Application Note 3315
Writing Parallel Port 2-Wire Software in C

www.maxim-ic.com

INTRODUCTION

This application note discusses the C source code that is being provided for customers that are interested in developing 2-Wire software for the parallel port hardware described in application note AN3230. The source code is available for free on Dallas Semiconductor's FTP site, and can run on any PC using the Windows 95 or Windows 98 operating systems. Additionally, a simple Windows program is available on the FTP site that provides basic 2-Wire communication software that can be used for simple evaluation and for debugging the parallel port hardware. For more information about the hardware refer to AN3230.

The software presented in this application note is free and available "as is" for use by our customers. Dallas Semiconductor accepts no liability for any damages the software may cause. Use the software at your own risk.

SOFTWARE REQUIREMENTS

As mentioned in the introduction, this program must be run on a PC with either the Windows 95 or Windows 98 operating system. This software directly accesses the parallel port, and Windows NT based operating systems require a driver to accomplish this task. Look for application notes in the near future on using drivers for accessing the parallel port for Windows NT/2000/XP.

Additionally, there are multiple parallel port modes of operation, some of which are not compatible with the software. Two modes that have been successfully used are EPP and ECP. Most PC's parallel port mode can be changed in the BIOS settings.

SOURCE CODE DESCRIPTION / USE

The source code (see Appendix A) is written in ANSI C, so it should be compatible with any C compiler. To make its use as simple as possible all the code and declarations are in a single file (2wire.c), so there is not a header file that must also be included into the project to use the resources.

To use the 2-Wire source code:

1. Place the "2wire.c" file into the project directory.
2. Add a #include "2wire.c" declaration at the top of the program that will be accessing the parallel port.
3. Call ParPortSelect(1) to select a parallel port. The argument determines the port number (e.g. LPT1 as shown) that will be utilized. Valid port numbers are 1, 2, and 3. Most PCs will utilize LPT1.
4. Calibrate the 2-Wire interface timing using the ChangeDelayCount(int i) command shown in Table 3.
5. Call either the basic 2-Wire functions or the multiple-byte 2-Wire functions, described in Tables 2 and 3.

Parallel port timing is heavily dependent on the speed of the PC that is executing the software, so the variation between computers can make it difficult to establish reliable communication. To resolve this issue, a variable length delay is inserted between the SDA and SCL signals to ensure that the timing on faster computers does not exceed the maximum rated interface speed (400kbits/second for fast mode devices). The delay time is controlled by calling the ChangeDelayCount(int i) function. This function changes the number of times the PC will execute a short delay (10 NOPs + a *for(; ;)* loop's execution time). The default value for i is 1000, which provides moderate to slow communication on a P3 600MHz machine. This will result in moderate performance on most PCs, but it should work reliably. Lower values of i will speed up the interface, but the programmer must ensure that the speed of communication is within the 2-Wire device's specification. On faster machines a larger value of i may be required to establish communication, so this is something to examine if debugging is required.

The basic 2-Wire functions can be used for most applications to access a 2-Wire device. The mechanics of sending start conditions, writing and reading bytes, and sending stop conditions have been handled in these routines, leaving only the timing and the order the routines are called as the last remaining obstacles of communicating with a device. To use these routines, adjust the timing as explained above and read the device's datasheet to determine the order the calls should be made to access registers.

The multiple-byte 2-Wire functions can be used to read and write up to 256 bytes of information from a device with one command, but not all devices utilize the same data sequences during communication. Please check the provided source code if considering using these functions to make sure they are compatible with the device being accessed. The primary advantage with the multiple-byte writes/reads is that it limits the number of calls that are required for the application because it transmits multiple bytes with a single command versus sending multiple commands to write/read a single byte. The multiple-byte write and read routines use the device address set by the SetSlaveAddress() command, so SetSlaveAddress() must be called before using multiple-byte write and read commands.

The LED enable and disable functions allow the LED shown in AN3230 to be used as a status indicator. The strobe set and clear functions allow the LED pin to be used to trigger an oscilloscope. These functions can be extremely useful for debugging hardware and software problems.

Table 1. Basic 2-Wire Functions

FUNCTION PROTOTYPE	FUNCTION DESCRIPTION	RETURN VALUE
int Start()	Generates 2-Wire start condition. Can also be called to generate a re-start condition.	1
int Stop()	Generates 2-Wire stop condition	1
int WriteData(unsigned char ucData);	Writes the argument to the slave	1 if slave acknowledges, 0 if slave does not acknowledges
int ReadDataAck(unsigned char *ucData);	Reads the data byte from slave to ucData and acknowledges	1
int ReadDataNack(unsigned char *ucData);	Reads the data byte from slave to ucData and does not acknowledge	1
int ResetBus()	Clocks SCL 9-times then generates a stop condition	1

Table 2. Multiple-Byte 2-Wire Functions

FUNCTION PROTOTYPE	FUNCTION DESCRIPTION	RETURN VALUE
int SetSlaveAddress(unsigned char ucADDR)	Sets the slave address for multiple-byte read and write accesses	1
int WriteBytes(int iCount, unsigned char ucMemAddr, unsigned char ucData[256])	Writes iCount bytes to the slave at the device address set by SetSlaveAddress(), beginning at memory address set by ucMemAddr.	1 if slave acknowledges, 0 if slave does not acknowledge any byte.
Int ReadBytes(int iCount, unsigned char ucMemAddr, unsigned char ucData[256])	Reads iCount bytes to the slave at the device address set by SetSlaveAddress(), beginning at memory address set by ucMemAddr.	1 if slave acknowledges during command writes, 0 if slave does not acknowledge during command writes.

Table 3. Additional Port Setup and Debugging Functions

FUNCTION PROTOTYPE	FUNCTION DESCRIPTION	RETURN VALUE
int ParPortSelect(int iLPT)	Sets parallel port access variables to specified port number. iLPT = 1 for LPT1.	1 for successful change 0 for failure
int ChangeDelayCount(int iCount)	This determines the "i" value used with the DelayASMx10() command during SDA and SCL communications. Call this function with higher i values to make communication slower. The default value is 1000, which should provide moderate to slow communication speed. This is a safe value for i. Most PCs will be able to use a lower value of i to speed up communications.	1
void DelayASMx10(int i)	Delays 10 clock cycles per i. This is called as the delay that determines SDA and SCL timing. It does not need to be called by the software developer, it is already embedded into the start/stop/read/write commands.	NULL
int EnableLED()	enables LED in AN3230 circuit	1
int DisableLED()	disables LED in AN3230 circuit	1
int SetStrobe()	Sets the LED pin in the AN3230 circuit high for oscilloscope triggering	1
int ClearStrobe()	Sets the LED pin in the AN3230 circuit low for oscilloscope triggering	1

BASIC 2-WIRE FUNCTION EXAMPLES

This section shows how to write two bytes to the DS1086 DAC register and then read them back using the basic 2-Wire functions. The DS1086's slave address is B0h, and the DAC register is 2-bytes starting at memory address 08h.

To write 0180h to addresses 08h and 09h the following procedure can be utilized.

```
unsigned char fail = 0;

Start();                    // Generates Start Condition
fail |= !WriteData(0xB0); // Writes the slave address
fail |= !WriteData(0x08); // Writes the memory address of the DAC register
fail |= !WriteData(0x01); // Writes the MSB of the DAC register
fail |= !WriteData(0x80); // Writes the LSB of the DAC register
Stop();                     // Generates Stop Condition
if(fail == 1)
        Error("Device failed to acknowledge during write attempt");
```

To read the bytes just written to the DAC register, the following code can be used.

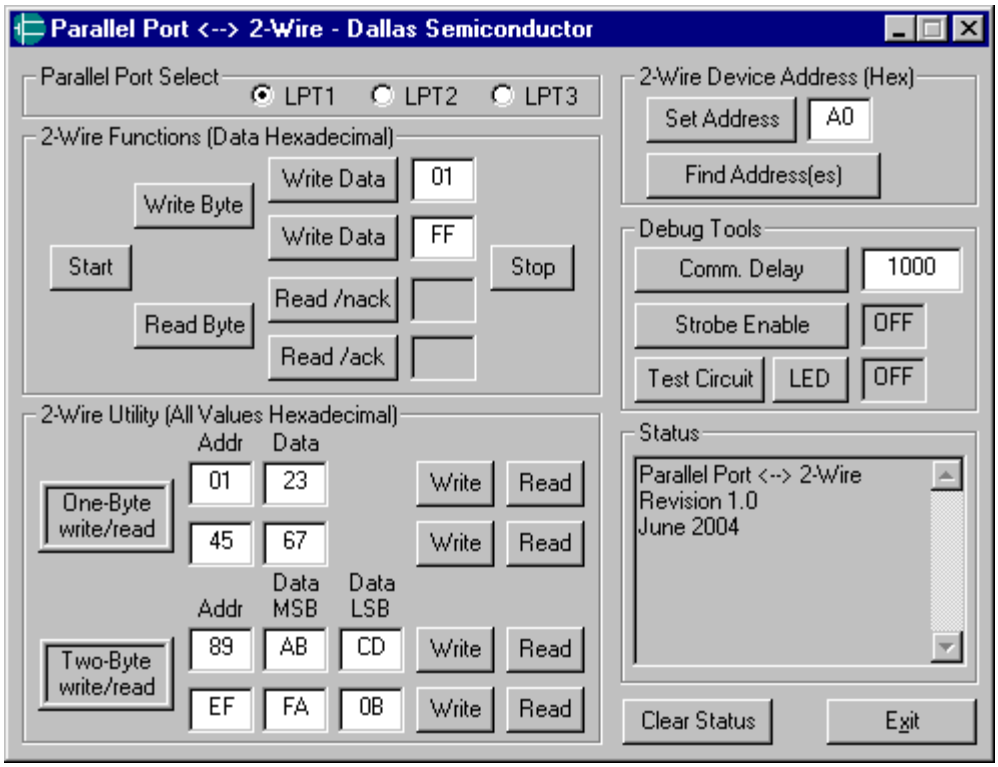
```
unsigned char ucDataMSB=0;          // define variable for MSB data to be stored after the read
unsigned char ucDataLSB=0;          // define variable for LSB data to be stored after the read
unsigned char fail = 0;

Start();                            // Generate Start Condition
fail |= !WriteData(0xB0);           // Write the slave address, LSbit=0 to signify write byte
fail |= !WriteData(0x08);           // Write the memory address of the DAC register
Start();                            // Generates a re-start condition
WriteData(0xB1);                    // Writes the slave address, LSbit=1 to signify read byte
ReadDataAck(&ucDataMSB);            // Reads the MSB of DAC and sends acknowledgement to the slave
ReadDataNack(&ucDataLSB);           // Reads the LSB of DAC and does not acknowledge the slave
Stop();                             // Generates Stop Condition
if(fail ==1)
        Error("Device failed to acknowledge during read attempt");
```

WINDOWS PARALLEL PORT 2-WIRE SOFTWARE

The software pictured in Figure 1 was written to demonstrate the basic functionality of the “2wire.c” software, and it can also be used to debug the AN3230 hardware as well.

Figure 1. Example Windows Parallel Port to 2-Wire Software



The software can communicate with any of the 3-parallel ports listed in the Parallel Port Select section.

The buttons in the 2-Wire Functions section simply accept the parameters in the dialog box and call the corresponding “2wire.c” functions. Start generates a start condition, and Write Data takes the parameter in the box to the right of the button and writes it to the slave. The two read buttons both read a byte from the slave, but one acknowledges the data transfer and the other does not acknowledge the data transfer. The stop button generates a stop condition. The Write Byte button writes the value in the 2-Wire Device address box with the LSbit set to zero, and the read button writes the same value with the LSbit set to one. These buttons allow the write data buttons to be used for data and memory addresses without having to frequently change one of the values to the slave address every time a part is accessed.

One-Byte writes transmit a single data byte (Data) to the slave address listed in the 2-Wire Device Address box at the memory address listed in the Addr box.

One-Byte reads read one byte from the slave at the slave address listed in the 2-Wire Device Address Box from the memory address listed in the Addr box. The dialog will read without acknowledgement to indicate it is only reading one byte.

Two-Byte writes transmit two data bytes (Data MSB and Data LSB) to the slave address listed in the 2-Wire Device Address box at the memory address listed in the Addr box.

Two-Byte reads read two data bytes from the slave at the slave address listed in the 2-Wire Device Address box beginning at the memory address listed in the Addr box. The dialog will acknowledge the first data byte read and will not acknowledge the second data byte read.

The Find Address(es) button writes to every slave address on the 2-Wire bus (00h-FEh) checking for addresses that acknowledge to determine which addresses have slaves. The addresses that respond with acknowledgement will be listed in the status box. This button can be used to determine if the AN3230 hardware is set up correctly and a slave capable of receiving data is connected to the 2-Wire bus.

The Comm. Delay button calls the ChangeDelayCount() function with the integer to the right of the button as the argument. This can be used to tune the timing of the 2-Wire interface.

The LED button enables and disables the indicating LED shown in the AN3230 circuit. The strobe enable causes the software to set the LED pin high before the one and two byte write/read functions, and sets the LED pin low after the end of the communication.

The Test button performs a loop back test, ensuring when the SDA output is set low that the SDA input reads low, and likewise for the high condition. It also tests that the LED pin can be set low and high, pausing long enough that the user should see the LED blink on then back off.

All commands listed above provide user feedback in the Status window.

CONCLUSION

This application note demonstrates the ease of programming 2-Wire devices using the parallel port 2-Wire circuit shown in AN3230 with the source code provided in the "2wire.c" file. The routines in "2wire.c" perform the signaling functions, which allows the programmer to concentrate on tuning the interface timing and the calling the 2-Wire routines in the proper order to communicate as outlined in the 2-Wire device's datasheet.

The source code for "2wire.c" is are the Windows program are available for download from Dallas Semiconductor's FTP site.

ftp://ftp.dalsemi.com/pub/system_extension/AppNotes/

Questions related to this application note can be directed to the Dallas Semiconductor Mixed Signal Applications group at MixedSignal.Apps@dalsemi.com.

DALLAS SEMICONDUCTOR/MAXIM CONTACT INFORMATION

Company Addresses:
Maxim Integrated Products, Inc
120 San Gabriel Drive
Sunnyvale, CA 94086
Tel: 408-737-7600
Fax: 408-737-7194

Dallas Semiconductor
4401 S. Beltwood Parkway
Dallas, TX 75244
Tel: 972-371-4448
Fax: 972-371-4799

Product Literature / Samples Requests:
(800) 998 – 8800

Sales and Customer Service:
(408) 737 - 7600

World Wide Website:
http://www.maxim-ic.com

Product Information:
http://www.maxim-ic.com/MaximProducts/products.htm

Ordering Information:
http://www.maxim-ic.com/BuyMaxim/Sales.htm

FTP Site:
ftp://ftp.dalsemi.com

Email Support:
MixedSignal.Apps@dalsemi.com

APPENDIX A – SOURCE CODE

```
////////////////////////////////////////////////////////////////////////////
// 2-Wire Parallel Port Functions for Win95/Win98
//
// Dallas Semiconductor (C) 2004
//
// This software is free and available “as is” for use by Dallas
// Semiconductor’s customers. Dallas Semiconductor accepts no liability for any
// damages the software may cause. Use at your own risk.
//
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// Include Files

#include <conio.h>


////////////////////////////////////////////////////////////////////////////
// Definitions

#define LPT1 0x378
#define LPT2 0x3BC
#define LPT3 0x278
#define LPT4 0x378    // not implemented - set to LPT1 for now,
                                  //   returns error code if assigned


////////////////////////////////////////////////////////////////////////////
// Global Variables
unsigned short LPTData;              // parallel port base address
unsigned short LPTStatus;            // LPT Base Address + 1
unsigned short LPTControl;           // LPT Base Address + 2

unsigned char ucDeviceAddress;       //  2-Wire device address for
                                     //   multibyte communication


int     DCNT = 1000;   // Delay Count...Sets the delay time used for SDA/SCL
        //   during 2-Wire communication. DCNT=25 works well for MY P3 600MHz
        //   Laptop. 1000 is the default value providing some guardband for
        //   faster machines so they do not attempt to talk faster than the
        //   400kHz rating of the 2-Wire interface. This value can be changed
        //   high or lower to find the best value using ChangeDelayCount();


////////////////////////////////////////////////////////////////////////////
// Function Prototypes

//    Parallel Port Utility Functions

void DelayASMx10(int i);             // delays 10 clock cycles per i
int ChangeDelayCount(int iCount);    // changes i for DelayASMx10
int ParPortSelect(int iLPT);         // sets access variables to specified
        //  port. Must be called before any other parallel port functions

//    Basic 2-Wire Functions

int Start();                                  // 2-Wire Start Command
int Stop();                                   // 2-Wire Stop Command
int WriteData(unsigned char ucDATA);          // 2-Wire Write Byte
int ReadDataNack(unsigned char *ucDATA);      // 2-Wire Read Byte with NACK
int ReadDataAck(unsigned char *ucDATA);       // 2-Wire Read Byte with ACK

//    MultiByte Write/Read 2-Wire Functions

int SetSlaveAddress(unsigned char ucADDR);   // sets slave address for
        // WriteBytes and ReadBytes Commands
int WriteBytes(int iCount, unsigned char ucMemAddr, unsigned char ucData[256]);
        // Write upto 256 bytes to device address set by SetSlaveAddress.
int ReadBytes(int iCount, unsigned char ucMemAddr, unsigned char ucData[256]);
        // Reads upto 256 bytes from device address set by SetSlaveAddress.

//    Utility 2-Wire Functions

int ResetBus();              // clocks SCL 9 times and performs stop to "free"
                             //   SDA and SCL for the software master.
int     SetStrobe();         // sets strobe pin for debugging (AN3230)
int ClearStrobe();           // clears strobe pin for debugging (AN3230)

//    Enable / Disable LED on AN3230 circuit

int EnableLED();             // enables LED in AN3230 circuit
int DisableLED();            // disables LED in AN3230 circuit
```

```
//////////////////////////////////////////////////////////////////////////
// Function Definitions

void DelayASMx10(int iDelayCount)  // delays 10 NOPs iDelayCount times.
{ // See DCNT description in variable declarations for more information
        int iLoopCount = 1;
        for (; iLoopCount<=iDelayCount; iLoopCount++)
        {
                __asm
                {
                        NOP
                        NOP
                        NOP
                        NOP
                        NOP
                        NOP
                        NOP
                        NOP
                        NOP
                        NOP
                }
        }
}
int ParPortSelect(int iLPT)
{       // Selects the parallel port used for operation.
        if(iLPT == 1)
        {
                LPTData = LPT1;
                LPTStatus  = LPTData + 1;     // LPT Data + 1
                LPTControl = LPTData + 2;     // LPT Data + 2
                return 1;                                       // legal port value
        }
        else if(iLPT == 2)
        {
                LPTData = LPT2;
                LPTStatus  = LPTData + 1;     // LPT Data + 1
                LPTControl = LPTData + 2;     // LPT Data + 2
                return 1;                                       // legal port value
        }
        else if(iLPT == 3)
        {
                LPTData = LPT3;               // legal port value
                LPTStatus  = LPTData + 1;     // LPT Data + 1
                LPTControl = LPTData + 2;     // LPT Data + 2
                return 1;                                       // legal port value
        }
        else if(iLPT == 4)
        {
                LPTData = LPT4;                                 // not implemented,but defined as LPT1
                LPTStatus  = LPTData + 1;     // LPT Data + 1
                LPTControl = LPTData + 2;     // LPT Data + 2
                return 0;                                       // return 0 for unimplemented value
        }
        else
                return 0;                                       // illegal value, return 0 for error
}

int ChangeDelayCount(int iCount)      // changes i for DelayASMx10
{       // Changes the DCNT to increase/decreas delays between 2-Wire edges
        DCNT = iCount; // See DCNT declaration in variable declarations.
        return 1;
}
int EnableLED()
{       // Enables the LED shown on the AN3230 circuit
        unsigned char Data;

        // Read original value of control byte
        Data = _inp(LPTControl);

        // Turn LED on
        _outp(LPTControl, (Data & 0xF7));     // Clear P17 to turn LED on
        return 1;
}

int DisableLED()
{       // Disables the LED shown on the AN3230 circuit
        unsigned char Data;

        // Read original value of control byte
        Data = _inp(LPTControl);

        // Turn LED on
        _outp(LPTControl, (Data | 0x08));     // Clear P17 to turn LED on
```

```
        return 1;
}

int Start()
{       // Performs 2-Wire start condition
        unsigned char Data;

        // Read original value of data byte
        Data = _inp(LPTData);

        // Ensure SDA and SCL are high

        // Set SDA high (write D1 to a 0)
        Data = Data & 0xFD;
        _outp(LPTData, Data);
        DelayASMx10(DCNT);

        // Set SCL high (write D0 to a 0)
        Data = Data & 0xFE;
        _outp(LPTData, Data);
        DelayASMx10(DCNT);

        // Bring SDA low, then bring SCL low

        // Bring SDA low (write D1 to a 1)
        Data = Data | 0x02;
        _outp(LPTData, Data);
        DelayASMx10(DCNT);

        // Bring SCL low (write D0 to a 1)
        Data = Data | 0x01;
        _outp(LPTData, Data);
        DelayASMx10(DCNT);

        return 1;
}

int Stop()
{       // Performs 2-Wire stop condition
        unsigned char Data;

        // Read original value of data byte
        Data = _inp(LPTData);
        // Ensure SDA and SCL are low

        // Make SCL low (write D0 to a 1)
        Data = Data | 0x01;
        _outp(LPTData, Data);
        DelayASMx10(DCNT);

        // Make SDA low (write D1 to a 1)
        Data = Data | 0x02;
        _outp(LPTData, Data);
        DelayASMx10(DCNT);

        // Bring SCL high, then bring SDA high

        // Bring SCL high (write D0 to a 0)
        Data = Data & 0xFE;
        _outp(LPTData, Data);
        DelayASMx10(DCNT);

        // Bring SDA high (write D1 to a 0)
        Data = Data & 0xFD;
        _outp(LPTData, Data);
        DelayASMx10(DCNT);

        return 1;
}

int WriteData(unsigned char ucDATA)
{
        // This routine writes 1 data byte and checks for slave acknoledgement
        // Assumes Start already issued and SDA and SCL are both low

        unsigned char shiftbyte, statusbyte;
        int i;
        unsigned char ucTempData;

        shiftbyte = ucDATA;
        ucTempData = _inp(LPTData);

        // loop for 8 bits
        for (i=0; i<8; i++)
        {
```

```
                if (shiftbyte & 0x80)
                {
                        // Set SDA high (D1=0)
                        ucTempData = ucTempData & 0xFD;
                        _outp(LPTData, ucTempData);
                        DelayASMx10(DCNT/2);

                        // Clock SCL (D0=0, D0=1)
                        ucTempData = ucTempData & 0xFE;
                        _outp(LPTData, ucTempData);
                        DelayASMx10(DCNT/2);
                        ucTempData = ucTempData | 0x01;
                        _outp(LPTData, ucTempData);
                        DelayASMx10(DCNT/2);

                        // Bring SDA low (D1=1)
                        ucTempData = ucTempData | 0x02;
                        _outp(LPTData, ucTempData);
                        DelayASMx10(DCNT/2);

                }
                else
                {
                        // Bring SDA low (D1=1)
                        ucTempData = ucTempData | 0x02;
                        _outp(LPTData, ucTempData);
                        DelayASMx10(DCNT/2);

                        // Clock SCL (D0=0, D0=1)
                        ucTempData = ucTempData & 0xFE;
                        _outp(LPTData, ucTempData);
                        DelayASMx10(DCNT/2);
                        ucTempData = ucTempData | 0x01;
                        _outp(LPTData, ucTempData);
                        DelayASMx10(DCNT/2);
                }
                shiftbyte = shiftbyte << 1;
        }

        // Release SDA (D1=0)
        ucTempData = ucTempData & 0xFD;
        _outp(LPTData, ucTempData);
        DelayASMx10(DCNT/2);

        // Check if slave ACKs

        // Bring SCL high (D0=0)
        ucTempData = ucTempData & 0xFE;
        _outp(LPTData, ucTempData);
        DelayASMx10(DCNT/2);

        // Read SDA
        statusbyte = _inp(LPTStatus);

        // Bring SCL low (D0=1)
        ucTempData = ucTempData | 0x01;
        _outp(LPTData, ucTempData);
        DelayASMx10(DCNT/2);

        // Display ACK
        if (statusbyte & 0x20)
        {  // Slave ACK'd
                return 1;
        }
        else
        {  // Slave did not ACK
                return 0;
        }
}

int ReadDataNack(unsigned char *ucDATA)
{       // This routine reads one byte and NACKs. It assumes SDA and SCL
        // are low becuase a start condition/communication has already occured
        unsigned char shiftbyte;
        int i;
        int ucTempData, Status;

        shiftbyte = 0x00;

        // Ensure SDA is released (D1=0) and SCL is low (D0=1)
        ucTempData = _inp(LPTData);
        ucTempData = ((ucTempData & 0xFD) | 0x01);
        _outp(LPTData, ucTempData);
        DelayASMx10(DCNT/2);
```

```
	// loop for 8 bits
	for (i=0; i<8; i++)
	{	// Clock in one bit
		// Bring SCL high (D0=0)
		ucTempData = ucTempData & 0xFE;
		_outp(LPTData, ucTempData);
		DelayASMx10(DCNT/2);

		// Read in SDA
		Status = _inp(LPTStatus);

		// Bring SCL low (D0=1)
		ucTempData = ucTempData | 0x01;
		_outp(LPTData, ucTempData);
		DelayASMx10(DCNT/2);

		if (Status & 0x20)
		{	// Bit is high (although inverted through the 7405)
			shiftbyte = shiftbyte << 1;
			shiftbyte = shiftbyte & 0xFE; // Just in case compiler does not shift in 0
		}
		else
		{  // Bit is low (although inverted through the 7405)
			shiftbyte = shiftbyte << 1;
			shiftbyte = shiftbyte | 0x01;
		}
	}

	// NACK - Release SDA and clock SCL
	// Release SDA high (D1=0)
	ucTempData = ucTempData & 0xFD;
	_outp(LPTData, ucTempData);
	DelayASMx10(DCNT/2);

	// Bring SCL high (D0=0)
	ucTempData = ucTempData & 0xFE;
	_outp(LPTData, ucTempData);
	DelayASMx10(DCNT/2);

	// Bring SCL low (D0=1)
	ucTempData = ucTempData | 0x01;
	_outp(LPTData, ucTempData);
	DelayASMx10(DCNT/2);

	*ucDATA = shiftbyte;

	return 1;
}

int ReadDataAck(unsigned char *ucDATA)
{
	// Read 8 bits of data and ACK
	unsigned char shiftbyte;
	int i;
	int ucTempData, Status;

	shiftbyte = 0x00;

	// Ensure SDA is released (D1=0) and SCL is low (D0=1)
	ucTempData = _inp(LPTData);
	ucTempData = ((ucTempData & 0xFD) | 0x01);
	_outp(LPTData, ucTempData);
	DelayASMx10(DCNT/2);

	// loop for 8 bits
	for (i=0; i<8; i++)
	{	// Clock in one bit
		// Bring SCL high (D0=0)
		ucTempData = ucTempData & 0xFE;
		_outp(LPTData, ucTempData);
		DelayASMx10(DCNT/2);

		// Read in SDA
		Status = _inp(LPTStatus);

		// Bring SCL low (D0=1)
		ucTempData = ucTempData | 0x01;
		_outp(LPTData, ucTempData);
		DelayASMx10(DCNT/2);

		if (Status & 0x20)
		{	// Bit is high (although inverted through the 7405)
			shiftbyte = shiftbyte << 1;
			shiftbyte = shiftbyte & 0xFE; // Just in case compiler does not shift in 0
		}
```

```
                else
                {       // Bit is low (although inverted through the 7405)
                        shiftbyte = shiftbyte << 1;
                        shiftbyte = shiftbyte | 0x01;
                }
        }

        // ACK - Pull SDA low and clock SCL, then release SDA
        // Pull SDA low (D1=1)
        ucTempData = ucTempData | 0x02;
        _outp(LPTData, ucTempData);
        DelayASMx10(DCNT/2);

        // Bring SCL high (D0=0)
        ucTempData = ucTempData & 0xFE;
        _outp(LPTData, ucTempData);
        DelayASMx10(DCNT/2);

        // Bring SCL low (D0=1)
        ucTempData = ucTempData | 0x01;
        _outp(LPTData, ucTempData);
        DelayASMx10(DCNT/2);

        // Release SDA (D1=0)
        ucTempData = ucTempData & 0xFD;
        _outp(LPTData, ucTempData);
        DelayASMx10(DCNT/2);

        *ucDATA = shiftbyte;

        return 1;
}

int SetSlaveAddress(unsigned char ucADDR)    // sets slave address for
{       // WriteBytes and ReadBytes Commands
        ucDeviceAddress = ucADDR & 0xFE;
        return 1;
}
int WriteBytes(int iCount, unsigned char ucMemAddr, unsigned char ucData[256])
{       // Write upto 256 bytes to device address set by SetSlaveAddress.
        int fail = 0;
        int i;

        fail |= !Start();
        fail |= !WriteData((unsigned char)((long)ucDeviceAddress & 0xFE));
        fail |= !WriteData(ucMemAddr);
        for(i = 0; i <iCount; i++)
        {
                fail |= !WriteData(ucData[i]);
        }
        fail |= !Stop();

        if(fail)
                return 0;
        else
                return 1;
}

int ReadBytes(int iCount, unsigned char ucMemAddr, unsigned char ucData[256])
{       // Reads upto 256 bytes from device address set by SetSlaveAddress.
        int fail = 0;
        int i;

        fail |= !Start();
        fail |= !WriteData((unsigned char)((long)ucDeviceAddress & 0xFE));
        fail |= !WriteData(ucMemAddr);
        fail |= !Start();
        fail |= !WriteData((unsigned char)((long)ucDeviceAddress | 0x01));
        for(i = 0; i <(iCount-1); i++)
        {
                fail |= !ReadDataAck(&ucData[i]);
        }
        fail |= !ReadDataNack(&ucData[i]);
        fail |= !Stop();

        if(fail)
                return 0;
        else
                return 1;
}

int ResetBus()
{
        unsigned char ucTempData;
        int i;
```

```
        ucTempData = _inp(LPTData);

        for (i = 0; i < 9; i++)
        {
                // Bring SCL low (D0=1)
                ucTempData = ucTempData | 0x01;
                _outp(LPTData, ucTempData);
                DelayASMx10(DCNT/2);

                // Bring SCL high (D0=0)
                ucTempData = ucTempData & 0xFE;
                _outp(LPTData, ucTempData);
                DelayASMx10(DCNT/2);
        }
        Stop();
        return 1;
}

int     SetStrobe()           // sets strobe pin (LED) for debuging (AN3230)
{
        EnableLED();
        //DisableLED();
        return 1;
}
int ClearStrobe()     // clears strobe pin (LED) for debugging (AN3230)
{
        DisableLED();
        //EnableLED();
        return 1;
}
```