CHAPTER 11

An Introduction to Digital Signal Processing

Chapter Outline

11.1 What is a Digital Signal Processor?

Digital signal processing (DSP) refers to the computation of mathematically intensive algorithms applied to data signals, such as audio signal manipulation, video compression, data coding/decoding and digital communications. A digital signal processor, also informally called a DSP chip, is a special type of microprocessor used for DSP applications. A DSP chip provides rapid instruction sequences, such as *shift-and-add* and *multiply-and-add* (sometimes called *multiply-and-accumulate* or MAC), which are commonly used in signal processing algorithms. Digital filtering and frequency analysis with the Fourier transform requires many numbers to be multiplied and added together, so a DSP chip provides specific internal

Fast and Effective Embedded Systems Design. DOI: 10.1016/B978-0-08-097768-3.00011-8

hardware and associated instructions to make these operations rapid in operation and easier to code in software.

A DSP chip is therefore particularly suitable for number crunching and mathematical algorithm implementations. It is possible to perform DSP applications with any microprocessor or microcontroller, although specific DSP chips will outperform a standard microprocessor with respect to execution time and code size efficiency.

11.2 Digital Filtering Example

Filters are used to remove chosen frequencies from a signal, as shown in Figure 11.1, for example. Here, there is a signal with both low-frequency and high-frequency components. We may wish to remove either the low-frequency component, by implementing a *high-pass filter* (HPF), or the high-frequency component, by implementing a *low-pass filter* (LPF). A filter

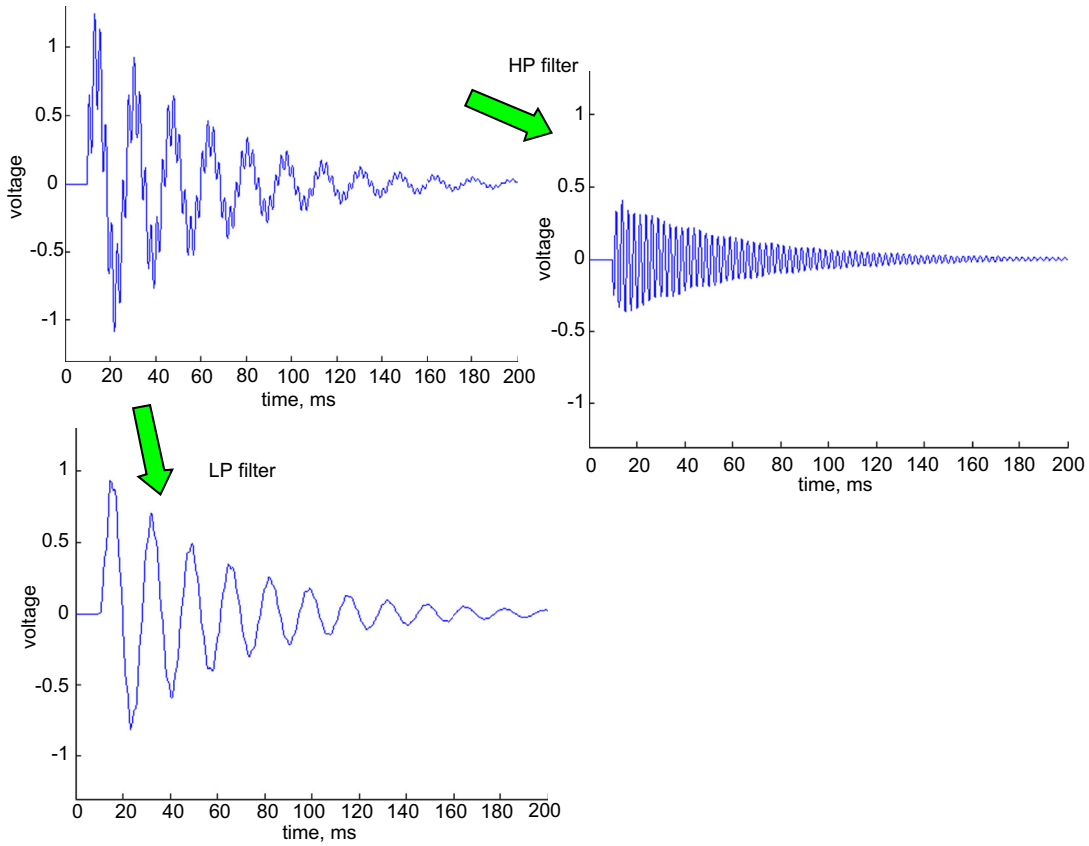


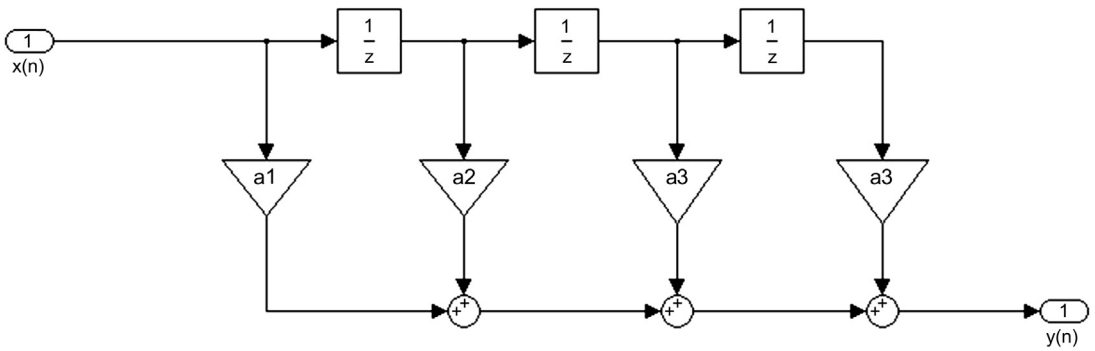
Figure 11.1:
High-pass and low-pass filtered signals

has a *cut-off frequency*, which determines which signal frequencies are in the *pass-band* (and are still evident after the filtering) and which are in the *stop-band* (which are removed by the filtering operation). The cut-off effect is not perfect, however, as frequencies in the stop-band are attenuated more the further away from the cut-off frequency they are. Filters can be designed to have different steepness of cut-off attenuation and so adjust the filter's *roll-off rate*. In general, the more complex the filter design, the steeper its roll-off can be. The filtering can be performed with an active or a passive analog filter, but the same process can also be performed in software with a DSP operation.

We will not go deeply into the mathematics of digital filtering, but the software process relies heavily on addition and multiplication. Figure 11.2 shows an example block diagram for a simple digital filtering operation.

Figure 11.2 shows that coefficient a_1 is multiplied by the most recent sample value. Coefficient a_2 is multiplied by the previous sample and a_3 is multiplied by the sample before that. The values of the *filter taps* determine whether the filter is high pass or low pass and what the cut-off frequency is. The *order* of the filter is given by the number of delays that are used, so the example shown in Figure 11.2 is a third order filter. The order of the filter determines the steepness of the filter roll-off curve; at one extreme a first order filter gives a very gentle roll-off, while at the other an eighth order is used for demanding anti-aliasing applications.

This filter is an example of a *finite impulse response* (FIR) filter, because it uses a finite number of input values to calculate the output value. Finding the required values for the filter taps is a complex process, but many design packages exist to simplify this process. As an



- x(n) is the input signal (the signal to be filtered)
- y(n) is the filtered signal
- a_1-a_4 are multiplication constants (filter coefficients or filter "taps")
- 1/z refers to a one sample delay

Figure 11.2:
Third order FIR filter

example, if the filter taps in Figure 11.2 all have a value of 0.25, then this implements a simple mean average filter (a crude LPF) which continuously averages the last four consecutive data values.

It can be seen that the filter has a number of multiply and addition processes, which can be grouped into a single MAC operation. A DSP chip has a special area of hardware for processing MAC operations. This is specifically designed so that MAC commands can be processed in a single clock cycle, i.e. much faster than on conventional processors. It should be noted that although the mbed is not a dedicated DSP processor, it is still powerful enough to perform many DSP operations.

11.3 An mbed DSP Example

11.3.1 Input and Output of Digital Data

We can develop an mbed program which reads signal data in via the analog-to-digital converter (ADC), processes the data digitally and then outputs the signal via the digital-to-analog converter (DAC). Since DSP algorithms need to sample and process data at regular fixed intervals, we will use the mbed Timer interrupts to enable a scheduled program. We will now write a program that reads audio data and implements low- and high-pass digital filters. In these examples, the data will be output from the mbed's DAC and through some simple electronic circuitry. The resulting audio output can then be routed to a loudspeaker amplifier (such as a set of portable PC speakers) so that you can listen to the processed signal.

First, we need a signal source to input to the mbed. A simple way to create a signal source is to use a host PC's audio output while playing an audio file of the desired signal data. A number of audio packages, such as Steinberg Wavelab, can be used to create wave audio files (with a .wav file extension); here, we will use three audio files as follows:

- 200hz.wav – an audio file of a 200 Hz sine wave
- 1000hz.wav – an audio file of a 1000 Hz sine wave
- 200hz1000hz.wav – an audio file with the 200 Hz and 1000 Hz audio mixed.

Each audio signal should be mono and around 60 seconds in duration. These files are available for download from the book website.

The audio files can be played directly from the host PC into a set of headphones, and the different sine wave signals can be heard. Now we can connect the audio signal to the mbed by connecting the host PC's audio cable to an mbed analog input pin. The signal can also be viewed on an oscilloscope. You will see that the signal oscillates positive and negative about 0 V. This is not much use for the mbed, as it can only read analog data between 0 and 3.3 V, so all negative data will be interpreted as 0 V.

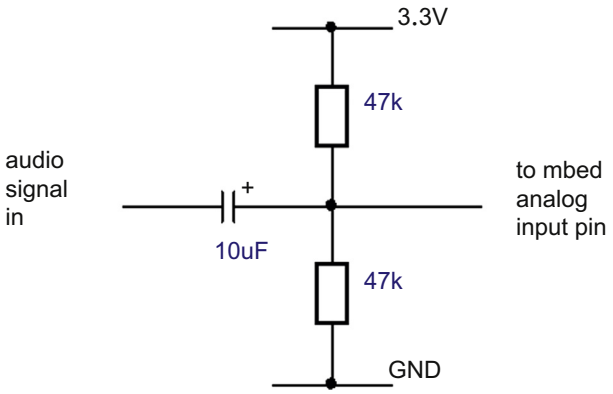


Figure 11.3:
Input circuit with coupling capacitor and bias resistors

Because the mbed can only accept positive voltage inputs, it is necessary to add a small coupling and biasing circuit to offset the signal to a midpoint of approximately 1.65 V. The offset here is often referred to as a direct current (DC) offset. The circuit shown in Figure 11.3 effectively couples the host PC's audio output to the mbed. Create a new project and enter the code of Program Example 11.1.

```
/* Program Example 11.1 DSP input and Output
*/
#include "mbed.h"
//mbed objects
AnalogIn Ain(p15);
AnalogOut Aout(p18);
Ticker s20khz_tick;

//function prototypes
void s20khz_task(void);
//variables and data
float data_in, data_out;

//main program start here
int main() {
  s20khz_tick.attach_us(&s20khz_task,50); // attach task to 50us tick
}

// function 20khz_task
void s20khz_task(void){
  data_in=Ain;
  data_out=data_in;
  Aout=data_out;
}
```

Program Example 11.1 DSP input and output

Program Example 11.1 first defines analog input and output objects (**data_in** and **data_out**) and a single Ticker object called **s20khz_tick**. There is also a function called **s20khz_task( )**.

The **main( )** function simply assigns the 20 kHz Ticker to the 20 kHz task, with a Ticker interval of 50 µs (which sets up the 20 kHz rate). The input is now sampled and processed at this regular rate, within **s20khz_task( )**.

Exercise 11.1

Compile Program Example 11.1 and use a two-channel oscilloscope to check that the analog input signal and the DAC output signal are similar. Use the oscilloscope to see how accurate the DAC output is with respect to the analog input signal for all three audio files. Consider amplitude, phase and the waveform profile.

You will also need to implement the input circuit shown in Figure 11.3.

11.3.2 Signal Reconstruction

If you look closely at the audio signals, particularly the 1000 Hz signal or the mixed signal, you will see that the DAC output has discrete steps. This is more obvious in the high-frequency signal as it is closer to the sampling frequency chosen, as shown in Figure 11.4a.

With many audio DSP systems, the analog output from the DAC is converted to a reconstructed signal by implementing an analog *reconstruction filter*, which removes all steps from the signal, leaving a smooth output. In audio applications, a reconstruction filter is usually designed to be an LPF with a cut-off frequency at around 20 kHz, because the human hearing range does not exceed 20 kHz. The reconstruction filter shown in Figure 11.5 can be implemented with the current project (which gives the complete DSP input/output circuit as shown in Figure 11.6). Note that after the LPF, a *decoupling capacitor* is also added to remove the 1.65 V DC offset from the signal. Once the DC offset has been removed, the signal can be routed to a loudspeaker amplifier to monitor the processed DAC output.

The mathematical theory associated with digital sampling and reconstruction is complex and beyond the scope of this book. As discussed in Chapter 4, in audio sampling systems it is often necessary to add an anti-aliasing filter prior to the analog-to-digital conversion. For simplicity this extra filter has not been implemented here. For those interested, the theory of sampling, aliasing and reconstruction is described well and in detail by a number of authors including Marven and Ewers (Reference 11.1) and Proakis and Manolakis (Reference 11.2).

11.3.3 Adding a Digital Low-Pass Filter

We will add a digital low-pass filter routine to filter out the 1000 Hz frequency component. This can be assigned to a switch input so that the filter is implemented in real time when a push-button is pressed.

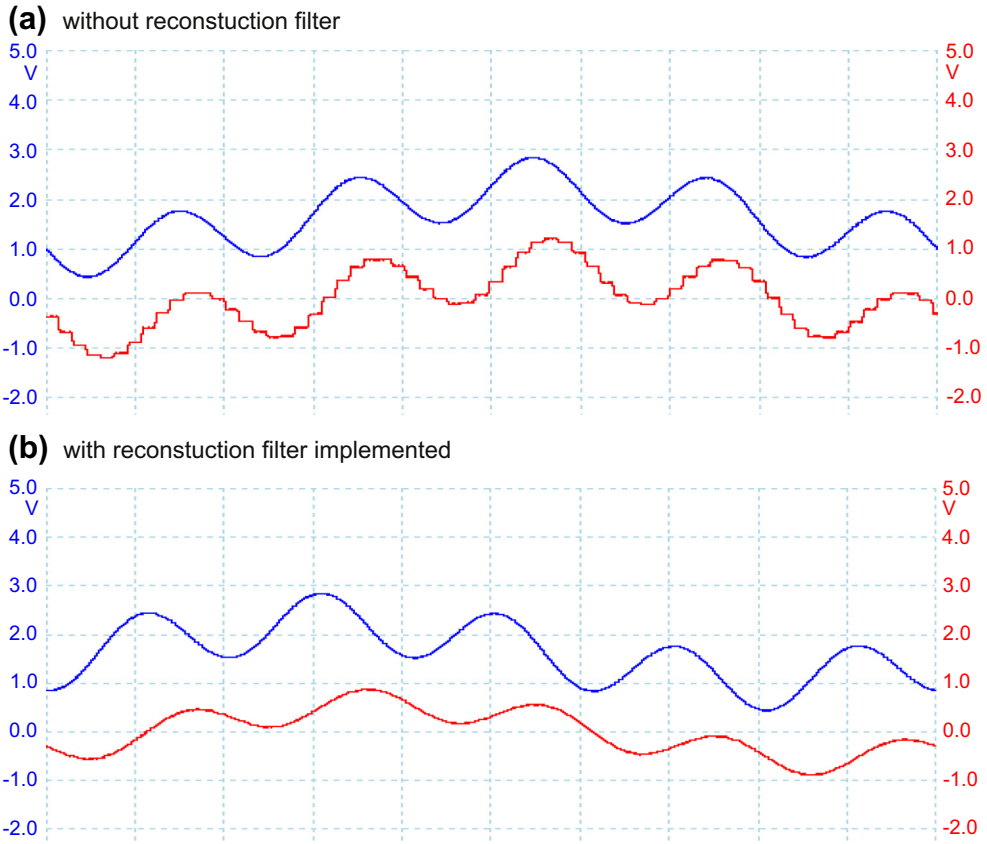


Figure 11.4:
Signal output

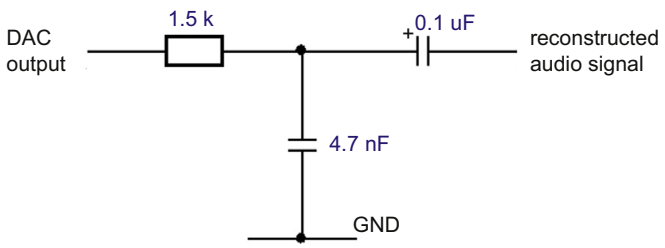


Figure 11.5:
Analog reconstruction filter and decoupling capacitor

This example will use a third order *infinite impulse response* (IIR) filter, as shown in Figure 11.7. The IIR filter uses recursive output data (i.e. data fed back from the output), as well as input data, to calculate the filtered output.

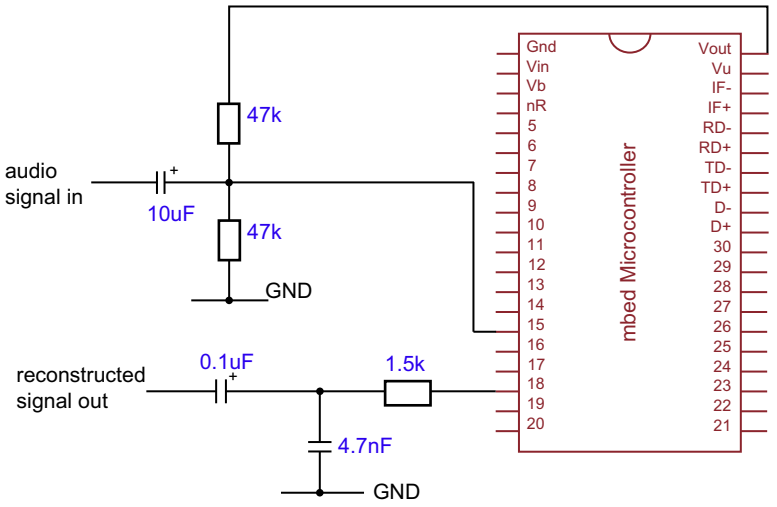


Figure 11.6:
DSP input/output circuit

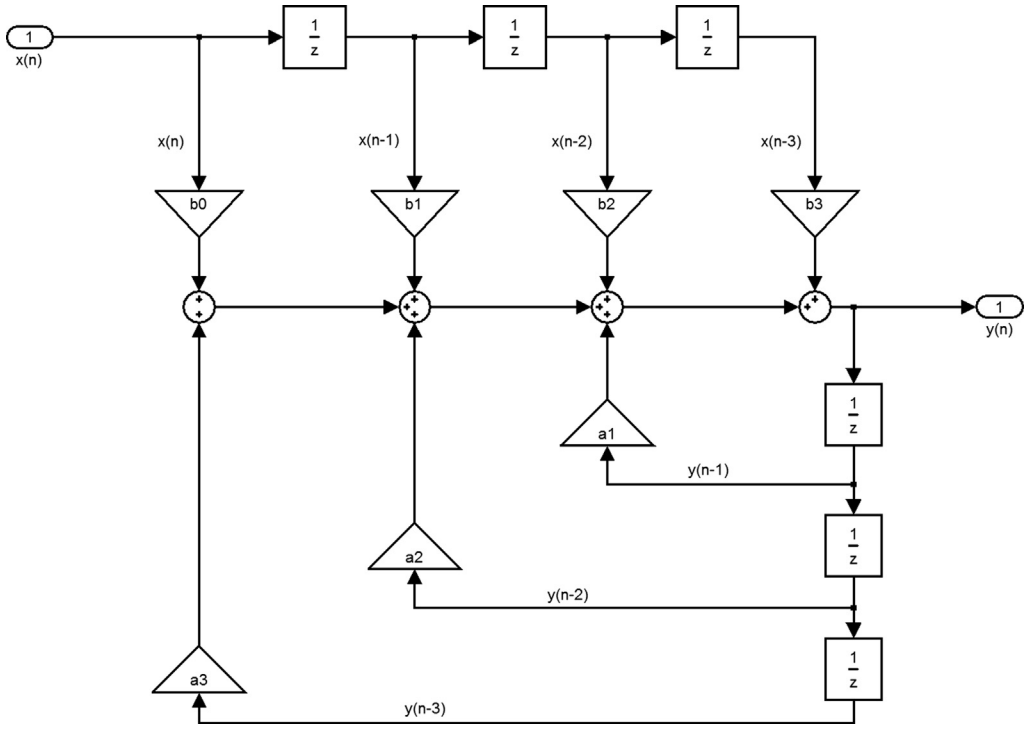


Figure 11.7:
Third order digital IIR filter

This filter results in the following equation for calculating the filtered value given the current input, the previous three input values and the previous three output values:

$$y(n) = b_0x(n) + b_1x(n-1) + b_2x(n-2) + b_3x(n-3) \\ +a_1y(n-1) + a_2y(n-2) + a_3y(n-3) \quad 11.1$$

where $x(n)$ is the current data input value; $x(n-1)$ is the previous data input value, $x(n-2)$ is the data value before that, and so on; $y(n)$ is the calculated current output value; $y(n-1)$ is the previous data output value $y(n-2)$ is the data value before that, and so on; and a_{0-3} and b_{0-3} are coefficients (filter taps) that define the filter's performance.

We can implement this equation to achieve filtered data from the input data. The challenging task is determining the values required for the filter coefficients to give the desired filter performance. Filter coefficients can, however, be calculated by a number of software packages, such as the Matlab Filter Design and Analysis Tool (Reference 11.3), and those provided in Reference 11.4.

The LPF designed for a 600 Hz cut-off frequency (third order with 20 kHz sampling frequency) can be implemented as a C function as shown in Program Example 11.2. We chose a 600 Hz LPF because the cut-off is exactly half way between the two signal frequencies.

```
/*Program Example 11.2 Low-pass filter function
*/
float LPF(float LPF_in){

  float a[4]={1,2.6235518066,-2.3146825811,0.6855359773};
  float b[4]={0.0006993496,0.0020980489,0.0020980489,0.0006993496};
  static float LPF_out;
  static float x[4], y[4];
  x[3] = x[2]; x[2] = x[1]; x[1] = x[0];     // move x values by one sample
  y[3] = y[2]; y[2] = y[1]; y[1] = y[0];     // move y values by one sample
  x[0] = LPF_in;                             // new value for x[0]
  y[0] =  (b[0]*x[0]) + (b[1]*x[1]) + (b[2]*x[2]) + (b[3]*x[3])
         + (a[1]*y[1]) + (a[2]*y[2]) + (a[3]*y[3]);
  LPF_out = y[0];
  return LPF_out;                            // output filtered value
}
```

Program Example 11.2 Low-pass filter function

Here, we can see the calculated filter values for the **a** and **b** coefficients. These coefficients are deduced by the online calculator provided by Reference 11.4. Note also that we have used the **static** variable type for the internally calculated data values. The static definition ensures that data calculated are held even after the function is complete, so the recursive data values for previous samples are held within the function and not lost during program execution.

Exercise 11.2

Create a new mbed program based on Program Example 11.1, only this time add the function for the LPF seen in Program Example 11.2. Now in the 20 kHz task process the input data by feeding it through the LPF function, for example:

```
data_out=LPF(data_in);
```

Compile and run the code to check that high-frequency components are filtered from the mbed's analog output.

Your system should use the mbed with the input and output circuitry shown in Figure 11.6.

11.3.4 Adding a Push-Button Activation

We can now assign a conditional statement to a digital input, allowing the filter to be switched in and out in real-time. The following conditional statement implemented in the 20 kHz task will allow real-time activation of the digital filter:

```
data_in=Ain-0.5;
  if (LPFswitch==1){
     data_out=LPF(data_in);
  }
  else {
     data_out=data_in;
  }
  Aout=data_out+0.5;
```

You will notice that before performing the calculation, the mean value of the signal is subtracted, in the line:

```
data_in=Ain-0.5;
```

This is to *normalize* the signal to an average value of zero, so the signal oscillates positive and negative and allows the filter algorithm to perform DSP with no DC offset in the data. As the DAC anticipates floating point data in the range 0.0−1.0, we must also add the mean offset back to the data before we output, in the line:

```
Aout=data_out+0.5;
```

Exercise 11.3

Implement the real-time push-button activation in your LPF program. You can now use the oscilloscope or headphones to listen to the signal which includes both the 200 Hz and 1000 Hz signal played simultaneously. When the switch is pressed, the high-frequency

component should be removed, leaving just the low-frequency component audible, or visible on the oscilloscope.

■

11.3.5 Digital High-Pass Filter

The implementation of an HPF function is identical to the low-pass function, but with different filter coefficient values. The filter coefficients for a 600 Hz HPF (third order with a 20 kHz sample frequency) are as follows:

```
float a[4]={1,2.6235518066,-2.3146825811,0.6855359773 };
float b[4]={0.8279712953,-2.4839138860,2.4839138860,-0.8279712953};
```

■ Exercise 11.4

Add a second switch to the circuit and a filter function to the program, to act as an HPF. This switch will enable filtering of the low-frequency component and leave the high-frequency signal. Implement the second function with a second conditional statement in the 20 kHz task, so that either the LPF or HPF can be activated in real time. You should now be able to listen to the simultaneous 200 Hz and 1000 Hz audio signal and filter either the low frequency or the high frequency, or both, dependent on which switch is pressed.

■

11.4 Delay/Echo Effect

Many audio effects use DSP systems for manipulating and enhancing audio. These can be useful for live audio processing (e.g. guitar effects) or for post-production. Audio production DSP effects include artificial reverb, pitch correction, dynamic range manipulation and many other techniques to enhance captured audio.

A feedback delay can be used to make an echo effect, which sees a single sound repeated a number of times. Each time the signal is repeated it is attenuated until it eventually decays away. Therefore, the speed of repetition and the amount of feedback attenuation can be manipulated. This effect is used commonly as a guitar effect, and for vocal processing and enhancement. Figure 11.8 shows a block diagram design for a simple delay effect unit.

To implement the system design shown in Figure 11.8, historical sample data must be stored so that it can be mixed back in with immediate data. Therefore, sampled digital data needs to be copied into a buffer (a large array) so that the feedback data is always available. The feedback gain determines how much buffer data is mixed with the sampled data.

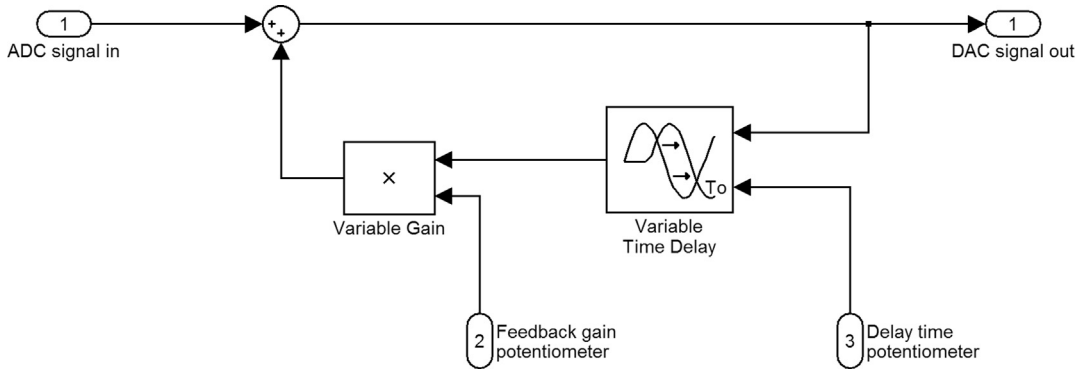


Figure 11.8:
Delay effect block diagram

For each sample coming in, an earlier value is mixed in also, but the length of time between the immediate and historical value can be varied by the delay time potentiometer. Effectively, this changes the size of the data buffer by resetting an array counter based on the value of the delay time input. If the delay time input is large, a large number of array data will be fed back before resetting, resulting in a long delay time. For smaller delay values the array counter will reset sooner, giving a more rapid delay effect.

```
/* Program Example 11.3 Delay / Echo Effect
*/
#include "mbed.h"
AnalogIn Ain(p15);                          //object definitions
AnalogOut Aout(p18);
AnalogIn delay_pot(p16);
AnalogIn feedback_pot(p17);
Ticker s20khz_tick;
void s20khz_task(void);                     // function prototypes
#define MAX_BUFFER 14000                    // max data samples
signed short data_in;                       // signed allows positive and negative
unsigned short data_out;                    // unsigned just allows positive values
float delay=0;
float feedback=0;
signed short buffer[MAX_BUFFER]={0};        // define buffer and set values to 0
int i=0;

//main program start here
int main() {
  s20khz_tick.attach_us(&s20khz_task,50);
}
// function 20khz_task
void s20khz_task(void){
  data_in=Ain.read_u16()-0x7FFF;             // read data and normalize
  buffer[i]=data_in+(buffer[i]*feedback);    // add data to buffer data
  data_out=buffer[i]+0x7FFF;                 // output buffer data value
```

```
  Aout.write_u16(data_out);        // write output
  if (i>(delay)){                  // if delay loop has completed
    i=0;                           // reset counter
    delay=delay_pot*MAX_BUFFER;    // calculate new delay buffer size
    feedback=(1-feedback_pot)*0.9; // calculate feedback gain value
  }else{
    i=i+1;                         // otherwise increment delay counter
  }
}
```

Program Example 11.3 Delay/echo effect

Notice that the **data_in** and **data_out** values are defined as follows:

```
signed short data_in;       // signed allows positive and negative
unsigned short data_out;    // unsigned just allows positive values
```

C code feature

The **short** data type (see Table B.4) ensures that the data is constrained to 16-bit values; this can be specified as **signed** (i.e. to use the range −32 768 to 32 767 decimal) or **unsigned** (to use the range 0 to 65535). We need the computation to take place with signed numbers. When outputting to the mbed's DAC, however, this needs an offset adding owing to the fact that DAC is configured to work with unsigned data.

The initial mbed hardware setup shown in Figure 11.6 can be used here. This demonstrates the advantage of a single hardware design that can operate multiple software features in a digital system. We will need to add potentiometers to mbed analog inputs in order to control the feedback speed and gain in real time, as shown in Figure 11.8. Program Example 11.3 defines the delay and feedback potentiometers being connected to mbed analog inputs on pins 16 and 17, respectively.

Exercise 11.5

Implement a new project with the code shown in Program Example 11.3. Initially, you will need to use a test signal which gives a short pulse followed by a period of inactivity to evaluate the echo effect performance. You should see a similar echo response to that shown in Figure 11.9. Verify that the delay and feedback gain potentiometers alter the output signal as expected. For testing purposes, a pulse signal called **pulse.wav** can be downloaded from the book website.

■

Figure 11.9 shows input and output waveforms for the delay/echo effect. It can be seen that for a single input pulse, the output is a combination of the pulse plus repeated echoes of that pulse with slowly diminishing amplitude. The rate of echo and the rate of attenuation are altered by adjusting the potentiometers as described. This project can be developed as a guitar

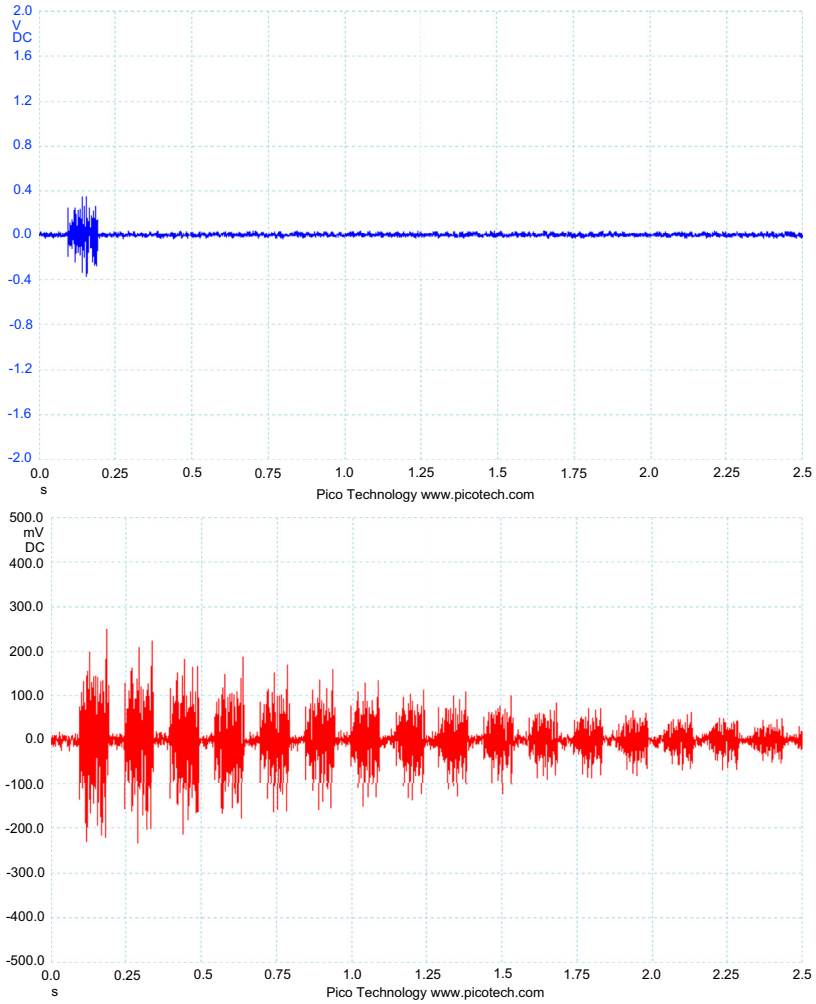


Figure 11.9:
Input signal (top) and output signal (bottom) for the digital echo effect system

effect unit by adding extra signal conditioning, variable amplification stages and enhanced output conditioning. Some good examples are given in Reference 11.5.

11.5 Working with Wave Audio Files

This chapter, so far, has analyzed a DSP application which captures data, manipulates the data and then outputs it. However, a number of signal processing applications rely on data that has previously been captured and stored in data memory. Continuing with examples relating to

digital audio, this can be explored by analyzing wave audio files and evaluating the requirements in order to output a continuous and controlled stream of audio data.

11.5.1 The Wave Information Header

There are many types of audio data file, of which the wave (.wav) type is one of the most widely used. Wave files contain a number of details about the audio data followed by the audio data itself. Wave files contain uncompressed (i.e. raw) data, mostly in a format known as *linear pulse code modulation* (PCM). Since PCM specifically refers to the coding of amplitude signal data at a fixed sample rate, each sample value is given to a specified resolution (often 16-bit) on a linear scale. Time data for each sample is not recorded because the sample rate is known, so only amplitude data is stored. The wave header information details the actual resolution and sample frequency of the audio, so by reading this header it is possible to accurately decode and process the contained audio data. The full wave header file description is shown in Table 11.1.

It is possible to identify much of the wave header information simply by opening a .wav file with a text editor application, as shown in Figure 11.10. Here we see the ASCII characters for

Table 11.1: Wave file information header structure

Data name	Offset (bytes)	Size (bytes)	Details
ChunkID	0	4	The characters 'RIFF' in ASCII
ChunkSize	4	4	Details the size of the file from byte 8 onwards
Format	8	4	The characters 'WAVE' in ASCII
Subchunk1ID	12	4	The characters 'fmt' in ASCII
Subchunk1Size	16	4	16 for PCM
AudioFormat	20	2	PCM = 1. Any other value indicates data compressed format
NumChannels	22	2	Mono = 1; stereo = 2
SampleRate	24	4	Sample rate of the audio data in Hz
ByteRate	28	4	ByteRate = SampleRate * NumChannels * BitsPerSample/8
BlockAlign	32	2	BlockAlign = NumChannels * BitsPerSample/8. The number of bytes per sample block
BitsPerSample	34	2	Resolution of audio data
SubChunk2ID	36	4	The characters 'data' in ASCII
Subchunk2Size	40	4	Subchunk2Size = Number of samples * BlockAlign. The total size of the audio data in bytes
Data	44	–	The actual data of size given by Subchunk2Size

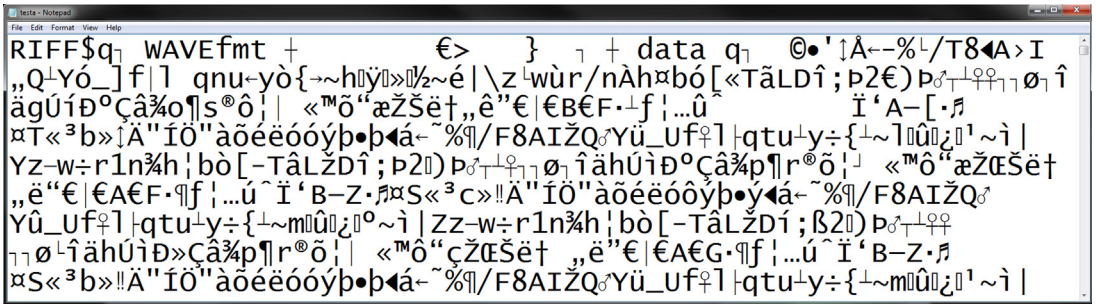


Figure 11.10:
Wave file opened in text editor

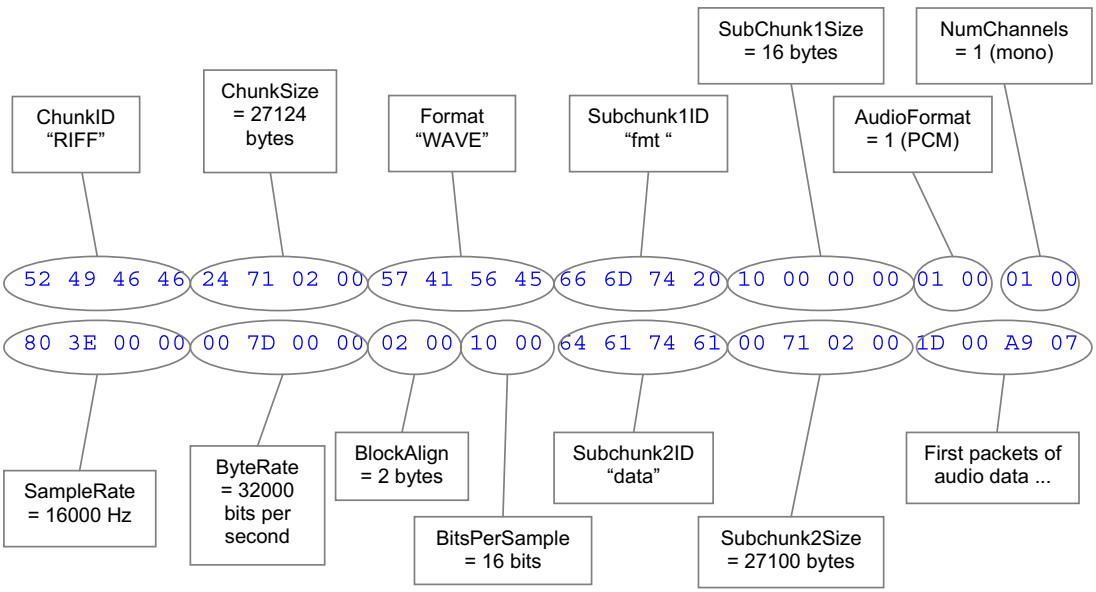


Figure 11.11:
Structure of the wave file header data

ChunkID (‘RIFF’), Format (‘WAVE’), Subchunk1ID (‘fmt’) and Subchunk2ID (‘data’). These are followed by the ASCII data which makes up the raw audio.

Looking more closely at the header information for this file in hexadecimal format, the specific values of the header information can be identified, as shown in Figure 11.11. Note also that for each value made up of multiple bytes, the least significant byte is always given first and the most significant byte last. For example, the four bytes denoting the sample rate is given as 0x80, 0x3E, 0x00 and 0x00, which gives a 32-bit value of 0x00003E80 = 16 000 decimal.

11.5.2 Reading the Wave File Header with the mbed

In order to accurately read and reproduce wave data via a DAC, we need to interpret the information given in the header data, and configure the read and playback software features with the correct audio format (from **AudioFormat**), number of channels (**NumChannels**), sample rate (**SampleRate**) and the data resolution (**BitsPerSample**).

To implement wave file manipulation on the mbed, the wave data file should be stored on a Secure Digital (SD) memory card and the program should be configured with the correct SD reader libraries, as discussed in Chapter 10. The SD card is required predominantly because wave audio files are generally quite large, much too large to be stored in the mbed's internal memory, but also because the file access through the serial peripheral interface (SPI)/SD card interface is very fast.

Program Example 11.4 reads a wave audio file called **test.wav**, utilizes the **fseek( )** function used to move the file pointer to the relevant position, and then reads the header data and displays this to a host terminal. Note also that to read .wav files on the mbed, you should use the '8.3 filename' convention (sometimes referred to as *short file name* or SFN) for audio files. The SFN convention stipulates that filenames should not be longer than eight characters with a three-character extension.

```
/* Program Example 11.4 Wave file header reader
*/
#include "mbed.h"
#include "SDFileSystem.h"
SDFileSystem sd(p5, p6, p7, p8, "sd");
Serial pc(USBTX,USBRX);            // set up terminal link
char c1, c2, c3, c4;               // chars for reading data in
int AudioFormat, NumChannels, SampleRate, BitsPerSample ;
int main() {
    pc.printf("\n\rWave file header reader\n\r");
      FILE *fp = fopen("/sd/sinewave.wav", "rb");
      fseek(fp, 20, SEEK_SET);         // set pointer to byte 20
      fread(&AudioFormat, 2, 1, fp);   // check file is PCM
      if (AudioFormat==0x01) {
      pc.printf("Wav file is PCM data\n\r");
      }
    else {
      pc.printf("Wav file is not PCM data\n\r");
    }
    fread(&NumChannels, 2, 1, fp);   // find number of channels
    pc.printf("Number of channels: %d\n\r",NumChannels);
      fread(&SampleRate, 4, 1, fp);    // find sample rate
      pc.printf("Sample rate: %d\n\r",SampleRate);
      fread(&BitsPerSample, 2, 1, fp); // find resolution
      pc.printf("Bits per sample: %d\n\r",BitsPerSample);
      fclose(fp);
}
```

Program Example 11.4 Wave header reader

Try reading the header of a number of different wave files and verify that the correct information is always read to a host PC. A wave audio file can be created in many simple audio packages, such as Steinberg Wavelab, or can be extracted from a standard music compact disc with music player software such as iTunes or Windows Media Player. When reading different wave files remember to update the **fopen( )** call to use the correct filename.

C code feature

In Program Example 11.4 the **fread( )** function is used to read a number of data bytes in a single command. Examples of **fread( )** show that the memory address for the data destination is specified, along with the size of each data packet (in bytes) and the total number of data packets to be read. The file pointer is also given. For example, the command

```
fread(&SampleRate, 4, 1, fp);        // find sample rate
```

reads a single 4-byte data value from the data file pointed to by **fp** and places the read data at the internal memory address location of variable **SampleRate**.

Exercise 11.6

Extend Program Example 11.4 to display **ByteRate** and **Subchunk2Size**, which indicates the size of the raw audio data within the file.

11.5.3 Reading and Outputting Mono Wave Data

Having accessed the wave file and gathered important information about its characteristics, it is possible to read the raw audio data and output that from the mbed's DAC. To do this, we need to understand the format of the audio data. Initially, we will use an oscilloscope to verify that the data outputs correctly, but it is possible to output the audio data to a loudspeaker amplifier. For this example, we will use a 16-bit mono wave file of a pure sine wave of 200 Hz (Figure 11.12). The same sine wave as utilized in Section 11.3 can be used here.

The audio data in a 16-bit mono .wav file is arranged similarly to the other data seen in the header. The data starts at byte 44 and each 16-bit data value is read as two bytes with the least significant byte first. Each 16-bit sample can be outputted directly on pin 18 at the rate defined by **SampleRate**. It is very important to take good care of data input and output timing when working with audio, as any timing inaccuracies in playback can be heard as clicks or stutters. This can be a challenge because interrupts and timing overheads sometimes make it difficult for the audio to be streamed directly from the SD card at a constant rate. Therefore, a buffered system is used to enable accurate timing control.

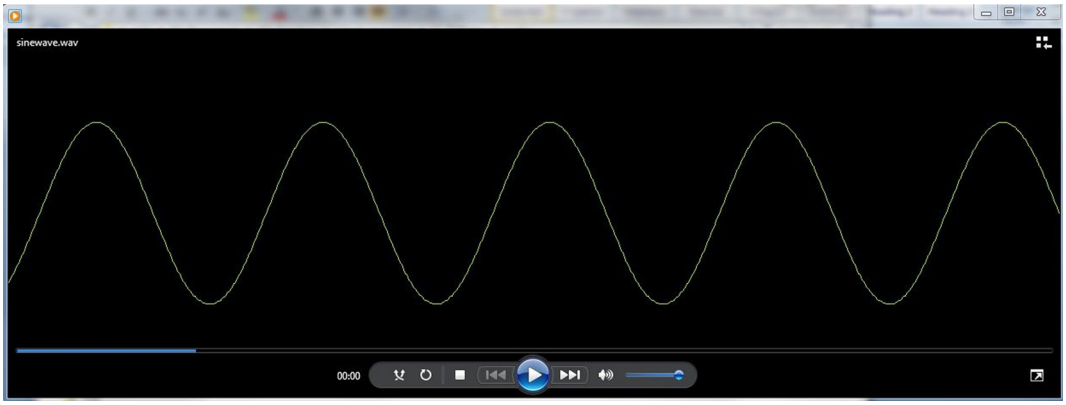


Figure 11.12:
A 200 Hz sine wave .wav file played in Windows Media Player

A good method to ensure accurate timing control when working with audio data is to use a *circular buffer*, as shown in Figure 11.13. The buffer allows data to be read in from the data file and read out from it to a DAC. If the buffer can hold a number of audio data samples, then, as long as the DAC output timer is accurate, it does not matter much if the time for data being read and processed from the SD card is somewhat variable. When the circular buffer write pointer reaches the last array element, it wraps around so that the next data is read into the first memory element of the buffer. There is a separate buffer read pointer for outputting data to the DAC, and this lags behind the write buffer.

It can be seen that two important conditions must be met for the circular buffer method to work; first, the data written to the buffer must be written at an equal or faster rate than data is read from the buffer (so that the write pointer stays ahead of the read pointer); and secondly,

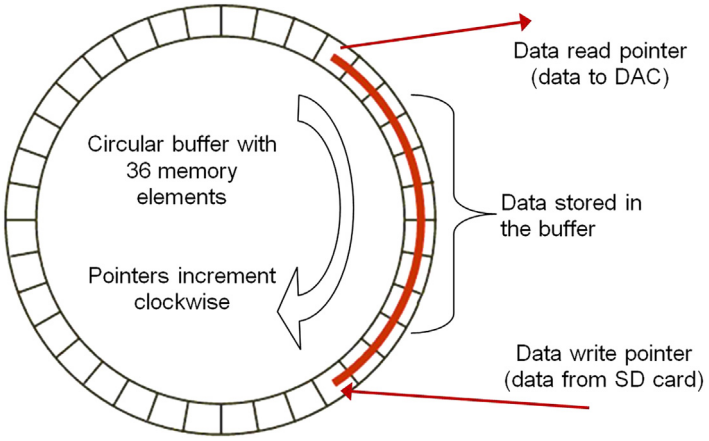


Figure 11.13:
Circular buffer example

the buffer must never completely fill up, or else the read pointer will be overtaken by the write pointer and data will be lost. It is therefore important to ensure that the buffer size is sufficiently large, or to implement control code to safeguard against data wrapping.

Program Example 11.5 reads a 16-bit mono audio file (called in this example **testa.wav**, which can be downloaded from the book website) and outputs at a fixed sample rate which is acquired from the wave file header. As discussed above, the circular buffer is used to iron out the timing inconsistencies found with reading from the wave data file.

```
/* Program Example 11.5: 16-bit mono wave player
*/
#include "mbed.h"
#include "SDFileSystem.h"
#define BUFFERSIZE 4096                          // number of data in circular buffer
SDFileSystem sd(p5, p6, p7, p8, "sd");
AnalogOut DACout(p18);
Ticker SampleTicker;
int SampleRate;
float SamplePeriod;                              // sample period in microseconds
int CircularBuffer[BUFFERSIZE];                  // circular buffer array
int ReadPointer=0;
int WritePointer=0;
bool EndOfFileFlag=0;
void DACFunction(void);                          // function prototype
int main() {
  FILE *fp = fopen("/sd/testa.wav", "rb");       // open wave file
  fseek(fp, 24, SEEK_SET);                       // move to byte 24
  fread(&SampleRate, 4, 1, fp);                  // get sample frequency
  SamplePeriod=(float)1/SampleRate;              // calculate sample period as float
  SampleTicker.attach(&DACFunction,SamplePeriod); // start output tick
  while (!feof(fp)) {        // loop until end of file is encountered
    fread(&CircularBuffer[WritePointer], 2, 1, fp);
    WritePointer=WritePointer+1;                 // increment Write Pointer
    if (WritePointer>=BUFFERSIZE) {              // if end of circular buffer
       WritePointer=0;                           // go back to start of buffer
    }
  }
  EndOfFileFlag=1;
  fclose(fp);
}
// DAC function called at rate SamplePeriod
void DACFunction(void) {
  if ((EndOfFileFlag==0)|(ReadPointer>0)) {          // output while data available
    DACout.write_u16(CircularBuffer[ReadPointer]);  // output to DAC
    ReadPointer=ReadPointer+1;                       // increment pointer
    if (ReadPointer>=BUFFERSIZE) {
      ReadPointer=0;                                 // reset pointer if necessary
    }
  }
}
```

Program Example 11.5 Wave file player with utilizing a circular buffer

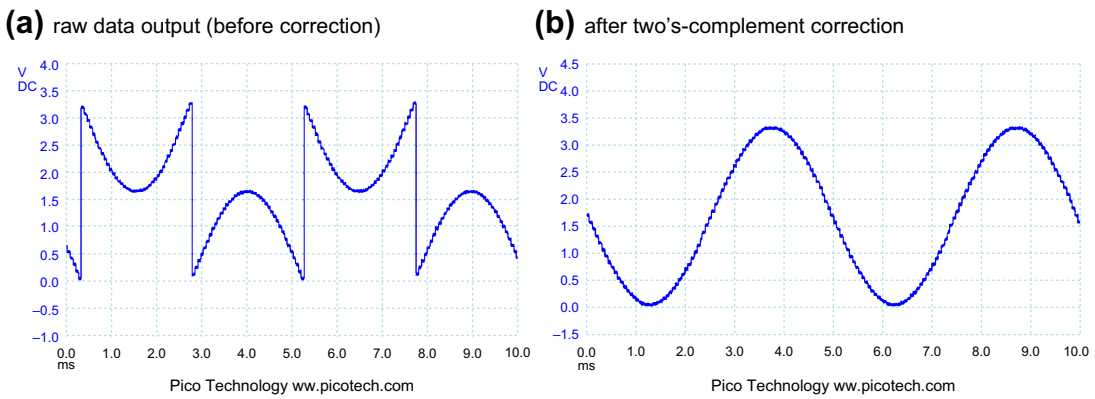


Figure 11.14:
Wave file sine wave

When Program Example 11.5 is implemented with a pure mono sine wave audio file, initially the results will not be correct. Figure 11.14a shows the actual oscilloscope trace of a sine wave read using this program example. Clearly, this is not an accurate reproduction of the sine wave data. The error occurs because the wave data is coded in 16-bit two's-complement form, which means that positive data occupies data values 0 to 0x7FFF and values 0x8000 to 0xFFFF represent the negative data. Two's complement arithmetic is introduced in Appendix A. A simple adjustment of the data is therefore required in order to output the correct waveform as shown in Figure 11.14b.

Exercise 11.7

Modify Program Example 11.5 to include two's complement correction and ensure that, when using an audio file of a mono sine wave, the output data observed on an oscilloscope is that of an accurate sine wave. Appendix A should help you to work out how to do this.

11.6 Summary on DSP

As shown in this chapter, DSP techniques require knowledge of a number of mathematical and data handling aspects. In particular, attention to detail of timing and data validity is required to ensure that no data overflow errors or timing inconsistencies occur. We have also looked at a new type of data file which holds signal data to be output at a specific and

controlled rate. The ability to apply simple DSP techniques, whether on a dedicated DSP chip or on a general-purpose processor like the mbed, hugely expands our capabilities as embedded system designers.

11.7 Mini-Project: Stereo Wave Player

This project can be done in basic form, or with several extensions.

11.7.1 Basic Stereo Wave Player

A stereo wave player program can be developed by adding additional elements to Program Example 11.5. You will need to update the program to first analyze whether the wave data is stereo or not, and then implement a conditional feature for the case where stereo data is stored.

Stereo wave data is stored in consecutive samples for left and right stereo playback as shown in Figure 11.15. If using just the standard mbed analog output you will need to average the left and right channel data to mono before outputting to the DAC. Full stereo ADC and DAC chips, such as the Texas Instruments TLV320AIC23b, can be purchased and interfaced with the mbed through its serial ports.

11.7.2 Stereo Wave Player with PC Interface

Add a user interface that will display all the wave filenames stored on an SD card to a host PC terminal. The user can then select a file to be played by selecting a number that represents the chosen data file.

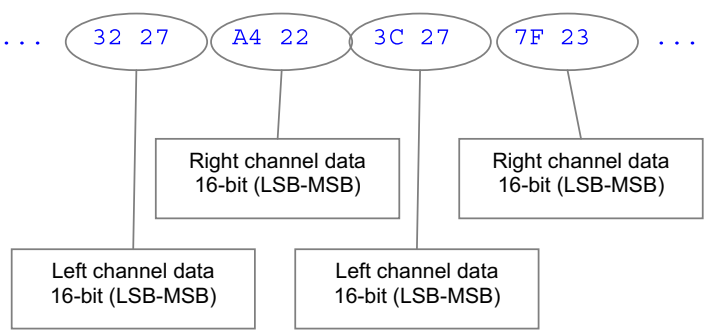


Figure 11.15:
Interleaved left and right channel stereo data

11.7.3 Portable Stereo Wave Player with Mobile Phone Display Interface

Experiment with using a liquid crystal display (LCD) mobile phone display and digital push-buttons to develop your own portable audio player.

Chapter Review

- DSP systems and algorithms are used for managing and manipulating streams of data and therefore require high precision and timing accuracy.
- A digital filtering algorithm can be used to remove unwanted frequencies from a data stream. Similar mathematical algorithms can be used for signal analysis, audio/video manipulation and data compression for communications.
- A DSP system communicates with the external world through analog-to-digital converters and digital-to-analog converters, so the analog elements of the system also require careful design.
- DSP systems usually rely on regularly timed data samples, so the mbed Timer and Ticker interfaces are useful for programming regular and real-time processing.
- Wave audio files hold high-resolution audio data which can be read from an SD card and output through the mbed DAC.
- Data management and effective buffering are required to ensure that timing and data overflow issues are avoided.

Quiz

1. What does the term MAC refer to and why is this important for DSP systems?
2. What differentiates a DSP microprocessor from a microcontroller?
3. What is the difference between an FIR and an IIR digital filter?
4. What are digital filter coefficients and how can they be acquired for a specific digital filter design?
5. Explain the role of analog biasing and anti-aliasing when performing an analog-to-digital conversion.
6. What is a reconstruction filter and where would this be found in a DSP system?
7. What is a circular buffer and why might it be used in DSP systems?
8. What are the potential effects of poor timing control in an audio DSP system?
9. A wave audio file has a 16-bit mono data value given by two consecutive bytes. What will be the correct corresponding voltage output, if this is output through the mbed's DAC, for the following data?
   (a) 0x35 0x04
   (b) 0xFF 0x5F
   (c) 0x00 0xE4

10. Draw a block diagram design of a DSP process for mixing two 16-bit data streams. If the data output is also to be 16 bit, what consequences will this process have on the output data resolution and accuracy?

References

11.1. Marven, C. and Ewers, G. (1996). A Simple Approach to Digital Signal Processing. Wiley Blackwell.
11.2. Proakis, J. G. and Manolakis, D. K. (1992). Digital Signal Processing: Principles, Algorithms and Applications. Prentice Hall.
11.3. The Mathworks (2010). FDATool – open filter design and analysis tool. http://www.mathworks.com/help/toolbox/signal/fdatool.html
11.4. Fisher, T. (2010). Interactive Digital Filter Design. Online Calculator. http://www-users.cs.york.ac.uk/~fisher/mkfilter/
11.5. Sergeev, I. (2010). Audio Echo Effect. http://dev.frozeneskimo.com/embedded_projects/audio_echo_effect