Chapter 7: Starting with Serial Communication

tw rev. 26.8.16

If you use or reference these slides or the associated textbook, please cite the original authors' work as follows:

Toulson, R. & Wilmshurst, T. (2016). Fast and Effective Embedded Systems Design - Applying the ARM mbed (2nd edition), Newnes, Oxford, ISBN: 978-0-08-100880-5.

www.embedded-knowhow.co.uk

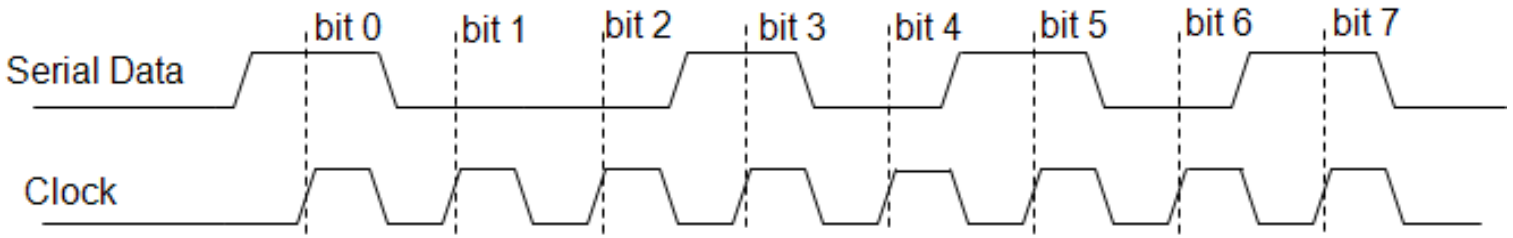
Introducing Serial Data Communication

In serial data transfer, one bit of the data is transmitted at a time, along a single interconnection.

While slower than parallel transfer, the small number of wires needed is a huge advantage, especially in the embedded world, i.e. less pcb tracks, interconnecting wires, and i.c. pins.

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?

One way is to send a clock signal alongside the data, with one clock pulse per data bit. The data is *synchronised* to the clock. This is called synchronous serial communication.

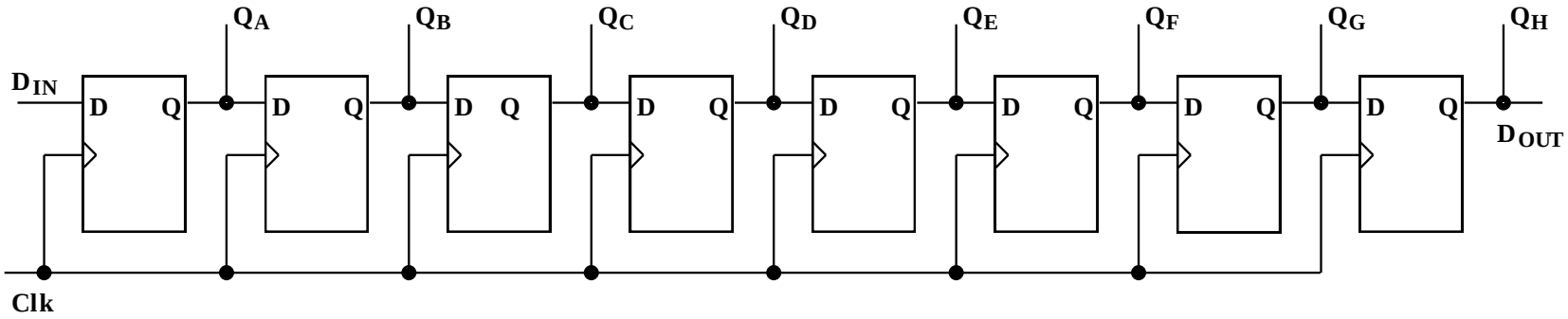


Synchronous serial data

The Basics of a Serial Port: the Shift Register

An essential feature of most serial links is a *shift register.* This is made up of a string of 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.

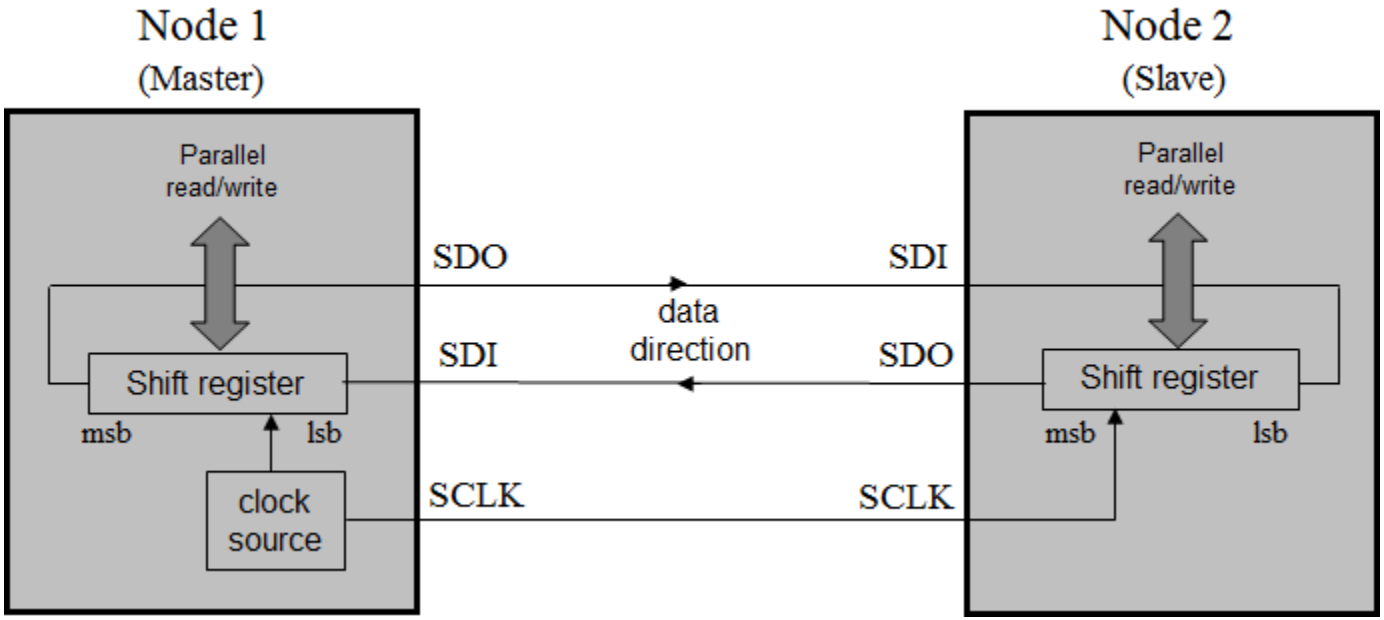
Every time the shift register is pulsed by the clock signal, each flip-flop passes its bit on to its neighbour on one side, and receives a new bit from its other neighbour. The one at the input end clocks in data received from the outside world, and the one of the output end outputs its bit.



An 8-Bit Shift Register – a Possible Receiver and/or Transmitter of Serial Data

A Simple Serial Link

A simple (synchronous) serial data link is shown. Node 1 is designated *Master*; it controls what's going on, as it controls the clock. The *Slave* is similar to the Master, but receives the clock signal from the Master.



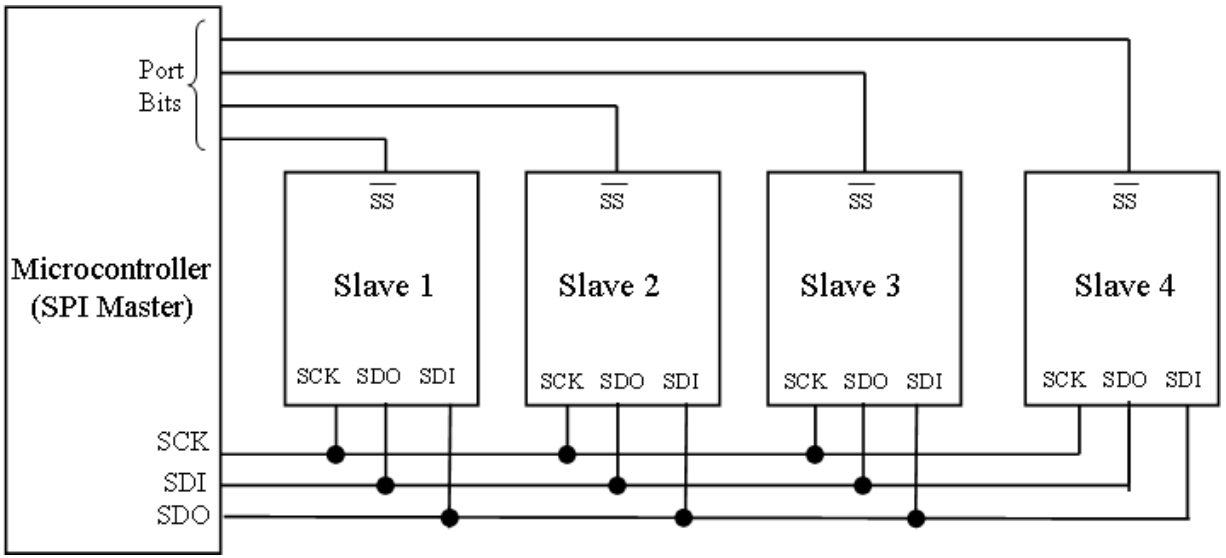
A simple Serial Link

Serial Peripheral Interface (SPI)

In the early days of microcontrollers, both National Semiconductors and Motorola started introducing simple serial communication, based on the previous Figure.

Each formulated a set of rules which governed how these links worked, and allowed others to develop devices which could interface correctly.

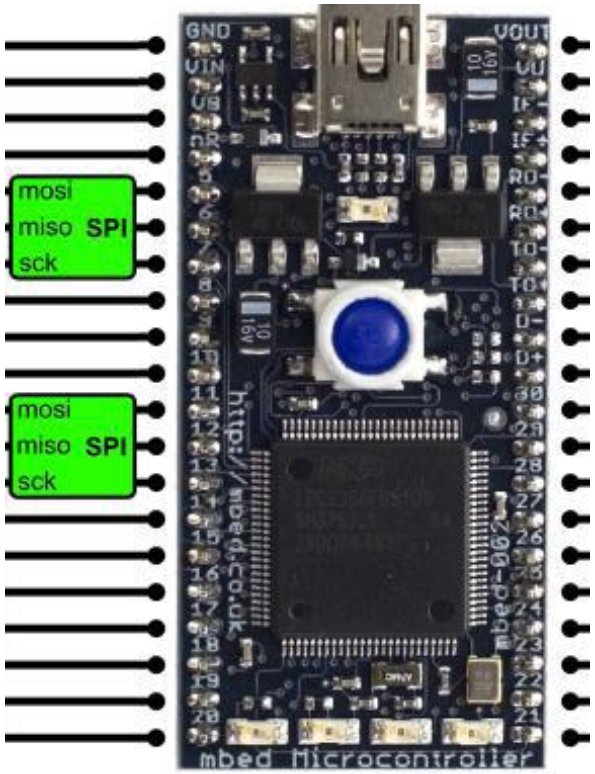
These became *de facto* standards. Motorola called its standard *Serial Peripheral Interface (SPI)*, and National Semiconductors called theirs *Microwire.* They're very similar to each other.



SPI interconnections for multiple Slaves

SPI on the mbed: Master

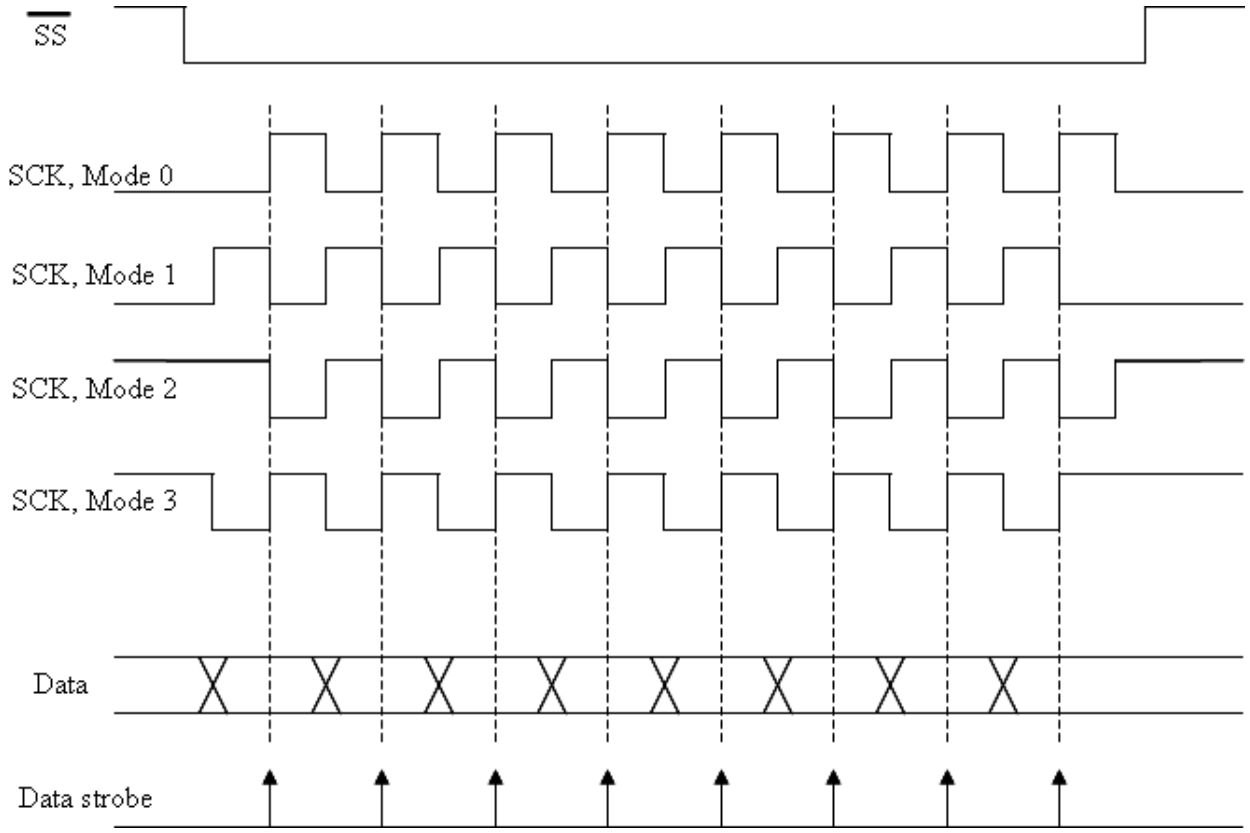
The mbed has two SPI ports, each can be configured as Master or Slave. The API summary for SPI Master is shown. On the mbed, as with many SPI devices, the same pin is used for SDI if in Master mode, or SDO if Slave. Hence this pin gets to be called MISO, Master in, Slave out. Its partner pin is MOSI.



Functions	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

SPI on the mbed: Mode

The mode is a feature of SPI which allows 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.



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

Simple SPI Master Program

This program shows a very simple setup for a SPI Master. It initialises 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 number of bits, and the mode. This program applies default values, i.e. 8 bits of data, and Mode 0 format.

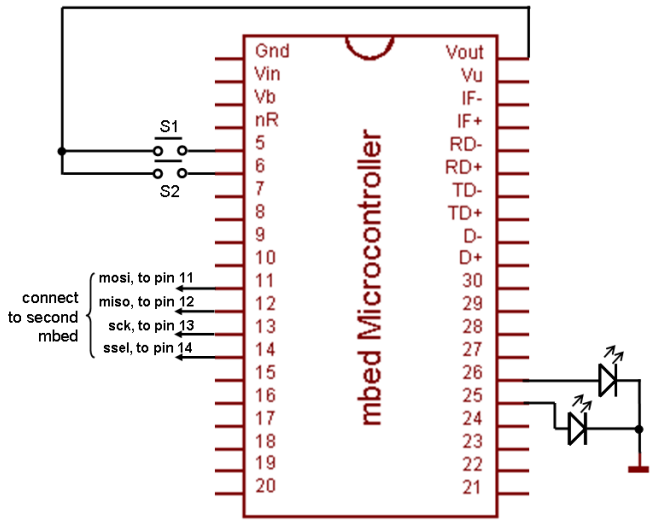
Output signals can be viewed on the oscilloscope.

```
/* 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);
  }
}
```

Creating an SPI data link: Master 1

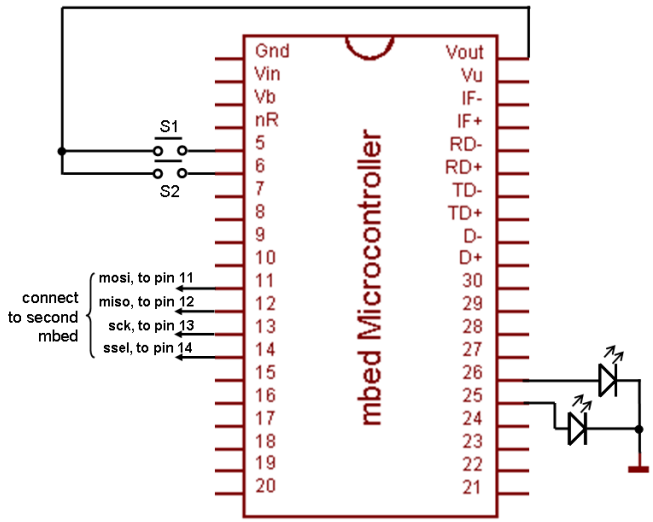
This program is written for the circuit shown. It declares a variable **switch_word,** the word that will be sent to the Slave, and the variable **recd_val,** which is the value received from the Slave.

```
/*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

//continued over
```



Creating an SPI data link: Master 2

```
    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 recognisable 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;
    }
  }
}
```



SPI on the mbed: Slave

The Slave program (next slide) uses the mbed functions shown. It mirrors the Master program.

The Slave program also declares variables **switch_word** and **recd_val,** and configures its **switch_word** just like the Master.

While the Master initiates a transmission when it wishes, the Slave must wait. It does this with the **receive( )** function. This returns 1 if data has been received, and 0 otherwise. If data has been received from the Master, then data has also been sent from Slave to Master.

Functions	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	Polls 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.

Creating an SPI data link: Slave

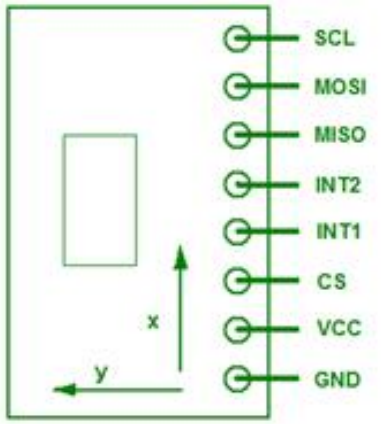
```
/*Program Example 7.3: Sets the mbed up as Slave, and exchanges data with a Master, sending
its own switch positions, and displaying those of the Master. as SPI slave.
                                                                                */
#include "mbed.h"
SPISlave ser_port(p11,p12,p13,p14); // 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 recognisable 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)
```

Using the ADXL345 Accelerometer

The ADXL345 measures acceleration on 3 axes, using an internal capacitor mounted in the plane of each axis. Acceleration causes the capacitor plates to move. It is an example of a Microelectromechanical system (MEMS).

It converts the analog voltages to digital and outputs these, either in SPI or I2C modes.

Because it is so small it is best purchased on a breakout board.



ADXL345 signal name	mbed pin
Vcc	Vout
Gnd	Gnd
SCL	13
MOSI	11
MISO	12
|CS	14

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 justify result; 0 = right justify 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

(From the ADXL345 datasheet: http://www.analog.com/static/imported-files/data_sheets/ADXL345.pdf)

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

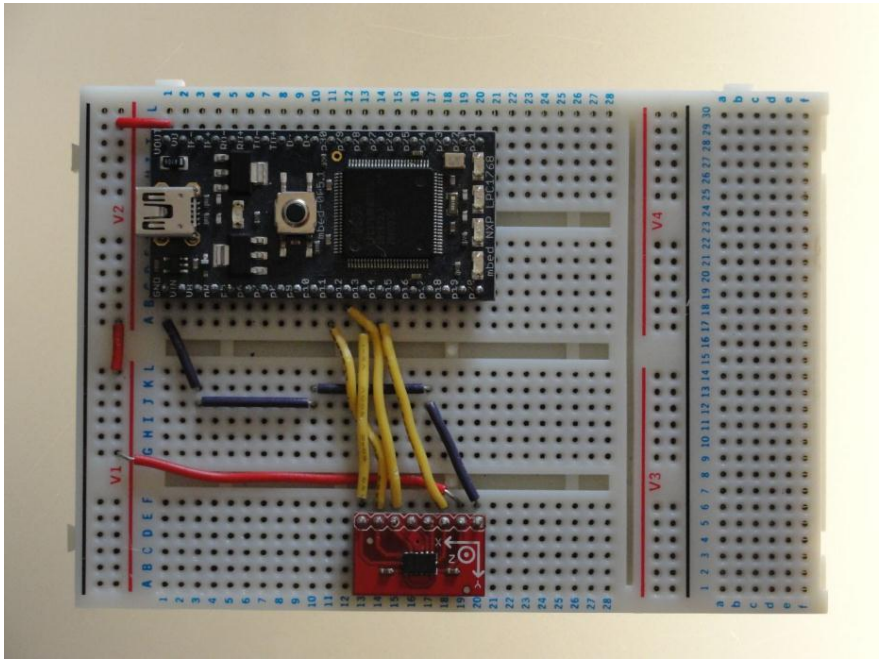
A simple ADXL345 program 1

This Program applies the ADXL345, reading acceleration in three axes, and outputting the data to the host computer screen. The SPI port on pins 11, 12 and 13 connects to the accelerometer.

```
/*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
  ...
//continued over
```

A simple ADXL345 program 2

```
//continued from previous
  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
  }
}
```



Evaluating SPI

The SPI standard is extremely effective. The electronic hardware is simple and therefore cheap, and data can be transferred rapidly.

There are disadvantages.

- There is no acknowledgement from the receiver, so in a simple system the Master cannot be sure that data has been received.
- There is no addressing. In a system where there are multiple slaves, a separate |SS line must be run to each Slave, as seen earlier. Therefore we begin to lose the advantage that serial communications should give us, i.e. a limited number of interconnect lines.
- There is no error-checking. Suppose some electromagnetic interference was experienced in a long data link, data or clock would be corrupted, but the system would have no way of detecting this, or correcting for it.

Overall SPI could be evaluated as simple, convenient and low-cost, but not appropriate for complex or high reliability systems.

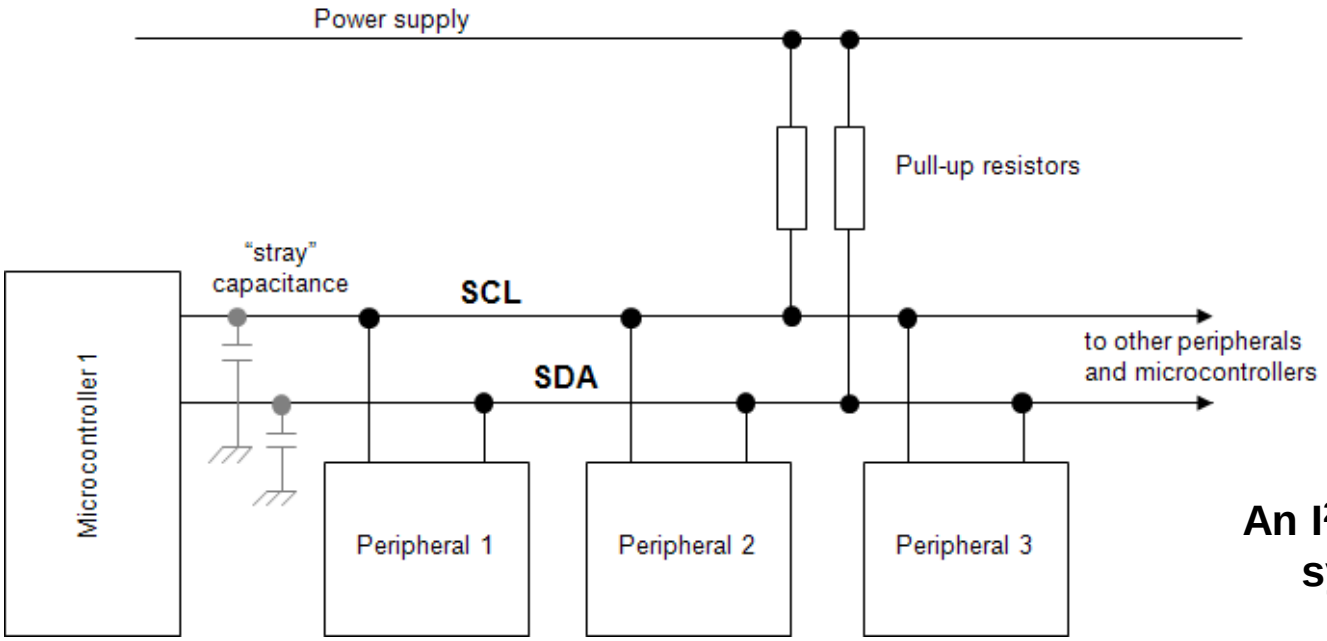
Introducing I²C

The name I²C stands for Inter-Integrated Circuit bus. It aims to resolve some of the perceived weaknesses of SPI.

I²C is a serial data protocol which operates with a master/slave relationship.

I²C only uses two physical wires, called serial data (SDA) and serial clock (SCL). This means that data only travels in one direction at a time.

Any node 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.



An I²C-based system

Simple I²C Communications

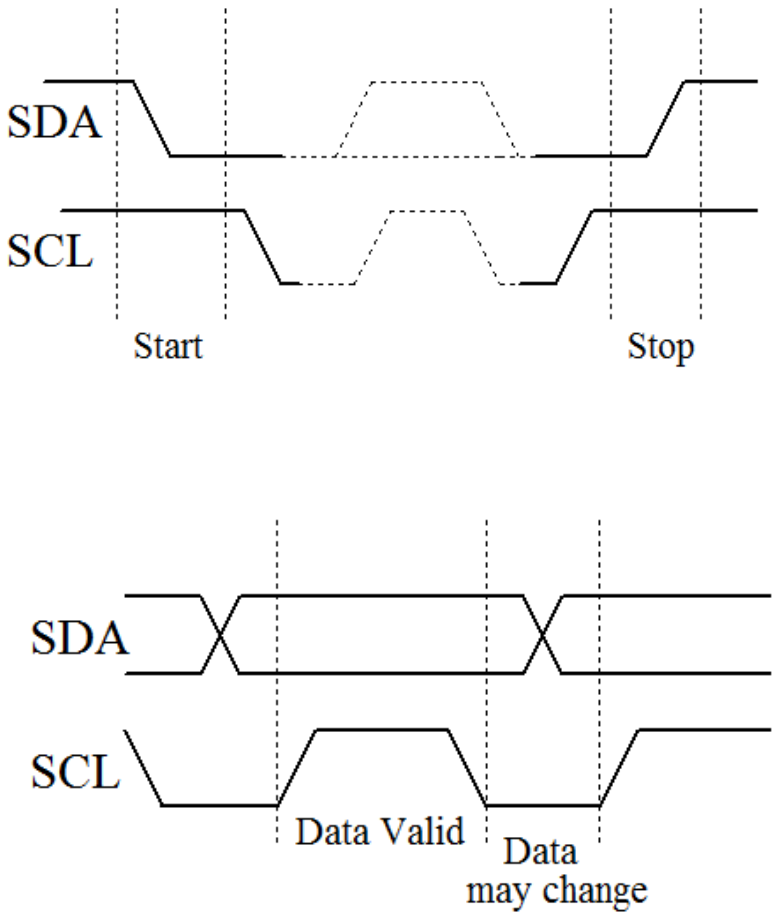
The device that initiates communication is termed the ‘master’. A device being addressed by the master is called a ‘slave’.

A data transfer is started by the master signalling a Start condition, followed by one or two bytes containing address and control information.

The Start condition is defined by a high to low transition of SDA when SCL is high.

A low to high transition of SDA while SCL is high defines a Stop condition

One SCL clock pulse is generated for each SDA data bit, and data may only change when the clock is low.



Start and Stop conditions

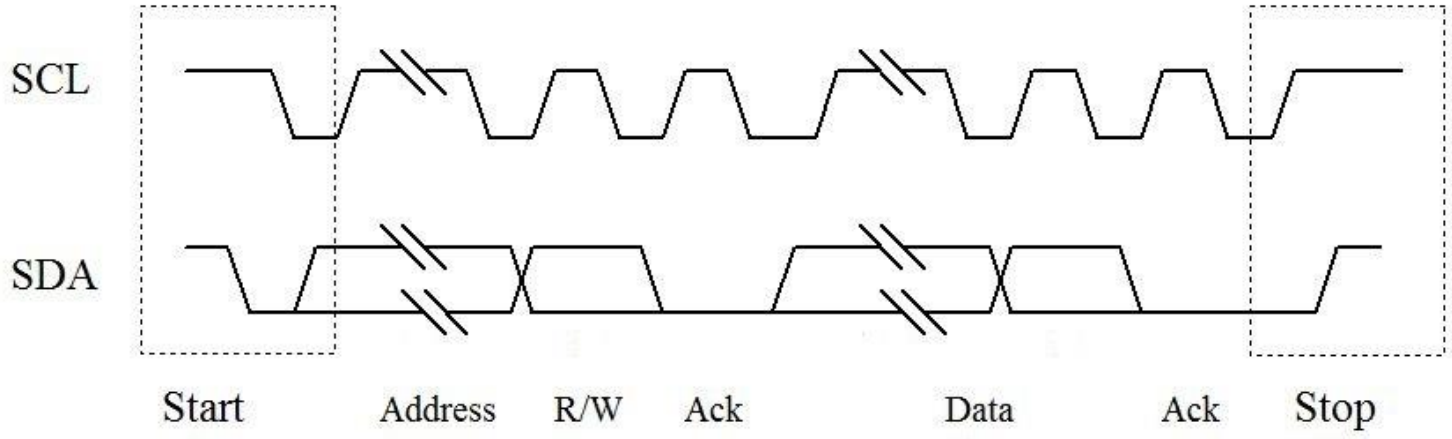
Simple I²C Communications

I²C has a built-in addressing scheme, which simplifies the task of linking multiple devices together. Each slave has a predefined address. Slaves monitor the bus and respond only to data and commands associate with their own address.

The byte following the Start condition is made up of seven address bits, and one data direction bit (Read/Write).

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 acknowledge from the receiver, during which time the transmitter relinquishes SDA control.



A Complete transfer of one byte

Questions from the Quiz

4. An 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 3 other mbeds, each configured as Slave. Sketch a circuit which shows 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 4MHz, 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.

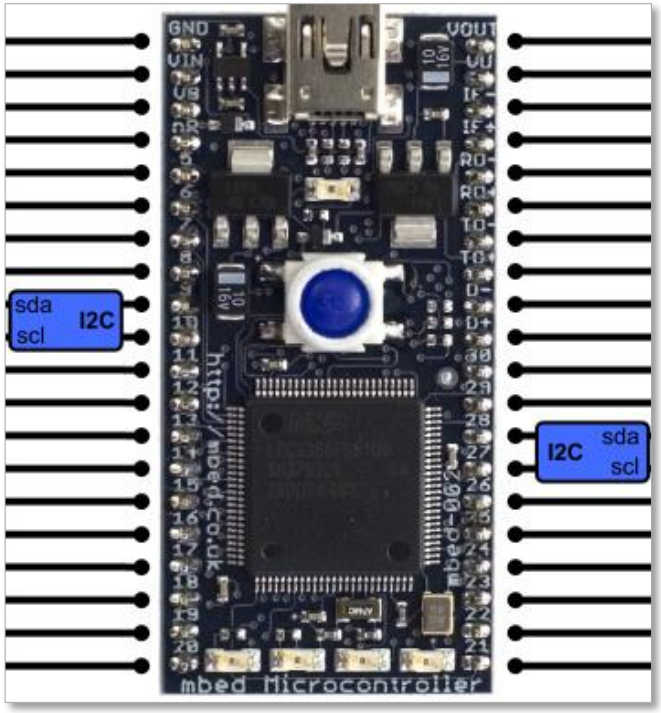
8. Repeat Question 5, but for I^2C. Identify carefully the advantages and disadvantages of each connection.

I²C on the mbed

There are two mbed I²C ports. Library Master and Slave functions are shown in the Tables below. These are more complex than SPI.

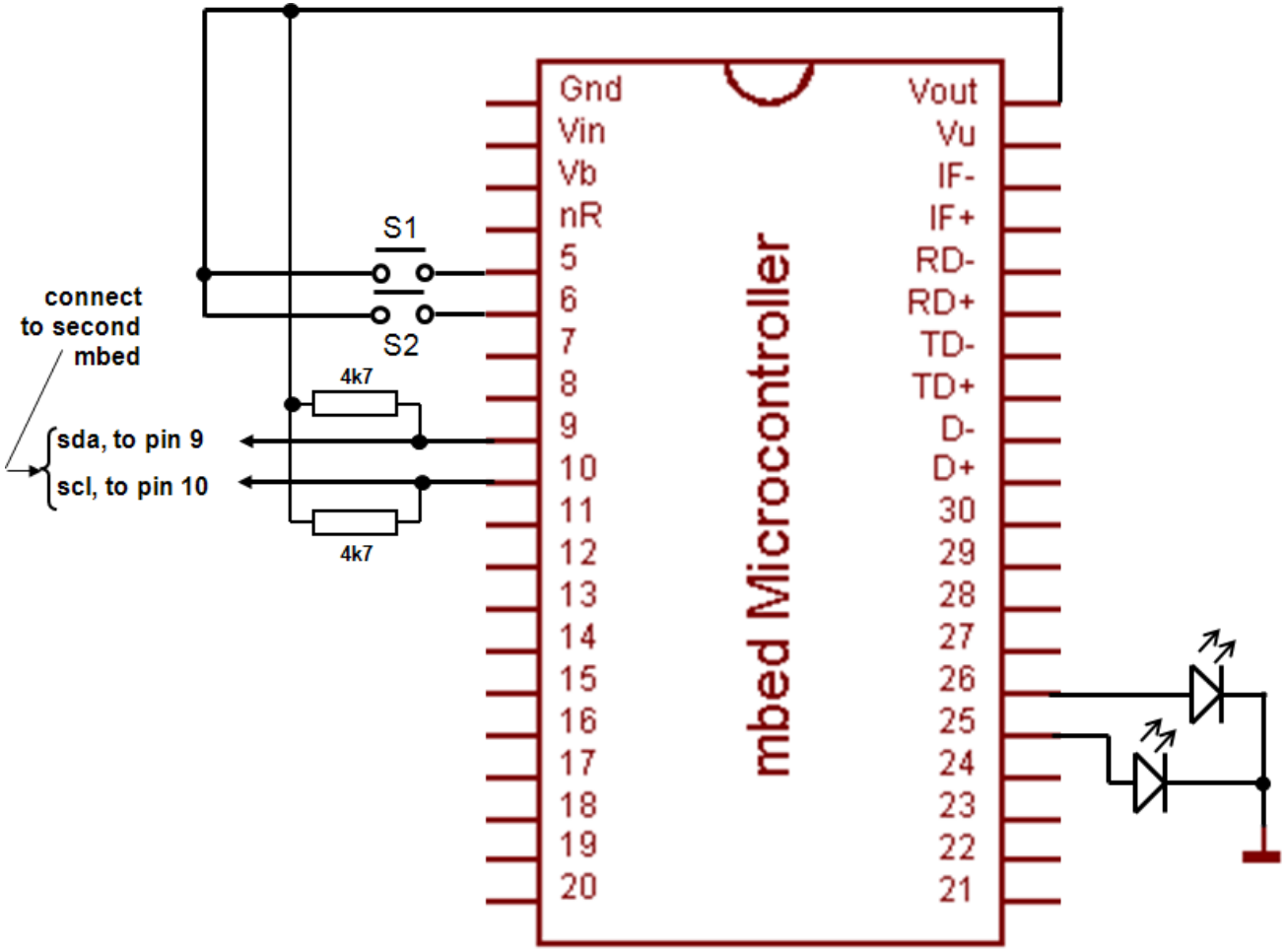
Functions	Usage
I2C	Create an I²C Master interface, connected to the specified pins
frequency	Set the frequency of the I²C interface
read	Read from an I²C slave
write	Write to an I²C slave
start	Creates a start condition on the I²C bus
stop	Creates a stop condition on the I²C bus

Function	Usage
I2CSlave	Create an I²C Slave interface, connected to the specified pins.
frequency	Set the frequency of the I²C interface
receive	Checks to see if this I²C Slave has been addressed.
read	Read from an I²C master.
write	Write to an I²C master.
address	Sets the I²C slave address.
stop	Reset the I²C slave back into the known ready receiving state.



Setting up an I²C mbed to mbed Data Link

This circuit is similar to the previous SPI one, but uses the I²C link. Pull-up resistors must be added externally.



Setting up the I²C Data Link (Master)

This program follows that of the SPI example, except that SPI elements are replaced by I²C.

```
/*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.
tjw 28.7.11*/

#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 return from slave
const int addr = 0x52;     // define the I2C slave address, an arbitrary even
number
```

Continued over…

Setting up the I²C Data Link (Master, cont.)

```
int main() {
  while(1) {
    switch_word=0xa0;     //set up a recognisable 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 Read/Write 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
    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;
    }
    wait(0.004);
  }
}
```

Setting up the I²C Data Link (Slave)

The slave program is similar to the SPI example, with SPI features replaced by I²C. The I²C slave responds to calls from the Master. The slave port is defined with the mbed utility **I2Cslave,** with **slave** chosen as the port name. The slave address is defined within the **main( )** function, the same 0x52 as in the Master program. The **receive( )** function tests if an I²C 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.

```
/*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.
tjw 28.7.11*/

#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
```

Continued over...

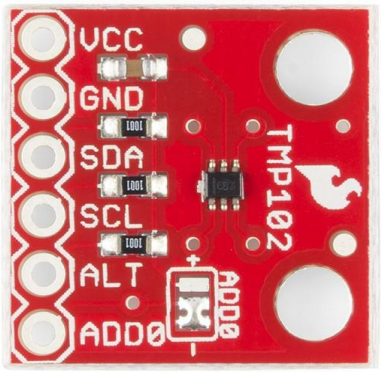
Setting up the I²C Data Link (Slave, cont.)

```
int main() {
  slave.address(0x52);
    while (1) {
      //set up switch_word from switches that are pressed
      switch_word=0xa0;          //set up a recognisable 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
      red_led=0;
      green_led=0;
      recd_val=recd_val&0x03;
      if (recd_val==1)
        red_led=1;
      if (recd_val==2)
        green_led=1;
      if (recd_val==3){
        red_led=1;
        green_led=1;
      }
   }                            //end of while
}                               //end of main
```

Communicating with an I²C temperature sensor

The Texas Instruments TMP102 temperature sensor has an I²C* data link. The TMP102 itself is a tiny device, just as we would want of a temperature sensor. We use it mounted on a small breakout board.

*Note from the data sheet that the TMP102 actually makes use of the SMbus - System Management Bus. This was defined by Intel in 1995, and is based on I²C. In simple applications the two standards can be mixed; for more advanced applications it is worth checking the small differences which there are.



Signal	Link to Mbed Pin	Notes
VCC (3.3V)	40	
SDA	9	2.2 kΩ pull-up to 3.3 V
SCL	10	2.2 kΩ pull-up to 3.3 V
GND (0V)	1	
ALT (Alert)	1	
ADD0	1	Connect to — Slave address 0V — 0x90 Vcc — 0x91 SDA — 0x92 SCL — 0x93

Communicating by I²C with the TMP102 temperature sensor

(check TMP102 data sheet for some of the codes used)

```
/*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, sc1
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);
  }
}
```

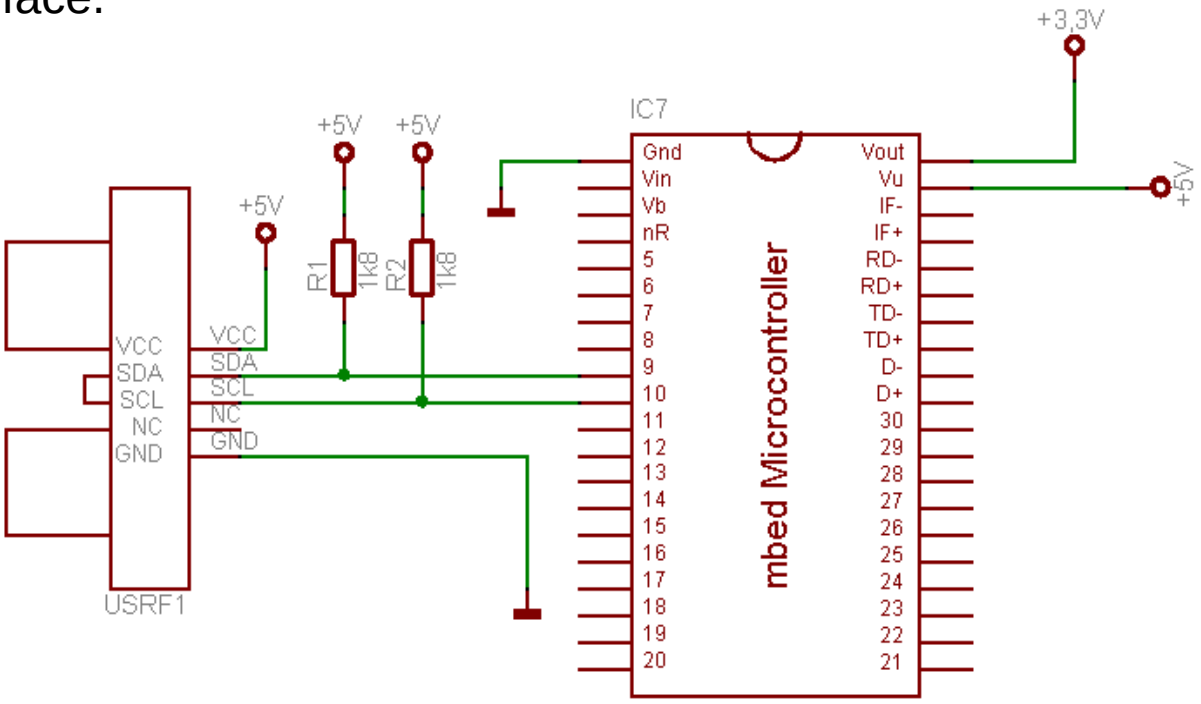
Using the SRF08 ultrasonic range finder

The SRF08 ultrasonic range finder 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. Knowing the speed of sound in air, the actual distance can be calculated.

The SRF08 has an I^2C interface.



Linking the SRF08 to an mbed

```
/*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, sc1
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;                    //initialise, 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);
  }
}
```

Note that:

- The SRF08 I^{2}C address is 0xE0.
- The pointer value for the command register is 0x00.
- A data value of 0x51 to the command register initialises 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.

Evaluating I^2C

The I^2C protocol is well-established and versatile. It is widely applied to short distance data communication, and can be used to set up more complex networks, and to add and subtract nodes with comparative ease.

I^2C provides a reasonably reliable system. If an addressed device doesn't send an acknowledgement, the Master can act upon that fault.

But

the bandwidth is comparatively limited, even in the faster versions of I^2C.

I^2C is still susceptible to interference, and does not check for errors. Therefore it would be unlikely to be used in a medical, motor vehicle or other high reliability application.

Evaluating synchronous serial data communication

Synchronous serial communication protocols (like SPI, I2C) are extremely useful ways of moving data around. But taking a clock signal to every node has 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 which limit the overall data rate.
- Over long distances, clock and data themselves could lose synchronisation.

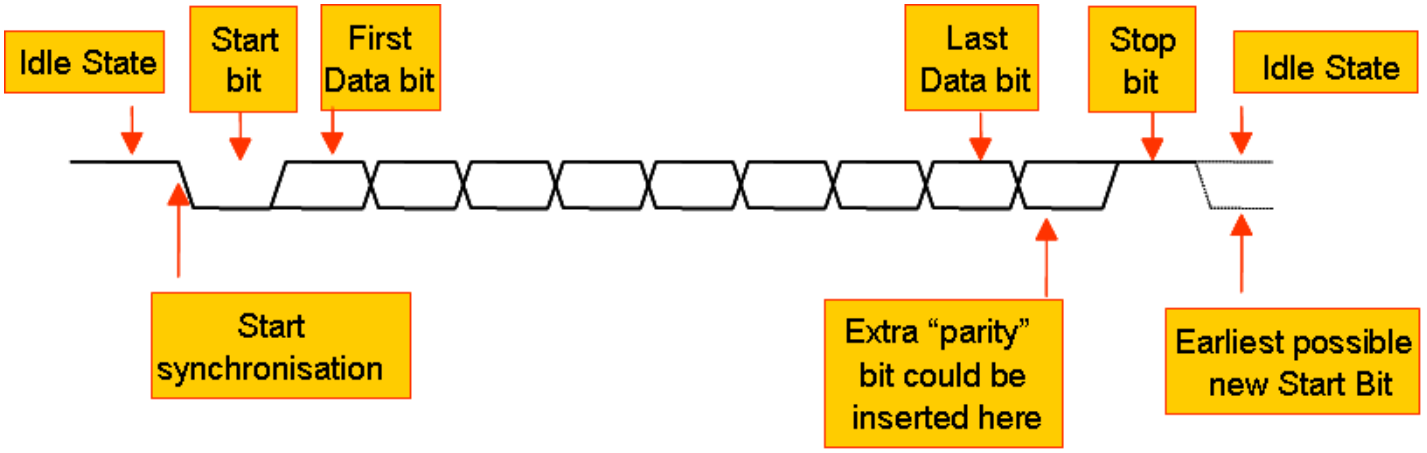
Asynchronous serial data communication

Asynchronous communication doesn't require the clock to be connected between nodes. A common approach is:

• Data rate is predetermined – both transmitter and receiver are pre-set to recognise 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.

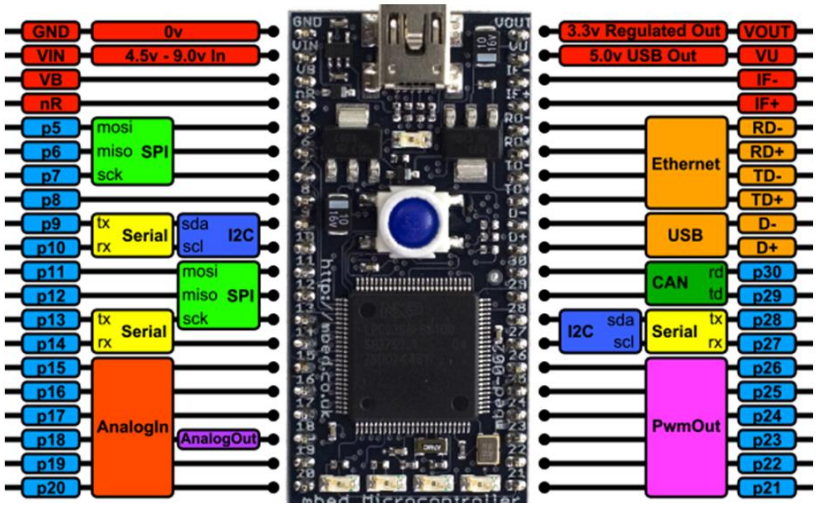
• Each byte or word is *framed* with a Start and Stop bit. These allow synchronisation to be initiated before the data starts to flow.

• An asynchronous serial port is generally called a UART, *Universal Asynchronous Receiver/ Transmitter.* A UART has one connection for transmitted data, called TX, and another for received data, called RX. The data rate that receiver and transmitter will operate at must be pre-determined; this is specified by its *baud rate.*



Applying asynchronous communication on the mbed

The LPC1768 has *four* UARTs. *Three* of these link to the mbed pins, simply labelled “Serial”. The API summary is shown.



Functions	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

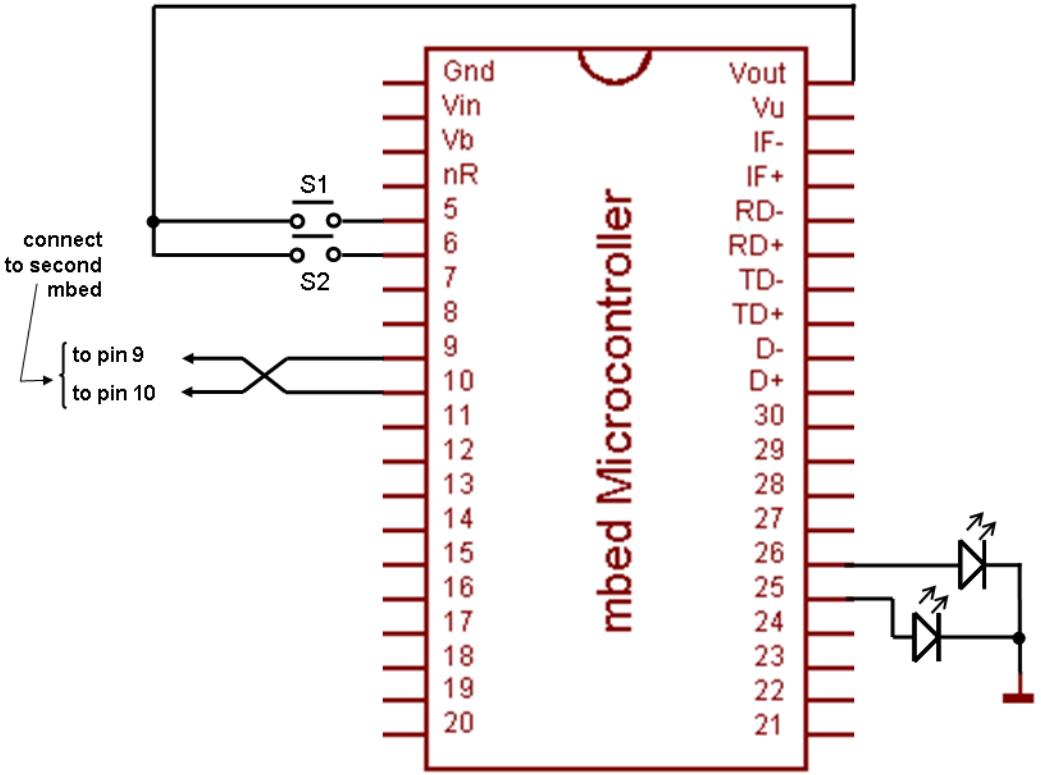
A Question from the Quiz

10. A UART is running with a 500 kHz baud rate. How long does it take for a single message containing one data byte to be transmitted? Ensure that you calculate time for the complete message.

Bidirectional data transfer between two mbed UARTs: Circuit

This replicates earlier circuits for SPI and I^2C, but now uses the UART. Notice how the TX from one mbed connects to the RX of the other, and vice versa.

The program appears in the next slide ; the *same* program should be loaded into both mbeds – there is no Master or Slave.



Bidirectional data transfer between two mbed UARTs

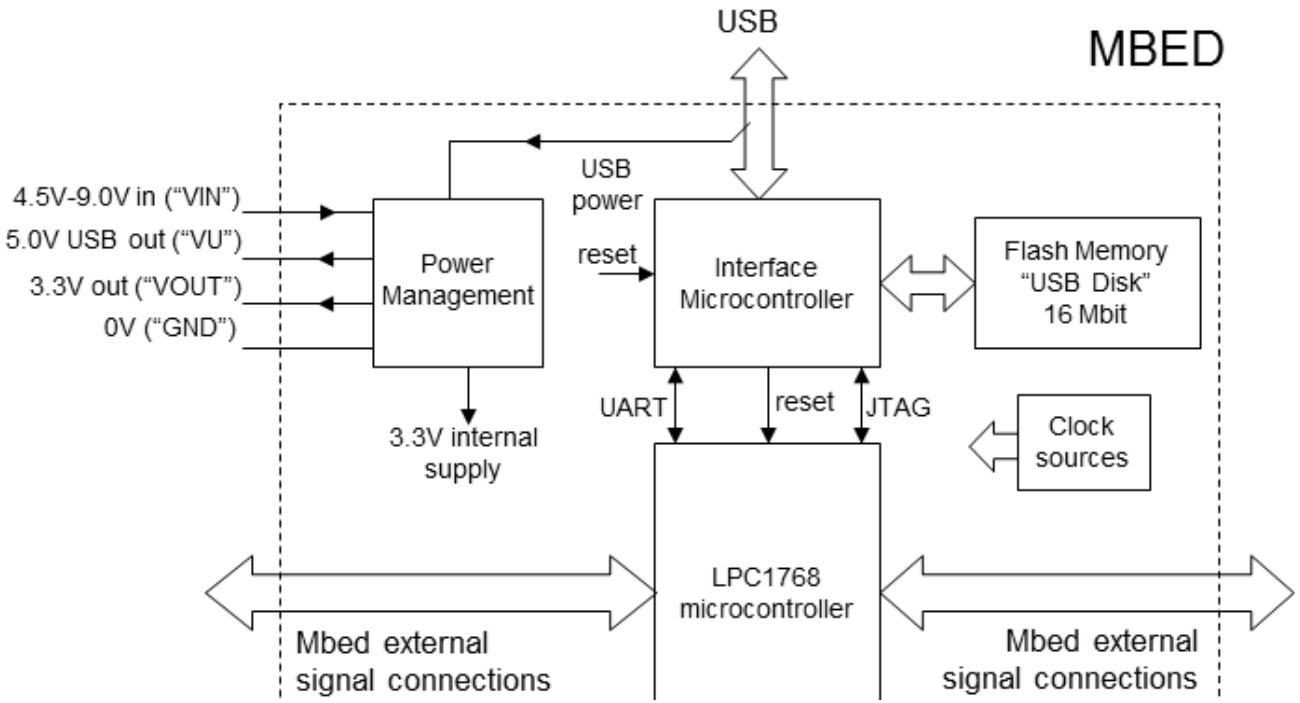
```
/*Program Example 7.9: Sets the mbed up for async communication, and exchanges data with a
similar node, sending its own switch positions, and displaying those of the other.
                                                                    */
#include "mbed.h"
Serial async_port(p9, p10);          //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 recognisable 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)
    ...
```

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 repeated below. This UART acts just like any of the others, in terms of its use of the API. The mbed compiler recognises **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.



Universal Serial Bus (USB)

In the early days of personal computing, different peripheral devices each came with their own type of connector, and each required software reconfigurations when they were fitted. This was annoying, inefficient, and inflexible.

The USB protocol was introduced to provide a more flexible and “universal” interconnection system, whereby peripherals could be added or removed without the need for reconfiguring the whole system (i.e. moving to a “plug and play” capability).

A USB network has one host, and can have one or many *functions*, i.e. USB compatible devices that can interact with the host.

USB version 2.0 uses a 4-wire interconnection. Two, labelled D+ and D-, carry the differential signal, and two are for power and earth. USB functions can draw power from the host, taking up to 100 mA at a nominal 5 V.

USB capability on the mbed

The mbed has two USB ports. One connects to the host PC, which provides power to the mbed. The second is on pins 31 and 32 .

There are a many USB mbed features available for use, supported by the **USBDevice** library. Most of them allow the mbed to emulate a number of external devices, through USB.

Mbed USB library	Description
USBMouse	Allows the mbed to emulate a USB mouse
USBKeyboard	Allows the mbed to emulate a USB keyboard
USBMouseKeyboard	A USB mouse and keyboard feature set combined in a single library
USBHID	Allows custom data to be sent and received from a Human Interface Device (HID) allowing custom USB features to be developed without the need for host drivers to be installed
USBSerial	Emulates an additional standard serial port on the mbed, through the USB connections
USBMIDI	Allows send and receive of MIDI messages in communication with a host PC using MIDI sequencer software
USBAudio	Allows the mbed to be recognised as an audio interface allowing streaming audio to be read, output or analysed and processed.
USBMSD	Emulates a mass storage device over USB, allowing interaction with a USB storage device.

Using the mbed to emulate a USB mouse

With **USBMouse** it is possible to make the mbed behave like a standard USB mouse, sending position and button press commands to the host. The program example implements a **USBMouse** interface and continuously sends relative position information to move the mouse pointer around four co-ordinates which make up a square. These are defined by the two arrays **dx** and **dy**.

To run this program it is necessary to import the USBDevice library through the compiler.

```
/* Program Example 7.10: Emulating a USB mouse
                                                                */
#include "mbed.h"                       // include mbed library
#include "USBMouse.h"                   // include USB Mouse library
USBMouse mouse;                         // define USBMouse interface

int dx[]={40,0,-40,0};                  // relative x position co-ordinates
int dy[]={0,40,0,-40};                  // relative y position co-ordinates

int main() {
  while (1) {
    for (int i=0; i<4; i++) {       // scroll through position co-ordinates
      mouse.move(dx[i],dy[i]);      // move mouse to co-ordinate
      wait(0.2);
    }
  }
}
```

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 supporting library.
- While a very useful standard, SPI has certain very clear limitations, relating to a lack of flexibility and robustness.
- The I^2C protocol is a more sophisticated serial alternative to SPI; it runs on a 2-wire bus, and includes addressing and acknowledgement.
- I^2C 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^2C has limitations which mean it cannot be used for high reliability applications.
- A very wide range of peripheral devices are available, including intelligent sensors, which communicate through SPI and I^2C.
- 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.
- The USB protocol is designed specifically for allowing plug-and-play communications between a computer and peripheral devices such as a keyboard or mouse.
- There are a number of mbed USB libraries allowing the mbed to operate as a mouse or a keyboard, or as an audio or MIDI interface for example.