CHAPTER 7

Starting with Serial Communication

Chapter Outline

7.1 Introducing Synchronous Serial Communication

There is an unending need in computer systems to move data around — lots of it. In Chapters 1 and 2 we came across the idea of data buses, on which data flies backwards and forwards between different parts of a computer. In these buses, data is transferred in *parallel.* There is

Fast and Effective Embedded Systems Design. DOI: 10.1016/B978-0-08-097768-3.00007-6
Copyright © 2012 Elsevier Ltd. All rights reserved.

one wire for each bit of data, and one or two more to provide synchronization and control; data is transferred a whole word at a time. This works well, but it requires a lot of wires and a lot of connections on each device that is being interconnected. It is bad enough for an 8-bit device, but for 16 or 32 bits the situation is far worse. An alternative to parallel communication is *serial*. Here, a single wire is effectively used for data transfer, with bits being sent in turn. A few extra connections are almost inevitably needed, for example for earth return, and synchronization and control.

Once we start applying the serial concept, a number of challenges arise. How does the receiver know when each bit begins and ends, and how does it know when each word begins and ends? There are several ways of responding to these questions. A straightforward approach is to send a clock signal alongside the data, with one clock pulse per data bit. The data is *synchronized* to the clock. This idea, called synchronous serial communication, is represented in Figure 7.1. When no data is being sent, there is no movement on the clock line. Every time the clock pulses, however, one bit is output by the transmitter and should be read in by the receiver. In general, the receiver synchronizes its reading of the data with one edge of the clock. In this example it is the rising edge, highlighted by a dotted line.

A simple serial data link is shown in Figure 7.2. Each device that connects to the data link is sometimes called a *node*. In this figure, Node 1 is designated *Master*; it controls what is going on, as it controls the clock. The *Slave* is similar to the master, but receives the clock signal from the master.

An essential feature of the serial link shown, and indeed of most serial links, is a *shift register*. This is made up of a string of digital flip-flops, connected so that the output of one is connected to the input of the next. Each flip-flop holds one bit of information. Every time the shift register is pulsed by the clock signal, each flip-flop passes its bit on to its neighbor on one side and receives a new bit from its other neighbor. The one at the input end clocks in data received from the outside world, and the one of the output end outputs its bit. Therefore, as the clock pulses the shift register can be both feeding in external data and outputting data. The data held by all the flip-flops in the shift register can, moreover, be read all at the same time, as a parallel word, or a new value can be loaded in. In summary, the shift register is an incredibly useful subsystem: it can convert serial data to parallel data, and vice versa, and it can act as a serial transmitter and/or a serial receiver.

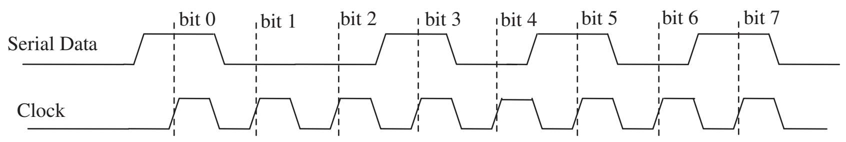


Figure 7.1:
Synchronous serial data

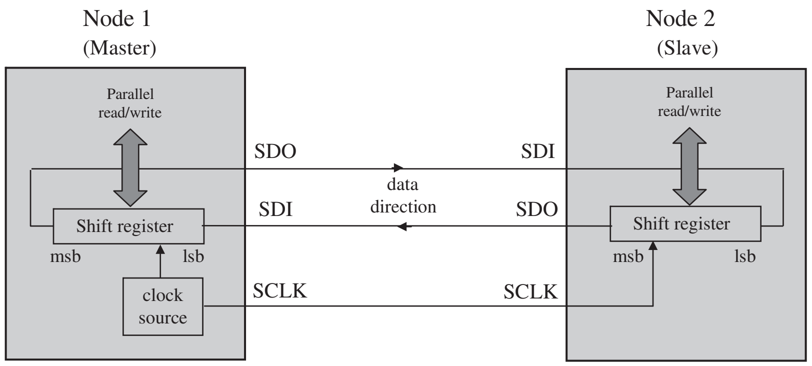


Figure 7.2:
A simple serial link. SDI: serial data in; SDO: serial data out; SCLK: serial clock

As an example, suppose both shift registers in Figure 7.2 are 8-bit, i.e. each has eight flip-flops. Each register is loaded with a new word, and the master then generates eight clock pulses. For each clock cycle, one new bit of data appears at the output end of each shift register, indicated by the SDO (serial data output) label. Each SDO output is, however, connected to the input (SDI: serial data input) of the other register. Therefore as each bit is clocked out of one register, it is clocked in to the other. After eight clock cycles, the word that was in the master shift register is now in the slave, and the word that was in the slave shift register is now held in the master.

The circuitry for the master is usually placed within a microcontroller. The slave might be another microcontroller or some other peripheral device. In general, the hardware circuitry that allows serial data to be sent and received, and interfaces between the microcontroller central processing unit (CPU) and the outside world, is called a serial port.

7.2 Serial Peripheral Interface

To ensure that serial data links are reliable, and can be applied by different devices in different places, several standards or *protocols* have been defined. One that is very widely used these days is the universal serial bus (USB). A protocol defines details of timing and signals, and may go on to define other things, such as the type of connector used. Serial peripheral interface (SPI) is a simple protocol that has had a large influence in the embedded world.

7.2.1 Introducing SPI

In the early days of microcontrollers, both National Semiconductors and Motorola started introducing simple serial communication, based on Figure 7.2. Each formulated a set of rules governing how their microcontrollers worked, and allowed others to develop devices that could interface correctly. These became *de facto* standards, in other words they were never initially formally designed as standards, but were adopted by others to the point where they acted as a formal standard. Motorola called its standard *Serial Peripheral Interface (SPI)* and National Semiconductors called theirs *Microwire*. They are very similar to each other.

It was not long before both SPI and Microwire were adopted by manufacturers of other integrated circuits (ICs), who wanted their devices to be able to work with the new generation of microcontrollers. The SPI has become one of the most durable standards in the world of electronics, applied to short-distance communications, typically within a single piece of equipment. There is no formal document defining SPI, but datasheets for the Motorola 68HC11 (now an old microcontroller) effectively define it in full. Good related texts also do, for example Reference 7.1.

In SPI one microcontroller is designated the master; it controls all activity on the serial interconnection. The master communicates with one or more slaves. A minimum SPI link uses just one master and one slave, and follows the pattern of Figure 7.2. The master generates and controls the clock, and hence all data transfer. The SDO of one device is connected to the SDI of the other, and vice versa. In every clock cycle one bit is moved from master to slave, and one bit from slave to master; after eight clock cycles a whole byte has been transferred. Thus, data is actually transferred in both directions – in old terminology this is called *full duplex* – and it is up to the devices to decide whether a received byte is intended for it or not. If data transfer in only one direction is wanted, then the data transfer line that is not needed can be omitted.

If more than one slave is needed, then the approach of Figure 7.3 can be used. Only one slave is active at any time, determined by which Slave Select (SS) line the master activates. Note that writing SS indicates that the line is active when low; if it were active high it would simply be SS. The terminology Chip Select ($\overline{CS}$) is also used for the same role, including in this chapter. Only the slave activated by its $\overline{SS}$ input responds to the clock signal, and normally only one slave is activated at any one time. The master then communicates with the one active

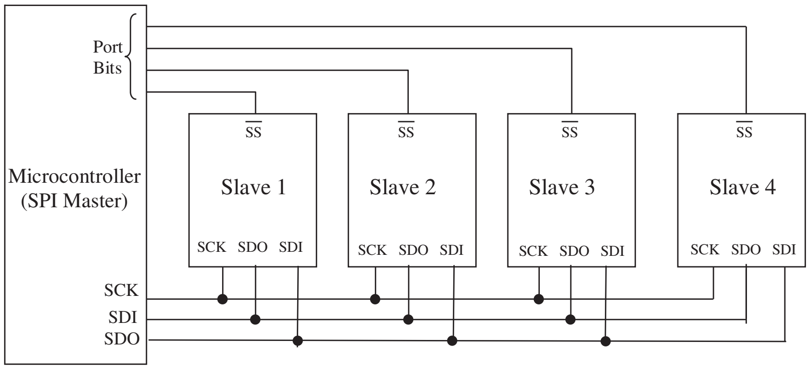


Figure 7.3:
SPI interconnections for multiple slaves

slave just as in Figure 7.2. Notice that for *n* slaves, the microcontroller needs to commit $(3 + n)$ lines. One of the advantages of serial communication, the small number of interconnections, is beginning to disappear.

7.2.2 SPI on the mbed

As the mbed diagram of Figure 2.1 shows, the mbed has two SPI ports, one appearing on pins 5, 6 and 7, and the other on pins 11, 12 and 13. The application programming interface (API) summary available for SPI master is shown in Table 7.1.

7.2.3 Setting up an mbed SPI Master

Program Example 7.1 shows a very simple setup for a SPI master. The program initializes the SPI port, choosing for it the name **ser_port**, with the pins of one of the possible ports being selected. The **format( )** function requires two variables: the first is the number of bits and the second is the mode. The mode is a feature of SPI which is illustrated in Figure 7.4, with associated codes in Table 7.2. It allows the choice of which clock edge is used to clock data into the shift register (indicated as 'Data strobe' in the diagram), and whether the clock idles high or low. For most applications the default mode, i.e. Mode 0, is acceptable. This program simply applies default values, i.e. 8 bits of data, and Mode 0 format. On the mbed, as with many SPI devices, the same pin is used for SDI if in master mode, or SDO if slave. Hence, this pin gets to be called MISO (master in, slave out). Its partner pin is MOSI.

```
/* Program Example 7.1: Sets up the mbed as SPI master, and continuously sends a single
byte
*/
#include "mbed.h"
SPI ser_port(p11, p12, p13); // mosi, miso, sclk
char switch_word ;           //word we will send

int main() {
  ser_port.format(8,0);        // Setup the SPI for 8 bit data, Mode 0 operation
  ser_port.frequency(1000000); // Clock frequency is 1MHz
  while (1){
    switch_word=0xA1;            //set up word to be transmitted
    ser_port.write(switch_word); //send switch_word
    wait_us(50);
  }
}
```

Program Example 7.1 Minimal SPI master application

Compile, download and run Program Example 7.1 on a single mbed, and observe the data (pin 11) and clock (pin 13) lines simultaneously on an oscilloscope. See how clock and data are active at the same time, and verify the clock data frequency. Check that you can read the transmitted data byte, 0xA1. Is the most significant bit (MSB) or least significant bit (LSB) sent first?

Table 7.1: mbed SPI master API summary

Function	Usage
`SPI`	Create a SPI master connected to the specified pins
`format`	Configure the data transmission mode and data length
`frequency`	Set the SPI bus clock frequency
`write`	Write to the SPI slave and return the response

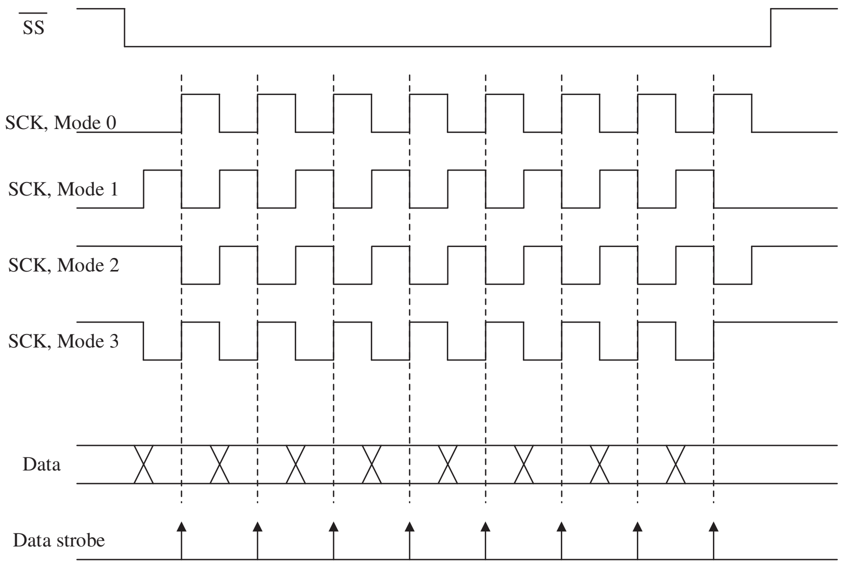


Figure 7.4:
Polarity and phase

Table 7.2: SPI modes

Mode	Polarity	Phase
0	0	0
1	0	1
2	1	0
3	1	1

Exercise 7.1

1. In Program Example 7.1, try invoking each of the SPI modes in turn. Observe both clock and data waveforms on the oscilloscope, and check how they compare to Figure 7.4.
2. Set the SPI format of Program Example 7.1 to 12 and then 16 bits, sending the words 0x8A1 and 0x8AA1, respectively (or to your choice). Check each on an oscilloscope.

7.2.4 Creating a SPI Data Link

We will now develop two programs, one master and one slave, and get two mbeds to communicate. Each will have two switches and two light-emitting diodes (LEDs); the aim will be to get the switches of the master to control the LEDs on the slave, and vice versa.

The program for the master is shown as Program Example 7.2. This is written for the circuit of Figure 7.5. It sets up the SPI port as before, and defines the switch inputs on pins 5 and 6. It declares a variable **switch_word**, the word that will be sent to the slave, and the variable **recd_val**, which is the value received from the slave. For this application the default settings

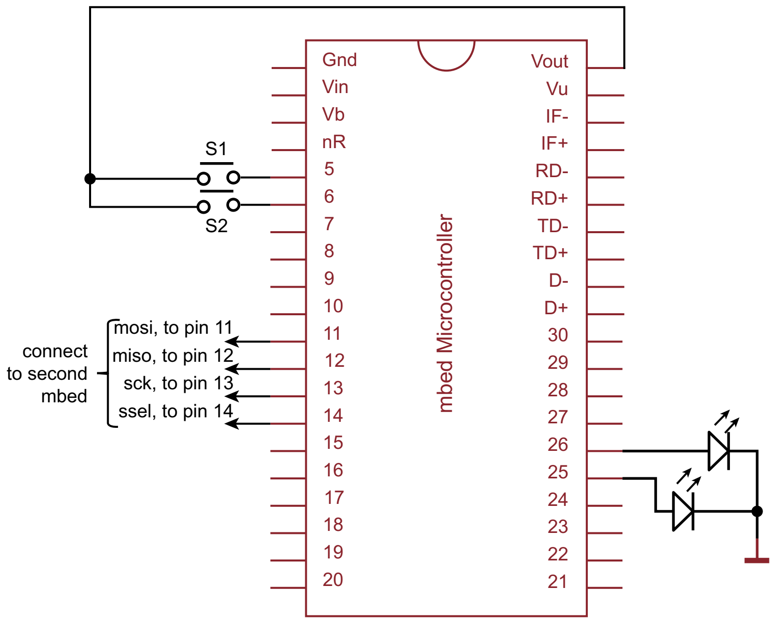


Figure 7.5:
Using SPI to link two mbeds

of the SPI port are chosen, so there is no further initialization in the program. Once in the main loop, the value of **switch_word** is established. To give a pattern to this that will be recognizable on the oscilloscope, the upper four bits are set to hexadecimal A. The two switch inputs are then tested in turn; if they are found to be high, the appropriate bit in **switch_word** is set, by ANDing with 0x01 or 0x02. The **cs** line is set low, and the command to send **switch_word** is made. The return value of this function is the received word, which is read accordingly.

```
/*Program Example 7.2. Sets the mbed up as Master, and exchanges data with a slave,
sending its own switch positions, and displaying those of the slave.
 */
#include "mbed.h"

SPI ser_port(p11, p12, p13); //mosi, miso, sclk
DigitalOut red_led(p25);     //red led
DigitalOut green_led(p26);   //green led
DigitalOut cs(p14);          //this acts as “slave select”
DigitalIn switch_ip1(p5);
DigitalIn switch_ip2(p6);
char switch_word ;     //word we will send
char recd_val;         //value return from slave

int main() {
  while (1){
    //Default settings for SPI Master chosen, no need for further configuration
    //Set up the word to be sent, by testing switch inputs
    switch_word=0xa0;       //set up a recognizable output pattern
    if (switch_ip1==1)
       switch_word=switch_word|0x01;       //OR in lsb
    if (switch_ip2==1)
       switch_word=switch_word|0x02;       //OR in next lsb
    cs = 0;                                //select slave
    recd_val=ser_port.write(switch_word);  //send switch_word and receive data
    cs = 1;
    wait(0.01);

    //set leds according to incoming word from slave
    red_led=0;              //preset both to 0
    green_led=0;
    recd_val=recd_val&0x03; //AND out unwanted bits
    if (recd_val==1)
       red_led=1;
    if (recd_val==2)
       green_led=1;
    if (recd_val==3){
       red_led=1;
       green_led=1;
    }
  }
}
```

Program Example 7.2 The mbed set up as a SPI master, with bidirectional data transfer

Table 7.3: mbed SPI slave API summary

Function	Usage
`SPISlave`	Create a SPI slave connected to the specified pins
`format`	Configure the data transmission format
`frequency`	Set the SPI bus clock frequency
`receive`	Poll the SPI to see if data has been received
`read`	Retrieve data from receive buffer as slave
`reply`	Fill the transmission buffer with the value to be written out as slave on the next received message from the master

The slave program draws upon the mbed functions shown in Table 7.3, and is shown as Program Example 7.3. It is almost the mirror image of the master program, with small but key differences. This emphasizes the very close similarity between the master and slave role in SPI. Let us check the differences. The serial port is initialized with **SPISlave**. Now four pins must be defined, the extra being the slave select input, **ssel**. As the slave will also be generating a word to be sent, and receiving one, it also declares variables **switch_word** and **recd_val**. Change these names if you would rather have something different. The slave program configures its **switch_word** just like the master. Now comes a difference. While the master initiates a transmission when it wishes, the slave must wait. The mbed library does this with the **receive( )** function. This returns 1 if data has been received, and 0 otherwise. Of course, if data has been received from the master, then data has also been sent from slave to master. If there is data, then the slave reads this and sets up the LEDs accordingly. It also sets up the next word to be sent to the master, by transferring its **switch_word** to the transmission buffer, using **reply( )**.

```
/*Program Example 7.3: Sets the mbed up as Slave, and exchanges data with a Master,
sending its own switch positions, and displaying those of the Master. as SPI slave.
*/
#include "mbed.h"
SPISlave ser_port(p11,p12,p13,p14); // mosi, miso, sclk, ssel
DigitalOut red_led(p25);            //red led
DigitalOut green_led(p26);          //green led
DigitalIn switch_ip1(p5);
DigitalIn switch_ip2(p6);
char switch_word ;                  //word we will send
char recd_val;                      //value received from master

int main() {
  //default formatting applied
  while(1) {
    //set up switch_word from switches that are pressed
    switch_word=0xa0;       //set up a recognizable output pattern
    if (switch_ip1==1)
       switch_word=switch_word|0x01;
    if (switch_ip2==1)
       switch_word=switch_word|0x02;
    if(ser_port.receive()) {   //test if data transfer has occurred
       recd_val = ser_port.read();  // Read byte from master
       ser_port.reply(switch_word); // Make this the next reply
    }
    //now set leds according to received word
    ...
    (continues as in Program Example 7.2)
    ...
  }
}
```

Program Example 7.3 The mbed set up as a SPI slave, with bidirectional data transfer

Now connect two mbeds together, carefully applying the circuit of Figure 7.5. It is simplest if each is powered individually through its own USB cable. Connections to both are identical, i.e. pin 11 goes to pin 11 and so on, so it does not matter which is chosen as master or slave. If you do not want to set up the full circuit straight away, then connect just the switches to the master, and just the LEDs to the slave, or vice versa.

Compile and download Program Example 7.2 into one mbed (which will be the master), and Program Example 7.3 into the other. Once you run the programs, you should find that pressing the switches of the master controls the LEDs of the slave, and vice versa. This is another big step forward; we are communicating data from one microcontroller to another, or from one system to another.

Exercise 7.2

Try the following, and be sure you understand the result. In each case test for data transmission from master to slave, and from slave to master.

1. Remove the pin 11 link, i.e. the data link from master to slave.
2. Remove the pin 12 link, i.e. the data link from slave to master.
3. Remove the pin 13 link, i.e. the clock.
4. Remove the pin 14 link, i.e. the chip select line.

Notice that in some cases if you disconnect a wire, but leave it dangling in the air, you might get odd intermittent behavior which changes if you touch the wire. This is an example of the impact of electromagnetic interference, where interference is being interpreted by an mbed input as a clock or data signal.

7.3 Intelligent Instrumentation and a SPI Accelerometer

7.3.1 Introducing the ADXL345 Accelerometer

Despite (or because of) its age, the SPI standard is wonderfully simple, and hence widely used. It is embedded into all sorts of electronic devices, ICs and gadgets. Given an understanding of how SPI works, we can now communicate with any of these.

With the very high level of integration found in modern ICs, it is common to find a sensor, signal conditioning, an analog-to-digital converter (ADC) and a data interface, all combined onto a single chip. Such devices are part of the new generation of *intelligent instrumentation.* Instead of just having a stand-alone sensor, as we did with the light-dependent resistor in Chapter 5, we can now have a complete measurement subsystem integrated with the sensor, with a convenient serial data output.

The ADXL345 accelerometer, made by Analog Devices, is an example of an integrated 'intelligent' sensor. Its datasheet appears as Reference 7.2. This is also an example of a *microelectromechanical system* (MEMS), in which the accelerometer mechanics are actually fabricated within the IC structure. The accelerometer output is analog in nature, and measures acceleration in three axes. The accelerometer has an internal capacitor mounted in the plane of each axis. Acceleration causes the capacitor plates to move, hence changing the output voltage in proportion to the acceleration or force. The ADXL345 accelerometer converts the analog voltage fluctuations to digital and can output these values over a SPI serial link (or I^2C; see later in this chapter).

Control of the ADXL345 is done by writing to a set of registers through the serial link. Examples of these are shown in Table 7.4. It is clear that the device goes well beyond just making direct measurements. It is possible to calibrate it, change its range and enable it to recognize certain events, for example when it is tapped or in free fall. Measurements are made in terms of g (where 1 g is the value of acceleration due to earth's gravity, i.e. 9.81 ms^{-2}).

The ADXL345 IC is extremely small and designed for surface mounting on a printed circuit board; therefore, it is ready-mounted on a 'breakout' board, as shown in Figure 7.6. It would otherwise be difficult to handle.

7.3.2 Developing a Simple ADXL345 Program

Program Example 7.4 applies the ADXL345, reading acceleration in three axes, and outputting the data to the host computer screen. The second SPI port (i.e. pins 11, 12 and 13) is used for connecting the accelerometer, applying the connections shown in Table 7.5.

Table 7.4: Selected ADXL345 registers

Address*	Name	Description
0x00	DEVID	Device ID
0x1D	THRESH_TAP	Tap threshold
0x1E/1F/20	OFSX, OFSY, OFSZ	*x*, *y*, *z*-axis offsets
0x21	DUR	Tap duration
0x2D	POWER_CTL	Power-saving features control. Device powers up in standby mode; setting bit 3 causes it to enter Measure mode
0x31	DATA_FORMAT	Data format control Bits:
		7: force a self-test by setting to 1
		6: 1 = 3-wire SPI mode; 0 = 4-wire SPI mode
		5: 0 sets interrupts active high, 1 sets them active low
		4: always 0
		3: 0 = output is 10-bit always; 1 = output depends on range setting
		2: 1 = left align result; 0 = right align result
		1-0: 00 = ± 2 *g*; 01 = ± 4 *g*; 10 = ± 8 *g*; 11 = ± 2 *g*
0x33:0x32	DATAX1:DATAX0	*x*-axis data, formatted according to DATA_FORMAT, in 2's complement.
0x35:0x34	DATAY1:DATAY0	*y*-axis data, as above
0x37:0x36	DATAZ1:DATAZ0	*z*-axis data, as above

*In any data transfer the register address is sent first, and formed: bit 7 = R/$\overline{\text{W}}$ (1 for read, 0 for write); bit 6:1 for multiple bytes, 0 for single; bits 5-0: the lower 6 bits found in the Address column.

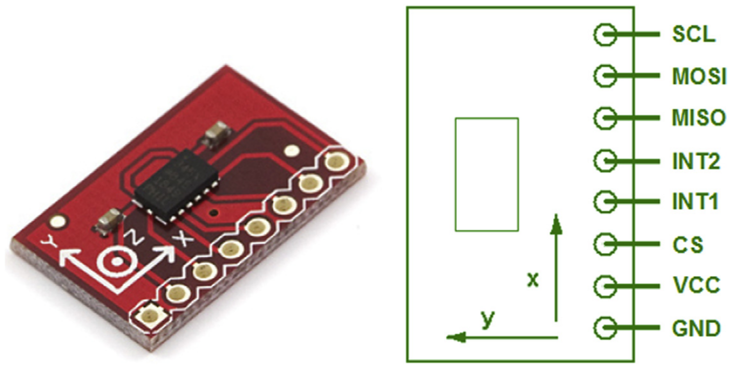


Figure 7.6:
The ADXL345 accelerometer on breakout board. *(Image reproduced with permission of SparkFun Electronics)*

Table 7.5: ADXL345 pin connections to mbed

ADXL345 signal name	mbed pin
Vcc	Vout
Gnd	Gnd
SCL	13
MOSI	11
MISO	12
$\overline{\text{CS}}$	14

C code feature

The program initializes a master SPI port, which we have chosen to call **acc**, and sets up the USB link to the host computer. It further declares two arrays; one is a buffer that will hold data read direct from the accelerometer's registers, two for each axis. The second array applies the **int16_t** specifier. This is from the C standard library **stdint**; it tells the compiler that exactly 16-bit (signed) integer type data is being declared. This array will hold the full accelerometer axis values, combined from the two bytes received from the registers.

The main function initializes the SPI port in a manner with which we are familiar. It then loads two of the accelerometer registers, writing the address first, followed by the data byte. It should be possible to work out what is being written by looking at either the datasheet or Table 7.4. A continuous **while** loop is then initiated. Following a pause, a multi-byte read is set up, using an address word formed from information shown in Table 7.4. This fills the **buffer** array. The **data** array is then populated, concatenating (i.e. combining to form a single number) pairs of bytes from the **buffer** array. These values are then scaled to actual *g* values, using the conversion factor from the datasheet, of 0.004 *g* per unit. Results are then displayed on-screen.

```
/*Program Example 7.4: Reads values from accelerometer through SPI, and outputs
continuously to terminal screen.
*/

#include "mbed.h"
SPI acc(p11,p12,p13);        // set up SPI interface on pins 11,12,13
DigitalOut cs(p14);          // use pin 14 as chip select
Serial pc(USBTX, USBRX);     // set up USB interface to host terminal
char buffer[6];              //raw data array type char
int16_t data[3];             // 16-bit twos-complement integer data
float x, y, z;               // floating point data, to be displayed on-screen

int main() {
  cs=1;                        //initially ADXL345 is not activated
  acc.format(8,3);             // 8 bit data, Mode 3
  acc.frequency(2000000);      // 2MHz clock rate
  cs=0;                        //select the device
  acc.write(0x31);             // data format register
  acc.write(0x0B);             // format +/-16g, 0.004g/LSB
  cs=1;                        //end of transmission
  cs=0;                        //start a new transmission
  acc.write(0x2D);             // power ctrl register
  acc.write(0x08);             // measure mode
  cs=1;                        //end of transmission
  while (1) {                  // infinite loop
    wait(0.2);
    cs=0;                      //start a transmission
    acc.write(0x80|0x40|0x32);  // RW bit high, MB bit high, plus address
    for (int i = 0;i<=5;i++) {
      buffer[i]=acc.write(0x00);         // read back 6 data bytes
    }
    cs=1;                                //end of transmission
    data[0] = buffer[1]<<8 | buffer[0];  //combine MSB and LSB
    data[1] = buffer[3]<<8 | buffer[2];
    data[2] = buffer[5]<<8 | buffer[4];
    x=0.004*data[0]; y=0.004*data[1]; z=0.004*data[2]; // convert to float,
                                                        //actual g value
    pc.printf("x = %+1.2fg\t y = %+1.2fg\t z = %+1.2fg\n\r", x, y,z); //print
  }
}
```

Program Example 7.4 Accelerometer continuously outputs three-axis data to terminal screen

Carefully make the connections of Table 7.5, and compile, download and run the code on your mbed. Open a Tera Term screen (as described in Appendix E) on your computer. Accelerometer readings should be displayed to the screen. You will see that when the accelerometer is flat on a table the *z*-axis should read approximately 1 *g*, with the *x*- and *y*-axes approximately 0 *g*. As you rotate and move the device the *g* readings will change. If the accelerometer is shaken or displaced at a quick rate, *g* values in excess of 1 *g* can be observed. Note that there are some inaccuracies in the accelerometer data, which can be reduced in a real application by developing configuration/calibration routines and data averaging functions.

Although we have not used it here, the mbed site provides an ADXL345 library, located in the Cookbook. This simplifies use of the accelerometer, and saves worrying about register addresses and bit values.

Exercise 7.3

Rewrite Program Example 7.4 using the library functions available on the mbed site Cookbook.

7.4 Evaluating SPI

The SPI standard is extremely effective. The electronic hardware is simple and therefore cheap, and data can be transferred rapidly. It does have its disadvantages, however. There is no acknowledgment from the receiver, so in a simple system the master cannot be sure that data has been received. Also there is no addressing. In a system where there are multiple slaves, a separate $\overline{\mathrm{SS}}$ line must be run to each slave, as shown in Figure 7.3. Therefore, we begin to lose the advantage that should come from serial communications, i.e. a limited number of interconnect lines. Finally, there is no error checking. Suppose some electromagnetic interference was experienced in a long data link; the data or clock would be corrupted, but the system would have no way of detecting this or correcting for it. You may have experienced this in a small way in Exercise 7.2. Overall, SPI could be graded as: simple, convenient and low cost, but not appropriate for complex or high-reliability systems.

7.5 The Inter-Integrated Circuit Bus

7.5.1 Introducing the I^2C Bus

The inter-integrated circuit (I^2C) standard was developed by Philips to resolve some of the perceived weaknesses of SPI and its equivalents. As its name suggests, it is also intended for interconnection over short distances and generally within a piece of equipment. It uses only

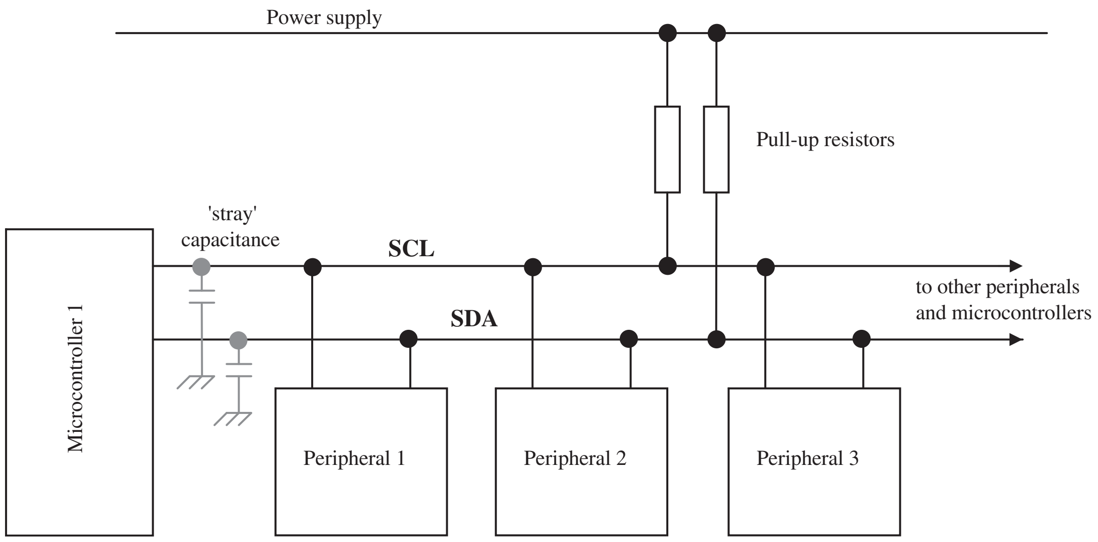


Figure 7.7:
An I^2C-based system

two interconnect wires, but many devices are connected to the bus. These lines are called SCL (serial clock) and SDA (serial data). All devices on the bus are connected to these two lines, as shown in Figure 7.7. The SDA line is bidirectional, so data can travel in either direction, but only one direction at any one time. In the jargon this is called *half duplex*. Like SPI, it is a synchronous serial standard.

One of the interesting features of I^2C, and one that makes it versatile, is that any node connected to it can only pull down the SCL or SDA line to Logic 0; it cannot force the line up to Logic 1. This role is played by a single pull-up resistor connected to each line. When a node pulls a line to Logic 0 and then releases it, it is returned to Logic 1 by the action of the pull-up resistor. There is, however, capacitance associated with the line. Although this is labeled 'stray' capacitance in Figure 7.7, it is in reality mainly unavoidable capacitance that exists in the semiconductor structures connected to the line. This capacitance is thus higher if there are many nodes connected, and lower otherwise. The higher the capacitance and/or pull-up resistance, the longer the rise time of the logic transition from 0 to 1. The I^2C standard requires that the rise time of a signal on SCL or SDA must be less than 1000 ns. Given a known bus setup, it is possible to perform reasonably precise calculations of pull-up resistor required (see Reference 1.1), particularly if you need to minimize power consumption. For simple applications default pull-up resistor values, in the range 2.2−4.7 kΩ, are quite acceptable.

The I^2C protocol has been through several revisions, which have dramatically increased the possible speeds and reflect technological changes, for example in reduced minimum operating voltages. The original version, standard mode, allowed data rates up to 100 kbit/s. Version 1.0, in 1992, increased the maximum data rate to 400 kbit/s. This latter is very well established and still probably accounts for most I^2C implementation. Version 2.0, in 1998, increased the possible bit rate to 3.4 Mbit/s. Version 3 is defined in a surprisingly readable manner in Reference 7.3, and forms the basis of the following description.

Nodes on an I^2C bus can act as master or slave. The master initiates and terminates transfers, and generates the clock. The slave is any device addressed by the master. A system may have more than one master, although only one may be active at any time. Therefore more than one microcontroller could be connected to the bus, and they can claim the master role at different times, when needed. An arbitration process is defined if more than one master attempts to control the bus.

A data transfer is made up of the master signaling a *start condition*, followed by one or two bytes containing address and control information. The start condition (Figure 7.8a) is defined by a high to low transition of SDA when SCL is high. All subsequent data transmission follows the pattern of Figure 7.8b. One clock pulse is generated for each data bit, and data may only change when the clock is low. The byte following the start condition is made up of seven address bits and one data direction bit, as shown in Figure 7.8c. Each slave has a predefined device address; the slaves are therefore responsible for monitoring the bus and

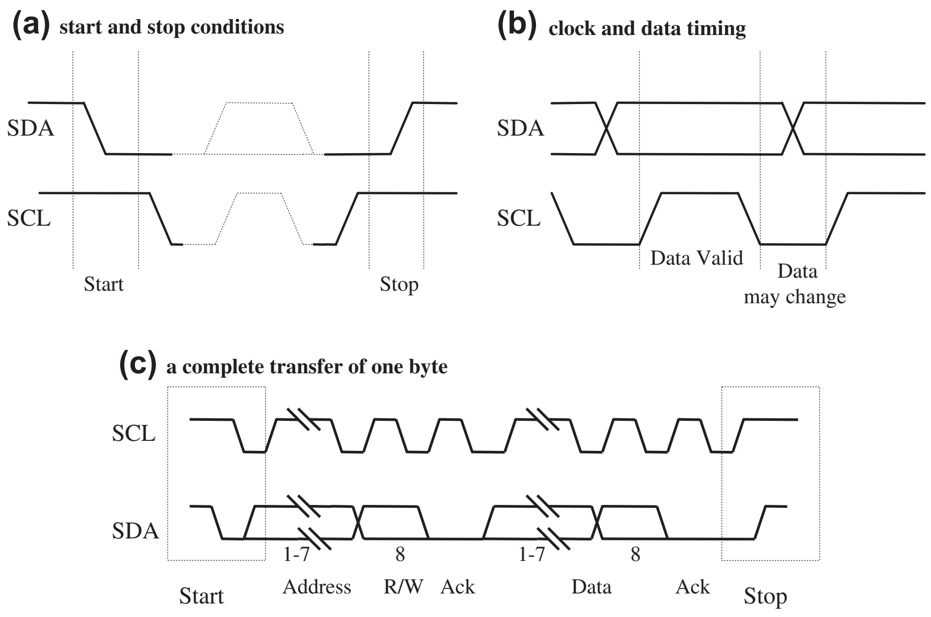

(a) start and stop conditions
SDA
SCL
Start
Stop
(b) clock and data timing
SDA
SCL
Data Valid
Data may change
(c) a complete transfer of one byte
SCL
SDA
1-7
8
1-7
8
Start
Address
R/W
Ack
Data
Ack
Stop

Figure 7.8:
I^{2}C data transfer

responding only to commands associated with their own address. A slave device that recognizes its address will then be readied either to receive data or to transmit it onto the bus. A 10-bit addressing mode is also available.

All data transferred is in units of one byte, with no limit on the number of bytes transferred in one message. Each byte must be followed by a 1-bit acknowledgment from the receiver, during which time the transmitter relinquishes SDA control. A low to high transition of SDA while SCL is high defines a *stop* condition. Figure 7.8c illustrates the complete transfer of a single byte.

7.5.2 I^{2}C on the mbed

Figure 2.1 shows us that the mbed offers two I^{2}C ports, on pins 9 and 10, or 27 and 28. Their use follows the pattern of other mbed peripherals, with available functions shown in Tables 7.6 and 7.7.

7.5.3 Setting up an I^{2}C Data Link

We will now replicate the action of Section 7.2.4, but using I^{2}C as the communication link, rather than SPI. We will use the I^{2}C port on pins 9 and 10. Program Example 7.5, the master,

Table 7.6: mbed I^2C master API summary

Function	Usage
`I2C`	Create an I^2C master interface, connected to the specified pins
`frequency`	Set the frequency of the I^2C interface
`read`	Read from an I^2C slave
`write`	Write to an I^2C slave
`start`	Creates a start condition on the I^2C bus
`stop`	Creates a stop condition on the I^2C bus

Table 7.7: mbed I^2C slave API summary

Function	Usage
`I2CSlave`	Create an I^2C slave interface, connected to the specified pins
`frequency`	Set the frequency of the I^2C interface
`receive`	Checks to see if this I^2C slave has been addressed
`read`	Read from an I^2C master
`write`	Write to an I^2C master
`address`	Sets the I^2C slave address
`stop`	Reset the I^2C slave back into the known ready receiving state

follows exactly the pattern of Program Example 7.2, except that SPI-related sections are replaced by those that relate to I^2C. Early in the program an I^2C serial port is configured using the mbed utility **I2C**. The name **i2c_port** is chosen for the port name, and linked to the port on pins 9 and 10. An arbitrary slave address is chosen, 0x52. Following determination of the variable **switch_word**, the I^2C transmission can be seen. This is created from the separate components of a single-byte I^2C transmission, i.e. start – send address – send data – stop, as allowed by the mbed functions. A way of grouping these together is presented later in the chapter. Further down the program we see a request for a byte of data from the slave, with similar message structure. Now the slave address is ORed with 0x01, which sets the $R/\overline{W}$ bit in the address word to indicate Read. The received word is then interpreted in order to set the LEDs, just as we did in the SPI program earlier.

```
/*Program Example 7.5: I2C Master, transfers switch state to second mbed acting as
slave, and displays state of slave's switches on its leds.
*/
#include "mbed.h"
I2C i2c_port(p9, p10);     //Configure a serial port, pins 9 and 10 are sda, scl
DigitalOut red_led(p25);   //red led
DigitalOut green_led(p26); //green led
DigitalIn switch_ip1(p5);  //input switch
DigitalIn switch_ip2(p6);

char switch_word ;      //word we will send
char recd_val;          //value received from slave
const int addr = 0x52;  //the I2C slave address, an arbitrary even number

int main() {
  while(1) {
    switch_word=0xa0;                 //set up a recognizable output pattern
    if (switch_ip1==1)
      switch_word=switch_word|0x01;    //OR in lsb
    if (switch_ip2==1)
      switch_word=switch_word|0x02;    //OR in next lsb
    //send a single byte of data, in correct I2C package
    i2c_port.start();                //force a start condition
    i2c_port.write(addr);            //send the address
    i2c_port.write(switch_word);     //send one byte of data, ie switch_word
    i2c_port.stop();                 //force a stop condition
    wait(0.002);
    //receive a single byte of data, in correct I2C package
    i2c_port.start();
    i2c_port.write(addr|0x01);       //send address, with R/W bit set to Read
    recd_val=i2c_port.read(addr);    //Read and save the received byte
    i2c_port.stop();                 //force a stop condition
    //set leds according to word received from slave
    red_led=0;                       //preset both to 0
    …
    (continues as in Program Example 7.2)
    …
  }
}
```

Program Example 7.5 I^{2}C data link master

The slave program is shown in Program Example 7.6, and is similar to Program Example 7.3, with SPI features replaced by I^{2}C. As in SPI, the I^{2}C slave just responds to calls from the master. The slave port is defined with the mbed utility **I2Cslave**, with **slave** chosen as the port name. Just within the **main** function the slave address is defined, importantly the same 0x52 as we saw in the master program. As before, the **switch_word** value is set up from the state of the switches; this is then saved using the **write** function, in readiness for a request from the master. The **receive( )** function is used to test whether an I^{2}C transmission has been received. This returns a 0 if the slave has not been addressed, a 1 if it has been addressed to read, and a 3 if addressed to write. If a read has been initiated, then the value already stored is automatically sent. If the value is 3, then the program stores the received value and sets up the LEDs on the master accordingly.

```
/*Program Example 7.6: I2C Slave, when called transfers switch state to mbed acting as
Master, and displays state of Master's switches on its leds.
*/
#include <mbed.h>
I2CSlave slave(p9, p10);      //Configure I2C slave
DigitalOut red_led(p25);      //red led
DigitalOut green_led(p26);    //green led
DigitalIn switch_ip1(p5);
DigitalIn switch_ip2(p6);
char switch_word ;            //word we will send
char recd_val;                //value received from master

int main() {
  slave.address(0x52);
  while (1) {
    //set up switch_word from switches that are pressed
    switch_word=0xa0; //set up a recognizable output pattern
    if (switch_ip1==1)
      switch_word=switch_word|0x01;
    if (switch_ip2==1)
      switch_word=switch_word|0x02;
    slave.write(switch_word); //load up word to send
    //test for I2C, and act accordingly
    int i = slave.receive();
    if (i == 3){      //slave is addressed, Master will write
      recd_val= slave.read();
          //now set leds according to received word
    ...
   (continues as in Program Example 7.2)
    ...
  }
}
```

Program Example 7.6 I^{2}C data link slave

Connect two mbeds together with an I^{2}C link, applying the circuit diagram of Figure 7.9. This is very similar to Figure 7.5, except that the SPI connection is removed, and replaced by the I^{2}C connection. It is essential to include the pull-up resistors; values of 4.7 kΩ are shown, but they can be anywhere in the range 2.2−4.7 kΩ. Note that each mbed should have two switches and two LEDs connected, but there should just be one pair of pull-up resistors between them. Compile and download Program Example 7.5 into either mbed, and Example 7.6 into the other. You should find that the switches of one mbed control the LEDs of the other, and vice versa.

Monitor SCL and SDA lines on an oscilloscope. This may require careful oscilloscope triggering. With appropriate setting of the oscilloscope time base you should be able to see the two messages being sent between the mbeds. Identify as many features of I^{2}C as you can, including the idle high condition, the start and stop conditions, and the address byte with the R/$\bar{W}$ bit embedded.

Exercise 7.4

In Program Example 7.6, replace the line `if (i == 3)` with `if (i == 4)`; in other words, the condition cannot be satisfied. Compile, download and run. Notice that the slave still

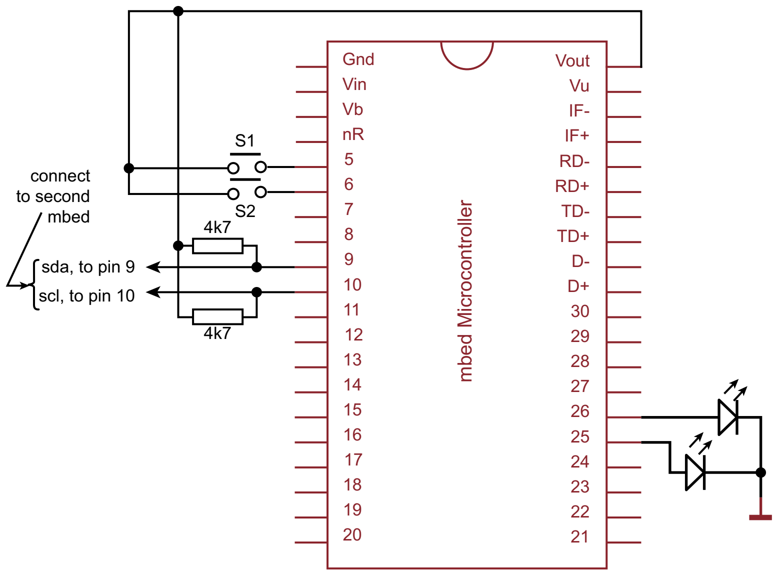


Figure 7.9:
Linking two mbeds with I^2C

continues to write to the master, but is now unable to respond to the master's message. Why is this?

■

7.6 Communicating with an I^2C Temperature Sensor

Just as we did with the SPI port and the accelerometer, we can use the mbed I^2C port to communicate with a very wide range of peripheral devices, including many intelligent sensors. The Texas Instruments TMP102 temperature sensor (Reference 7.4) has an I^2C data link. This is similar to the accelerometer introduced in Section 7.3, in that an analog sensing device is integrated with an ADC and a serial port, producing an ideal and easy-to-use system element. Note from the datasheet that the TMP102 actually makes use of the system management bus (SMbus). This was defined by Intel in 1995, and is based on I^2C. In simple applications the two standards can be mixed; for more advanced applications it is worth checking the small differences between them.

The TMP102 is a tiny device, as required of a temperature sensor. Like the accelerometer, it is used mounted on a small breakout board, seen in Figure 7.10a. It has six possible connections, shown in Table 7.8. Connections on connector JP1 are the essential ones of power supply and I^2C. On connector JP2 the sensor has an address pin, ADD0; this is used to select the address

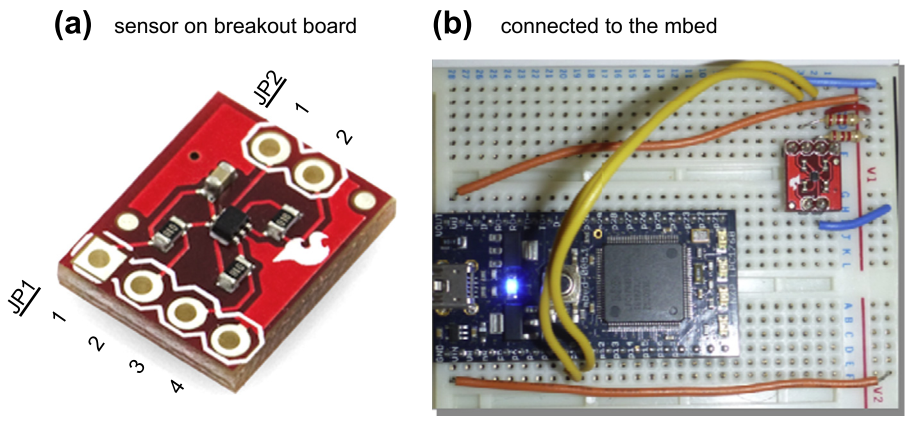


Figure 7.10:
The TMP102 temperature sensor *(Image reproduced with permission of SparkFun Electronics)*

of the device, as seen in the table. This allows four different address options, hence four of the same sensor can be used on the same I^2C bus.

Program Example 7.7 can be applied to link the mbed to the sensor. It defines an I^2C port on pins 9 and 10, and names this **tempsensor**. The serial link to be used to communicate with the PC is set up. As sensor pin ADD0 is tied to ground, Table 7.8 shows that the sensor address will be 0x90; this is defined in the program, with name **addr**. Two small arrays are also defined, one to hold the sensor configuration data, and the other to hold the raw data read from the sensor. A further variable **temp** will hold the scaled decimal equivalent of the reading.

Table 7.8: Connecting the TMP102 sensor to the mbed

Signal	TMP102 pin	mbed pin	Notes	
Vcc (3.3 V)	JP1: 1	40		
SDA	JP1: 2	9	2.2 kΩ pull-up to 3.3 V	
SCL	JP1: 3	10	2.2 kΩ pull-up to 3.3 V	
Gnd (0 V)	JP1: 4	1		
Alert	JP2: 1	1		
			Connect to	Slave address
ADD0	JP2: 2	1	0V	0x90
			Vcc	0x91
			SDA	0x92
			SCL	0x93

The configuration options can be found from the TMP102 datasheet. To set the configuration register we first need to send a data byte of 0x01 to specify that the pointer register is set to 'Configuration Register'. This is followed by two configuration bytes, 0x60 and 0xA0. These select a simple configuration setting, initializing the sensor to normal mode operation. These values are sent at the start of the **main** function. Note the format of the write command used here, which is able to send multi-byte messages. This is different from the approach used in Program Example 7.5, where only a single byte was sent in any one message. Using the I^2C **write( )** function, we need to specify the device address and the array of data, followed by the number of bytes to send.

The sensor will now operate and acquire temperature data, so we simply need to read the data register. To do this we need first to set the pointer register value to 0x00. In this command we have only sent one data byte to set the pointer register. The program now starts an infinite loop to read continuously the 2-byte temperature data. This data is then converted from a 16-bit reading to an actual temperature value. The conversion required (as specified by the datasheet) is to shift the data right by 4 bits (it is actually only 12-bit data held in two 8-bit registers) and to multiply by the specified conversion factor, 0.0625 degrees C per LSB. The value is then displayed on the PC screen.

```
/*Program Example 7.7: Mbed communicates with TMP102 temperature sensor, and scales and
displays readings to screen.
*/
#include "mbed.h"
I2C tempsensor(p9, p10);   //sda, scl
Serial pc(USBTX, USBRX);   //tx, rx
const int addr = 0x90;
char config_t[3];
char temp_read[2];
float temp;

int main() {
  config_t[0] = 0x01;                        //set pointer reg to 'config register'
  config_t[1] = 0x60;                        // config data byte1
  config_t[2] = 0xA0;                        // config data byte2
  tempsensor.write(addr, config_t, 3);
  config_t[0] = 0x00;                        //set pointer reg to 'data register'
  tempsensor.write(addr, config_t, 1);       //send to pointer 'read temp'
  while(1) {
    wait(1);
    tempsensor.read(addr, temp_read, 2);     //read the two-byte temp data
    temp = 0.0625 * (((temp_read[0] << 8) + temp_read[1]) >> 4); //convert data
    pc.printf("Temp = %.2f degC\n\r", temp);
  }
}
```

Program Example 7.7 Communicating by I^2C with the TMP102 temperature sensor

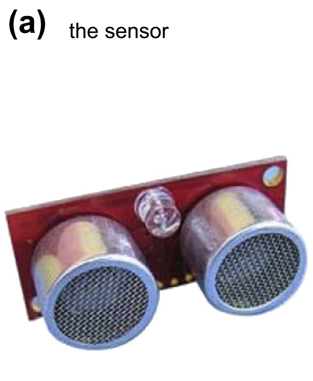
Connect the sensor according to the information in Table 7.8, leading to a circuit that looks something like Figure 7.10b. Once again, the SDA and SCL lines each need to be pulled up to 3.3 V through a resistor, of value between 2.2 and 4.7 kΩ. Compile, download and run the code of Program Example 7.7. Test that the displayed temperature increases when you press your finger against the sensor. You can try placing the sensor on something warm, for example a radiator, in order to check that it responds to temperature changes. If you have a calibrated temperature sensor, try to compare readings from both sources.

Exercise 7.5

Rewrite Program Example 7.5 using the I^{2}C master **read( )** and **write( )** functions, as seen in Program Example 7.7. Compile, download and test that it works as expected.

7.7 Using the SRF08 Ultrasonic Range Finder

The SRF08 ultrasonic range finder, as shown in Figure 7.11, can be used to measure the distance between the sensor and an acoustically reflective surface or object in front of it. It makes the measurement by transmitting a pulse of ultrasound from one of its transducers, and then measuring the time for an echo to return to the other. If there is no echo it times out. The distance to the reflecting object is proportional to the time taken for the echo to return. Given knowledge of the speed of sound in air, the actual distance can be calculated. The SRF08 has



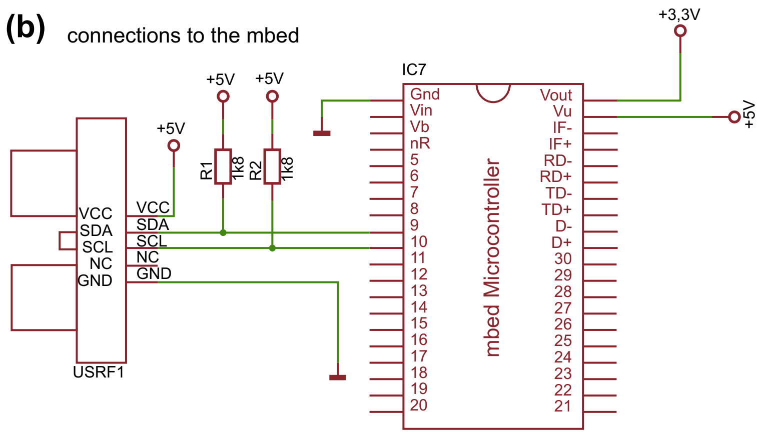


Figure 7.11:
The SRF08 ultrasonic range finder. *(Image reproduced with permission of SparkFun Electronics)*

an I^2C interface. Data is readily available for it, for example Reference 7.5, although at the time of writing not as a formalized document.

The SRF08 can be connected to the mbed as shown in Figure 7.11b. It must be powered from 5 V, with I^2C pull-up resistors connected to that voltage. The mbed remains powered from 3.3 V, and is able to tolerate the higher voltage being presented at its I^2C pins.

Having worked through the preceding programs, it should be easy to grasp how Program Example 7.8 works. Note the following information, taken from the device data:

- The SRF08 I^2C address is 0xE0.
- The pointer value for the command register is 0x00.
- A data value of 0x51 to the command register initializes the range finder to operate and return data in cm.
- A pointer value of 0x02 prepares for 16-bit data (i.e. two bytes) to be read.

```
/*Program Example 7.8: Configures and takes readings from the SRF08 ultrasonic range
finder, and displays them on screen.
*/
#include "mbed.h"
I2C rangefinder(p9, p10); //sda, scl
Serial pc(USBTX, USBRX); //tx, rx
const int addr = 0xE0;
char config_r[2];
char range_read[2];
float range;

int main() {
  while (1) {
    config_r[0] = 0x00;                    //set pointer reg to 'cmd register'
    config_r[1] = 0x51;                    //initialize, result in cm
    rangefinder.write(addr, config_r, 2);
    wait(0.07);
    config_r[0] = 0x02;                    //set pointer reg to 'data register'
    rangefinder.write(addr, config_r, 1);  //send to pointer 'read range'
    rangefinder.read(addr, range_read, 2); //read the two-byte range data
    range = ((range_read[0] << 8) + range_read[1]);
    pc.printf("Range = %.2f cm\n\r", range); //print range on screen
    wait(0.05);
  }
}
```

Program Example 7.8 Communicating by I^2C with the SRF08 range finder

Connect the circuit of Figure 7.11b. Compile Program Example 7.8 and download to the mbed. Verify correct operation by placing the range finder a known distance from a hard, flat surface. Then explore its ability to detect irregular surfaces, narrow objects (e.g. a broom handle) and distant objects.

7.8 Evaluating I^{2}C

As we have seen, the I^{2}C protocol is well established and versatile. Like SPI, it is widely applied to short distance data communication. However, it goes well beyond SPI in its ability to set up more complex networks, and to add and subtract nodes with comparative ease. Although we have not really explored it here, it provides for a much more reliable system. If an addressed device does not send an acknowledgment, the master can act upon that fault. Does this mean that I^{2}C is going to meet all our needs for serial communication, in any application? The answer is a clear No, for at least two reasons. One is that the bandwidth is comparatively limited, even in the faster versions of I^{2}C. The second is the security of the data. While fine in, say, a domestic appliance, I^{2}C is still susceptible to interference and does not check for errors. Therefore, one would be unlikely to consider using it in a medical, motor vehicle or other high-reliability application.

7.9 Asynchronous Serial Data Communication

The synchronous serial communication protocols described so far this chapter, in the form of SPI and I^{2}C, are extremely useful ways of moving data around. The question remains, however: do we really need to send that clock signal wherever the data goes? Although it allows an easy way of synchronizing the data, it does have these disadvantages:

- An extra (clock) line needs to go to every data node.
- The bandwidth needed for the clock is always twice the bandwidth needed for the data; therefore, it is the demands of the clock that limit the overall data rate.
- Over long distances, clock and data themselves could lose synchronization.

7.9.1 Introducing Asynchronous Serial Data

For the reasons just stated, several serial standards have been developed that do not require a clock signal to be sent with the data. This is generally called *asynchronous* serial communication. It is now up to the receiver to extract all timing information directly from the signal itself. This has the effect of laying new and different demands on the signal, and making transmitter and receiver nodes somewhat more complex than comparable synchronous nodes.

A common approach to achieving asynchronous communication is based on this:

- Data rate is predetermined – both transmitter and receiver are preset to recognize the same data rate. Hence, each node needs an accurate and stable clock source, from which the data rate can be generated. Small variations from the theoretical value can, however, be accommodated.

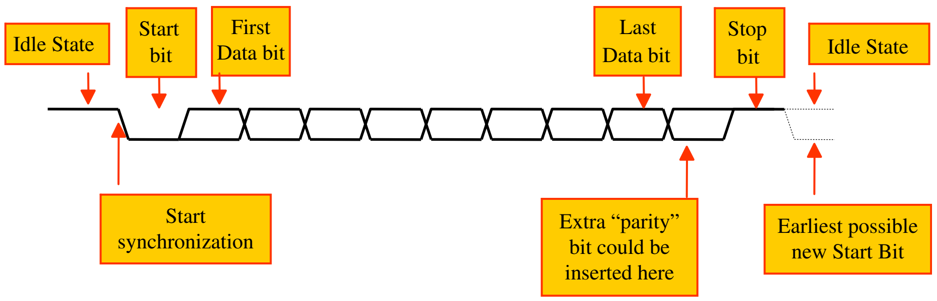


Figure 7.12:
A common asynchronous serial data format

- Each byte or word is *framed* with a Start and Stop bit. These allow synchronization to be initiated before the data starts to flow.

An asynchronous data format, of the sort used by such standards as RS-232, is shown in Figure 7.12. There is now only one data line. This idles in a predetermined state, in this example at Logic 1. The start of a data word is initiated by a *Start* bit, which has polarity opposite to that of the idle state. The leading edge of the Start bit is used for synchronization. Eight data bits are then clocked in. A ninth bit, for parity checking, is also sometimes used. The line then returns to the idle state, which forms a Stop bit. A new word of data can be sent immediately, following the completion of a single Stop bit, or the line may remain in the idle state until it is needed again.

An asynchronous serial port integrated into a microcontroller or peripheral device is generally called a UART, standing for *universal asynchronous receiver/transmitter.* In its simplest form, a UART has one connection for transmitted data, usually called TX, and another for received data, called RX. The port should sense when a start bit has been initiated, and automatically clock in and store the new word. It can initiate a transmission at any time. The data rate that receiver and transmitter will operate at must be predetermined; this is specified by its *baud rate*. For the present purposes baud rate can be viewed as being equivalent to bit rate; for more advanced applications one should check the distinctions between these two terms.

7.9.2 Applying Asynchronous Communication on the mbed

If we look back at Figure 2.3. we see that the LPC1768 has four UARTs. Three of these appear on the pinout of the mbed, simply labeled 'Serial', on pins 9 and 10, 13 and 14, and 27 and 28. Their not insignificant API summary is given in Table 7.9.

We will now repeat what we have already done with SPI and I^2C, which is to connect two mbeds to demonstrate a serial link, but this time an asynchronous one. You can view Figure 7.13, the build we will use, as a variation on Figures 7.5 and 7.9; notice that it is slightly simpler than either of these. We will apply Program Example 7.9. It will be the same

Table 7.9: Serial (asynchronous) API summary

Function	Usage
`Serial`	Create a serial port, connected to the specified transmit and receive pins
`baud`	Set the baud rate of the serial port
`format`	Set the transmission format used by the serial port
`putc`	Write a character
`getc`	Read a character
`printf`	Write a formatted string
`scanf`	Read a formatted string
`readable`	Determine if there is a character available to read
`writeable`	Determine if there is space available to write a character
`attach`	Attach a function to call whenever a serial interrupt is generated

program we load into both mbeds. It follows a similar pattern to Program Example 7.2, but replaces all SPI code with UART code. The code itself applies a number of functions from Table 7.9, and — reading the comments — should not be too difficult to follow.

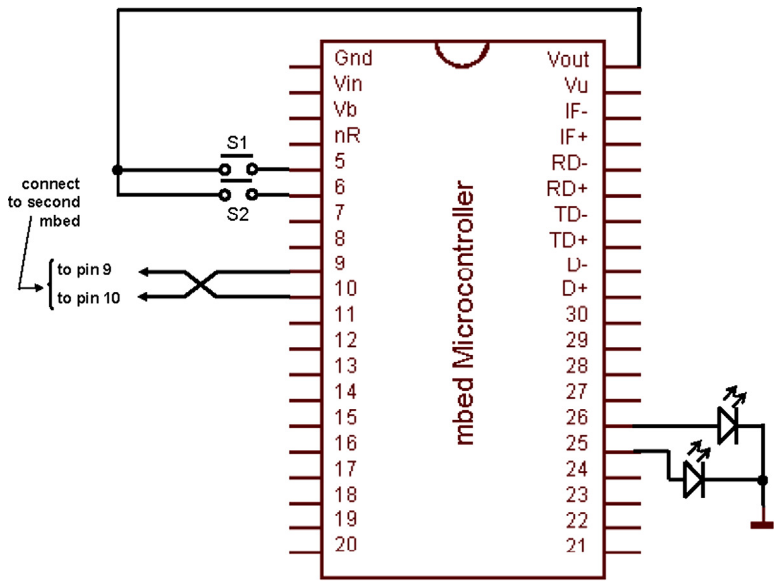


Figure 7.13:
Linking two mbed UARTs

```
/*Program Example 7.9: Sets the mbed up for async communication, and exchanges data with
a similar node, sending its own switch positions, and displaying those of the other.
 */
#include "mbed.h"
Serial async_port(p9, p10);          //set up TX and RX on pins 9 and 10
DigitalOut red_led(p25);             //red led
DigitalOut green_led(p26);           //green led
DigitalOut strobe(p7);               //a strobe to trigger the scope
DigitalIn switch_ip1(p5);
DigitalIn switch_ip2(p6);
char switch_word ;                   //the word we will send
char recd_val;                       //the received value

int main() {
  async_port.baud(9600);             //set baud rate to 9600 (ie default)
  //accept default format, of 8 bits, no parity
  while (1){
    //Set up the word to be sent, by testing switch inputs
    switch_word=0xa0;                //set up a recognizable output pattern
    if (switch_ip1==1)
      switch_word=switch_word|0x01;  //OR in lsb
    if (switch_ip2==1)
      switch_word=switch_word|0x02;  //OR in next lsb
    strobe =1;                       //short strobe pulse
    wait_us(10);
    strobe=0;
    async_port.putc(switch_word);    //transmit switch_word
    if (async_port.readable()==1)    //is there a character to be read?
      recd_val=async_port.getc();    //if yes, then read it
    ...
    (continues as in Program Example 7.2)
    ...
  }
}
```

Program Example 7.9 Bidirectional data transfer between two mbed UARTs

Connect two mbeds as shown in Figure 7.13, and compile and download Program Example 7.9 into each. You should find that the switches from one mbed control the LEDs from the other, and vice versa.

Exercise 7.6

On an oscilloscope, observe the data waveform of one of the TX lines. It helps to trigger the oscilloscope from the strobe pulse (pin 7). See how the pattern changes as the switches are pressed.

1. What is the time duration of each data bit? How does this relate to the baud rate? Change the baud rate and measure the new duration.

2. Is the data byte transmitted MSB first or LSB? How does this compare to the serial protocols seen earlier in this chapter?
3. What is the effect of removing one of the data links in Figure 7.13?

■

7.9.3 Applying Synchronous Communication with the Host Computer

While the mbed has three UARTs connecting to the external pins, the LPC1768 has a fourth. This is reserved for communication back to the USB link, and can be seen in the mbed block diagram in Figure 2.2. This UART acts just like any of the others, in terms of its use of the API (Table 7.8). We have been using it already, for the first time in Program Example 5.4. As we saw there, the mbed compiler will recognize **pc**, **USBTX** and **USBRX** as identifiers to set up this connection, as in the line:

```
Serial pc(USBTX, USBRX);
```

With **pc** thus created, the API member functions can be exploited. The requirement to set up the host computer correctly in order to communicate with this link was outlined in Section 5.3.

7.10 Mini-Project: Multi-Node I^{2}C Bus

Design a simple circuit which has a temperature sensor, range finder and mbed connected to the same I^{2}C bus. Merge Program Examples 7.8 and 7.9, so that measurements from each sensor are displayed in turn on the screen. Verify that the I^{2}C protocol works as expected.

Chapter Review

- Serial data links provide a ready means of communication between microcontroller and peripherals, and/or between microcontrollers.
- SPI is a simple synchronous standard, which is still very widely applied. The mbed has two SPI ports and a supporting library.
- While a very useful standard, SPI has certain very clear limitations, relating to a lack of flexibility and robustness.
- The I^{2}C protocol is a more sophisticated serial alternative to SPI; it runs on a two-wire bus, and includes addressing and acknowledgment.
- I^{2}C is a flexible and versatile standard. Devices can be readily added to or removed from an existing bus, multi-master configurations are possible, and a master can detect if a slave fails to respond, and can take appropriate action. Nevertheless, I^{2}C has limitations which mean that it cannot be used for high-reliability applications.
- A very wide range of peripheral devices is available, including intelligent sensors, which communicate through SPI and I^{2}C.

- A useful asynchronous alternative to I^2C and SPI is provided by the UART. The mbed has four of these, one of which provides a communication link back to the host computer.

Quiz

1. What do the abbreviations SPI, I^2C and UART stand for?
2. Draw up a table comparing the advantages and disadvantages of using SPI versus I^2C for serial communications.
3. What are the limitations for the number of devices that can be connected to a single SPI, I^2C or UART bus?
4. A SPI link is running with a 500 kHz clock. How long does it take for a single message containing one data byte to be transmitted?
5. An mbed configured as SPI master is to be connected to three other mbeds, each configured as slave. Sketch a circuit showing how this interconnection could be made. Explain your sketch.
6. An mbed is to be set up as SPI master, using pins 11, 12 and 13, running at a frequency of 4 MHz, with 12-bit word length. The clock should idle at Logic 1, and data should be latched on its negative edge. Write the necessary code to set this up.
7. Repeat Question 4, but for I^2C, ensuring that you calculate time for the complete message.
8. Repeat Question 5, but for I^2C. Identify carefully the advantages and disadvantages of each connection.
9. You need to set up a serial network, which will have one master and four slaves. Either SPI or I^2C can be used. Every second, data has to be distributed, such that one byte is sent to Slave 1, four to Slave 2, three to Slave 3 and four to Slave 4. If the complete data transfer must take not more than 200 μs, estimate the minimum clock frequency that is allowable for SPI and I^2C. Assume there are no other timing overheads.
10. Repeat Question 4, but for asynchronous communication through a UART, assuming a baud rate of 500 kHz. Ensure that you calculate time for the complete message.

References

7.1. Spasov, P. (1996). Microcontroller Technology, The 68HC11. 2nd edition. Prentice Hall.
7.2. ADXL345 Datasheet, Rev. C. www.analog.com
7.3. NXP Semiconductors. The I2C Bus Specification and User Manual. Rev. 03. 2007. Document number UM10204.
7.4. Texas Instruments. TMP102. Low Power Digital Temperature Sensor. August 2007. Rev. October 2008. Document number SBOS397B.
7.5. SRF08 data. http://www.robot-electronics.co.uk/htm/srf08tech.shtml. Accessed 30 July 2011.

This page intentionally left blank