

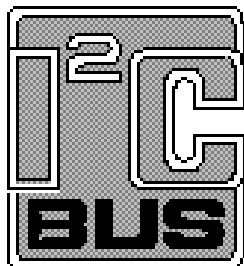
Win-I2CNTDLL

I²C and SMBus Control DLL

User's Manual

Version 4.2

Date: November 28, 2002



Information provided in this document is solely for use with Win-I2CNTDLL. SB Solutions reserves the right to make changes or improvements to this document at any time without notice. We assume no liability whatsoever in the sale or use of this product, including infringement of any patent or copyright. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of SB Solutions.

Microsoft Visual Basic, Windows and Windows NT are registered trademarks of Microsoft Corporation. Other brand names are trademarks or registered trademarks of their respective owners.

Delphi is a trademark of Borland International, Inc.

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

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

Table of Contents

<u>I2C PROTOCOL</u>	5
GENERAL CHARACTERISTICS.....	5
BIT TRANSFER.....	5
START AND STOP CONDITIONS	5
I2C ADDRESS.....	5
SUBADDRESS.....	5
DATA TRANSFER.....	6
ACKNOWLEDGE.....	6
I2C BUS DOCUMENTATION.....	6
<u>MINIMUM SYSTEM CONFIGURATION</u>	7
<u>WIN-I2CNTDLL CONTENTS</u>	7
FILES INSTALLED SPECIFICALLY FOR WIN-I2CNTDLL.....	7
FILES INSTALLED SPECIFICALLY FOR WIN-I2CNT	7
COMMON FILES INSTALLED FOR WIN-I2CNTDLL AND WIN-I2CNT:	7
<u>LOCATION OF DLL</u>	7
<u>TESTING THE INSTALLATION</u>	7
<u>UPGRADING FROM EARLIER VERSIONS</u>	8
WHY UPGRADE?	8
UPGRADING FROM WIN-I2CNTDLL V3.2 AND EARLIER TO WIN-I2CNTDLL V4.X.....	8
<u>EXPORTED FUNCTIONS USING THE STDCALL CONVENTION</u>	9
CHECKDRIVERSTATUS.....	9
CLOSEI2CDRIVER	9
GETACCESSMODE	9
GETI2CFREQUENCY.....	9
GETLPTADDRESS	9
GETLPTNUMBER	10
GETMAXI2CFREQUENCY	10
HARDWAREDETECT	10
I2C_READ	10
I2C_READARRAY	10
I2C10_READARRAY	11
I2C_READARRAYNS	11
I2C_READBYTE	12
I2C_START.....	12
I2C_STOP	12
I2C_WRITE	12

Win-I2CNTDLL

I2C_WRITEARRAY.....	12
I2C10_WRITEARRAY.....	13
I2C_WRITEBYTE	13
MILLISECONDS.....	14
OPENI2CDRIVER.....	14
SETNORMALACCESSMODE	14
SETI2CFREQUENCY	14
SETLPTNUMBER.....	15
SETSLOWACCESSMODE	15
SETWAITTIME.....	15

EXPORTED FUNCTIONS USING A C TYPE CALLING CONVENTION (CDECL)16

CHECKDRIVER	16
GET_ACCESSMODE	16
GETADDRESS	16
GETFREQUENCY.....	16
GETLPT	16
GETMAXFREQUENCY	16
HWDETECT.....	16
I2CREAD	17
I2CSTART.....	17
I2CSTOP	17
I2CWRITE	17
MSDELAY	17
READARRAY	17
READARRAYNS	18
READBYTE.....	18
SETFREQUENCY	18
SETLPT	19
SETNORMALMODE	19
SETSLOWMODE	19
SETWAIT	19
STARTI2CDRIVER	19
STOPI2CDRIVER	20
WRITEARRAY.....	20
WRITEBYTE	20

ERROR CODES21

EXAMPLES22

VISUAL BASIC EXAMPLE	22
DELPHI EXAMPLE.....	23

I²C Protocol

General Characteristics

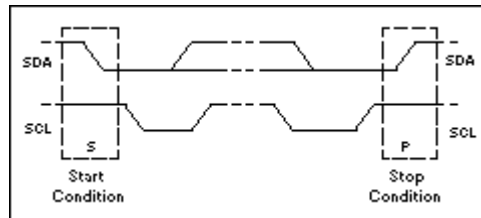
The I²C protocol allows data to be transferred between devices using two open-drain (or open-collector) bi-directional lines. One line is the serial clock (SCL) and the other is the serial data (SDA). The bus master generates the Start conditions, the clock signals on SCL, as well as the Stop condition.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

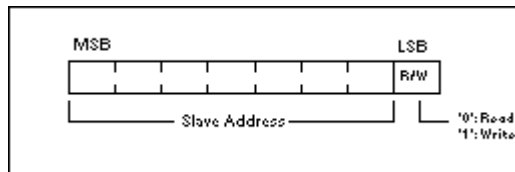
Start and Stop Conditions

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 Win-I2CNTDLL, the user should use the I2CWrite or I2C_Write functions 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, Win-I2CNTDLL has set an arbitrary maximum wait state length of approximately 50ms.

Acknowledge

The acknowledge related clock pulse is generated by the master (Win-I2CNT and Win-I2CNTDLL are 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.philipslogic.com/products/collateral/i2c/>.

Minimum System Configuration

- ✓ PC with a Pentium 60 and 8MB RAM or better
- ✓ Windows 95, 98, ME, NT3.5, NT4, 2000, or XP
- ✓ 6 MB of free HDD space
- ✓ CD ROM drive (used for installation only)
- ✓ Bi-directional Parallel Port (DB-25, LPT port)

Win-I2CNTDLL Contents

- Win-I2CNTDLL installation CD ROM
- Parallel Port adapter (I2CPORT2.0) with 256-byte EEPROM

Files installed specifically for Win-I2CNTDLL

- `sbsi2c4.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
- Win-I2CNTDLL User's manual (`Win-I2CNTDLL v4 User's Manual.PDF`; this document)
- Visual C++, Delphi, and Visual Basic example files

Files installed specifically for Win-I2CNT

- Win-I2CNT.exe application
- Win-I2CNT Installation and User's Guide (`Win-I2CNT UG.pdf`; Acrobat version)
- Win-I2CNT Software User's Manual (`Win-I2CNT Software User's Manual v4.pdf`)
- Help file (`I2C.hlp`)

Common files installed for Win-I2CNTDLL and Win-I2CNT:

`TVICHW32.SYS` - installed in the `WinNT\system32` directory (Windows NT, 2000, XP only)
`TVICHW32.VXD` - installed in the `Windows\System` directory (Windows 95, 98, ME only)
`TVicHW32.DLL` - installed for all operating systems
Parallel Port Adapter schematic (`I2cprt20.pdf`)
Software license agreement (`license.txt`)
Win-I2CNT user Registration Form (`Regfrm4x.txt`)

Location of DLL

The `sbsi2c4.dll` is placed in the `Windows\System` directory during installation. The `sbsi2c4.dll` may be moved to the same directory as the application using the dll.

Testing the Installation

After Win-I2CNTDLL has been installed on your hard disk, the installation of the driver can be tested with the included Win-I2CNT application. The I2CPort2.0 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 Win-I2CNT Device pull-down menu.

Note that when installing the software to a Windows NT, 2000, or XP 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.

Upgrading from earlier versions

Why upgrade?

- Win-I2CNTDLL v4.x adds support for Windows XP
Win-I2CNTDLL V3.2 and earlier versions did not support Windows XP.
- Three new functions were added to Win-I2CNTDLL v4.x:
 - SetWaitTime - allows the user to increase the wait/hold times for slow I²C peripherals
 - I2C10_WriteArray (10-bit addressing)
 - I2C10_ReadArray (10-bit addressing)

Upgrading from Win-I2CNTDLL V3.2 and earlier to Win-I2CNTDLL V4.x

1. Uninstall the previous version of Win-I2CNTDLL, if desired
2. Install Win-I2CNTDLL V4.x
3. Your existing projects will need minor modifications to work with v4.x depending on your development software:

Visual C++

- replace the sbsi2c.h file with sbsi2c4.h
- replace the sbsi2c.lib file with sbsi2c4.lib if using stdcall calling convention or cdecl4.lib if using cdecl convention
- these files can be found in the Examples folder

Delphi

- add the file I2Cdeclarations.pas in your project
- the I2Cdeclarations.pas file can be found in the Delphi example directory

Visual Basic

- add the module sbsi2c4.bas in your project
- the sbsi2c4.bas file can be found in the VB example directory

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 sbsi2c4.dll) to clean up the stack.

CheckDriverStatus

This function takes no argument and returns the current state of the hardware driver. If the return value is '1' (true) when the driver is functioning and a '0' (false) is returned if the driver is not 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
```

CloseI2CDriver

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 CloseI2CDriver(void)
Delphi:   CloseI2CDriver
VB:      CloseI2CDriver()
```

GetAccessMode

The GetAccessMode function determines whether the kernel-mode driver is using 'Normal' or 'Slow' access to the I/O ports. If true, 'Normal' access is in use while a false indicates that 'Slow' access is in use. 'Normal' access provides higher performance access to ports, but may fail if the port(s) addressed is already in use by another kernel-mode driver. While slower, 'Slow' access provides more reliable access to ports that have already been opened by another kernel-mode driver.

```
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
```

GetMaxI2CFrequency

The GetMaxI2CFrequency function returns the maximum I²C clock frequency possible with the user's computer hardware. The value is hardware dependent.

```
C/C++:   int GetMaxI2CFrequency(void)
Delphi:  GetMaxI2CFrequency: integer;
VB:     GetMaxI2Cfrequency() As Long
```

HardwareDetect

This function checks to see if the I2CPort2.0 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 Win-I2CNT hardware was detected.

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

I2C_Read

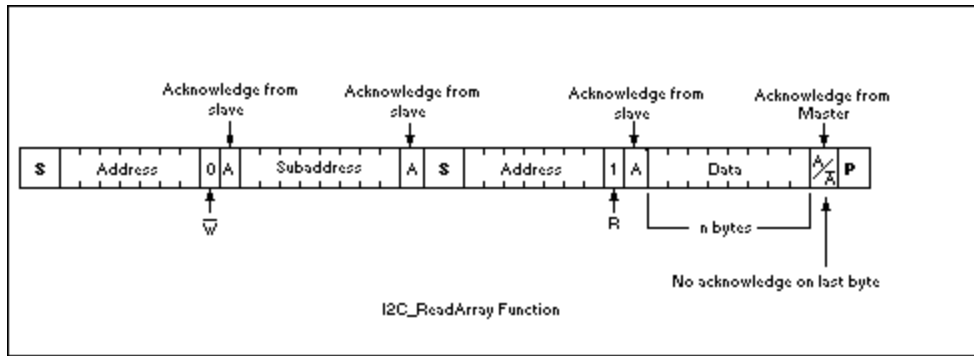
This function takes a two byte boolean value and a pointer to data byte and then reads one byte from the I²C bus. The boolean value indicates whether the byte will be the last byte read from the I²C bus. A true ('1') indicates that it is the last byte to read while a false ('0') indicates that additional 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 Data.

```
C/C++:   unsigned char I2C_Read(short int LastByte, unsigned char *Data)
Delphi:  I2C_Read(LastByte: wordbool; var Data: byte): byte;
VB:     I2C_Read(ByVal LastByte As Boolean, ByRef Data As Byte) As Byte
```

I2C_ReadArray

The I2C_ReadArray 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. I2C_ReadArray 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 I2C_ReadArray(unsigned char address,
unsigned char subaddress, int nBytes, unsigned char *ReadData)
Delphi:  I2C_ReadArray(address, subaddress: byte; nBytes: integer;
var ReadData): byte;
VB:     I2C_ReadArray(ByVal address, ByVal subaddress As Byte, nBytes As
Long, ByRef ReadData As Byte) As Byte
```



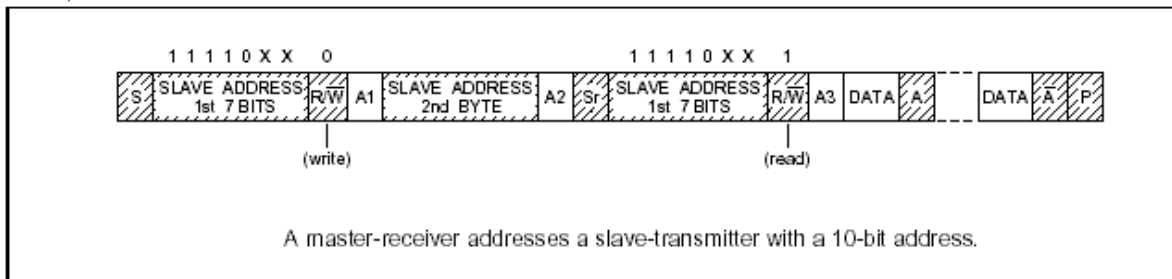
I2C10_ReadArray

The I2C10_ReadArray function (read an array with 10-bit device addressing) is similar to the I2C_ReadArray 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 I2C10_ReadArray(short int address, unsigned char subaddress, int nBytes, unsigned char *ReadData)`

Delphi: `I2C10_ReadArray(address: word, subaddress: byte; nBytes: integer; var ReadData): byte;`

VB: `I2C10_ReadArray(ByVal address As Integer, ByVal subaddress As Byte, nBytes As Long, ByRef ReadData As Byte) As Byte`

I2C_ReadArrayNS

The I2C_ReadArrayNS 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 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 I2C_ReadArrayNS(unsigned char address, int nBytes, unsigned char *ReadArray)`

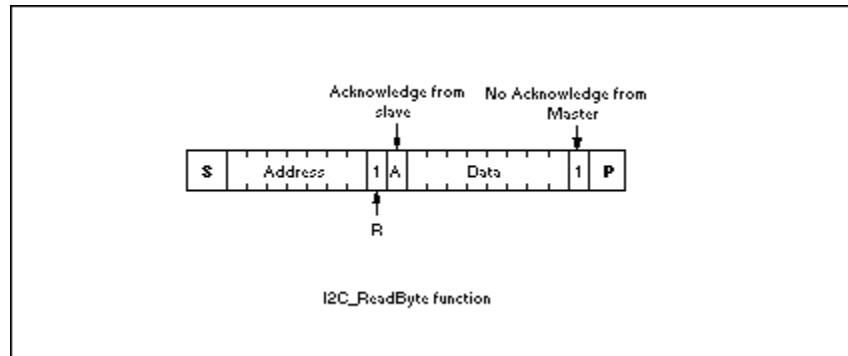
Delphi: `I2C_ReadArrayNS(address: byte; nBytes: integer; var ReadData: byte): byte;`

VB: `I2C_ReadArrayNS(ByVal address, nBytes As Long, ByRef ReadData As Byte) As Byte`

I2C_ReadByte

The I2C_ReadByte function reads one byte from an I2C bus/SMBus device. The function takes the device address and a pointer to a location in memory to store the data byte. I2C_ReadByte 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 ReadByte(unsigned char address, unsigned char
                                         *ReadData)
Delphi:  ReadByte(address: byte; var ReadData: byte): byte;
VB:      ReadByte(ByVal address As Byte, ByRef ReadData As Byte) As Byte
```



I2C_Start

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 I2C_Start(void)
Delphi:  I2C_Start: byte;
VB:      I2C_Start() As Byte
```

I2C_Stop

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 I2C_Stop(void)
Delphi:  I2C_Stop: byte;
VB:      I2C_Stop() As Byte
```

I2C_Write

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 I2C_Write(unsigned char DataByte)
Delphi:  I2C_Write(DataByte: byte): byte;
VB:      I2C_Write(ByVal DataByte As Byte) 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. It is up to the calling program to correctly define the array of

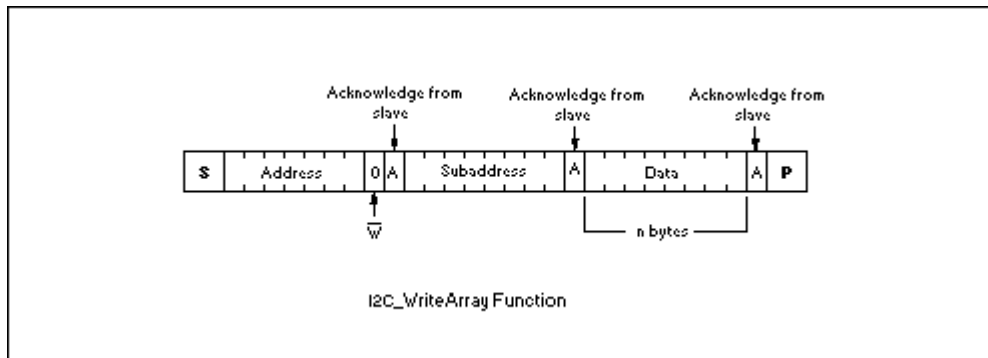
Win-I2CNTDLL

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 conditions it encounters. See Error Code section for return values.

C/C++: **unsigned char** I2C_WriteArray(**unsigned char** address, **unsigned char** subaddress; **int** nBytes; **unsigned char** *WriteData)

Delphi: I2C_WriteArray(address,subaddress: **byte**; nBytes: **integer**;
var WriteData: **byte**): **byte**;

VB: I2C_WriteArray (ByVal address As **Byte**, ByVal subaddress As **Byte**, ByVal nBytes As **Long**, ByRef WriteData As **Byte**) As **Byte**



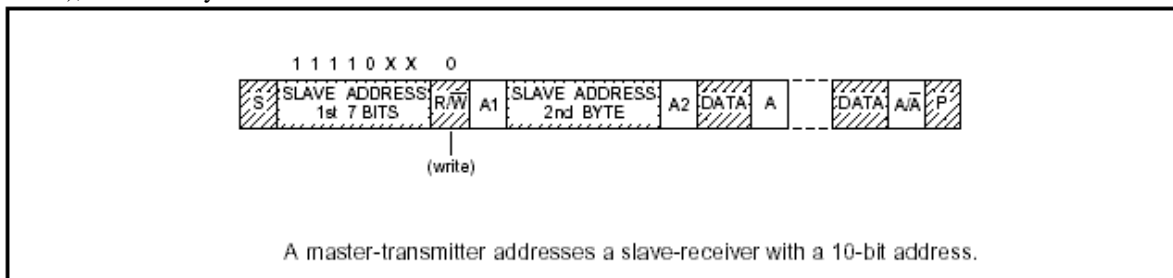
I2C10_WriteArray

The I2C10_WriteArray function (write an array with 10-bit device addressing) is similar to the I2C_WriteArray 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** I2C10_WriteArray(**short int** address, **unsigned char** subaddress; **int** nBytes; **unsigned char** *WriteData)

Delphi: I2C10_WriteArray(address,subaddress: **byte**; nBytes: **integer**;
var WriteData: **byte**): **byte**;

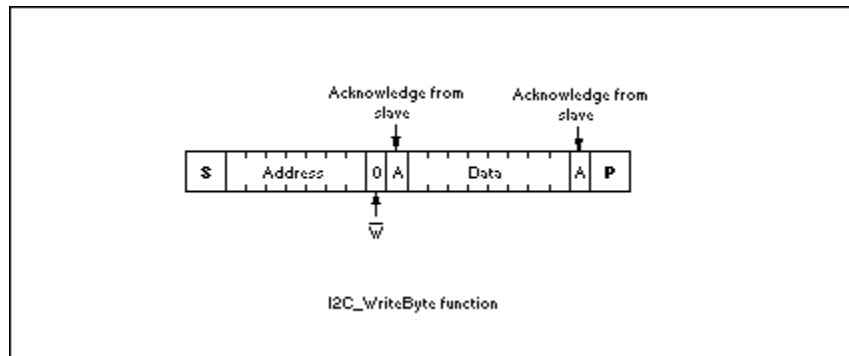
VB: I2C10_WriteArray (ByVal address As **Integer**, ByVal subaddress As **Byte**, ByVal nBytes As **Long**, ByRef WriteData As **Byte**) As **Byte**

I2C_WriteByte

The I2C_WriteByte 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.

Win-I2CNTDLL

```
C/C++: unsigned char I2C_WriteByte(unsigned char address, unsigned char Data)
Delphi: I2C_WriteByte(address, Data: byte): byte;
VB:     I2C_WriteByte(ByVal address As Byte, ByVal Data As Byte) As Byte
```



milliseconds

The milliseconds delay function allows the user to program a delay, measured in milliseconds, into their 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)
```

OpenI2CDriver

Loads the virtual device driver under Windows 95/98/ME or the kernel mode device driver under Windows NT/2000/XP, providing direct access to the LPT ports. If the driver was successfully opened, the function returns True; if the function fails it returns False.

The only valid device numbers that may be passed to OpenI2CDriver is either '0' or '1'. Only one instance of device '0' and one instance of device '1' may be started at one time, therefore, only two applications may access the kernel mode driver. Note that Win-I2CNT uses '1' so if you have Win-I2CNT running, you will be limited to using '0'.

```
C/C++: short int OpenI2CDriver(int device)
Delphi: OpenI2CDriver(device: integer): wordbool;
VB:     OpenI2CDriver(ByVal device As Long) As Boolean
```

SetNormalAccessMode

The SetNormalAccessMode function puts the kernel-mode driver in 'Normal' access mode to the I/O ports. 'Normal' access provides higher performance access to ports, but may fail if the port(s) addressed is already in use by another kernel-mode driver. While slower, 'Slow' access provides more reliable access to ports that have already been opened by another kernel-mode driver.

If a false is returned by the function, the kernel-mode driver failed and it will need to be reopened in Slow access mode.

```
C/C++: short int SetNormalAccessMode(void)
Delphi: SetNormalAccessMode: wordbool;
VB:     SetNormalAccessMode() 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 `GetMaxI2CFrequency` 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 Win-I2CNTDLL 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' ('1') 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
```

SetSlowAccessMode

The `SetSlowAccessMode` function puts the kernel-mode driver in 'Slow' access mode to the I/O ports. 'Slow' access provides lower performance access to ports, but is more reliable when another kernel-mode driver has already opened the port.

There may also be situations where you want to use 'Slow' access mode as it adds additional set-up and hold times to the I²C signal.

When the driver is initially started, the default state is 'Slow' access mode.

```
C/C++:  short int SetSlowAccessMode(void)
Delphi: SetSlowAccessMode: wordbool;
VB:     SetSlowAccessMode() 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, this 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
```

Exported Functions using a C type calling convention (cdecl)

For programming languages that require a C calling convention (cdecl) use the functions listed below. In the cdecl calling convention, parameters are passed on the stack from right to left, and it is up to the application using the dll to clean up the stack.

CheckDriver

This function takes no argument and returns the current state of the hardware driver. If the return value is '1' (true) when the driver is functioning and a '0' (false) is returned if the driver is not started. Use this function to ensure that the driver is active before attempting any I²C communications.

```
short int CheckDriver(void)
```

Get_AccessMode

The Get_AccessMode function determines whether the kernel-mode driver is using 'Normal' or 'Slow' access to the I/O ports. If true, 'Normal' access is in use while a false indicates that 'Slow' access is in use. 'Normal' access provides higher performance access to ports, but may fail if the port(s) addressed is already in use by another kernel-mode driver. While slower, 'Slow' access provides more reliable access to ports that have already been opened by another kernel-mode driver.

```
short int Get_AccessMode(void)
```

GetAddress

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.

```
short int GetAddress(void)
```

GetFrequency

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

```
int GetFrequency(void)
```

GetLPT

The GetLPT 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.

```
unsigned char GetLPT(void)
```

GetMaxFrequency

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

```
int GetMaxFrequency(void)
```

HWDetect

This function checks to see if the I2CPort2.0 hardware is attached to the currently selected parallel port. The two byte value returned by HWDetect contains true ('1') 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 SetLPT function, it is a good idea to call HWDetect to see if the I2CPort2.0 hardware was detected. The dll initialization routine attempts to detect the hardware at LPT1 if the driver was successfully started.

```
short int HWDetect(void)
```

I2CRead

This function passes a two byte boolean value and a pointer to data byte and returns any error condition it encounters during the I²C bus transmission. See Error Code section for return values. The boolean value indicates whether the byte will be the last byte read from the I²C bus. A true ('1') indicates that it is the last byte to read while a false ('0') indicates that additional bytes will be read. The data byte, which is read from the I²C bus, is stored in the memory location specified by ReadData.

```
unsigned char I2CRead(short int LastByte, unsigned char *ReadData)
```

I2CStart

This function takes no arguments and generates an I²C Start Condition via the parallel port. The function returns a byte containing the error condition it encounters. See Error Code section for return values.

```
unsigned char I2CStart(void)
```

I2CStop

This function generates an I²C Stop Condition via the parallel port. The function returns a byte containing any error condition it finds. See Error Code section for return values.

```
unsigned char I2CStop(void)
```

I2CWrite

This function writes one byte, passed by the calling program, to the I²C bus via the parallel port. The function returns one byte containing error information. See Error Code section for return values.

```
unsigned char I2CWrite(unsigned char DataByte)
```

msDelay

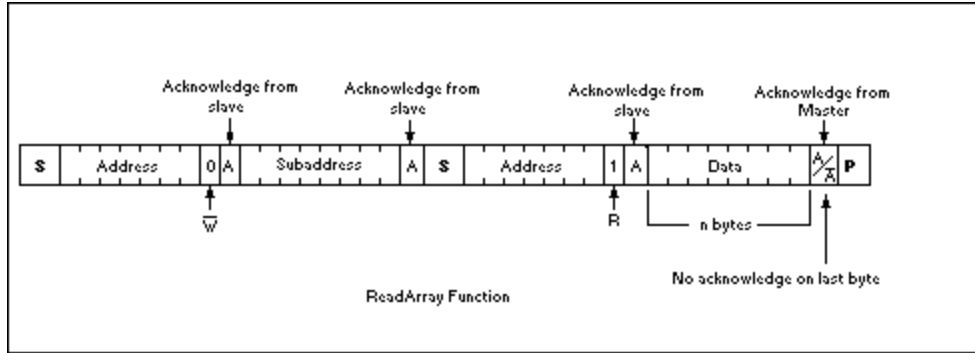
The msDelay delay function allows the user to program a delay, measured in milliseconds, into their I²C messages. This is particularly useful when programming EEPROM devices that require a minimum erase/write time between write transactions.

```
void msDelay(int milliseconds)
```

ReadArray

The ReadArray 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. ReadArray 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.

```
unsigned char ReadArray(unsigned char address, unsigned char subaddress,  
                        int nBytes, unsigned char *ReadData)
```



ReadArrayNS

The ReadArrayNS (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.

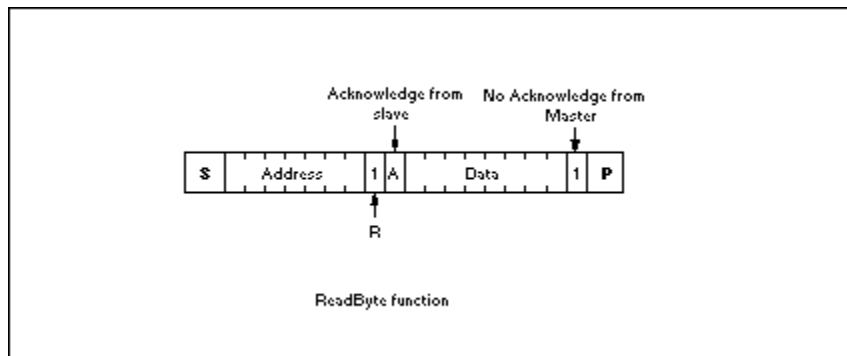
This function takes three arguments: the device address, the number of bytes to read, and a pointer to an element within an array of bytes. I2C_ReadArray sends the I2C message shown below and returns any error condition it encounters. It is the calling program's responsibility to allocate the correct memory space for the array. The function ensures that the lsb of the address is set to a '1' before it is sent to the target device.

```
unsigned char ReadArrayNS(unsigned char address, int nBytes,
                          unsigned char *ReadArray)
```

ReadByte

The ReadByte function reads one data byte from an I2C bus device. The function takes the device address and a pointer to the memory location to store the data byte. ReadByte returns any error condition it encounters. See the Error Codes section for the list of error conditions. The function ensures that the lsb of the address is a '1' before it is sent to the target device.

```
unsigned char ReadByte(unsigned char: address, unsigned char *ReadData)
```



SetFrequency

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 Win-I2CNTDLL cannot be guaranteed. If an accurate frequency is required, it is recommended that the frequency be verified using appropriate test equipment.

```
int SetFrequency(int frequency)
```

SetLPT

If you have more than one parallel port in your computer, you can choose which parallel port to communicate with, by using the SetLPT 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' ('1') if the parallel port was detected, while it returns 'false' ('0') if the chosen parallel port was not available. LPT is set back to LPT1 if the requested LPT was not available.

```
short int SetLPT(unsigned char LPT)
```

SetNormalMode

The SetNormalMode function puts the kernel-mode driver in 'Normal' access mode to the I/O ports. 'Normal' access provides higher performance access to ports, but may fail if the port(s) addressed are already in use by another kernel-mode driver. While slower, 'Slow' access provides more reliable access to ports which have already been opened by another kernel-mode driver. If a false is returned by the function, the kernel-mode driver failed and it will need to be reopened in Slow access mode.

```
short int SetNormalMode(void)
```

SetSlowMode

The SetSlowMode function puts the kernel-mode driver in 'Slow' access mode to the I/O ports. 'Slow' access provides lower performance access to ports, but is more reliable when another kernel-mode driver has already opened the port. There may also be situations where you want to use 'Slow' access mode as it adds additional set-up and hold times to the I²C signal. When the driver is initially started, the default state is 'Slow' access mode.

```
short int SetSlowMode(void)
```

SetWait

The I²C bus specification does not specify the amount of time that a device can hold the clock line low to inject a wait state in a transmission. In order to ensure that the software responds in a predictable manner, this 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 to 50ms. If this time is exceeded, the dll will exit the function. The SetWait 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 SetWait(int NewWaitTime)
Delphi:   SetWait(NewWaitTime: integer): integer;
VB:      SetWait(ByVal NewWaitTime As Long) As Long
```

StartI2CDriver

Loads the virtual device driver under Windows 95/98/ME or the kernel-mode device driver under Windows NT/2000/XP, providing direct access to the LPT ports. If the driver was successfully opened, the function returns True; if the function fails it returns False.

The only valid device numbers that may be passed to OpenI2CDriver is either '0' or '1' (Other values have no effect). Only one instance of device '0' and one instance of device '1' may be started at one time, therefore, only two applications may access the kernel-mode driver. Note that Win-I2CNT uses '1' so if you have Win-I2CNT running, you will be limited to using device '0'.

```
short int StartI2CDriver(device: integer)
```

StopI2CDriver

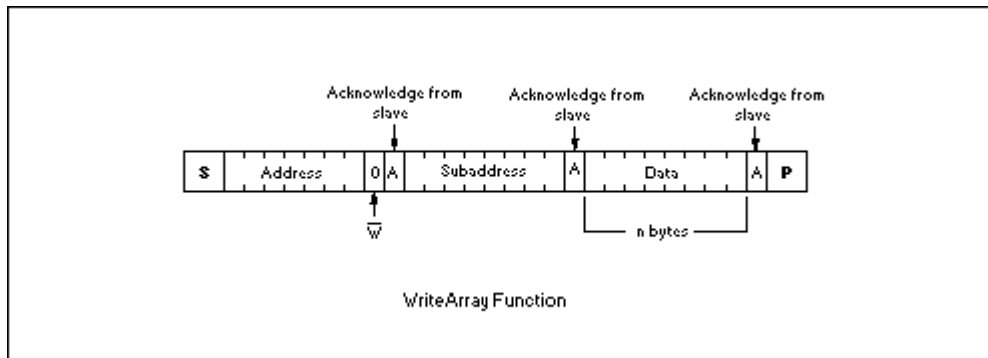
Closes the kernel-mode driver and releases memory allocated to it. It is highly recommended that this function be called before an application is terminated.

```
void StopI2CDriver(void)
```

WriteArray

The 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. It is up to the calling program to correctly define the array of data. The function ensures that the lsb of the address is a '0' before it is sent to the target device.

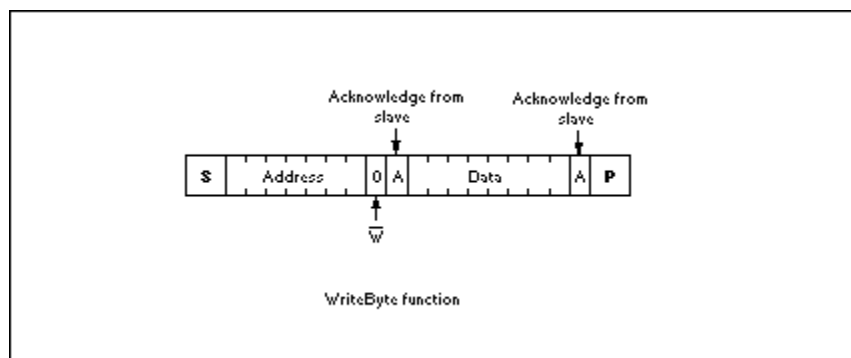
```
unsigned char WriteArray(unsigned char address, unsigned char subaddress,  
int nBytes, unsigned char *WriteArray)
```



WriteByte

The WriteByte 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.

```
unsigned char WriteByte(unsigned char address, unsigned char Data)
```



Error Codes

The following error codes are returned by the various functions in sbsi2c4.dll:

0x00: No error
0x01: Address not acknowledged (only valid for a WriteByte, ReadByte, WriteArray, ReadArray function)
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 (I2CPort2.0 not detected)

Examples

Visual Basic Example

This example writes two bytes to the EEPROM, located on the I2CPort2.0 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 Win-I2CNTDLL) into your project. See the Visual Basic example included with Win-I2CNTDLL.

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=I2C_Start        'Generate I2C Start Condition.  
ErrorCode=I2C_Write(174)   'Send the EEPROM address...in this case 0xAE.  
ErrorCode=I2C_Write(0)     'Send the subaddress.  
ErrorCode=I2C_Write(0)     'Send first data byte.  
ErrorCode=I2C_Write(1)     'Send second data byte.  
ErrorCode=I2C_Stop        'Generate I2C Stop Condition.  
  
milliseconds(10)          'wait 40ms for the erase/write cycle to complete.  
  
ErrorCode=I2C_Start        'Generate I2C Start Condition.  
ErrorCode=I2C_Write(174)   'Send the EEPROM write address...in this case 0xAE.  
ErrorCode=I2C_Write(0)     'Send the subaddress.  
ErrorCode=I2C_Start        'Generate I2C Start Condition.  
ErrorCode=I2C_Write(175)   'Send the EEPROM read address... in this case 0xAF.  
ErrorCode=I2C_Read(false,ReadData) 'Read one byte (not last byte).  
Label1.Caption=Int(ReadData)  
ErrorCode=I2C_Read(true,ReadData) 'Read one byte (last byte).  
Label2.Caption=Int(ReadData)  
ErrorCode=I2C_Stop        'Generate I2C Stop Condition.  
End Sub
```

Delphi Example

This example writes two bytes to an EEPROM, located on the I2CPort2.0 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 Win-I2CNTDLL) 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 Win-I2CNTDLL).

```
procedure TDLLForm.btnWriteandRead(Sender: TObject);
var
  ErrorCode, ReadData: byte;
  InitOK: WordBool;
  Freq: integer;
begin
  if CheckDriver 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:=I2C_Start; //Generate I2C Start Condition.
    ErrorCode:=I2C_Write(174); //Send the EEPROM address...in this case 0xAE.
    ErrorCode:=I2C_Write(0); //Send the subaddress.
    ErrorCode:=I2C_Write(0); //Send first data byte.
    ErrorCode:=I2C_Write(1); //Send second data byte.
    ErrorCode:=I2C_Stop; //Generate I2C Stop Condition.
    milliseconds(10); //wait 10ms for the erase/write cycle to complete.
    ErrorCode:=I2C_Start; //Generate I2C Start Condition.
    ErrorCode:=I2C_Write(174); //Send the EEPROM write address...in this case 0xAE.
    ErrorCode:=I2C_Write(0); //Send the subaddress.
    ErrorCode:=I2C_Start; //Generate I2C Start Condition.
    ErrorCode:=I2C_Write(175); //Send the EEPROM read address... in this case 0xAF.
    ErrorCode:=I2C_Read(false,ReadData); //Read one byte from the EEPROM, not last byte.
    Label1.Caption := IntToStr(ReadData);
    ErrorCode:=I2C_Read(true); //Read one byte from the EEPROM, last byte.
    Label2.Caption := IntToStr(ReadData);
    ErrorCode:=I2C_Stop; //Generate I2C Stop Condition.
  end
  else ShowMessage('Driver not started');
end;
```