

Digital Input and Output

Chapter Outline

3.1 Starting to Program 29

- 3.1.1 Thinking about the First Program 30
- 3.1.2 Understanding the mbed API 32
- 3.1.3 Exploring the *while* Loop 33

3.2 Voltages as Logic Values 34

3.3 Digital Output on the mbed 36

- 3.3.1 Using LEDs 36
- 3.3.2 Using mbed External Pins 37

3.4 Using Digital Inputs 39

- 3.4.1 Connecting Switches to a Digital System 39
- 3.4.2 The DigitalIn API 40
- 3.4.3 Using *if* to Respond to a Switch Input 40

3.5 Interfacing Simple Opto Devices 42

- 3.5.1 Opto Reflective and Transmissive Sensors 42
- 3.5.2 Connecting an Opto-Sensor to the mbed 43
- 3.5.3 Seven-Segment Displays 45
- 3.5.4 Connecting a Seven-segment Display to the mbed 46

3.6 Switching Larger DC Loads 49

- 3.6.1 Applying Transistor Switching 49
- 3.6.2 Switching a Motor with the mbed 50
- 3.6.3 Switching Multiple Seven-Segment Displays 51

3.7 Mini-Project: Letter Counter 53

Chapter Review 53

Quiz 53

References 55

3.1 Starting to Program

This chapter will consider that most basic of microcontroller activity, the input and output of digital signals. Moving beyond this, it will show how we can make simple decisions within a program based on the value of a digital input. Figure 1.1 also predicted the importance of time in the embedded environment; no surprise therefore that at this early stage we meet timing activity as well.

You may also be embarking on C programming for the first time. This chapter takes you through quite a few of the key concepts. If you are new to C, we suggest you now read through Sections B1–B5 inclusive of Appendix B.

One thing that is not expected of you in reading this book is a deep knowledge of digital electronics. Some understanding of electronic theory will, however, be useful. If you need support in this area, you might like to have a book like Reference 3.1 available, to access when needed. There are many good websites in this area as well.

3.1.1 Thinking about the First Program

We now take a look at our first program, introduced as Program Example 2.1. This is shown again below for convenience, this time with comments inserted. Compare this with the original appearance of the program, shown in Chapter 2. Remember to check across to Appendix B when you need further C background.

Comments are text messages that you write to yourself within the program; they have no impact on the actual working of the program. It is good practice to introduce comments widely; they help your own thought process as you write, remind you what the program was meant to do as you look back over it, and help others to read and understand the program. This last point is essential for any sort of team-working, or if you are handing in work to be marked!

There are two ways of inserting comments into a program, and both are used in this example. One is to place the comment between the markers `/*` and `*/`. This is useful for a block of text information running over several lines. Alternatively, when two forward-slash symbols (`//`) are used, the compiler ignores any text that follows on that line only; this can then be used for comment.

The opening comment in the program, lasting three lines, gives a brief summary of what the program does. We adopt this practice in all subsequent programs in the book. Notice also that we have introduced some blank lines, to make the program a little more readable. We then put a number of in-line comments; these indicate what individual lines of code do.

```
/*Program Example 2.1: A program which flashes mbed LED1 on and off.
Demonstrating use of digital output and wait functions. Taken from the mbed site.
*/

#include "mbed.h" //include the mbed header file as part of this program
// program variable myled is created, and linked with mbed LED1
DigitalOut myled(LED1);

int main() { //the function starts here
    while(1) { //a continuous loop is created
        myled = 1; //switch the led on, by setting the output to logic 1
        wait(0.2); //wait 0.2 seconds
    }
}
```

```

    myled = 0;    //switch the led off
    wait(0.2);   //wait 0.2 seconds
}               //end of while loop
}               //end of main function

```

Program Example 2.1 Repeated (and commented) for convenience

C code feature

Let's now identify all the C programming features of [Program Example 2.1](#). The action of *any* C or C++ program is contained within its **main()** function, so that is always a good place to start looking. By the way, writing **main()** with those brackets after it reminds us that it is the name of a C function; all function names are written in this way in the book. The *function definition* — what goes on inside the function — is contained within the opening curly bracket or brace, appearing immediately after **main()**, and goes on until the closing brace. There are further pairs of braces inside these outer ones. Using pairs of braces like this is one of the ways that C groups blocks of code together and creates program structure.

C code feature

Many programs in embedded systems are made up of an endless loop, i.e. a program which just goes on and on repeating itself indefinitely. Here is our first example. We create the loop by using the **while** keyword; this controls the code within the pair of braces that follow. Section B.7 (in Appendix B) tells us that normally **while** is used to set up a loop, which repeats if a certain condition is satisfied. However, if we write **while (1)**, then we “trick” the **while** mechanism to repeat indefinitely.

C code feature

The real action of the program is contained in the four lines within the **while** loop. These are made up of two calls to the library function **wait()**, and two statements, in which the value of **myled** is changed. The **wait()** function is from the mbed library; further options are shown in [Table 3.1](#). The 0.2 parameter is in seconds, and defines the delay length caused by this function — another value can be chosen. In the two statements containing **myled** we make use of a C operator for the first time. Here, we use the *assign* operator, which applies the conventional equals sign. Note carefully that in C this operator means that a variable is set to the value indicated. Thus,

```
myled = 1;
```

means that the variable **myled** is set to the value 1, whatever its previous value was. Conventional “equals” is represented by a double equals sign, i.e. **==**. It is used later in this chapter. There are many C operators, and it is worth noticing and learning each one as you meet it for the first time. They are summarized in Section B.5.

Table 3.1: mbed library wait functions

C/C++ function	Action
<code>wait</code>	Waits for the number of seconds specified
<code>wait_ms</code>	Waits for the number of milliseconds specified
<code>wait_us</code>	Waits for the number of microseconds specified



The program starts by including the all-important mbed header file, which is the connecting pathway to all mbed library features. This means that the header file is literally inserted into the program, before compiling starts in earnest. The **#include** compiler directive is used for this (Section B.2.3). The program then applies the mbed application programming interface (API) utility **DigitalOut** to define a digital output, and gives it the name **myled**. Once declared, **myled** can be used as a variable within the program. The name LED1 is reserved by the API for the output associated with that light-emitting diode (LED) on the mbed board, as labeled in Figure 2.1b.

Notice that we indent the code by two spaces within **main()**, and then two more within the **while** code block. This has no effect on the action of the program, but does help to make it more readable and hence cut down on programming errors; it is a practice we apply for programs in this book. Check Section B.11 for other code layout practices that we adopt. Companies working with C often apply ‘house styles’, to ensure that programmers write code in a readable and consistent fashion.

Exercise 3.1

Get familiar with [Program Example 2.1](#), and the general process of compiling, by trying the following variations. Observe the effects.

1. Add the comments, or others of your own, given in the example. See that the program compiles and runs without change.
2. Vary the 0.2 parameter in the **wait()** functions.
3. Replace the **wait()** functions with **wait_ms()** functions, with accompanying parameter set to give the same wait time. Also vary this time.
4. Use LED2, 3 or 4 instead of 1.
5. Make a more complex light pattern using two or more LEDs, and more **wait()** function calls.

3.1.2 Understanding the mbed API

The mbed API has a pattern that we will see repeated many times, so it is important to get used to it. The library is made up of a set of utilities, which are all itemized in the mbed website Handbook. The first that we look at is **DigitalOut**, which we have already used. The API summary for this is given in [Table 3.2](#).

In a pattern that will become familiar, the **DigitalOut** API component creates a C++ *class*, called **DigitalOut**. This appears at the head of the table. The class then has a set of member functions, which are listed below. The first of these is a C++ *constructor*, which must have the same name as the class itself. This can be used to create C++ objects. By

Table 3.2: The mbed digital output API summary (from www.mbed.org)

Function	Usage
DigitalOut	Create a DigitalOut connected to the specified pin
write	Set the output, specified as 0 or 1 (int)
read	Return the output setting, represented as 0 or 1 (int)
operator=	A shorthand for write
operator int()	A shorthand for read

using the **DigitalOut** constructor, we can create C++ objects. In our first example we create the object **myled**. We can then write to it and read from it, using the functions **write()** and **read()**. These are member functions of the class, so their format would be **myled.write()**, and so on. Having created the **myled** object we can, however, invoke the API operator shorthand, mentioned in Table 3.2, which applies the assign operator = . Hence, when we write

```
myled = 1;
```

the variable value is then not only changed (normal C usage), but also written to the digital output. This replaces **myled.write(1)**; . We will find similar shorthands offered for all peripheral API groups in the mbed API.

3.1.3 Exploring the while Loop

Program Example 3.1 is the first “original” program in this book, although it builds directly on the previous example. Create a new program in the mbed compiler and copy this example into it.



Look first at the structure of the program, derived from the *three* uses of **while**; one of these is a **while (1)**, which we are used to, and two are conditional. These latter are based on the value of a variable **i**. In the first conditional use of **while**, the loop repeats as long as **i** is less than 10. You can see that **i** is incremented within the loop, to bring it up this value. In the second conditional loop, the value is decremented, and the loop repeats as long as **i** is greater than zero.



A very important new C feature seen in this program is the way that the variable **i** is *declared* at the beginning of **main()**. It is essential in C to declare the data type of any variable before it is used. The possible data types are given in Section B.3. In this example, **i** is declared as a character, effectively an 8-bit number. In the same line, its value is initialized to zero. There are also four new operators introduced: +, −, < and >. The use of these is the same as in conventional algebra, so should be immediately familiar.

```

/*Program Example 3.1: Demonstrates use of while loops. No external connection required
*/
#include "mbed.h"
DigitalOut myled(LED1);
DigitalOut yourled(LED4);

int main() {
    char i=0;           //declare variable i, and set to 0
    while(1){           //start endless loop
        while(i<10) {   //start first conditional while loop
            myled = 1;
            wait(0.2);
            myled = 0;
            wait(0.2);
            i = i+1;      //increment i
        }               //end of first conditional while loop
        while(i>0) {     //start second conditional loop
            yourled = 1;
            wait(0.2);
            yourled = 0;
            wait(0.2);
            i = i-1;
        }
    }                   //end infinite loop block
}                       //end of main

```

Program Example 3.1 Using *while*

Having compiled [Program Example 3.1](#), download it to the mbed, and run it. You should see LED1 and LED4 flashing 10 times in turn. Make sure you understand why this is so.

Exercise 3.2

C code
feature

1. A slightly more elegant way of incrementing or decrementing a variable is by using the increment and decrement operators seen in Section B.5. Try replacing
 $i = i + 1;$ and $i = i - 1;$
with $i++;$ and $i--;$ respectively,
and run the program again.
2. Change the program so that the LEDs flash only five times each.
3. Try replacing the `myled = 1;` statement with `myled.write(1);`

3.2 Voltages as Logic Values

Computers deal with binary numbers made up of lots of binary digits, or bits, and each one of these bits takes a logic value of 0 or 1. Now that we are starting to use the mbed in earnest, it is

worth thinking about how those logic values are actually represented inside the mbed's electronic circuit, and at its connection pins.

In any digital circuit, logic values are represented as electrical voltages. Here now is the *big* benefit of digital electronics: we do not need a precise voltage to represent a logical value. Instead, we accept a *range* of voltages as representing a logic value. This means that a voltage can pick up some noise or distortion and still be interpreted as the right logic value. The microcontroller we are using in the mbed, the LPC1768, is powered from 3.3 V. We can find out which range of voltages it accepts as Logic 0, and which as Logic 1, by looking at its technical data. This is found in Reference 2.4 (Chapter 2), with important points summarized in Appendix C. (There will be moments when it will be useful to look at this appendix, but do not rush to it now.) This data shows that, for most digital inputs, the LPC1678 interprets *any* input voltage below 1.0 V (specified as 0.3×3.3 V) as Logic 0, and *any* input voltage above 2.3 V (specified as 0.7×3.3 V) as Logic 1. This idea is represented in the diagram of Figure 3.1.

If we want to input a signal to the mbed, hoping that it is interpreted correctly as a logic value, then it will need to satisfy the requirements of Figure 3.1. If we are outputting a signal from the mbed, then we can expect it to comply with the same figure. The mbed will normally output Logic 0 as 0 V and Logic 1 as 3.3 V, as long as no electrical current is flowing. If current is flowing, for example into an LED, then we can expect some change in output voltage. We will return to this point. The neat thing is, when we output a logic value, we are also getting a predictable voltage to make use of. We can use this to light LEDs (light emitting diodes), switch on motors, or many other things.

For many applications in this book we do not worry much about these voltages, as the circuits we build meet the requirements of Figure 3.1. There are situations, however, where it is necessary to take some care over logic voltage values, and these are mentioned as they come up.

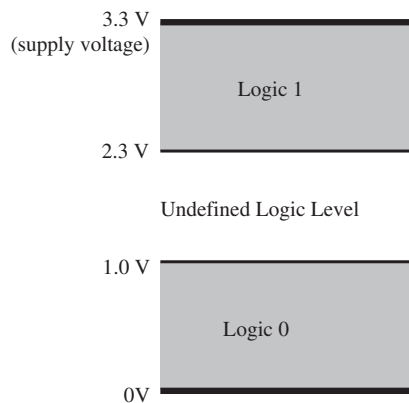


Figure 3.1:
Input logic levels for the mbed

3.3 Digital Output on the mbed

In our first two programs we have just switched the diagnostic LEDs which are fitted on the mbed board. The mbed, however, has 26 digital input/output (I/O) pins (pins 5–30), which can be configured as either digital inputs or outputs. These are shown in Figure 3.2. A comparison of this figure with the master mbed pinout diagram, i.e. Figure 2.1c, shows that these pins are multi-function. While we can use any of them for simple digital I/O, they all have secondary functions, connecting with the microcontroller peripherals. It is up to the programmer to specify, within the program, how they are configured.

3.3.1 Using LEDs

For the first time we are connecting an external device to the mbed. While you may not be an electronics specialist it is important — whenever you connect anything to the mbed — that you have some understanding of what you are doing. The LED is a semiconductor diode, and behaves electrically as one. It will conduct current in one direction, sometimes called the ‘forward’ direction, but not the other. What makes it so useful is that when it is connected so that it conducts, it emits photons from its semiconductor junction. The LED has the voltage/current characteristic shown in Figure 3.3a. A small forward voltage will cause very little current to flow. As the voltage increases there comes a point where the current suddenly starts flowing rather rapidly. For most LEDs this voltage is in the range shown, typically around 1.8 V.

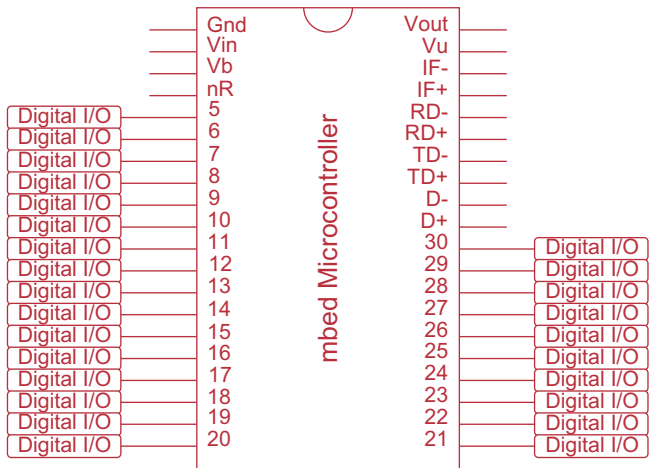


Figure 3.2:
The mbed digital I/O

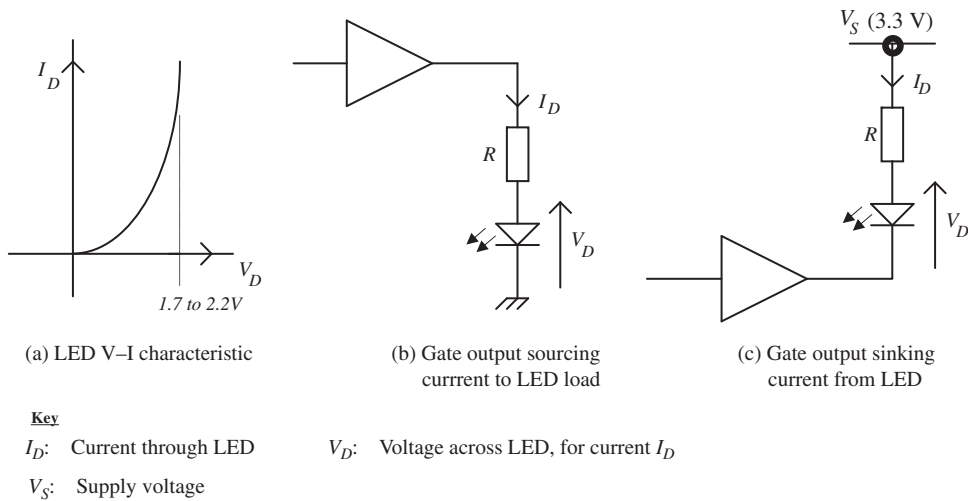


Figure 3.3:
Driving LEDs from logic gates

Figures 3.3b and c show circuits used to make direct connections of LEDs to the output of logic gates, for example an mbed pin configured as an output. The gate is here shown as a logic buffer (the triangular symbol). If the connection of Figure 3.3b is made, the LED lights when the gate is at logic high. Current then flows out of the gate into the LED. Alternatively, the circuit of Figure 3.3c can be made; the LED now lights when the logic gate is at logic zero, with current flowing *into* the gate. Usually a current-limiting resistor needs to be connected in series with the LED, to control how much current flows. The exception to this is if the combination of output voltage and gate internal resistance is such that the current is limited to a suitable value anyway. Some LEDs, such as the ones recommended for the next program, have the series resistance built into them. They therefore do not need any external resistor connected.

3.3.2 Using mbed External Pins

The digital I/O pins are named and configured to output using **DigitalOut**, just as we did in the earlier programs, by defining them at the start of the program code, for example:

```
DigitalOut myname(p5);
```

The **DigitalOut** object created in this way can be used to set the state of the output pin, and also to read back its current state.

We will be applying the circuit of Figure 3.4a. Appendix D gives example part numbers for all parts used; equivalent devices can of course be implemented. Within this circuit we are applying the connection of Figure 3.3b. The particular LED recommended in Appendix D, however, has an internal series resistor, of value around 240 Ω , so an external one is not required.

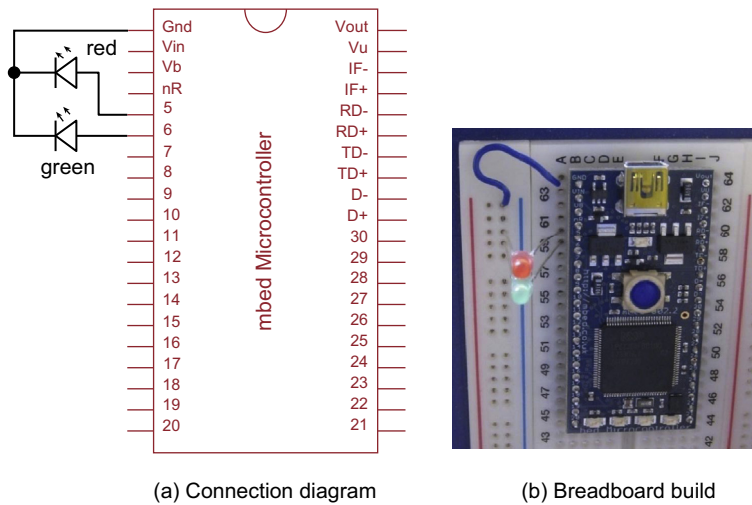


Figure 3.4:
mbed set-up for simple LED flashing

Place the mbed in a breadboard, as seen in [Figure 3.4b](#), and connect the circuit shown. The mbed has a common ground on pin 1. Connect this across to one of the outer rows of connections of the breadboard, as seen in [Figure 3.4b](#). You should adopt this as a habit for all similar circuits that you build. Remember to attach the anode of the LEDs (the side with the longer leg) to the mbed pins. The negative side (cathode) should be connected to ground. For this and many circuits we will take power from the universal serial bus (USB).

Create a new program in the mbed compiler, and copy across [Program Example 3.2](#).

```
/*Program Example 3.2: Flashes red and green LEDs in simple time-based pattern
*/
#include "mbed.h"
DigitalOut redled(p5); //define and name a digital output on pin 5
DigitalOut greenled(p6); //define and name a digital output on pin 6
int main() {
    while(1) {
        redled = 1;
        greenled = 0;
        wait(0.2);
        redled = 0;
        greenled = 1;
        wait(0.2);
    }
}
```

Program Example 3.2 Flashing external LEDs

Compile, download and run the code on the mbed. The code extends ideas already applied in [Program Examples 2.1](#) and [3.1](#), so it should be easy to understand that the green

and red LEDs are programmed to flash alternately. You should see this happen when the code runs.

■ Exercise 3.3

Using any digital output pin, write a program which outputs a square wave, by switching the output repeatedly between Logic 1 and 0. Use `wait()` functions to give a frequency of 100 Hz (i.e. a period of 10 ms). View the output on an oscilloscope. Measure the voltage values for Logic 0 and 1. How do they relate to Figure 3.1? Does the square wave frequency agree with your programmed value?

3.4 Using Digital Inputs

3.4.1 Connecting Switches to a Digital System

Ordinary electromechanical switches can be used to create logic levels, which will satisfy the logic level requirements seen in Figure 3.1. Three commonly used ways are shown in Figure 3.5. The simplest, Figure 3.5a, uses an SPDT (single-pole, double-throw) switch. This is what we will use in the next example. A resistor in series with the logic input is sometimes included in this circuit, as a precautionary measure. However, an SPST (single-pole, single-throw) switch can be lower cost and smaller, and so is very widely used. They are connected with a pull-up or pull-down resistor to the supply voltage, as shown in Figure 3.5b or c. When the switch is open, the logic level is defined by the connection through the resistor. When it is closed, the switch asserts the other logic state. A wasted electrical current then flows through the resistor. This is kept small by having a high-value resistor. On the mbed, as with many microcontrollers, pull-up and pull-down resistors are available within the microcontroller, saving the need to make the external connection.

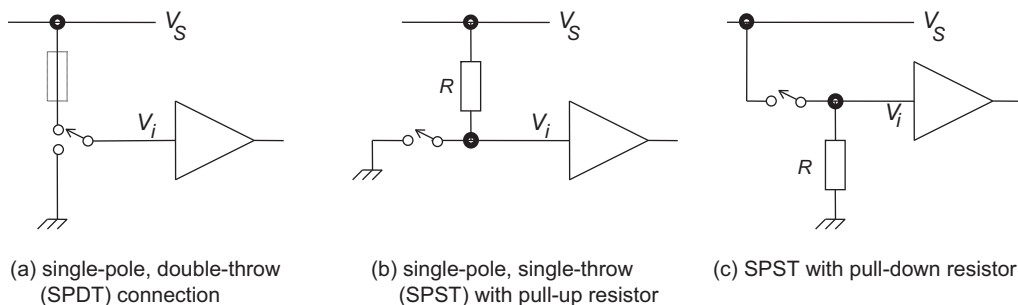


Figure 3.5:
Connecting switches to logic inputs

Table 3.3: The mbed digital input API summary (from www.mbed.org)

Function	Usage
DigitalIn	Create a DigitalIn connected to the specified pin
read	Read the input, represented as 0 or 1 (int)
mode	Set the input pin mode
operator int()	An operator shorthand for read()

3.4.2 The DigitalIn API

The mbed API has the digital input functions listed in Table 3.3, with format identical to that of **DigitalOut**, which we have already seen. This section of the API creates a class called **DigitalIn**, with the member functions shown. The **DigitalIn** constructor can be used to create digital inputs, and the **read()** function used to read the logical value of the input. In practice, the shorthand offered allows us to read input values through use of the digital object name. This will be seen in the program example that follows. As with digital outputs, the same 26 pins (pins 5–30) can be configured as digital inputs. Input voltages will be interpreted according to Figure 3.1. Note that use of **DigitalIn** enables by default the internal pull-down resistor, i.e. the input circuit is configured as Figure 3.5c. This can be disabled, or an internal pull-up enabled, using the **mode()** function. See the mbed Handbook for details on this.

3.4.3 Using if to Respond to a Switch Input

We will now connect to our circuit a digital input, a switch, and use it to control LED states. In so doing, we make a significant step forward. For the first time we will have a program that makes a decision based on an external variable, the switch position. This is the essence of many embedded systems. Program Example 3.3 is applied to achieve this; the digital input is connected to pin 7, and created with the **DigitalIn** constructor.



To make the decision within the program, we use the statement

```
if(switchinput==1).
```

This line sees use of the C equal operator, `==`, for the first time. Read Section B.6.1 to review use of the **if** and **else** keywords. The line causes the line or block of code which follows it to execute, if the specified condition is met. In this case, the condition is that the variable **switchinput** is equal to 1. If the condition is not satisfied, then the code that follows **else** is executed. Looking now at the program we can see that if the switch gives a value of Logic 1, the green LED is switched off and the red LED is programmed to flash. If the switch input is 0, the **else** code block is invoked and the roles of the LEDs are reversed. Again, the **while (1)** statement is used to create an overall infinite loop, so the LEDs flash continuously.

```

/*Program Example 3.3: Flashes one of two LEDs, depending on the state of a 2-way switch
*/
#include "mbed.h"
DigitalOut redled(p5);
DigitalOut greenled(p6);
DigitalIn switchinput(p7);
int main() {
    while(1) {
        if (switchinput==1) {    //test value of switchinput
            //execute following block if switchinput is 1
            greenled = 0;        //green led is off
            redled = 1;          // flash red led
            wait(0.2);
            redled = 0;
            wait(0.2);
        }                        //end of if
        else {                  //here if switchinput is 0
            redled = 0;          //red led is off
            greenled = 1;        // flash green led
            wait(0.2);
            greenled = 0;
            wait(0.2);
        }                        //end of else
    }                            //end of while(1)
}                                //end of main

```

Program Example 3.3 Using *if* and *else* to respond to external switch

Adjust the circuit of [Figure 3.4](#) to that of [Figure 3.6a](#), by adding an SPDT switch as shown. The input it connects to is configured in the program as a digital input. The photograph in [Figure 3.6b](#) shows the addition of the switch. In this case, wires have been soldered to the switch and connected to the breadboard. It is also possible to find switches that plug in directly. Create a new program and copy across [Program Example 3.3](#). Compile, download and run the code on the mbed.

■ Exercise 3.4

Write a program that creates a square wave output, as in [Exercise 3.3](#). Include now two possible frequencies, 100 Hz and 200 Hz, depending on an input switch position. Use the same switch connection you have just used. Observe the output on an oscilloscope.

■ Exercise 3.5

The circuit of [Figure 3.6](#) uses an SPDT switch connected as shown in [Figure 3.5a](#). However, the mbed Handbook tells us that the **DigitalIn** utility actually configures an on-chip pull-down resistor. Reconfigure the circuit using [Figure 3.5c](#), using either the same toggle switch or an SPST push button. Test that the program runs correctly.

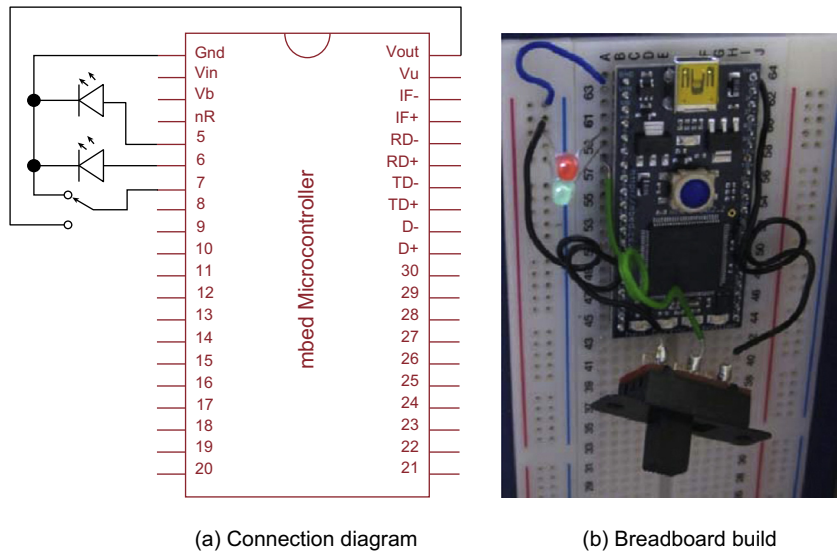


Figure 3.6:
Controlling the LED with a switch

3.5 Interfacing Simple Opto Devices

Now that we have the ability to input and output single bits of data, a wide range of possibilities opens up. Many simple sensors can interface directly with digital inputs. Others have their own designed-in interfaces, which produce a digital output. This section looks at some simple and traditional sensors and displays that can be interfaced directly to the mbed. In later chapters this is taken further, connecting to some very new and hi-tech devices.

3.5.1 Opto Reflective and Transmissive Sensors

Opto-sensors, such as seen in Figure 3.7a and b, are simple examples of sensors with ‘almost’ digital outputs. When a light falls on the base of an opto-transistor, it conducts; when there is no light it does not. In the reflective sensor (Figure 3.7a), an infrared LED is mounted in the same package as an opto-transistor. When a reflective surface is placed in front, the light bounces back, and the transistor can conduct. In the transmissive sensor (Figure 3.7b), the LED is mounted opposite the transistor. With nothing in the way, light from the LED falls directly on the transistor, and so it can conduct. When something comes in the way the light is blocked, and the transistor stops conducting. This sensor is also sometimes called a slotted opto-sensor or photo-interrupter. Each sensor can be used to detect certain types of object.

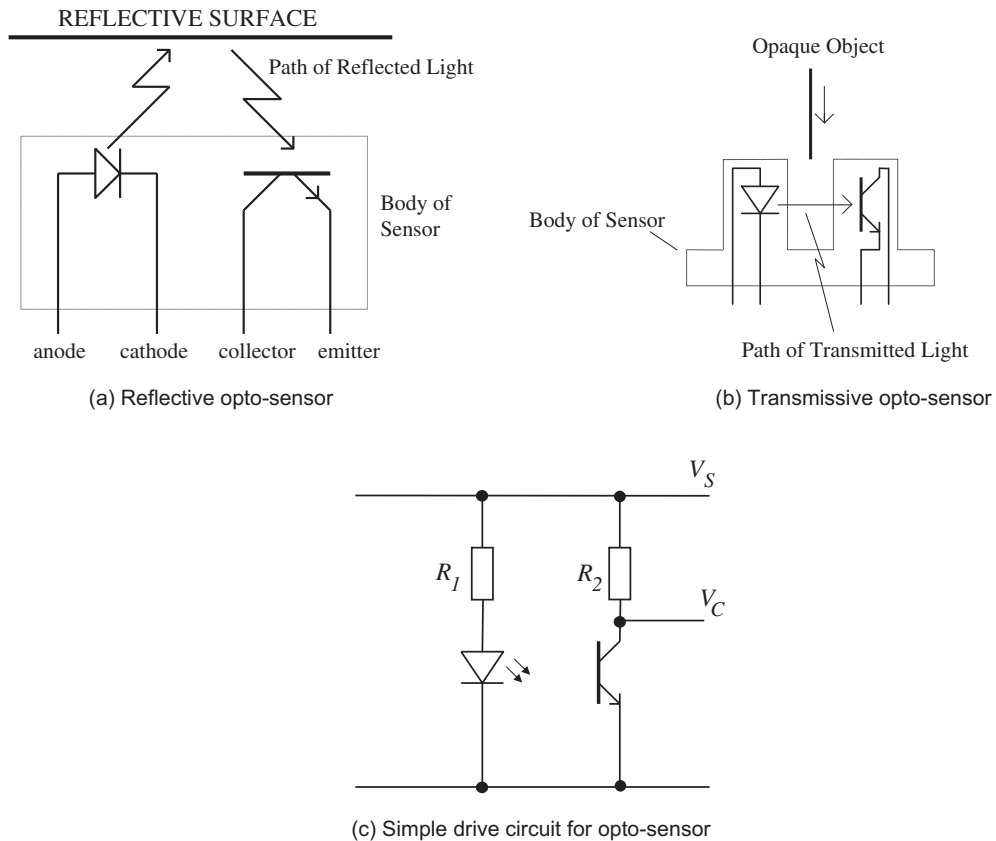


Figure 3.7:
Simple opto-sensors

Either of the sensors shown can be connected in the circuit of Figure 3.7c. Here, R_1 is calculated to control the current flowing in the LED, taking suitable values from the sensor datasheet. The resistor R_2 is chosen to allow a suitable output voltage swing, invoking the voltage thresholds indicated in Figure 3.1. When light falls on the transistor base, current can flow through the transistor. The value of R_2 is chosen so that with that current flowing, the transistor collector voltage V_C falls almost to 0 V. If no current flows, then V_C rises to V_S . In general, the sensor is made more sensitive by either decreasing R_1 or increasing R_2 .

3.5.2 Connecting an Opto-Sensor to the mbed

Figure 3.8 shows how a transmissive opto-sensor can be connected to an mbed. The device used is a KTIR0621DS, made by Kingbright; similar devices can be used.

The particular sensor used has pins that can plug directly into the breadboard. Take care

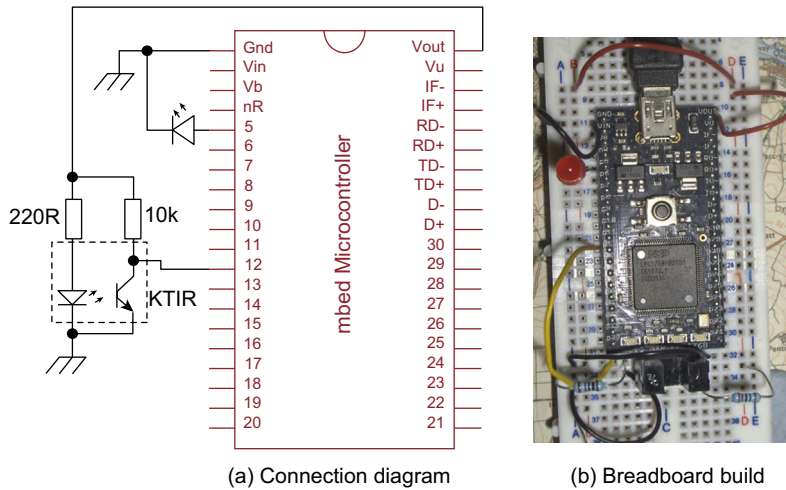


Figure 3.8:
mbed connected to a transmissive opto-sensor

when connecting these four pins. Connections are indicated on its housing; alternatively, you can look these up in the data that Kingbright provides. This can be obtained from the Kingbright website, Reference 3.2, or from the supplier you use to buy the sensor.

Program Example 3.4 controls this circuit. The output of the sensor is connected to pin 12, which is configured in the program as a digital input. When there is no object in the sensor, then light falls on the photo-transistor. It conducts, and the sensor output is at Logic 0. When the beam is interrupted, then the output is at Logic 1. The program therefore switches the LED on when the beam is interrupted, i.e. an object has been sensed. To make the selection, we use the **if** and **else** keywords, as in the previous example. Now, however, there is just one line of code to be executed for either state. It is not necessary to use braces to contain these single lines.

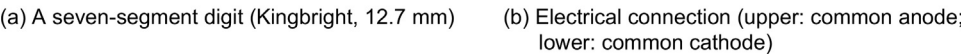
*/*Program Example 3.4: Simple program to test KTIR slotted optosensor. Switches an LED according to state of sensor
/

```
#include "mbed.h"
DigitalOut redled(p5);
DigitalIn opto_switch(p12);

int main() {
    while(1) {
        if (opto_switch==1)           //input = 1 if beam interrupted
            redled = 1;               //switch led on if beam interrupted
        else
            redled = 0;               //led off if no interruption
    }
}
```

Program Example 3.4 Applying the photo-interrupter

The seven-segment display is a particularly versatile configuration. An example single digit, made by Kingbright (Reference 3.2), is shown in Figure 3.9. By lighting different combinations of the seven segments, all numerical digits can be displayed, as well as a surprising number of alphabetic characters. A decimal point is usually included, as shown. This means that there are eight LEDs in the display, needing 16 connections. To simplify matters, either all LED anodes are connected together, or all LED cathodes. This is seen in Figure 3.9b; the two possible connection patterns are called *common cathode* or *common anode*. Now instead of 16 connections being needed, there are only nine, one for each LED and one for the common connection. The actual pin connections in the example shown lie in two rows, at the top and bottom of the digit. There are 10 pins in all, with the common anode or cathode taking two pins.



The seven-segment display. (Image reproduced with permission of Kingbright Elec. Co. Ltd.)

A small seven-segment display as seen in Figure 3.9 can be driven directly from a microcontroller. In the case of common cathode, the cathode is connected to ground, and each segment is connected to a port pin. If the segments are connected in this sequence to form a byte,

$$(MSB) \text{ DP g f e d c b a (LSB)}$$


then the values shown in Table 3.4 apply. For example, if 0 is to be displayed, then all outer segments, i.e. abcdef, must be lit, with the corresponding bits from the microcontroller set to 1. If 1 is to be displayed, then only segments b and c need to be lit. Note that larger displays have several LEDs connected in series, for each segment. In this case, a higher voltage is needed to drive each series combination and, depending on the supply voltage of the microcontroller, it may not be possible to drive the display directly.

3.5.4 Connecting a Seven-segment Display to the mbed

As we know, the mbed runs from a 3.3 V supply. The datasheet of our display (accessed from Reference 3.2), shows that each LED requires around 1.8 V across it in order to light. This is within the mbed capability. If there were two LEDs in series in each segment however, the mbed would barely be able to switch them into conduction. But can we just connect the mbed output directly to the segment, or do we need a current-limiting resistor, as seen in Figure 3.3b? Looking at the LPC1768 data summarized in Appendix C, we see that the output voltage of a port pin drops around 0.4 V, when 4 mA is flowing. This implies an output resistance of 100 Ω. Let's not get into the electronics of all of this; suffice it to say, this value is approximate, and only applies in this region of operation. Applying Ohm's law, the current flow in an LED connected directly to a port pin of this type is given by

$$I_D \cong (3.3 - 1.8)/100 = 15 \text{ mA} \quad (3.1)$$

Table 3.4: Example seven-segment display control values

Display value	0	1	2	3	4	5	6	7	8	9
Segment drive (B)										
(MSB)	0011	0000	0101	0100	0110	0110	0111	0000	0111	0110
(LSB)	1111	0110	1011	1111	0110	1101	1101	0111	1111	1111
B (hex)	0x3F	0x06	0x5B	0x4F	0x66	0x6D	0x7D	0x07	0x7F	0x6F
Actual display										

This current, 15 mA, will light the segments very brightly, but is acceptable. It can be reduced, for example for power-conscious applications, by inserting a resistor in series with each segment.

Connect a seven-segment display to an mbed, using the circuit of Figure 3.10. In this simple application the common cathode is connected direct to ground, and each segment is connected to one mbed output.

The circuit can be driven by Program Example 3.5. This first of all applies the **BusOut** mbed API class. **BusOut** allows you to group a set of digital outputs into one bus, so that you can write a digital word direct to it. The equivalent input is the **Busin** object, though that is not used here. Applying **BusOut** simply requires you to specify a name, in this case **display**, and then list in brackets the pins which will be members of that bus.



Within this program we see for the first time a **for** loop. This is an alternative to **while**, for creating a conditional loop. Review its format in Section B.7.2.

In this example, the variable **i** is initially set to 0, and on each iteration of the loop it is incremented by 1. The new value of **i** is applied within the loop. When **i** reaches 4, the

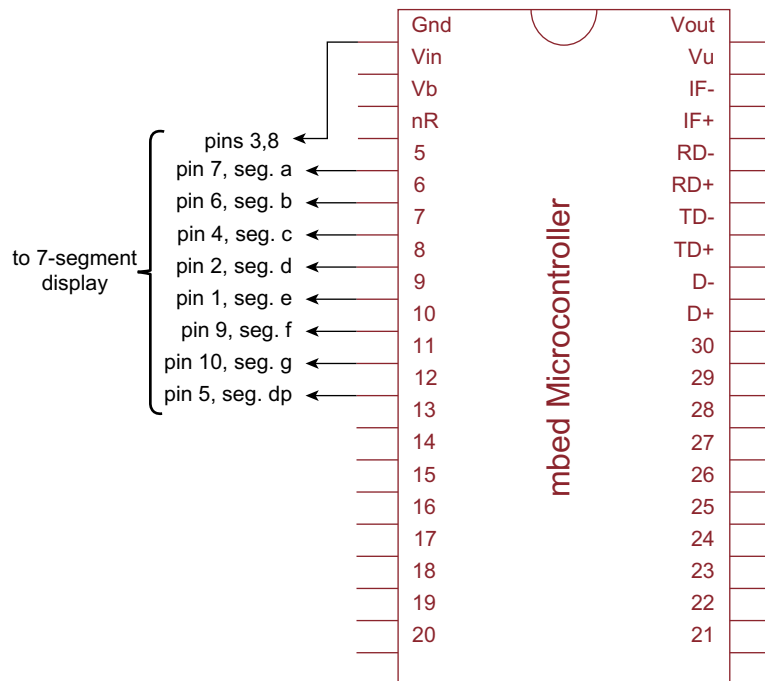


Figure 3.10:

The mbed connected to a common cathode seven-segment display

loop terminates. However, as the **for** loop is the only code within the endless **while** loop, it simply starts again.

C code feature

The program goes on to apply the **switch**, **case** and **break** keywords. Used together, these words provide a mechanism to allow one item to be chosen from a list, as described in Section B.6.2. In this case, as the variable **i** is incremented, its value is used to select the word that must be sent to the display, in order for the required digit to illuminate. This method of choosing one value from a list is one way of achieving a *look-up table*, which is an important programming technique.

There are now quite a few nested blocks of code in this example. The **switch** block lies within the **for** block which lies inside the **while** block, which finally lies within the **main** block. The closing brace for each is commented in the program listing. When writing more complex C programs it becomes very important to ensure that each block is ended with a closing brace, at the right place.

```
/*Program Example 3.5: Simple demonstration of 7-segment display. Display digits 0, 1,
2, 3 in turn.
*/

#include "mbed.h"
BusOut display(p5,p6,p7,p8,p9,p10,p11,p12); // segments a,b,c,d,e,f,g,dp

int main() {
    while(1) {
        for(int i=0; i<4; i++) {
            switch (i){
                case 0: display = 0x3F; break; //display 0
                case 1: display = 0x06; break; //display 1
                case 2: display = 0x5B; break;
                case 3: display = 0x4F; break;
            } //end of switch
            wait(0.2);
        } //end of for
    } //end of while
} //end of main
```

Program Example 3.5 Using *for* to sequence values to a seven-segment display

Compile, download and run the program. The display should appear similar to Figure 3.11. Notice carefully, however, by looking at where pin 8 is connected, that this is a common anode connection. It does not exactly replicate Figure 3.10. Pin 3 has been left unconnected.

■ Exercise 3.6

Write a program which flashes the letters H E L P in turn on a seven-segment display, using the circuit of Figure 3.10.



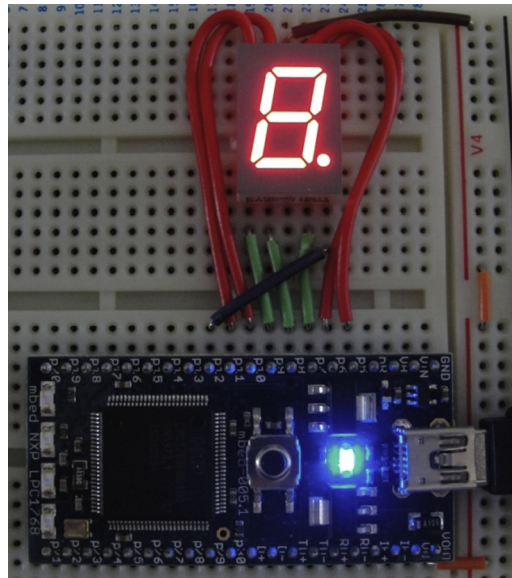


Figure 3.11:
mbed connected to a common anode seven-segment display

3.6 Switching Larger DC Loads

3.6.1 Applying Transistor Switching

The mbed can drive simple DC loads directly with its digital I/O pins, as we saw with the LEDs. The mbed summary data (repeated in Appendix C) tells us that a port pin can source up to around 40 mA. However, this is a short circuit current, so we are unlikely to be able to benefit from it with an actual electrical load connected to the port.

If it is necessary to drive a load — say a motor — which needs more current than an mbed port pin can supply, or which needs to run from a higher voltage, then an interface circuit will be needed. Three possibilities, which allow DC loads to be switched, are shown in Figure 3.12. Each has an input labeled V_L , which is the logic voltage supplied from the port pin. The first two circuits show how a resistive load, such as a motor or heater, can be switched, using a bipolar transistor and a metal oxide semiconductor field effect transistor (MOSFET — a mouthful, but an important device in today's electronics), respectively. In the case of the bipolar transistor, the simple formulae shown can be applied to calculate R_B , starting with knowledge of the required load current and the value of the current gain (β) of the transistor. In the case of the MOSFET, there is a threshold gate-to-source voltage, above which the transistor switches on. This is a particularly useful configuration, as the MOSFET gate can be readily driven by a microcontroller port bit output.

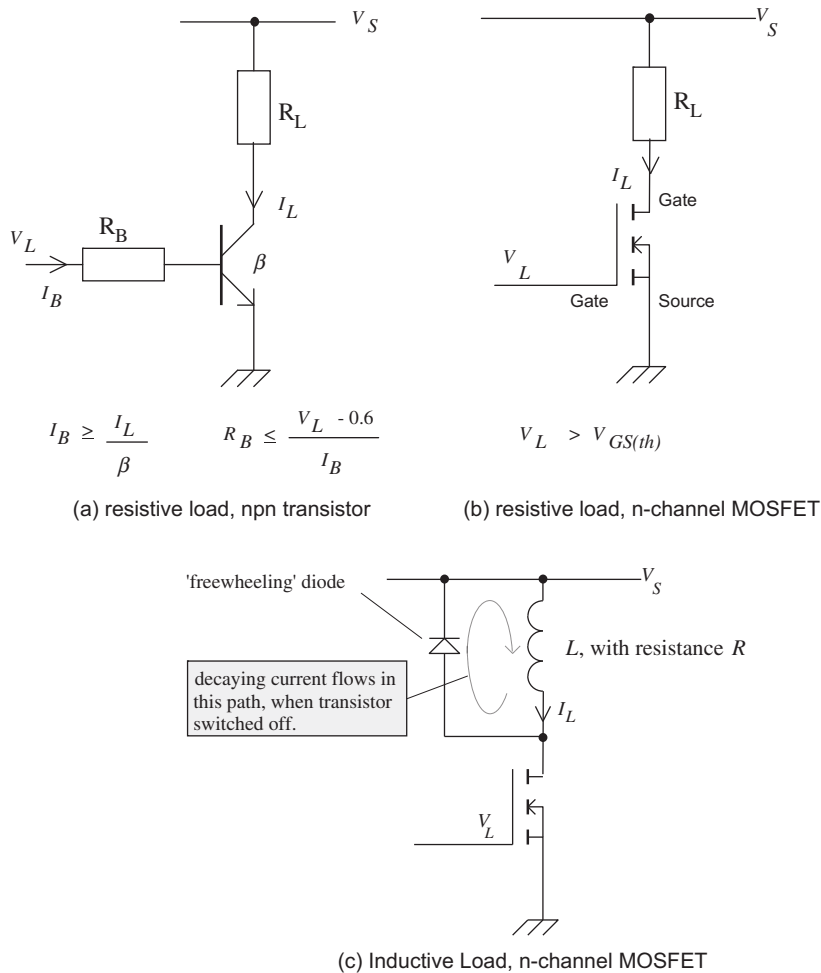


Figure 3.12:
Transistor switching of DC loads

Figure 3.12c shows an inductive load, such as a solenoid or DC motor, being switched. An important addition here is the *freewheeling diode*. This is needed because any inductance with current flowing in it stores energy in the magnetic field that surrounds it. When that current is interrupted, in this case by the transistor being switched off, the energy has to be returned to the circuit. This happens through the diode, which allows decaying current to continue to circulate. If the diode is not included then a high-voltage transient occurs, which can/will destroy the field effect transistor.

3.6.2 Switching a Motor with the mbed

A good switching transistor for small DC loads is the ZVN4206A, whose main characteristics are listed in Table 3.5. An important value is the maximum V_{GS} threshold

Table 3.5: Characteristics of the ZVN4206A n-channel MOSFET

Characteristic	ZVN4206A
Maximum drain-source voltage V_{DS}	60 V
Maximum gate-source threshold $V_{GS(th)}$	3 V
Maximum drain-source resistance when 'On'. $R_{DS(on)}$	1.5 Ω
Maximum continuous drain current I_D	600 mA
Maximum power dissipation	0.7 W
Input capacitance	100 pF

value, shown as 3 V. This means that the MOSFET will respond, just, to the 3.3 V Logic 1 output level of the mbed.

Exercise 3.7

Connect the circuit of Figure 3.13, using a small DC motor. The 6 V for the motor can be supplied from an external battery pack or bench power supply. Its exact voltage is non-critical, and depends on the motor you are using. Write a program so that the motor switches on and off continuously, say 1 s on, 1 s off. Increase the frequency of the switching until you can no longer detect that it is switching on and off. How does the motor speed compare to when the motor was just left switched on?

3.6.3 Switching Multiple Seven-Segment Displays

We saw earlier in the chapter how a single seven-segment display could be connected to the mbed. Each display required eight connections, and if we wanted many displays, then we

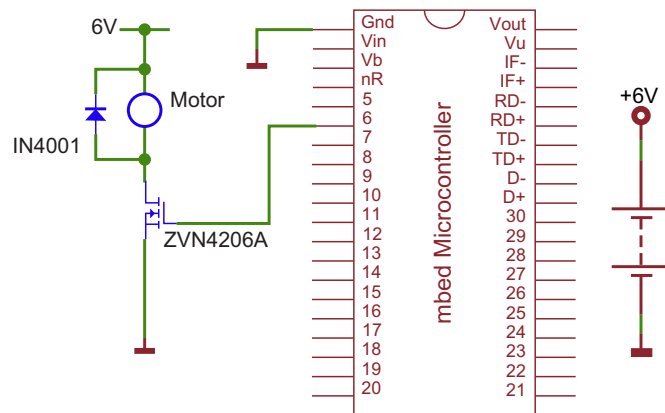


Figure 3.13:
Switching a DC motor

would quickly run out of I/O pins. There is a very useful technique to get around this problem, shown in Figure 3.14. Each segment type on each display is wired together, as shown, and connected back to a microcontroller pin configured as digital output. The common cathode of each digit is then connected to its own drive MOSFET. The timing diagram shown then applies. The segment drives are configured for Digit 1, and that digit's drive transistor is activated, illuminating the digit. A moment later the segment drives are configured for Digit 2, and that digit's drive transistor is activated. This continues endlessly with each digit in turn. If it is done rapidly enough, then the human eye perceives all digits as being continuously illuminated; a useful rate is for each digit to be illuminated in turn for around 5 ms.

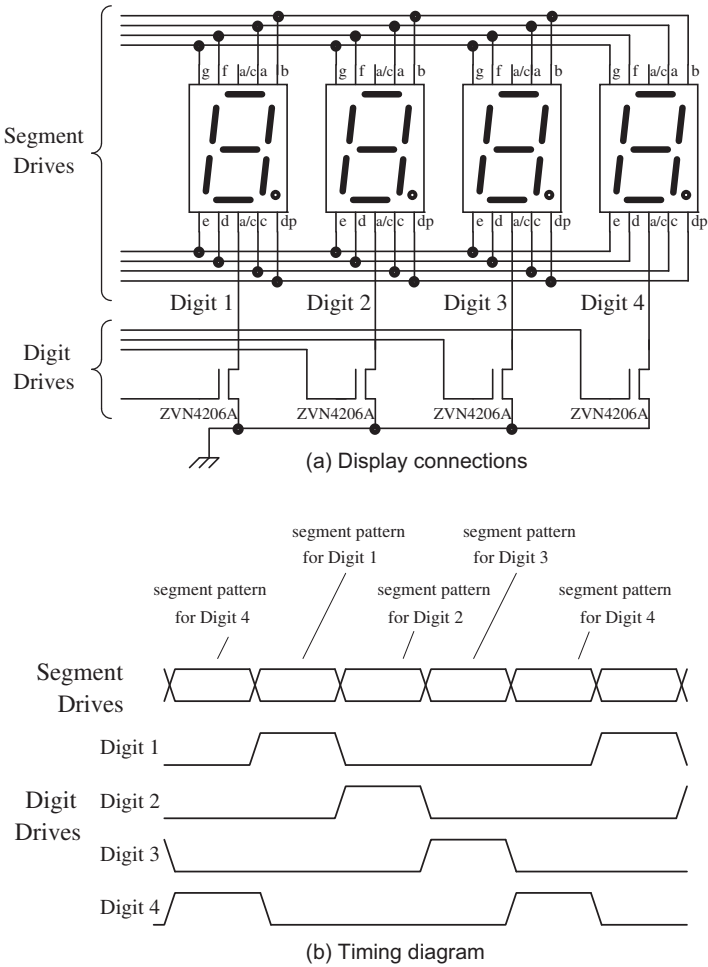


Figure 3.14:
Multiplexing seven-segment displays

Copyright © 2012, Elsevier Science & Technology. All rights reserved.

3.7 Mini-Project: Letter Counter

Use the slotted opto-sensor, push-button switch and one seven-segment LED display to create a simple letter counter. Increment the number on the display by one every time a letter passes the sensor. Clear the display when the push button is pressed. Use an LED to create an extra 'half' digit, so you can count from 0 to 19.

Chapter Review

- Logic signals, expressed mathematically as 0 or 1, are represented in digital electronic circuits as voltages. One range of voltages represents 0, another represents 1.
- The mbed has 26 digital I/O pins, which can be configured as either input or output.
- LEDs can be driven directly from the mbed digital outputs. They are a useful means of displaying a logic value, and of contributing to a simple human interface.
- Electromechanical switches can be connected to provide logic values to digital inputs.
- A range of simple opto-sensors have almost digital outputs, and can with care be connected directly to mbed pins.
- Where the mbed pin cannot provide enough power to drive an electrical load directly, interface circuits must be used. For simple on/off switching a transistor is often all that is needed.

Quiz

1. Complete [Table 3.6](#), converting between the different number types. The first row of numbers is an example.
2. Is it possible to display unambiguously all of the capitalized alphabet characters A, B, C, D, E and F on the seven-segment display shown in [Figure 3.9](#)? For those that can usefully be displayed, determine the segment drive values. Give your answers in both binary and hexadecimal formats.
3. A loop in an mbed program is untidily coded as follows:

```
while (1)
{
    redled = 0;
    wait_ms(12);
    greenled = 1;
    wait(0.002);
    greenled = 0;
    wait_us(24000);
}
```

What is the total period of the loop, expressed in seconds, milliseconds and microseconds?

Table 3.6:

Binary	Hexadecimal	Decimal
0101 1110	5E	94
1101		
	77	
		129
	6F2	
1101 1100 1001		
		4096

4. The circuit of Figure 3.5b is used eight times over to connect eight switches to 8 mbed digital inputs. The pull-up resistors have a value of 10 k Ω , and are connected to the mbed supply of 3.3 V. What current is consumed due to this circuit configuration when all switches are closed simultaneously? If this current drain must be limited to 0.5 mA, to what value must the pull-up resistors be increased? Reflect on the possible impact of pull-up resistors in low-power circuits.
5. What is the current taken by the display connected in Figure 3.10, when the digit 3 is showing?
6. If in Figure 3.10 a segment current of approximately 4 mA was required, what value of resistor would need to be introduced in series with each segment?
7. A student builds an mbed-based system. To one port he connects the circuit of Figure 3.15a, using LEDs of the type used in Figure 3.4, but is then disappointed that the

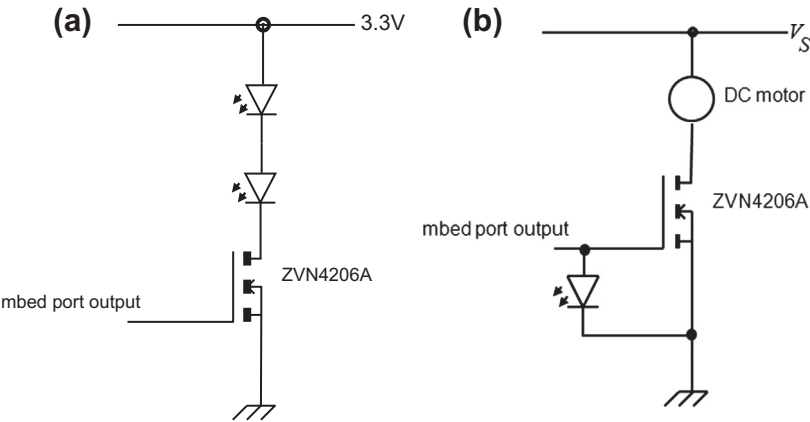


Figure 3.15:
Trial switching circuits

Copyright © 2012, Elsevier Science & Technology. All rights reserved.

LEDs do not appear to light when expected. Explain why this is so, and suggest a way of changing the circuit to give the required behavior.

8. Another student wants to control a DC motor from the mbed, and therefore builds the circuit of Figure 3.15b, where V_S is a DC supply of appropriate value. As a further indication that the motor is running, she connects a standard LED, as seen in Figure 3.3, directly to the port bit. She then complains of unreliable circuit behavior. Explain any necessary changes that should be made to the circuit.
9. Look at the mbed circuit diagram, Reference 2.2, and find the four onboard LEDs. Estimate the current each one takes when 'on', assuming a forward voltage of 1.8 V.

References

- 3.1. Floyd, T. (2008). Digital Fundamentals. 10th edition. Pearson Education.
- 3.2. The Kingbright home site. <http://www.kingbright.com/>

This page intentionally left blank