

Chapter 7: Starting with Serial Communication

tw rev. 11.03.2024

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

Wilmshurst, T., Spink, T. & Toulson, R. (2024). Fast and Effective Embedded Systems Design – From Bits and Bytes to IoT, with the ARM Mbed (3rd edition), Newnes, Oxford, ISBN: 978-0323951975.

www.embedded-knowhow.co.uk

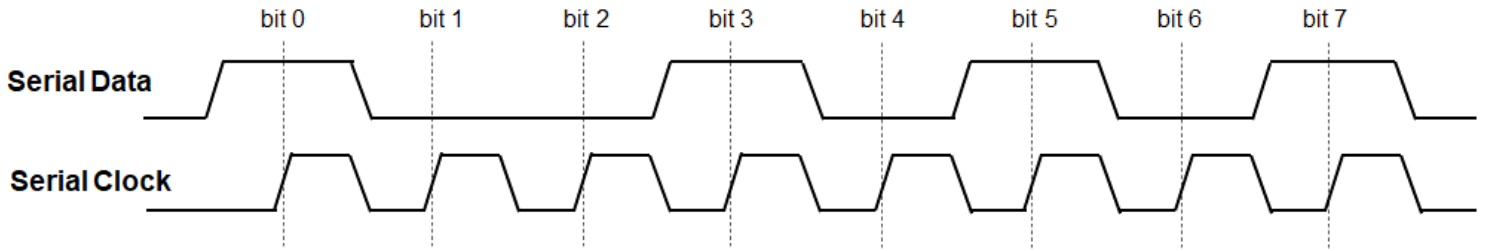
Introducing Serial Data Communication

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.

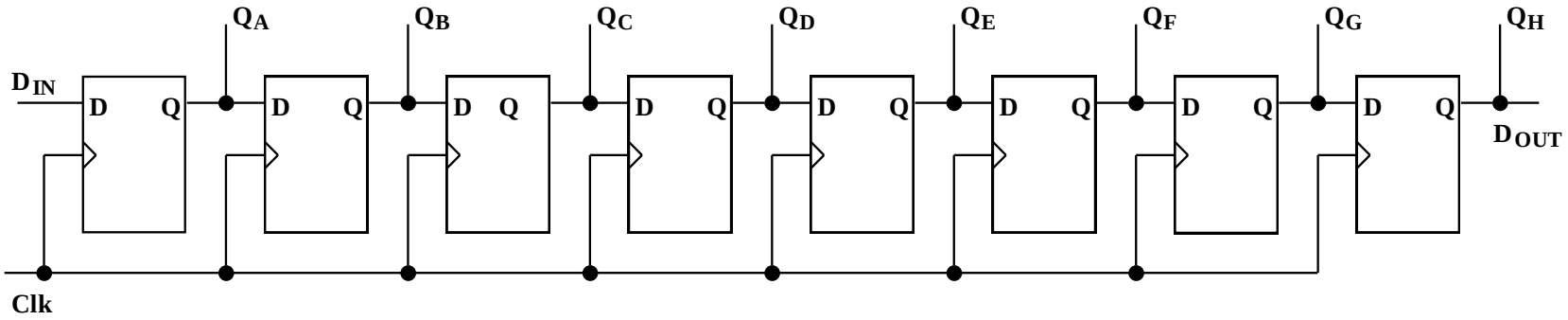


Synchronous serial data

The Basics of a Serial Port: the Shift Register

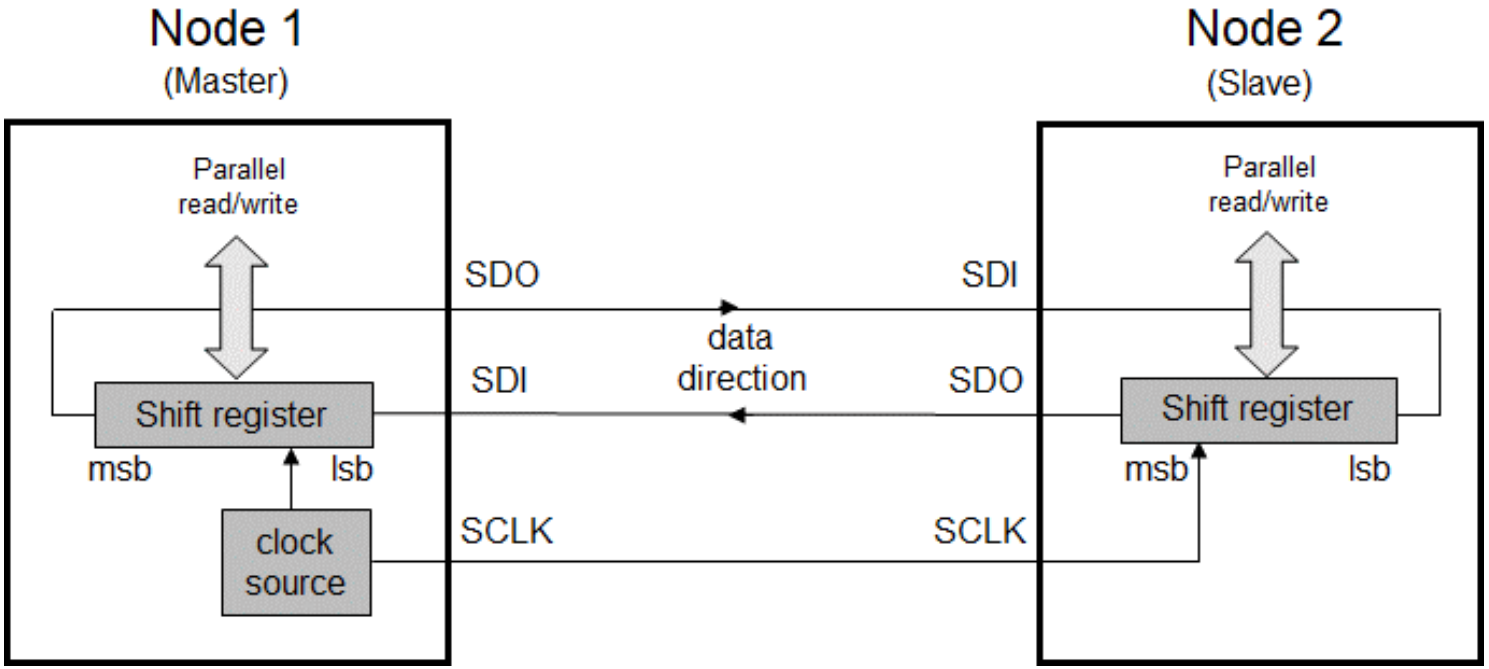
An essential feature of most serial links is a *shift register.* This is made up of a string of bistables, connected so that the output of one is connected to the input of the next. Each bistable holds one bit of information.

Every time the shift register is pulsed by the clock signal, each bistable 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.



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 its clock signal from the Master.

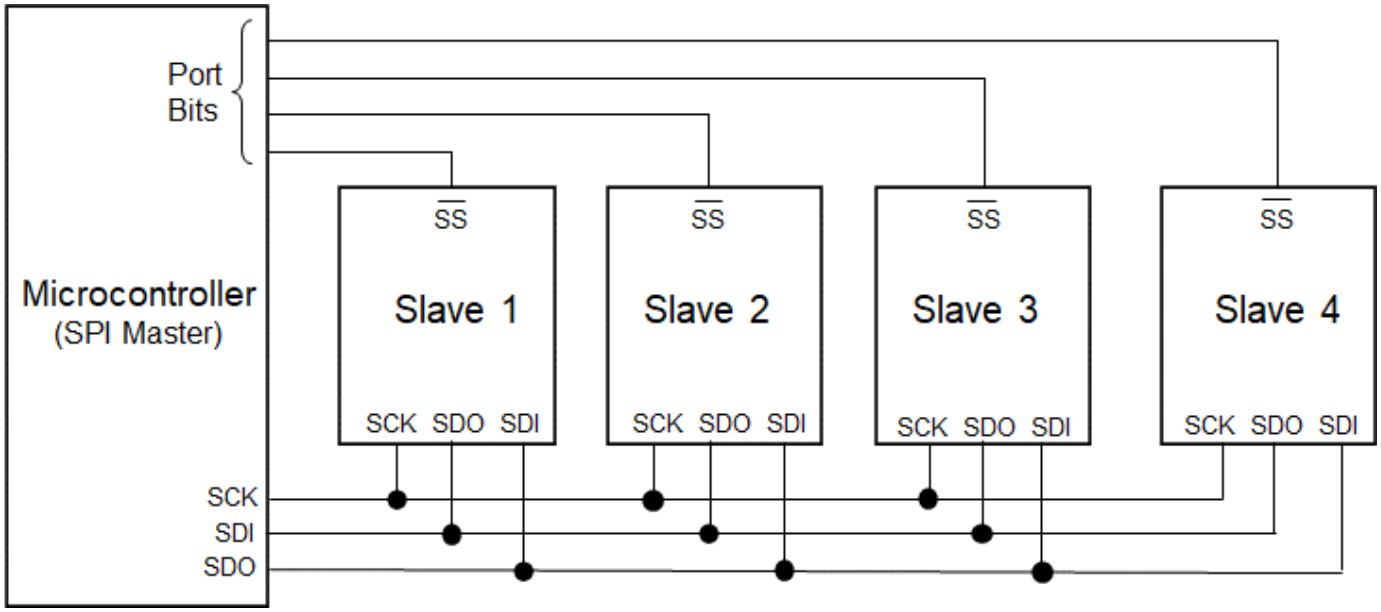


Serial Peripheral Interface (SPI)

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

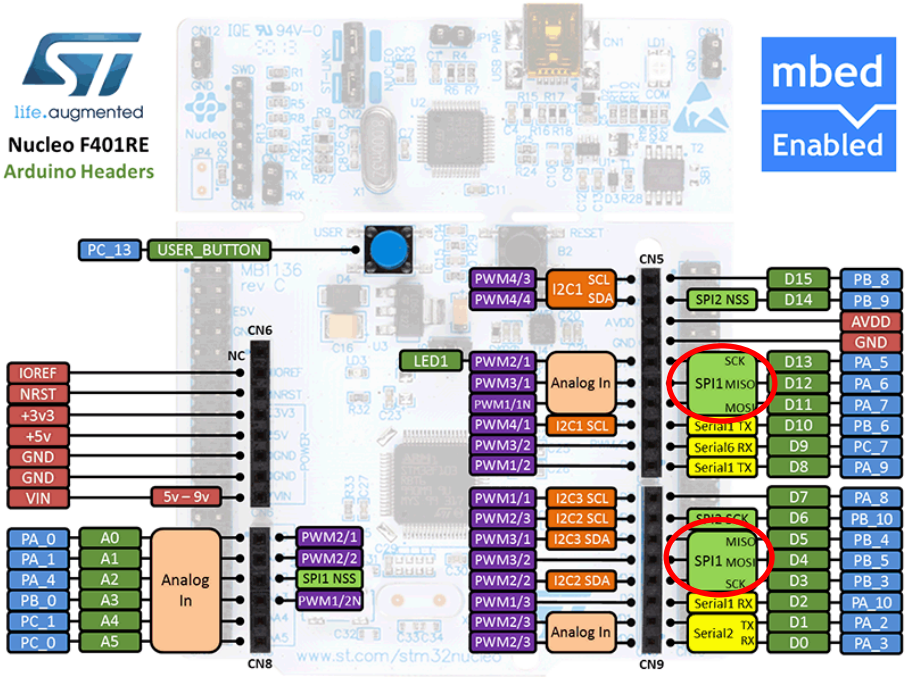
Each formulated a set of rules which governed how these links worked, and allowed others to develop devices which could interface correctly.

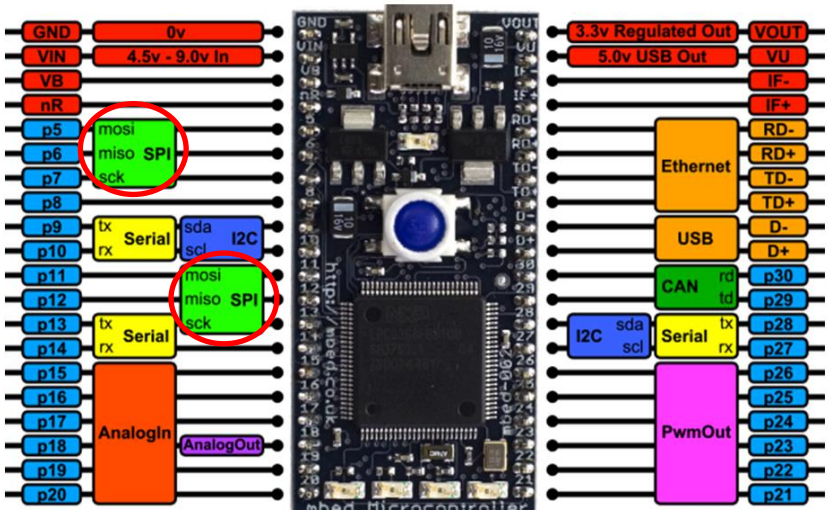
These became *de facto* standards. Motorola called its standard *Serial Peripheral Interface (SPI)*, and National Semiconductors called theirs *Microwire.* They're very similar to each other.



SPI in the Mbed Environment

Both Mbed LPC1768 and Nucleo F401RE have SPI ports. As with many SPI devices, the same pin is used for SDI if in Master mode, or SDO if Slave. Hence this pin is often called MISO (or miso), i.e. Master in, Slave out. Its partner pin is MOSI (Master out, Slave in).





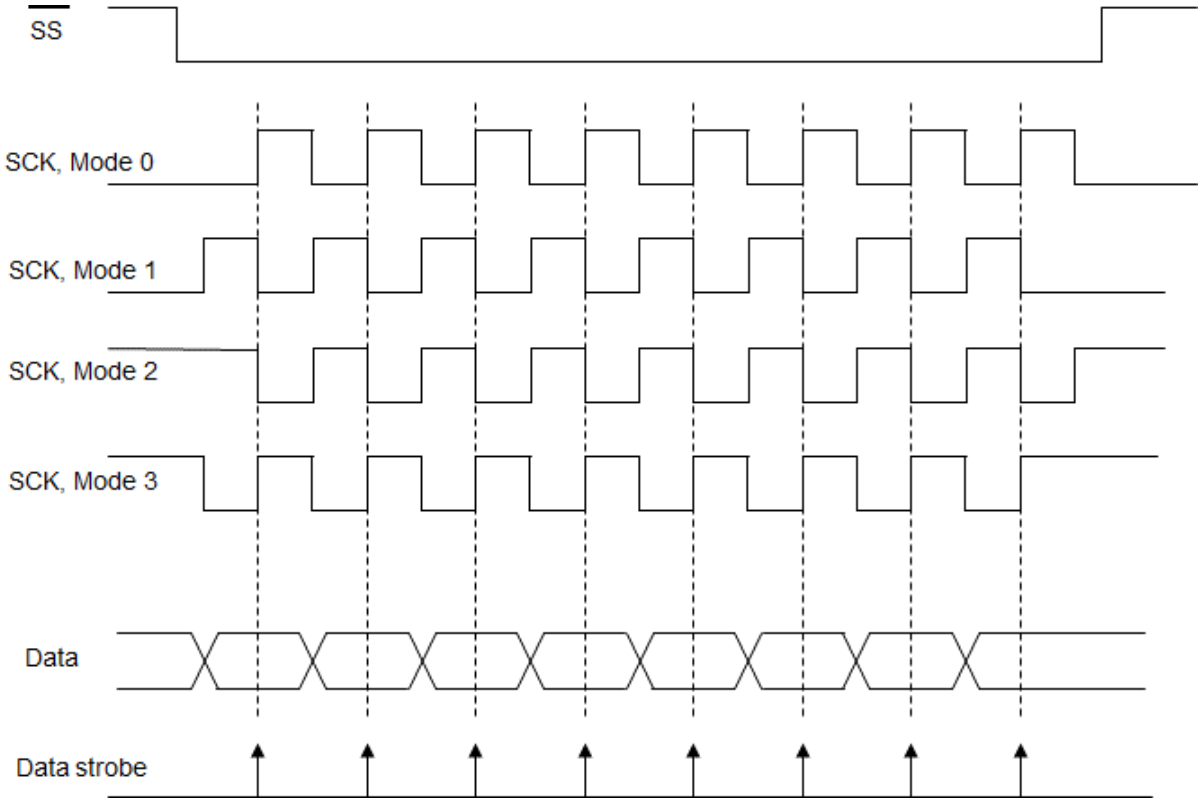
SPI in the Mbed Environment

The API summary available for SPI Master is shown.

Functions	Usage
Constructor	
`SPI (PinName mosi, PinName miso, PinName sclk, PinName ssel)`	Create a SPI Master connected to the specified pins (which must be a valid SPI port for the target board in use)
Member Functions	
`void format(int bits,int mode)`	Configure the data length and transmission mode
`void frequency(int Hz)`	Set the SPI bus clock frequency
`virtual int write(int val`	Write to the SPI Slave and return the response

SPI with the Mbed API: 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 target as SPI Master, and continuously sends
a single byte
Works for both LPC1768 and F401RE, with pin allocations shown
                                                                        */
#include "mbed.h"
//Apply line below for F401RE
SPI ser_port(D11, D12, D13);   //mosi, miso, sclk
//Apply line below for LPC1768
//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 1 MHz
  while (true){
    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.

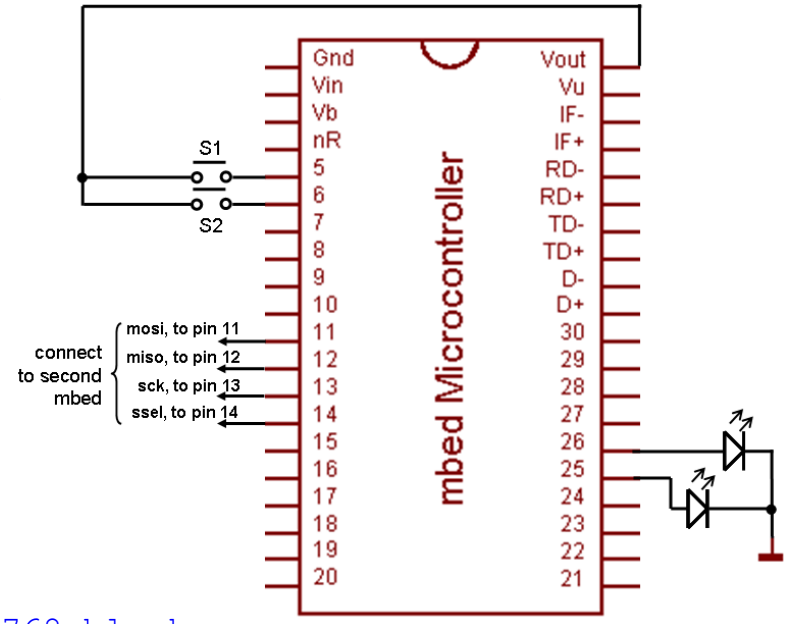
```
/*Program Example 7.2. Sets the Mbed up as Master, and exchanges data with a Slave,
sending its own switch positions, and displaying those of the Slave.
Works for both LPC1768 and F401RE, with pin allocations shown     */

#include "mbed.h"
//Use the following 6 lines for F401RE
SPI ser_port(D11, D12, D13);  // mosi, miso, sclk
DigitalOut red_led(A0);       //red led
DigitalOut green_led(A1);     //green led
DigitalOut cs(D14);           //"Slave select"
DigitalIn  switch_ip1(D4,PullUp);
DigitalIn  switch_ip2(D5,PullUp);

/* Use the following 6 lines for LPC1768
ser_port(p11, p12, p13);   // mosi, miso, sclk
DigitalOut red_led(p25);   //red led
DigitalOut green_led(p26); //green led
DigitalOut cs(p14);        //"Slave select"
DigitalIn  switch_ip1(p5,PullUp);
DigitalIn  switch_ip2(p6,PullUp);  *//end of LPC1768 block

char switch_word ;         //word we will send
char recd_val;             //value return from Slave

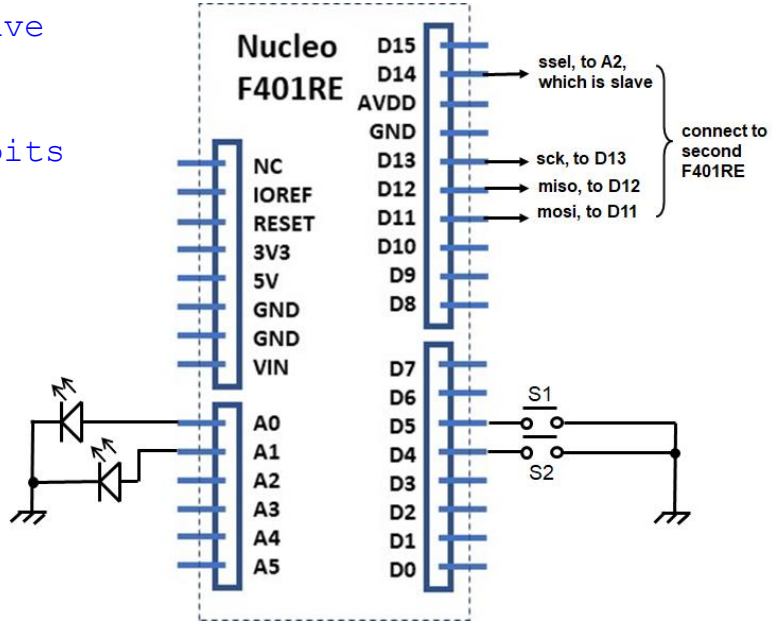
//continued over
```



Creating an SPI data link: Master 2

```
int main() {
  while (true){
    //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_us(10000);

    //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 with the Mbed API: Slave

The Slave program (next slide) uses the API 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
Constructor	
`SPISlave(PinName mosi, PinName miso, PinName sclk, PinName ssel)`	Create a SPI slave connected to the specified pins
Member Functions	
`void format(int bits, int mode)`	Configure the data length and transmission mode (see Table 7.2)
`int receive(void)`	Polls the SPI to see if data has been received: 0=no data waiting 1=data waiting
`int read(void)`	Retrieve data from receive buffer as slave, returns data in slave buffer
`void reply(int value)`	Fill the transmission buffer with the data *value* to be clocked out on the next received message from the master.

Creating an SPI Data Link: Slave (1)

```
/*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.
Works for both LPC1768 and F401RE, with pin allocations shown. F401RE interconnect may
give improved performance with slave select link not connected.      */

#include "mbed.h"

//The following block for Nucleo F401RE
SPISlave ser_port(D11,D12,D13,A2);//mosi, miso, sclk, ssel
DigitalOut red_led(A0);              //red led
DigitalOut green_led(A1);            // green led
DigitalIn  switch_ip1(D4,PullUp);
DigitalIn  switch_ip2(D5,PullUp);
/* The following block for LPC1768
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);          */ //end of LPC1768 block

char switch_word ;                  //word we will send
char recd_val;                      //value received from Master

//continued over
```

Creating an SPI Data Link: Slave (2)

```
int main() {
  //default formatting applied
  while(true) {
    //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)
    ...
  }
}
```

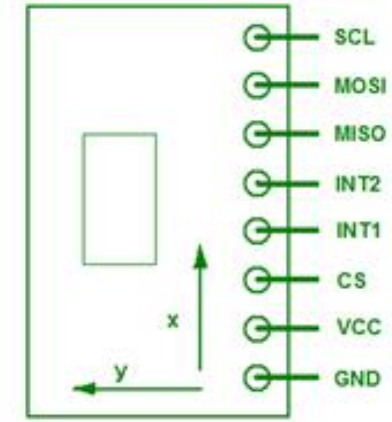
Using the ADXL345 Accelerometer

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 LPC1768 pin	F401RE pin
Vcc	Vout	3V3
Gnd	Gnd	GND
SCL	13	D13
MOSI	11	D11
MISO	12	D12
ǀCS	14	D10

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 = ± 16 *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 outputs data to the host computer screen. The SPI port connects to the accelerometer.

```
/*Program Example 7.4: Reads values from accelerometer through SPI, and outputs
continuously to terminal screen.
Works for both LPC1768 and F401RE, with pin allocations shown
*/

#include "mbed.h"
SPI acc(D11,D12,D13);             //Use pins p11,p12,p13 for LPC1768
DigitalOut cs(D10);               //p14 for LPC1768. For chip/slave select
char buffer[6];                   //raw data array type char
int16_t data[3];                  //16-bit twos-complement integer data
// floating point data, to be displayed on-screen
float x, y, z;

int main() {
  cs = 1;                           //cs idles high
  acc.format(8,3);                //8 bit data, Mode 3
  acc.frequency(2000000);         //2MHz clock rate
  cs = 0;                           //start a data transfer
  acc.write(0x31);                //data format register
  acc.write(0x0B);                //format +/-16g, 0.004g/LSB
  cs = 1;                           //end of transfer
  cs = 0;                           //start a new transfer
  acc.write(0x2D);                //power ctrl register
  acc.write(0x08);                //measure mode
  cs = 1;                           //end of transfer

  ...
//continued over
```

A Simple ADXL345 Program (2)

```
while (true) {                  //infinite loop
    cs = 0;                         //start a transfer
    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 transfer
    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];
    //convert to g value, float
    x = 0.004*data[0]; y = 0.004*data[1]; z = 0.004*data[2];
    printf("x = %+1.2fg\t y = %+1.2fg\t z = %+1.2fg\n\r", x, y,z); // print
    ThisThread::sleep_for(500ms);
  }
}
```

a) the main.cpp file

```
{
    "target_overrides":{
        "*": {
            "target.printf_lib": "std"
        }
    }
}
```

b) the mbed_app.json file (for printing of floating point numbers)

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.

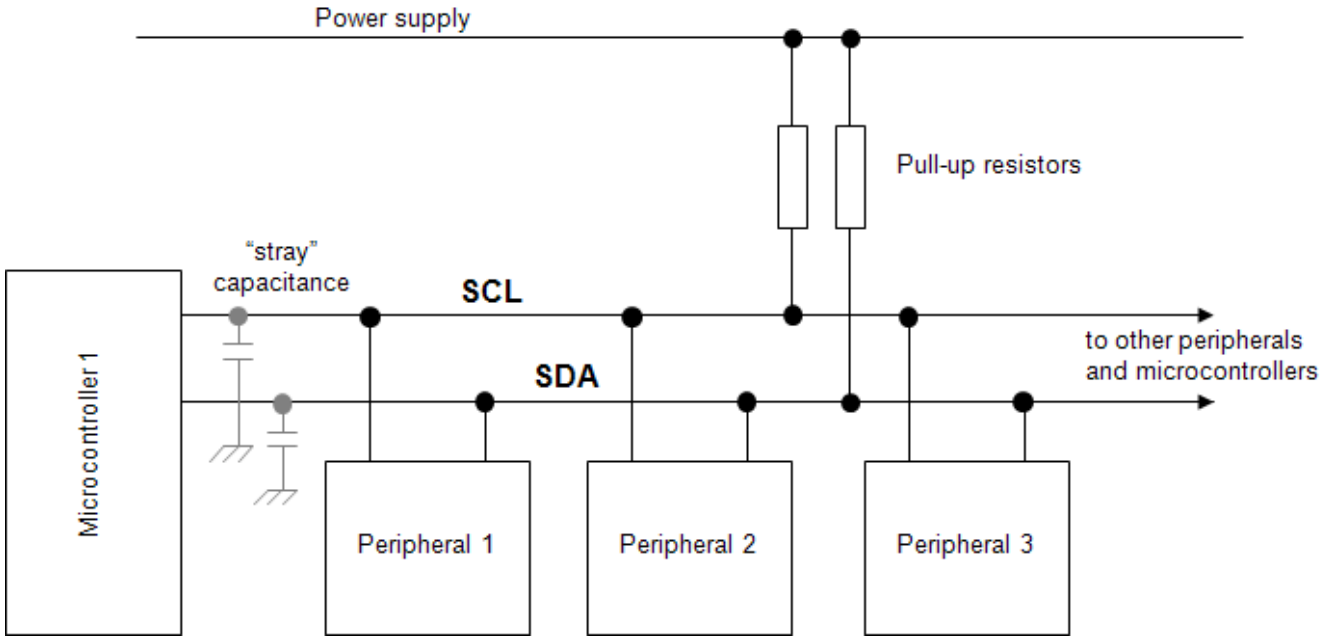
Introducing I²C

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.



Simple I^2C Communications

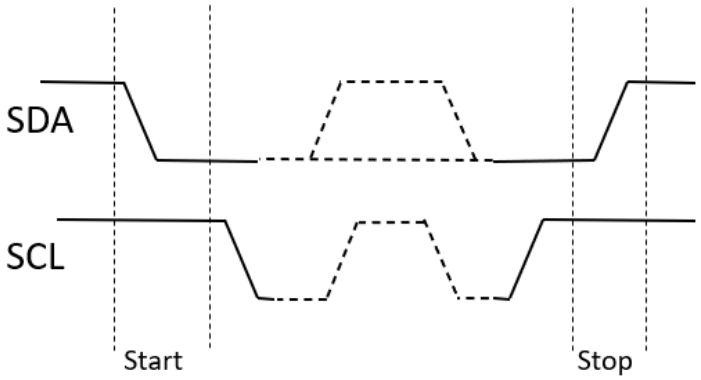
The device that initiates communication is termed the ‘master’. A device being addressed by the master is called a ‘slave’.

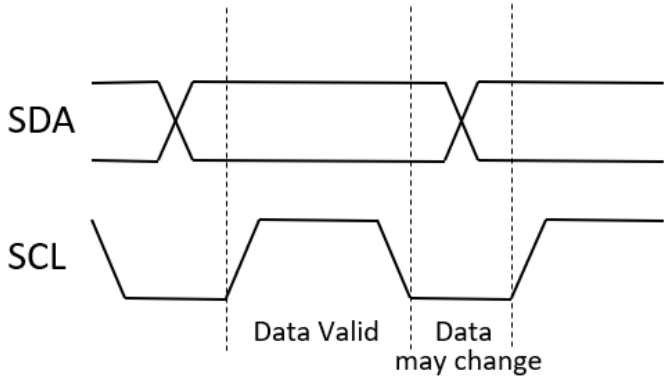
A data transfer is started by the master signalling a Start condition, followed by one or two bytes containing address and control information.

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.





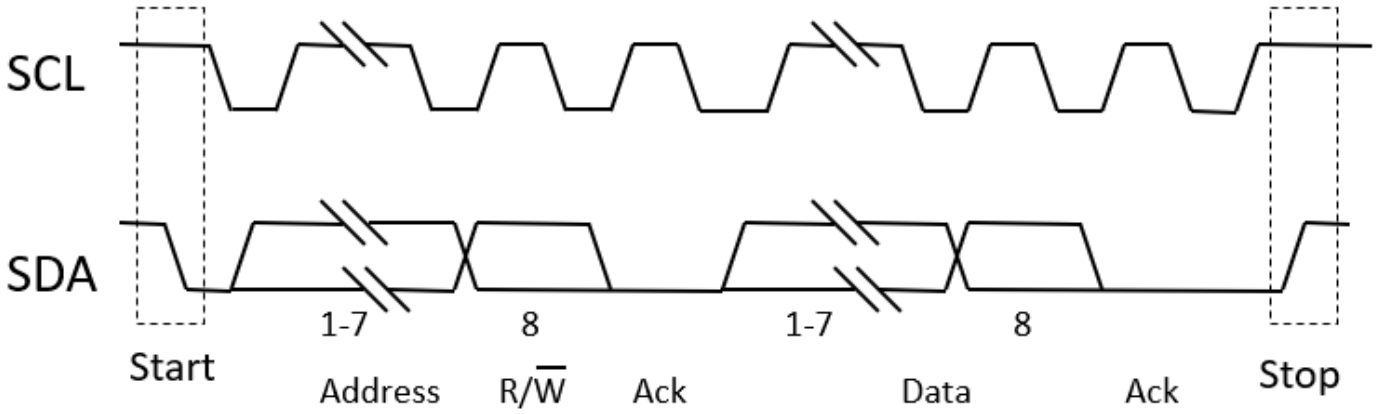
Simple I²C Communications

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/$\overline{\text{Write}}$).

All data transferred is in units of one byte, with no limit on the number of bytes transferred in one message.

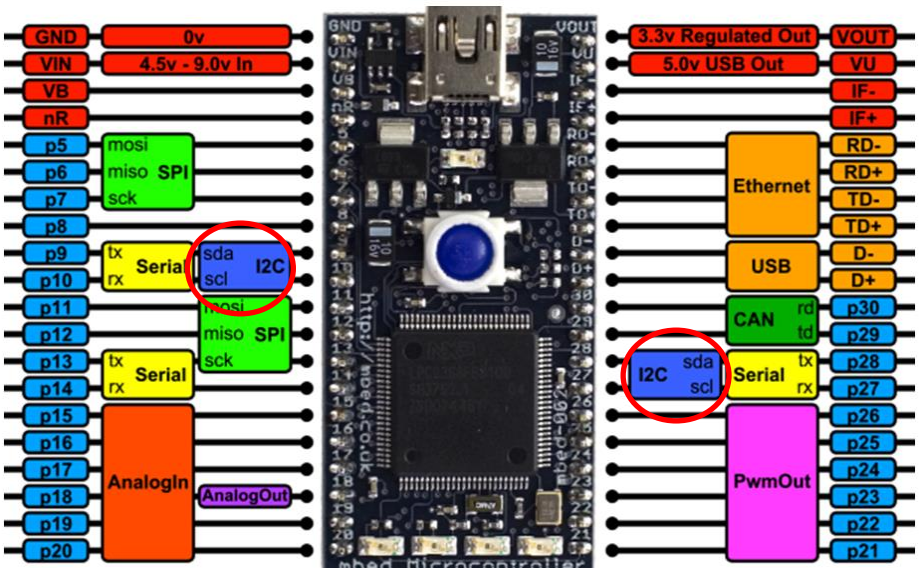
Each byte must be followed by a 1-bit acknowledge from the receiver, during which time the transmitter relinquishes SDA control.

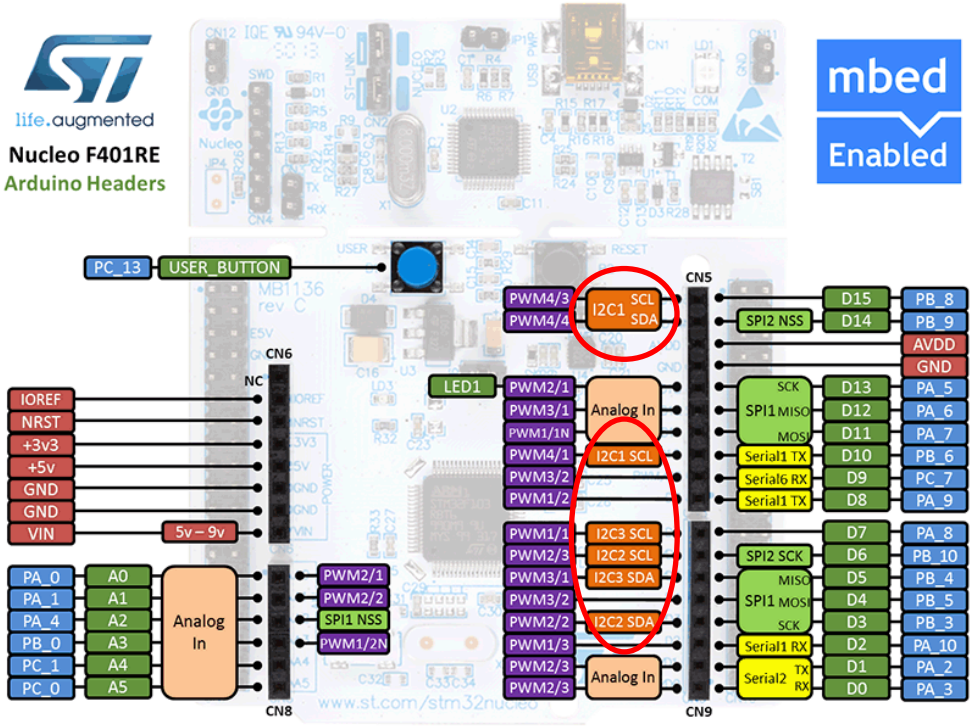


A Complete transfer of one byte

I²C in the Mbed Environment

The Mbed LPC1768 offers two I²C ports, while the Nucleo F401RE offers three.





I²C in the Mbed Environment: Master

Master API functions are shown here. These are more complex than SPI.

Functions	Usage
Constructor	
`I2C (PinName sda, PinName scl)`	Create an I²C Master interface, connected to the specified pins
Member Functions	
`void frequency (int Hz)`	Set the frequency of the I²C interface
`int read (int address, char *data, int length, bool repeated)`	Full read from I²C slave. The address lsb is forced to 1 to indicate a read; *data* is pointer to byte array to be sent, *length* is number of bytes, *repeated* = 1 for repeated start. Returns 0 for success.
`int read(int ack)`	Reads single byte from !2C slave; ack is 1 for acknowledge.
`int write (int data)`	Write single byte to I²C slave; returns 1 for ack. received, 0 for no ack., 2 for timeout.
`int write (int address, const char *data, int length, bool repeated)`	Write a complete message to I²C slave; *data* is pointer to byte array to be sent, *length* is number of bytes, *repeated* = 1 for repeated start; returns 0 for success.
`void start (void)`	Creates a start condition on the I²C bus
`void stop (void)`	Creates a stop condition on the I²C bus

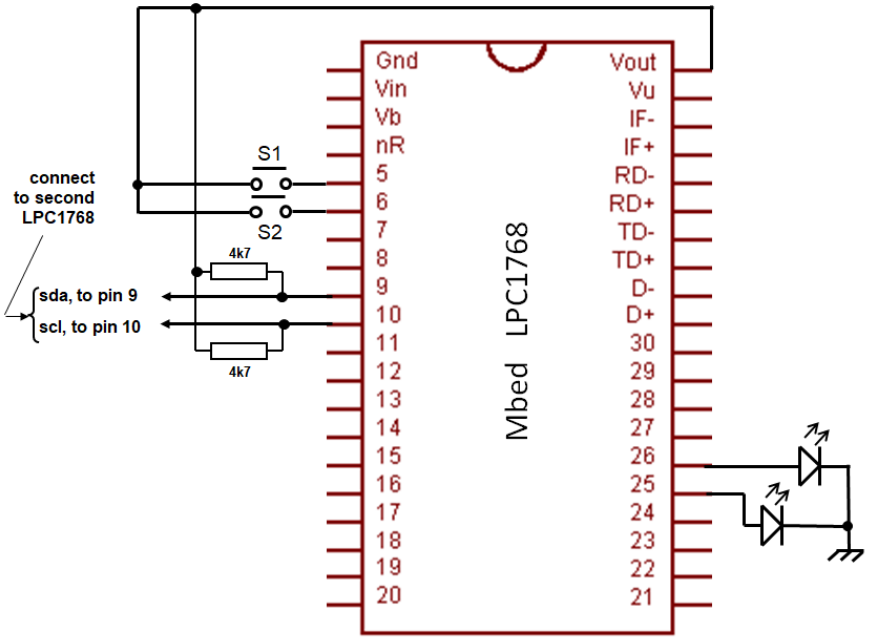
I²C in the Mbed Environment: Slave

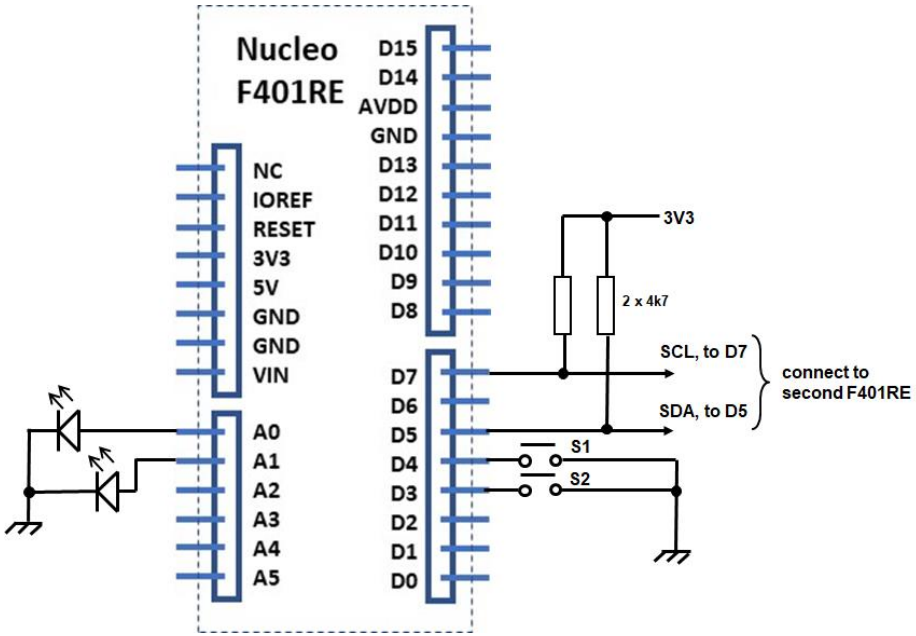
Slave API functions are shown here.

Function	Usage
Constructor	
I2CSlave(PinName sda, PinName scl)	Create an I²C Slave interface, connected to the specified pins.
Member Functions	
void **address** (int *address*)	Sets the I²C Slave address.
void **frequency** (int *Hz*)	Set the frequency of the I²C interface
int **read**(void)	Read single byte from I²C Master.
int **read** (char **data*, int *length*)	Read specified number of bytes from I²C Master; *data* points to buffer to read data into, *length* is number of bytes.
int **receive** (void)	Check if this I²C Slave has been addressed. Return value: NoData = 0 ReadAddressed = 1 WriteGeneral = 2 WriteAddressed = 3
int **write** (int *data*)	Write a single byte to I²C Master.
int **write**(const char **data*,int *length*)	Write specified number of bytes to I²C Master; *data* is pointer to buffer holding data to be sent, *length* is number of bytes; returns 0 for success.
void **stop**(void)	Reset the I²C slave 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.
Works with LPC1768 and F401RE, with pin allocations shown. Action for F401RE may be
incomplete, as this style of I2C implementation, with separate start, write, stop
statements, does not map ideally to F401RE I2C architecture.
                                                                              */
#include "mbed.h"

//Configure a serial port, naming sda, scl
I2C i2c_port(p9, p10);     //Use D5, D7 for F401RE
BusOut leds(p25, p26);     //Use A0, A1 for F401RE
DigitalOut error_light(LED1);
DigitalIn switch_ip1(p5,PullUp);  //Replace p5 with D3 for F401RE
DigitalIn switch_ip2(p6,PullUp);  //Replace p6 with D4 for F401RE

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


//Continued over…
```

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

```
int main() {
    char ack;
  while(true) {
    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
    ack = i2c_port.write(switch_word);  //send byte of data, and save ack value
    error_light = !ack;                //LED lights if no acknowledge
    i2c_port.stop();                   //force a stop condition
    wait_us(200);
    //request and 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(1);      //read and save the received byte
    i2c_port.stop();                  //force a stop condition
    leds = recd_val & 0x03;  //set leds according to incoming word from slave
  }
```

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 API 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 master, and
displays state of master's switches on its leds.
Applies to LPC1768 and F401RE, with pin allocations shown. See also note in master
program header.      */
#include "mbed.h"

I2CSlave slave(p9,p10);              //Use D5, D7 for F401RE
BusOut leds(p25,p26);                //Use A0, A1 for F401RE
DigitalIn switch_ip1(p5,PullUp);     //Replace p5 with D3 for F401RE
DigitalIn switch_ip2(p6,PullUp);     //Replace p6 with D4 for F401RE
char switch_word;                    //word we will send
char recd_val;                      //value received from Master
const int addr = 0x52;

//Continued over...
```

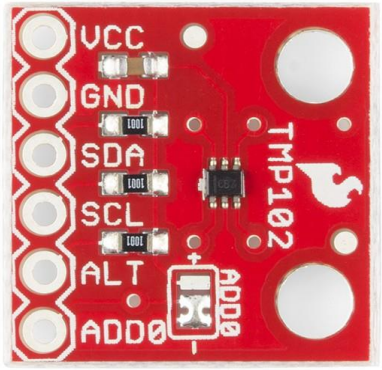
Setting up the I²C Data Link (Slave, cont.)

```
int main() {
  int how_addressed;
  slave.address(addr);
  while (true) {
    //set up switch_word from switches that are pressed
    switch_word = 0xa0;
    if (switch_ip1) switch_word = switch_word|0x01;
    if (switch_ip2) switch_word = switch_word|0x02;
    how_addressed = slave.receive();   //check if addressed, and if so, how.
    if (how_addressed == I2CSlave::ReadAddressed) {
      slave.write(switch_word);
      slave.stop();
    }
    if (how_addressed == I2CSlave::WriteAddressed) {
      recd_val = slave.read();
      slave.stop();
      leds = recd_val & 0x03;
    }
  }
}
```

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	Mbed LPC1768 Pin	F401RE Pin	Notes
VCC (3.3V)	40	+3V3	
SDA	9	D5	2.2 kΩ pull-up to 3.3 V
SCL	10	D7	2.2 kΩ pull-up to 3.3 V
GND (0V)	1	GND	
ALT (Alert)	1	GND	
ADD0	1	GND	Connect to — 8-bit 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.
Works for both LPC1768 and F401RE, with pin allocations shown.
                                                                        */
#include "mbed.h"

I2C tempsensor(p9, p10);      //Use D5, D7 for F401RE
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(true) {
    ThisThread::sleep_for(1s);             //wait for one second
    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
    printf("Temp = %.2f degC\n\r", temp);
  }
}
```

a) the main.cpp file

```
{
    "target_overrides":{
        "*": {
            "target.printf_lib": "std"
        }
    }
}
```

b) the mbed_app.json file

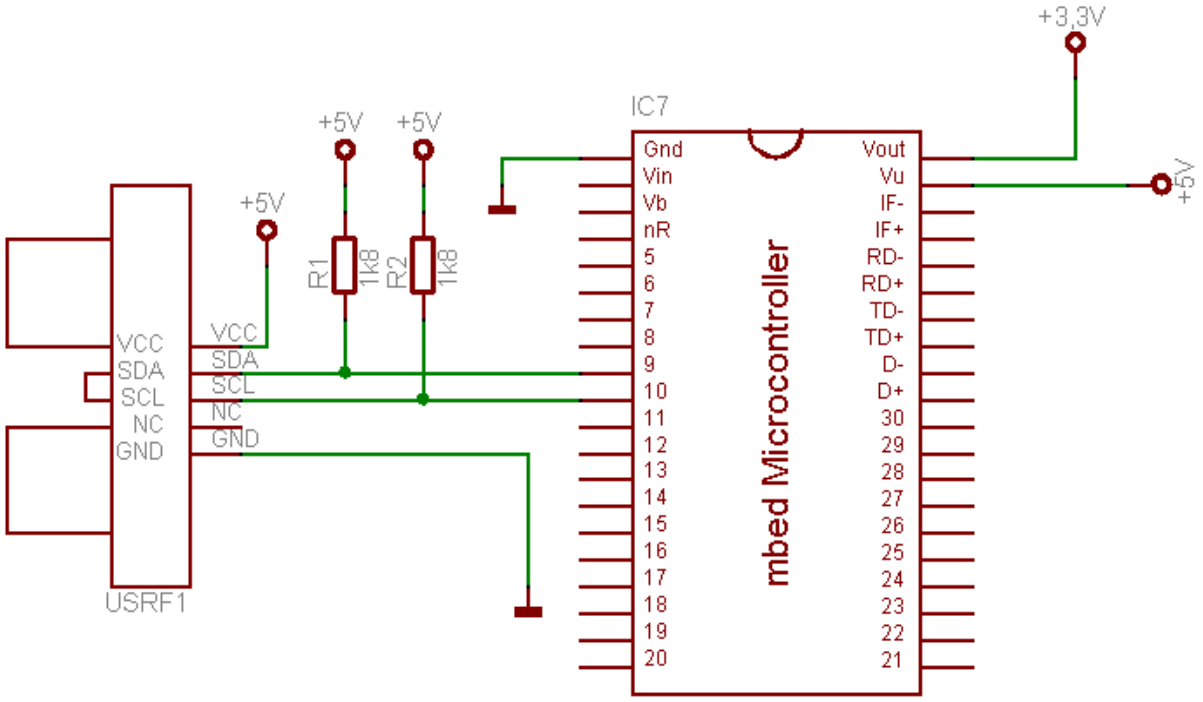
Using the SRF08 Ultrasonic Range Finder

The SRF08 ultrasonic range finder is used to measure the distance between the sensor and an acoustically reflective surface in front of it. It has an I^{2}C interface.

It transmits a pulse of ultrasound from one transducer, and then measures 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.





In the program over, 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.

Linking the SRF08 to an Mbed target

```
/*Program Example 7.8: Configures and takes readings from the SRF08 ultrasonic
range finder, and displays them on screen. A transmission error is indicated on
LED1.
Works for both LPC1768 and F401RE, with pin allocations shown.
                                                                        */
#include "mbed.h"

I2C rangefinder(p9, p10); //Use D5, D7 for F401RE
DigitalOut error_led(LED1);  //will indicate if error in I2C transmission
const int addr = 0xE0;       //SRF08 I2C address
char config_r[2];           //will hold config data
char range_read[2];         //will receive raw data from SRF08
float range;                //will hold calculated range value

int main() {
  int tr_error;         //will indicate transmission error (1 for error)
  while (true) {
    config_r[0] = 0x00;                     //set pointer reg to 'cmd register'
    config_r[1] = 0x51;                     //initialise, result in cm
    //send first message, and test for success
    tr_error = rangefinder.write(addr, config_r, 2);
    error_led = tr_error;        //light LED if transmission not successful.
    ThisThread::sleep_for(70ms); //wait for measurement to complete (around 65 ms)
    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]);
    printf("Range = %.2f cm\n\r", range);  //print range on screen
    ThisThread::sleep_for(500ms);        //set this further delay as desired
  }
}
```

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.

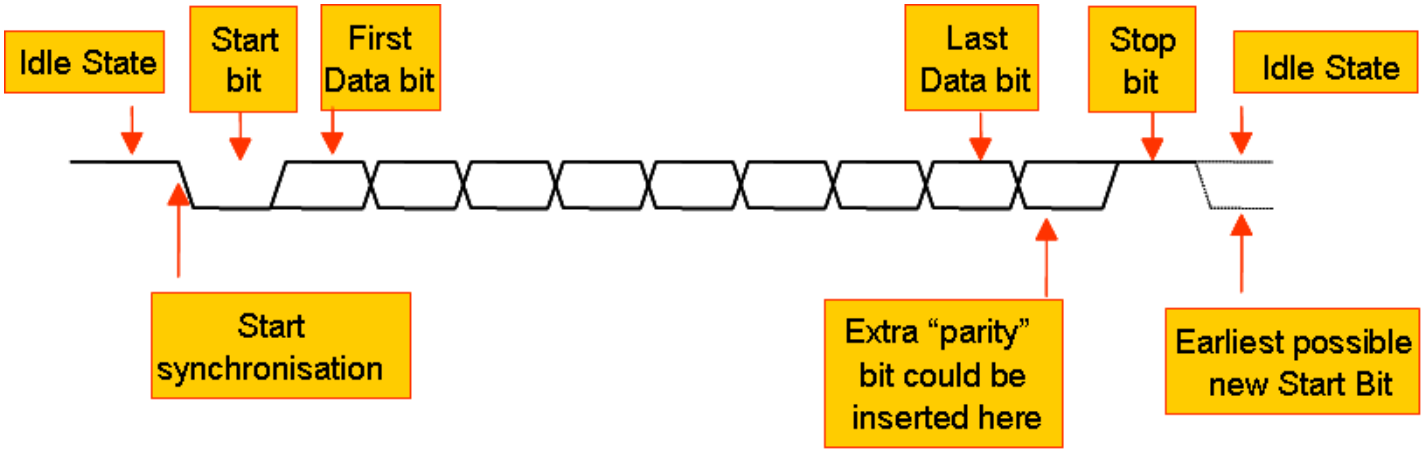
Asynchronous Serial Data Communication

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 be accepted.

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

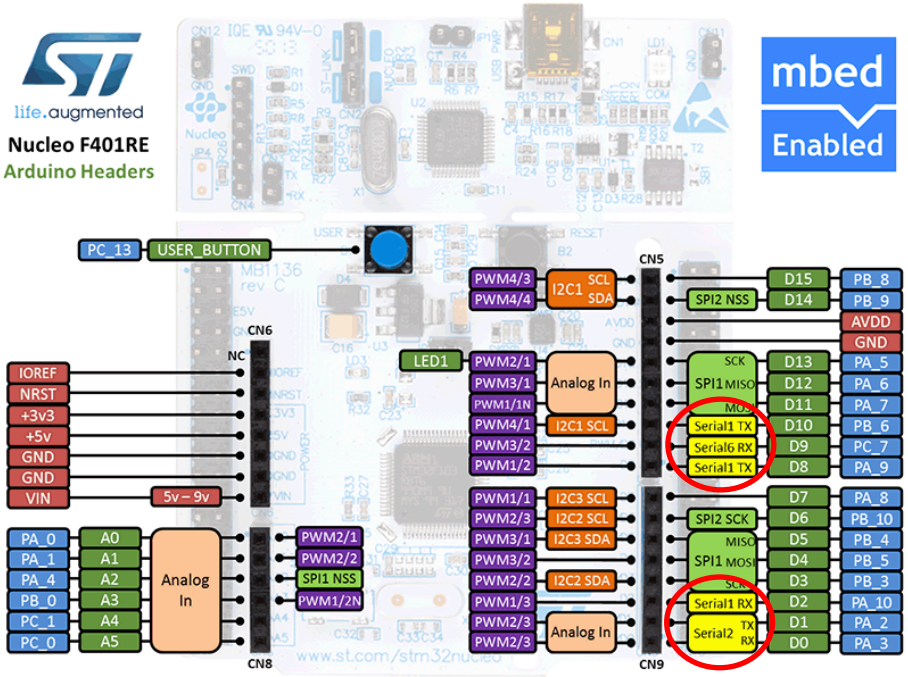
• An asynchronous serial port is generally called a UART, *Universal Asynchronous Receiver/ Transmitter.* A UART has one connection for transmitted data, called TX, and another for received data, called RX. The data rate that receiver and transmitter will operate at is specified by its *baud rate.*

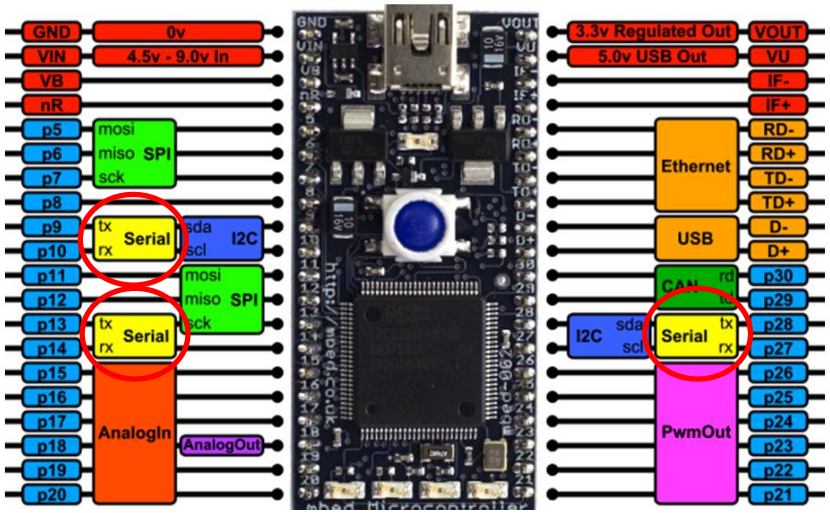


Asynchronous Communication in the Mbed Environment

The LPC1768 microcontroller has four UARTs. Three appear on the pinout of the Mbed LPC1768, labelled “Serial”.

The Nucleo F401RE appears a little more complex. It shows “Serial1”, “Serial2” and “Serial6”. The latter only has an RX pin, so can receive but not transmit. Serial2 is not available to the user without adjustment to on-board solder bridges. Therefore we do not use of this serial port.





Asynchronous Communication in the Mbed Environment

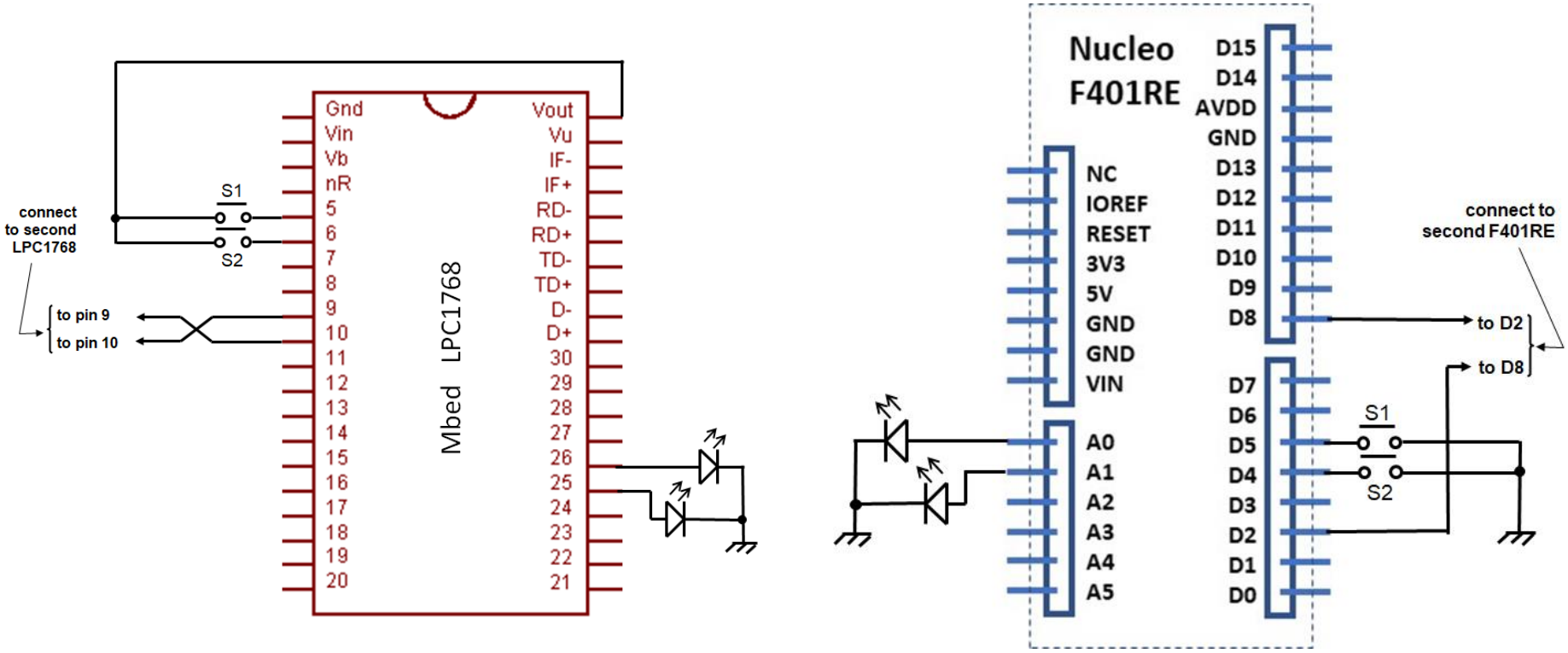
The Mbed OS has two classes to control the UARTs, buffered (**BufferedSerial**) and unbuffered (**UnbufferedSerial**). The former provides internal data buffering, both for data waiting to be transmitted and for data which has been received. The application code effectively writes to that buffer. We apply buffered serial here, with API summary below.

Functions	Usage
Constructor	
`BufferedSerial(PinName tx, PinName rx, int baud)`	Create a Serial port, connected to the specified transmit and receive pins
Member Functions	
`void set_baud(int baud)`	Set the baud rate of the serial port
`void set_format (int bits, Parity parity, int stop_bits)`	Set the transmission format used by the serial port
`bool readable()`	Determine if there is a character available to read, in which case returns true.
`bool writeable()`	Determine if there is space available to write a character, in which case returns true.
`ssize_t read(void *buffer, size_t length)`	Read the contents of a file into a buffer.
`ssize_t write(const void *buffer, size_t length)`	Write the contents of a buffer to a file.

Bidirectional Data Transfer between 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.
Works for both LPC1768 and F401RE, with pin allocations shown.        */

#include "mbed.h"

//set up TX and RX
BufferedSerial async_port(p9,p10);  //TX, RX. Use D8,D2 for F401RE
BusOut leds(p25,p26);               //LEDs. Use A0, A1 for F401RE
DigitalOut strobe(p7);              //a strobe for 'scope. Use D7 for F401RE
DigitalIn switch_ip1(p5,PullUp);    //Use D5 for F401RE
DigitalIn switch_ip2(p6,PullUp);    //Use D4 for F401RE

char switch_word ;                  //the word we will send
char recd_val;                      //the received value
char buf[4] = {0};            //declare 4-byte buffer and initialise to 0

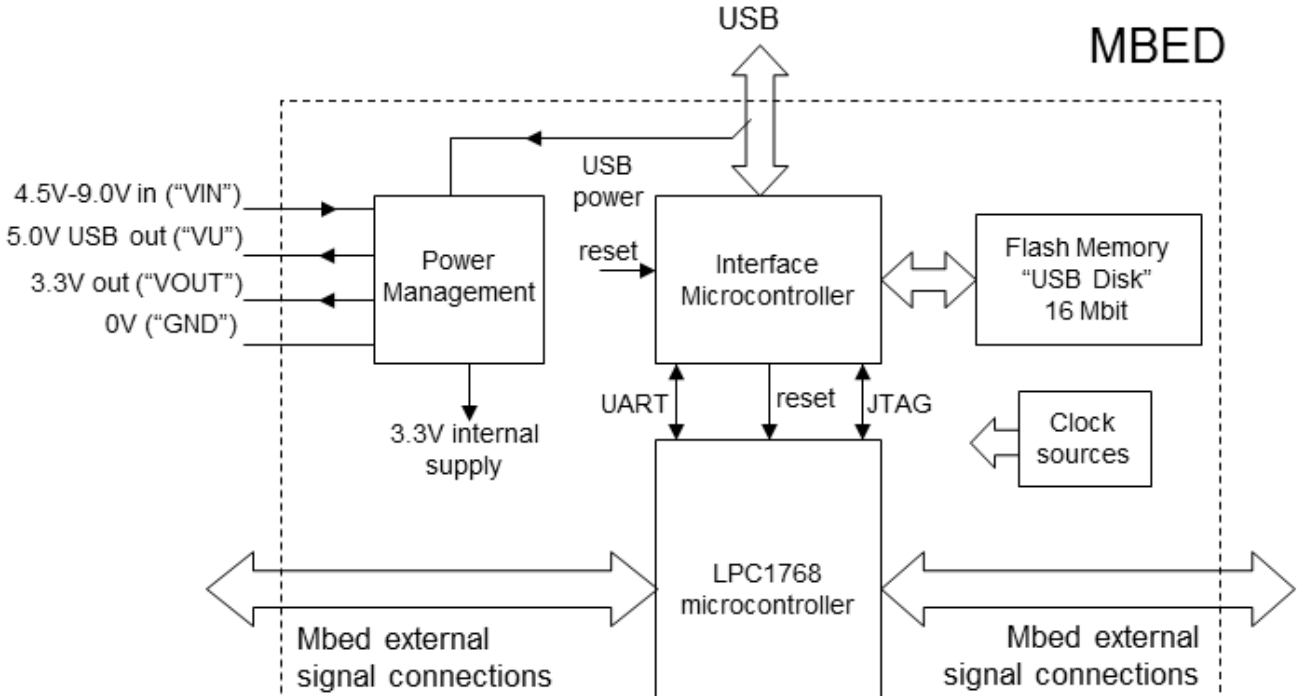
//Continued over…
```

Bidirectional Data Transfer between two Mbed UARTs

```
int main() {
  //accept default format, of 9600 Baud, 8 bits, no parity
  while (true){
    //Set up the word to be sent, by testing switch inputs
    switch_word = 0xa0;            //set up a recognisable output pattern
    if (switch_ip1) switch_word = switch_word|0x01;
    if (switch_ip2) switch_word = switch_word|0x02;
    buf[0] = switch_word;
    async_port.write(buf,1);       //transmit switch_word
    if (async_port.readable() == 1){     //is there a character to be read?
      async_port.read(buf,1);      //if yes, then read it
      recd_val = buf[0];
      leds = recd_val & 0x03;      //set leds according to incoming word
    }
    strobe=1;                      //short strobe pulse.
    wait_us(100);
    strobe=0;
    //At 9600 baud one character transmission requires 10x104us, ie approx. 1ms
    //Adjust loop length for ease of observation on 'scope, as required.
    ThisThread::sleep_for(2ms);
  }
}
```

Applying Asynchronous Communication with the Host Computer

Each Mbed target has one UART which is committed to communicating back to the USB link. This link can be accessed by the programmer using the identifiers **USBTX** and **USBRX**. The USB UART then acts just like any of the others, in terms of its use of the API. We've been using it already, but only via the **printf( )** function.



Applying Asynchronous Communication with the Host Computer

This Program Example demonstrates using the computer link, and explores using the buffer in Buffered Serial mode. The number of characters in the buffer which are written each time is controlled by the variable i. Once running, the program should lead to a display as shown; one extra character is printed on every line.

```
/*Program Example 7.10:
A simple demonstration of the serial buffer.
Runs on LPC1768 and F401RE without adjustment, no external connection*/

#include "mbed.h"

BufferedSerial serial_port(USBTX, USBRX);

char buf[14] = "\n\rHello World";    //declare buffer and load text message
int i;
int main() {
    for(i = 2;i < 15;i++){          //i starts at 2, to include control characters
    serial_port.write(buf,i);   //Write i bytes to serial port.
    }
}
```

```
H
He
Hel
Hell
Hello
Hello
Hello W
Hello Wo
Hello Wor
Hello Worl
Hello World
```

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 in the Mbed Environment

Both Nucleo F401RE and LPC1768 Mbed have a USB connection, which we use for powering, programming and other interaction with the host computer. In each case this links to an on-board interface microcontroller.

The LPC1768 microcontroller itself has a USB port. This connects to pins 31 and 32, labelled D+ and D- .

There are many USB APIs available for use, as shown here. Each has its own constructor and set of member functions.

Mbed USB library	Description
USBAudio	Configures the Mbed to be recognised as an audio interface allowing streaming audio to be read, output or analysed and processed.
USBCDC	Emulates a basic serial port over USB to send or receive data.
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
USBKeyboard	Configures the Mbed to emulate a USB keyboard
USBMIDI	Allows send and receive of MIDI messages in communication with a host PC using MIDI sequencer software
USBMouse	Configures the Mbed to emulate a USB mouse
USBMouseKeyboard	A USB mouse and keyboard feature set combined in a single library
USBSerial	Emulates an additional standard serial port on the Mbed, through the USB connections
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**.

```
/* Program Example 7.10: Emulating a USB mouse
Note this program cannot run on F401RE, as it has no USB port.  */

#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 (true) {
    for (int i = 0; i < 4; i++) {       // scroll through position co-ordinates
      mouse.move(dx[i],dy[i]);    // move mouse to co-ordinate
      ThisThread::sleep_for(200ms);
    }
  }
}
```

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. Mbed target boards generally have SPI ports, and the Mbed OS provides a supporting API.
- 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, and a master can detect if a slave fails to respond. 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. Mbed target boards generally have several UARTs, and reserve one for 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 APIs allowing it, for example, to operate as a mouse or keyboard, or as an audio or MIDI interface.
- The ST IoT Discovery Board uses I^2C extensively, to link its on-board sensors to the central microcontroller.