

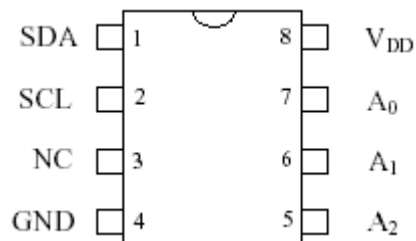
I²C Communication - the DS1624

[Jonathan Zacharko](#)

This project shows how to communicate with I²C Devices. The example shown below uses the DS1624 Digital Thermometer and Memory IC available from Dallas Semiconductor <http://www.dalsemi.com>.

The DS1624 is a good choice in temperature sensors because it requires no external components and has a wide temperature range of +125°C to -55°C. The DS1624 also has 256 bytes of available E²prom memory for general storage (although that will not be covered in this article).

DS1624 Pinout:



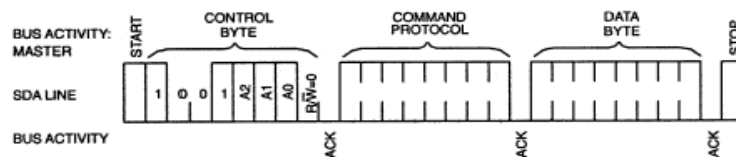
SDA	Data Pin	V _{DD}	V _{SUPPLY} (+5V)
SCL	Clock Pin	A0	I ² C Address bit 0
NC	No Connect	A1	I ² C Address bit 1
GND	Ground (0V)	A2	I ² C Address bit 2

For this experiment:

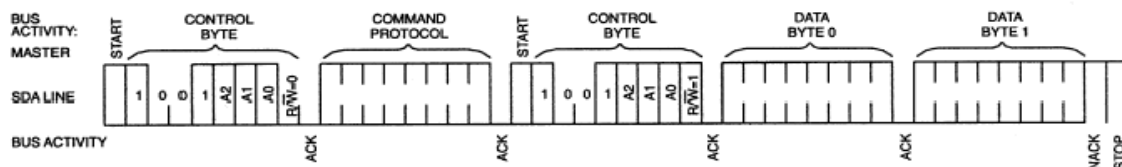
- SDA is connected to PortC.1
- SCL is connected to PortC.2
- A0, A1, A2 are all connected to GND
- V_{DD} is connected to +5V
- GND is connected to Ground

The command protocol for the DS1624 can be shown by the following diagram:

Write to DS1624



Read from DS1624



Before each read or write, a 4 bit device code along with a 3 bit address code and a read or write bit must be sent. This allows up to 8 devices of a particular type to be added on 2 I/O lines, a great advantage, but can be difficult to control in software. The 4 bit device code for the DS1624 is %1001. In this experiment, all address bits are logic low, therefore the address of this Temperature sensor is %000. If you wish to add multiple temperature sensors to the same bus, then just tie each address pin high or low and change the address accordingly. Depending upon which operation is desired to be performed, bit 0 of the control byte will either be 0 for a "write" command or 1 for a "read" command.

Onto the code....

'DS1624 Temperature sensor code example

'A pic with a hardware USART was used to send serial data to a terminal @ 19200N1, but an LCD 'could easily be added instead. If using the USART, a level translator such as the MAX232 is 'necessary since inversion cannot be used with the hardware serial port.

'PIC16F876 used @ 20Mhz, see annotation for changes necessary

```

DEFINE OSC 20           ' 20Mhz Oscillator was used
DEFINE HSER_RCSTA 90h  ' Enable Asynchronous Serial Receive
DEFINE HSER_TXSTA 20h  ' Enable Asynchronous Serial Transmit
DEFINE HSER_BAUD 19200 ' Baud = 19200
DEFINE HSER_SPBRG 15   ' Must be changed depending on Osc. used, see datasheet
DEFINE HSER_CLROERR 1  ' Errors automatically cleared when encountered

```

```

SDA      var PORTC.1  ' Alias "SDA" to pin C.1
SCL      var PORTC.2  ' Alias "SDA" to pin C.1
i2c_read con 1        ' R/W configuration bit (1 = read)
i2c_write con 0       ' R/W configuration bit (0 = write)
i2c_out  var byte     ' data to sent over I2C bus
i2c_in   var byte[2]  ' data received over I2C bus
i2c_ack  var bit      ' acknowledgement bit
temp     var word

```

```

GOSUB Config_Register ' Set Configuration
GOSUB Start_Convert   ' Start continuous conversion

```

TOP:

```

    Pause 5000
    GOSUB Read_Temp    ' Read the current temperature
    i2c_in[2] = i2c_in[1] >> 3 ' Shift 5 Decimal bits to LS position
    temp = (i2c_in[1]*1000) ' PIC Doesn't like Decimals, but there is a way to work around this
    Hserout [Dec i2c_in[0], ".",Dec2 (temp ** 2048)/100,13,10] ' Outputs temperature to terminal
GOTO top              ' Loops forever

```

```

Config_Register:      ' Set continuous conversion
    GOSUB I2C_START   ' Start Condition
    i2c_out = %10010000 ' Send Address, (device= %1001, A0= 0, A1= 0, A2= 0, R/W= 0)
    GOSUB I2C_TX      ' Send data in "i2c_out"

```

```

i2c_out = $AC          ' Send "Access Configuration" command
GOSUB I2C_TX          ' Send data in "i2c_out"
i2c_out = $00         ' Send "Continuous Conversion" command
GOSUB I2C_TX          ' Send data in "i2c_out"
GOSUB I2C_STOP        ' Stop Condition
Return

Start_Convert:        ' Start Conversion
  GOSUB I2C_START
  i2c_out = %10010000 ' Send Address, (device= %1001, A0= 0, A1= 0, A2= 0, R/W= 0)
  GOSUB I2C_TX
  i2c_out = $EE        ' Send "Start to Convert" command
  GOSUB I2C_TX
  GOSUB I2C_STOP
Return

Read_Temp:            ' Read temperature
  GOSUB I2C_START
  i2c_out = %10010000 ' You must "write" the command to read the temperature before
  GOSUB I2C_TX        ' reading it, therefore R/W still is 0
  i2c_out = $AA        ' Send "Read Temperature" command
  GOSUB I2C_TX
  GOSUB I2C_START    ' * Reissue Start Condition *
  i2c_out = %10010001 ' Send Address, (device= %1001, A0= 0, A1= 0, A2= 0, R/W= *1*)
  GOSUB I2C_TX        ' Transmit address with R/W bit as 1 ("read")
  GOSUB I2C_RX        ' Start getting data coming in
  GOSUB I2C_STOP      ' Issue stop condition
Return

I2C_START:            ' I2C start (start communication on I2C bus)
  high SDA
  high SCL
  low SDA
  low SCL
Return

I2C_STOP:              ' I2C stop (terminate communication on I2C bus)
  low SDA
  high SCL
  high SDA
  pause 1
Return

I2C_RX:                ' I2C receive -> receive data from slave
  Shiftin SDA,SCL,0,[i2c_in[0]] ' Shift in first byte MSBfirst
  Shiftout SDA,SCL,1,[%0\1]      ' Send acknowledge (ACK) = 0

```

```

Shiftin SDA,SCL,0,[i2c_in[1]]      ' Shift in second byte MSBfirst
Shiftout SDA,SCL,1,[%1\1]         ' Send not acknowledge (NACK) = 1
Return

I2C_TX:                             ' I2C transmit -> send data to the slave
  Shiftout SDA,SCL,1,[i2c_out]      ' Shift out "i2c_out" LSBfirst
  Shiftin SDA,SCL,0,[i2c_ack\1]    ' Receive ACK bit
Return

```

The only part of this code that gets really tricky is under the TOP: label where the data coming in is converted to a serial transfer. To understand what is happening, one must first know how data from the DS1624 comes in. 13bits are shifted into 2 bytes of an array, but they must first be manipulated to be of any use. Data in "i2c_in[0]" gets received first.

Alone this data could be used as-is for 1°C resolution measurements. What about negative values? Well we know the data can only go as high as +125°C, but yet some values come in as high then this. Take 255 (the maximum value of a byte) and subtract only those numbers > 125, and you will get a bunch of negative values; isn't that convenient! (Ex. 250 > 255 therefore 255 - 250 = -5°C).

This conversion doesn't deal with negative numbers, because they are easy enough to implement and there is enough to deal with already. That takes care of the first byte, but now what about i2c_in[1]?

Well this is a 5 bit number that represents .03125°C Decimal precision, but it is in the 5 Most Significant Bits. In order to work with this number, it must be shifted over 3 places to the right (>> 3). The value is then multiplied by 1000 because it is hard to work with Decimals, and we would prefer to have 00.000 Decimal precision (although with integer math, only 00.00 Decimal precision will be accurately attainable).

The number is then modified with the operation (" ** 2048") which essentially is like multiplying by 2048/65536 = 0.03125 with a no rounding until the end. The value is then divided by 100 and displayed with Dec2 because we want a result of 00.00°C at the end.

For Example:

```

The DS1624 gives a value of $1980 Shifted in
$19 = 25 so we know there is a measured temperature of 25.X°C
$80 = %10000000 and we shift over 3 giving %10000 as the 5 bit Decimal value
%10000 = 16 * 10000 = 16000
          *this will never overflow because (1/0.03125) * 10000 = 32000)*
16000 * (2048/65536) = 5000 in integer math
5000 / 100 = 50
Because of the format we are using to send serial data to the terminal, the end
result is "25", ".", "50" ending up being "25.50" degrees Celsius

```

Any conversion to Fahrenheit must be done using a lookup table or conversion factor, and is not a part of this code

The code itself is quite simple, and easy to understand, but it is not very efficient. First of all, only one byte can be sent out at a time. Take for instance when you want to set the DS1624 for continuous conversion, you must first send the device code, then \$AC then \$00. There are many places where the "device code" is send out with either a 1 or a 0 Read/Write bit, but the way this code stands, each device needs its own subroutine. This code could be improved to allow you to

send and receive any number of bytes, and would let you specify which device code to perform these operations on.

The following code does the same as the subroutines above, only now things are a lot more condensed. Because you may wish to use this in different projects, it has been created as 2 "include" files. The reason that 2 files are need is that some variables must be Declared before subroutines are run, and it is undesirable to put all the subroutines before the main body of the code because the PIC will "flow" into these instructions and the program will not work. Therefore it is easier to just type 2 includes into the code rather than remember the exact variables that need to be added for the subroutines to work the first part is included below as well as a brief explanation.

```
' File: "I2C_A.bas"
' Subroutines for using custom I2C
' Any Questions, jonz@shaw.ca

' To send data, put the number of bytes to send in var "FLAG" (IE: flag = 2) will send 2 bytes
' After Declaring number of bytes, set the data to send into the array I2C_OUT[$??]
' Then run the subroutine "I2CTX"

' To receive data set the FLAG variable again, data will be placed in the string I2C_IN MSBFIRST
' in string position [0] the [1] ect. Run the subroutine "I2CRX" After setting the flag variable

' Include the following into your code that needs I2C Routines:
' SDA    Var PortC.1 ' The port your SDA line is on
' SCL    Var PortC.2 ' The port your SCL line is on
' ADDR   Var Byte    ' The address of the I2C device you wish to access
' I2C_OUT Var Byte[2] '[2] is max bytes sent out, has to be >= max strings being sent out
' I2C_IN  Var Byte[2] '[2] is max bytes received, has to be >= max strings being sent in at once
' Include "I2C_A.bas"

' The Rest of your program goes below. All variables must be declared before "include" statement
' To get around the fact that some variables must be Declared before the subroutines ect, a second
' include file is needed. Add it below your program as follows:

' Include "I2C_B.bas"

' ***** Start of "I2C_A.bas" content ***** '
I2C_ACK Var Bit      ' ACK bit receive
N        Var Byte    ' "FOR" loop Variable
FLAG     Var Byte    ' FLAG Variable for Bytes to send/receive
RW       Var Bit     ' Read/Write bit variable used by subroutines

' ***** End of "I2C_A.bas" content ***** '
```

The Second file of the I2C Routines contains subroutines. "Include "I2C_B.bas" after the main body of your program

' File: "I2C_B.bas"

' Subroutines for using custom I2C

' Any Questions, jonz@shaw.ca

' ***** Start of "I2C_B.bas" content ***** '

```
I2CTX:                                     ' Transmit I2C Routine
  GOSUB I2C_Start                          ' Issue Start Condition
  RW = 0                                    ' "Write" command
  GOSUB Set_ADDR                           ' Send Address from variable ADDR
    For N = 1 to FLAG                       ' Shifts out byte[0]..[1]... for number in FLAG times
      Shiftout SDA, SCL, 1, [I2C_OUT[N-1]\8]
      Shiftin SDA, SCL, 0, [I2C_ACK\1]
    NEXT N
  GOSUB I2C_Stop                            ' Issue Stop Command
Return

I2CRX:                                     ' Receive I2C Routine
  GOSUB I2C_Start                          ' "Read" command
  RW = 1
  GOSUB Set_ADDR
  IF FLAG >= 2 then                         ' If there are more then 1 byte, send all but last then ACK
    For N = 1 to (FLAG - 1)
      Shiftin SDA, SCL, 0, [I2C_IN[N-1]\8]
      Shiftout SDA, SCL, 1, [%0\1]
    NEXT N
  ENDIF
  Shiftin SDA, SCL, 0, [I2C_IN[FLAG-1]\8]   '*** Send last byte with NACK ***
  Shiftout SDA, SCL, 1, [%1\1]
  GOSUB I2C_Stop                            ' Issue Stop command
Return

Set_ADDR:                                  ' Send Address of device from Var ADDR
  Shiftout SDA, SCL, 1, [ADDR >> 1\7,RW\1] ' Shifts ADDR bits over, adds on R/W bit
  Shiftin SDA, SCL, 0, [I2C_ACK\1]         ' Receives ACK
Return

I2C_Start:                                 ' Start Condition
  HIGH SDA
  HIGH SCL
  LOW SDA
  LOW SCL
Return
```

```

I2C_Stop:                                'Stop Condition
    LOW SDA
    HIGH SCL
    HIGH SDA
    Pause 1
RETURN

```

‘ ***** End of “I2C_B.bas” content ***** ‘

To see the result of the new subroutines, the following file can be used and compiled as long at the above 2 include files are in the same folder and named accordingly.

‘ File: DS1624.bas

‘ Uses I2C Subs in I2C_A.bas and I2C_B.bas

```

DEFINE LOADER_USED 1
DEFINE OSC 20
DEFINE HSER_RCSTA 90h
DEFINE HSER_TXSTA 20h
DEFINE HSER_BAUD 19200
DEFINE HSER_SPBRG 15
DEFINE HSER_CLROERR 1

```

```

SDA    Var PortC.1      ' The port your SDA line is on
SCL    Var PortC.2      ' The port your SCL line is on
ADDR   Var Byte         ' The address of the I2C device you wish to access
I2C_OUT Var Byte[2]     ' [2] is maximum bytes being sent out, can change to send more
I2C_IN  Var Byte[2]     ' [2] is maximum bytes being sent in, can chage to receive more
TEMP   Var word

```

```

Include "I2C_A.bas"          ' Include Subroutine variables

```

```

ADDR = %10010000          ' Address' can be set by a variable

```

```

FLAG = 2 : I2C_OUT[0] = $AC : I2C_OUT[1] = $00  ' Send 2 bytes, Address taken care of
GOSUB I2CTX

```

```


```

```

FLAG = 1 : I2C_OUT[0] = $EE          ' Send out 1 byte
GOSUB I2CTX

```

```


```

```

TOP:

```

```

    Pause 500

```

```

    GOSUB Read_Temp

```

```

    I2C_IN[2] = I2C_IN[1] >> 3      ' Shift data so that 5 bits are in the LS position

```

```

    TEMP = (I2C_IN[1]*1000)          ' Takes the Decimal data and prepares it for * by .03125

```

```

    HSEROUT [DEC I2C_IN[0],".",Dec1 (TEMP ** 2048)/100,13,10]

```

```

GOTO TOP

```

Read_Temp:

```
FLAG = 1 : I2C_OUT[0] = $AA      ' 1 Byte out, R/W = 0 because TX subroutine used
GOSUB I2CTX
FLAG = 2                          ' 2 bytes in
GOSUB I2CRX                        ' Receive subroutine, R/W = 1 because RX subroutine used
Return
```

Include "I2C_B.bas"

' Include Subroutines

When you compile the program you'll notice the second version will use more memory. While it may seem at first this is a disadvantage to the first approach, you must take into consideration the first was used to just access a certain address using a specific byte out, byte in pattern. The second approach allows as many bytes in or out as once as the PIC's RAM allows, and the address can be specified by a variable. It is then more desirable to use the subroutine approach if different and/or multiple devices on the same buss were to be used.

Connection diagram for experiment:

