

Working With Digital Audio

13.1 An Introduction to Digital Audio

Digital audio systems are nowadays all around us, built into computers, mobile phones, toys, and car stereos, as well as in professional music systems used in music recording studios and performance venues. Digital audio brings many benefits above the traditional methods of recording and playback from analog tape or vinyl. In particular, there are advantages of reliability, sound quality, and storage space, as well as significant benefits that can be delivered through portability, online music catalogs and mobile (wireless) data access.

A digital *audio interface* (sometimes called a *soundcard*) incorporates analog-to-digital and digital-to-analog convertors (ADCs and DACs). These allow analog audio data from a microphone to be recorded to a digital data stream, as well as digital data to be converted to an analog signal that can be played out through a loudspeaker. While the audio is represented as digital data, it can also be manipulated through *digital signal processing* (DSP), which relies on mathematical algorithms to somehow enhance or change the audio properties of the data. Mathematical DSP algorithms can be used to change the level of specific bass or treble frequencies through filtering, or to delay and repeat sounds to provide a perceived sense of space or echo. One of the most recent advances in audio DSP is an algorithm called *auto-tune*, which can take a musical performance and modify each frequency so that it sounds perfectly in tune with the pitch or melody of a song. DSP algorithms are also used to reduce the file size of music, such as the MP3 algorithm, which enables large amounts of audio data to be streamed over the internet with only a small perceivable loss in audio quality.

There are many aspects of digital audio focused on ensuring a high-fidelity sound, other aspects focus on expanding the creative toolset for musicians. *Musical Instrument Digital Interface (MIDI)* is a serial message protocol that allows digital musical instruments to communicate with host digital audio systems which can turn those signals into sound. MIDI was first developed in 1982 by the MIDI Manufacturers Association. MIDI data contains a number of parameters, as described in detail in Ref. [1]. MIDI is still a valuable method for enabling communications between electronic music systems, particularly given that in 1999 a new MIDI standard was developed to allow messaging through the more modern USB protocol.

13.2 USB MIDI on the mbed

An example instrument using a MIDI interface is a MIDI piano-style keyboard. The keyboard itself does not make any sound. Instead the MIDI signals communicate with a sound module or computer with MIDI software installed, so that the signals can be turned into music. In a very simple MIDI system the most valuable information is

1. whether a musical note is to be switched on or off,
2. the pitch of the note.

The method for setting up and interfacing the mbed with MIDI sequencing software varies subtly depending on the software used. In most cases, however, the audio sequencer software (such as Ableton Live, Apple Logic, or Steinberg Cubase) automatically recognizes the mbed MIDI interface and allows it to control a software instrument or synthesizer.

A MIDI interface can be created by first importing the **USBDevice** library, importing the **USBMIDI.h** header file, and initializing the interface (in this case named “midi”) as follows:

```
USBMIDI midi;           // initialise MIDI interface
```

(Check Section 6.7 if you need to refresh on the methods for importing libraries and **#include**-ing header files.)

A MIDI message to sound a note is activated by the following command:

```
midi.write(MIDIMessage::NoteOn(note)); // play musical note
```



This command uses the C++ *scope resolution operator* (::) which relates to a number of advanced C++ programming features. It is not our intention to explore advanced C++ programming concepts, so for the purpose of this example it is sufficient simply to accept the described syntax for sending a MIDI message from the mbed.

The value **note** represents notes on a piano keyboard; this is a 7-bit value so there are a possible 128 notes that can be described by MIDI. The value zero represents a very low C note and, as there are 12 notes in the chromatic musical scale, octaves of C occur at multiples of 12. So the MIDI note value 60 represents middle C (also referred to as C₄), which has a fundamental frequency of 261.63 Hz. An excerpt of the full MIDI note table is shown in [Table 13.1](#).

13.2.1 Sending USB MIDI Data From an mbed

Program Example 13.1 sets up an mbed MIDI interface to continuously step through the notes shown in [Table 13.1](#). The Program Example also implements an analog

Table 13.1: MIDI note values and associated musical notes and frequencies.

MIDI note	48	49	50	51	52	53	54	55	56	57	58	59
Musical Note	C3	C#3	D3	D#3	E3	F3	F#3	G3	G#3	A3	A#3	B3
Frequency (Hz)	130.1	138.6	146.8	155.6	164.8	174.6	185.0	196.0	207.7	220.0	233.1	246.9
MIDI note	60	61	62	63	64	65	66	67	68	69	70	71
Musical Note	C4	C#4	D4	D#4	E4	F4	F#4	G4	G#4	A4	A#4	B4
Frequency (Hz)	261.6	277.2	293.7	313.1	329.6	349.2	370.0	392.0	415.3	440.0	466.2	493.9

input, which can be connected to a potentiometer, so that the speed of the steps can be manipulated.

```

/* Program Example 13.1: MIDI messaging with variable scroll speed
*/


#include "mbed.h"
#include "USBMIDI.h"
USBMIDI midi;           // initialise MIDI interface
AnalogIn Ain(p19);      // create analog input

int main() {
    while (1) {
        for(int i=48; i<72; i++) {           // step through notes
            midi.write(MIDIMessage::NoteOn(i)); // note on
            wait(Ain);                         // pause
            midi.write(MIDIMessage::NoteOff(i)); // note off
            wait(2*Ain);                       // pause
        }
    }
}

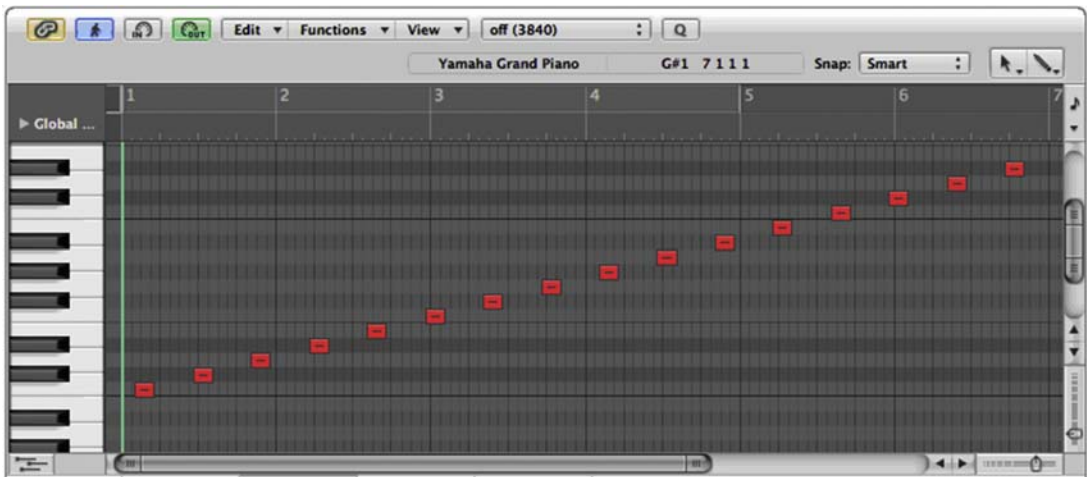
```

Program Example 13.1: MIDI messaging with variable scroll speed

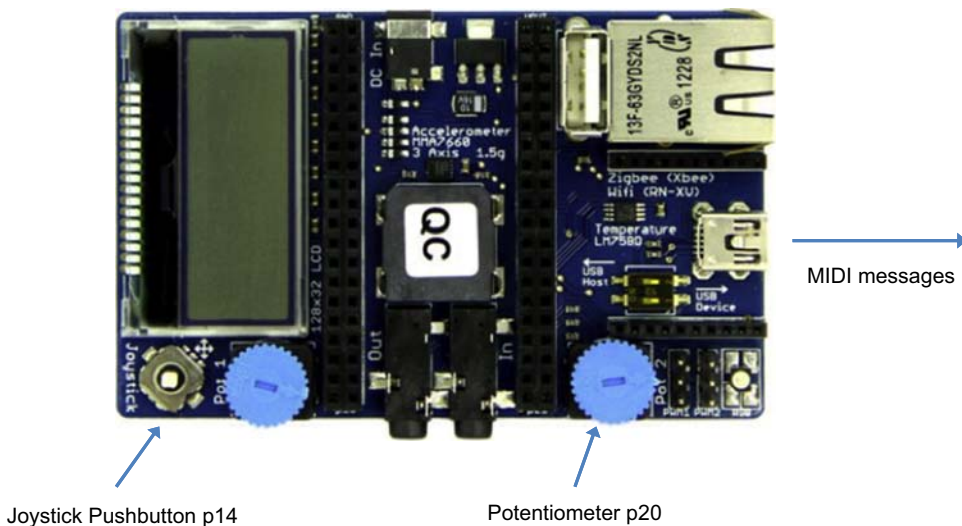
The output of Example 12.6 is shown in the MIDI control window of Apple Logic software in [Fig. 13.1](#).

 It is possible to use the mbed application board to easily build a more complex MIDI controller, making use of its on-board switches and potentiometers. For example, the app board's joystick incorporates a push button connected to pin 14; this can be used to action a midi note “on” message, while the potentiometer on pin 20 can be used to set the pitch of the note to be played, as shown in [Fig. 13.2](#).

Program Example 13.2 shows the simple program code to implement a MIDI controller with the mbed application board. This example automatically sends a **NoteOff()** message 200 ms after the note is switched on, meaning that short burst midi sounds are heard with each push button press. The note to be played is set to have a minimum value of 48 (MIDI note C3) and a maximum value of $48 + 72 = 120$ (MIDI note C8) dependent on the value of the analog input attached to the potentiometer on pin 20.

**Figure 13.1**

MIDI notes generated by the mbed and recorded into Apple Logic software.

**Figure 13.2**

Using the mbed application board as a MIDI controller.

```

/* Program Example 13.2: MIDI messaging controlled by switch and potentiometer
*/

#include "mbed.h"
#include "USBMIDI.h"

USBMIDI midi;                                // initialise MIDI interface
DigitalIn Switch(p14);
AnalogIn Ain(p20);

```

```

int main(){
  while (1) {
    if (Switch == 1) {
      int note = 48+72*Ain;           // calculate note value
      midi.write(MIDIMessage::NoteOn(note)); // note on
      wait(0.2);
      midi.write(MIDIMessage::NoteOff(note)); // note on
    }
  }
}

```

Program Example 13.2: MIDI messaging controlled by a switch and a potentiometer

■ Exercise 13.1

Modify Program Example 13.2 so that the MIDI note remains on for the duration of the push button press. This can be done by moving the **NoteOff** message to be within an **else** statement of the conditional expression.

How does this affect the performance of your MIDI controller? Can you think of ways to further improve or enhance the system with more advanced programming or by utilizing more sensors or inputs on the mbed application board?



13.2.2 Reading USB MIDI Data on the mbed

It is possible to use the mbed to read MIDI messages from a standard USB MIDI controller device. There are many types of MIDI controller on the market including MIDI keyboards, guitars, harps, woodwind instruments, drum pad controllers, and a whole number of abstract devices that do not resemble any type of traditional musical instrument. The simplest MIDI controller is a digital keyboard that sends note on and off data as well as the velocity of the key press. The KORG nanoKEY2 MIDI keyboard (as shown in [Fig. 13.3](#)) is an inexpensive MIDI controller that has two octaves of keys arranged in the design of a conventional piano keyboard.

To read MIDI messages from an external controller, the mbed program code needs to define a function that will run whenever a MIDI message is received. Essentially, this is a hardware interrupt routine that is managed by the **USBMIDI** library and is called every time a MIDI message event is encountered. Program Example 13.3 sets up a MIDI message interrupt and attaches it to a function that displays the MIDI key and velocity data to a host terminal. If using the mbed application board, Program Example 13.3 can easily be altered to output the MIDI data direct to the on-board LCD.

**Figure 13.3**

KORG nanoKEY2 MIDI keyboard. *Image courtesy of KORG.*

/ Program Example 13: Read MIDI messages and display key and velocity data to a host terminal application*

```

                                                                    */
#include "mbed.h"
#include "USBMIDI.h"

USBMIDI midi;

void read_message(MIDIMessage msg){
    switch (msg.type()) {
        case MIDIMessage::NoteOnType:
            printf("Note On key:%d vel:%d\n", msg.key(),msg.velocity());
            break;
        case MIDIMessage::NoteOffType:
            printf("Note Off key:%d vel:%d\n", msg.key(),msg.velocity());
            break;
    }
}

int main()
{
    midi.attach(read_message);          // call back for messages receive
    while (1) {
    }
}

```

Program Example 13.3: Reading and displaying MIDI messages

A more advanced program will generate a musical output based on the content of the MIDI messages. It is possible to create a simple audio output with a pulse width modulation (PWM) output of the mbed, which can have frequency determined by the note key value of the MIDI message. The correct musical frequency can be calculated from the note key value by [Eq. \(13.1\)](#):

$$\text{Frequency} = 440 * 2^{(d-69)/12} \quad (13.1)$$

It is then possible to calculate the correct PWM period and activate a square PWM output signal whenever a **NoteOn** MIDI message is received. **NoteOff** messages should set the PWM output to a 0% duty cycle. Based on this approach, Program Example 13.4 implements a simple mbed MIDI *synthesizer*.

```
/* Program Example 13.4: Simple mbed synthesizer                                     */
#include "mbed.h"
#include "USBMIDI.h"

USBMIDI midi;
PwmOut sound(p21);

void read_message(MIDIMessage msg){
    float freq=440*pow(2,(msg.key()-69)/12.0); // calculate note frequency
    switch (msg.type()) {
        case MIDIMessage::NoteOnType:
            printf("NoteOn key:%d vel: %d%d\n", msg.key(), msg.velocity());
            sound.period(1 / freq);           // Set PWM frequency
            sound = 0.5;                      // Switch PWM on (50% duty cycle)
            break;
        case MIDIMessage::NoteOffType:
            printf("NoteOff key:%d, vel: %d\n", msg.key(), msg.velocity());
            sound = 0;                        // Switch PWM off (0% duty cycle)
            break;
    }
}

int main()
{
    midi.attach(read_message); // attach read_message function to interrupt
    while (1) {
    }
}
```

Program Example 13.4: Simple mbed MIDI synthesizer

■ Exercise 13.2

Modify Program Example 13.4 to incorporate the MIDI velocity value. You can use the MIDI velocity to set the volume by modifying the PWM duty cycle. MIDI velocity values are read as integers in the range 0–127, so it will be necessary to convert this value with a suitable calculation. Design a conversion calculation so that minimum volume is represented by a PWM duty cycle of 10% and maximum volume represented by duty cycle of 50%.

With the velocity addition made, the mbed MIDI system should give louder musical notes determined by the strength of the key press on the MIDI keyboard.

13.3 Digital Audio Processing

Digital audio processing, or more generally *digital signal processing* (DSP), refers to the real-time computation of mathematically intensive algorithms applied to data signals, for example, 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 algorithms usually require many numbers to be multiplied and added together, so a DSP chip provides specific internal 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 actually possible to perform DSP applications with any microprocessor or microcontroller, though specific DSP chips will out-perform a standard microprocessor with respect to execution time and code size efficiency. Even though the mbed LPC1768 is not a dedicated DSP chip, it is sufficiently powerful to experiment with simple DSP concepts and projects.

13.3.1 Input and Output of Digital Audio Data With the mbed

It is possible to develop an mbed program that reads an analog audio signal in via the ADC, processes the data digitally, and then outputs the signal via the DAC. The analog output signal can be visualized on an oscilloscope or evaluated audibly by connecting it to a loudspeaker amplifier (such as a set of portable PC speakers).

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:

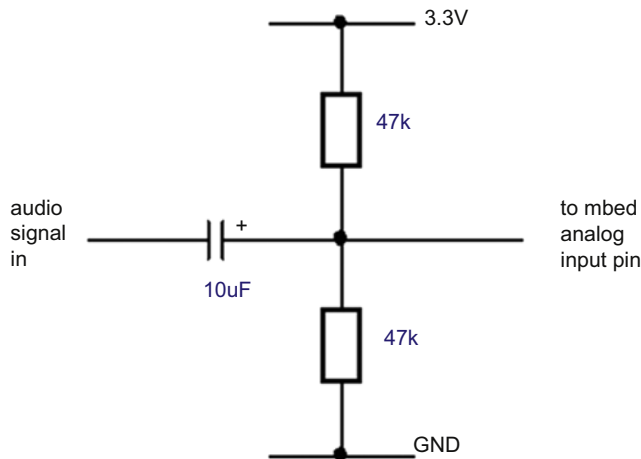
200hz.wav—an audio file of a 200 Hz sine wave

1000hz.wav—an audio file of a 1000 Hz sine wave

200hz1000hz.wav—an audio file with the 200 and 1000 Hz audio mixed

Each audio signal should be mono and around 60 s 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 route the audio signal to the mbed

**Figure 13.4**

Input circuit with coupling capacitor and bias resistors.

by connecting the host PC's audio output 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 isn't 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.

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 DC (direct current) offset. The circuit shown in [Fig. 13.4](#) effectively couples the host PC's audio output to the mbed. Create a new project and enter the code of Program Example 13.5.

```
/* Program Example 13.5 Audio signal input and output
*/

#include "mbed.h"
//mbed objects
AnalogIn Ain(p15);           // audio signal in
AnalogOut Aout(p18);         // audio signal out
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 13.5: Audio signal input and output

As can be seen, Program Example 13.5 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 μ s, which sets up the 20 kHz rate. The input is now sampled and processed at this regular rate, within **s20khz_task()**.

■ Exercise 13.3

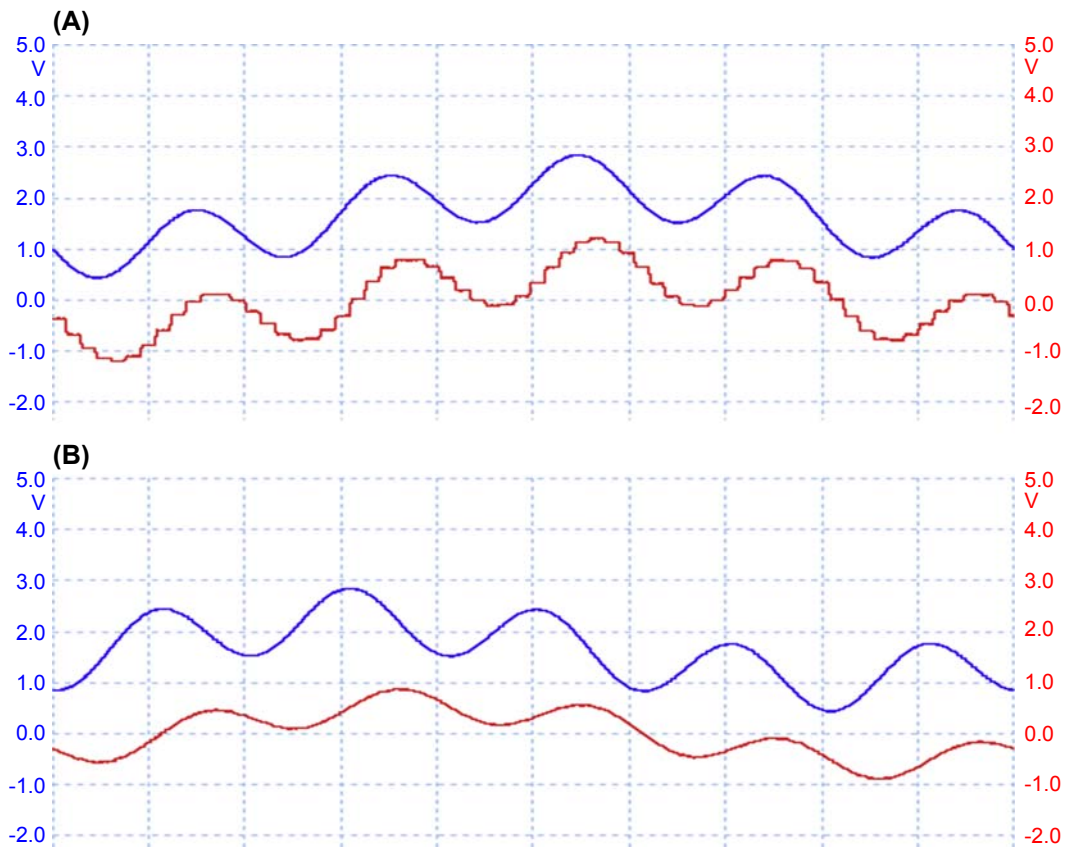
Compile Program Example 13.5 and use a two-channel oscilloscope to check that the analog input signal and the DAC output signal are similar. Use the oscilloscope to see how accurate the DAC output is with respect to the analog input signal for all three audio files. Consider amplitude, phase, and the waveform profile.

You will also need to implement the input circuit shown in [Fig. 13.4](#). ■

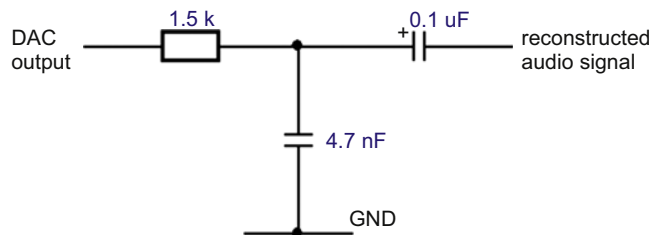
13.3.2 Signal Reconstruction

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 [Fig. 13.5A](#).

With many audio DSP systems, the analog output from the DAC is converted to a reconstructed signal by implementing an analog *reconstruction filter*, this removes all steps from the signal leaving a smooth output. In audio applications, a reconstruction filter is usually designed to be a low-pass filter (LPF) with a cut-off frequency at around 20 kHz, because the human hearing range doesn't exceed 20 kHz. The reconstruction filter shown in [Fig. 13.6](#) can be implemented with the current project (which gives the complete DSP input/output circuit as shown in [Fig. 13.7](#)). Note that after the LPF, a *decoupling capacitor* is also added to remove the 1.65 V DC offset from the signal. Having removed the DC offset the signal can now be routed to a loudspeaker amplifier to monitor the processed DAC output.

**Figure 13.5**

Shows signal output without reconstruction filter (A) and with reconstruction filter implemented (B).

**Figure 13.6**

Analog reconstruction filter and decoupling capacitor.

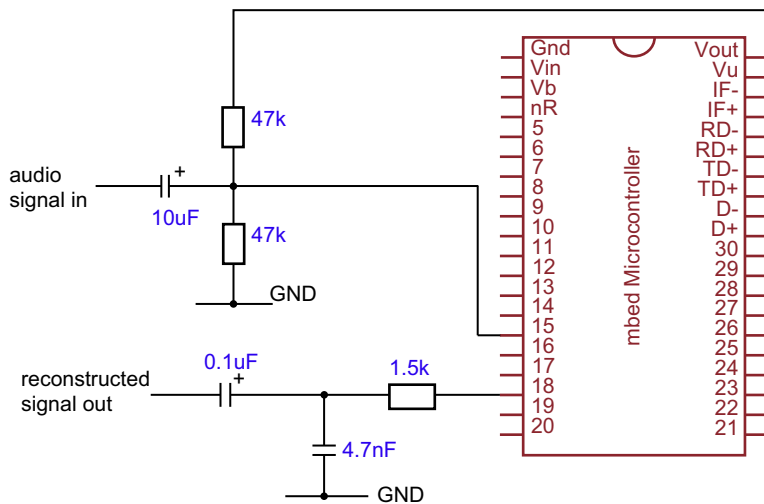
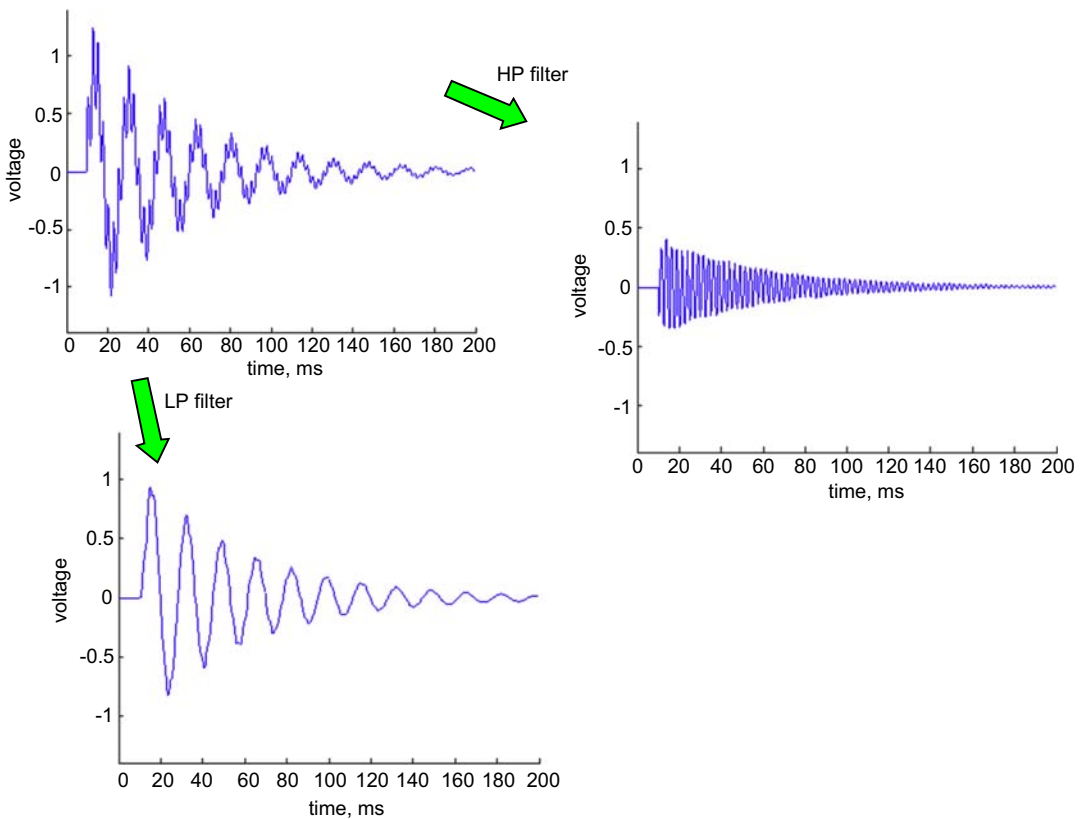


Figure 13.7
Audio signal input/output circuit.

As discussed in Chapter 4, in audio sampling systems it is often necessary to add an antialiasing filter prior to the analog-to-digital conversion. For simplicity we have not implemented this extra filter here. The mathematical theory associated with digital sampling and reconstruction is complex and beyond the scope of this book. For those interested, the theory of sampling, aliasing, and reconstruction is described well and in detail by a number of authors, including classic textbooks by Marven and Ewers [2] and Proakis and Manolakis [3].

13.4 Digital Audio Filtering Example

Filters are used to remove chosen frequencies from a signal, as shown in Fig. 13.8 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 remove the high frequency component—by implementing an LPF. A filter also has a *cut-off frequency*, which determines which signal frequencies are in the *pass-band* (and are still evident after the filtering) and those which are in the *stop-band* (which are removed by the filtering operation). The cut-off effect is not perfect, however, as frequencies which are 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. We can perform the filtering with an active or passive analog filter, but we can also perform the same process in software with a DSP operation. Digital filters are used extensively in digital audio applications, both for

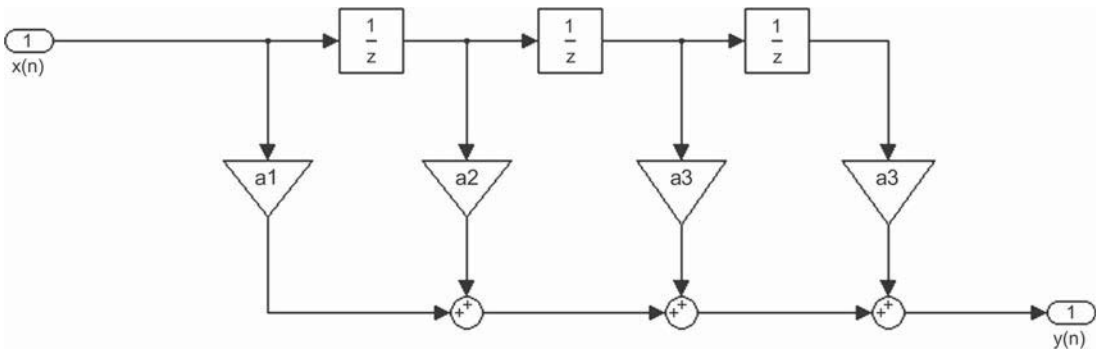
**Figure 13.8**

High-pass and low-pass filtered signals.

removing unwanted (noise induced) frequencies and for creative uses with graphical and parametric equalizer effects, among others.

We won't go deeply into the maths for digital filtering, but the software process relies heavily on addition and multiplication. Fig. 13.9 shows an example block diagram for a simple digital filtering operation.

Fig. 13.9 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 Fig. 13.9 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 antialiasing applications.



- $x(n)$ 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 13.9

Third-order finite impulse response filter.

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 example, if the filter taps in Fig. 13.9 all have a value of 0.25, then this implements a simple mean average filter (a crude low-pass filter) which uses multiply-and-accumulate calculations to continuously average the last four consecutive data values.

13.4.1 Implementing a Digital Low-Pass Filter on the mbed

Considering the audio file **200hz1000hz.wav** (an audio file with 200 and 1000 Hz signals mixed together), we will add a digital LPF 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.

In this example we will use a third-order *infinite impulse response* (IIR) filter, as shown in Fig. 13.10. 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.

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:

$$y(n) = b_0x(n) + b_1x(n-1) + b_2x(n-2) + b_3x(n-3) + a_1y(n-1) + a_2y(n-2) + a_3y(n-3) \quad (13.2)$$

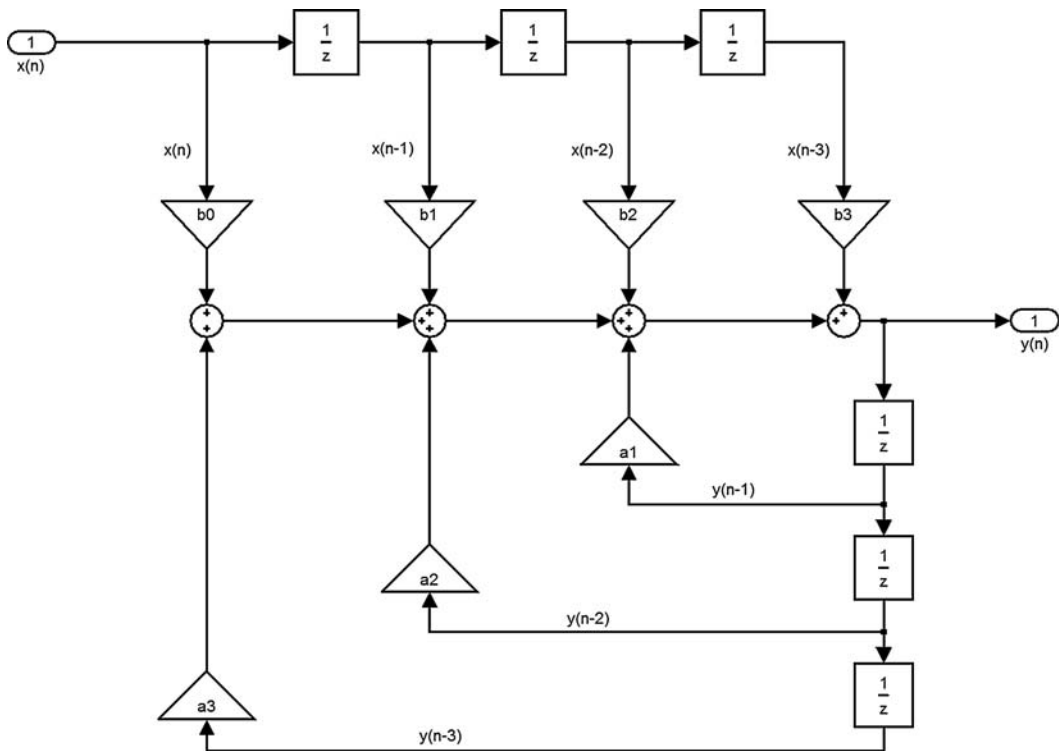


Figure 13.10
Third-order digital IIR filter.

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, a_{0-3} and b_{0-3} are coefficients (filter taps) which define the filter's performance.

Eq. (13.2) can be implemented 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 [4], and those provided in Ref. [5].

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 13.6. We chose a 600 Hz LPF because the cut-off is exactly half way between the two signal frequencies contained in the audio file **200hz1000hz.wav**.

```

/*Program Example 13.6 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 13.6: 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 Ref. [5]. 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 13.4

Create a new mbed program based on Program Example 13.5, only this time add the function for the LPF seen in Program Example 13.6. 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 Fig. 13.7.

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 13.5

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 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.



13.4.2 Digital High-Pass Filter

The implementation of a 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 13.6

Add a second switch to the circuit, and 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. You should now be able to listen to the simultaneous 200 and 1000 Hz audio signal and filter either the low frequency or the high frequency, or both, dependent on which switch is pressed.



13.5 Delay/Echo Effect

A number of audio effects use DSP systems for manipulating and enhancing audio. These can be useful for live audio processing (guitar effects, for example) or for postproduction. 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. We can therefore manipulate the speed of repetition and the amount of feedback attenuation. This effect is used commonly as a guitar effect and for vocal processing and enhancement. Fig. 13.11 shows a block diagram design for a simple delay effect unit.

To implement the system design shown in Fig. 13.11 we need to store historical sample data, so that it can be mixed back in with immediate data. We therefore need to copy sampled digital data 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.

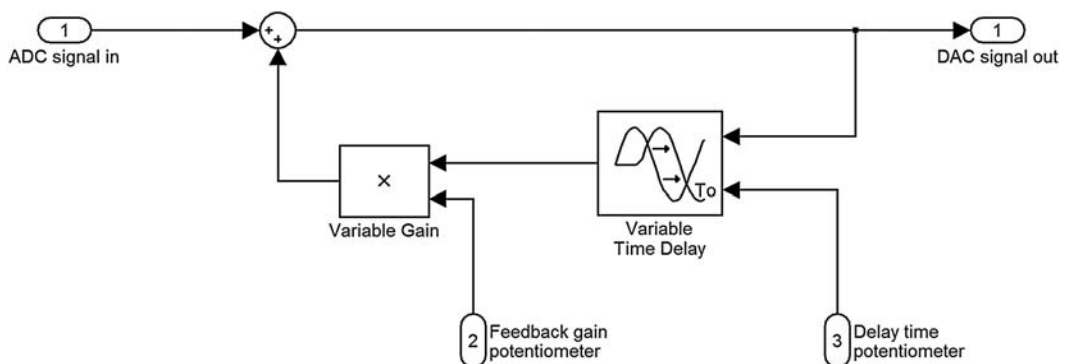


Figure 13.11
Delay effect block diagram.

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 13.7 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 normalise
    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 13.7: 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

```



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 -32768 to 32767 decimal) or **unsigned** (to use the range $0-65535$). 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 Fig. 13.7 can be used here. This demonstrates the advantage of a single hardware design which can operate multiple software features in a digital system. We will need to add potentiometers to mbed analog inputs to control the feedback speed and gain, as shown in Fig. 13.11. Program Example 13.7 defines the delay and feedback potentiometers being connected to mbed analog inputs on pins 16 and 17, respectively.

■ Exercise 13.7

Implement a new project with the code shown in Program Example 13.7. 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 similar echo response to that shown in Fig. 13.12. 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.

Fig. 13.12 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 effect unit by adding extra signal conditioning, variable amplification stages, and enhanced output conditioning. Some good examples are given in Ref. [6].

13.6 Working With Wave Audio Files

13.6.1 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). PCM specifically refers to the coding of amplitude signal data at a fixed sample rate, so 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

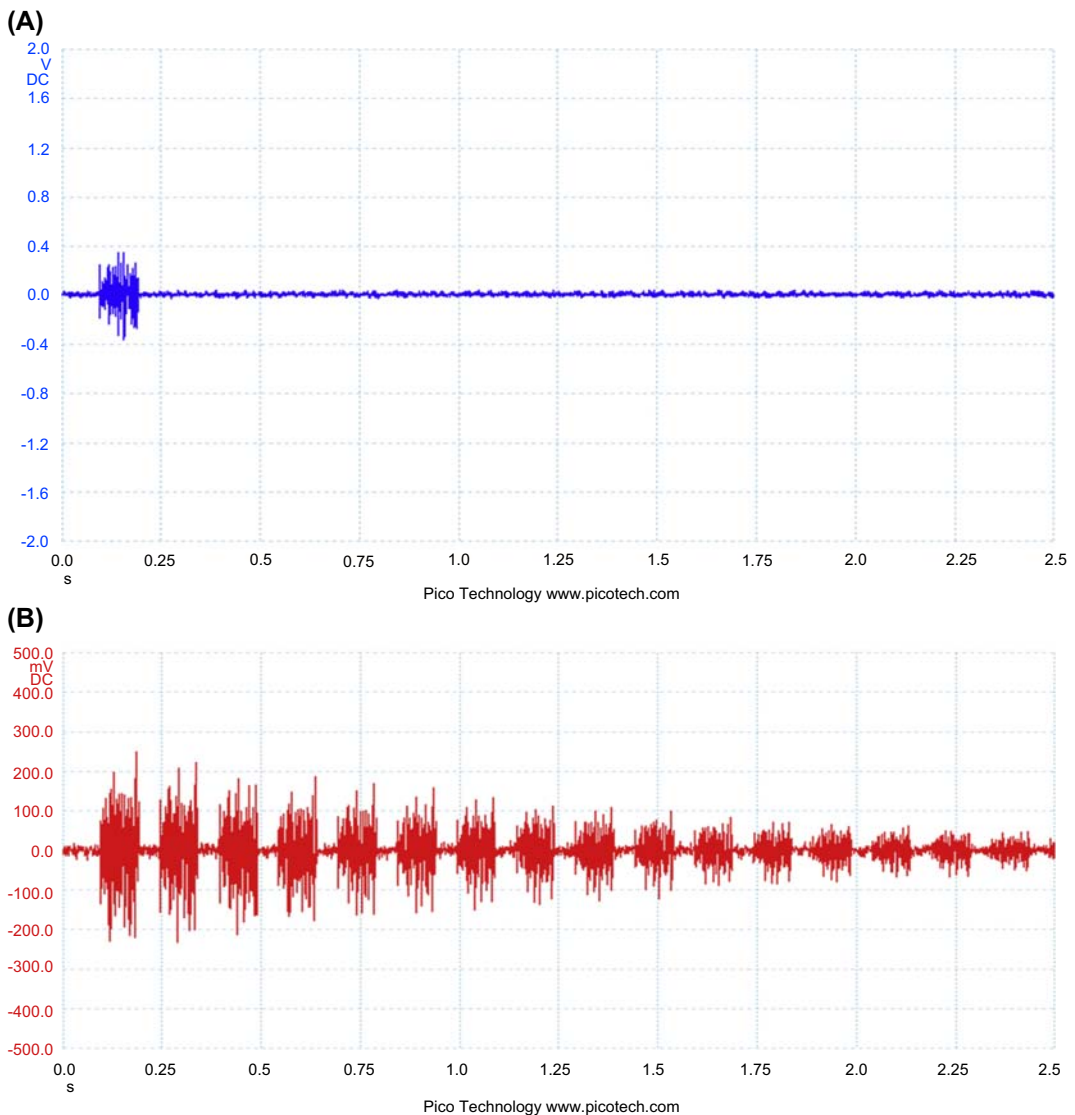


Figure 13.12

Input signal (A) and output signal (B) for the digital echo effect system.

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 13.2](#).

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 [Fig. 13.13](#). Here we see (the ASCII characters for ChunkID (“RIFF”), Format (“WAVE”), Subchunk1ID (“fmt”)),

Table 13.2: 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 This is the total size of the audio data in bytes.
Data	44	—	This is the actual data of size given by Subchunk2Size

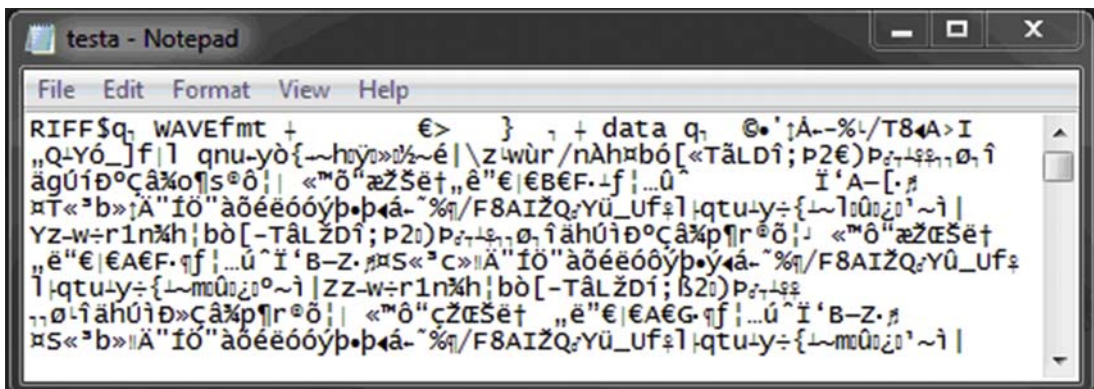


Figure 13.13

Wave file opened in text editor.

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 Fig. 13.14. 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 $0x00003E80 = 16000$ decimal.

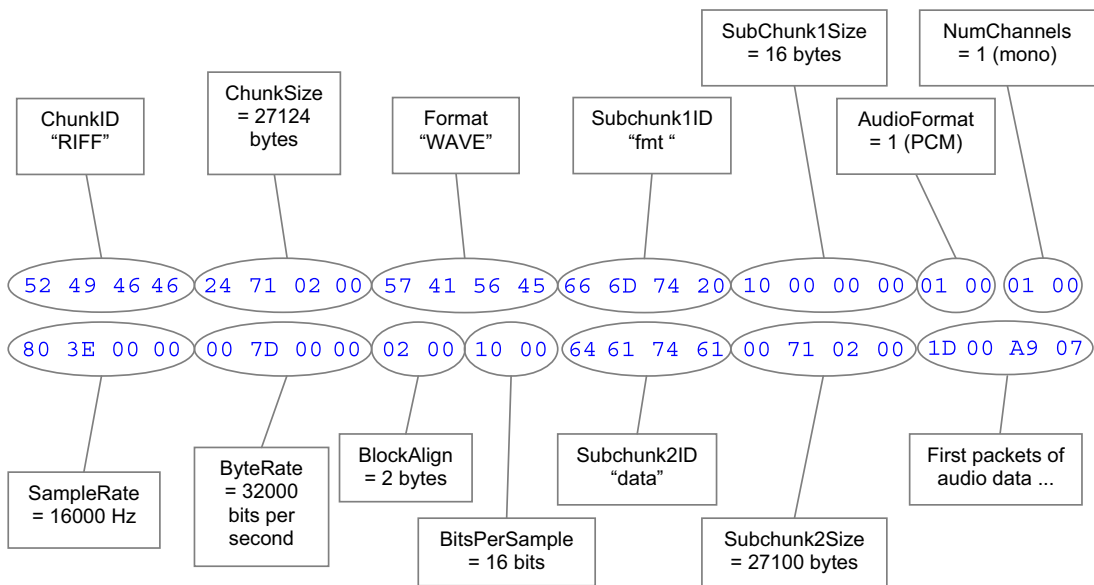


Figure 13.14
Structure of the wave file header data.

13.6.2 Reading the Wave File Header With the mbed

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 an 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 larger than the mbed's internal memory can store, but also because the file access through the SPI/SD card interface is very fast.

Program Example 13.8 reads a wave audio file called **test.wav**, uses the **fseek()** function to move the file pointer to the relevant position, and then reads the header data and displays this to a host terminal.

```
/* Program Example 13.8 Wave file header reader
```

*/

```
#include "mbed.h"
```

```
#include "SDFFileSystem.h"
```

```
SDFFileSystem sd(p5, p6, p7, p8, "sd");
```

```
char c1, c2, c3, c4;
```

```
// chars for reading data in
```

```
int AudioFormat, NumChannels, SampleRate, BitsPerSample ;

int main() {
    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) {
        printf("Wav file is PCM data\n\r");
    }
    else {
        printf("Wav file is not PCM data\n\r");
    }

    fread(&NumChannels, 2, 1, fp);      // find number of channels
    printf("Number of channels: %d\n\r",NumChannels);

    fread(&SampleRate, 4, 1, fp);       // find sample rate
    printf("Sample rate: %d\n\r",SampleRate);

    fread(&BitsPerSample, 2, 1, fp);    // find resolution
    printf("Bits per sample: %d\n\r",BitsPerSample);

    fclose(fp);
}
```

Program Example 13.8: 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.



In Program Example 13.8 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 13.8

Extend Program Example 13.8 to display **ByteRate** and **Subchunk2Size**, which indicates the size of the raw audio data within the file.

13.6.3 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 also 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 (see Fig. 13.15). The audio file **200hz.wav** as utilized previously in Section 13.3 can be used here.

The audio data in a 16-bit mono .wav file is arranged similar 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 known as *jitter*. Jitter occurs when audio data is not consistently available for the DAC, because interrupts and timing overheads sometimes make it



Figure 13.15

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

difficult for the audio to be streamed directly from data memory at a constant rate. We therefore need to use a buffered system to enable accurate timing control.

A good method to ensure accurate timing control when working with audio data is to use a *circular buffer*, as shown in Fig. 13.16. 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 doesn't matter so 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; firstly, 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); secondly, 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 implement control code to safeguard against data wrapping.

Program Example 13.9 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.

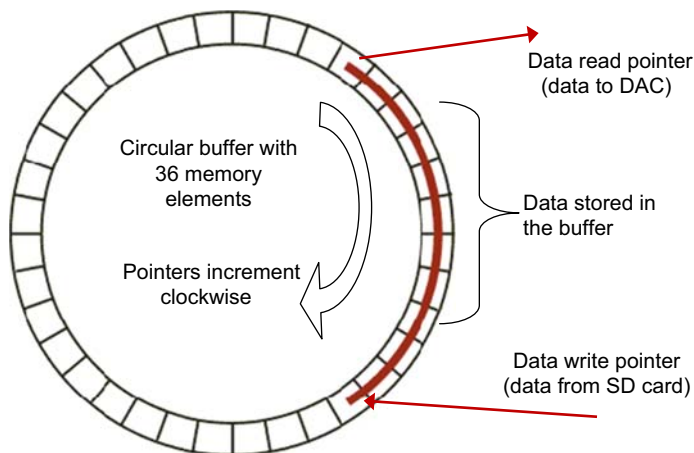


Figure 13.16
Circular buffer example.

```

/* Program Example 13.9: 16-bit mono wave player
                                                                    */
#include "mbed.h"
#include "SDFFileSystem.h"

#define BUFFERSIZE 0xffff      // number of data in circular buffer = 4096

SDFFileSystem sd(p11, p12, p13, p14, "sd");    //SD Card object
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 13.9: Wave file player utilising a circular buffer

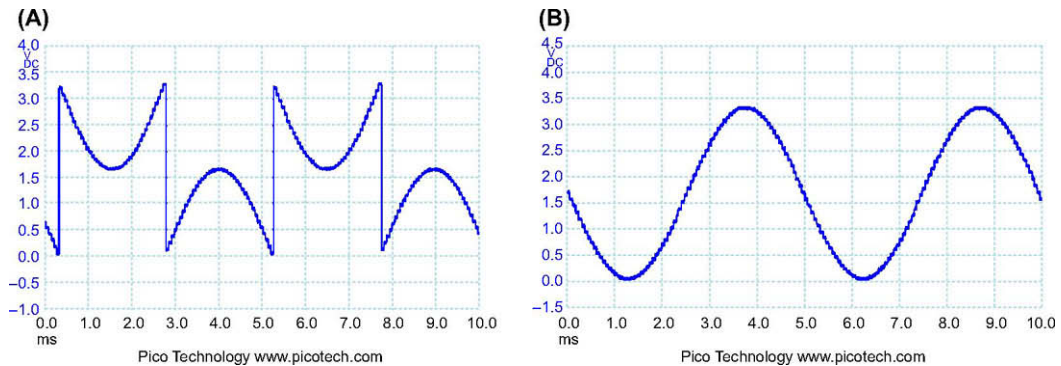


Figure 13.17

Wave file sine wave before and after two's-complement correction. (A) Raw data output and (B) two's-complement corrected output.

When Program Example 13.9 is implemented with a pure mono sine wave audio file, initially the results will not be correct. Fig. 13.17A 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 is 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 also discussed in Appendix A. A simple adjustment of the data is therefore required to output the correct waveform as shown in Fig. 13.17B.

■ Exercise 13.9

Modify Program Example 13.9 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 work out how to do this.

13.7 High-Fidelity Digital Audio With the mbed

As we have seen, the mbed LPC1768 has a built in 12-bit ADC and a 10-bit DAC. However, professional digital audio systems rely heavily on 16-bit and 24-bit systems to record and playback high-fidelity (i.e., low distortion) audio; the standard compact disc audio format uses 16-bit data at a sample frequency of 44.1 kHz, for example. Additionally, high-quality audio is often in a stereo format with left and right audio

channels. Therefore, to work with 16-bit (or greater) resolution audio data, a more powerful audio interface chip must be used.

13.7.1 Texas Instruments TLV320 Audio Codec and the I²S Protocol

The Texas Instruments TLV320AIC23B chip is a high-performance stereo audio convertor with integrated analog functionality. The TLV320's ADCs and DACs can process two channels of 16, 20, 24, or 32 bit data at sample rates up to 96 kHz. Conveniently, the Synergy AudioCODEC by DesignSpark is designed to easily interface the TLV320 to the mbed, with PCB mounted 3.5 mm stereo audio sockets and on-board analog signal conditioning (as shown in [Fig. 13.18](#)).

The TLV320 uses a serial protocol called I²S (*Inter-IC Sound*) to communicate with a microcontroller. This is a serial interface similar to those we have seen before, though specifically designed for communicating between digital audio devices. Phillips Semiconductors first introduced I²S as a communication standard in 1986. The standard was updated in 1996 and can be accessed from Ref. [7].

The I²S protocol uses three signal lines, shown in [Fig. 13.19](#). These are the continuous serial clock (SCK), word select (WS), and serial data (SD). The device generating SCK and WS is defined as the master. Serial data is transmitted in two's complement format with the MSB first. The word select line indicates the channel being transmitted (0 = left and 1 = right).

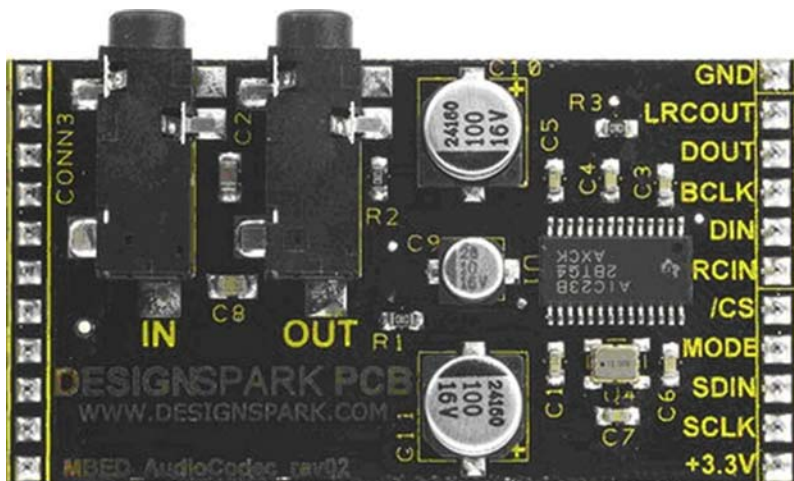


Figure 13.18

Synergy AudioCODEC. Image courtesy of DesignSpark.

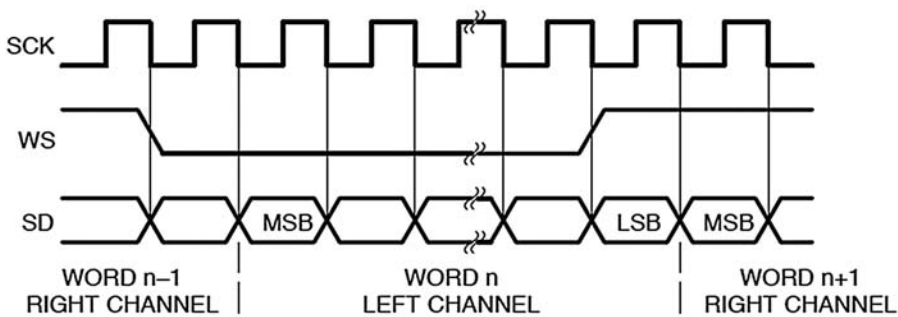


Figure 13.19
I²S interface timing.

The mbed has a built-in two-way I²S port on pins 5 (SD In), 6 (WS In), 7 (SCK), 8 (SD Out) and 29 (WS Out) so can be used to control the Synergy AudioCODEC. To connect the mbed to the Synergy AudioCODEC, we also need to use a standard I²C interface (for example on mbed pins 9 and 10) to send all control and setup data, since the I²S protocol only transfers audio data. The connections for wiring the AudioCODEC to the mbed is shown in [Table 13.3](#).

Two mbed libraries, developed by Daniel Worrall, are useful for implementing audio programs with this device, the **I2SSlave** library and the **TLV320** library (found at Refs. [\[8,9\]](#) respectively).

13.7.2 Outputting Audio Data From the TLV320

To output high-quality audio from the Synergy AudioCODEC, the board should be connected to the mbed as detailed in [Table 13.3](#). The **I2SSlave** and **TLV320** libraries by

Table 13.3: Connections for the Synergy AudioCODEC.

Synergy AudioCODEC	Protocol	mbed Pin	Pull Up/Pull Down
GND	—	1	—
LRCOUT	I ² S	29	—
DOUT	I ² S	8	—
BCLK	I ² S	7	—
DIN	I ² S	5	—
LRCIN	I ² S	6	—
/CS	—	1	Connect direct to GND
MODE	—	1	Connect direct to GND
SDIN	I ² C	9	2.2 kΩ pull up to 3.3 V
SCLK	I ² C	10	2.2 kΩ pull up to 3.3 V
+3.3V	—	40	—

developer Daniel Worrall (Refs. [8,9] respectively) should also be imported to a new mbed project. A simple mbed project can then be implemented using the structure shown in Program Example 13.10.

```
// Program Example 13.10 program structure for outputting high-quality audio
//
#include "mbed.h"
#include "TLV320.h"          // the I2SSlave header is linked from within TLV320.h
#define BUFFERSIZE 0xff      // number of data in circular buffer = 256
#define PACKETSIZE 8         // number of data values in a single audio package

TLV320 audio(p9, p10, 52, p5, p6, p7, p8, p29);          //TLV320 object

int CircularBuffer[BUFFERSIZE];          // circular buffer array
int ReadPointer=0;
int WritePointer=0;

// ** Additional variables to be declared here

// ** Function prototypes
void SetupAudio (void);
void PlayAudio(void);
void FillBuffer(void);

// ** Main function
int main(){
    SetupAudio();          // call SetupAudio function
    while (1) {
        FillBuffer(); //continually fill circular buffer
    }
}

// ** Additional functions to be added here
```

Program Example 13.10 program structure for outputting high-quality audio

Program Example 13.10 itself does not compile and run, because it is incomplete. However, it can be seen that the program's main function calls two other functions. Firstly, a single execution of the **SetupAudio()** function to initialize the TLV320, and secondly, a continuously executed function called **FillBuffer()**, which creates audio data and places it in the circular buffer.

A third function called **PlayAudio()** is also defined, which will be configured and called from within the **SetupAudio()** routine. The **SetupAudio()** function is shown in Program Example 13.11. It can be seen that the TLV320 object (named **audio**) is calibrated to power up and use a sample frequency of 44,100 Hz, 16-bit stereo data, and an output volume of 0.8 on both channels. A full and detailed explanation of the TLV320 API can

be found at Ref. [8]. Within the **SetupAudio()** function, the TLV320 is also configured to attach the function **PlayAudio()** to a timed interrupt that streams the audio data at the specified sample rate. The final `audio.start(TRANSMIT)` command actions the continuous audio transfer.

Program Example 13.11 includes the **PlayAudio()** function. The function uses the `audio.write(CircularBuffer, ReadPointer, PACKETSIZE)` function to write 8 integer values (because in this example **PACKETSIZE**=8) to the TLV320 from the circular buffer. After the write action completes the buffer's read pointer then increments by **PACKETSIZE** (since eight integers have been written). The read pointer value is reset to zero if it exceeds the buffer limit of 256 (0xff hexadecimal). A new variable called *theta* is also introduced in this function. The value of *theta* is intended to represent the difference between the values of the circular buffer's read and write pointers, so it is incremented every time data comes in and decremented every time data goes out. Obviously, this value cannot exceed the buffer size, which in this case is 255.

```
// Program Example 13.11 functions for initialising high-quality audio output
//
// ** Function to setup TLV320 ***
void SetupAudio(void){
    audio.power(0x02);           //power up TLV to audio input/output mode
    audio.frequency(44100);      //set sample frequency
    audio.format(16, 0);         //set transfer protocol - 16 bit, stereo
    audio.outputVolume(0.8, 0.8);
    audio.attach(&PlayAudio);    //attach interrupt to send data to TLV320
    audio.start(TRANSMIT);       //interrupt come from the I2STXFIFO only
}
// ** Function to read from circular buffer and send data to TLV320 ***
void PlayAudio(void){
    audio.write(CircularBuffer, ReadPointer, PACKETSIZE); // write to buffer
    ReadPointer += PACKETSIZE;           // increment read pointer
    ReadPointer &= BUFFERSIZE;           // if end of buffer then reset to 0
    theta -= PACKETSIZE;                 // decrement theta
}
```

Program Example 13.11: Functions for initializing high-quality audio output

Finally, the **FillBuffer()** function is needed to output audio data from the AudioCODEC board, this is shown in Program Example 13.12. The function describes the creation of a square wave data stream that is continuously written to the circular buffer. Data is only written if the value of *theta* is less than the buffer size, to avoid memory overflows. Data is written to the circular buffer in blocks of eight data samples, which is implemented by a simple **for** loop. Within the loop, the program decides whether to output a +1 or −1 value (i.e., the two different values for a simple square wave). Since the data output is in 16-bit two's-compliment format, the data output value (*x*) is therefore 0x8000 hexadecimal for −1 and 0x7fff for +1.

The switching period of the square wave is controlled by a simple counter that continuously increments and resets, outputting a high value for the first half period and a low value for the second half period.

The data is written to the circular buffer in the correct format to be output as stereo audio data. The TVL320 outputs both the 16-bit left and right stereo audio data within a single data 32-bit integer. In this format the most significant 16 bits represent the left channel and the least significant 16 bits represent the right channel. To output the generated square wave to both the left and right channels, we shift the variable *x* to the left by 16 bits and then perform a logic OR with itself. Once the loop has completed, the counters are all updated.

```
// Program Example 13.12 Function to load data to circular buffer
//
void FillBuffer(void){
    if (theta < BUFFERSIZE) {
        for (int i=0; i<PACKETSIZE; i++) {          // loop for PACKETSIZE samples
            // create squarewave with freq=fs/(2*halfperiod)
            if ((counter+i)<halfperiod) {
                x=0x8000;                          // 2s compliment for -1 (16 bit)
            } else if ((counter+i)<(halfperiod*2)) {
                x=0x7fff;                          // 2s compliment for +1 (16 bit)
            } else {
                counter=0;
            }
            //put data in buffer (right = MS 16 bits, left = LS 16 bits)
            CircularBuffer[WritePointer+i]=(x<<16)|(x);
        }
        theta+=PACKETSIZE;                          // increment theta
        WritePointer+=PACKETSIZE;                    // increment write pointer
        WritePointer &=BUFFERSIZE;                  // if end of buffer then reset to 0
        counter+=PACKETSIZE;                        // increment square wave counter
    }
}
```

Program Example 13.12: Function to load square wave data to the circular buffer

The frequency of the square wave will depend on the value of **halfperiod**. The square wave frequency in Hz is given by

$$\text{square wave frequency} = \frac{\text{audio sample rate}}{2 \times \text{halfperiod}} \quad (13.3)$$

For example, with a sample rate of 44,100 Hz and a **halfperiod** value of 50, the square wave frequency will be 441 Hz.

■ Exercise 13.10

Build, compile, and run the high-quality audio output program as specified in Program Examples 13.10, 13.11, and 13.12. You will need to connect the mbed and the AudioCODEC board as described in [Table 13.3](#).

The functions shown in Program Example 13.11 and 13.12 need adding to Program Example 13.10. The variables **theta**, **x**, and **halfperiod** will need to be defined as integers. Initialize **theta** and **x** to zero and initialize **halfperiod** to 50.

Check with an oscilloscope that the expected square wave is output from both the left and right channels of the AudioCODEC's output socket. You can also check it audibly with a loudspeaker or set of headphones plugged into the socket.

Change the initialization value of **halfperiod** and check that the square wave frequency changes as expected.

■

■ Exercise 13.11

It is possible to use a potentiometer to adjust the frequency of the square wave output in real time. To do this you will need to connect and configure a potentiometer as an analog input, for example, on pin 16 as follows:

```
AnalogIn Ain(p16);
```

You can then add a statement to the **FillBuffers()** function that use the **Ain** value to modify the value of **halfperiod**. For example, including the statement

```
halfperiod=50*(1+Ain);
```

will mean that the value of **halfperiod** ranges between 50 and 100 based on the analog input value.

Implement this code update and check that the output frequency can be controlled by a potentiometer.

■

13.7.3 High-Fidelity Wave File Player

Given the technologies already discussed, it is now possible to build a high-fidelity wave file player that utilizes both an SD card (for file storage) and the Synergy AudioCODEC for audio output—essentially a portable digital music player. The program code for the wave file player combines the code presented above in [Sections 13.6 and 13.7](#).

Program Example 13.13 shows the main structure for a high-quality wave file player. It can be seen that the structure is virtually identical to that in Program Example 13.11, except that the **SDFFileSystem** object has been created and a new **OpenFiles()** function has been included. For accessing data from the SD card, we will need a larger circular buffer to account for greater timing inconsistencies. In Program Example 13.13 the circular buffer size is therefore set to 4096 (0xFFF hexadecimal).

```
// Program Example 13.13: Code structure for a high-quality wave file player.
//
#include "mbed.h"
#include "SDHCFileSystem.h"
#include "TLV320.h"

SDFFileSystem sd(p11, p12, p13, p14, "sd");          //SD Card object
TLV320 audio(p9, p10, 52, p5, p6, p7, p8, p29); //TLV320 object

#define BUFFERSIZE 0xfff    // number of data in circular buffer = 4096
#define PACKETSIZE 8        // number of data values in a single audio package

int CircularBuffer[BUFFERSIZE];
int ReadPointer = 0;
int WritePointer = 0;
int theta = 0;
int WavData[PACKETSIZE];
FILE *WavFile;

// ** Function prototypes
void SetupAudio (void);
void PlayAudio(void);
void FillBuffer(void);
void OpenFiles(void);

// ** Main function
int main() {
    OpenFiles();
    SetupAudio();
    while (1) {
        FillBuffer();          //continually fill circular buffer
    }
}

// ** Additional functions to be added here
```

Program Example 13.13: Code structure for a high-quality wave file player

The **OpenFiles()** and **FillBuffer()** functions are shown in Program Example 13.14. The **FillBuffer()** function simply reads a packet of 32-bit (4-byte) audio data from the SD card's wave file (in this example called **Audio.wav**) and puts each sample into the circular buffer.

```

// Program Example 13.14 Functions to open SD wave file and read audio data
//
// *** function to open file ***
void OpenFiles(void) {
    wait(1);
    wavFile = fopen("/sd/Audio.wav", "r");    //open file
    fseek(wavFile, 44, SEEK_SET);            // offset to data
    wait(1);
}

// *** Function to load circular buffer from SD Card ***
void FillBuffer(void) {
    if (theta < BUFFERSIZE) {
        fread(&WavData, 4, PACKETSIZE, WavFile); //read 4 byte (i.e. 32 bit)
                                                    // data from the wav file

        for (int i=0; i<PACKETSIZE; i++) {
            CircularBuffer[WritePointer]=WavData[i];
            theta++;
            WritePointer++;
            WritePointer &= BUFFERSIZE;
        }
    }
}

```

Program Example 13.14 Functions to open SD wave file and read audio data

For completing the high-quality wave player, the same **SetupAudio()** and **PlayAudio()** functions can be used as before, i.e., those given in Program Example 13.11.

■ Exercise 13.12

Build, compile, and run the high-quality wave player program as specified in Program Examples 13.9 and 13.10 and incorporating the audio functions defined in program Example 13.7. Don't forget you will need to import the following mbed libraries to your program:

```

FATFileSystem
SDHCFFileSystem
I2SSlave
TLV320

```

Using a PC with an SD card reader, copy a 16-bit, 44.1 kHz audio file called **Audio.wav** onto the SD card and ensure that it plays back clearly (i.e., with no significant noise or jitter) through the AudioCODEC device. (If you don't have access to a 16-bit, 44.1 kHz audio file, then go to the book website where a file called **Audio.wav** can be downloaded.)

13.7.4 High-Fidelity Audio Input (Recording)

To record or process high-quality audio with the TLV320, we need to implement the **read()** function of the TLV320 class. When called, the **read()** function puts four 32-bit words from the ADC into a receive buffer array called **rxBuffer**.

Program Example 13.15 shows audio being captured (recorded) by the TLV320 and immediately being played out also. Note that for input and output to be enabled at the same time, the TLV **start()** function needs to specify **BOTH**, as in the line of code **audio.start(BOTH)** seen in the **PlayAudio()** function. In this program, all the processing is managed within the **PlayAudio()** function that is attached to the TLV320's interrupt handler, since both the reading and writing of data both need to be executed at timed intervals to avoid any audible jitter.

```
// Program Example 13.15: Audio input / output with the TLV320
//
#include "mbed.h"
#include "TLV320.h"

TLV320 audio(p9, p10, 52, p5, p6, p7, p8, p29); //TLV320 object

#define BUFFERSIZE 0xff          // number of data in circular buffer = 256
#define PACKETSIZE 4            // number of data values in a single audio package
int CircularBuffer[BUFFERSIZE];
int ReadPointer = 0;
int WritePointer = 0;

// ** Function to read from circular buffer and send data to TLV320 ***
void PlayAudio(void){
    // read data to circular buffer and increment pointers
    audio.read();                      // read 4 values to rxBuffer
    for (int i=0; i<PACKETSIZE; i++) {
        CircularBuffer[WritePointer+i] = audio.rxBuffer[i]; // transfer data
    }
    WritePointer+=PACKETSIZE;
    WritePointer &= BUFFERSIZE;

    // write data from circular buffer and increment counters
    audio.write(CircularBuffer, ReadPointer, PACKETSIZE);
    ReadPointer += PACKETSIZE;
    ReadPointer &= BUFFERSIZE;
}

// ** Function to setup TLV320 ***
void SetupAudio(void){
```

```
    audio.power(0x02);           //power up TLV
    audio.frequency(44100);       //set sample frequency
    audio.format(16, 0);          //set transfer protocol - 16 bit, stereo
    audio.outputVolume(0.8, 0.8);
    audio.inputVolume(0.8, 0.8);
    audio.attach(&playAudio);     //attach interrupt handler
    audio.start(BOTH);
}

// ** Main function
int main(){
    SetupAudio();
    while (1) {
    }
}
```

Program Example 13.15: Audio input/output with the TLV320

■ Exercise 13.13

Run Program Example 13.15 and verify that high-fidelity stereo audio can be read and output through the TLV320. You will need to plug an audio source (from a portable music player or compact disc player) into the TLV320 and then listen to the audio output on a set of headphones or portable loudspeakers. Verify that the audio out of the TLV320 is of a similar fidelity to the output directly from the music player. When running audio through the TLV320 you should notice no discernable difference in audio signal quality (i.e., clear music and lyrics with no clicks, noise, or jitter).

Modify the program to swap the left and right channel audio data. To do this you will need to extract the 16-bit left and right data values from each 32-bit audio sample that is received. You can then recombine the left and right data in a reversed form, before putting the modified 32-bit value into the circular buffer.

The audio file **StereoSW.wav** (which is available from the book website) can be useful for verifying that the channels have successfully been swapped. This stereo audio file contains both 200 and 1000 Hz sine waves on the left and right channels respectively, so it is easy to verify if these signals have been successfully swapped by your program.

■

Program Example 13.15 describes the basic data input code needed to build high-fidelity recording or DSP applications. The program could be modified to include more advanced DSP, such as creating mono or stereo audio effects that maintain high-quality audio. Equally, the program could be modified to record audio data and generate a wave audio file on an SD card for future recall and playback.

13.8 Summary on Digital Audio and Digital Signal Processing

In this chapter we have introduced a number of aspects of digital audio that can be explored with the mbed. Of course, digital audio itself is a vast subject area, which requires many more specialist resources to explore at a complete and professional level. The MIDI digital audio standard has been introduced, and a number of mbed examples for reading and writing MIDI data have been presented.

As shown, 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, the wave audio 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.

13.9 Mini Project: Portable Music Player

Experiment with using an LCD display and digital push buttons to develop your own portable stereo audio player. A good design will have an SD card report the names of the audio wave files that are contained within and show those file names on an LCD display. Using four digital push buttons for “up,” “down,” “play,” and “stop”, the user can scroll through the different audio files and choose which is to be played out through the Synergy AudioCODEC. Once the program is complete, power the system with a DC battery voltage to realize your own portable music player.

Chapter Review

- The MIDI protocol allows music systems to be connected together and communicate musical data, such as pitch and velocity, so that novel electronic music performance and playback systems can be configured.
- Digital audio processing 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, and other audio processing algorithms can be used for manipulating the sonic attributes of music and audio.
- 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 come in 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.
- Digital audio systems require high-resolution (minimum 16-bit) ADCs and DACs to record and playback high-quality music files.
- Data management and effective buffering is required to ensure that timing and data overflow issues are avoided.

Quiz

1. What does the acronym MIDI stand for?
2. What data might be contained in a MIDI message (give two examples)?
3. What is a circular buffer and why might it be used in digital audio systems?
4. What is the difference between an FIR and an IIR digital filter?
5. Explain the role of analog biasing and antialiasing when performing an analog-to-digital conversion.
6. What is a reconstruction filter and where would this be found in an audio DSP system?
7. What are the potential effects of poor timing control in an audio DSP system?
8. 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
9. Explain the I²S communications protocol and how it is utilized in digital audio applications.
10. Give 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

- [1] Summary of MIDI Messages. <http://www.midi.org/techspecs/midimessages.php>.
- [2] C. Marven, G. Ewers, A Simple Approach to Digital Signal Processing, Wiley Blackwell, 1996.
- [3] J.G. Proakis, D.K. Manolakis, Digital Signal Processing: Principles, Algorithms and Applications, Prentice Hall, 1992.

- [4] The Mathworks, FDATool — Open Filter Design and Analysis Tool, 2010. <http://www.mathworks.com/help/toolbox/signal/fdatool.html>.
- [5] T. Fisher, Interactive Digital Filter Design (Online Calculator), 2010. <http://www-users.cs.york.ac.uk/~fisher/mkfilter/>.
- [6] I. Sergeev, Audio Echo Effect, 2010. http://dev.frozenskimo.com/embedded_projects/audio_echo_effect.
- [7] Phillips Semiconductors, I²S Bus Specification, 1996. <https://www.sparkfun.com/datasheets/BreakoutBoards/I2SBUS.pdf>.
- [8] I2SSlave Library by Daniel Worrall. https://developer.mbed.org/users/d_worrall/code/I2SSlave/.
- [9] TLV320 Library by Daniel Worrall. https://developer.mbed.org/users/d_worrall/code/TLV320/.