LPT-to-I2C SE Professional©

'I²C Made Simple'

I²C and SMBus Control DLL

User's Manual

Version 4

Date: November 1, 2008

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

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

 Revision 4 Nov-08

Table of Contents

I²C Protocol

General Characteristics

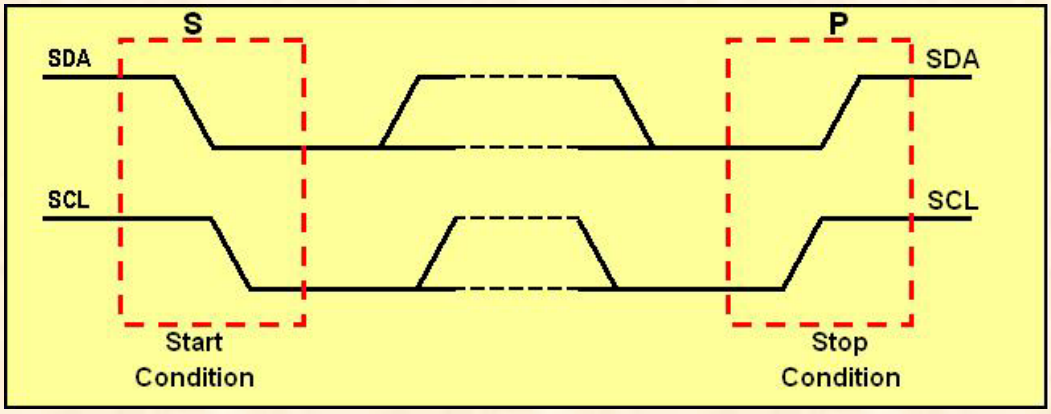
The I²C protocol allows data to be transferred between devices using two open-drain (or open-collector) bi-directional lines. One line is the serial clock (SCL) and the other is the serial data (SDA). The bus master generates the Start conditions, the clock signals on SCL, as well as the Stop condition.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

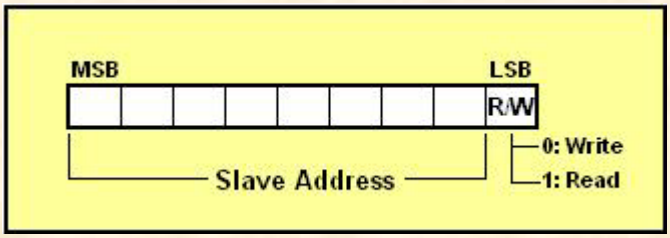
Start and Stop Conditions

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 I²C 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.
When transmitting an address using LPT-to-I2CSEpro.dll, the user should use the I2CWrite function and then ensure that the correct least significant bit has been appended (‘1’ for read, ‘0’ for write). See the Examples section for further information.

Subaddress

When an I²C device contains more than one register, the various registers are generally accessed using a subaddress that is sent following the device address (see the WriteArray and 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. Although the I²C specification does not specify a maximum wait state, LPT-to-I2C SE Professional has set a default maximum wait state length of approximately 50ms. This value may be changed by using the SetWaitTime function.

Acknowledge

The acknowledge related clock pulse is generated by the master (LPT-to-I2C SE Professional is always the bus masters). 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 I²C Bus specification can be found at http://www.nxp.com/products/interface_control/i2c/.

Minimum System Configuration

- ✓ PC with a Pentium 60 and 8MB RAM or better
- ✓ Windows 95, 98, ME, NT4, 2000, XP, XP64, Vista, and Vista64
- ✓ 6 MB of free HDD space
- ✓ CD ROM drive (used for installation only)
- ✓ Bi-directional Parallel Port (DB-25, LPT port) or PCI-based LPT add-in card

LPT-to-I2C SE Professional Contents

- LPT-to-I2C SE Professional installation CD ROM
- Parallel Port adapter with 256-byte EEPROM

Files installed for LPT-to-I2C SE Professional

- LPT-to-I2CSEpro.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
- LPT-toI2C SE Professional User's Manual (this document)
- Visual C++, Delphi, C++ Builder, and Visual Basic example files
- LPT-to-I2C SE.exe application
- LPT-to-I2C SE Software User's Manual.pdf
- `TVICHW32.SYS` - installed in the Windows\System32 directory
- `TVICHW32.VXD` - installed in the Windows\System directory
- TVicHW32.DLL – installed for all operating systems
- Getting Started with LPT-to-I2C SE.pdf
- LPT-to-I2C SE Hardware User's Manual.pdf
- Software license agreement (`license.txt`)
- Registration Form (`RegFile.txt`)

Location of DLL

The LPTtoI2CSEpro.dll is placed in the Windows\System32 directory during installation.

Testing the Installation

After LPT-to-I2C SE Professional has been installed on your hard disk, the installation of the driver can be tested with the included LPT-to-I2C SE application. The hardware should be inserted into an available parallel port, and then the application can be started. If the installation was successful, you should be able to read and write from the on-board EEPROM using the 256x8 EEPROM from the LPT-to-I2C SE device pull-down menu.

Note that when installing the software to a Win32 or Win64 system, you must have Administrator privileges or the parallel port drivers will not be loaded correctly. After the software has successfully been installed, normal user privileges can be restored.

Users who do not install with Administrator privileges commonly encounter a 'Privileged Instruction' error. If you see this error message, please log back on as the Administrator and reinstall the software.

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 LPTtoI2CSEpro.dll) to clean up the stack.

AddLPTPort

This function takes the LPT address and returns an LPT number. The AddLPTPort function is only needed when a PCI-based parallel port add-in card needs to be used, rather than the standard parallel ports found on a PC. The LPTaddress can be found in Device Manager within Windows Control Panel.

```
C++:     unsigned char AddLPTPort (short int LPTaddress)
Delphi:  AddLPTPort(LPTaddress: word): byte;
VB:      AddLPTPort (ByVal LPTaddress As Integer) As Byte
```

CheckDriverStatus

This function takes no argument and returns the current state of the hardware driver. The result is true (non-zero) when the driver is functioning normally and a '0' (false) is if the driver is not started or cannot be started. Use this function to ensure that the driver is active before attempting any I²C communications.

```
C++:    short int CheckDriverStatus(void);
Delphi: CheckDriverStatus: WordBool;
VB:     CheckDriverStatus() As Boolean
```

GetAccessMode

The GetAccessMode function is available for backwards compatibility only. The function always returns false which indicates 'Slow' access.

```
C/C++:  short int GetAccessMode(void);
Delphi: GetAccessMode: WordBool;
VB:     GetAccessMode() As Boolean
```

GetI2CFrequency

This function takes no arguments and returns the current I²C clock frequency.

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

GetLPTAddress

The computer's parallel ports have a physical address that can be found using the GetLPTAddress function. This function takes no arguments and returns a two byte unsigned integer containing the LPT address. Using this function is not required but is available for the user's information.

```
C/C++:  short int GetLPTAddress(void);
Delphi: GetLPTAddress: word;
VB:     GetLPTAddress() As Integer
```

GetLPTNumber

The GetLPTNumber function takes no arguments and returns the value of the currently selected parallel port.

The function returns 1, 2, or 3, corresponding to LPT1, LPT2, and LPT3, respectively.

```
C/C++:  unsigned char GetLPTNumbr(void);
Delphi: GetLPTNumber: byte;
VB:     GetLPTNumber() As byte
```

GetMaxFrequency

The GetMaxFrequency function returns the maximum I²C clock frequency possible with the user's computer hardware. The value is hardware dependent.

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

HardwareDetect

This function checks to see if the LPT-to-I2C SE hardware is attached to the currently selected parallel port. The two byte Boolean value returned by HardwareDetect contains true (non-zero) if hardware was detected while the return value is false ('0') if the hardware was not detected. It is important to ensure that hardware is detected since no I²C communications will begin until hardware has been detected. Therefore, after calling the SetLPTNumber function, it is a good idea to call HardwareDetect to see if the hardware was detected.

```
C/C++:  short int HardwareDetect(void);
Delphi: HardwareDetect: wordbool;
VB:     HardwareDetect() As Boolean
```

I2CRead

This function takes a two-byte Boolean value and a pointer to data byte and then reads one byte from a device on the I²C bus. The Boolean value indicates whether the byte will be the last byte read from the I²C bus. A true (non-zero) indicates that it is the last byte to read while a false ('0') indicates that additional bytes will be read. The returned value contains the error condition. See Error Codes section for return values. The data byte is written to the memory location specified by ReadData.

```
C/C++:  unsigned char I2CRead(short int LastByte, unsigned char *ReadData);
Delphi: I2CRead(LastByte: wordbool; var ReadData: byte): byte;
VB:     I2CRead(ByVal LastByte As Boolean, ByRef ReadData As Byte) As Byte
```

I2CReadBit

This function reads one bit from the I²C bus and places it in the least significant bit of the memory location specified by 'ReadData'. The returned value contains the error condition. See Error Codes section for return values.

These bit functions (I2CWriteBit and I2CReadBit) are useful for testing for error conditions. For example, how does a device on the bus react to a misplaced Stop or Start condition? This can be simulated by sending out a Start, followed by four data bits, and then another Start or Stop condition.

An example using I2CReadBit and I2CWriteBit is included in the installation package.

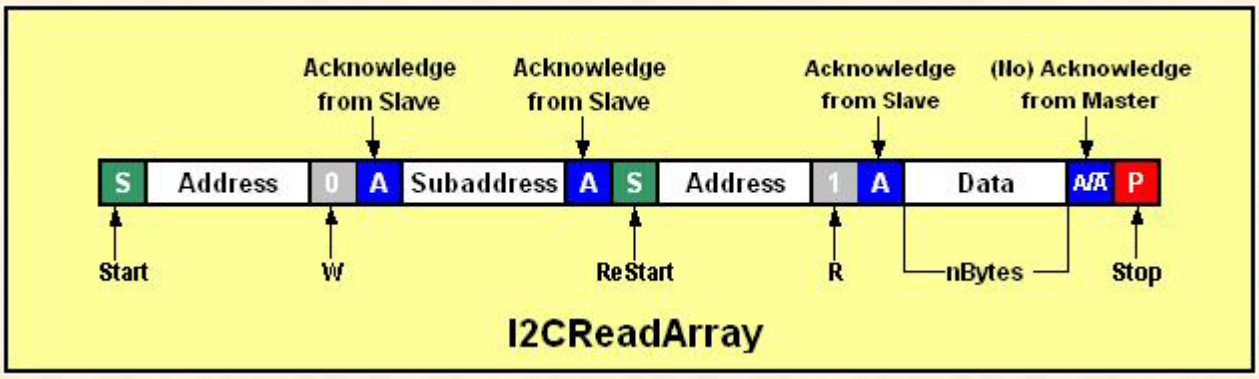
```
C/C++:  unsigned char I2CReadBit(unsigned char *ReadData);
Delphi: I2CReadBit(var ReadData: byte): byte;
VB:     I2CReadBit(ByRef ReadData As Byte) As Byte
```

I2CReadArray

The I2CReadArray function takes four arguments: the device address, the device subaddress, the number of bytes to read, and a pointer to an element within an array of bytes. I2CReadArray sends the I²C message shown below and returns any error condition it encounters. It is the calling program's responsibility to allocate

the correct memory space for the array. The function ensures that the lsb of the address is appropriate ('1' or '0' depending on Write or Read) before it is sent to the target device.

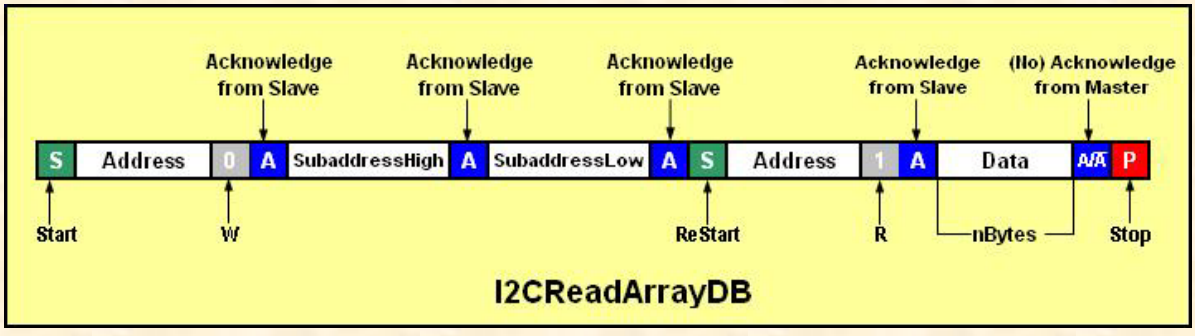
```
C/C++:  unsigned char I2CReadArray(unsigned char address,
        unsigned char subaddress, int nBytes, unsigned char *ReadData);
Delphi: I2CReadArray(address,subaddress: byte; nBytes: integer;
        var ReadData): byte;
VB:     I2CReadArray(ByVal address As Byte, ByVal subaddress As Byte, nBytes As
        Long, ByRef ReadData As Byte) As Byte
```



I2CReadArrayDB

The I2CReadArrayDB function takes five arguments: the device address, the high byte of the device subaddress, the low byte of the device subaddress, the number of bytes to read, and a pointer to an element within an array of bytes. I2CReadArrayDB sends the I²C message shown below and returns any error condition it encounters. It is the calling program's responsibility to allocate the correct memory space for the array. The function ensures that the lsb of the address is appropriate ('1' or '0' depending on Write or Read) before it is sent to the target device.
This function is useful for communicating with large eeproms that use a two byte subaddressing scheme.

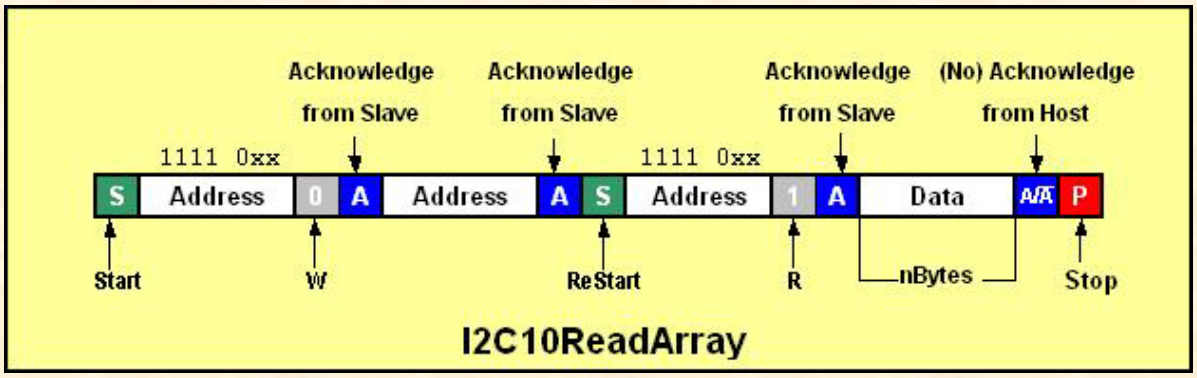
```
C/C++:  uchar I2CReadArrayDB(uchar address, uchar subaddressHigh, uchar
        subaddressLow, int nBytes, uchar *ReadData);
Delphi: I2CReadArrayDB(address, subaddressHigh, subaddressLow: byte;
        nBytes: integer; var ReadData: byte): byte;
VB:     I2CReadArrayDB(ByVal address As Byte, ByVal subaddressHigh As Byte, ByVal
        subaddressLow As Byte, ByVal nBytes As Long,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 I²C addressing. The I²C 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 I²C compliant format. A subaddress is also sent after the second byte of the address (not shown in diagram below).

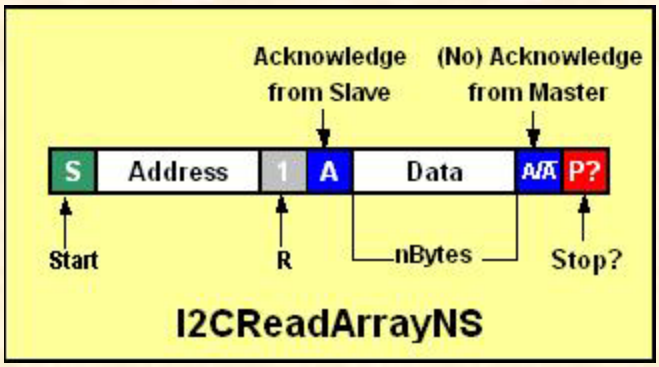



```
C/C++:  unsigned char I2C10ReadArray(short int address,
        unsigned char subaddress, int nBytes, unsigned char *ReadData);
Delphi: I2C10ReadArray(address: word; subaddress: byte; nBytes: integer;
        var ReadData): byte;
VB:     I2C10ReadArray(ByVal address As Integer, ByVal subaddress As Byte,
        nBytes As Long, ByRef ReadData As Byte) As Byte
```

I2CReadArrayNS

The I2CReadArrayNS 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 three arguments: the device address, 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 lsb of the address is set to a '1' before it is sent to the target device.

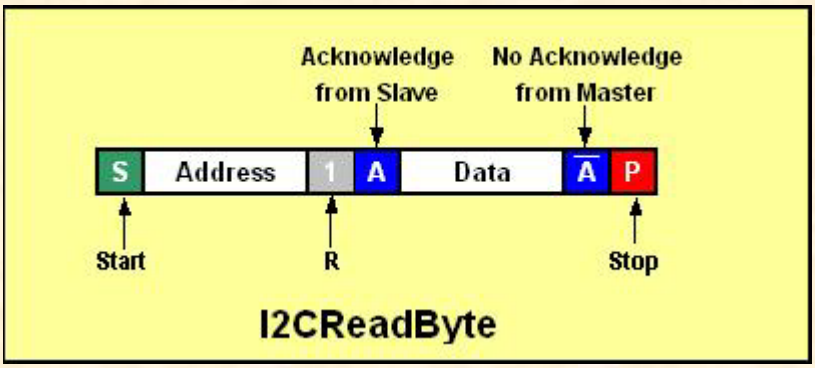
```
C/C++:  unsigned char I2CReadArrayNS(unsigned char address, int nBytes,
        unsigned char *ReadArray);
Delphi: I2CReadArrayNS(address:byte; nBytes: integer; var ReadData: byte): byte;
VB:     I2CReadArrayNS(ByVal address As Byte, ByVal nBytes As Long, ByRef
        ReadData As Byte) As Byte
```



I2CReadByte

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

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



I2CStart

This function takes no arguments and generates an I²C Start Condition via the parallel port. The function returns any error condition it encounters. If the hardware has not been detected, the Start condition will not be performed. See Error Code section for return values.

```
C/C++:  unsigned char I2CStart(void);
Delphi: I2CStart: byte;
VB:     I2CStart() As Byte
```

I2CStop

This function generates an I²C Stop Condition via the parallel port. The function returns any error condition it encounters during the transmission. See Error Code section for return values.

```
C/C++:  unsigned char I2CStop(void);
Delphi: I2CStop: byte;
VB:     I2CStop() As Byte
```

I2CWrite

This function writes one byte, passed by the calling program, to the I²C Bus via the parallel port. The function returns any error condition it encounters during the transmission. See Error Code section for return values.

```
C/C++:  unsigned char I2CWrite(unsigned char DataByte);
Delphi: I2CWrite(DataByte: byte): byte;
VB:     I2CWrite(ByVal DataByte As Byte) As Byte
```

I2CWriteBit

This function takes on byte and writes the most significant bit to the I²C bus. The returned value contains the error condition. See Error Codes section for return values.
These bit functions (I2CWriteBit and I2CReadBit) are useful for testing for error conditions. For example, how does a device on the bus react to a misplaced Stop or Start condition? This can be simulated by sending out a Start, followed by four data bits, and then another Start or Stop condition.
An example using I2CReadBit and I2CWriteBit is included in the installation package.

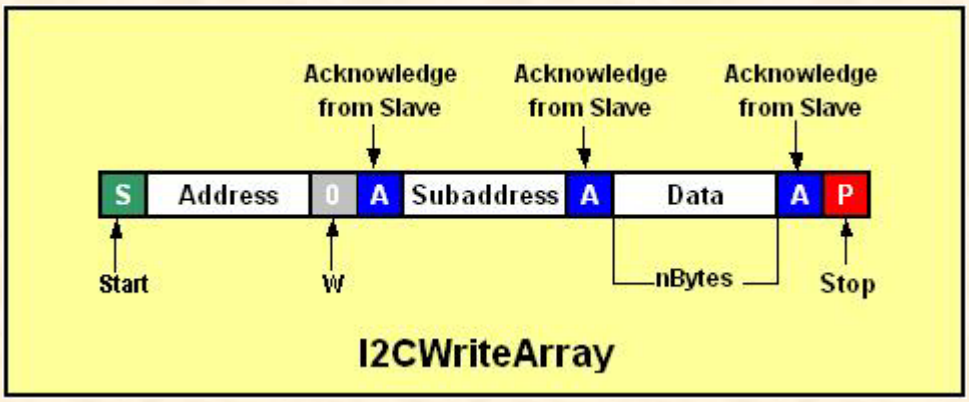
```
C/C++:  unsigned char I2CWriteBit(unsigned char DataByte);
```

```
Delphi: I2CWriteBit(DataByte: byte): byte;
VB:     I2CWriteBit(ByVal DataByte 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. It is up to the calling program to correctly define the array of memory to store the data. 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. See Error Code section for return values.

```
C/C++:  unsigned char I2CWriteArray(unsigned char address, unsigned char
        subaddress; int nBytes; unsigned char *WriteData);
Delphi: I2CWriteArray(address,subaddress: byte; nBytes: integer;
        var WriteData: byte): byte;
VB:     I2CWriteArray (ByVal address As Byte, ByVal subaddress As
        Byte, ByVal nBytes As Long, ByRef WriteData As Byte) As Byte
```

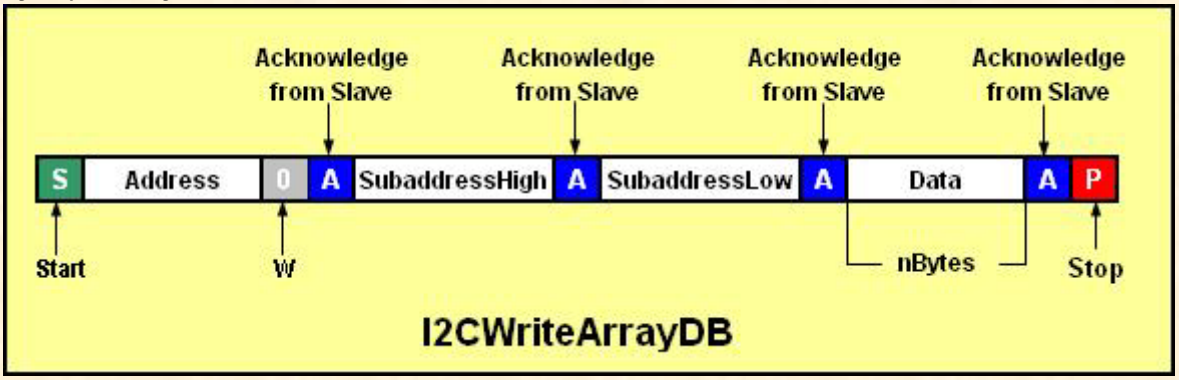


I2CWriteArrayDB

The I2CWriteArrayDB (write array with a double-byte subaddress) takes five parameters: device address, the high byte of the device subaddress, the low byte of the 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. See Error Code section for return values.

This function is useful for communicating with devices such as large eeproms that use a two byte subaddressing scheme.

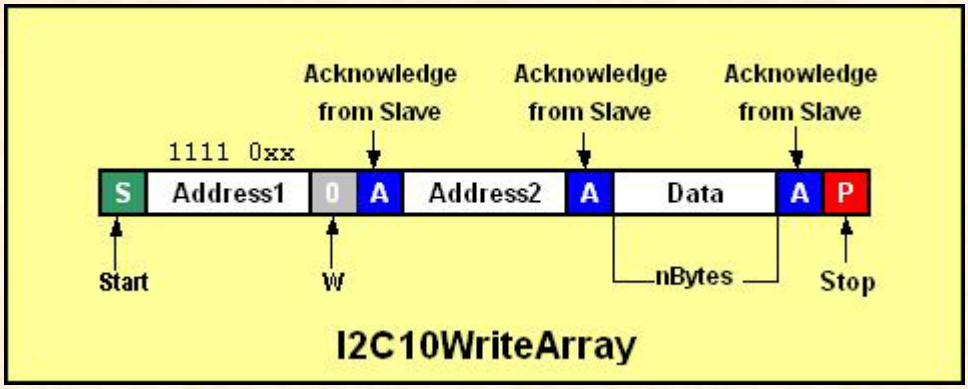
```
C/C++:  uchar I2CWriteArrayDB(uchar address, uchar subaddressHigh, uchar
        subaddressLow, int nBytes, uchar *WriteData);
Delphi: I2CWriteArrayDB(address, subaddressHigh, subaddressLow: byte; nBytes:
        integer; var WriteData: byte): byte;
VB:     I2CWriteArrayDB (ByVal address As Byte, ByVal subaddressHigh As Byte,
        ByVal subaddressLow As Byte, ByVal nBytes As Long, 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 I²C addressing. The I²C 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 I²C compliant format. A subaddress is also sent after the second byte of the address (not shown in diagram below), followed by the data.

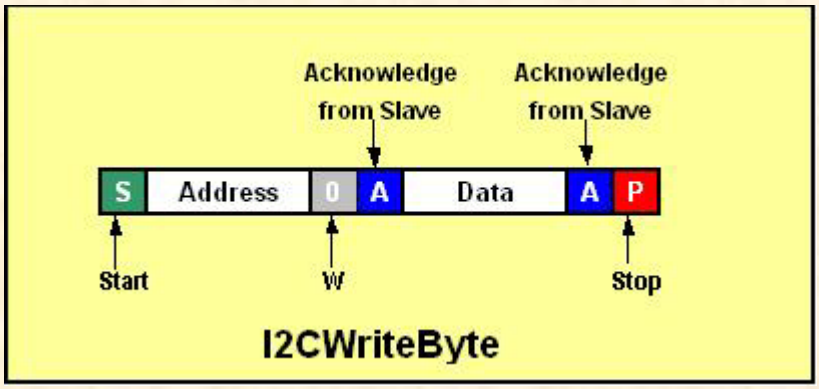
```
C/C++:  unsigned char I2C10WriteArray(short int address, unsigned char
        subaddress, int nBytes, unsigned char *WriteData);
Delphi: I2C10WriteArray(address: word; subaddress: byte; nBytes: integer;
        var WriteData: byte): byte;
VB:     I2C10WriteArray(ByVal address As Integer, ByVal subaddress As Byte,
        ByVal nBytes As Long, ByRef WriteData As Byte) As Byte
```



I2CWriteByte

The I2CWriteByte function writes one data byte to an I²C bus device. The function takes two parameters: the device address and a single data byte and returns any error condition it encounters (see Error Codes section). The function ensures that the lsb of the address is a '0' before it is sent to the target device.

```
C/C++:  unsigned char I2CWriteByte(unsigned char address, unsigned char Data);
Delphi: I2CWriteByte(address, Data: byte): byte;
VB:     I2CWriteByte(ByVal address As Byte, ByVal Data As Byte) As Byte
```



InputState

Reads the state of the IN port on the LPT-to-I2C SE Professional hardware Adapter. The function returns a '1' if high and a '0' if low.

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

milliseconds

The milliseconds delay function allows the user to program a delay, measured in milliseconds, into I²C messages. This is particularly useful when programming EEPROM devices that require a minimum erase/write time between write transactions.

```
C/C++:  void milliseconds(int Data);
Delphi: milliseconds(Data: integer);
VB:     milliseconds(ByVal Data As Long)
```

Trigger

Sends a high or low level to the OUT pin of the LPT-to-I2C SE Professional Adapter. The function takes a '1' for a high level and a '0' for a low level. Valid values are '0' and '1'. All other values will be ignored.

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

PulseLow

Sends a short low level pulse on the OUT pin of the LPT-to-I2C SE Professional Adapter followed by a high level. The function does not check the intial state of the OUT to ensure it is high before beginning.

```
C/C++:  void Trigger(void);
Delphi: PulseLow;
VB:     PulseLow()
```

PulseHigh

Sends a short high level pulse on the OUT pin of the LPT-to-I2C SE Professional Adapter followed by a low level. The function does not check the intial state of the OUT to ensure it is low before beginning.

```
C/C++:  void PulseHigh(void);
Delphi: PulseHigh;
VB:     PulseHigh()
```

SCL_Control

Allows the user to change the state of the SCL pin. The state of the SCL pin is returned.

```
C/C++:  short int SCL_Control(short int SCL_state);
Delphi: SCL_Control(SCL_state: wordbool): wordbool;
VB:     SCL_Control (ByVal SCL_state As Boolean) As Boolean
```

SDA_Control

Allows the user to change the state of the SDA pin. The state of the SDA pin is returned.

```
C/C++:  short int SDA_Control(short int SDA_State);
Delphi: SDA_Control(SDA_State: wordbool): wordbool;
VB:     SDA_Control (ByVal SDA_state As Boolean) As Boolean
```

SetNormalMode

The SetNormalMode function is only available for backward compatibility. Only 'Slow' access mode is available.

```
C/C++:  short int SetNormalMode(void);
Delphi: SetNormalMode: wordbool;
VB:     SetNormalMode() As Boolean
```

SetI2CFrequency

This function sets the I²C clock frequency to the value passed by the user's program. The frequency must be a positive integer. If a frequency is selected which is above the maximum frequency (use GetMaxFrequency to determine the value), the dll will set the frequency to the maximum I²C frequency. The function returns the measured frequency.
It is important to note that although we expect better than 5% tolerance on the frequency, the actual frequency generated by LPT-to-I2C SE Professional cannot be guaranteed. If an accurate frequency is needed, it is recommended that the frequency be verified using test equipment.

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

SetLPTNumber

If you have more than one parallel port in your computer, you can choose which parallel port to communicate with, by using the SetLPTNumber function. The function allows you to choose values 1, 2, or 3 corresponding to LPT1, LPT2, and LPT3 respectively. The 16-bit Boolean return value is 'true' ('non-zero') if the parallel port was detected and set to the value passed to the function, while it returns 'false' ('0') if the chosen parallel port was not available. The dll initialization routine attempts to set the active parallel port to LPT1.

```
C/C++:  short int SetLPTNumber(unsigned char LPT);
Delphi: SetLPTNumber(LPT: byte): wordbool;
VB:     SetLPT (ByVal LPT As Byte) As Boolean
```

SetSlowMode

The SetSlowMode function is only available for backward compatibility. Only 'Slow' access mode is available. 'Normal' access is no longer supported.

```
C/C++:  short int SetSlowMode(void);
Delphi: SetSlowMode: wordbool;
VB:     SetSlowMode() As Boolean
```

SetWaitTime

The I²C Bus specification does not specify the amount of time that a device can hold the clock line low to inject a wait/hold state in a transmission. In order to ensure that the software responds in a predictable manner, LPTotI2CSEpro.dll has arbitrarily set the maximum wait time between the time it releases the clock and the time that any device holding the clock line low must release the bus as 50ms. If this time is exceeded, the dll will exit the function. The SetWaitTime function allows the user to change the maximum wait/hold times for the situation where a slow device needs more than the 50ms initialized by the dll. The function returns the value set by the function. The minimum time allowed by the function is 5ms.

```
C/C++:  int SetWaitTime(int NewWaitTime);
Delphi: SetWaitTime(NewWaitTime: integer): integer;
VB:     SetWaitTime(ByVal NewWaitTime As Long) As Long
```

StartI2CDriver

Loads the virtual device driver or the kernel mode device driver, providing direct access to the LPT ports. If the driver was successfully started, the function returns True; if the function fails it returns False.
The StartI2CDriver function does not actually need a value of '0' or '1' sent to it, as previously required. This is only included for backwards compatibility. Any integer value may be sent.

```
C/C++:  short int StartI2CDriver(int device);
Delphi: StartI2CDriver(device: integer): wordbool;
VB:     StartI2CDriver(ByVal device As Long) As Boolean
```

StopI2CDriver

Closes the kernel-mode driver and releases memory allocated to it. It is highly recommended that this function be called before an application is terminated.

```
C++:    void StopI2CDriver(void);
Delphi: StopI2CDriver;
VB:     StopI2CDriver()
```

Error Codes

The following error codes are returned by the various functions in LPT-to-I2CSEpro.dll:

0x00: No error
0x01: Address not acknowledged (only valid for I2CWriteByte, I2CReadByte, I2CWriteArray, and I2CReadArray functions)
0x02: Acknowledge not received
0x03: Read acknowledge corrupted – should be a '1' but a '0' was found
0x04: SCL/SDA stuck low – both lines found low while they should be high
0x08: SDA stuck low – SDA line could not be set to a logic '1'
0x09: SDA stuck high – SDA line could not be set to a logic '0'
0x0A: SCL stuck high – SCL line could not be set to a logic '0'
0x0B: SDA and SCL stuck high – both SDA and SCL could not be set low
0x80: SCL stuck low – SCL line could not be set to a logic '1'
0xFF: Hardware not detected

Examples

Visual Basic Example

This example writes two bytes to the EEPROM, located on the LPT-to-I2C SE hardware, and then reads them back.

In order to use the dll functions, they must be imported into the calling program.
Add the Module1.bas file (included with LPT-to-I2CSEpro.dll) into your project.
See the Visual Basic example included with LPTtoI2C SE Professional.

The following code example writes two bytes to an EEPROM, and then reads them back. The user should ensure that the error codes returned from the functions are handled appropriately.

```
Private Sub Command1_Click()
Dim ErrorCode As Byte
Dim ReadData As Byte
Dim Init As Boolean
Frequency As Integer

Init=CheckDriverStatus    'Check to ensure that the hardware driver has been
Init=HardwareDetect       'loaded successfully and the hardware is detected.
Init=SetLPTNumber(1)      'Set the LPT Number.
Frequency=SetI2CFrequency(50) 'Set the I²C clock frequency to 50KHz.

ErrorCode=I2CStart        'Generate I²C Start Condition.
ErrorCode=I2CWrite(160)   'Send the EEPROM address…in this case 0xA0.
ErrorCode=I2CWrite(0)     'Send the subaddress.
ErrorCode=I2CWrite(0)     'Send first data byte.
ErrorCode=I2CWrite(1)     'Send second data byte.
ErrorCode=I2CStop         'Generate I²C Stop Condition.

milliseconds(10)          'wait 40ms for the erase/write cycle to complete.

ErrorCode=I2CStart        'Generate I²C Start Condition.
ErrorCode=I2CWrite(160)   'Send the EEPROM write address…in this case 0xAE.
ErrorCode=I2CWrite(0)     'Send the subaddress.
ErrorCode=I2CStart        'Generate I²C Start Condition.
ErrorCode=I2CWrite(161)   'Send the EEPROM read address… in this case 0xAF.
ErrorCode=I2CRead(false,ReadData) 'Read one byte (not last byte).
Label1.Caption=Int(ReadData)
ErrorCode=I2CRead(true,ReadData)  'Read one byte (last byte).
Label2.Caption=Int(ReadData)
ErrorCode=I2CStop         'Generate I²C Stop Condition.
End Sub
```

Delphi Example

This example writes two bytes to an EEPROM, located on the LPT-to-I2C SE hardware, and then reads them back. Before writing or reading, it is best to go through an initialization process to ensure everything is functioning correctly. This example code shows the minimum functionality and it is up to the user to ensure the returned error codes are handled appropriately.

In order to use the dll functions, they must be imported into the calling program. The easiest way to do this is to add the I2Cdeclarations.pas file (included with LPT-to-I2C SE Pro) to your project by placing this file in the same folder as your project and then using the 'Add to Project' menu item from the 'Project' menu in Delphi. You must also add 'I2Cdeclarations' statement in the **uses** clause of your application (see example in the Delphi folder which was installed with LPT-to-I2C SE Professional).

```
procedure TDLLForm.btnWriteandRead(Sender: TObject);
var
ErrorCode, ReadData: byte;
InitOK: WordBool;
Freq: integer;
begin
if CheckDriverStatus then  //Check to ensure that the hardware driver has been
begin                      //loaded successfully.
InitOK:=HardwareDetect;    //Check to ensure that the hardware is detected.
InitOK:=SetLPTNumber(1);   //Set the LPT Number.
Freq:=SetI2CFrequency(50); //Set the I²C clock frequency to 50KHz.
                           //It is not necessary to set the frequency
                           //since it defaults to 10KHz at start-up
ErrorCode:=I2CStart;       //Generate I²C Start Condition.
ErrorCode:=I2CWrite($A0);  //Send the EEPROM address…in this case 0xA0.
ErrorCode:=I2CWrite($00);  //Send the subaddress.
ErrorCode:=I2CWrite($00);  //Send first data byte.
ErrorCode:=I2CWrite($01);  //Send second data byte.
ErrorCode:=I2CStop;        //Generate I²C Stop Condition.
milliseconds(10);          //wait 10ms for the erase/write cycle to complete.
ErrorCode:=I2CStart;       //Generate I²C Start Condition.
ErrorCode:=I2CWrite($A0);  //Send the EEPROM write address…in this case 0xA0.
ErrorCode:=I2CWrite($00);  //Send the subaddress.
ErrorCode:=I2CStart;       //Generate I²C Start Condition.
ErrorCode:=I2CWrite($A1);  //Send the EEPROM read address… in this case 0xA1.
ErrorCode:=I2CRead(false,ReadData);//Read one byte from the EEPROM, not last byte.
Label1.Caption := IntToStr(ReadData);
ErrorCode:=I2CRead(true);  //Read one byte from the EEPROM, last byte.
Label2.Caption := IntToStr(ReadData);
ErrorCode:=I2CStop;        //Generate I²C Stop Condition.
end
else ShowMessage('Driver not started');
end;
```