



Using the NS486SXF Evaluation Board Access.bus Interface
NS486SXF Application Note

Version 1.0
April 22, 1997

Rhinda Bal-Firpo
Software Engineer
National Semiconductor Corporation

1.0 Overview

The NS486SXF Access.bus is a two wire serial interface that allows bi-directional communication between ICs. This industry standard permits easy interfacing to a wide range of low-cost specialty memories and I/O devices. These include EEPROMs, SRAMs, timers, A/D converters, D/A converters, clock chips, and peripheral drivers. Access.bus supports multiple masters and multiple slaves.

The Access.bus is derived from the I²C serial bus developed by Phillips. Access.bus has additional refinements and the ability to drive cables for desktop peripheral use. For this application note, I²C and Access.bus can be considered identical. In the NS486SXF the MICROWIRE and the Access.bus interfaces share the use of the SCLK (pin #43) and SI (pin #42). Only Microwire or Access.bus can be selected at any one time.

This application note introduces the LM75 Digital Temperature Sensor, which was used to test the Access.bus interface. In addition to this, the application note describes the software modules used to test the functionality of the Access.bus interface with the LM75. Please refer to the NS486SXF data book for more detailed information on the Access.bus interface.

2.0 LM75 Operation

The LM75 is a temperature sensor, Delta-Sigma analog-to-digital converter, and digital over-temperature detector with the I²C interface. The host can query the LM75 at any time to read the temperature. The LM75 has a separate open-drain Overtemperature Shutdown (O.S.) output pin that can operate in either "Interrupt" or "Comparator" mode. This output becomes active when the temperature exceeds a programmable limit. The LM75's 3.0V to 5.5V supply voltage range, low supply current, and I²C interface make it ideal for a wide range of applications. These include thermal management and protection applications in personal computers, electronic test equipment, and office electronics. The LM75 has a pointer register that is programmed to select one of the four internal LM75 registers. The internal four registers are:

1. Temperature
 - at power-up the LM75 pointer register is pointing at this Temperature register
 - address 0x00
 - two bytes, read only register
2. T_{OS}
 - temperature alarm threshold
 - default thresholds at 80 °C
 - address 0x02
 - two bytes, read/write register
3. T_{HYST}
 - temperature at which the alarm condition goes away
 - default at 75 °C
 - address 0x03
 - two bytes, read/write register
4. Configuration
 - sets the operating modes; Comparator or Interrupt modes

- sensor powers up in Comparator mode
- address 0x01
- one byte, read/write register

Temperature accuracy is -25 °C to 100 °C with ± 2 °C or -55 °C to 125 °C with ± 3 °C.

The LM75 operates as a slave on the I²C bus, so the SCL line is an input (no clock is generated by the LM75) and the SDA line is a bi-directional serial data path. As dictated by the I²C bus specifications, the LM75 has a 7-bit slave address. The four most significant bits of the slave address are hardwired inside the LM75 and are “1001”. The three least significant bits of the address are assigned to pins A2-A0, and are set by connecting these pins to ground for a low, (0); or to +Vs for a high, (1). The least significant bit is the Read (1) / Write (0) bit; see Figure 1.0.

| | | | | | | | |
|---|---|---|---|----|----|----|-----|
| 1 | 0 | 0 | 1 | A2 | A1 | A0 | R/W |
|---|---|---|---|----|----|----|-----|

Figure 1.0, address of the slave LM75

The ambient temperature can be read from the Temperature register (this temperature reading will be the temperature of the surrounding area of the LM75). Also, an overtemperature condition can be set by writing the desired values to the T_{OS} and the T_{HYST} registers. For instance, if the LM75's ambient temperature is to reach the value in the T_{OS} register, indicating an overtemperature condition; if the LM75 is in the Comparator mode, then the open-drain Overtemperature Shutdown (O.S.) output reacts like a thermostat. The output can be used to turn on a cooling fan, initiate an emergency shutdown, or reduce system clock speed. The output remains active until the temperature drops below the T_{HYST} limit. If it is in the Interrupt mode, and the temperature exceeds the T_{OS} limit, the O.S. becomes active and remains active indefinitely until it is reset by reading any register via the I²C. Both the T_{OS} and T_{HYST} registers can be read. Temperature data is represented by a 9-bit two's complement word with the LSB (least significant bit) equal to

0.5 °C. Please refer to the sample code and figure 1.1 on reading and writing data bytes. Please refer to the LM75 datasheet for more details.

A write to the LM75 consists of an address byte, and the pointer byte.

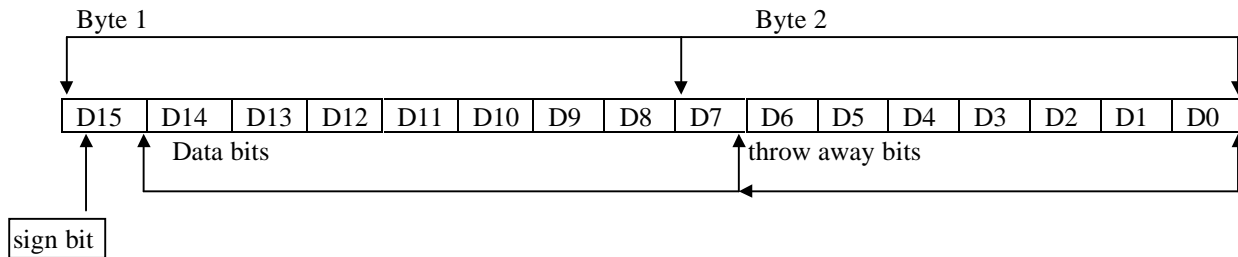
- A write to the Configuration register consists of one byte.
- A write to the T_{OS} and T_{HYST} registers consists of two bytes.

A read from the LM75 can be done in two ways:

1. If the pointer register is latched to the correct location then a read is simply an address byte and then receiving the number of corresponding data bytes.
2. If the pointer register needs to be changed; then a read consists of an address byte, a pointer byte, repeat start and another address byte; then reading the corresponding number of data bytes.

For more details please look at the sample code and read the code descriptions below.

Figure 1.1; reading the data bytes for Temperature, T_{OS} and T_{HYST} registers.



3.0 Access.bus Protocol Overview

The Access.bus interface signals are required to be driven by open-drain devices, so the system is required to pull up these signals. The two interface serial lines, the serial data line (SI), and the serial clock line (SCLK) should be connected to a positive supply via a pull-up resistor and are to remain HIGH when the bus is not busy. The supply voltage (V_{DD}) and the maximum output LOW level determine the minimum value of pull-up resistor R_p . For example, with a supply voltage of $V_{DD} = 5V \pm 10\%$ and $V_{OL, max} = 0.4V$ at 3 mA, $R_{p, min} = (5.5 - 0.4)/0.003 = 1.7k\Omega$. (Phillips Semiconductor I²C-bus datasheet.) 2.2k Ω pull-up resistors were used in this test.

Each device connected to the Access.bus has a unique address and can operate as a transmitter and/or a receiver. During the data transfer a device can either be a master or slave. The initiating device is considered the master, it also generates the clock (SCLK) and the start condition for the transfer. The addressed device is considered to be the slave during the transfer. When the NS486SXF initiates a data transfer to the Access.bus peripheral, it is the master and the peripheral is the slave. When the peripheral responds and sends data back the NS486SXF, it is a transmitter (even though it still remains the slave), and the NS486SXF is the receiver (even though it still remains the master). Therefore, both the master and the slave can be a receiver and transmitter. The key is that the initiator of the transfer provides the clock signal and is considered the present Access.bus master. The bus is busy after the generation of a START condition and will be free after some time interval after the STOP condition. A STOP condition is generated right before the last byte is read from or written to the slave. After this last step is executed the transmits or receives have stopped, and a START condition by the master is needed to transmit or receive data from the slave again. START and STOP conditions are generated by the present master.

Data is transferred at one byte at a time with the most significant bit first and least significant bit last. After the least significant bit of each byte is received the receiving side must generate an Acknowledge (bit [0] in the Access.bus Control register 1, at address 0x55). The following bits in the Access.bus Control Register 1 at address 0x55, are checked to see if the data byte is received or transmitted correctly.

When the Access.bus is in Master Transmit mode:

- ACK bit [0], Acknowledge flag. This bit is a status bit for transmit modes. If the slave sends an acknowledge of receiving the byte, the ACK bit [0] should be zero. If

it is one, then the master will abort and generate a STOP condition. Software must clear this by writing a zero to this bit.

- ABINT bit [3], Access.bus interrupt flag will be set to a 1 at the end of a transfer by hardware. Software must clear this bit by writing a zero to it.
- BUSY bit [1], Busy Transmitting flag. Software must set this bit to a one before transmitting or receiving data. This bit will be cleared to a zero by hardware at the end of an Access.bus transfer cycle.

When the Access.bus is in Master Receive mode:

- ACK bit [0], Acknowledge flag. This bit is a control bit for receive mode, and is under software control. Setting this bit to a one, will result in the generation of an Acknowledge at the appropriate time. Clearing this bit to a zero will result in no Acknowledge generation.
- ABINT bit [3], Access.bus interrupt flag will be set to a 1 at the end of a transfer by hardware. Software must clear this bit by writing a zero to it.
- BUSY bit [1], Busy Transmitting flag. Software must set this bit to a one before transmitting or receiving data. This bit will be cleared to a zero by hardware at the end of an Access.bus transfer cycle.

These bits should be checked in a loop for several thousand times. Refer to the NS486SXF databook for more details on this register. Also, refer to “Things to be aware of when writing code for the Access.bus Interface” section in this application note.

Both the slave and the master must generate an acknowledge after a reception. If the receiving side did not receive the data immediately, it will force the transmitter into a wait state by holding the SCLK low. If this happens, the master, after 2 - 3 msec., will abort the transfer and set bit 5 (overflow bit) in the Access.bus Control Register 1 (NOTE: this will definitely happen if code is stepped through a debugger). Software should check this condition and resolve it. This might leave the Access.bus in a unrecoverable state. Data will not be transmitted or received correctly.

There are two situations when the an acknowledge is not required after the reception of each byte:

1. When the master is receiving data, before receiving the last byte, master signals an end of transmit by generating a STOP condition and does NOT generate an acknowledge.
2. When the slave sends a negative acknowledge which indicates that it can no longer accept any more data. This can happen when the receive FIFO is full or the receiver is recovering from some error condition. If the slave does send a negative acknowledge the master will abort the transmit and generate a STOP condition.

3.1 Checking the Acknowledge Bit

There is an algorithm used to determine when to check the acknowledge bit. The acknowledge bit is a status bit for transmit mode and a control bit for receive mode. At the beginning of a transmit, (with the exception of the initial transmit) ACK is a “don’t care”. After the transmit the ACK is checked to see if data was received by the receiver. This is reverse for receiving. When master is in the master receive mode, the ACK bit set to one before the data is read. Please refer to sample code for more specifics. When the master is initiating a START condition, the Serial

Input/Output DATA Register (SDA) is written to first then the Access.bus Control Register 1 is written. When a “repeat start” is needed, the Access.bus Control Register 1 is written to first, then the SDA register is written to. Please see sample code on reading and writing to the slave. Any number of bytes can be transferred in a given transfer sequence.

3.2 Calculate clock divide down number for the Access.bus

The Access.bus interface supports a 0 - 400kHz serial clock frequency. Bits 7 - 3 of the MICROWIRE/Access.bus Control Register, 0x50, may be programmed to generate these required serial clock frequencies. NS486SXF databook, version March 1996 and earlier, has an error. In the MICROWIRE/Access.bus Control Register the reference to the OSCX1 clock should be CPU_CLK. Also, the Access.bus supports a 0 - 400kHz instead of 0 - 100kHz serial clock frequency.

Programming the clock prescaler; MICROWIRE/ Access.bus Control Register, NS486SXF data sheet. Bits SL [1:0] and Con [0:2].

If the NS486SXF is running at 25MHZ and the slave is at X Hz ($X < 400k$), and the divide down number is Y, then $Y = 25M/X$. Choose two numbers from the tables, such that when the numbers are multiplied they give you a number closest to Y.

Example: To run the Access.bus at 100kHz then:

$$X = 100kHz$$

$$Y = \text{CPU_CLK frequency} / X, \text{ therefore}$$

$$Y = 25000 / 100 = 250$$

We choose SL [1:0] to “10” which corresponds to CPU_CLK / 32 and Con [2:0] to be “011” for $(\text{CPU_CLK}/32) / 8$; therefore the actual Access.bus frequency will be:

$$25,000,000 / 32 / 8 = \sim 98kHz$$

We choose 32 in the first table and 8 in the second table, since $32 \times 8 = 256$. This is the closest number to 250. Refer to the NS486SXF databook for the MICROWIRE/Access.bus Control Register. Below are the tables for SL and Con copied from the NS486SXF databook:

| SL1 | SL0 | Input OSCX1 clock divided by |
|-----|-----|---------------------------------|
| 0 | 0 | 8 |
| 0 | 1 | 16 |
| 1 | 0 | 32 |
| 1 | 1 | 64 |

| Con2 | Con1 | Con0 | SL1 and SL0 selected prescaled Clock divided by |
|------|------|------|---|
| 0 | 0 | 0 | 2 |
| 0 | 0 | 1 | 4 |
| 0 | 1 | 0 | 6 |
| 0 | 1 | 1 | 8 |
| 1 | 0 | 0 | 10 |
| 1 | 0 | 1 | 12 |
| 1 | 1 | 0 | 14 |
| 1 | 1 | 1 | 16 |

Con0, Con1, Con2, SL1, and SL0 should be programmed according to the frequency of the input CPU_CLK clock. These five bits determine frequency of the clock used in this block., and are used to optimize performance at differ oscillator speeds.

4.0 Connecting the LM75 demo board to the Evaluation Board

To run the NSDEMO sample I²C code on the NS486SXF evaluation board, a custom cable must be made to connect an LM75 demonstration board, which is available from National Semiconductor, to the evaluation board. The LM75 demo board includes a parallel port for use with a PC. This port is not used with the evaluation board, as the NS486SXF has a true Access.bus device whose signals go to the feature connector on the evaluation board.

The LM75 demo board does have a row of pins with each of the LM75 pins. A simple connector can be made that connects the relevant pins (SDA, SCL, GND, V+) to the evaluation board. A schematic of this connection is included in this document.

5.0 Things to be aware of when writing code for the Access.bus Interface:

- Check if the Access.bus is busy with any other devices by checking the BB (bit 3) in the Access.bus Status register 0x54.
- Give ample time for the transmits and receives to occur. In the Access.bus Control Register 1, 0x55, in a while loop check for the ABINT bit [3] to go to one, the BUSY bit [1] to go to zero. If the Access.bus is in a Master transmit mode check for the ACK bit [0] for one; if one then the transmit was not acknowledged by the slave and the master will abort the transmit. In Master or Slave receive mode the ACK bit is under software control. The while loop should loop through several thousand times.
- Make sure the serial clock (SCLK) is not held low for more than 2-3 msec. This is bit 5 in the Access.bus Control Register 1, 0x55. This will cause an overflow and the transmit should be aborted at this time. This will definitely happen when stepping through the code during debugging. Therefore, step-debugging is not a good idea. Also, if the wait time in the while loop is too long (corresponds to above comment).
- Make sure the pull-up resistors are present.

6.0 Sample code to Read/Write using the Access.bus Interface

To run the sample code, simply plug the LM75 demo board to the NS486SXF feature connector as described earlier. When NSDEMO is ran, the sample code will detect the LM75 and perform several simple tests.

Below is a brief explanation of functions used to test the Access.bus interface. This software is part of NSDEMO, version 1.7.

USHORT I²C_Initialize()

Checks if the Access.bus/MICROWIRE is initialized. Sets up the Access.bus interface.
Returns FAIL if the Access.bus is not initialized.
Returns SUCCESS if the Access.bus is initialized

USHORT I²C_Data_Read(BYTE SlaveAddress, int reg_read, int numberbytes, BYTE *data_read)

the Sets up the Access.bus to read from the slave. Activates the desired slave by sending the slaves address and the read bit. Sets the pointer register to read the correct register in slave. Reads data from the slave, currently reading two bytes only.

Parameters:

BYTE SlaveAddress - takes in the slave address.

int reg_read - takes in the slave register to read.

int numberbytes - takes in the number of bytes to read, currently only reading 2 bytes.

BYTE *data_read - outputs the data read to the calling function.

Returns FAIL if unable to read data.

Returns SUCCESS if data was read.

USHORT I²C_Data_Write(BYTE SlaveAddress, int reg_write, int bytes_write, BYTE *data_write)

Sets up the Access.bus to write data to the slave. Activates the desired slave by sending the address and the write bit. Sets up the pointer register in the slave to write to the correct location in the slave. Transmits the data bytes to the slave.

Parameters:

BYTE SlaveAddress - takes the slave address.

int reg_write - takes in the slave register that needs to be written.

int bytes_write - takes in the number of bytes that needs to be written, currently only writing two bytes.

BYTE *data_write - takes in the data that needs to be written

Returns FAIL if unable to write to the slave.

Returns SUCCESS if wrote data successfully.

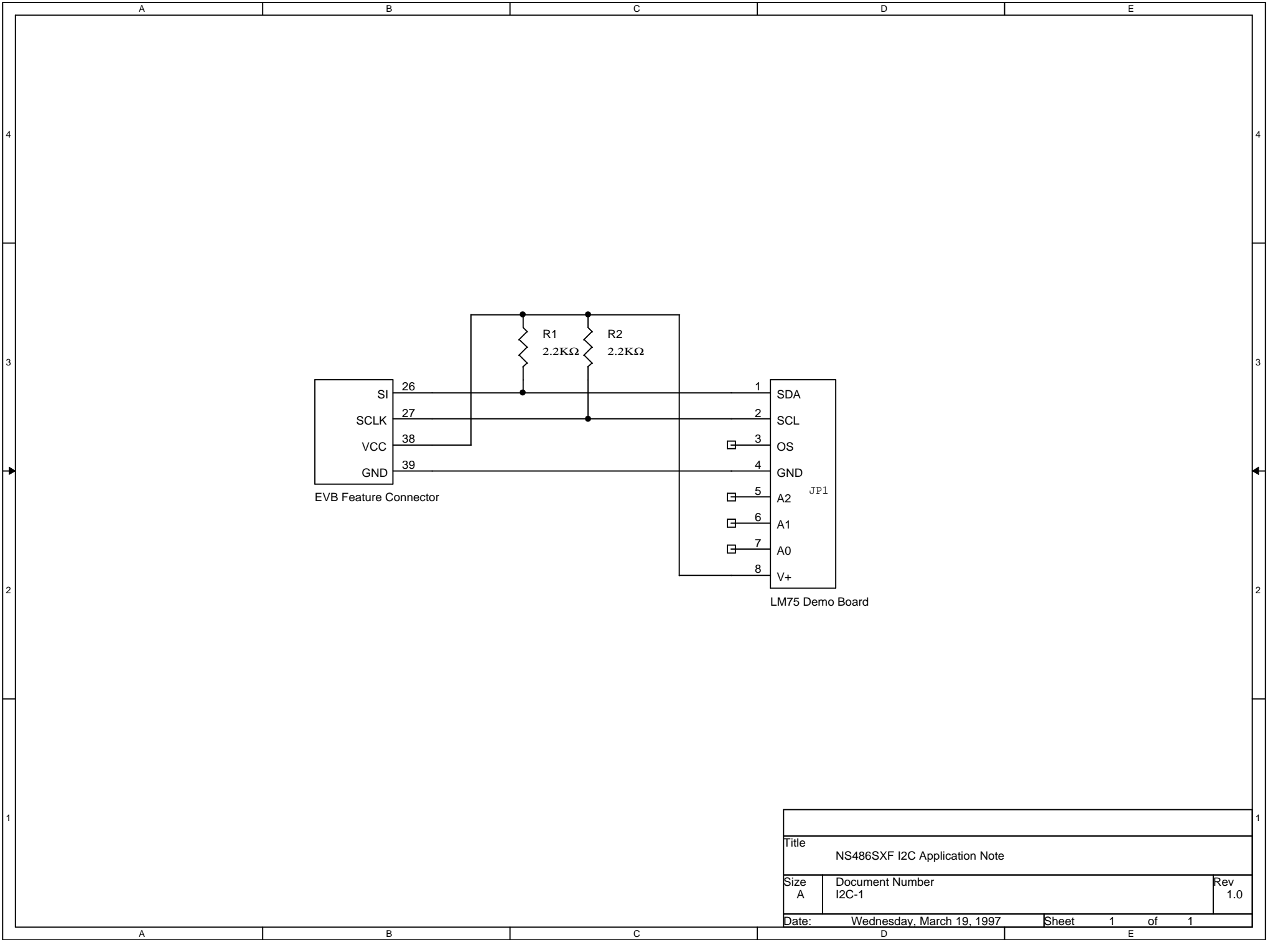
References: NS486TMSXF Optimized 32-bit 486-class Controller with On-Chip Peripherals for Embedded Systems Datasheet, National Semiconductor.

LM75 Digital Temperature Sensor and Thermal Watchdog with Two-Wire Interface, National Semiconductor, Lit # 106488-002.

The I²C-bus and how to use it, Phillips Semiconductor.

Appendices

- Schematic of LM75 demo board to NS486SXF Evaluation Board connector
- NSDEMO I2C code
- First page of LM75 data sheet



| | | |
|---------------------------------|-----------------|-----|
| Title | | |
| NS486SXF I2C Application Note | | |
| Size | Document Number | Rev |
| A | I2C-1 | 1.0 |
| Date: Wednesday, March 19, 1997 | | |
| Sheet 1 of 1 | | |

NSDEMO I2C code - I2C.C

```
//-----  
//  
// %FILE      i2c.c  
// %VSS-REV   $Revision: 5 $  
// %CREATED   1997.03.04  
// %REVISED   $Date: 4/16/97 3:04p $  
// %AUTHOR    Rhinda Bal-Firpo  
// %PROJECT    NS486SXF evaluation board software  
// %PART       NS486SXF, NS486SXL  
// %SUMMARY    i2c (Access.bus) module  
//  
// %VSS        $Author: Rhinda $ $Date: 4/16/97 3:04p $ $Revision: 5 $  
//  
// DESCRIPTION  
//  
//   This file contains code that will initialize the i2c and perform read and  
//   writes from the i2c.  
//  
//   I2C and Access.bus are synonymous.  
//  
// HISTORY  
//  
/*  
*  
* $History: I2C.C $  
*  
* ***** Version 5 *****  
* User: Rhinda      Date: 4/16/97      Time: 3:04p  
* Updated in $/nsdemo  
* needed it to do some testing  
*  
* ***** Version 4 *****  
* User: Miked       Date: 4/11/97      Time: 6:27p  
* Updated in $/nsdemo  
* Formatting changes. Also fixed usage of I2C_SET_BUSY and I2C_BUSY.  
*  
* ***** Version 3 *****  
* User: Rhinda      Date: 4/01/97      Time: 6:45p  
* Updated in $/nstest  
* maintenance changes  
*  
* ***** Version 2 *****  
* User: Rhinda      Date: 3/21/97      Time: 4:03p  
* Updated in $/nsdemo  
* Made many changes to initial I2C code.  
*  
*/  
//  
// COPYRIGHT  
//  
//   (c) 1997 National Semiconductor Corporation  
//  
// NOTES  
//  
//   In this file the i2c (Access.bus) is in the master transmit and master  
//   receive mode. The slave used to test this file is the LM75 Digital  
//   Temperature Sensor. Data is transmitted to and received from the LM75.  
//   Please see the application note for more information.  
//  
//-----
```

```

#include "i2c.h"

//-----
//
// FUNCTION      I2c_Initialize()
//
// INPUT         none
// OUTPUT        none
// RETURN        USHORT
//               FAIL - If bus is unable to initialize
//               SUCCESS - if successfull
// DESCRIPTION
//
//       This function is called to initialize the i2c interface.
//
//-----

USHORT I2C_Initialize()
{
    // initialize the MicroWire
    if ((IOR_BYTE(BIU_CONTROL1) & 0x80) != 0x80)
        return FAIL;

    // configure the Access.Bus; I2C
    IOW_BYTE( TWI_CONTROL, 0x9f ); // selects access.bus and does software reset
    IOW_BYTE( TWI_CONTROL, 0x9e ); // selects access.bus and enables
    IOW_BYTE( TWI_A_CTRL2, 0x03 ); // set the OSCX1 Clock periods

    // all done
    return SUCCESS;
}

//-----
//
// FUNCTION      I2c_Data_Read(int reg_read, int numberbytes, BYTE *data_read)
//
// INPUT         BYTE SlaveAddress: I2C address of device to read
//               int reg_read: address of register that will be read.
//               int numberbytes: number of bytes that will read; this is not
//                               being used currently; currently we are reading
//                               2 bytes.
//
// OUTPUT        BYTE *data_read: pointer to the array that will hold the data
//                               read
//
// RETURN        USHORT
//               FAIL - If unable to read data
//               SUCCESS - if successful
//
// DESCRIPTION
//
//       This function is called to read data from I2C device.
//
//-----

USHORT I2C_Data_Read(BYTE SlaveAddress, int reg_read, int numberbytes,
                    BYTE *data_read)
{
    BYTE busy;           //reading of the register is stored in this
    int i = 0;           //counter for the while loops

```

```

//set the access.bus to read the slave
//write the address byte, and set the access.bus for writing to the pointer
//address

IOW_BYTE (TWI_SDA, (SlaveAddress | I2C_SLAVE_WRITE));

//If the access.bus is a multiple master or slave, it might be busy with other
//check if access.bus is busy with other transmits or receives.

busy = IOR_BYTE(TWI_A_STATUS);
while (((busy & I2C_BB) == I2C_BB) && (i < 10000))
{
    i++;
    busy = IOR_BYTE(TWI_A_STATUS);
}

//This should only happen if the i2c is a multiple master or slave bus
if ((busy & I2C_BB) == I2C_BB)
    return FAIL;

//setup access.bus control register for initial byte transfer
// set master mode, set BUSY and ACK
IOW_BYTE (TWI_A_CONTROL, ((I2C_MASTRQ | I2C_SET_BUSY) | I2C_ACK) );

//wait for transfer to finish; check the control register
i=0;
busy = IOR_BYTE(TWI_A_CONTROL);
while(((busy & ( I2C_SET_BUSY | I2C_ABINT | I2C_ACK)) !=
      ( I2C_BUSY | I2C_ABINT | I2C_ACK)) && (i < 10000))
{
    i++;
    busy = IOR_BYTE(TWI_A_CONTROL);
}

//The transfer did not complete
if ((busy & ( I2C_SET_BUSY | I2C_ABINT | I2C_ACK)) !=
    ( I2C_BUSY | I2C_ABINT | I2C_ACK))
    return FAIL;

//Set the pointer register to point to the register to read in the slave
IOW_BYTE(TWI_SDA, reg_read);
IOW_BYTE(TWI_A_CONTROL, I2C_SET_BUSY); //set BUSY

//wait for transfer to finish; check the control register
i=0;
busy = IOR_BYTE(TWI_A_CONTROL);
while(((busy & ( I2C_SET_BUSY | I2C_ABINT | I2C_ACK)) !=
      ( I2C_BUSY | I2C_ABINT | I2C_ACK)) && (i < 10000))
{
    i++;
    busy = IOR_BYTE(TWI_A_CONTROL);
}

//The transfer did not complete
if ((busy & ( I2C_SET_BUSY | I2C_ABINT | I2C_ACK)) !=
    ( I2C_BUSY | I2C_ABINT | I2C_ACK))
    return FAIL;

//setup the i2c to read the data
//repeat start; set BUSY and MASTRQ, 0x12
IOW_BYTE(TWI_A_CONTROL, (I2C_MASTRQ | I2C_SET_BUSY));
//set the address of the slave and set for read
IOW_BYTE(TWI_SDA, (SlaveAddress | I2C_SLAVE_READ));

```

```

//wait for transfer to finish; check the control register
i=0;
busy = IOR_BYTE(TWI_A_CONTROL);
while(((busy & ( I2C_SET_BUSY | I2C_ABINT | I2C_ACK)) !=
      ( I2C_BUSY | I2C_ABINT | I2C_ACK)) && (i < 10000))
{
    i++;
    busy = IOR_BYTE(TWI_A_CONTROL);
}

//The transfer did not complete
if ((busy & ( I2C_SET_BUSY | I2C_ABINT | I2C_ACK)) !=
    ( I2C_BUSY | I2C_ABINT | I2C_ACK))
    return FAIL;

//program to receive data
IOW_BYTE(TWI_A_CONTROL, (I2C_SET_BUSY | I2C_ACK)); //set BUSY and ACK

i = 0;
busy = IOR_BYTE(TWI_A_CONTROL);
while(((busy & I2C_ABINT) != I2C_ABINT) && (i < 10000))
{
    i++;
    busy = IOR_BYTE(TWI_A_CONTROL);
}

//The data was not received by i2c
if ((busy & I2C_ABINT) != I2C_ABINT)
    return FAIL;

//Read the slave temperature, currently pointing to the temperature register
data_read[0] = IOR_BYTE(TWI_SDA);

//Generate STOP condition before reading the last byte, and set for NO ACK
IOW_BYTE (TWI_A_CONTROL, (I2C_SET_BUSY | I2C_STOP));

//wait for receive to finish
i = 0;
busy = IOR_BYTE(TWI_A_CONTROL);
while(((busy & I2C_SET_BUSY) == I2C_SET_BUSY) && (i < 10000))
{
    i++;
    busy = IOR_BYTE(TWI_A_CONTROL);
}

//Read the last byte of temperature data from slave
data_read[1] = IOR_BYTE(TWI_SDA);

//Transfer is finished
return SUCCESS;
}

//-----
//
// FUNCTION      I2c_Data_Write(int reg_write, int bytes_write, BYTE *data_write)
//
// INPUT         BYTE SlaveAddress:  I2C address of device to read
//               int reg_write:  register to write to
//               int bytes_write: number of bytes to write
//               BYTE *data_write: pointer to the array holding the data to send
//                               to the slave
//
// OUTPUT        none

```

```

//
// RETURN          USHORT
//                  FAIL - If unable to read data from slave
//                  SUCCESS - successful
// DESCRIPTION
//
//      This function is called to send data to the slave
//
//-----
USHORT I2C_Data_Write(BYTE SlaveAddress, int reg_write, int bytes_write,
                     BYTE *data_write)
{
    BYTE busy;          //reading of the register is stored in this
    int i = 0;          //counter for the while loops

    //set the access.bus to write the slave
    //write the address byte, and set the R/W bit for write
    IOW_BYTE (TWI_SDA, (SlaveAddress | I2C_SLAVE_WRITE));

    //If the access.bus is a multiple master or slave, it might be busy with other
    //check if access.bus is busy with other transmits or receives.
    busy = IOR_BYTE(TWI_A_STATUS);
    while (((busy & I2C_BB) == I2C_BB) && (i < 10000))
    {
        i++;
        busy = IOR_BYTE(TWI_A_STATUS);
    }

    //This should only happen if the i2c is a multiple master or slave bus
    if ((busy & I2C_BB) == I2C_BB)
        return FAIL;

    //setup access.bus control register for initial byte
    // set master mode, set BUSY and ACK
    IOW_BYTE (TWI_A_CONTROL, ((I2C_MASTRQ | I2C_SET_BUSY) | I2C_ACK));

    //wait for transfer to finish; check the control register
    i=0;
    busy = IOR_BYTE(TWI_A_CONTROL);
    while(((busy & ( I2C_SET_BUSY | I2C_ABINT | I2C_ACK)) !=
           ( I2C_BUSY | I2C_ABINT | I2C_ACK)) && (i < 10000))
    {
        i++;
        busy = IOR_BYTE(TWI_A_CONTROL);
    }

    //The transfer did not complete
    if ((busy & ( I2C_SET_BUSY | I2C_ABINT | I2C_ACK)) !=
        ( I2C_BUSY | I2C_ABINT | I2C_ACK))
        return FAIL;

    //program to write data
    //set the pointer register to point to the needed register
    IOW_BYTE(TWI_SDA, reg_write);
    // set BUSY and ACK
    IOW_BYTE (TWI_A_CONTROL, I2C_SET_BUSY);

    //check the ABINT bit and ACK bit to see if transfer completed
    i = 0;
    busy = IOR_BYTE(TWI_A_CONTROL);
    while(((busy & ( I2C_SET_BUSY | I2C_ABINT | I2C_ACK)) !=

```



```

        ( I2C_BUSY | I2C_ABINT | I2C_ACK)) && (i < 10000))
    {
        i++;
        busy = IOR_BYTE(TWI_A_CONTROL);
    }

    //Access.bus is not ready to write data
    if ((busy & ( I2C_SET_BUSY | I2C_ABINT | I2C_ACK)) !=
        ( I2C_BUSY | I2C_ABINT | I2C_ACK))
        return FAIL;

    //write the data to the appropriate register
    IOW_BYTE( TWI_SDA, data_write[0] );
    IOW_BYTE (TWI_A_CONTROL, I2C_SET_BUSY); //BUSY and ACK

    //check the ABINT bit and ACK bit to see if transfer completed
    i = 0;
    busy = IOR_BYTE(TWI_A_CONTROL);
    while(((busy & ( I2C_SET_BUSY | I2C_ABINT | I2C_ACK)) !=
        ( I2C_BUSY | I2C_ABINT | I2C_ACK)) && (i < 10000))
    {
        i++;
        busy = IOR_BYTE(TWI_A_CONTROL);
    }

    //Access.bus is not ready to write data
    if ((busy & ( I2C_SET_BUSY | I2C_ABINT | I2C_ACK)) !=
        ( I2C_BUSY | I2C_ABINT | I2C_ACK))
        return FAIL;

    //Generate STOP condition before sending the last byte
    IOW_BYTE( TWI_SDA, data_write[1] );
    IOW_BYTE (TWI_A_CONTROL, (I2C_SET_BUSY | I2C_STOP));

    //wait for the write to finish
    i = 0;
    busy = IOR_BYTE(TWI_A_CONTROL);
    while(((busy & I2C_SET_BUSY) == I2C_SET_BUSY) && (i < 10000))
    {
        i++;
        busy = IOR_BYTE(TWI_A_CONTROL);
    }

    //Transfer is finished
    return SUCCESS;
}

//-----
//  END          i2c.c
//-----

```

NSDEMO I2C code - I2C.H

```
//-----
//
// %FILE      i2c.h
// %VSS-REV   $Revision: 4 $
// %CREATED   1997.03.04
// %REVISED   $Date: 4/16/97 3:03p $
// %AUTHOR    Rhinda Bal-Firpo
// %PROJECT    NS486SXF evaluation board software
// %PART       NS486SXF, NS486SXL
// %SUMMARY    include file I2C modules
//
// %VSS        $Author: Rhinda $ $Date: 4/16/97 3:03p $ $Revision: 4 $
//
// DESCRIPTION
//
//   This header file contains definitions and function prototypes for file i2c.c
//
// HISTORY
//
/*
 *
 * $History: I2C.H $
 *
 * ***** Version 4 *****
 * User: Rhinda      Date: 4/16/97      Time: 3:03p
 * Updated in $/nsdemo
 * needed to do some testing
 *
 * ***** Version 3 *****
 * User: Miked       Date: 4/11/97      Time: 6:27p
 * Updated in $/nsdemo
 * Formatting changes. Moved LM75 I2C address to LM75.h.
 *
 * ***** Version 2 *****
 * User: Rhinda      Date: 3/21/97      Time: 4:03p
 * Updated in $/nsdemo
 * Made many changes to initial I2C code.
 *
 */
//
// COPYRIGHT
//
//   (c) 1997 National Semiconductor Corporation
//
//-----

#ifndef I2C_H_INC
#define I2C_H_INC

//-----
//include files

#include "nsglobal.h"

//-----
// bit definitions

// TWI_SDA - used when writing address to select read or write operation
#define I2C_SLAVE_READ      0x01
#define I2C_SLAVE_WRITE    0x00
```

```

// TWI_A_STATUS
#define I2C_BB                                0x08

// TWI_A_CONTROL register

#define I2C_MASTRQ                            0x10
#define I2C_ABINT                            0x08
#define I2C_STOP                             0x04
#define I2C_SET_BUSY                         0x02
#define I2C_ACK                             0x01

// TWI_A_CONTROL - bit value
#define I2C_BUSY                             0x00

//-----
// sign for temperature

#define I2C_POSITIVE                        1    //temperature zero or above zero
#define I2C_NEGATIVE                       0    //temperatrue below zero

//-----
// External Function Prototypes

extern USHORT I2C_Initialize();
extern USHORT I2C_Data_Read(BYTE SlaveAddress, int reg_read, int numberbytes,
                           BYTE *data_read);
extern USHORT I2C_Data_Write(BYTE SlaveAddress, int reg_write, int bytes_write,
                             BYTE *data_write);

//-----

#endif // #ifndef I2C_H_INC

```

NSDEMO I2C code - LM75.C

```
//-----
//
// %FILE      lm75.c
// %VSS-REV   $Revision: 3 $
// %CREATED   1997.03.04
// %REVISED   $Date: 4/16/97 3:04p $
// %AUTHOR    Rhinda Bal-Firpo
// %PROJECT    NS486SXF evaluation board software
// %PART       NS486SXF, NS486SXL
// %SUMMARY    Access.bus test module using NSC LM75 demo board.
//
// %VSS        $Author: Rhinda $ $Date: 4/16/97 3:04p $ $Revision: 3 $
//
// DESCRIPTION
//
//   This file contains code to test the NS486 Access.bus (I2C) interface.
//   The National Semiconductor LM75 Digital Temperature Sensor is used
//   for this purpose.  A modified LM75 demo board must be connected to the
//   NS486 evaluation board feature connector for this code to run.  Please
//   see the application note "Using the Access.bus interface on the
//   NS486SXF Evaluation Board" for additional information on the hardware
//   and software.
//
// HISTORY
//
/*
 *
 * $History: Lm75.c $
 *
 * ***** Version 3 *****
 * User: Rhinda      Date: 4/16/97      Time: 3:04p
 * Updated in $/nsdemo
 * needed it to do some testing
 *
 * ***** Version 2 *****
 * User: Miked       Date: 4/11/97      Time: 6:29p
 * Updated in $/nsdemo
 * Re-did conversion routines (simpler).  Formatting changes.
 *
 * ***** Version 1 *****
 * User: Rhinda      Date: 3/21/97      Time: 4:04p
 * Created in $/nsdemo
 * Initial version of LM75 test code, which tests the I2C interface with a
 * National LM75 evaluation board.
 *
 */
//
// COPYRIGHT
//
//   (c) 1997 National Semiconductor Corporation
//
// NOTES
//
//   As documented in the LM75 datasheet, the LM75 supports temperatures
//   from -55 degrees to 125 degrees Celsius.
//
//-----

#include "lm75.h"

//-----
```

```

// INPUT          none
// OUTPUT         none
// RETURN         none
//
// DESCRIPTION    Performs a simple test of the Access.bus interface using
//                the LM75 demo hardware.

USHORT LM75()
{
    int degreeCelsius;    //temperature in degree Celsius
    BYTE value_read[2];   //array that stores the temperature read from the LM75
    BYTE value_write[2];  //array that stores the temperature to write to the LM75
    BOOL halfDegree;      //store 0.5 degree result

    // print message indicating start of test
    dprintf("Access.bus Test:\r\n");

    // initialize Access.bus interface
    if (I2C_Initialize() != SUCCESS)
    {
        dprintf(" Initialization - FAILED\r\n\n");
        return FAIL;
    }

    // try to read temperature

    if (I2C_Data_Read(LM75_ADDRESS, LM75_TEMP, 2, value_read) == FAIL)
    {
        dprintf(" Unable to read data! Could indicate failure, or\r\n\n");
        dprintf(" LM75 test board not plugged in to feature connector!\r\n\n");
        return FAIL;
    }

    // convert result to temperature and display

    LM75_Convert_to_Celsius(value_read, &degreeCelsius, &halfDegree);
    if (halfDegree == TRUE)
        dprintf(" Temperature register at %d.5 degrees Celsius.\r\n",
            degreeCelsius);
    else
        dprintf(" Temperature register at %d degrees Celsius.\r\n", degreeCelsius);

    // write and read Alarm Threshold Register to verify writes

    dprintf(" Setting Alarm Threshold Register to -35 degrees Celsius.\r\n");
    LM75_Convert_Write(-35, value_write);

    if (I2C_Data_Write(LM75_ADDRESS, LM75_TOS, 2, value_write) == FAIL)
    {
        dprintf(" Write to Alarm Threshold register - FAILED\r\n\n");
        return FAIL;
    }
    if (I2C_Data_Read(LM75_ADDRESS, LM75_TOS, 2, value_read) == FAIL)
    {
        dprintf(" Read from Alarm Threshold register - FAILED\r\n\n");
        return FAIL;
    }

    LM75_Convert_to_Celsius(value_read, &degreeCelsius, &halfDegree);
    if (halfDegree == TRUE)
        dprintf(" Alarm Threshold register is at %d.5 degree Celsius. \r\n",

```

```

        degreeCelsius);
else
    dprintf(" Alarm Threshold register is at %d degree Celsius. \r\n",
        degreeCelsius);

if ( degreeCelsius == -35 )
    dprintf(" Alarm Threshold register test - PASSED\r\n");
else
{
    dprintf(" Alarm Threshold register test - FAILED\r\n\n");
    return FAIL;
}

// thats all

dprintf("\r\n");
return SUCCESS;
}

//-----
// INPUT      BYTE value_read[2]: value read from LM75 temperature register
// OUTPUT     int *pDegree:      temperature in celsius (signed)
//           BOOL *pHalfDegree:  TRUE if LSB (0.5 degrees) is set
//                               -24 degrees would become -24.5 degrees
//                               52 degrees would become 52.5 degrees
// RETURN     none
//
// DESCRIPTION This function is called to convert the data read from the LM75
//             to degree Celsius
//
// NOTES      Temperature data is represented by a 9-bit, two's complement
//             word with an LSB equal to 0.5 degree Celsius.

void LM75_Convert_to_Celsius(BYTE value_read[2], int *pDegree,
                             BOOL *pHalfDegree)
{
    int CelsiusTemp;
    BYTE temperature;
    BOOL isNegative;

    temperature = value_read[0]; // this is temperature (except for 0.5
                                // degree portion) in 2's complement form

    // adjust if negative

    if ( temperature > 127 )
        isNegative = TRUE;
    else
        isNegative = FALSE;

    if ( isNegative == TRUE )
        temperature = ~temperature + 0x01;

    // assign to integer

    CelsiusTemp = (int) temperature; // at this point, this is the temperature
                                    // in an integer without sign or 0.5
                                    // degree portion

    // make negative if applicable

```

```

    if ( isNegative == TRUE )
        CelsiusTemp = ( 0 - CelsiusTemp ); // now this is temperature with correct
                                            // sign. still no 0.5 degree portion

    // assign to return variable

    *pDegree = CelsiusTemp;

    // set half degree flag accordingly

    if ( (value_read[1] & 0x80)==0x80 )
        *pHalfDegree = TRUE;
    else
        *pHalfDegree = FALSE;
}

//-----

// INPUT      int convert_value_write: temperature to convert
//                                     must be from -55 to 125 degrees C
//                                     half degrees are not supported
// OUTPUT     BYTE *pwrite_bytes:  data to write to LM75 register
// RETURN     none
//
// DESCRIPTION This function takes an input of the desired temperature and
// returns the proper LM75 register value for the given
// temperature
//
// This function will return 0 in pwrite_bytes if the input
// temperature is not valid.

void LM75_Convert_Write(int convert_value_write, BYTE write_bytes[2])
{
    // if out of range, return 0

    if ( ( convert_value_write < -55 ) || ( convert_value_write > 125 ) )
    {
        write_bytes[0] = 0;
        write_bytes[1] = 0;
        return;
    }

    // write_bytes[1] will always be 0 since 0.5 degree increments are not
    // supported in this function

    write_bytes[1] = 0;

    // if we cast the integer as a BYTE, C will automatically get the sign
    // and everything correct, since convert_value_write is signed.

    write_bytes[0] = (BYTE) convert_value_write;
}

//-----
// END      lm75.c
//-----

```

NSDEMO I2C code - LM75.H

```
//-----  
//  
// %FILE      lm75.h  
// %VSS-REV   $Revision: 3 $  
// %CREATED   1997.03.04  
// %REVISED   $Date: 4/16/97 3:04p $  
// %AUTHOR    Rhinda Bal-Firpo  
// %PROJECT    NS486SXF evaluation board software  
// %PART       NS486SXF, NS486SXL  
// %SUMMARY    Access.bus test module header  
//  
// %VSS        $Author: Rhinda $ $Date: 4/16/97 3:04p $ $Revision: 3 $  
//  
// DESCRIPTION  
//  
//   This header file contains definitions and function prototypes for the  
//   file lm75.c  
//  
// HISTORY  
//  
/*  
*  
* $History: lm75.h $  
*  
* ***** Version 3 *****  
* User: Rhinda      Date: 4/16/97      Time: 3:04p  
* Updated in $/nsdemo  
* needed it to do some testing  
*  
* ***** Version 2 *****  
* User: Miked       Date: 4/11/97      Time: 6:28p  
* Updated in $/nsdemo  
* Formatting changes, and moved LM75 Address to this file (from i2c.h).  
*  
* ***** Version 1 *****  
* User: Rhinda      Date: 3/21/97      Time: 4:04p  
* Created in $/nsdemo  
* Initial version of LM75 test code, which tests the I2C interface with a  
* National LM75 evaluation board.  
*  
*/  
//  
// COPYRIGHT  
//  
//   (c) 1997 National Semiconductor Corporation  
//  
//-----  
  
#ifndef LM75_H_INC  
#define LM75_H_INC  
  
//-----  
// include files  
  
#include "i2c.h"  
  
//-----  
// LM75 Address  
//  
// The three jumpers on the LM75 demo board set the low order three bits  
// of the address.  The high order four bits must be 1001B on the LM75.
```



```

// Therefore, the address is:
//
// 1001abcX
//
// Where a,b,c are the bits set by hte jumpers. For this code, the jumpers
// must be all on for an address of 1001000X.
//
// Bit 0 is used to determine read or write accesses, so it not set here.

#define LM75_ADDRESS 0x90

//-----
// LM75 registers

#define LM75_TEMP 0x00 // Temperature register; read-only; 2 bytes
#define LM75_CONT 0x01 // Configuration register; read-write, 1 byte
#define LM75_THYST 0x02 // Temperature at which the alarm goes away;
                        // read-write; 2 bytes
#define LM75_TOS 0x03 // Temperature alarm threshold; read-write; 2 bytes

//-----
// function prototypes

extern USHORT LM75();
extern void LM75_Convert_Write(int convert_value_write, BYTE write_bytes[2]);
extern void LM75_Convert_to_Celsius(BYTE value_read[2], int *pDegree,
                                   BOOL *pHalfDegree);

//-----

#endif // #ifndef LM75_H_INC

```

LM75 Digital Temperature Sensor and Thermal Watchdog with Two-Wire Interface

General Description

The LM75 is a temperature sensor, Delta-Sigma analog-to-digital converter, and digital over-temperature detector with I²C® interface. The host can query the LM75 at any time to read temperature. The open-drain Overtemperature Shutdown (O.S.) output becomes active when the temperature exceeds a programmable limit. This pin can operate in either "Comparator" or "Interrupt" mode.

The host can program both the temperature alarm threshold (T_{OS}) and the temperature at which the alarm condition goes away (T_{HYST}). In addition, the host can read back the contents of the LM75's T_{OS} and T_{HYST} registers. Three pins (A0, A1, A2) are available for address selection. The sensor powers up in Comparator mode with default thresholds of 80°C T_{OS} and 75°C T_{HYST}.

The LM75's 3.0V to 5.5V supply voltage range, low supply current, and I²C® interface make it ideal for a wide range of applications. These include thermal management and protection applications in personal computers, electronic test equipment, and office electronics.

Features

- Tiny SO-8 package saves space (SOT-8 under development)
- I²C® Bus interface
- Separate open-drain output pin operates as interrupt or comparator/thermostat output
- Register readback capability
- Power up defaults permit stand-alone operation as thermostat
- Shutdown mode to minimize power consumption
- Up to 8 LM75s can be connected to a single bus

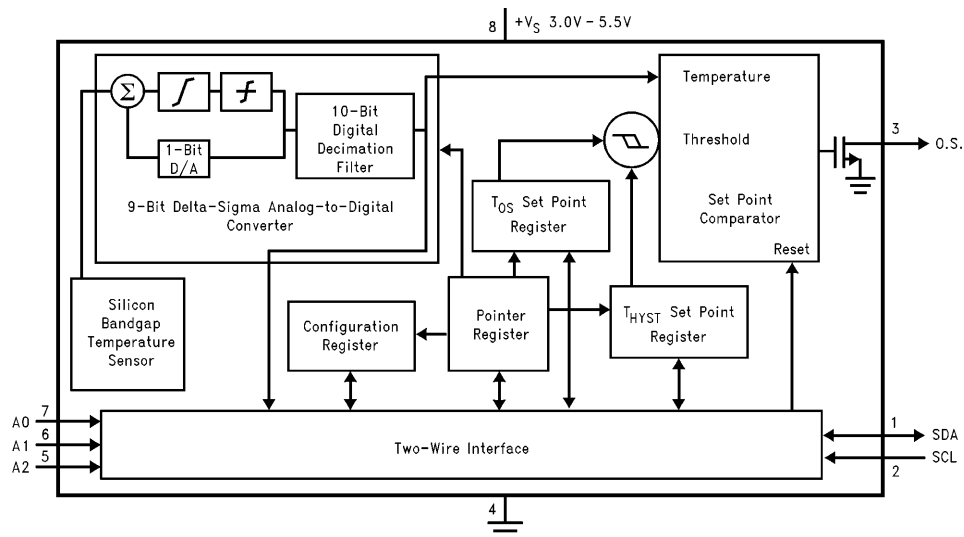
Key Specifications

| | |
|----------------------------|--|
| ■ Supply Voltage | 3.0V to 5.5V |
| ■ Supply Current operating | 250 μ A (typ) 1 mA (max) |
| ■ Supply Current shutdown | 1 μ A (typ) |
| ■ Temperature Accuracy | -25°C to 100°C \pm 2°C(max) -55°C to 125°C \pm 3°C(max) |

Applications

- System Thermal Management
- Personal Computers
- Office Electronics
- Electronic Test Equipment

Simplified Block Diagram



TL/H/12658-1

I²C® is a registered trademark of Phillips Corporation.