CHAPTER 13

I^2C Communication

Each of the PIC32's four inter-integrated circuit (I^2C, pronounced *eye-squared-see*) peripherals allows it to communicate with multiple devices using only two pins. Many devices have I^2C interfaces, including RAM, accelerometers, ADCs, and OLED screens. An advantage of I^2C over SPI is that the I^2C bus has only two wires, no matter how many devices are connected, and each device can act as a master or a slave. A disadvantage is the more complicated support software and the typically lower bit rates—the standard mode is defined as 100 kbit/s and the fast mode is defined as 400 kbit/s—though some I^2C devices allow higher rates.

13.1 Overview

The I^2C bus consists of two wires, one for data (SDA) and one for a clock (SCL), in addition to the common ground reference. Multiple chips can be connected to the same two wires and communicate with each other. A chip can be a master or a slave. Multiple masters and slaves can be connected to the same bus; however, only one master can operate at one time. Masters control the clock signal and hence the speed at which data flows. Usually, I^2C devices operate either in standard 100 kHz or fast 400 kHz mode, although the PIC32 can set arbitrary frequencies.

Figure 13.1 shows a circuit diagram of a typical bus connection. Rather than driving the lines high and low, each pin on the I^2C bus switches between high impedance output (disconnected, floating, high-z) and logic low. The high-z state is equivalent to the open-drain digital output mode, where the pin, rather than being high, is effectively disconnected. Pull-up resistors on each I^2C line ensure that the line is low only when a device outputs a low signal. The PIC32's four I^2C peripherals can handle pull-up voltages between 3.3 and 5 V.

If all devices on a line are high-z, the line is pulled high, a logical 1. If any of the devices on a line are pulling it low, the line is low, a logical 0. When the bus is idle, i.e., no data is being transmitted, all device outputs are high-z, and both SDA and SCL are high.

When a device assumes the role of a master to begin a transmission, it pulls the SDA line low while leaving SCL high. This is the start bit; it tells other devices on the bus that the bus has been claimed by a master, and that they should not attempt to claim it until the transmission is

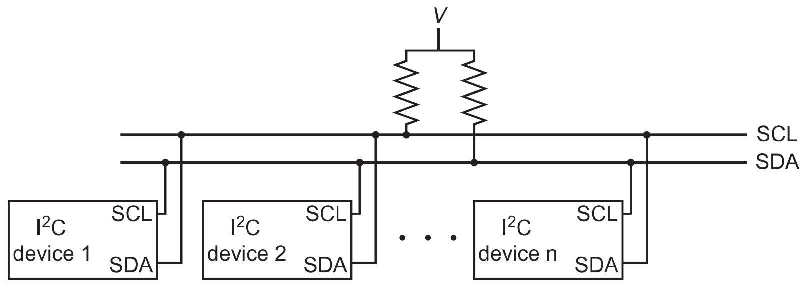

Figure 13.1
n devices on an I^{2}C bus. Each of the two lines is pulled up, typically to 3.3 or 5 V, by a pull-up resistor, unless one of the devices holds the line low. The PIC32 can work with either 3.3 or 5 V. A typical pull-up resistance is 2.4 kΩ, but any nearby value should be fine.

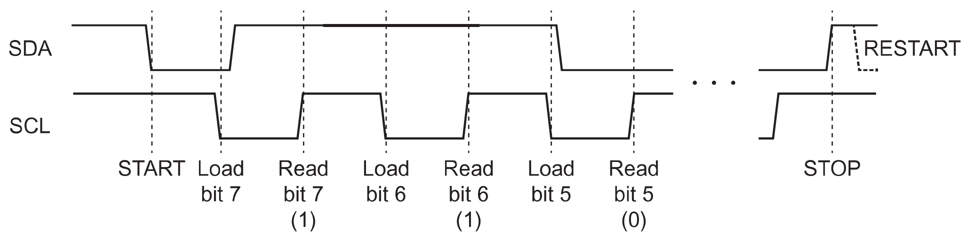

Figure 13.2
An I^{2}C transmission begins when the master pulls SDA low while leaving SCL high. Then the master begins driving the clock SCL. Data is loaded onto SDA at every falling edge of SCL and read from SDA on every rising edge of SCL. In this figure, the transmission begins with 0b110... (these are the highest three bits of the address of the slave being selected). Either the master or a slave can control SDA, depending on whether the master wants to send information to the slave or receive information from the slave. Transmission stops when the master releases SDA, allowing it to go high while SCL is high. A RESTART (dashed line) occurs if the master quickly pulls the SDA line low again, taking control of the bus again before another master can claim it.

finished. The master (or a slave; see below) then transmits data over the SDA line while the master controls the clock SCL. Data is loaded onto SDA when SCL drops low and read from SDA when SCL rises. Eight-bit data bytes are transferred most significant bit first. Transmission stops when the master issues the stop bit: an SDA transition from low to high while SCL is high. See the timing diagram in Figure 13.2.

Unlike UARTs and SPI, I^{2}C employs a handshaking protocol between master and slave. A typical data transaction consists of the following primitives, in order:

1. START: The master issues a start bit, dropping SDA from high to low while SCL stays high.

2. ADDRESS: The master transmits a byte consisting of a 7-bit address and a read-write bit, $R\overline{W}$, in the least significant bit. The 7-bit address indicates which slave the master is addressing; each device on the bus has a unique 7-bit address.[1] If $R\overline{W} = 0$, the master will write to the slave; if it is a 1, the master expects to read data from the slave.
3. ACKNOWLEDGMENT ($\overline{ACK}$/NACK): If a slave has recognized its address, it will respond with a single acknowledgment bit of 0, holding SDA low for the next clock cycle. This is called an $\overline{ACK}$. If SDA is high (no $\overline{ACK}$, also called a NACK), the master knows an error has occurred.
4. WRITE or RECEIVE: If $R\overline{W} = 0$ (write), the master sends a byte over SDA. If $R\overline{W} = 1$ (read), then the slave controls the SDA and sends a byte.
5. ACKNOWLEDGMENT ($\overline{ACK}$/NACK): If it is a write, the slave must send an $\overline{ACK}$ bit to acknowledge that it has received the data. Otherwise the master knows an error has occurred. If it is a read, the master either sends an $\overline{ACK}$ if it wants another byte or a NACK if it is done requesting bytes. If the master wishes to send another byte, or has requested another byte, return to step 4.
6. STOP or RESTART: The master issues a stop bit (SDA raised to high while SCL is high) to end the transmission. This allows other devices to claim control of the bus. If the master wants to keep control of the bus, possibly to change the communication from a write to a read or vice versa, the master can issue a RESTART (or "repeated start") instead of a STOP. This is simply a STOP followed quickly by another START, before another master can claim the bus. In this case, return to step 2.

If two or more masters attempt to take control of an idle bus at approximately the same time, an arbitration process ensures that all but one of them drop out. Each device monitors the state of the SDA line, and if it is ever a 0 (low) while the device is transmitting a 1 (high), the device knows that another master is driving the line, and therefore the device drops out. Devices that lose arbitration report a *bus collision*.

Although the master nominally drives the clock line SCL, a slave may also pull it low. For example, if the slave needs more time to process data sent to it, it can hold the SCL line low when the master tries to let SCL return to high. The master senses this and waits until the slave allows SCL to return high before resuming its normal clock rate. This is called *clock stretching*.

13.2 Details

Seven SFRs control the behavior of I^2C peripherals. Many of the fields in these SFRs initiate an "event" on the I^2C bus. All bits default to zero except I2CxCON.SCLREL. In the SFRs below, x refers to the I^2C peripheral number, 1, 3, 4, or 5.

[1] 10-bit addressing is also supported, but this chapter focuses on 7-bit addresses.

I2CxCON The I^{2}C control register. Setting bits in this register initiates primitive operations used by the I^{2}C protocol. Some bits control an I^{2}C slave's behavior.

I2CxCON⟨15⟩ or I2CxCONbits.ON: Setting this bit enables the I^{2}C module.

I2CxCON⟨12⟩ or I2CxCONbits.SCLREL: In slave mode only, this SCL release control bit is set to release the clock, telling the master it may continue to clock the communication, or cleared to hold the clock low (clock stretching). When the master detects that SCL is low, it delays sending a clock signal, giving the slave more time before responding to the master.

I2CxCON⟨6⟩ or I2CxCONbits.STREN: This SCL stretch enable bit is used in slave mode to control clock stretching. If set to one, the slave will hold SCL low prior to transmitting to and after receiving from the master. When SCL is low, the clock is stretched and the master pauses the clock. If clear, clock stretching only happens at the beginning of a slave transmission. A slave transmission begins whenever the master expects data from the slave, according to the I^{2}C protocol. The transmission does not actually occur until the slave sets I2CxCONbits.SCLREL.

I2CxCON⟨5⟩ or I2CxCONbits.ACKDT: Acknowledge data bit, used only in master mode. If set to one, the master will send a NACK when sending an acknowledgment, signaling that no more data is requested. If set to zero, the master sends an $\overline{\text{ACK}}$ during the acknowledge, signaling that it wants more data.

I2CxCON⟨4:0⟩ These bits initiate various control signals on the bus. While any of these bits is high, you should not set any of the other bits.

I2CxCON⟨4⟩ or I2CxCONbits.ACKEN: Setting this bit initiates an acknowledgment, transmitting the I2CxCON.ACKDT bit. Hardware clears this bit after the ACK ends.

I2CxCON⟨3⟩ or I2CxCONbits.RCEN: Setting this bit initiates a RECEIVE. Hardware clears this bit after the receive has finished.

I2CxCON⟨2⟩ or I2CxCONbits.PEN: Setting this bit initiates a STOP. Hardware clears the bit after the stop is sent.

I2CxCON⟨1⟩ or I2CxCONbits.RSEN: Setting this bit initiates a RESTART. Hardware clears this bit after the restart is finished.

I2CxCON⟨0⟩ or I2CxCONbits.SEN: Setting this bit initiates a START. Hardware clears this bit after the start is finished.

I2CxSTAT Contains the status of the I^{2}C peripheral and results from signals on the I^{2}C bus.

I2CxSTAT⟨15⟩ or I2CxSTATbits.ACKSTAT: If clear (0) an $\overline{\text{ACK}}$ (acknowledge) has been received; otherwise an ACK has not been received.

I2CxSTAT⟨14⟩ or I2CxSTATbits.TRSTAT: If this transmission status bit is set (1) the master is transmitting; otherwise the master is not transmitting. Only used in master mode.

I2CxSTAT⟨6⟩ or I2CxSTATbits.I2COV: Useful for debugging. If set (1), a receive overflow has occurred, meaning the receive buffer contains a byte but another byte has been received. Software must clear this bit.

I2CxSTAT⟨5⟩ or I2CxSTATbits.D_A: In slave mode, indicates whether the most recently received or transmitted information was an address (0) or data (1).

I2CxSTAT⟨2⟩ or I2CxSTATbits.R_W: In slave mode, this bit is a 1 if the master is requesting data from the slave and a 0 if the master is sending data.

I2CxSTAT⟨1⟩ or I2CxSTATbits.RBF: The receive buffer full bit, when set, indicates that a byte has been received and is ready in I2CxRCV.

I2CxSTAT⟨0⟩ or I2CxSTATbits.TBF: When set (1), indicates that the transmit buffer is full and that a transmission is occurring.

I2CxADD This register contains the I^2C peripheral's address. The address is contained in the lower ten bits (although only seven are used in 7-bit address mode). Whenever an ADDRESS is initiated on the I^2C bus, the slave will respond if the ADDRESS matches its own address. Only slaves need to set an address.

I2CxMSK Allows the slave to ignore some address bits.

I2CxBRG The lower 16 bits of this register determine the baud. Typically the baud is either 100 kHz or 400 kHz. To compute the value of I2CxBRG, use the formula

$$\text{I2CxBRG} = \left(\left(\frac{1}{2 \times F_{sck}} - T_{PGD}\right) F_{pb}\right) - 2, \tag{13.1}$$

where F_{sck} is the desired baud, F_{pb} is the peripheral bus clock frequency, and T_{PGD} is 104 ns, according to the Reference Manual. With an 80 MHz peripheral bus clock, I2CxBRG = 390 for 100 kHz and I2CxBRG = 90 for 400 kHz. Only masters need to set a baud rate.

I2CxTRN Used to transmit a byte of data. Write an address to this register when performing the ADDRESS command, or write data when sending a data byte.

I2CxRCV Used to receive data from the I^2C bus. This register contains any data received. On the master, this register only contains data after a RECEIVE request has been initiated. On the slave, this register is loaded whenever the master sends any data, including address bytes.

Interrupts can be generated for a master or the slave by any of the data transaction primitives or by errors detected in the handshaking. Interrupts can also be generated by bus collisions. See the Reference Manual for details. The interrupt vectors are `_I2C_x_VECTOR`, where x is 1, 3, 4, or 5. For I^2C peripheral 1, the flag status bits are in IFS0bits.I2C1BIF (bus collision), IFS0bits.I2C1SIF (slave event), and IFS0bits.I2C1MIF (master event); the enable bits are in IEC0bits.I2C1BIE (bus collision), IEC0bits.I2C1SIE (slave event), and IEC0bits.I2C1MIE (master event); and the priority and subpriority bits are in IPC6bits.I2C1IP and IPC6bits.I2C1IS, respectively. For I^{2}C peripherals 3 to 5, the bits are named similarly, replacing I2C1 with I2Cx, x = 3 to 5. For I2C3, the bits are also in IFS0, IEC0, and IPC6. For I2C4, the bits are in IFS1, IEC1, and IPC7, and for I2C5, they are in IFS1, IEC1, and IPC8.

13.3 Sample Code

13.3.1 Loopback

In this example, a single PIC32 communicates with itself using I^{2}C. Here we use I^{2}C 1 as the master and I^{2}C 5 as a slave.[2]

We have divided the code into three modules: the master, the slave, and the main function, allowing you to test the code on a single PIC32 and then to use two PIC32's to test inter-PIC communication.

First, the master code. The implementation of the I^{2}C master contains functions roughly corresponding to the primitives discussed earlier. Each function executes the primitive command and waits for it to complete. By calling the primitive functions in succession, you can form an I^{2}C transaction.

Code Sample 13.1 `i2c_master_noint.h`**. Header File for I^{2}C Master with No Interrupts.**

```
#ifndef I2C_MASTER_NOINT_H__
#define I2C_MASTER_NOINT_H__
// Header file for i2c_master_noint.c
// helps implement use I2C1 as a master without using interrupts

void i2c_master_setup(void);              // set up I2C 1 as a master, at 100 kHz

void i2c_master_start(void);              // send a START signal
void i2c_master_restart(void);            // send a RESTART signal
void i2c_master_send(unsigned char byte); // send a byte (either an address or data)
unsigned char i2c_master_recv(void);      // receive a byte of data
void i2c_master_ack(int val);             // send an ACK (0) or NACK (1)
void i2c_master_stop(void);               // send a stop

#endif
```

[2] Technically, a single I^{2}C peripheral can simultaneously be a master and a slave. When the master sends data to its slave address, the slave will respond!

Code Sample 13.2 `i2c_master_noint.c`**. Implementation of I^{2}C Master with No Interrupts.**

```
#include "NU32.h"          // constants, funcs for startup and UART
// I2C Master utilities, 100 kHz, using polling rather than interrupts
// The functions must be called in the correct order as per the I2C protocol
// Master will use I2C1 SDA1 (D9) and SCL1 (D10)
// Connect these through resistors to Vcc (3.3 V). 2.4k resistors recommended,
// but something close will do.
// Connect SDA1 to the SDA pin on the slave and SCL1 to the SCL pin on a slave

void i2c_master_setup(void) {
  I2C1BRG = 390;                  // I2CBRG = [1/(2*Fsck) - PGD]*Pblck - 2
                                  // Fsck is the freq (100 kHz here), PGD = 104 ns
  I2C1CONbits.ON = 1;             // turn on the I2C1 module
}

// Start a transmission on the I2C bus
void i2c_master_start(void) {
    I2C1CONbits.SEN = 1;          // send the start bit
    while(I2C1CONbits.SEN) { ; }  // wait for the start bit to be sent
}

void i2c_master_restart(void) {
    I2C1CONbits.RSEN = 1;         // send a restart
    while(I2C1CONbits.RSEN) { ; } // wait for the restart to clear
}

void i2c_master_send(unsigned char byte) { // send a byte to slave
  I2C1TRN = byte;                 // if an address, bit 0 = 0 for write, 1 for read
  while(I2C1STATbits.TRSTAT) { ; } // wait for the transmission to finish
  if(I2C1STATbits.ACKSTAT) {      // if this is high, slave has not acknowledged
    NU32_WriteUART3("I2C2 Master: failed to receive ACK\r\n");
  }
}

unsigned char i2c_master_recv(void) { // receive a byte from the slave
    I2C1CONbits.RCEN = 1;             // start receiving data
    while(!I2C1STATbits.RBF) { ; }    // wait to receive the data
    return I2C1RCV;                   // read and return the data
}

void i2c_master_ack(int val) {        // sends ACK = 0 (slave should send another byte)
                                      // or NACK = 1 (no more bytes requested from slave)
    I2C1CONbits.ACKDT = val;          // store ACK/NACK in ACKDT
    I2C1CONbits.ACKEN = 1;            // send ACKDT
    while(I2C1CONbits.ACKEN) { ; }    // wait for ACK/NACK to be sent
}

void i2c_master_stop(void) {          // send a STOP:
  I2C1CONbits.PEN = 1;                // comm is complete and master relinquishes bus
  while(I2C1CONbits.PEN) { ; }        // wait for STOP to complete
}
```

Next the slave code. The slave code uses I^{2}C 5, and, upon a read request, will return the last two bytes written to the slave. As the slave is interrupt driven, it will work as soon as you call `i2c_slave_setup`, providing the desired 7-bit address (the master is configured to talk to a slave on address 0x32). The slave interrupt reads the status flags so that it can discriminate between reads, writes, address bytes, and data bytes. Notice that, when the slave wishes to send data to the master, it must release SCL to be controlled by the master by setting I2C5CONbits.SCLREL to one.

Code Sample 13.3 `i2c_slave.h`**. Header File for I^{2}C Slave.**

```
#ifndef I2C_SLAVE_H__
#define I2C_SLAVE_H__
// implements a basic I2C slave

void i2c_slave_setup(unsigned char addr); // set up the slave at the given address

#endif
```

Code Sample 13.4 `i2c_slave.c`**. Implementation of an I^{2}C Slave.**

```
#include "NU32.h"   // constants, funcs for startup and UART
// Implements a I2C slave on I2C5 using pins SDA5 (F4) and SCL5 (F5)
// The slave returns the last two bytes the master writes

void __ISR(_I2C_5_VECTOR, IPL1SOFT) I2C5SlaveInterrupt(void) {
  static unsigned char bytes[2];    // store two received bytes
  static int rw = 0;                // index of the bytes read/written
  if(rw == 2) {                     // reset the data index after every two bytes
    rw = 0;
  }
  // We have to check why the interrupt occurred.  Some possible causes:
  // (1) slave received its address with RW bit = 1:  read address & send data to master
  // (2) slave received its address with RW bit = 0:  read address (data will come next)
  // (3) slave received an ACK in RW = 1 mode:        send data to master
  // (4) slave received a data byte in RW = 0 mode:   store this data sent by master

  if(I2C5STATbits.D_A) {       // received data/ACK, so Case (3) or (4)
    if(I2C5STATbits.R_W) {     // Case (3):  send data to master
      I2C5TRN = bytes[rw];     // load slave's previously received data to send to master
      I2C5CONbits.SCLREL = 1;  // release the clock, allowing master to clock in data
    } else {                   // Case (4):  we have received data from the master
      bytes[rw] = I2C5RCV;     // store the received data byte
    }
    ++rw;
  } else {                     // the byte is an address byte, so Case (1) or (2)
    I2C5RCV;                   // read to clear I2C5RCV (we don't need our own address)
    if(I2C5STATbits.R_W) {     // Case (1):  send data to master
      I2C5TRN = bytes[rw];     // load slave's previously received data to send to master
      ++rw;
      I2C5CONbits.SCLREL = 1;  // release the clock, allowing master to clock in data
    }                          // Case (2):  do nothing more, wait for data to come
  }
  IFS1bits.I2C5SIF = 0;
}

// I2C5 slave setup (disable interrupts before calling)
void i2c_slave_setup(unsigned char addr) {
  I2C5ADD = addr;                   // the address of the slave
  IPC8bits.I2C5IP  = 1;            // slave has interrupt priority 1
  IEC1bits.I2C5SIE = 1;            // slave interrupt is enabled
  IFS1bits.I2C5SIF = 0;            // clear the interrupt flag
  I2C5CONbits.ON   = 1;            // turn on i2c2
}
```

Next, the main program, in which the PIC32 communicates with itself over I^2C. The main program initializes the slave and master and performs some I^2C transactions, sending the results over the UART to your terminal. Notice how the primitive operations are assembled into a single transaction. You should connect the clock (SCL) and data (SDA) lines of I^2C 1 and I^2C 5, along with the pull-up resistors. To examine what happens when the slave does not return an ACK, disconnect the I^2C bus wires.

Code Sample 13.5 `i2c_loop.c`**. The Main I^2C Loopback Program.**

```
#include "NU32.h"          // config bits, constants, funcs for startup and UART
#include "i2c_slave.h"
#include "i2c_master_noint.h"
// Demonstrate I2C by having the I2C1 talk to I2C5 on the same PIC32
// Master will use SDA1 (D9) and SCL1 (D10).  Connect these through resistors to
// Vcc (3.3 V) (2.4k resistors recommended, but around that should be good enough)
// Slave will use SDA5 (F4) and SCL5 (F5)
// SDA5 -> SDA1
// SCL5 -> SCL1
// Two bytes will be written to the slave and then read back to the slave.
#define SLAVE_ADDR 0x32

int main() {
  char buf[100] = {};                       // buffer for sending messages to the user
  unsigned char master_write0 = 0xCD;       // first byte that master writes
  unsigned char master_write1 = 0x91;       // second byte that master writes
  unsigned char master_read0  = 0x00;       // first received byte
  unsigned char master_read1  = 0x00;       // second received byte

  NU32_Startup();                 // cache on, interrupts on, LED/button init, UART init
  __builtin_disable_interrupts();
  i2c_slave_setup(SLAVE_ADDR);              // init I2C5, which we use as a slave
                                            //  (comment out if slave is on another pic)
  i2c_master_setup();                       // init I2C2, which we use as a master
  __builtin_enable_interrupts();

  while(1) {
    NU32_WriteUART3("Master: Press Enter to begin transmission.\r\n");
    NU32_ReadUART3(buf,2);
    i2c_master_start();                     // Begin the start sequence
    i2c_master_send(SLAVE_ADDR << 1);       // send the slave address, left shifted by 1,
                                            // which clears bit 0, indicating a write
    i2c_master_send(master_write0);         // send a byte to the slave
    i2c_master_send(master_write1);         // send another byte to the slave
    i2c_master_restart();                   // send a RESTART so we can begin reading
    i2c_master_send((SLAVE_ADDR << 1) | 1); // send slave address, left shifted by 1,
                                            // and then a 1 in lsb, indicating read
    master_read0 = i2c_master_recv();       // receive a byte from the bus
    i2c_master_ack(0);                      // send ACK (0): master wants another byte!
    master_read1 = i2c_master_recv();       // receive another byte from the bus
    i2c_master_ack(1);                      // send NACK (1):  master needs no more bytes
    i2c_master_stop();                      // send STOP:  end transmission, give up bus

    sprintf(buf,"Master Wrote: 0x%x 0x%x\r\n", master_write0, master_write1);
    NU32_WriteUART3(buf);
    sprintf(buf,"Master Read: 0x%x 0x%x\r\n", master_read0, master_read1);
    NU32_WriteUART3(buf);
    ++master_write0;                        // change the data the master sends
    ++master_write1;
  }
  return 0;
}
```

Using I^2C to have the PIC32 communicate with itself may not seem practical, but it helps demonstrate the peripheral without involving other chips. If you have another PIC32 available, you can compile the slave as a standalone program using `i2c_slave_loop.c` (below). By connecting the master loopback example to a slave on another chip you can witness inter-PIC communication.

To use `i2c_slave_loop.c`, compile and link it with `i2c_slave.c` and run it on another NU32. Connect the SDA and SCL pins on the two NU32s, as well as the pull-up resistors. Power the slave NU32 from the master NU32 by connecting the GND rails of the two NU32s together, and by connecting the two 6 V rails together. Plug in the master NU32 while making sure that the slave NU32 is unplugged but with its power switch on. Both NU32's will turn on and off based on the state of the master's switch.

Code Sample 13.6 `i2c_slave_loop.c`**. A Standalone I^2C Slave.**

```
#include "i2c_slave.h"
#include "NU32.h"

int main() {
  NU32_Startup();
  i2c_slave_setup(0x32); // enable the slave w/ address 0x32

  while(1) {             // the slave is handled in an interrupt in i2c_slave.c
    _nop();              // so we do nothing.
  }
  return 0;
}
```

13.3.2 Interrupt-Based Master

In Section 13.3.1, we created functions for each I^2C primitive: the functions initiate a command and wait for it to complete. In this section, we provide interrupt-based master code. The function `i2c_write_read` allows the master to initiate a write-read transaction by providing a slave address, an input array with the bytes to write, and an output array with the bytes that are read. If the length of the write or read array is zero, then that specific action will not be performed.

The interrupt removes the need to wait for each primitive operation to complete. Instead, an interrupt triggers at the end of each primitive. The ISR tracks the state of the current communication and performs the appropriate state transition. As coded, the function `i2c_write_read` waits for the whole transaction to complete before returning; however, in time-critical applications this behavior could be modified. Calling `i2c_write_read` would initiate the transaction, but not wait for it to finish. Mainline code could continue to execute, and either check for the result of the transaction at a later time or handle the transaction results from within the ISR.

Code Sample 13.7 `i2c_master_int.h`**. Header File for Interrupt-Based I^2C Master.**

```
#ifndef I2C_MASTER_INT__H__
#define I2C_MASTER_INT__H__

// buffer pointer type.  The buffer is shared by an ISR and mainline code.
// the pointer to the buffer is also shared by an ISR and mainline code.
// Hence the double volatile qualification
typedef volatile unsigned char * volatile buffer_t;

void i2c_master_setup(); //sets up I2C1 as a master using an interrupt

// Initiate an I2C write read operation at the given address.
// You can optionally only read or only write by passing 0 length for reading or writing.
// This will not return until the transaction is complete.  Returns false on error.
int i2c_write_read(unsigned int addr, const buffer_t write, unsigned int wlen,
    const buffer_t read, unsigned int rlen );

// write a single byte to the slave
int i2c_write_byte(unsigned int addr, unsigned char byte);

#endif
```

Code Sample 13.8 `i2c_master_int.c`**. Implementation of an Interrupt-Based I^2C Master.**

```
#include "NU32.h"          // constants, funcs for startup and UART
#include "i2c_master_int.h"
// I2C Master utilities, using interrupts
// Master will use I2C1 SDA1 (D9) and SCL1 (D10)
// Connect these through resistors to Vcc (3.3V). 2.4k resistors recommended, but
// something close will do.
// Connect SDA1 to the SDA pin on a slave device and SCL1 to the SCL pin on a slave.

// keeps track of the current I2C state
static volatile enum {IDLE,START,WRITE,READ,RESTART,ACK,NACK,STOP,ERROR} state = IDLE;

static buffer_t to_write = NULL;           // data to write
static buffer_t  to_read = NULL;           // data to read
static volatile unsigned char address = 0; // the 7-bit address to write to / read from
static volatile unsigned int  n_write = 0; // number of data bytes to write
static volatile unsigned int  n_read  = 0; // number of data bytes to read


void __ISR(_I2C_1_VECTOR, IPL1SOFT) I2C1MasterInterrupt(void) {
  static unsigned int write_index = 0, read_index = 0;  //indexes the read/write arrays

  switch(state) {
    case START:                           // start bit has been sent
      write_index = 0;                    // reset indices
      read_index = 0;
      if(n_write > 0) {                   // there are bytes to write
        state = WRITE;                    // transition to write mode
        I2C1TRN = address << 1;           // send the address, with write mode set
      } else {
        state = ACK;                      // skip directly to reading
        I2C1TRN = (address << 1) & 1;
      }

      break;
    case WRITE:                           // a write has finished
      if(I2C1STATbits.ACKSTAT) {          // error: didn't receive an ACK from the slave
        state = ERROR;
      } else {
        if(write_index < n_write) {          // still more data to write
          I2C1TRN = to_write[write_index];  // write the data
          ++write_index;
        } else {                             // done writing data, time to read or stop
          if(n_read > 0) {                   // we want to read so issue a restart
            state = RESTART;
            I2C1CONbits.RSEN = 1;            // send the restart to begin the read
          } else {                           // no data to read, issue a stop
            state = STOP;
            I2C1CONbits.PEN = 1;
          }
        }
      }
      break;
    case RESTART: // the restart has completed
      // now we want to read, send the read address
      state = ACK;       // when interrupted in ACK mode, we will initiate reading a byte
      I2C1TRN = (address << 1) | 1; // the address is sent with the read bit sent
      break;
    case READ:
      to_read[read_index] = I2C1RCV;
      ++read_index;
      if(read_index == n_read) { // we are done reading, so send a nack
        state = NACK;
        I2C1CONbits.ACKDT = 1;
      } else {
        state = ACK;
        I2C1CONbits.ACKDT = 0;
      }
      I2C1CONbits.ACKEN = 1;
      break;
    case ACK:
            // just sent an ack meaning we want to read more bytes
      state = READ;
      I2C1CONbits.RCEN = 1;
      break;
    case NACK:
      //issue a stop
      state = STOP;
      I2C1CONbits.PEN = 1;
      break;
    case STOP:
      state = IDLE; // we have returned to idle mode, indicating that the data is ready
      break;
    default:
      // some error has occurred
      state = ERROR;
  }
  IFS0bits.I2C1MIF = 0;       //clear the interrupt flag
}

void i2c_master_setup() {
  int ie = __builtin_disable_interrupts();
  I2C1BRG = 90;                      // I2CBRG = [1/(2*Fsck) - PGD]*Pblck - 2
                                     // Fsck is the frequency (400 kHz here), PGD = 104ns
                                     // this is 400 khz mode
                                     // enable the i2c interrupts
  IPC6bits.I2C1IP  = 1;              // master has interrupt priority 1
  IEC0bits.I2C1MIE = 1;              // master interrupt is enabled
  IFS0bits.I2C1MIF = 0;              // clear the interrupt flag
  I2C1CONbits.ON = 1;                // turn on the I2C2 module

  if(ie & 1) {
    __builtin_enable_interrupts();
  }
}

// communicate with the slave at address addr.  first write wlen bytes to the slave,
// then read rlen bytes from the slave
int i2c_write_read(unsigned int addr, const buffer_t write,
    unsigned int wlen, const buffer_t read, unsigned int rlen ) {
  n_write = wlen;
  n_read = rlen;
  to_write = write;
  to_read = read;
  address = addr;
  state = START;
  I2C1CONbits.SEN = 1;        // initialize the start
  while(state != IDLE && state != ERROR) { ; }  // initialize the sequence
  return state != ERROR;
}

// write a single byte to the slave
int i2c_write_byte(unsigned int addr, unsigned char byte) {
  return i2c_write_read(addr,&byte,1,NULL,0);
}
```

13.3.3 Accelerometer/Magnetometer

The STMicroelectronics LSM303D is a three-axis accelerometer and magnetometer that can be used as a digital compass. It also has a temperature sensor. More details about this sensor and the breakout board are discussed in Chapter 12. Here we use the same accelerometer library and example that we used in Chapter 12, except now our implementation uses I^2C rather than SPI.

When used with the Pololu breakout board, the accelerometer has an I^2C address of 0x1D. The example code continuously displays the raw accelerometer, magnetometer, and temperature sensor values. The necessary code is Code Samples 12.3 and 12.4 from Chapter 12, as well as the following code.

Code Sample 13.9 `i2c_accel.c`. **I^2C Implementation of the Basic Accelerometer Library. Requires** `i2c_master_int.h`.

```
#include "accel.h"
#include "i2c_master_int.h"
#include <stdlib.h>

#define I2C_ADDR 0x1D // the I2C slave address

// Wire GND to GND, VDD to 3.3V, SDA to SDA2 (RA3) and SCL to SCL2 (RA2)

// read data from the accelerometer, given the starting register address.
// return the data in data
void acc_read_register(unsigned char reg, unsigned char data[], unsigned int len)
{
  unsigned char write_cmd[1] = {};
  if(len > 1) { // want to read more than 1 byte and we are reading from the accelerometer
    write_cmd[0] = reg | 0x80; // make the MSB of addr 1 to enable auto increment
  }
  else {
    write_cmd[0] = reg;
  }
  i2c_write_read(I2C_ADDR,write_cmd, 1, data,len);
}

void acc_write_register(unsigned char reg, unsigned char data)
{
  unsigned char write_cmd[2];
  write_cmd[0] = reg;   // write the register
  write_cmd[1] = data;  // write the actual data
  i2c_write_read(I2C_ADDR, write_cmd, 2, NULL, 0);
}

void acc_setup() {                 // set up the accelerometer, using I2C 2
  i2c_master_setup();
  acc_write_register(CTRL1, 0xAF); // set accelerometer data rate to 1600 Hz.
                                   // Don't update until we read values
  acc_write_register(CTRL5, 0xF0); // 50 Hz magnetometer, high resolution, temp sensor on
  acc_write_register(CTRL7, 0x0);  // enable continuous reading of the magnetometer
}
```

13.3.4 OLED Screen

An organic light emitting diode (OLED) screen is a low-power, high-resolution monochrome display. In this example we use an inexpensive 128 x 64 pixel OLED screen (128 columns and 64 rows) with an onboard SSD1306 controller chip, as can be found on hobbyist websites. The PIC32 uses I^{2}C to communicate with the SSD1306. The OLED library we provide gives a simple interface to the OLED display; however, it does not provide comprehensive access to all of the controller's functions.

Each pixel is represented by a single bit. The PIC32 stores pixel data in a framebuffer in PIC32 RAM, a copy of the OLED controller's RAM. The functions `display_pixel_set` and `display_pixel_get` access this framebuffer rather than directly accessing the OLED controller's memory. The function `display_draw` copies the whole framebuffer to the OLED controller, which updates the screen.

The sample code consists of three files: the two files of the OLED library `i2c_display.{c,h}` and a main program `i2c_pixels.c`. The main program uses the OLED library to draw diagonal lines across the screen (Figure 13.3).

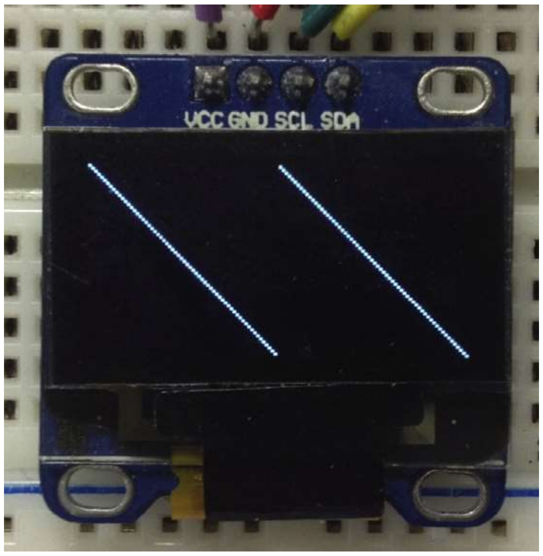

Figure 13.3
An OLED screen.

Code Sample 13.10 `i2c_display.h`**. Header File for Controlling an OLED Display.**

```
#ifndef I2C_DISPLAY_H__
#define I2C_DISPLAY_H__
// bare-bones driver for interfacing with the SSD1306 OLED display via I2C
// not fully featured, just demonstrates basic operation
// note that resetting the PIC doesn't reset the OLED display, only power cycling does

#define WIDTH 128 //display width in bits
#define HEIGHT 64 //display height, in bits

void display_init(void); // initialize I2C1

void display_command(unsigned char cmd); // issue a command to the display

void display_draw(void);                 // draw the buffer in the display

void display_clear(void);                // clear the display

void display_pixel_set(int row, int col, int val); // set pixel at given row and column

int display_pixel_get(int row, int col);  // get the pixel at the given row and column

#endif
```

Code Sample 13.11 `i2c_display.c`**. OLED Screen Interfacing Code.**

```
#include "i2c_master_int.h"
#include "i2c_display.h"
#include <stdlib.h>
// control the SSD1306 OLED display

#define DISPLAY_ADDR 0x3C

#define SIZE WIDTH*HEIGHT/8 //display size, in bytes

static unsigned char video_buffer[SIZE+1] = {0};// buffer corresponding to display pixels
                                                // for sending over I2C. The first byte
                                                // lets us to store the control character
static unsigned char * gddram = video_buffer + 1; // the video buffer start, excluding
                                                  // address byte we write these pixels
                                                  // to GDDRAM over I2C


void display_command(unsigned char cmd) {// write a command to the display
  unsigned char to_write[] = {0x00,cmd}; // 1st byte = 0 (CO = 0, DC = 0), 2nd is command
  i2c_write_read(DISPLAY_ADDR, to_write,2, NULL, 0);
}

void display_init() {
  i2c_master_setup();
                          // goes through the reset procedure
  display_command(0xAE);  // turn off display

  display_command(0xA8);     // set the multiplex ratio (how many rows are updated per
                             // oled driver clock) to the number of rows in the display
  display_command(HEIGHT-1); // the ratio set is the value sent+1, so subtract 1
```

```
                         // we will always write the full display on a single update.
  display_command(0x20); // set address mode
  display_command(0x00); // horizontal address mode
  display_command(0x21); // set column address
  display_command(0x00); // start at 0
  display_command(0xFF); // end at 127
                         // with this address mode, the address will go through all
                         // the pixels and then return to the start,
                         // hence we never need to set the address again

  display_command(0x8d); // charge pump
  display_command(0x14); // turn on charge pump to create ~7 Volts needed to light pixels
  display_command(0xAF); // turn on the display
  video_buffer[0] = 0x40;// co = 0, dc =1, allows us to send data directly from video
                         // buffer, 0x40 is the "next bytes have data" byte
}

void display_draw() {    // copies data to the gddram on the oled chip
  i2c_write_read(DISPLAY_ADDR, video_buffer, SIZE + 1, NULL, 0);
}

void display_clear() {
  memset(gddram,0,SIZE);
}

// get the position in gddram of the pixel position
static inline int pixel_pos(int row, int col) {
  return (row/8)*WIDTH + col;
}

// get a bitmask for the actual pixel position, based on row
static inline unsigned char pixel_mask(int row) {
  return 1 << (row % 8);
}

// invert the pixel at the given row and column
void display_pixel_set(int row, int col,int val) {
  if(val) {
    gddram[pixel_pos(row,col)] |= pixel_mask(row);   // set the pixel
  } else {
    gddram[pixel_pos(row,col)] &= ~pixel_mask(row);  // clear the pixel
  }
}

int display_pixel_get(int row, int col) {
  return (gddram[pixel_pos(row,col)] & pixel_mask(row)) != 0;
}
```

Code Sample 13.12 `i2c_pixels.c`**. Draw Some Lines on an OLED Screen.**

```
#include "NU32.h"          // constants, funcs for startup and UART
#include "i2c_display.h"
// Tests the OLED driver by drawing pixels

int main() {
  NU32_Startup();  // cache on, interrupts on, LED/button init, UART init
  display_init();
```

```
    int row, col;
    for(col = 0; col < WIDTH; ++col) { // draw a diagonal line
      row = col % HEIGHT;              // when we hit the last row
      display_pixel_set(row,col,1);   // start from row 0, but keep advancing
                                      // the column
      display_draw();                 // we draw every update, to display progress.
    }
    display_draw();

    return 0;
  }
```

13.3.5 Multiple Devices

To test three devices on the same I^2C bus—the PIC32, the accelerometer, and the OLED screen—and to have a little fun, we implemented the classic arcade game *Snake* (Figure 13.4). The goal of this game is to move the snake, represented by a string of pixels, to eat food without the snake's head running into the boundaries of the screen or the body of the snake itself. The snake's head moves north, south, east, or west depending on the direction the player tilts the accelerometer, and the snake's body trails along behind the head. When the snake's head passes over a food pixel, a new food pixel appears, and the body of the snake grows by one pixel, increasing the challenge.

The OLED screen and the accelerometer are used as slaves, and they have different slave addresses. The code relies on `i2c_snake.c` (below), `i2c_accel.c`, `accel.h`, `i2c_master_int.c`, and `i2c_master_int.h`.

Code Sample 13.13 `i2c_snake.c`**. The Game of Snake, on an OLED Screen.**

```
#include "NU32.h"        // cache on, interrupts on, LED/button init, UART init
#include "i2c_display.h"
#include "accel.h"

// the game of snake, on an oled display. eat those pixels!
```

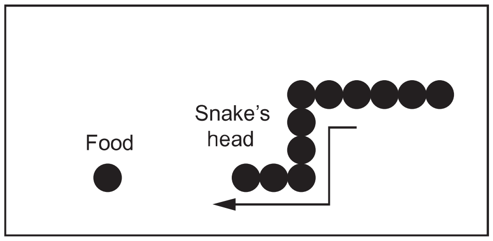

Figure 13.4
The game of *Snake*. Eat the food, and do not crash into the wall or yourself!

```
#define MAX_LEN WIDTH*HEIGHT

typedef struct {
  int head;
  int tail;
  int rows[MAX_LEN];
  int cols[MAX_LEN];
} snake_t; // hold the snake

// direction of the snake
typedef enum {NORTH = 0, EAST = 1, SOUTH = 2, WEST= 3} direction_t;

// grow the snake in the appropriate direction, returns false if snake has crashed
int snake_grow(snake_t * snake, direction_t dir) {
  int hrow = snake->rows[snake->head];
  int hcol = snake->cols[snake->head];

  ++snake->head;
  if(snake->head == MAX_LEN) {
    snake->head = 0;
  }
  switch(dir) {                 // move the snake in the appropriate direction
    case NORTH:
      snake->rows[snake->head] = hrow -1;
      snake->cols[snake->head] = hcol;
      break;
    case SOUTH:
      snake->rows[snake->head] = hrow + 1;
      snake->cols[snake->head] = hcol;
      break;
    case EAST:
      snake->rows[snake->head] = hrow;
      snake->cols[snake->head] = hcol + 1;
      break;
    case WEST:
      snake->rows[snake->head] = hrow;
      snake->cols[snake->head] = hcol -1;
      break;
   }
  // check for collisions with the wall or with itself and return 0, otherwise return 1
  if(snake->rows[snake->head] < 0 || snake->rows[snake->head] >= HEIGHT
      || snake->cols[snake->head] < 0 || snake->cols[snake->head] >= WIDTH) {
    return 0;
  } else if(display_pixel_get(snake->rows[snake->head],snake->cols[snake->head]) == 1) {
      return 0;
  } else {
    display_pixel_set(snake->rows[snake->head],snake->cols[snake->head],1);
    return 1;
  }

}

void snake_move(snake_t * snake)  { // move the snake by deleting the tail
  display_pixel_set(snake->rows[snake->tail],snake->cols[snake->tail],0);
  ++snake->tail;
  if(snake->tail == MAX_LEN) {
    snake->tail = 0;
  }
}
```

```
int main(void) {
  NU32_Startup();
  display_init();
  acc_setup();

  while(1) {
    snake_t snake = {5, 0, {20,20,20,20,20,20},{20,21,22,23,24,25}};
    int dead = 0;
    direction_t dir = EAST;
    char dir_key = 0;
    char buffer[3];
    int i;
    int crow, ccol;
    int eaten = 1;
    int grow = 0;
    short acc[2]; // x and y accleration
    short mag;
    for(i = snake.tail; i <= snake.head; ++i) { // draw the initial snake
      display_pixel_set(snake.rows[i],snake.cols[i],1);
    }
    display_draw();
    acc_read_register(OUT_X_L_M,(unsigned char *)&mag,2);
    srand(mag); // seed the random number generator with the magnetic field
                // (not the most random, but good enough for this game)
    while(!dead) {
      if(eaten) {
        crow = rand() % HEIGHT;
        ccol = rand() % WIDTH;
        display_pixel_set(crow,ccol,1);
        eaten = 0;
      }

      //determine direction based on largest magnitude accel and its direction
      acc_read_register(OUT_X_L_A,(unsigned char *)&acc,4);
      if(abs(acc[0]) > abs(acc[1])) { // move snake in direction of largest acceleration
        if(acc[0] > 0) {              // prevent snake from turning 180 degrees,
          if(dir != EAST) {           // resulting in an automatic self crash
            dir = WEST;
          }
        } else
          if( dir != WEST) {
            dir = EAST;
          }
      } else {
        if(acc[1] > 0) {
          if( dir != SOUTH) {
            dir = NORTH;
          }
        } else {
          if( dir != NORTH) {
            dir = SOUTH;
          }
        }
      }
      if(snake_grow(&snake,dir)) {
        snake_move(&snake);
      } else if(snake.rows[snake.head] == crow && snake.cols[snake.head] == ccol) {
          eaten = 1;
          grow += 15;
      } else {
        dead = 1;
```

```
            display_clear();
          }
          if(grow > 0) {
            snake_grow(&snake,dir);
            --grow;
          }
          display_draw();
        }
      }
      return 0;
    }
```

13.4 Chapter Summary

- I^2C communication requires two wires, one for the clock (SCL) and another for data (SDA). Both lines should be pulled up to 3.3 or 5 V with resistors.
- An I^2C device can be either a master or a slave. The master controls the clock and initiates all communication. Usually the PIC32 operates as the master, but it can also be a slave.
- Each slave has a unique address, and it only responds when the master issues its address, allowing multiple slaves to be connected to a single master. Multiple devices can assume the role of master, but only one master can operate at a time.
- The I^2C peripheral automatically handles the primitives of I^2C communication, like ADDRESS, WRITE, and RECEIVE. A full I^2C transaction, however, requires a sequence of these primitive commands that must be handled in software. The master can sequence the commands by polling for status flags indicating the completion of a primitive, or by generating interrupts on the completion of primitives.

13.5 Exercises

1. Why do the I^2C bus lines require pull-up resistors?
2. Write a program that reads a series of bytes (entered as hexadecimal numbers) from the terminal emulator and sends them over I^2C to an external chip. It should then display the data received over I^2C as hexadecimal numbers. If you want, you may also allow for direct control over sending various I^2C primitives such as START or RESET. Note: you can read and write hexadecimal numbers using the `\%x` format specifier with `sscanf` and `sprintf`.

Further Reading

LSM303D ultra compact high performance e-compass 3D accelerometer and 3D magnetometer module. (2012). STMicroelectronics.
PIC32 family reference manual. Section 24: Inter-integrated circuit. (2013). Microchip Technology Inc.
SSD1306 OLED/PLED segment/common driver with controller. (2008). Solomon Systech.
UM10204 I^2C*-bus specification and user manual* (v. 6). (2014). NXP Semiconductors.