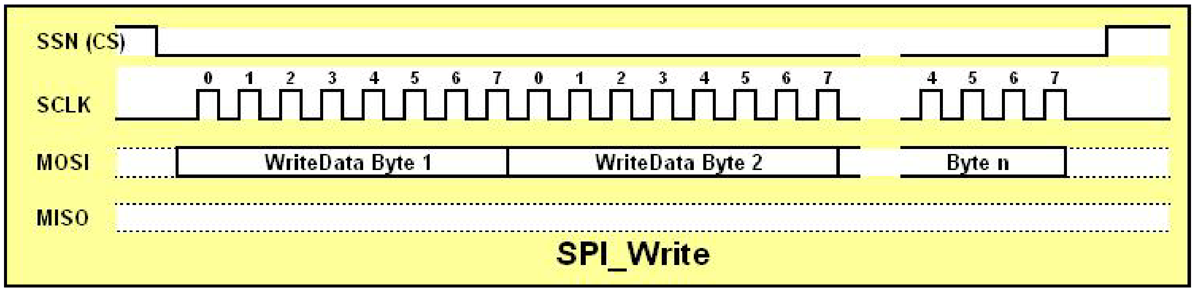


SPI_Write

The SPI_Write function writes the number of bytes defined by nBytes with values defined by pointer WriteData. The returned value will be 0 to indicate if the transmission was successful. The maximum value of nBytes is 4096 bytes.
The parameter SS defines the Slave Select output that will be active during the SPI communications. Valid values for SSN are 0, 1, and 2.

SS	Active Slave Select Output
0	SSN
1	IO5/SSN1
2	IO4/SSN2

```
C/C++:  uchar SPI_Write (short int nBytes, int SS, uchar * WriteData);
Delphi: SPI_Write (nBytes: word; SS: integer; var WriteData: byte): byte;
VB6:    SPI_Write (ByVal nBytes As Integer, ByVal SS As Long, ByRef WriteData As Byte) As Byte
```



SPI_WriteRead

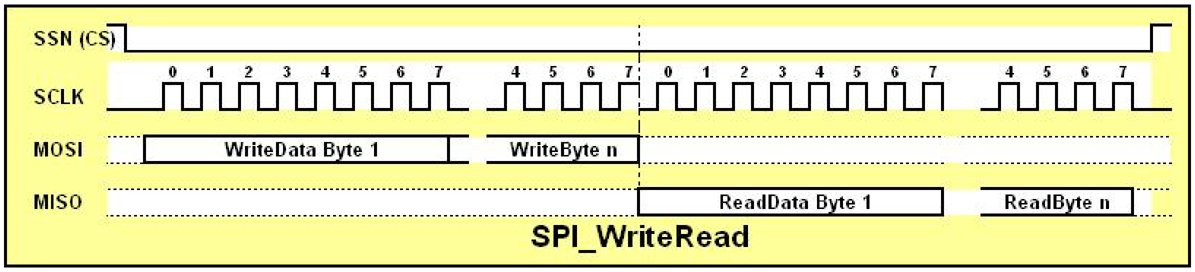
The SPI_WriteRead function writes an array of data to an SPI slave device and reads an array from it. The nBytes0 value defines the number of bytes written while nBytes1 defines the number of bytes read. WriteData defines a pointer to the array of data to be written while ReadData points to an array where the nBytes1 bytes of data will be returned by the device.
The maximum size of nBytes0 and nBytes1 is 4096 bytes (each).
The parameter SSN defines the Slave Select output that will be active during the SPI communications. Valid values for SSN are 0, 1, and 2.

SS	Active Slave Select Output
0	SSN
1	IO5/SSN1
2	IO4/SSN2

```
C/C++:  uchar SPI_WriteRead (short int nBytesWrite, short int nBytesRead, int SS,
        uchar * WriteData, uchar * ReadData);
```

Delphi: `SPI_WriteRead (nBytesWrite, nBytesRead: word; SS: integer; var WriteData: byte; var ReadData: byte): byte;`

VB6: `SPI_WriteRead (ByVal nBytesWrite As Integer, ByVal nBytesRead As Integer, ByVal SS As Long, ByRef WriteData As Byte, ByRef ReadData As Byte) As Byte`



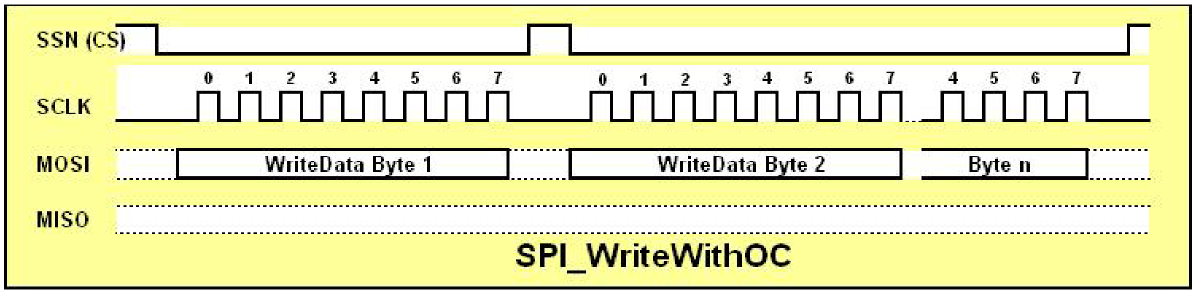
SPI_WriteWithOC

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 4096 bytes. The value nBytes is the total number of bytes sent by the function, including the byte used in the first CS cycle. The SSN (CS) signal is deasserted for 3us after the first byte is sent.

C/C++: `uchar SPI_WriteWithOC (short int nBytes, int SS, uchar * WriteData);`

Delphi: `SPI_WriteWithOC (nBytes: word; SS: integer; var WriteData: byte): byte;`

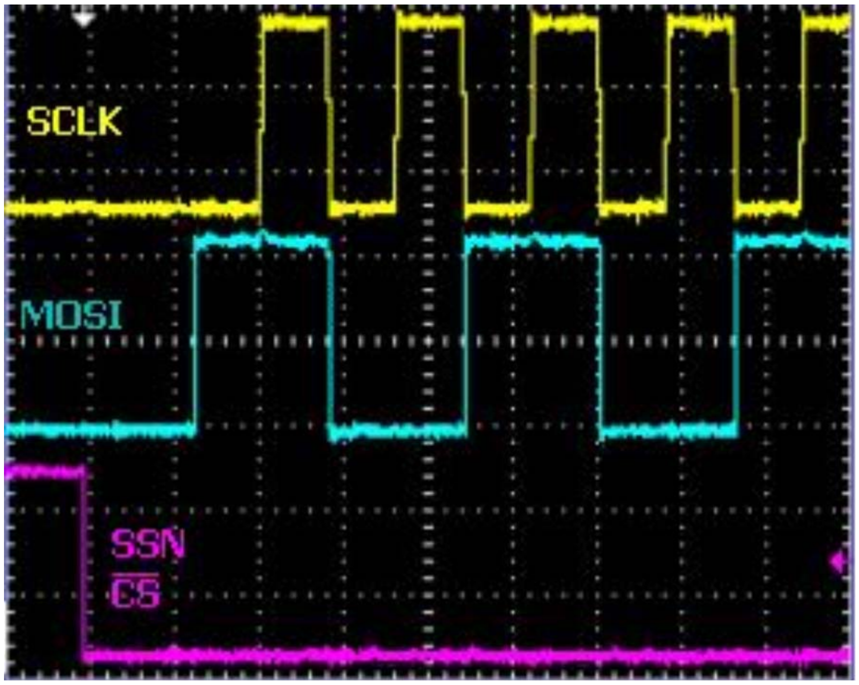
VB6: `SPI_WriteWithOC (ByVal nBytes As Integer, ByVal SS As Long, ByRef WriteData As Byte) As Byte`



APPENDIX A

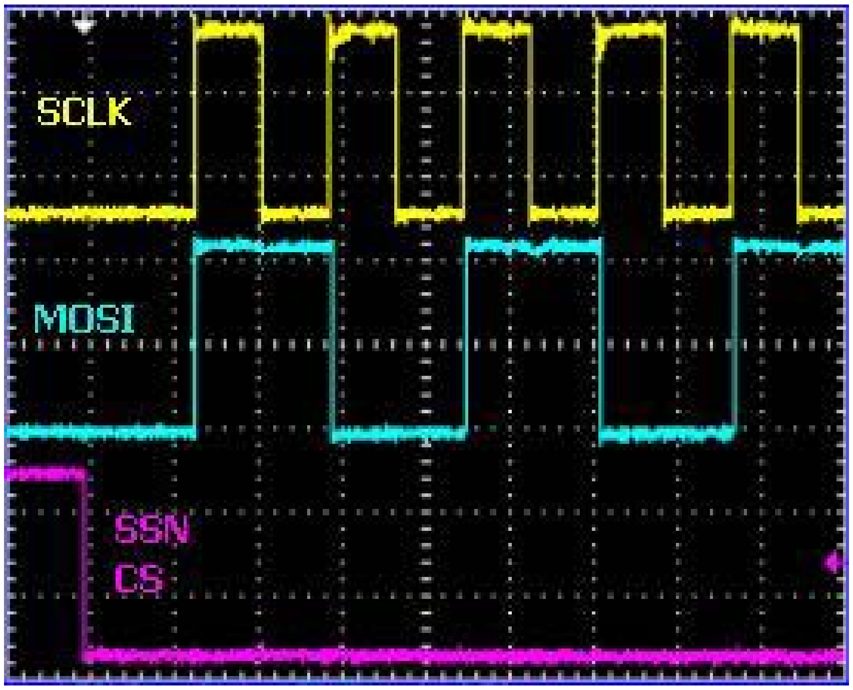
SPI Modes of Operation

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 Elite DLL.



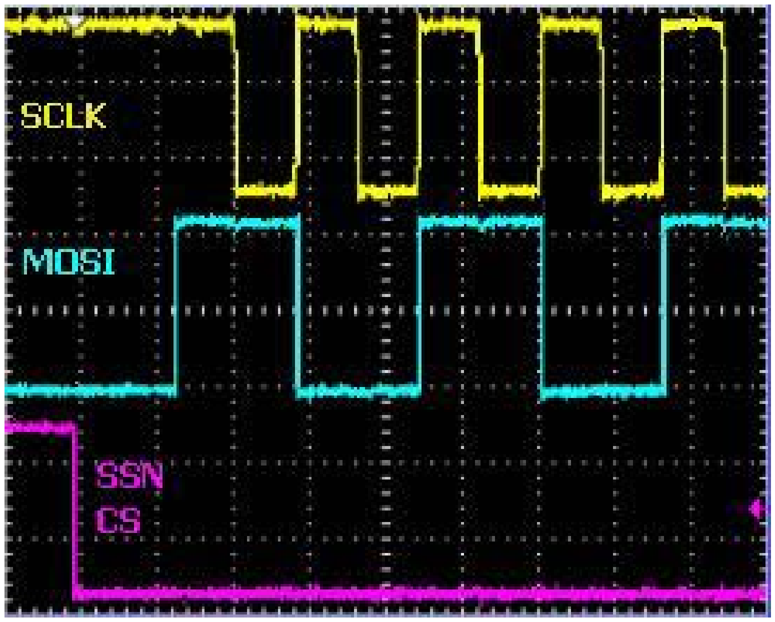
SPI MODE 0

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.



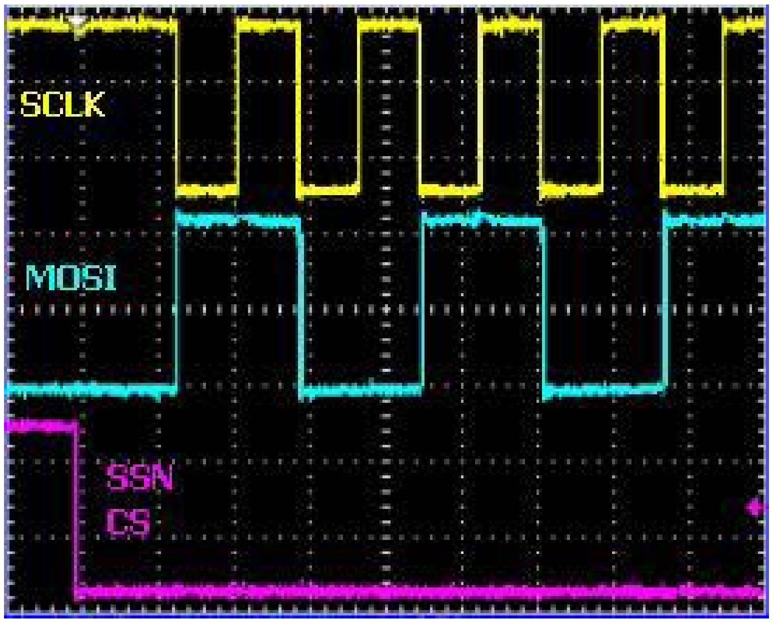
SPI MODE 1

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 USBtoI2C32.dll:

```
0x00: No error (transmission successful)
0x01: Address not Acknowledged
0x02: Data not Acknowledged
0x03: I2C bus error. A bus error is caused when a START or STOP condition
      occurs at an illegal position in the format frame.
0x04: Arbitration lost
0x08: I2C Time Out (I2C message not completed in 1 second).
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.
0x80: Unsupported function (make sure you have the latest firmware)
0xFF: Hardware not detected or USB error
```

TROUBLESHOOTING

- If you have problems with the software installation, ensure that you have Admin Privileges.

- If the installer application doesn't launch when the CD-ROM is inserted, then manually browse the CD-ROM and find the Setup.exe file in the root directory of the CD. Double-click on the Setup.exe file to start the installation process.

- The Adapter has 3.3k□pull-up resistors to 3.3V on the I^{2}C bus lines. Ensure that additional pull-up resistors in the user target system do not bring the pull-up resistance below 1.6k□□for 5V systems or 1.1K□for 3.3V systems. The on-board pull-up resistors can be removed from the system by removing the pull-up resistor jumpers RPU (see Hardware Description above).

- The USB-to-I2C Elite software will not function on Windows 95, Windows 98 (First edition), or NT systems; therefore, installation will not proceed if you attempt to install the software on these Operating Systems.

- Some development environments (for example, console applications and Visual Basic) do not automatically load the DLL when they run. The DLL is loaded when the first function is called. When the first function is called, the DLL must find the USB-to-I2C Elite devices on the USB bus. This takes a little time, so it is a good idea to call a function in the DLL that does not require the hardware. Get_DLL_Version is a good function to call since it loads the DLL but does not require the hardware to be found. You may need to add a small delay between the Get_DLL_Version and the first hardware call. Note that this procedure is not required by most development tools.

- USB-to-I2C Elite monitors the communications on the I^2C bus for proper operation of connected peripherals; any errors on the bus are detected and reported by the software. Bus communication is stopped if errors are detected and can be resumed when the (hardware) problem is corrected and the transmission retried.

- Keep the original USB-to-I2C Elite installation CD-ROM in the event the software needs to be re-installed. Future USB-to-I2C Elite updates from our Website may require a previous installation from the original media.

- Check for new versions of the software at http://www.i2ctools.com/downloads.html

- The firmware for the Elite hardware can be upgraded by the user, if the firmware loaded on the Elite hardware is version 2.0 or later. We will also upgrade the firmware for you if you have an earlier revision. The hardware will need to be returned to us for version 1.x. Please visit the i2ctools website to find the latest firmware information.

If all else fails, email a description of the problem you are having to us at support@i2ctools.com.
Note that all technical support requests must begin with an email to this email address.

We are interested in receiving feedback from our customers. Is there is a feature that should be added to make this tool better? Please send your requests and comments to support@i2ctools.com.