CHAPTER 4

Analog Output

4.1 Introducing Data Conversion

Microcontrollers, as we know, are digital devices; but they spend most of their time dealing with a world which is analog, as illustrated in Fig. 1.1. To make sense of incoming analog signals, for example, from a microphone or temperature sensor, they must be able to convert them into digital form. After processing the data, they may then need to convert digital data back to analog form, for example, to drive a loudspeaker or DC motor. We give these processes the global heading *data conversion*. Techniques applied in data conversion form a huge and fascinating branch of electronics. They are outside the subject matter of this book; however, if you wish to learn more about them, consider any major electronics text. To get deep into the topic, read [1].

While conversion in both directions between digital and analog is necessary, it is conversion from analog to digital form that is the more challenging task; therefore, in this earlier chapter we consider the easier option—conversion from digital to analog (DAC).

4.1.1 The DAC

A *digital-to-analog converter* is a circuit which converts a binary input number into an analog output. The actual circuitry inside a DAC is complex, and need not concern us. We can, however, represent the DAC as a block diagram, as in Fig. 4.1. This has a digital input, represented by D, and an analog output, represented by V_o. The "yardstick" by

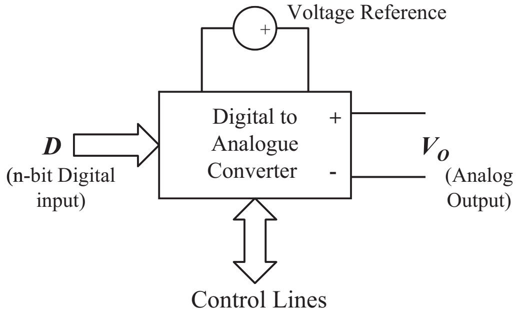


Figure 4.1
The digital-to-analog converter.

which the DAC calculates its output voltage is a *voltage reference*, a precise, stable, and known voltage.

Most DACs have a simple relationship between their digital input and analog output, with many (including the one inside the LPC1768) applying Eq. (4.1). Here, V_r is the value of the voltage reference, D is the value of the binary input word, n is the number of bits in that word, and V_o is the output voltage. Fig. 4.2 shows this equation represented graphically. For each input digital value, there is a corresponding analog output. It's as if we are creating a voltage staircase with the digital inputs. The number of possible output values is given by 2^n, and the step size by $V_r/2^n$; this is called the *resolution*. The maximum possible output value occurs when $D = (2^n - 1)$, so the value of V_r as an output is never quite reached. The *range* of the DAC is the difference between its maximum and minimum output values. For example, a 6-bit DAC will have 64 possible output values; if it has a 3.2 V reference, it will have a resolution (step size) of 50 mV.

$$V_o = \frac{D}{2^n} V_r \tag{4.1}$$

In Fig. 2.3, we see that the LPC1768 has a 10-bit DAC; there will therefore be 2^{10} steps in its output characteristic, i.e., 1024. Ref. [4] of Chapter 2 further tells us that it normally uses its own power supply voltage, i.e., 3.3 V, as voltage reference. The step size, or resolution, will therefore be 3.3/1024, i.e., 3.22 mV.

4.2 Analog Outputs on the mbed

The mbed pin connection diagram, Fig. 2.1C, has already shown us that the mbed is rich in analog input/output capabilities. We see that pins 15–20 can be analog input, while pin 18 is the only analog output.

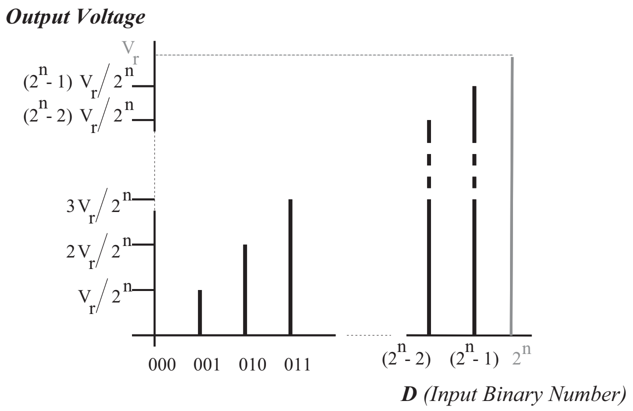


Figure 4.2
The DAC input/output characteristic.

Table 4.1: Application programming interface summary for mbed analog output.

Functions	Usage
`AnalogOut`	Create an AnalogOut object connected to the specified pin
`write`	Set the output voltage, specified as a percentage (float)
`write_u16`	Set the output voltage, represented as an unsigned short in the range [0x0, 0xFFFF]
`read`	Return the current output voltage setting, measured as a percentage (float)
`mbed-defined operator: =`	An operator shorthand for write()

The application programming interface (API) summary for analog output is shown in Table 4.1. It follows a pattern similar to the digital input and output utilities we have already seen. Here, with **AnalogOut**, we can initialize and name an output; using **write( )** or **write_u16( )**, we can set the output voltage either with a floating point number or with a hexadecimal number. Finally, we can simply use the = sign as a shorthand for write, which is what we will mostly be doing.

C code feature

Notice in Table 4.1 the way that data types *float* and *unsigned short* are invoked. In C/C++, all data elements have to be declared before use; the same is true for the return type of a function and the parameters it takes. A review of the concept of *floating point* number representation can be found in Appendix A. Table B.4 in Appendix B summarizes the different data types that are available. The DAC itself requires an unsigned binary number as input, so the **write_u16( )** function represents the more direct approach of writing to it.

We try now a few simple programs which apply the mbed DAC, creating first fixed voltages and then waveforms.

4.2.1 Creating Constant Output Voltages

Create a new program using Program Example 4.1. In this we create an analog output labeled **Aout**, by using the **AnalogOut** utility. It's then possible to set the analog output simply by setting **Aout** to any permissible value; we do this three times in the program. By default **Aout** takes a floating point number between 0.0 and 1.0 and outputs this to pin 18. The actual output voltage on pin 18 is between 0 and 3.3 V, so the floating point number that is output is scaled to this.

```
/*Program Example 4.1: Three values of DAC are output in turn on Pin 18. Read the
output on a DVM.
                                                                              */
#include "mbed.h"
AnalogOut Aout(p18);          //create an analog output on pin 18
int main() {
  while(1) {
  Aout=0.25;          // 0.25*3.3V = 0.825V
  wait(2);
  Aout=0.5;           // 0.5*3.3V = 1.65V
  wait(2);
  Aout=0.75;          // 0.75*3.3V = 2.475V
  wait(2);
  }
}
```

Program Example 4.1: Trial DAC output

Compile the program in the usual way and let it run. Connect a digital voltmeter (DVM) between pins 1 and 18 of the mbed. You should see the three output voltages named in the comments of Program Example 4.1 being output in turn.

■ Exercise 4.1

Adjust Program Example 4.1 so that the **write_u16( )** function is used to set the analog output, giving the same output voltages.

■

4.2.2 Saw Tooth Waveforms

Let's now make a saw tooth wave and view it on an oscilloscope. Create a new program and enter the code of Program Example 4.2. As in many cases previously, the program is made up of an endless **while(1)** loop. Within this we see a **for** loop. In this example, the variable **i** is initially set to 0, and on each iteration of the loop, it is incremented by 0.1. The new value of **i** is applied within the loop. When **i** reaches 1, the loop terminates. As the **for** loop is the only code within the endless while loop, it then restarts, repeating this action continuously.

```
/*Program Example 4.2: Saw tooth waveform on DAC output. View on
oscilloscope
                                                                            */
#include "mbed.h"
AnalogOut Aout(p18);
float i;

int main() {
  while(1){
    for (i=0;i<1;i=i+0.1){       // i is incremented in steps of 0.1
      Aout=i;
      wait(0.001);         // wait 1 millisecond
    }
  }
}
```

Program Example 4.2: Saw tooth waveform

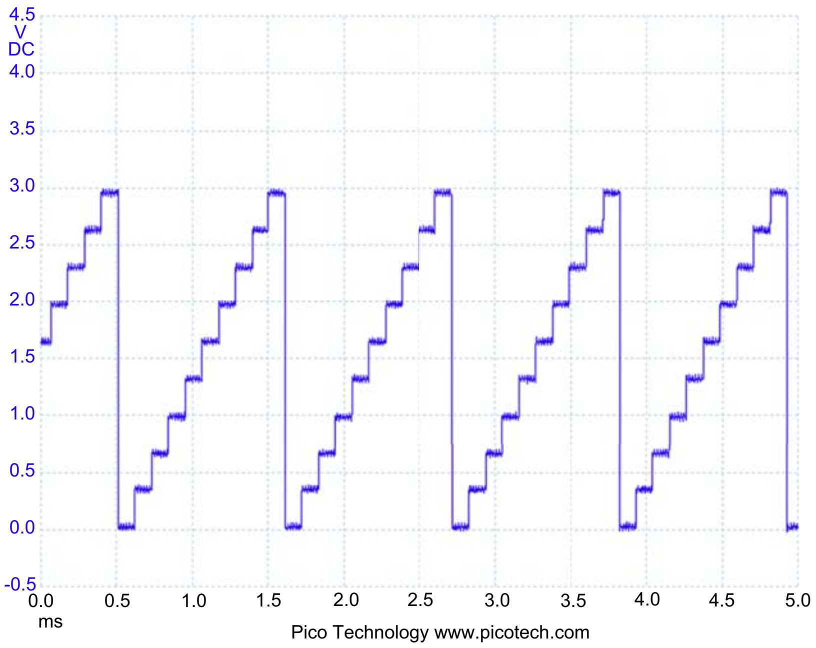


Figure 4.3
A stepped saw tooth waveform.

Connect an oscilloscope probe to pin 18 of the mbed, with its earth connection to pin 1. Check that you get a saw tooth waveform similar to that shown in Fig. 4.3. Ensure that the duration of each step is the 1 ms you define in the program, and try varying this. The waveform should start from 0 V and go up to a maximum of 3.3 V. Check for these values.

If you don't have an oscilloscope you can set the wait parameter to be much longer (say 100 ms) and use the DVM; you should then see the voltage step up from 0 to 3.3 V and then reset back to 0 V again.

Exercise 4.2

Improve the resolution of the saw tooth by having more but smaller increments, i.e., reduce the value by which **i** increments. The result should be as seen in Fig. 4.4.

Exercise 4.3

Create a new project and devise a program which outputs a triangular waveform (i.e., one that counts down as well as up). The oscilloscope output should look like Fig. 4.5.

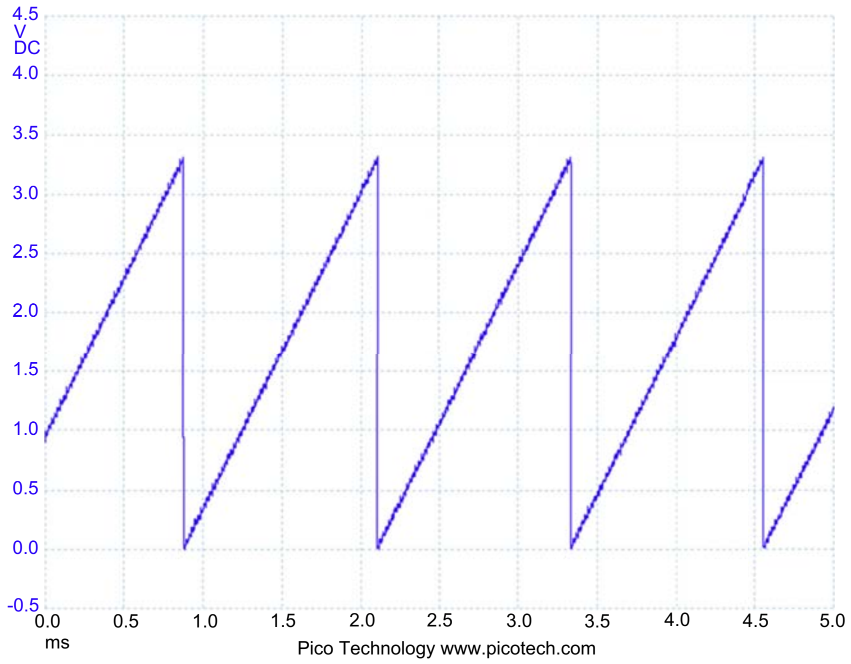


Figure 4.4
A smooth saw tooth waveform.

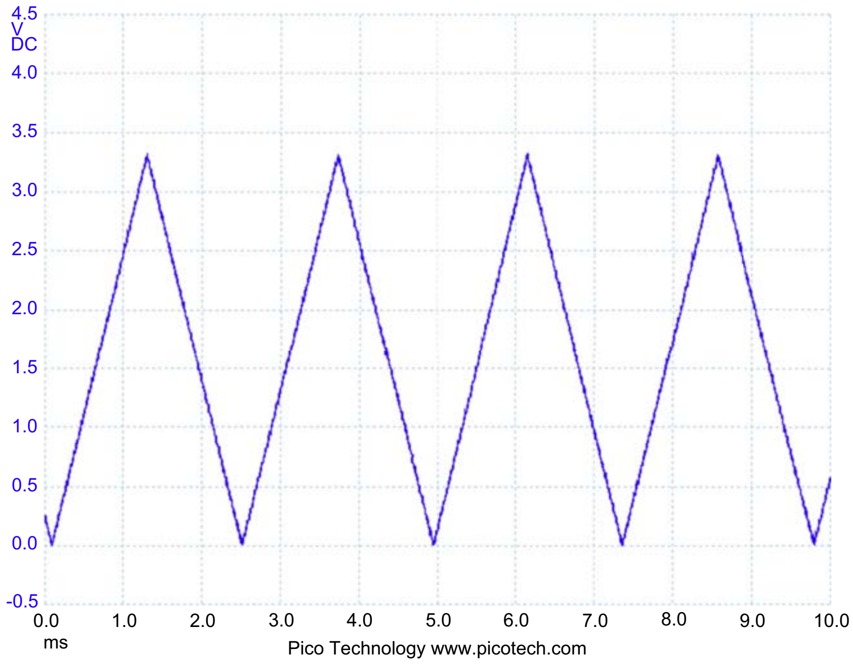


Figure 4.5
A triangular waveform.

4.2.3 Testing the DAC Resolution

Now let's return to the question of the DAC resolution, which we touched on in Section 4.1.1. Try changing the **for** loop in your version of Program Example 4.2 to this:

```
for (i=0;i<1;i=i+0.0001){
  Aout=i;
  wait(1);
  led1=!led1;
}
```

We've also included a little LED indication here, so add this line before **main( )** to set it up:

```
DigitalOut led1(LED1);
```

C code feature

This adjusted program will produce an extremely slow saw tooth waveform, which will take 10,000 steps to reach the maximum value, each one taking a second (hence the Period of the waveform is 10,000 s, or two and three quarter hours!). Our purpose is not, however, to view the waveform but to explore carefully the DAC characteristic of Fig. 4.2. Notice the use of the NOT operator, in the form of an exclamation mark (**!**), for the first time. This causes logical inversion, so a Logic 0 is replaced by 1 and vice versa.

Switch your DVM to its finest voltage range, so that you have millivolt resolution on the scale; the 200 mV scale is useful here. Connect the DVM between pins 1 and 18 and run the program. The LED changes state every time a new value is output to the DAC. However, you will notice an interesting thing. Your DVM reading does *not* change with every LED change. Instead it changes state in distinct steps, with each change being around 3 mV. You will notice also that it takes around 5 LED "blinks", i.e., ten updates of the DAC value, before each DAC change. All this is what we anticipated in Section 4.1.1, where we predicted a step size of 3.22 mV; each step size is equal to the DAC resolution. The float value is rounded to the nearest digital input to the DAC, and it takes around 10 increments of the float value for the DAC digital input to be incremented by one.

4.2.4 Generating a Sine Wave

C code feature

It is an easy step from here to generate a sine wave. We will apply the **sin( )** function, which is part of the C standard library (see Section B9.2). Take a look at Program Example 4.3. To produce one cycle of the sine wave, we want to take sine values of a number which increases from 0 to 2π radians. In this program we use a **for** loop to increment variable **i** from 0 to 2 in small steps and then multiply that number by π when the sine value is calculated. There are, of course, other ways of getting this result. The final challenge is that the DAC cannot output negative values. Therefore we add a fixed value (an *offset*) of 0.5 to the number being sent to the DAC; this ensures that all output values lie within its available range. Notice that we are using the multiply operator, *, for the first time.

```
/*Program Example 4.3: Sine wave on DAC output. View on oscilloscope
                                                                   */
#include "mbed.h"
AnalogOut Aout(p18);
float i;
int main() {
  while(1)  {
    for (i=0;i<2;i=i+0.05) {
      Aout=0.5+0.5*sin(i*3.14159);  // Compute the sine value, + half the range
      wait(.001);                   // Controls the sine wave period
    }
  }
}
```

Program Example 4.3: Generating a sinusoidal waveform

Exercise 4.4

Observe on the oscilloscope the sine wave that Program Example 4.3 produces. Estimate its frequency from information in the program and then measure it. Do the two values agree? Try varying the frequency by varying the wait parameter. What is the maximum frequency you can achieve with this program?

4.3 Another Form of Analog Output: Pulse Width Modulation

The DAC is a fine circuit, and we have many uses for it. Yet it adds complexity to a microcontroller and sometimes moves us to the analog domain before we're ready to go. *Pulse width modulation* (PWM), an alternative, represents a neat and remarkably simple way of getting a rectangular digital waveform to control an analog variable, usually voltage or current. PWM control is used in a variety of applications, ranging from telecommunications to robotic control. Its importance is reflected in the fact that the mbed has *six* PWM outputs (Fig. 2.1), compared with its one analog output.

Three example PWM signals are shown in Fig. 4.6. In keeping with a typical PWM source, each has the same period, but a different pulse width, or "on" time; the

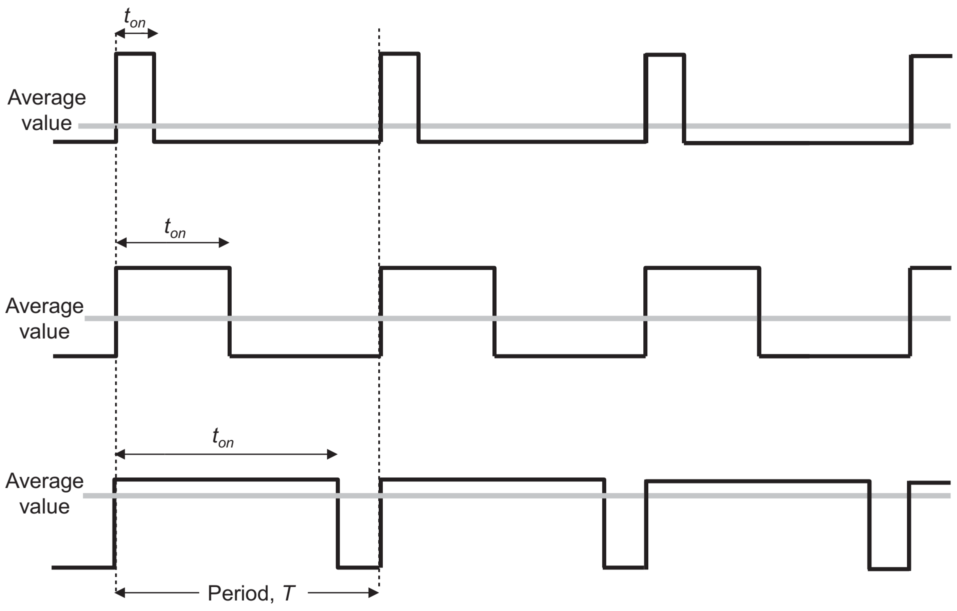


Figure 4.6
Pulse width modulation waveforms.

pulse *width* is being modulated. The *duty cycle* is the proportion of time that the pulse is "on" or "high" and is expressed as a percentage, i.e., (applying symbols from Fig. 4.6):

$$\text{duty cycle} = \frac{\text{pulse on time}}{\text{pulse period}} * 100\% = \frac{\text{ton}}{T} * 100\% \tag{4.2}$$

A 100% duty cycle therefore means "continuously on" and a 0% duty cycle means "continuously off." PWM streams are easily generated by digital counters and comparators, which can readily be designed into a microcontroller. They can also be produced simply by program loops and a standard digital output, with no dedicated hardware at all. We see this later in the chapter.

Whatever duty cycle a PWM stream has, there is an average value, as indicated in the figure. If the on time is small, the average value is low; if the on time is large, the average value is high. By controlling the duty cycle, we therefore control this average value. When using PWM, it is this average that we're usually interested in. It can be extracted from the PWM stream in a number of ways. Electrically, we can use a low-pass filter, e.g., the resistor-capacitor combination of Fig. 4.7A. In this case, and as long as PWM frequency and values of R and C are appropriately chosen, V_{out} becomes an analog output, with a bit of ripple, and the combination of PWM and filter acts like a simple DAC. Alternatively, if we switch the current flowing in an inductive load, as seen in Fig. 4.7B, then the

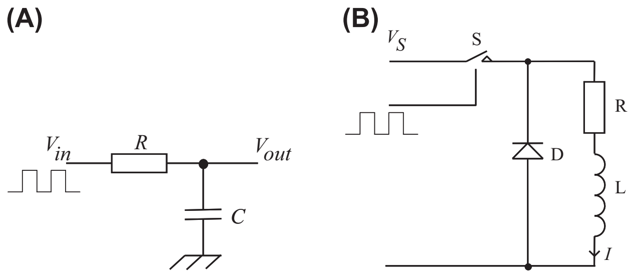


Figure 4.7
Simple averaging circuits. (A) A resistor-capacitor low-pass filter and (B) an inductive load.

inductance has an averaging effect on the current flowing through it. This is very important, as the windings of any motor are inductive, so we can use this technique for motor control. The switch in Fig. 4.7B is controlled by the PWM stream and can be a transistor. We need incidentally to introduce the freewheeling diode, just as we did in Fig. 3.14C, to provide a current path when the switch is open.

In practice, this electrical filtering is not always required. Many physical systems have internal inertias which, in reality, act like low-pass filters. We can, for example, dim a conventional filament light bulb with PWM. In this case, varying the pulse width varies the average temperature of the bulb filament, and the dimming effect is achieved.

As an example, the control of a DC motor is a very common task in robotics; the speed of a DC motor is proportional to the applied DC voltage. We *could* use a conventional DAC output, drive it through an expensive and bulky power amplifier, and use the amplifier output to drive the motor. Alternatively, a PWM signal can be used to drive a power transistor directly, which replaces the switch of Fig. 4.7B; the motor is the inductor/resistor combination in the same circuit. This technique is taken much further in the field of power electronics, with the PWM concept being taken far beyond these simple but useful applications.

4.4 Pulse Width Modulation on the mbed

4.4.1 Using the mbed Pulse Width Modulation Sources

As mentioned, it is easy to generate PWM pulse streams using simple digital building blocks. We don't explore how that is done in this book, but you can find the information elsewhere if you wish, for example, in Ref. [1] of Chapter 1. As for the mbed, Fig. 2.1C shows that there are six PWM outputs available, from pin 21 to 26 inclusive. Note that on the LPC1768 microcontroller, and hence the mbed, the PWM sources all share the same period/frequency; if the period is changed for one, then it is changed for all.

As with all peripherals, the mbed PWM ports are supported by library utilities and functions, as shown in Table 4.2. This is rather more complex than the API Tables we

Table 4.2: Application programming interface summary for pulse width modulation output.

Functions	Usage
`PwmOut`	Create a PwmOut object connected to the specified pin
`write`	Set the output duty cycle, specified as a normalized float (0.0—1.0)
`read`	Return the current output duty cycle setting, measured as a normalized float (0.0—1.0)
`period`	Set the PWM period, specified in seconds (float), keeping the duty cycle the same.
`period_ms`	Set the PWM period, specified in milliseconds (int), keeping the duty cycle the same.
`period_us`	Set the PWM period, specified in microseconds (int), keeping the duty cycle the same.
`pulsewidth`	Set the PWM pulse width, specified in seconds (float), keeping the period the same.
`pulsewidth_ms`	Set the PWM pulse width, specified in milliseconds (int), keeping the period the same.
`pulsewidth_us`	Set the PWM pulse width, specified microseconds (int), keeping the period the same.
`mbed-defined` `operator: =`	An operator shorthand for write()

have seen so far, so we can expect a little more complexity in its use. Notably, instead of having just one variable to control (for example, a digital or analog output), we now have two, period and pulse width, or duty cycle derived from this combination. Similar to previous examples, a PWM output can be established, named, and allocated to a pin using **PwmOut.** Subsequently, it can be varied by setting its period, duty cycle, or pulse width. As shorthand, the **write( )** function can simply be replaced by =.

4.4.2 Some Trial Pulse Width Modulation Outputs

As a first program using an mbed PWM source, let's create a signal which we can see on an oscilloscope. Make a new project and enter the code of Program Example 4.4. This will generate a 100 Hz pulse with 50% duty cycle, i.e., a perfect square wave.

```
/*Sets PWM source to fixed frequency and duty cycle. Observe output on
oscilloscope.
                                                                    */
#include "mbed.h"
PwmOut PWM1(p21);          //create a PWM output called PWM1 on pin 21
int main() {
  PWM1.period(0.010);          // set PWM period to 10 ms
  PWM1=0.5;                    // set duty cycle to 50%
}
```

Program Example 4.4: Trial PWM output

In this program example we first set the PWM period. There is no shorthand for this, so it is necessary to use the full **PWM1.period(0.010);** statement. The duty cycle is then defined as a decimal number, between the value of 0 and 1. We could also set the duty cycle as a pulse time with the following:

```
PWM1.pulsewidth_ms(5);       // set PWM pulsewidth to 5 ms
```

When you run the program you should be able to see the square wave on the oscilloscope and verify the output frequency.

Exercise 4.5

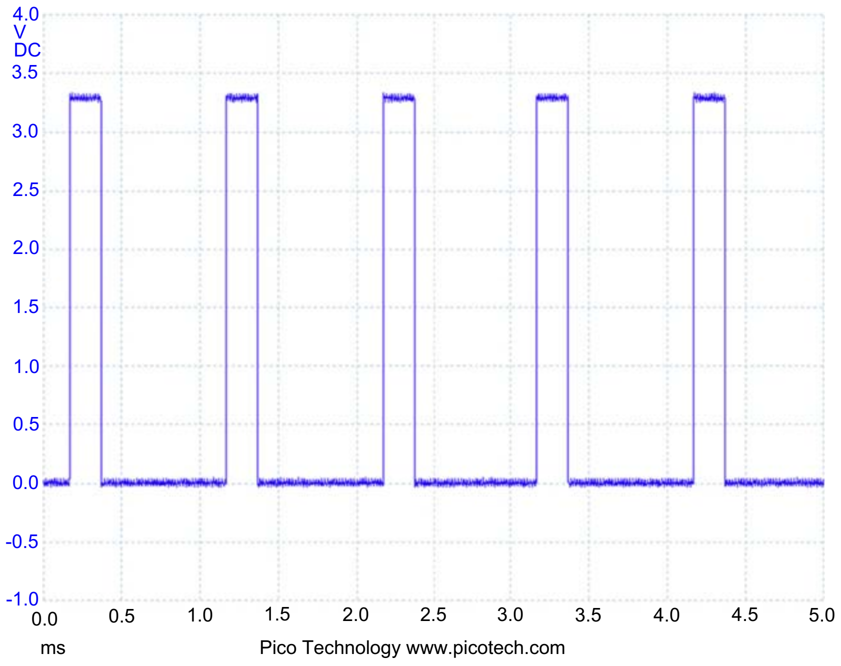
1. Change the duty cycle of Program Example 4.4 to some different values, say 0.2 (20%) and 0.8 (80%) and check the correct display is seen on the oscilloscope, for example, as shown in Fig. 4.8.
2. Change the program to give the same output waveforms but using **period_ms( )** and **pulsewidth_ms( )**.

4.4.3 Speed Control of a Small Motor

Let's now apply the mbed PWM source to control motor speed. Use the simple motor circuit already applied in the previous chapter (Fig. 3.15) but move the mbed motor drive output from pin 6 to pin 21. Create a project using Program Example 4.5. This ramps the PWM up from a duty cycle of 0% to 100%, using programming features that are already familiar.

```
/*Program Example 4.5: PWM control to DC motor is repeatedly ramped
                                                              */
#include "mbed.h"
PwmOut PWM1(p21);
float i;
int main() {
  PWM1.period(0.010);        //set PWM period to 10 ms
  while(1) {
    for (i=0;i<1;i=i+0.01) {
      PWM1=i;                // update PWM duty cycle
      wait(0.2);
      }
  }
}
```

Program Example 4.5: Controlling motor speed with mbed PWM source



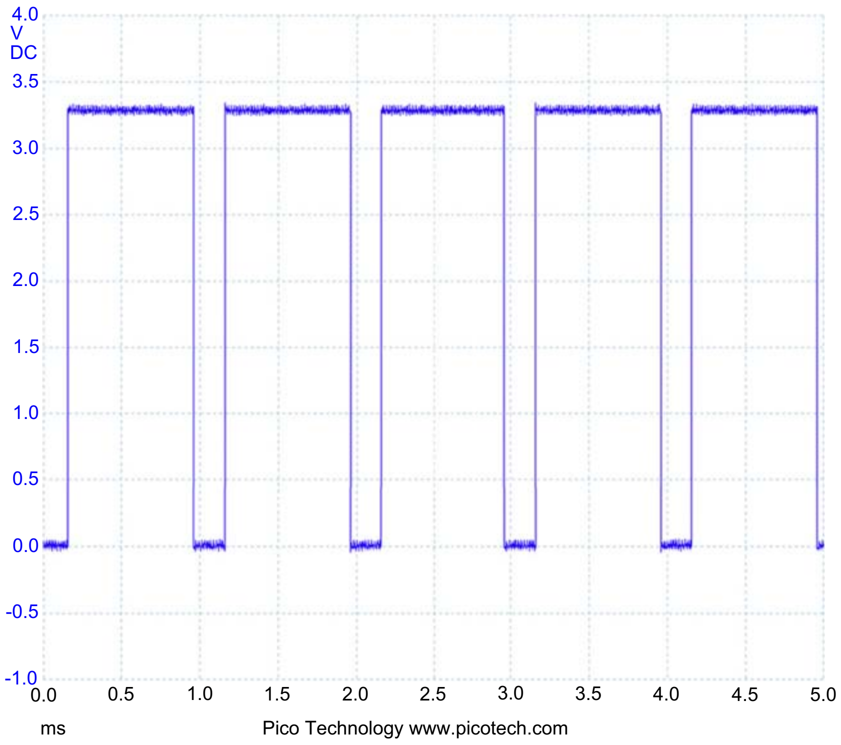


Figure 4.8
Pulse width modulation observed from the mbed output.

Compile, download, and run the program. See how the motor performs and observe the waveform on the oscilloscope. You are seeing one of the classic applications of PWM, controlling motor speed through a simple digital pulse stream.

4.4.4 Generating Pulse Width Modulation in Software

Although we have just used PWM sources that are available on the mbed, it is useful to realize that these aren't essential for creating PWM; we can actually do it just with a digital output and some timing. In Exercise 3.8, you were asked to write a program which switched a small DC motor on and off continuously, 1 s on, 1 s off. If you speed this switching up, to say 1 ms on, 1 ms off, you will immediately have a PWM source.

Create a new project with Program Example 4.6 as source code. Notice carefully what the program does. There are two **for** loops in the main **while** loop. The motor is initially switched off for 5 s. The first **for** loop then switches the motor on for 400 µs, and off for 600 µs; it does this 5000 times. This results in a PWM signal of period 1 ms, and duty cycle 40%. The second **for** loop switches the motor on for 800 µs, and off for 200 µs; the period is still 1 ms, but the duty cycle is 80%. The motor is then switched full on for 5 s. This sequence continues indefinitely.

```
/*Program Example 4.6: Software generated PWM. 2 PWM values generated in turn, with
full on and off included for comparison.
                                                                                 */
#include "mbed.h"
DigitalOut motor(p6);
int i;
int main() {
  while(1) {
    motor = 0;                    //motor switched off for 5 secs
    wait (5);
    for (i=0;i<5000;i=i+1) {      //5000 PWM cycles, low duty cycle
      motor = 1;
      wait_us(400);               //output high for 400us
      motor = 0;
      wait_us(600);               //output low for 600us
    }
    for (i=0;i<5000;i=i+1) {      //5000 PWM cycles, high duty cycle
      motor = 1;
      wait_us(800);               //output high for 800us
      motor = 0;
      wait_us(200);               //output low for 200us
    }
    motor = 1;                    //motor switched fully on for 5 secs
    wait (5);
}
}
```

Program Example 4.6: Generating PWM in software

Compile and download this program and apply the circuit of Fig. 3.12. You should find that the motor runs with the speed profile indicated. PWM period and duty cycle can be readily verified on the oscilloscope. You may wonder for a moment if it's necessary to have dedicated PWM ports, when it seems quite easy to generate PWM with software. Remember, however, that in this example the CPU becomes totally committed to this task and can do nothing else. With the hardware ports, we can set the PWM running, and the CPU can then get on with some completely different activity.

Exercise 4.6

Vary the motor speeds in Program Example 4.6 by changing the duty cycle of the PWM, initially keeping the frequency constant. Depending on the motor you use, you will probably find that for small values of duty cycle the motor will not run at all, due to its own friction. This is particularly true of geared motors. Observe the PWM output on the oscilloscope and confirm that on and off times are as indicated in the program. Try also at much higher and lower frequencies.

4.4.5 Servo Control

A servo is a small, lightweight, rotary position control device, often used in radio-controlled cars and aircraft to control angular position of variables such as steering, elevators, and rudders. The Hitec HS-422 servo is shown in Fig. 4.9A. Servos are now popular in a range of robotic applications. The servo shaft can be positioned to specific angular positions by sending the servo a PWM signal. As long as the modulated signal exists on the servo input, it will maintain the angular position of the shaft. As the modulated signal changes, the angular position of the shaft changes. This is illustrated in Fig. 4.9B. Many servos use a PWM signal with a 20 ms period, as is shown here. In this example, the pulse width is modulated from 1.25 to 1.75 ms to give the full 180 degree range of the servo. Servos are usually supplied with a 3-way connector, as seen in Fig. 4.9A. These connect 0 V, 5 V, and the PWM signal.

Connect a servo to the mbed, using either a breadboard or application board. Whether using breadboard or app board, the servo requires a higher current than the USB standard can provide, and so it is essential that you power it using an external supply. A 4xAA (6 V) battery pack meets the supply requirement of most small servos.

The circuit of Fig. 4.10A should be applied if using the breadboard; the mbed itself can still be supplied from the USB. Alternatively, it can also be supplied from the battery pack, through VIN (pin 2). The mbed then regulates the incoming 6 V to the 3.3 V that it requires.

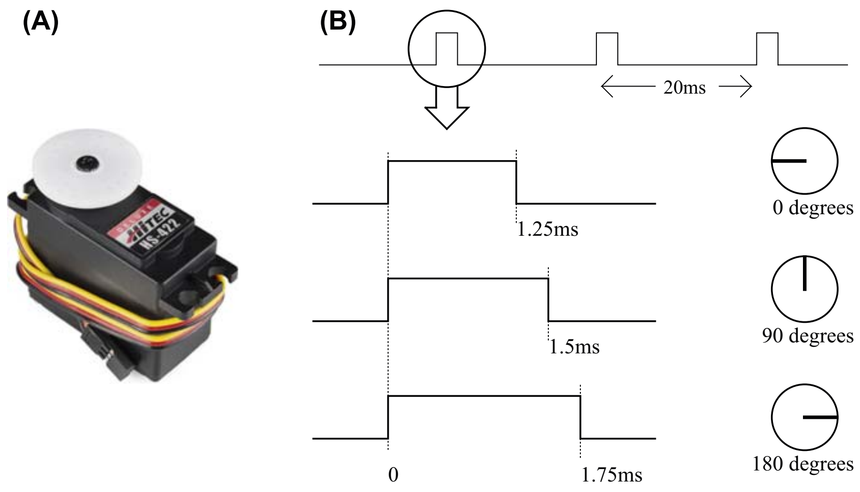


Figure 4.9
Servo essentials. (A) The Hitec HS-422 servo and (B) example servo drive pulse width modulation waveforms. *Image courtesy of Sparkfun.*

App Board

Conveniently, the app board has two PWM connectors which match the standard connector used on most servos. These are seen as Item 8 in Fig. 2.7; they are labeled PWM1 and PWM2, and link to mbed pins 21 and 22, respectively. To align with the circuit diagram of Fig. 4.10A, connect to PWM1, i.e., the one next to the potentiometer. With the app board, the external supply connects onto the board through a jack plug immediately below the mbed USB connector, not shown in the Figure. A battery pack is again used, though not shown. This battery pack can power the whole system, so the USB can be left disconnected once the program has been downloaded.

Exercise 4.7 (For App Board or Breadboard)

Create a new project and write a program which sets a PWM output on pin 21. Set the PWM period to 20 ms. Try a number of different duty periods, taking values from Fig. 4.9, and observe the servo's position. Then write a program which continually moves the servo shaft from one limit to the other.

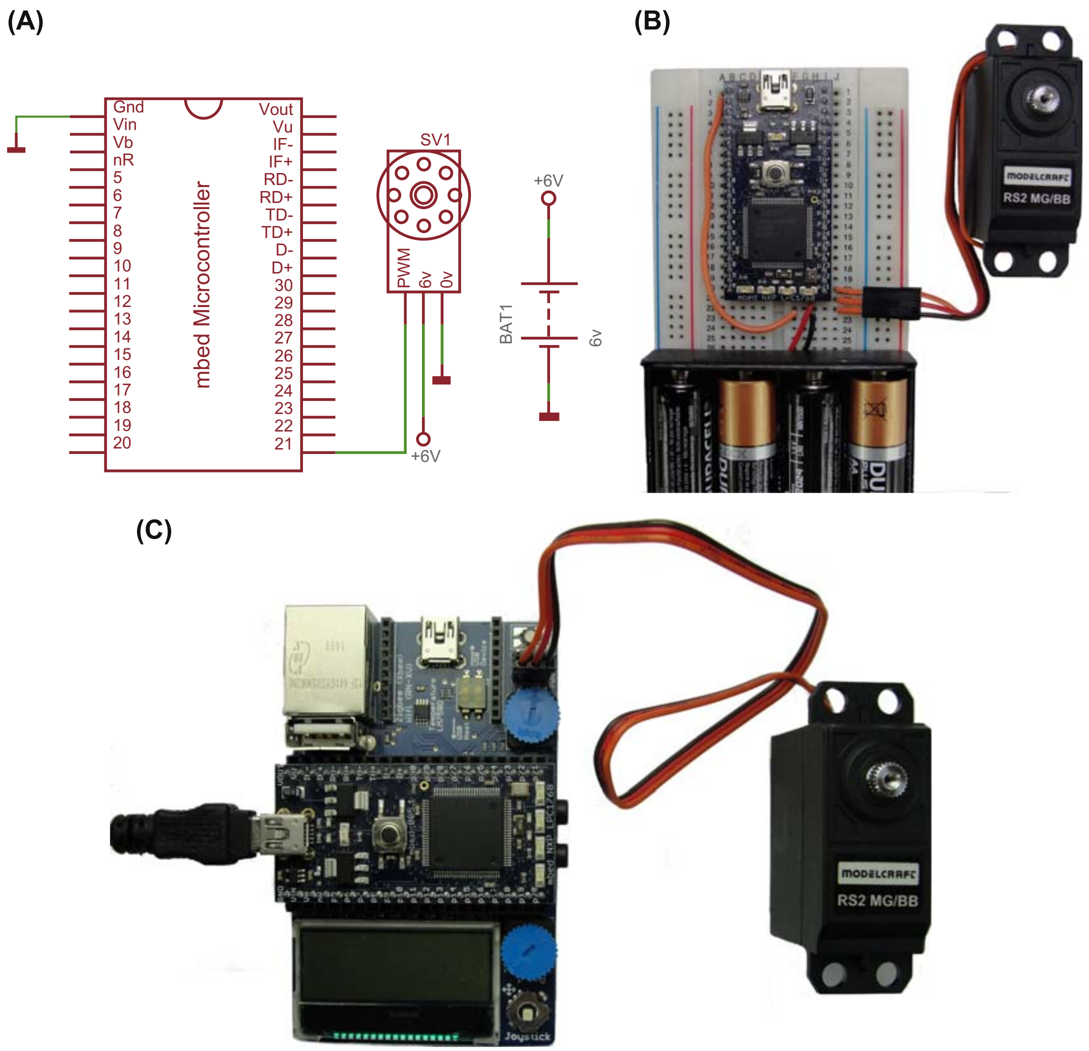


Figure 4.10
Using pulse width modulation (PWM) to drive a servo. (A) The connection diagram, (B) the breadboard build, and (C) using the app board for PWM out.

4.4.6 Producing Audio Output

We can use the PWM source simply as a variable frequency signal generator. In this example we use it to sound a piezo transducer or speaker and play the start of an old London folk song called "Oranges and Lemons." This song imagines that the bells of each church in London calls a particular message. If you're a reader of music you may recognize the tune in Fig. 4.11; if you're not, don't worry! You just need to

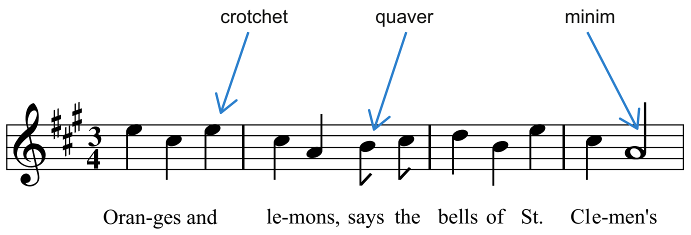


Figure 4.11
The "Oranges and Lemons" tune.

know that any note which is a minim ("half note" in the United States) lasts twice as long as a crotchet ("quarter note" in the United States), which in turn lasts twice as long as a quaver ("eighth note" in the United States). Put another way, a crotchet lasts one beat, a minim two, and a quaver a half. The pattern for the music is as shown in Table 4.3. Note that here we are simply using the PWM as a variable frequency signal source and not actually modulating the pulse width as a proportion of frequency at all.

C code feature

Create a new program and enter Program Example 4.7. This introduces an important new C feature, the *array*. If you're unfamiliar with this, then read the review in Section B8.1. The program uses two arrays, one defined for frequency data, the other for beat length. There are 12 values in each, for each of the 12 notes in the tune. The

Table 4.3: Frequencies of notes used in tune.

Word/Syllable	Musical Note	Frequency (Hz)	Beats
Oran-	E	659	1
ges	C#	554	1
and	E	659	1
le-	C#	554	1
mons,	A	440	1
says	B	494	½
the	C#	554	½
bells	D	587	1
of	B	494	1
St	E	659	1
Clem-	C#	554	1
en's	A	440	2

program is structured round a **for** loop, with variable **i** as counter. As **i** increments, each array element is selected in turn. Notice that **i** is set just to reach the value 11; this is because the value 0 addresses the first element in each array, and the value 11 hence addresses the 12th. From the frequency array the PWM period is calculated and set, always with a 50% duty ratio. The beat array determines how long each note is held, using the **wait** function.

```
/*Program Example 4.7: Plays the tune "Oranges and Lemons" on a piezo buzzer, using
PWM
                                                                              */
#include "mbed.h"
PwmOut buzzer(p26);

                                                           //frequency array
float frequency[]={659,554,659,554,440,494,554,587,494,659,554,440};
float beat[]={1,1,1,1,1,0.5,0.5,1,1,1,1,2};         //beat array
int main() {
  while (1) {
    for (int i=0;i<=11;i++) {
      buzzer.period(1/(2*frequency[i]));     // set PWM period
      buzzer=0.5;                            // set duty cycle
      wait(0.4*beat[i]);                     // hold for beat period
    }
  }
}
```

Program Example 4.7: "Oranges and Lemons" program

App Board

Compile the program and download. If you're using the app board, then the on-board speaker is hard-wired to pin 26 of the mbed (item 5 of Fig. 2.7), and the sequence should play on reset. Otherwise, connect a piezo transducer, as seen in Fig. 4.12, between pins 1 and 26. Let the program run. The transducer seems very quiet when held in air. You can increase the volume significantly by fixing or holding it to a flat surface like a table top.

■ Exercise 4.8 (For App Board or Breadboard)

Try the following:

1. Make the "Oranges and Lemons" sequence play an octave higher by doubling the frequency of each note.
2. Change the tempo by modifying the multiplier in the wait command.
3. (For the more musically inclined) Change the tune, so that the mbed plays the first line of "Twinkle Twinkle Little Star". This uses the same notes, except it also needs F#, of frequency 740 Hz. The tune starts on A. Because there are repeated notes, consider putting a small pause between each note.

■

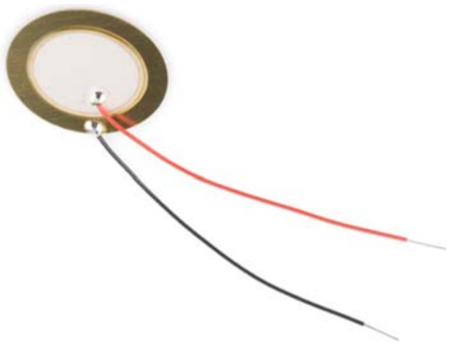


Figure 4.12
A piezo transducer. *Image courtesy of Sparkfun.*

Chapter Review

- A DAC converts an input binary number to an output analog voltage, which is proportional to that input number.
- DACs are widely used to create continuously varying voltages, for example, to generate analog waveforms.
- The mbed has a single DAC and associated set of library functions.
- PWM provides a way of controlling certain analog quantities, by varying the pulse width of a fixed frequency rectangular waveform.
- PWM is widely used for controlling flow of electrical power, for example LED brightness or motor control.
- The mbed has six possible PWM outputs. They can all be individually controlled but must all share the same frequency.

Quiz

1. A 7-bit DAC obeys Eq. (4.1) and has a voltage reference of 2.56 V.
   a. What is its resolution?
   b. What is its output if the input is 100 0101?
   c. What is its output if the input is 0x2A?
   d. What is its digital input in decimal and binary if its output reads 0.48 V?
2. What is the mbed's DAC resolution and what is the smallest analog voltage step increase or decrease which can be output from the mbed?
3. What is the output of the LPC1768 DAC, if its input digital word is
   a. 00 0000 1000
   b. 0x80
   c. 10 1000 1000?

4. What output voltages will be read on a DVM while this program loop runs on the mbed?

```
while(1){
  for (i=0;i<1;i=i+0.2){
    Aout=i;
    wait(0.1);
  }
}
```

5. The program in Question 4 gives a crude saw tooth waveform. What is its period?
6. What are the advantages of using pulse width modulation (PWM) for control of analog actuators?
7. A PWM data stream has a frequency of 4 kHz and duty cycle of 25%. What is its pulse width?
8. A PWM data stream has period of 20 ms and on time of 1 ms. What is its duty cycle?
9. The PWM on an mbed is set up with these statements. What is the on time of the waveform?

```
PWM1.period(0.004);  // set PWM period
PWM1=0.75;           // set duty cycle
```

10. How long does Program Example 4.7 take to play through the tune once? Calculate this by checking the program, then measure it as the program plays.

References

[1] Walt Kestner. Newnes (Ed.), The Data Conversion Handbook, Analog Devices Inc., 2005.