CHAPTER 5

Analog Input

5.1 Analog-to-Digital Conversion (ADC)

The world around the embedded system is largely an analog one, and sensors—of temperature, sound, acceleration, and so on—mostly have analog outputs. Yet it is essential for the microcontroller to have these signals available in digital form. This is where the *analog-to-digital converter* comes in. We can convert analog signals into digital representation, with an accuracy determined by the ADC. Having performed this analog-to-digital conversion, we can then use the microcontroller to process or analyze this information, based on the value of the analog input.

5.1.1 The ADC

An ADC is an electronic circuit whose digital output is proportional to its analog input. Effectively it "measures" the input voltage and gives a binary output number proportional to its size. The list of possible analog input signals is endless, including such diverse sources as audio and video, medical or climatic variables, and a host of industrially generated signals. Of these some, like temperature, change very slowly. Others, like sound and other vibrations, are periodic, with a frequency range up to tens of kilohertz. Still others, like video or radar, have a very high frequency content. These examples display very different signal characteristics, and it is not surprising to discover that many types of ADC have been developed, with characteristics optimized for these differing applications.

The ADC almost always operates within a larger environment, often called a *data acquisition system*. Some features of a general purpose data acquisition system are shown in Fig. 5.1. To the right of the diagram is the ADC itself. This has an analog input and digital output. It is under computer control; the computer can start a conversion. The conversion takes finite time, maybe some microseconds or more, so the ADC needs to signal when it has finished. The output data can then be read. The ADC works with a voltage reference—an accurate and stable voltage source. Think of this as a ruler or tape measure. In one way or other the ADC compares the input voltage with the voltage reference, and comes up with the output number, based on this comparison. As with so many digital or digital/analog subsystems, there is also a clock input—a continuously running square wave, which sequences the internal operation of the ADC. The clock frequency, subject to its own set of constraints, determines how fast the ADC operates.

Fast and Effective Embedded Systems Design. http://dx.doi.org/10.1016/B978-0-08-100880-5.00005-0

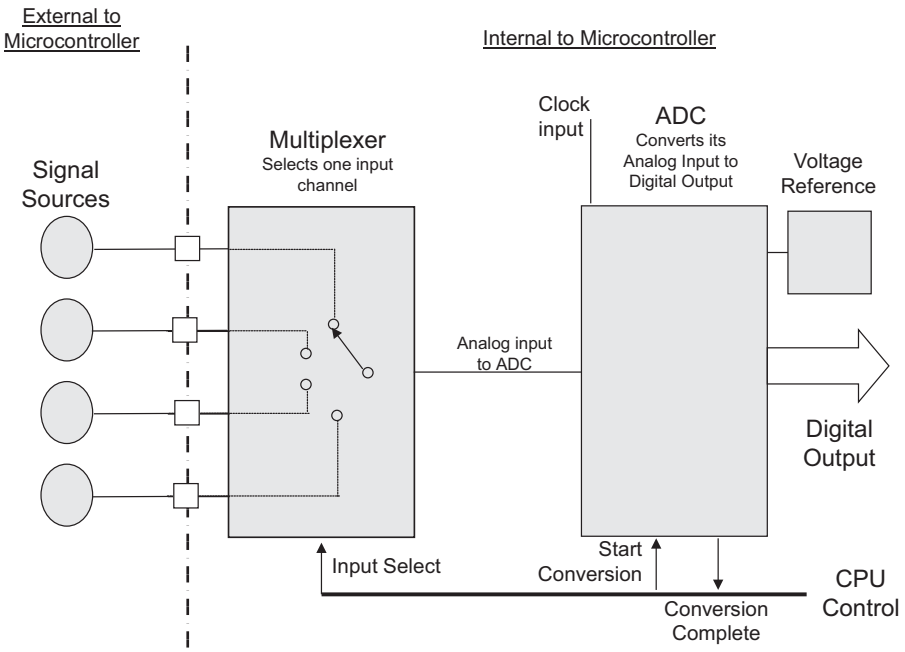

Figure 5.1
An example data acquisition system.

Once we start working with an ADC, we usually find that we want to work with more than one signal. We *could* just use more ADCs, but this is costly and takes up semiconductor space. Instead, the usual practice is to put an *analog multiplexer* in front of the ADC. This acts as a selector switch. The user can then select any one of several inputs to the ADC. If this is done quickly enough, it's as if all inputs are being converted at the same time. Many microcontrollers, including the LPC1768, include an ADC and multiplexer on chip. The inputs to the multiplexer are connected to microcontroller pins, and multiple inputs can be used. This is shown in Fig. 5.1, for a 4-bit multiplexer. We return to some of the detail of Fig. 5.1 in Chapter 14.

5.1.2 Range, Resolution, and Quantization

Many ADCs obey Eq. (5.1), where V_i is the input voltage; V_r the reference voltage; n the number of bits in the converter output; and D the digital output value. The output binary number D is an integer, and for an n-bit number can take any value from 0 to $(2^n - 1)$. The internal ADC process effectively rounds or truncates the calculation in Eq. (5.1) to produce an integer output. Clearly, the ADC can't just convert any input voltage but has maximum and minimum permissible input values. The difference between this maximum and minimum is called the *range*. Often the minimum value is 0 V, so the range is then just the maximum possible input value. Analog inputs which exceed the maximum or

minimum permissible input values are likely to be digitized as the maximum and minimum values respectively, i.e., a limiting (or "clipping") action takes place. The input range of the ADC is very directly linked to the value of the voltage reference. In many ADC circuits the range is actually equal to the reference voltage; this is what is assumed in the equation, hence V_r stands for both reference voltage and range.

$$D = \frac{V_i}{V_r} \times 2^n \tag{5.1}$$

Eq. (5.1) is represented in graphical form in Fig. 5.2, for a 3-bit converter. If the input voltage is gradually increased from 0 V, and the converter is running continuously, then the ADC's output is initially 000. If the input slowly increases, there comes a point when the output will take the value 001. As the input increases further, the output changes to 010, and so on. At some point, it reaches 111, i.e., 7 in decimal, or $(2^3 - 1)$. This is the maximum possible output value. The input may be increased further, but it cannot force any increase in output value.

As Fig. 5.2 demonstrates, by converting an analog signal to digital, we run the risk of approximating it. This is because any one digital output value has to represent a small range of analog input voltages, i.e., the width of any of the steps on the "staircase" of Fig. 5.2. For example, the digital output 001 in the figure must represent *any* analog voltage along the step that it represents. The width of this step is called the *resolution* of the ADC; resolution is a measure of how precisely an ADC can convert and represent a given input voltage. Clearly, the more steps we have representing the range, the narrower the steps will be, and hence the resolution is reduced (and improved). We get more steps

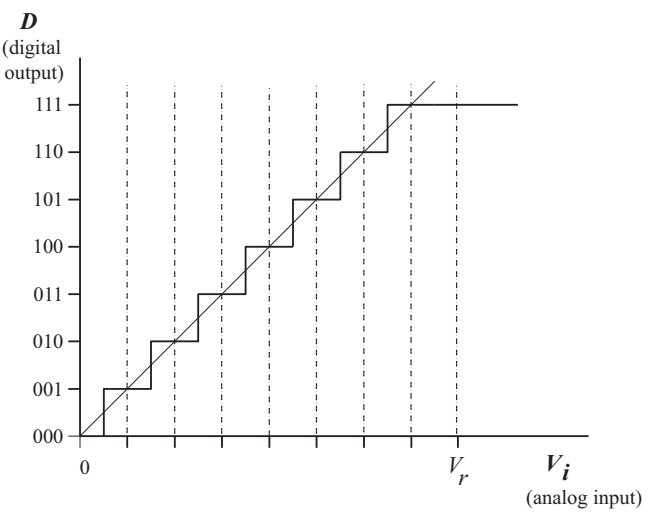

Figure 5.2
A 3-bit ADC characteristic.

by increasing the number of bits in the ADC process. Hence a common shorthand is to say that a certain ADC has, for example, a 12-bit resolution. Increasing the number of bits inevitably increases the complexity and cost of the ADC, and usually the time it takes to complete a conversion. Eq. (5.2) shows how resolution relates to range and number of bits, for an ideal ADC.

$$\text{resolution} = \frac{V_{\text{r}}}{2^n} \tag{5.2}$$

To take this discussion a little further, as an example, let us return to the step represented by 001 in Fig. 5.2. If the output value of 001 is precisely correct for the input voltage at the middle of the step, then as we move away from the center, a measurement error occurs. The greatest error appears at either end of the step. This is called *quantization error*. Following this line of reasoning, the greatest quantization error is one half of the step width, i.e., one half of the resolution, or half of one least significant bit (LSB) equivalent of the voltage scale.

As an example, if we want to convert an analog signal that has a range 0−3.3 V to an 8-bit digital signal, then there are 256 (i.e., 2^8) distinct output values. Each step has a width of 3.3/256 = 12.89 mV, which is the ADC resolution. The worst case quantization error is half of this, i.e., 6.45 mV. As far as the mbed is concerned, Fig. 2.3 shows that the LPC1768 ADC is 12-bit. This leads to a resolution of $3.3/2^{12}$, or 0.8 mV, with a worst case quantization error of 0.4 mV.

For many applications, an 8, 10, or 12-bit ADC allows sufficient resolution; it all depends on the required accuracy. For example, in certain audio applications, listening tests have demonstrated that 16-bit resolution is adequate; improved quality can, however, be noticed using 24-bit conversion.

All of this assumes that all other aspects of the ADC are perfect, which they aren't. The reference voltage can be inaccurate, or can drift with temperature, and the staircase pattern can have nonlinearities. Furthermore, different methods of analog-to-digital conversion each bring their unique inaccuracies to the equation. Ref. [1] of Chapter 4 and Ref. [1] of this chapter discuss in detail a number of analog-to-digital conversion designs, including *successive approximation*, *flash*, *dual slope*, and *delta-sigma* methods, which all have their own relative advantages. A knowledge of the specific design of the ADC however, is not necessary for us to understand the main concepts of data conversion, and then to apply these to the mbed's ADC.

5.1.3 Sampling Frequency

When converting a changing analog signal to digital form, in most cases a "sample" is taken repeatedly, as illustrated in Fig. 5.3. Generally sampling occurs at a fixed rate, called

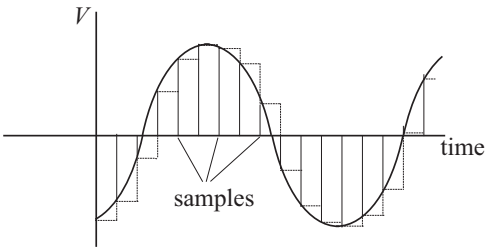

Figure 5.3
Digitizing a sine wave.

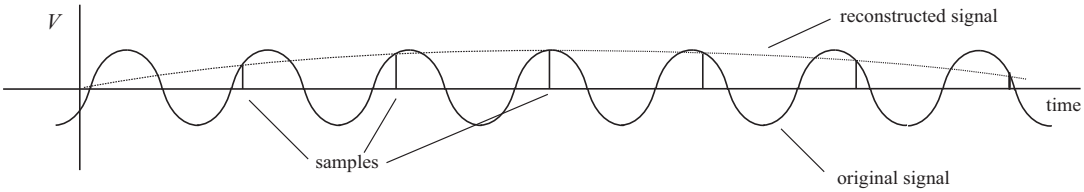

Figure 5.4
The effect of aliasing.

the *sampling frequency*. Once a sample is taken, it represents the value of the signal, until the next one is taken. The more samples taken, the more accurate the digital representation is likely to be.

The *minimum* sampling frequency depends on the maximum frequency of the signal being digitized. If the sampling frequency is too low, then rapid changes in the analog signal will not be represented in the resulting digital data. The *Nyquist sampling criterion* states that the sampling frequency must be at least double that of the highest signal frequency. For example, the human auditory system is known to extend up to approximately 20 kHz, so standard audio CDs are sampled and played back at 44.1 kHz to adhere to the Nyquist sampling criterion. If the sampling criterion is not satisfied, then a phenomenon called *aliasing* occurs—a new lower frequency is generated. This is illustrated in Fig. 5.4 and demonstrated later in the chapter. Aliasing is very damaging to a signal and must always be avoided. A common approach is to use an *antialiasing filter*, which limits all signal components to those which satisfy the sampling criterion.

5.1.4 Analog Input With the mbed

App Board

Fig. 2.1 shows us that the mbed has up to six analog inputs, on pins 15–20; one of these, pin 18, can also be the analog output, as described in Chapter 4. Meanwhile, Fig. 2.8 (or Table 2.1) shows how these connections are used on the

Table 5.1: API summary for analog input.

Functions	Usage
`AnalogIn`	Create an AnalogIn object, connected to the specified pin
`read`	Read the input voltage, represented as a float in the range (0.0—1.0)
`read_u16`	Read the input voltage, represented as an unsigned short in the range (0x0—0xFFFF)

application board. Pins 19 and 20 are usefully connected to two on-board potentiometers, while pins 17 and 18 are available externally through two audio jack connectors (items 4 and 7 in Fig. 2.7); these sit between the potentiometers on the board. Pins 15 and 16 are hard-wired to the joystick, so are not available for analog input.

The analog input API summary, following a pattern which is quite familiar, is shown in Table 5.1. It's useful to note that the ADC output is available either as an unsigned integer (as it would be at the ADC output) or as a floating point number.

5.2 Combining Analog Input and Output

The ADC is an input device, which transfers data *into* the microcontroller. If used on its own, we will have no idea of what output values it has created. We now therefore go on to do two things to make that data visible. We will first of all use the ADC output values to immediately control an output variable, for example, the DAC or pulse width modulation (PWM). We will later transfer its output values to the PC screen and explore some measurement applications.

5.2.1 Controlling LED Brightness by Variable Voltage

Let's start with a simple program which reads the analog input and uses it to control the brightness of an LED by varying the voltage drive to the LED. Here we will use a potentiometer to generate the analog input voltage and will then pass the value read straight to the analog output.

Connect up the circuit of Fig. 5.5. This uses pin 20 as the analog input, connecting the potentiometer across 0—3.3 V. The LED is connected to pin 18, the analog output. Start a new program and copy into it the very simple code of Program Example 5.1. This just sets up the analog input and output and then continuously transfers the input to the output.

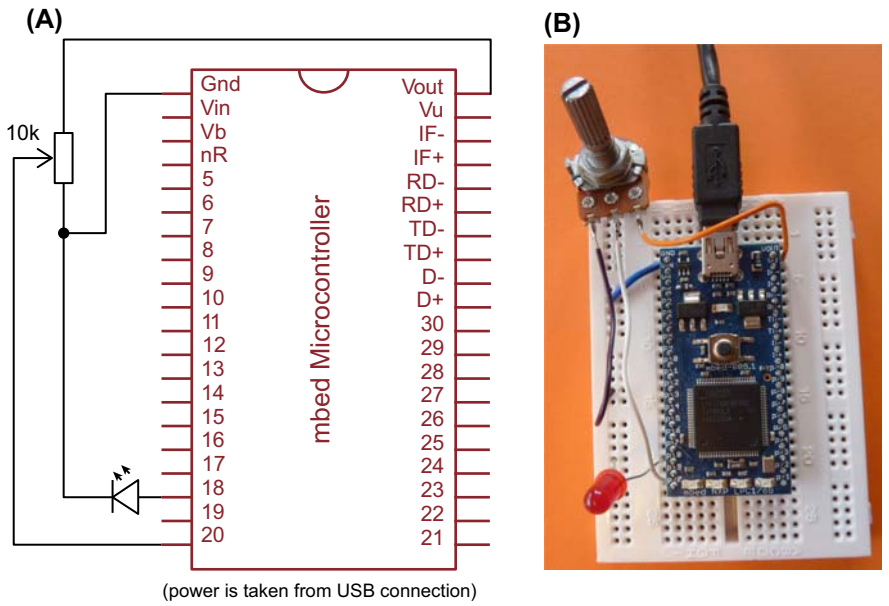

Figure 5.5
A potentiometer controlling LED brightness. (A) Circuit diagram and (B) construction detail.

```
/*Program Example 5.1: Uses analog input to control LED brightness, through DAC
output
                                                                          */

#include "mbed.h"
AnalogOut Aout(p18);      //defines analog output on Pin 18
AnalogIn Ain(p20)         //defines analog input on Pin 20

int main() {
  while(1) {
    Aout=Ain;   //transfer analog in value to analog out, both are type float
  }
}
```

Program Example 5.1: Controlling LED brightness by variable voltage

Compile the program and download to the mbed. With the program running, the potentiometer should control the brightness of the LED. You will probably find, however, that there is a range of the potentiometer rotation where the LED is off. The LED will be following the curve of Fig. 3.3A, and there will be very little illumination when the drive voltage is low.

Exercise 5.1 (For Breadboard)

Measure the DAC output voltage at pin 18, as you adjust the potentiometer. You will find that when this is above around 1.8 V, the LED will be lit, with varying levels of brightness. When it is below 1.8 V, the LED no longer conducts, and there is no illumination.

■

Exercise 5.2 (For App Board)

Adapt this application to the app board. One of the on-board potentiometers can be used as analog input. An external LED will need to be connected to the analog output connector, as the on-board LEDs are hard-wired to other pins.

■

5.2.2 Controlling LED Brightness by PWM

App Board

We can use the potentiometer input to alter the PWM duty cycle, using the same approach as we did for the analog output. Program Example 5.2 will run on the app board, lighting the red LED. Alternatively, use the circuit of Fig. 5.5, except that the LED should now be connected to the PWM output on pin 23. Create a new program and enter the code of Program Example 5.2. Here we see the analog input value being transferred to the PWM duty cycle.

```
/*Program Example 5.2: Uses analog input to control PWM duty cycle, fixed period
                                                                        */
#include "mbed.h"
PwmOut PWM1(p23);
AnalogIn Ain(p20);          //defines analog input on Pin 20

int main() {
  while(1){
    PWM1.period(0.010);  // set PWM period to 10 ms
    PWM1=Ain;            //Analog in value becomes PWM duty, both are type float
    wait(0.1);
  }
}
```

Program Example 5.2: Controlling PWM pulse width with potentiometer

The LED brightness should again be controlled by the potentiometer. While the outcome is very similar to the previous program, the means of doing it is quite different. In practice

we would normally not wish to commit a whole DAC to controlling the brightness of an LED, but would be more ready to make use of the simpler PWM source.

5.2.3 Controlling PWM Frequency

App Board

Instead of using the potentiometer to control the PWM duty cycle, we can use it to control the PWM frequency. Use the same hardware as in the previous section, either app board or breadboard. Create a new program and enter the code of Program Example 5.3.

Notice that the PWM period is calculated in the line:

```
PWM1.period(Ain/10+0.001);   // set PWM period
```

C code feature

It's first worth noting that a calculation is placed where we might have expected to find a simple parameter. This isn't a problem for C. The program will first evaluate the expression inside the brackets and then call the **period( )** function. This calculation invokes for the first time the divide operator, /. It also raises the thorny little question, which all children face when they learn arithmetic, of what order operators should be evaluated. In C, this is very clearly defined and can be seen by checking Table B.5 in Appendix B. This shows a precedence for each operator, with / having precedence 3, and + having precedence 4. Therefore, the division will be done before the addition, and there will be no uncertainty in evaluating the expression. The values used mean that the minimum period, when **Ain** is zero, is 0.001 s (i.e., 1000 Hz). The maximum is when **Ain** is 1, leading to a period of 0.101 s, i.e., around 10 Hz.

```
/*Program Example 5.3: Uses analog input to control PWM period.
                                                                    */
#include "mbed.h"
PwmOut PWM1(p23);
AnalogIn Ain(p20);

int main() {
    while(1){
      PWM1.period(Ain/10+0.001);   // set PWM period
      PWM1=0.5;                    // set duty cycle
      wait(0.5);
    }
}
```

Program Example 5.3: Controlling PWM frequency with potentiometer

When running the program you should be able to see the frequency change as the potentiometer is adjusted.

Exercise 5.3 (For App Board or Breadboard)

Observe the PWM waveform on an oscilloscope, setting the time base initially to 5 ms/div.

1. Adjust the values in the PWM period calculation to give different ranges of frequency output.
2. At what frequency does the LED appear not to flash, but seems to be continuously on? Measure this as carefully as you can and see if the perceived frequency varies between different people. This is an important question, as knowing its value allows us to know at what frequency we can "trick" the eye into thinking that a flashing image is continuous, for example, in conventional raster scan TV screens, oscilloscope traces, and multiplexed LED displays.

■

The 0.5 s delay in Program Example 5.3 is added to the loop so that each new PWM period is implemented, before the next update. If it was omitted, then the PWM would potentially be updated repeatedly within each cycle, which would lead to a large amount of instability or *jitter* in the output. Try removing the delay to see the effect. Notice there can be discontinuities in the PWM output as the frequency values are updated. With care (and especially if you're using a storage oscilloscope) you can see this as the PWM is updated every 0.5 s. This is one reason why it is good to fix the frequency for a PWM signal.

Exercise 5.4 (For App Board or Breadboard)

Connect a servo to the mbed as indicated in Fig. 4.10 (either app board or breadboard), with potentiometer connected as in Section 5.2.1. Write a program which allows the potentiometer to control servo position. Scale values so that the full range of potentiometer adjustment leads to the full range of servo position change.

■

5.3 Processing Data From Analog Inputs

5.3.1 Displaying Values on the Computer Screen

We turn now to the second way of making use of the ADC output, promised at the beginning of Section 5.2. It is possible to read analog input data through the ADC and then print the value to the PC screen. This is a very important step forward, as it gives the possibility of displaying on the computer screen any data we're working within the mbed. To do this, both mbed and host computer need to be configured correctly to send and

receive the data, and we need the host computer to be able to display that data. For the computer a *terminal emulator program* is needed, also called a *host terminal*. The mbed site recommends use of Tera Term for Microsoft Windows users, and CoolTerm for Apple OS X developers; Appendix E explains how to configure either of these. Once in place, the mbed can be made to appear to the computer as a serial port, communicating through the USB connection. It links up with the USB through one of its own asynchronous serial ports. This can be set up simply by adding this program line:

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

There is further explanation of this in Section 7.8.3.

C code feature

Start a new mbed project and enter the code of Program Example 5.4. Use either app board or breadboard (in which case leave the potentiometer connected as in Fig. 5.5). Writing to the computer and hence the terminal emulator is achieved using the **printf( )** function. We see this for the first time, along with some of its far-from-friendly format specifiers. Check Section B9 for some background on this.

```
/*Program Example 5.4: Reads input voltage through the ADC and transfers to PC
terminal. Works for either App Board or Breadboard.
                                                              */
#include "mbed.h"
Serial pc(USBTX, USBRX);              //enable serial port which links to USB
AnalogIn Ain(p20);

float ADCdata;

int main() {
  pc.printf("ADC Data Values...\n\r");   //send an opening text message
  while(1){
    ADCdata=Ain;
    pc.printf("%1.3f \n\r",ADCdata);  //send the data to the terminal
    wait(0.5);
  }
}
```

Program Example 5.4: Logging data to the PC

You should now be able to compile and run the code to give an output on Tera Term or CoolTerm. If you have problems, check from Appendix E or the mbed site that you have set up the host terminal correctly.

5.3.2 Scaling ADC Outputs to Recognized Units

The data displayed through Program Example 5.4 is just a set of numbers proportional to the voltage input, in the range 0−1. Yet they represent a range of voltages in the range 0−3.3. The numbers can therefore readily be scaled to give a voltage reading, by

Figure 5.6
Logged data on Tera Term.

multiplying by 3.3. Substitute the code lines below into the **while** loop of Program Example 5.4 to do just this and to place a unit after the voltage value.

```
ADCdata=Ain*3.3;            //read and scale the data
pc.printf("%1.3f",ADCdata); //send the data to the terminal
pc.printf(" V\n\r");        // insert a unit
wait(0.5);
```

Run the adjusted program, its output should appear similar to Fig. 5.6. View the measured voltage on the PC screen and read the actual input voltage on a digital voltmeter. How well do they compare?

5.3.3 Applying Averaging to Reduce Noise

If you leave Program Example 5.4 running, with a fixed input and values displayed on Tera Term or CoolTerm, you may be surprised to see that the measured value is not always the same but varies around some average value. You may already have noticed that the PWM value in Sections 5.2.2 or 5.2.3 also appeared to vary, even when the potentiometer was not being moved. Several effects may be at play here, but almost certainly you are seeing the effect of some interference, and all the problems it can bring. If you look with the oscilloscope at the ADC input (i.e., the "wiper" of the potentiometer) you are likely to see some high frequency noise superimposed on this; exactly how much will depend on what equipment is running nearby, how long your interconnecting wires are, and a number of other things.

A very simple first step to improve this situation is to average the incoming signal. This should help to find the underlying average value and remove the high frequency noise element. Try inserting the **for** loop shown below, replacing the `ADCdata=Ain;` line in

Program Example 5.4. You will see that this code fragment sums 10 ADC values and takes their average. Try running the revised program and see if a more stable output results. Note that while this sort of approach gives some benefit, the actual measurement now takes 10 times as long. This is a very simple example of digital signal processing.

```
for (int i=0;i<=9;i++) {
  ADCdata=ADCdata+Ain*3.3;    //sum 10 samples
}
ADCdata=ADCdata/10;           //divide by 10
```

5.4 Some Simple Analog Sensors

Now that we are equipped with analog input, it is appropriate to explore some simple analog sensors. For now, these are the simpler and more traditional ones, which have an analog output voltage that can be connected to the mbed ADC input. Later in the book, other sensors will be introduced, which can communicate with the mbed by digital interface. Although the application board has a number of sensors, they all communicate digitally. It's simplest, therefore, if the following builds are done with a breadboard.

5.4.1 The Light-Dependent Resistor

The *light-dependent resistor* (LDR) is made from a piece of exposed semiconductor material. When light falls on it, its energy flips some electrons out of the crystalline structure; the brighter the light, the more electrons are released. These electrons are then available to conduct electricity, with the result that the resistance of the material falls. If the light is removed the electrons pop back into their place, and the resistance goes up again. The overall effect is that as illumination increases, the LDR resistance falls.

The NORPS-12 LDR [2], made originally by Silonex, is readily available and low cost. It is shown in Fig. 5.7, connected in a simple potential divider, giving a voltage output. Indicative data appear in Table 5.2. This shows that it has a resistance when completely dark of at least 1.0 MΩ, falling to a few hundred ohms when very brightly illuminated. The value of the series resistor, shown here as 10 kΩ, is chosen to give an output value of approximately mid-range for normal room light levels. It can be adjusted to modify the output voltage range. Putting the LDR at the bottom of the potential divider, as shown here, gives a low output voltage in bright illumination and a high one in low illumination. This can be reversed by putting the LDR at the top of the divider.

The LDR is a simple, effective, and low-cost light sensor. Its output is not however linear, and each device tends to give slightly different output from another. Hence it isn't used for precision measurements.

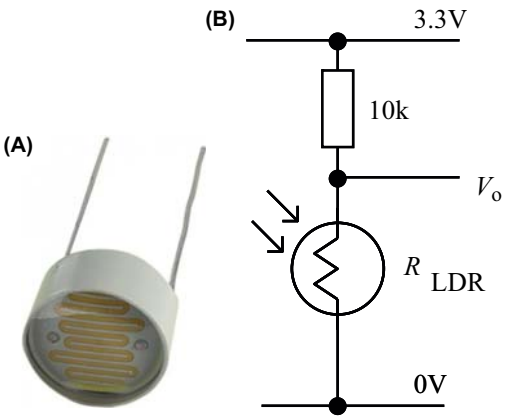

Figure 5.7
The NORPS-12 Light-dependent resistor. (A) The NORPS-12 LDR and (B) connected in a potential divider.

Table 5.2: NORPS-12 LDR—indicative resistance and output values.

Illumination (lux)	R_{LDR} (Ω)	V_o (V)
Dark	≥ 1.0 M	≥ 3.27
10	9 k	1.56
1000	400	0.13

Exercise 5.5

Using the circuit of Fig. 5.7, connect a NORPS-12 LDR to the mbed on a breadboard. Use any analog input. Write a program to display light readings on the Tera Term or CoolTerm screen. You will not be able to scale these into any useful unit. Try reversing resistor and LDR and note the effect.

■

5.4.2 Integrated Circuit Temperature Sensor

Semiconductor action is highly dependent on temperature, so it's not surprising that semiconductor temperature sensors are made. A very useful form of sensor is one which is contained in an integrated circuit, such as the LM35, Fig. 5.8. This device has an output of 10 mV/°C, with operating temperature up to 110°C (for the LM35C version). It is thus immediately useful for a range of temperature-sensing applications. The simplest connection for the LM35, which we can use with the mbed, is shown in Fig. 5.8. A range

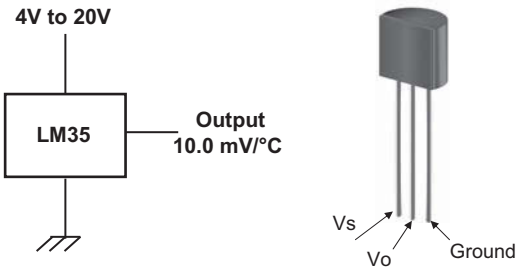

Figure 5.8
The LM35 integrated circuit temperature sensor.

of more advanced connections, for example, to get an output for temperatures below 0°C are shown in the data sheet [3].

Exercise 5.6

Design, build, and program a simple temperature measurement system using an LM35 sensor, which displays temperature on the computer screen. The V_S pin of the sensor is the power supply (4–20 V), which can be connected to pin 39 of the mbed. When connecting the sensor, it is possible to plug it directly into a suitable location in the mbed breadboard. Each terminal can, however, also be soldered to a wire, so that remote sensing can be undertaken. If these wires are insulated appropriately, for example, with silicone rubber at the sensor end, then the sensor can be used to measure liquid temperatures.

Noting its maximum operating temperature, how well does the LM35C exploit the input range of the mbed ADC?

■

5.5 Exploring Data Conversion Timing

Nyquist's sampling theorem suggests that a slow ADC will only be able to convert low frequency signals. When designing a carefully specified system it's therefore very important to know how long each data conversion takes. For this reason it's interesting to make a measurement of mbed ADC and DAC conversion times and then put Nyquist to the test.

5.5.1 Estimating Conversion Time and Applying Nyquist

Program Example 5.5 provides a very simple mechanism for measuring conversion times and then viewing Nyquist's sampling theorem in action. It adapts Program Example 5.1 but pulses a digital output between each stage. Enter this as a new program, compile, and run.

```
/*Program Example 5.5: Inputs signal through ADC and outputs to DAC. View DAC
output on oscilloscope. To demonstrate Nyquist, connect variable frequency signal
generator to ADC input. Allows measurement of conversion times and explores Nyquist
limit.                                                                    */

#include "mbed.h"
AnalogOut Aout(p18);       //defines analog output on Pin 18
AnalogIn Ain(p20);         //defines analog input on Pin 20
DigitalOut test(p5);
float ADCdata;

int main() {
  while(1) {
    ADCdata=Ain;   //starts A-D conversion, and assigns analog value to ADCdata
    test=1;        //switch test output, as time marker
    test=0;
    Aout=ADCdata;  // transfers stored value to DAC, and forces a D-A conversion
    test=1;        //a double pulse, to mark the end of conversion
    test=0;
    test=1;
    test=0;
    //wait(0.001);    //optional wait state, to explore different cycle times
   }
}
```

Program Example 5.5: Estimating data conversion times

This program allows us to make a number of measurements which are of very great importance, and which require careful use of the oscilloscope. The measurements are presented as the two exercises which follow.

Exercise 5.7

Running Program Example 5.5, observe carefully the waveform displayed by the "test" output (i.e., pin 5) on an oscilloscope—a digital storage oscilloscope will give best results. This may require some patience, they are very narrow pulses. You can widen the pulses if needed by inserting a wait while the output is high. Note that for this test, you don't need anything connected to the ADC input.

You will be able to detect the single pulse at the end of the analog-to-digital conversion and the double one at the end of the loop. Measure the time duration of the analog-to-digital conversion and the digital-to-analog conversion. What comment can you make on these? Note that the conversion times you measure are not the actual conversion times of the ADC and DAC themselves; they include all associated programming overheads. Keep a note of these values as we aim to account for them later in Exercises 5.8 and 14.8.

■

Exercise 5.8

Armed with the knowledge of the conversion times, connect a signal generator as input to the ADC. Set the signal amplitude so that it is just under 3.3 V peak-to-peak and apply a DC offset so that the voltage value never goes below 0 V. This facility is available on most signal generators. Insert the **wait(0.001);** line at the end of the loop ("commented out" in Program Example 5.5). This will give a sampling frequency of a little below 1 kHz. Nyquist's sampling theorem predicts that the maximum signal frequency that we can digitize will be 500 Hz, for this sampling frequency. Let's test it.

Start initially with an input signal of around 200 Hz. Observe input signal and DAC output on the two beams of the oscilloscope. You should see the input signal, and a reconstructed version of it, something like Fig. 5.3, with a new conversion approximately every millisecond. Now gradually increase the signal frequency toward 500 Hz. As you approach Nyquist's limit the output becomes a square wave. When input frequency equals sampling frequency, a straight line on the oscilloscope should occur, though, in practice, it may be difficult to find this condition exactly. As the input frequency increases further, an *alias* signal (as illustrated in Fig. 5.4) appears at the output.

Decrease the duration of the wait state, and predict and observe the new Nyquist frequency. Finally, remove the wait state altogether. The data conversion should now be taking place at the fastest possible rate, with conversion time corresponding to your earlier measurement. The Nyquist limit that you now find is the limit for this particular hardware/software configuration.

■

5.6 Mini Projects

5.6.1 Two-Dimensional Light Tracking

Light tracking devices are very important for the capture of solar energy. Often they operate in three dimensions, and tilt a solar panel so that it is facing the sun as accurately as possible. To start rather more simply, create a two-dimensional light tracker by fitting two LDRs, angled away from each other by around 90 degree, to a servo. Connect the LDRs using the circuit of Fig. 5.7B to two ADC inputs. Write a program which reads the light value sensed by the two LDRs and rotates the servo so that each is receiving equal light. The servo can of course only rotate 180 degree. This is not, however, unreasonable, as a sun-tracking system will be located to track the sun from sunrise to sunset, i.e., not more than 180 degree. Can you think of a way of meeting this need using only one ADC input?

5.6.2 Temperature Alarm

Using an LM35 and a piezo transducer (Fig. 4.12), make a temperature alarm. Define two threshold temperatures, which should be above room temperature, but not hazardous. When the lower threshold is passed, the transducer should beep at a slow rate, say once per second. When the higher one is passed, it should beep at a fast rate, say 10 times a second. Example thresholds could be 26 and 32°C. Then, assuming a room temperature of around 20°C, it should be possible to hand warm the sensor to these temperatures. Those working in hotter environments may wish to adjust these temperatures. Different heat sources and temperatures can also be explored, at all times ensuring safe operating conditions are maintained.

Chapter Review

- An ADC is available in the mbed; it can be used to digitize analog input signals.
- It is important to understand ADC characteristics, in terms of input range, resolution, and conversion time.
- Nyquist's sampling theorem must be understood and applied with care when sampling AC signals. The sampling frequency must be at least twice that of the highest frequency component in the sampled analog signal.
- Aliasing occurs when the Nyquist criterion is not met, this can introduce false frequencies to the data. Aliasing can be avoided by introducing an antialiasing filter to the analog signal before it is sampled.
- Data gathered by the ADC can be further processed and displayed or stored.
- There are numerous sensors available which have an analog output; in many cases, this output can be directly connected to the mbed ADC input.

Quiz

1. Give three types of analog signal which might be sampled through an ADC.
2. An ideal 8-bit ADC has an input range of 5.12 V. What is its resolution and greatest quantization error?
3. Give an example of how a single ADC can be used to sample four different analog signals.
4. An ideal 10-bit ADC has a reference voltage of 2.048 V and behaves according to Eq. (5.1). For a particular input its output reads 10 1110 0001. What is the input voltage?
5. What will be the result if an mbed is required to sample an analog input value of 4.2 V?
6. An ultrasound signal of 40 kHz is to be digitized. Recommend the minimum sampling frequency.

7. The conversion time of an ADC is found to be 7.5 μs. The ADC is set to convert repeatedly, with no other programming requirements. What is the maximum frequency signal it can digitize?
8. The ADC in Question 7 is now used with a multiplexer, so that 4 inputs are repeatedly digitized in turn. A further time of 2500 ns per sample is required, to save the data and switch the input. What is the maximum frequency signal that can now be digitized?
9. An LM35 temperature sensor is connected to an mbed ADC input and senses a temperature of 30°C. What is the binary output of the ADC?
10. What will be the value of integer **x** for input values of 1.5 and 2.5 V sampled by the mbed using the following program code?

```
#include "mbed.h"
AnalogIn Ain(p20);
int main(){
    int x=Ain.read_u16();
}
```

References

[1] P. Horowitz, W. Hill, The Art of Electronics, third ed., Cambridge University Press, 2016.
[2] The NORPS-12 data sheet. http://www.farnell.com/datasheets/409710.pdf.
[3] LM35 Precision Centigrade Temperature Sensors, Texas Instruments, January 2016. http://www.ti.com/lit/ds/symlink/lm35.pdf.