SPI, I²C Serial Interfaces

• The SPI and I²C are two synchronous serial interfaces on the PIC24 µC

• Both are commonly used in industry

• The SPI port requires a minimum of three wires (and usually 4), and is technically duplex, even though most transfers are half-duplex. Its top speed on the PIC24 µC is 10 MHz.
  • Best for high-speed serial transfer
  • Very simple

• The I²C port requires only two wires regardless of the number of peripherals, is half-duplex, and top speed is 1 MHz.
  • Best if you are trying to reduce external pin usage.
Serial Peripheral Interface (SPI)

Data is sent MSb first; received data is clocked in as transmitted data is clocked out. Every transmission is a duplex transmission because data is exchanged on SDOx/SDIx. Device_Selec# must be low before transmission starts to select the Slave and must remain low for the duration of the transfer.

SDO

SDI

SCK

SCK

PIO

PIO

SDOx

SDLx

SCKx

Device_Selec#
SPI Transmission Formats

CKP – clock polarity.
CKE – controls which edge output data is transmitted on.

Which format is used depends on peripheral.

CKP = 0, CKE = 1 seems common.
void checkRxErrorSPI1() {  
if (SPI1STATbits.SPIROV) {  
    // clear the error  
    SPI1STATbits.SPIROV = 0;  
    reportError("SPI1 Receive Overflow\n");  
}  
}  

Only function needed besides configuration

uint16 ioMasterSPI1(uint16 u16_c) {  
    checkRxErrorSPI1();  
    _SPI1IF = 0;  // clear interrupt flag since we are about to  
    // write new value  
    SPI1BUF = u16_c;  
    while (!_SPI1IF) {  // wait for operation to complete  
        doHeartbeat();  
    }  
    return SPI1BUF;  // return the shifted in value

Must ALWAYS read the input buffer or SPI overflow can occur!
To read data from a slave device, the master has to write data in order to get data back!
We use RB3 from the PIC24 µC as the chip select for the DS1722. This chip select is high true.
(a) Configuration byte

<table>
<thead>
<tr>
<th>MSb</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1SHOT</th>
<th>R2</th>
<th>R1</th>
<th>R0</th>
<th>SD</th>
<th>LSb</th>
</tr>
</thead>
</table>

SD: 0- continuous conversion, 1- complete current conversion, enter low power mode.

R2/R1/R0: 000 8-bit mode, 0.075s conversion time, 1.0° C resolution (8.0 signed fixed-point)
001 9-bit mode, 0.15s conversion time, 0.5° C resolution (8.1 signed fixed point)
010 10-bit mode, 0.3s conversion time, 0.25° C resolution (8.2 signed fixed point)
011 11-bit mode, 0.6s conversion time, 0.125° C resolution (8.3 signed fixed point)
1xx 12-bit mode, 1.2s conversion time, 0.0625° C resolution (8.4 signed fixed point)
1SHOT: when SD=1 writing a 1 to this bit starts conversion, is cleared when conversion finished.

(b) Single-byte transfer, write configuration

![Diagram](image)

To read configuration byte, use address 0x00

(c) Multi-byte transfer, read temperature

![Diagram](image)

Read from address 0x01  Read from address 0x02

(d) Temperature data format is 8.4 two’s complement fixed point (integer portion is MSByte, fractional is LSByte).

Celsius (float) = 16-bit temperature (int16) / 256

---

DS1722 Details

Write configuration byte to get the DS1722 started.
We will use continuous conversion mode.
Utility Functions for DS1722

```c
#define CONFIG_SLAVE_ENABLE() CONFIG_RB3_AS_DIG_OUTPUT()
#define SLAVE_ENABLE() _LATB3 = 1  //high true assertion
#define SLAVE_DISABLE() _LATB3 = 0  // RB3 used for the DS1722 chip select.

void configSPI1(void) {
    //spi clock = 40MHz/1*4 = 40MHz/4 = 10MHz  // 10 MHz SPI clock
    SPI1CON1 = SEC_PRESCAL_1_1 |  //1:1 secondary prescale
               PRI_PRESCAL_4_1 |  //4:1 primary prescale
               CLK_POL.ACTTIVE_HIGH |  //clock active high (CKP = 0)
               SPI_CKE_OFF |  //out changes inactive to active (CKE=0)
               SPI_MODE8_ON |  //8-bit mode
               MASTER_ENABLE_ON;  //master mode

    //configure pins. Need SDO, SCK, SDI
    CONFIG_SDO1_TO_RP(6);  //use RP6 for SDO
    CONFIG_SCK1OUT_TO_RP(7);  //use RP7 for SCLK
    CONFIG_SDI1_TO_RP(5);  //use RP5 for SDI
    SLAVE_SLAVE_ENABLE();  //chip select for DS1722
    SLAVE_DISABLE();  //disable the chip select
    SPI1STATbits.SPIEN = 1;  //enable SPI mode
}
```

Macros for SPI configuration are defined in `pic24_spi.h`
Utility Functions for DS1722 (cont.)

```c
tvoid writeConfigDS1722(uint8 u8_i) {
    SLAVE_ENABLE();          //assert chipselect
    ioMasterSPI1(0x80);       //config address
    ioMasterSPI1(u8_i);       //config value
    SLAVE_DISABLE();
}  // Writes to the DS1722 configuration register.

int16 readTempDS1722() {
    uint16 u16_lo, u16_hi;
    SLAVE_ENABLE();          //assert chipselect
    ioMasterSPI1(0x01);       //LSB address
    u16_lo = ioMasterSPI1(0x00); //read LByte
    u16_hi = ioMasterSPI1(0x00); //read MSByte
    SLAVE_DISABLE();
    return((u16_hi<<8) | u16_lo);
}  // Reads 16-bit temperature value from DS1722.
```

Send dummy data to get data back.

Upper/lower bytes of temperature returned as single 16-bit value.
int main (void) {
    (a) main() function.
    int16 i16_temp;
    float f_tempC, f_tempF;
    configBasic(HELLO_MSG);
    configSPI1();
    writeConfigDS1722(0xE8); //12-bit mode
    while (1) {
        DELAY_MS(1500);
        i16_temp = readTempDS1722();
        f_tempC = i16_temp; //convert to floating point
        f_tempC = f_tempC/256; //divide by precision
        f_tempF = f_tempC*9/5 + 32;
        printf("Temp is: 0x%0X, %4.4f (C), %4.4f (F)\n", i16_temp,
               (double) f_tempC, (double) f_tempF);
    }
}

(b) Sample Output

ds1722_spi_tempsense.c, built on Jun 27 2008 at 21:56:03
Temp is: 0x1BC0, 27.7500 (C), 81.9500 (F)
Temp is: 0x1BD0, 27.8125 (C), 82.0625 (F)
Temp is: 0x1BD0, 27.8125 (C), 82.0625 (F)
Temp is: 0x1C30, 28.1875 (C), 82.7375 (F)
Temp is: 0x1D70, 29.4375 (C), 84.9875 (F)
Temp is: 0x1DC0, 29.7500 (C), 85.5500 (F)
Temp is: 0x1E10, 30.0625 (C), 86.1125 (F)
Temp is: 0x1E30, 30.1875 (C), 86.3375 (F)
Temp is: 0x1D90, 29.5625 (C), 85.2125 (F)
Temp is: 0x1D30, 29.1875 (C), 84.5375 (F)
Temp is: 0x1CFO, 28.9375 (C), 84.0875 (F)

Testing the DS1722

Finger placed on sensor to raise temperature.

Finger removed from sensor.

Copyright Delmar Cengage Learning 2008. All Rights Reserved.
From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.
Bus Definition

When a device on a bus talks, all hear what is said.

An address is used to specify what device the communication is intended for.
Inter-Integrated Circuit (I²C) Bus

Recommended value for a typical PIC24 system.

Encoded within device, device specific

“0” Master to Slave (write)
“1” Slave to Master (read)

I²C Peripheral
(address = 0b \( n_3n_2n_1n_0 \) A2 A1 A0 R/W#)

External connections determine address

SCLx: Clock
SDAx: Data
Both SCL, SDA are bidirectional

PIC24 µC

Copyright Delmar Cengage Learning 2008. All Rights Reserved.
From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

V 0.2 13
I²C Bus Signaling

Every byte transferred takes 9 bits because of acknowledgement bit.
# I²C on the PIC24 µC

## (a) I²C Module Registers

<table>
<thead>
<tr>
<th>I²C Registers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2CxCON</td>
<td>Control Register</td>
</tr>
<tr>
<td>I2CxSTAT</td>
<td>Status Register</td>
</tr>
<tr>
<td>I2CxBRG</td>
<td>Baud Rate Register</td>
</tr>
<tr>
<td>I2CxTRN</td>
<td>Transmit Register</td>
</tr>
<tr>
<td>I2CxRCV</td>
<td>Receive Register</td>
</tr>
<tr>
<td>I2CxMSK</td>
<td>Slave Mode Address Mask Register</td>
</tr>
<tr>
<td>I2CxADD</td>
<td>Slave Mode Address Register</td>
</tr>
</tbody>
</table>

## (b) Commonly used I²C Control and Status Bits

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEN</td>
<td>I2CxCON&lt;0&gt;</td>
<td>Set to begin Start sequence, cleared by HW</td>
</tr>
<tr>
<td>RSEN</td>
<td>I2CxCON&lt;1&gt;</td>
<td>Set to begin Repeated Start sequence, cleared by HW</td>
</tr>
<tr>
<td>PEN</td>
<td>I2CxCON&lt;2&gt;</td>
<td>Set to begin Stop condition, cleared by HW</td>
</tr>
<tr>
<td>RCEN</td>
<td>I2CxCON&lt;3&gt;</td>
<td>Set to enable receive, cleared in HW</td>
</tr>
<tr>
<td>ACKEN</td>
<td>I2CxCON&lt;4&gt;</td>
<td>Set to enable acknowledge sequence, cleared by HW</td>
</tr>
<tr>
<td>ACKDT</td>
<td>I2CxCON&lt;5&gt;</td>
<td>ACK bit to send; 1 for NAK, 0 for ACK.</td>
</tr>
<tr>
<td>I2CEN</td>
<td>I2CxCON&lt;15&gt;</td>
<td>Enable the I2Cx module</td>
</tr>
<tr>
<td>RBF</td>
<td>I2CxSTAT&lt;1&gt;</td>
<td>Set when I2CxRCV register is full, cleared by HW after read of I2CxRCV</td>
</tr>
<tr>
<td>S12Cx1F</td>
<td>Interrupt Flag Status Registers</td>
<td>Interrupt flag set on detection of address reception in Slave mode, reception of data, or request to transmit data</td>
</tr>
</tbody>
</table>
Support Functions – I²C Operations

(a) Support Functions for I²C Operations

<table>
<thead>
<tr>
<th>I²C Support Functions (Operations)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void configI2C1(uint16 u16_FkHZ)</td>
<td>Enables the I²C module for operation at u16_FkHZ kHz clock rate</td>
</tr>
<tr>
<td>void startI2C1(void)</td>
<td>Performs start operation</td>
</tr>
<tr>
<td>void rstartI2C1(void)</td>
<td>Performs repeated start operation</td>
</tr>
<tr>
<td>void stopI2C1(void)</td>
<td>Performs stop operation</td>
</tr>
<tr>
<td>void putI2C1(uint8 u8_val)</td>
<td>Transmits u8_val; software reset if NAK returned.</td>
</tr>
<tr>
<td>uint8 putNoAckCheckI2C1(uint8 u8_val)</td>
<td>Transmits u8_val and returns received acknowledge bit</td>
</tr>
<tr>
<td>uint8 getI2C1(uint8 u8_ack2Send)</td>
<td>Receive one byte and send u8_ack2Send as acknowledge bit</td>
</tr>
</tbody>
</table>

These are primitive operations.
Support Functions – $I^2C$ Transactions

(b) Support Functions for $I^2C$ Transactions

<table>
<thead>
<tr>
<th>$I^2C$ Support Functions (Transactions)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void write1I2C1(uint8 u8_addr,uint8 u8_d1)</td>
<td>Write 1 byte (u8_d1)</td>
</tr>
<tr>
<td>void write2I2C1(uint8 u8_addr,uint8 u8_d1, uint8 u8_d2)</td>
<td>Write 2 bytes (u8_d1)</td>
</tr>
<tr>
<td>void writeNI2C1(uint8 u8_addr,uint8* pu8_data, uint16 u16_cnt)</td>
<td>Write u16_cnt bytes in buffer pu8_data</td>
</tr>
<tr>
<td>void read1I2C1 (uint8 u8_addr,uint8* pu8_d1)</td>
<td>Read 1 byte; return in *pu8_d1</td>
</tr>
<tr>
<td>void read2I2C1 (uint8 u8_addr,uint8* pu8_d1, uint8* pu8_d2)</td>
<td>Read 2 bytes; return in *pu8_d1, *pu8_d2</td>
</tr>
<tr>
<td>void readNI2C1 (uint8 u8_addr,uint8* pu8_data, uint16 u16_cnt)</td>
<td>Read u16_cnt bytes; return in *pu8_data</td>
</tr>
</tbody>
</table>

These are use the primitive operations to read/write 1 or more bytes to a slave.
I²C Read/Write Transactions

(a) Write two bytes to slave:
write2I2C1(uint8 u8_addr, uint8 u8_d1, uint8 u8_d2)
putI2C1(u8_addr & 0xFE);

putI2C1(u8_d1);
putI2C1(u8_d2);

startI2C1();

ACK sent by slave,
read by putI2C1();

Data sent by
PIC24 µC

stopI2C1();

R/W#=0
(write)

S u8_addr: 0
A u8_d1
A u8_d2
A P

(b) Read two bytes from slave:
read2I2C1(uint8 u8_addr, uint8* pu8_d1, uint8* pu8_d2)
putI2C1(u8_addr | 0x01);

*pu8_d1 = getI2C1(I²C_ACK);
0, sent by
PIC24 µC

*pu8_d2 = getI2C1(I²C_NAK);

NAK must
be sent for
last byte

S u8_addr: 1
A *pu8_d1
A *pu8_d2
N P

R/W#=1
(read)

startI2C1();

ACK sent by slave,
read by putI2C1();

Data returned by
slave

stopI2C1();
Example primitive function

void configI2C1(uint16 u16_FkHZ) {
    uint32 u32_temp;
    u32_temp = (FCY/1000L)/((uint32) u16_FkHZ);
    u32_temp = u32_temp - FCY/10000000L - 1;
    I2C1BRG = u32_temp;
    I2C1CONbits.I2CEN = 1; // Enable I2C module
}

void startI2C1(void) {
    // Functions stopI2C1(), rstartI2C1() are similar but use bits PEN, RSEN respectively.
    sz_lastTimeoutError = "I2C Start";
    u8_wdtState = _SWDTEN; //save WDT state
    _SWDTEN = 1; //enable WDT
    I2C1CONbits.SEN = 1; // initiate start
    // wait until start finished
    while (I2C1CONbits.SEN);
    _SWDTEN = u8_wdtState; //restore WDT
    sz_lastTimeoutError = NULL;
}
Transactions

(a) Write Transactions

```c
#define I2C_WADD(x) (x & 0xFE)

void write1I2C1(uint8 u8_addr, uint8 u8_d1){
    startI2C1();
    putI2C1(I2C_WADD(u8_addr));
    putI2C1(u8_d1);
    stopI2C1();
}

void write2I2C1(uint8 u8_addr, uint8 u8_d1, uint8 u8_d2){
    startI2C1();
    putI2C1(I2C_WADD(u8_addr));
    putI2C1(u8_d1);
    putI2C1(u8_d2);
    stopI2C1();
}

void writeNI2C1(uint8 u8_addr, uint8* pu8_data, uint16 u16_cnt){
    uint16 u16_i;
    startI2C1();
    putI2C1(I2C_WADD(u8_addr));
    for (u16_i=0; u16_i < u16_cnt;){
        putI2C1(*pu8_data);
        pu8_data++; u16_i++;
    }
    stopI2C1();
}
```

(b) Read Transactions

```c
#define I2C_RADD(x) (x | 0x01)

void read1I2C1 (uint8 u8_addr, uint8* pu8_d1) {
    startI2C1();
    putI2C1(I2C_RADD(u8_addr));
    *pu8_d1 = getI2C1(I2C_NAK);
    stopI2C1();
}

void read2I2C1 (uint8 u8_addr, uint8* pu8_d1, uint8* pu8_d2) {
    startI2C1();
    putI2C1(I2C_RADD(u8_addr));
    *pu8_d1 = getI2C1(I2C_ACK);
    *pu8_d2 = getI2C1(I2C_NAK);
    stopI2C1();
}

void readNI2C1 (uint8 u8_addr, uint8* pu8_data, uint16 u16_cnt) {
    uint16 u16_i;
    startI2C1();
    putI2C1(I2C_RADD(u8_addr));
    for (u16_i=0; u16_i < u16_cnt;){
        if (u16_i != u16_cnt-1)
            *pu8_data = getI2C1(I2C_ACK);
        else *pu8_data = getI2C1(I2C_NAK);
        pu8_data++; u16_i++;
    }
    stopI2C1();
}
```
PIC24 μC Master to DS1631 Thermometer

Similar to the DS1722 but does not have as many precision options. Also has a thermostat function – Two internal registers named TH, TL used for that. When temp > TH, the TOUT output goes high. When Temp falls below TL, the TOUT output goes back low.

TH, TL are stored in non-volatile memory.
DS1631 Details

Address byte format for DS1631 Temperature Sensor

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A2</td>
<td>A1</td>
<td>A0</td>
<td>R/W#</td>
</tr>
</tbody>
</table>

9 (8.1), 10 (8.2), 11 (8.3), 12-bit (8.4) temperature in Celsius, fixed-point two’s complement format.

\[-27.3125 \, ^\circ\mathrm{C} = 0x\text{E4B0}\]

(12-bit mode)

0xE4B0 = -6992 = -6992/256 = -27.3125 °C

(a) Standalone command

R/W# = 0, Write operation

Command byte \downarrow

CMD = 0x51 to start continuous conversion.
CMD = 0x22 to stop continuous conversion.

Start condition \rightarrow ACK

Stop condition

(b) 8-bit Write

R/W# = 0, Write operation

Command byte ↓ Data byte ↓

CMD = 0xAC accesses configuration register.

Used to configure the device
DS1631 Details

(c) 16-bit Write

R/W# = 0, Used to set TH, TL regs.
Write operation → Command byte ↓ Data byte ↓ Data byte ↓

\[ \text{S} \begin{array}{c} \text{I}^{2} \text{C} \text{ addr} \end{array} \begin{array}{c} 0 \end{array} \begin{array}{c} \text{A} \end{array} \begin{array}{c} \text{CMD} \end{array} \begin{array}{c} \text{A} \end{array} \begin{array}{c} \text{MSByte} \end{array} \begin{array}{c} \text{A} \end{array} \begin{array}{c} \text{LSByte} \end{array} \begin{array}{c} \text{AP} \end{array} \]

Used to read the config.

(d) 8-bit Read

CMD = 0xA1 Temp. High Trigger.
CMD = 0xA2 Temp. Low Trigger.

R/W# = 0, Used to read the 9-bit temperature value.
Write operation → Command byte ↓ Data byte from

\[ \text{Repeate} \begin{array}{c} \text{d Start condition} \end{array} \begin{array}{c} \text{S} \begin{array}{c} \text{I}^{2} \text{C} \text{ addr} \end{array} \begin{array}{c} 0 \end{array} \begin{array}{c} \text{A} \end{array} \begin{array}{c} \text{CMD} \end{array} \begin{array}{c} \text{A} \end{array} \begin{array}{c} \text{R} \text{I}^{2} \text{C} \text{ addr} \end{array} \begin{array}{c} 1 \end{array} \begin{array}{c} \text{A} \end{array} \begin{array}{c} \text{Data} \end{array} \begin{array}{c} \text{NP} \end{array} \]

\[ \text{DS}1631 \begin{array}{c} \text{used to read the config.} \end{array} \]

(e) 16-bit Read

CMD = 0xAA accesses 16-bit temperature register.

R/W# = 0, Used to read the 9-bit temperature value.
Write operation → Command byte ↓ Data byte from Data byte from

\[ \text{V} \begin{array}{c} 0.2 \end{array} \]

\[ \text{DS}1631 \begin{array}{c} \text{used to read the 9-bit temperature value.} \end{array} \]
### DS1631 Configuration Register

<table>
<thead>
<tr>
<th>CONFIG Register</th>
<th>DONE</th>
<th>THF</th>
<th>TLF</th>
<th>NVB</th>
<th>1</th>
<th>0</th>
<th>POL</th>
<th>1SHOT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DONE</td>
<td>Conversion Done Flag: &quot;1&quot; when conversion is complete, &quot;0&quot; when conversion is in progress.</td>
</tr>
<tr>
<td>THF</td>
<td>Temperature High Flag: &quot;1&quot; if temperature has exceeded the TH register value since power-on; reset on power down, write to CONFIG register, or software power-on-reset command.</td>
</tr>
<tr>
<td>TLF</td>
<td>Temperature Low Flag: &quot;1&quot; if temperature has dropped below the TL register value since power-on; reset on power down or write to CONFIG register, or software power-on-reset command.</td>
</tr>
<tr>
<td>NVB</td>
<td>Nonvolatile Memory Busy flag: &quot;1&quot; when write to nonvolatile memory is in progress, &quot;0&quot; otherwise.</td>
</tr>
<tr>
<td>R1:R0</td>
<td>Resolution selection bits, 00: 9-bit (93.75 ms), 10-bit (187.5 ms), 11-bit (375 ms), 12-bit (750 ms)</td>
</tr>
<tr>
<td>POL</td>
<td>Polarity bit: &quot;1&quot; TOUT is active high, &quot;0&quot; TOUT is active low. Stored in NVM.</td>
</tr>
<tr>
<td>1SHOT</td>
<td>One-Shot Mode: &quot;1&quot; DS1631 only performs conversions upon receiving a Start Conversion command; &quot;0&quot; the DS1621 performs continuous conversions. Store in NVM.</td>
</tr>
</tbody>
</table>

After configuring for continuous conversion, must send the **Start** command (0xEE) to start conversions.
Support Functions

```c
#define DS1631ADDR 0x90  // DS1631 address with all pins tied low
#define ACCESS_CONFIG 0xAC
#define START_CONVERT 0x51
#define READ_TEMP 0xAA

void writeConfigDS1631(uint8 u8_i) {
    write1I2C1(DS1631ADDR, ACCESS_CONFIG, u8_i);
}

void startConversionDS1631() {
    write1I2C1(DS1631ADDR, START_CONVERT);
}  // Implements standalone command.

int16 readTempDS1631() {
    uint8 u8_lo, u8_hi;
    int16 i16_temp;
    write1I2C1(DS1631ADDR, READ_TEMP);
    read2I2C1(DS1631ADDR, &u8_hi, &u8_lo);
    i16_temp = u8_hi;
    return ((i16_temp<<8)|u8_lo);
}  // Implements 16-bit read command.
```

These use the ‘transaction’ functions to communicate with the DS1621
Testing the DS1631

```c
int main (void) {
    int16 i16_temp;
    float f_tempC, f_tempF;
    configBasic(HELLO_MSC);
    configI2C1(400);  // configure I2C for 400 kHz
    writeConfigDS1631(0x0C);  // continuous conversion, 12-bit mode
    startConversionDS1631();  // start conversions
    while (1) {
        DELAY_MS(750);
        i16_temp = readTempDS1631();
        f_tempC = i16_temp;  // convert to floating point
        f_tempC = f_tempC/256;  // divide by precision
        f_tempF = f_tempC*9/5 + 32;
        printf("Temp is: 0x\%0X, \%4.4f (C), \%4.4f (F)\n",
               i16_temp, (double) f_tempC, (double) f_tempF);
    }
}
```

While(1){} loop is basically the same as used for DS1722.
I²C Bus Activity when Reading DS1631 Temp

```c
write1I2C1(DS1621ADDR, READ_TEMP)
```

```
read2I2C1(DS1621ADDR, &u8_hi, &u8_lo, I2C_NAK)
```

Read Temperature (write command byte)

Read Temperature (read MSB, LSB)

Temperature = 0x1D30/256 = 7472/256 = 29.1875 °C
PIC24 μC Master to 24LC515 Serial EEPROM

24LC515 Serial EEPROM

VDD A2
SCL WP
SDA A1
VSS A0

3.3 V
2.2 kΩ

A2 must be high for device to function
Write protect disabled if low or left open (internal pulldown)
Connect A1, A0 to combination of VDD or VSS to personalize address

EEPROM is 64 Ki x 8, internally arranged as two separate 32 Ki x 8 memories.

NOTE: The diagram above is a logical layout, not the physical pinout, shown on the right.
24LC515 I²C Address Format, Write Operation

(a) Address byte format
for 24LC515 serial EEPROM

B : Memory block select, if “0” then operation is to low memory block (0x0000-0x7FFF), if “1” then operation is to high memory block (0x8000-0x7FFF)

A1, A0: Used to personalized address, up to four LC515 EEPROMs can be on bus.

R/W#: “1” if read operation, “0” if write operation

(b) Write Operation

Memory address
\[ 7 \quad 6 \quad 5 \quad 4 \quad 3 \quad 2 \quad 1 \quad 0 \]

MSb of address high byte is a don’t care as ‘B’ bit of I²C address is used for this.

(c) End-of-Write Polling

Ack bit returns as “1” (NAK) if write is in progress
Ack bit returns as “0” (ACK) when write is not in progress

Most efficient to write 64 bytes at a time as write takes 3 – 5 ms to complete.

Poll device to see when the write is finished.
24LC515 I²C Read Operation

(a) Sequential Read

Read data ↓

ACK continues read, next byte output by 24LC515.

NAK ends read, 24LC515 halts data output.

R/W# = 1, Read operation

Any number of bytes

(b) Random Read

Memory address high byte

Repeated Start Condition

Read data

Read data

Use write operation to set internal address counter. Repeated Start condition begins new transaction.

Any number of bytes

We will do this to read the device, and always will read 64 bytes at a time.
Support Functions

#define EEPROM 0xA0  //LC515 address assuming both address pins tied low.
#define BLKSIZE 64   24LC151 address = 0b 1010 B A1 A0 R/W

0xA0 = 0b 1010 0 0 0 0

void waitForWriteCompletion(uint8 u8_i2cAddr){
    uint8 u8_ack, u8_savedSWDTEN;
    u8_savedSWDTEN = _SWDTEN;  // Enable WDT to escape infinite loop, assumes
    _SWDTEN = 1;                // WDT timeout is greater than EEPROM write time.
    u8_i2cAddr = I2C_WADDR(u8_i2cAddr); //write transaction, so R/W# = 0;
    do {
        startI2C1();
        u8_ack = putNoAckCheckI2C1(u8_i2cAddr);  // Send I2C address with R/W=0,
        stopI2C1();
    } while (u8_ack == I2C_NAK);    // Keep looping until get an ACK back.
    _SWDTEN = u8_savedSWDTEN;     //restore WDT to original state
}

Write 64 bytes in *pu_buf to EEPROM starting at address u16_MemAddr

void memWriteLC515(uint8 u8_i2cAddr, uint16 u16_MemAddr, uint8 *pu8_buf){
    uint8 u8_AddrLo, u8_AddrHi;
    u8_AddrLo = u16_MemAddr & 0x00FF;   // Get the high, low bytes of the memory
    u8_AddrHi = (u16_MemAddr >> 8);     // address.
    pu8_buf[0] = u8_AddrHi;             // First two bytes of pu8_buf are reserved for the
    pu8_buf[1] = u8_AddrLo;             // EEPROM address.
    if (u16_MemAddr & 0x8000) {
        // if MSB set , set block select bit
        u8_i2cAddr = u8_i2cAddr | 0x08;    // Set the “B” bit of the I2C memory
    }
    waitForWriteCompletion(u8_i2cAddr);  // Wait for last write to finish.
    writeNI2C1(u8_i2cAddr,pu8_buf,BLKSIZE+2);  // I2C block write transaction.

Poll device to see when the write is finished.

Write 64 bytes in *pu_buf to device
Support Functions

Read 64 bytes into \*pu8_buf from EEPROM starting at address u16_MemAddr

```c
void memReadLC515(uint8 u8_i2cAddr, uint16 u16_MemAddr, uint8 *pu8_buf){
    uint8 u8_AddrLo, u8_AddrHi;
    u8_AddrLo = u16_MemAddr & 0x00FF;
    u8_AddrHi = (u16_MemAddr >> 8);
    if (u16_MemAddr & 0x8000) {
        // if MSB set, set block select bit
        u8_i2cAddr = u8_i2cAddr | 0x08;  // Set the “B” bit of the I2C memory address if reading upper 32 Ki block.
    }
    waitForWriteCompletion(u8_i2cAddr);  // Wait for last write to finish.
    // set address counter
    write2I2C1(u8_i2cAddr, u8_AddrHi, u8_AddrLo);  // Set EEPROM’s internal address counter.
    // read data
    readNI2C1(u8_i2cAddr, pu8_buf, BLKSIZE);  // I2C block read transaction.
}
```

Read 64 bytes from device, return in \*pu_buf
int main (void) {
    uint8 au8_buf[64+2]; // 2 extra bytes for address
    uint16 u16_MemAddr;
    uint8 u8_Mode;

    configBasic(HELLO_MSG);
    configI2C1(400); // configure I2C for 400 KHz
    outString("
Enter 'w' for write mode, anything else reads: ");
    u8_Mode = inCharEcho();
    outString("\n");
    u16_MemAddr = 0; // start at location 0 in memory
    while (1) {
        uint8 u8_i;
        if (u8_Mode == 'w') {
            outString("Enter 64 chars.\n");
            // first two buffer locations reserved for starting address
            for (u8_i = 2; u8_i < 64+2; u8_i++) {
                au8_buf[u8_i] = inCharEcho();
            } // Get 64 bytes from the console.
            outString("\nDoing Write\n");
            // write same string twice to check Write Busy polling
            memWriteLC515(EEPROM,u16_MemAddr, au8_buf); // do write
            u16_MemAddr = u16_MemAddr + 64;
            memWriteLC515(EEPROM,u16_MemAddr, au8_buf); // do write
            u16_MemAddr = u16_MemAddr + 64;
        } else {
            memReadLC515(EEPROM, u16_MemAddr, au8_buf); // do read
            for (u8_i = 0; u8_i < 64; u8_i++) outChar(au8_buf[u8_i]);
            outString("\nAny key continues read...\n");
            inChar();
            u16_MemAddr = u16_MemAddr + 64;
        }
    }
}

Testing the 24LC515

In write mode, read 64 characters from the console, write to the 24LC515

In read mode, read 64 characters from the memory, write to the console.

Echo 64 bytes to the console.
Reset cause: Power-on.
Device ID = 0x00000F1D (PIC24HJ32GP202), revision 0x00003002 (A3)
Fast RC Osc with PLL

i2c_serialeepromtest.c, built on Jun 28 2008 at 19:21:32

Enter 'w' for write mode, anything else reads: w
Enter 64 chars.
A person who graduates today and stops learning tomorrow is
Doing Write
Enter 64 chars.
uneducated the day after. Life long learning is very important.
Doing Write
Enter 64 chars.

Reset cause: MCLR assertion.
Device ID = 0x00000F1D (PIC24HJ32GP202), revision 0x00003002 (A3)
Fast RC Osc with PLL

i2c_serialeepromtest.c, built on Jun 28 2008 at 19:21:32

Enter 'w' for write mode, anything else reads: r
A person who graduates today and stops learning tomorrow is
Any key continues read...
A person who graduates today and stops learning tomorrow is
Any key continues read...
uneducated the day after. Life long learning is very important.
Any key continues read...
uneducated the day after. Life long learning is very important.
Any key continues read...

Strings read back from EEPROM.

Two strings entered; each string saved twice to EEPROM.

Reset button pressed.
I²C Lab Assignment

Read streaming data from serial port, write to EEPROM

Serial Input data → RX → TX → I²C Bus → Serial EEPROM

Data is arriving in a steady stream

Capture 64 bytes from serial port, save in buffer

Do page write to serial EEPROM

Interrupt service routine stores arriving bytes in buffer.

Problem: While writing bytes to serial EEPROM, more bytes are arriving! Where are they placed?

Solution: Use two buffers! Second buffer captures data while first buffer data is written to EEPROM.
Double buffering for Streaming Data

Interrupt Service Routine (background code) captures incoming data.

Normal program flow (foreground code) does EEPROM writes.

Page write to EEPROM

Empty au8_buffer0 while au8_buffer1 is filling, swap when au8_buffer1 is full.

Incoming Data

From serial port
UART Receive ISR

UART RX
Interrupt Service Routine

Character arrival via serial port triggers interrupt

0

\texttt{u8\_activeBuffer?}

1

\texttt{Save character in au8\_buffer0}

\texttt{Save character in au8\_buffer1}

64 Bytes?

\texttt{64 Bytes?}

\texttt{Set semaphore to inform main() that buffer is full, switch to empty buffer.}

\texttt{u8\_writeFlag = 1}

\texttt{u8\_activeBuffer = 1}

\texttt{u8\_writeFlag = 1}

\texttt{u8\_activeBuffer = 0}

Exit ISR

This is a flowchart of the UART Receive ISR you must write. This is NOT a software FIFO, you are just placing input data in one of two separate buffers.
This is a flowchart of the `while(1){}` loop in `main()` that you must write. The ISR places data into the buffer, and then the `while(1){}` loop writes it to EEPROM.
What do you have to know?

• SPI port operation
• SPI slave to PIC24 master communication, hookup
• DS1722 operation
• I2C Bus operation
• I2C primitive function operation and usage
• I2C Transaction function operation and usage
• DS1621 operation
• 24LC515 EEPROM Operation