Microcomputers

Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I2C) Communications

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
  • We will only use one simple mode of operation of the SPI interface
• 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 Select# must be low before transmission starts to select the Slave and must remain low for the duration of the transfer.

SPIx Block Diagram for PIC24 µC
SPI Transmission Formats

Which format is used depends on peripheral.

CKP = 0, CKE = 1 is common.

SPI Transmission Formats (cont.)

- CKP (SPIxCON1<6>) selects the SPI clock polarity which determines if the clock will idle high or low
  - 1 = Idle state for clock is a high level; active state is a low level
  - 0 = Idle state for clock is a low level; active state is a high level

- CKE (SPIxCON1<8>) – Clock Edge Select
  - Controls when data is transmitted relative to the clock (SCK)
    - 1 = Serial output data changes on transition from active clock state to Idle clock state
    - 0 = Serial output data changes on transition from Idle clock state to active clock state

- CKP=0, CKE=1
  - Idle low clock, data transmission on active-to-idle clock transition
  - i.e. Data transmission on falling clock edge for CKP=0, CKE=1 combination

- SMP (SPIxCON1<9>) : SPIx Data Input Sample Phase bit
  - Master mode:
    - 1 = Input data sampled at end of data output time
    - 0 = Input data sampled at middle of data output time
**SPI C Functions**

```c
void checkRxErrorSPI1() {
    if (SPI1STATbits.SPIROV) {
        // clear the error
        SPI1STATbits.SPIROV = 0;
        reportError("SPI1 Receive Overflow\n");
    }
}
```

`checkRxErrorSPI1();` is the only function needed besides configuration. It clears the error flag and reports an error message.

```c
uint16_t ioMasterSPI1(uint16_t 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);
}
```

Must ALWAYS read the input buffer or SPI overflow can occur!

---

**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.
I²C Serial Interfaces

- I²C is a synchronous serial interface on the PIC24 µC
- Commonly used in industry
- 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.

Inter-Integrated Circuit (I²C) Bus

- Recommended value for a typical PIC24 system:
  - 2.2 kΩ
- Encoded within device, device specific:
  - FC Peripheral (address = 0b n3n2n1n0 A2 A1 A0 R/W#)
- “0” Master to Slave (write)
- “1” Slave to Master (read)
- External connections determine address
- SCLx: Clock
- SDAx: Data
- Both SCL, SDA are bidirectional

- I²C Peripheral:
  - SCL
  - SDA
  - A2
  - A1
  - A0

- I²C Peripheral:
  - SCL
  - SDA
  - A2
  - A1
  - A0
Every byte transferred takes 9 bits because of acknowledgement bit.

I\(^2\)C Registers on the PIC24 µC

- I\(^2\)CxCON and I\(^2\)CxSTAT are control and status registers, respectively.
- I\(^2\)CxRCV is the receive buffer and the register to which data bytes are written, or from which data bytes are read
- I\(^2\)CxTRN is the transmit register to which bytes are written during a transmit operation
- The I\(^2\)CxADD register holds the slave address
- The I\(^2\)CxBRG acts as the Baud Rate Generator (BRG)
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 by 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>Acknowledge Data bit: 1 for NAK, 0 for ACK</td>
</tr>
<tr>
<td>I²CEN</td>
<td>I²CxCON&lt;15&gt;</td>
<td>Enables the I²C module</td>
</tr>
<tr>
<td>RBF</td>
<td>I²CxSTAT&lt;1&gt;</td>
<td>Receive Buffer Full Status bit: 1 = Receive complete, 0 = Receive not complete</td>
</tr>
<tr>
<td>SI2CxIF</td>
<td>Interrupt Flag Status Registers</td>
<td>Interrupt flag set on detection of a valid device address 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_FkH2)</td>
<td>Enables the I²C module for operation at u16_FkH2 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 and will be used to implement higher level functions.
### Support Functions – I²C Transactions

#### I²C Support Functions (Transactions)

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

These are the primitive operations to read/write 1 or more bytes to a slave.

### I²C Read/Write Transactions

#### (a) Write two bytes to slave:

```c
void write2I2C1(uint8 u8_addr, uint8 u8_d1, uint8 u8_d2) {
    putI2C1(u8_addr);  // S u8_addr : 0 A u8_d1 A u8_d2 A P
    putI2C1(u8_d1);  // S u8_addr : 0 A u8_d1 A u8_d2 A P
    putI2C1(u8_d2);  // S u8_addr : 0 A u8_d1 A u8_d2 A P
    startI2C1();     // ACK sent by slave, read by putI2C1()
    stopI2C1();      // Data sent by PIC24 µC
}
```

#### (b) Read two bytes from slave:

```c
void read2I2C1(uint8 u8_addr, uint8* pu8_d1, uint8* pu8_d2) {
    putI2C1(u8_addr);  // S u8_addr : 1 A *pu8_d1 A *pu8_d2 A N P
    *pu8_d1 = getI2C1(I2C_ACK); // 0, sent by PIC24 µC
    *pu8_d2 = getI2C1(I2C_NAK); // NAK must be sent for last byte
    startI2C1();       // ACK sent by slave, read by putI2C1()
    stopI2C1();        // Data returned by slave
```
Example primitive functions

```c
void configI2C1(uint16_t u16_FkHZ) {
    uint32_t u32_temp;

    u32_temp = (FCY/1000L)/u16_FkHZ;
    u32_temp = u32_temp - FCY/10000000L - 1;
    I2C1BRG = u32_temp;
    I2C1CONbits.I2CEN = 1;
}
```

Compute $I2C1BRG = \frac{Fcy}{FSCL} - \frac{Fcy}{10,000,000} - 1$

Example primitive functions

```c
void startI2C1(void) {
    uint8_t u8_wdtState;

    sz_lastTimeoutError = "I2C1 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;
}
```

Functions stopI2C1() and rstartI2C1() are similar, but use bits PEN and RSEN respectively.
### I2C Write Transactions

```c
#define I2C_WADDR(x) (x & 0xFE) //clear R/W bit of I2C address

void write1I2C1(uint8_t u8_addr, uint8_t u8_d1) {
    startI2C1();
    putI2C1(I2C_WADDR(u8_addr));
    putI2C1(u8_d1);
    stopI2C1();
}

void write2I2C1(uint8_t u8_addr, uint8_t u8_d1, uint8_t u8_d2) {
    startI2C1();
    putI2C1(I2C_WADDR(u8_addr));
    putI2C1(u8_d1);
    putI2C1(u8_d2);
    stopI2C1();
}
```

*LSB must be 0 for write*

```c
void writeNI2C1(uint8_t u8_addr, uint8_t* pu8_data, uint16_t u16_cnt) {
    uint16_t u16_i;
    startI2C1();
    putI2C1(I2C_WADDR(u8_addr));
    for (u16_i=0; u16_i < u16_cnt; u16_i++) {
        putI2C1(*pu8_data);
        pu8_data++;
    }
    stopI2C1();
}
```

*LSB must be 0 for write*

### I2C Write Transactions (cont.)

```c
#define I2C_WADDR(x) (x & 0xFE) //clear R/W bit of I2C address

void writeNI2C1(uint8_t u8_addr, uint8_t* pu8_data, uint16_t u16_cnt) {
    uint16_t u16_i;
    startI2C1();
    putI2C1(I2C_WADDR(u8_addr));
    for (u16_i=0; u16_i < u16_cnt; u16_i++) {
        putI2C1(*pu8_data);
        pu8_data++;
    }
    stopI2C1();
}
```

*LSB must be 0 for write*

*Used for block data transfers.*
### I2C Read Transactions

```c
#define I2C_RADDR(x) (x | 0x01) // set R/W bit of I2C address

void read1I2C1(uint8_t u8_addr, uint8_t* pu8_d1) {
    startI2C1();
    putI2C1(I2C_RADDR(u8_addr));
    *pu8_d1 = getI2C1(I2C_NAK); // last ack bit from master to slave
    // during read must be a NAK
    stopI2C1();
}

void read2I2C1(uint8_t u8_addr, uint8_t* pu8_d1, uint8_t* pu8_d2) {
    startI2C1();
    putI2C1(I2C_RADDR(u8_addr));
    *pu8_d1 = getI2C1(I2C_ACK);
    *pu8_d2 = getI2C1(I2C_NAK);
    stopI2C1();
}
```

**LSB must be 1 for read**

Used for block data transfers.
PIC24 μC Master to 24LC515 Serial EEPROM

EEPROM is 64 K x 8, internally arranged as two separate 32 K 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

<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>1</td>
<td>0</td>
<td>B</td>
<td>A1</td>
<td>A0</td>
<td>R/W#</td>
</tr>
</tbody>
</table>

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

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

(b) Write Operation

<table>
<thead>
<tr>
<th>Memory address high byte</th>
<th>Memory address low byte</th>
<th>Write data</th>
<th>Write data</th>
<th>1 to 64 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSb of address high byte is a don’t care as ‘B’ bit of I2C address is used for this.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(c) End-of-Write Polling

Ack bit returns as “1” (NAK) if write is in progress

Send write command to poll for end-of-write

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

<table>
<thead>
<tr>
<th>S</th>
<th>I²C addr</th>
<th>Rdata</th>
<th>A</th>
<th>Rdata</th>
<th>A</th>
<th>Rdata</th>
<th>A</th>
<th>Rdata</th>
<th>N</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/W# = 1, Read operation</td>
<td>Any number of bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

(b) Random Read

<table>
<thead>
<tr>
<th>S</th>
<th>I²C addr</th>
<th>Addr (hi)</th>
<th>Addr (lo)</th>
<th>A</th>
<th>Rdata</th>
<th>A</th>
<th>Rdata</th>
<th>N</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory address high byte</td>
<td>Start Condition</td>
<td>Read data</td>
<td>Read data</td>
<td>Any number of bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

24LC515 Support Functions (ready polling)

```c
#define EEPROM 0xA0 // LC515 address with both address pins tied low
#define BLKSIZE (64) // Assumes WDT is configured for longer than EEPROM write time

void waitForWriteCompletion(uint8_t u8_i2cAddr) {
    uint8_t u8_ack, u8_savedSWDTEN;
    u8_savedSWDTEN = _SWDTEN;
    _SWDTEN = 1; // enable WDT so that do not get stuck in infinite loop
    u8_i2cAddr = I2C_WADDR(u8_i2cAddr); // write operation, R/W# = 0;
    do {
        startI2C1();
        u8_ack = putNoAckCheckI2C1(u8_i2cAddr);
        stopI2C1();
    } while (u8_ack == I2C_NAK);
    _SWDTEN = u8_savedSWDTEN; // restore WDT to original state
}
```

Assume lower 32k block
24LC515 Support Functions (write)

Write 64 bytes in *pu_buf to device

```c
void memWriteLC515(uint8_t u8_i2cAddr, uint16_t u16_MemAddr, uint8_t *pu8_buf) {
    uint8_t u8_AddrLo, u8_AddrHi;
    u8_AddrLo = u16_MemAddr & 0x00FF;
    u8_AddrHi = (u16_MemAddr >> 8);
    pu8_buf[0] = u8_AddrHi; //place address into buffer
    pu8_buf[1] = u8_AddrLo;
    if (u16_MemAddr & 0x8000) {
        // if MSB set, set block select bit
        u8_i2cAddr = u8_i2cAddr | 0x08;
    }
    waitForWriteCompletion(u8_i2cAddr);
    writeNI2C1(u8_i2cAddr, pu8_buf, BLKSIZE+2);
}
```

- Calculate the high & low bytes of the memory address
- Set the 'B' bit of the I2C memory address if writing to upper 32k block
- Wait for last write to finish
- I2C block write transaction

24LC515 Support Functions (read)

Read 64 bytes from device, return in *pu_buf

```c
void memReadLC515(uint8_t u8_i2cAddr, uint16_t u16_MemAddr, uint8_t *pu8_buf) {
    uint8_t 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;
    }
    waitForWriteCompletion(u8_i2cAddr);
    write2I2C1(u8_i2cAddr, u8_AddrHi, u8_AddrLo);
    readNI2C1(u8_i2cAddr, pu8_buf, BLKSIZE);
}
```

- Calculate the high & low bytes of the memory address
- Set the 'B' bit of the I2C memory address if writing to upper 32k block
- Wait for last write to finish
- Set EEPROM internal address counter
- I2C block read transaction
Testing the 24LC515

```c
int main (void) {
    uint8_t au8_buf[BLKSIZE+2];  // 2 extra bytes for address
    uint16_t u16_MemAddr;
    uint8_t u8_Mode;

    configClock();    // clock configuration
    configDefaultUART(9600);  // serial port configuration
    outString(HELLO_MSG);     // say Hello!
    printResetCause();
    configI2C1(400);        // configure I2C for 400 KHz
    outString("\nEnter 'w' for write mode, anything else reads: ");
    u8_Mode = inCharEcho();
    outString("\n");
    u16_MemAddr = 0;  // start at location 0 in memory

    while (1) {    // In write mode, read 64 characters from the console, write to the 24LC515
        if (u8_Mode == 'w') {
            outString("Enter chars.\n");
            for (u8_i = 2; u8_i < BLKSIZE+2; u8_i++) {  // reserved for starting address
                au8_buf[u8_i] = inCharEcho();
            }
            outString("\nDoing Write\n");
            memWriteLC515(EEPROM, u16_MemAddr, au8_buf);  // do write
            u16_MemAddr = u16_MemAddr + BLKSIZE;
        } else {  // In read mode, read 64 characters from the memory, write to the console.
            memReadLC515(EEPROM, u16_MemAddr, au8_buf);  // do read
            for (u8_i = 0; u8_i <BLKSIZE; u8_i++) outChar(au8_buf[u8_i]);
            outString("\nAny key continues read...\n");
            inChar();
            u16_MemAddr = u16_MemAddr + BLKSIZE;
        }
    }
}
```

Testing the 24LC515 (cont.)
What do you have to know?

- Basic SPI Bus Operation
- I2C Bus operation
- I2C primitive function operation and usage
- I2C Transaction function operation and usage
- 24LC515 EEPROM Operation