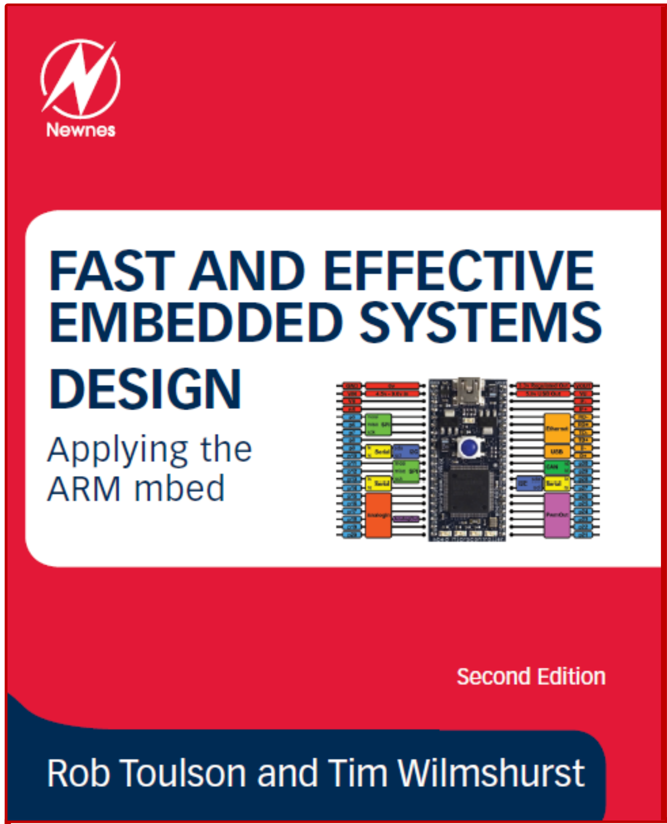

Newnes

FAST AND EFFECTIVE EMBEDDED SYSTEMS DESIGN

Applying the ARM mbed

Second Edition

Rob Toulson and Tim Wilmshurst

Chapter 13: Working with Digital Audio

rt rev. 12.9.16

If you use or reference these slides or the associated textbook, please cite the original authors' work as follows:

Toulson, R. & Wilmshurst, T. (2016). Fast and Effective Embedded Systems Design - Applying the ARM mbed (2nd edition), Newnes, Oxford, ISBN: 978-0-08-100880-5.

www.embedded-knowhow.co.uk

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 advantages of reliability, sound quality and storage space, as well as significant benefits that can be delivered through portability, online music catalogues 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).

While the audio is represented as digital data, it can also be manipulated through *digital signal processing* (DSP), which relies on mathematical algorithms to enhance or change the audio properties of the data, such as through filtering, or to delay and repeat sounds, or to make a poor quality recording sound perfectly in tune.

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.

Sending USB MIDI data from an mbed

MIDI is a serial message protocol that allows digital musical instruments to communicate with digital signal processing systems which can turn those signals into sound. In 1999 a new MIDI standard was developed to allow messaging through the more modern USB protocol.

In a very simple MIDI system the most valuable information is

- Whether a musical note is to be switched on or off,
- The pitch of the note.

In most cases an audio sequencer software package will automatically recognise an mbed MIDI interface and allow 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 initialising the interface as follows:

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

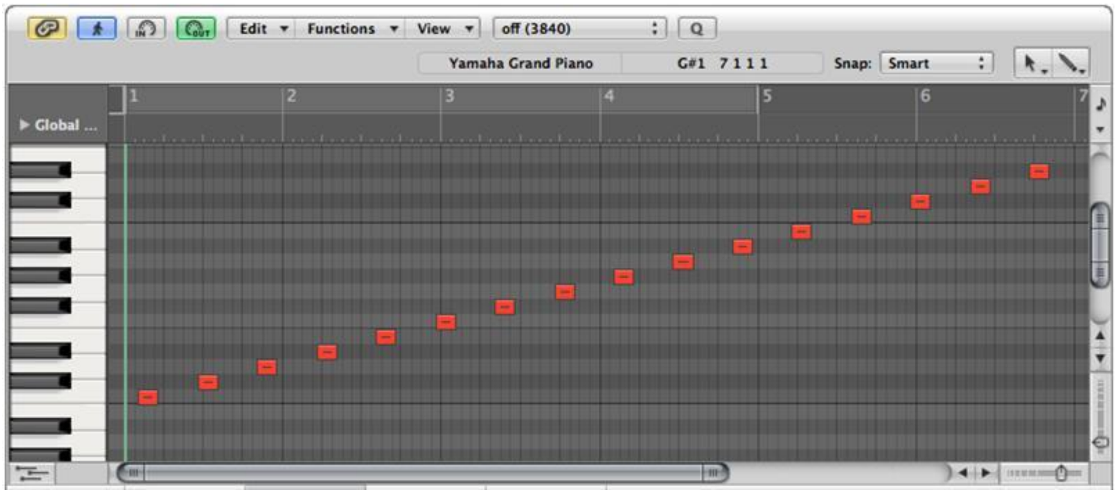
A MIDI message to sound a note is activated by the following command:

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

Read further: See table of MIDI note values in Section 13.2

Sending USB MIDI data from an mbed

```
/* 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 (potentiometer)

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
        }
    }
}
```



The output of Program Example 13.1 is shown here in the MIDI control window of Apple Logic software.

Read further:
Experiment with more MIDI features shown in Program Examples 13.2 and 13.3 in the textbook.

Digital audio processing?

- Digital audio processing, or more generally *digital signal processing* (DSP) refers to the computation of mathematically intensive algorithms applied to data signals

- 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 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.

An mbed DSP example

- It is possible to develop an mbed program that:
  - reads an analog audio signal in via the mbed ADC
  - processes the data digitally
  - then outputs the signal via the mbed DAC

- The analog output signal can be visualised on an oscilloscope or evaluated audibly by connecting it to a loudspeaker amplifier (such as a set of portable PC speakers).

- We need three audio files, one of a 200 Hz sine wave, a 1000 Hz sine wave and one of the two sine waves mixed together.

- 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.

An mbed DSP example

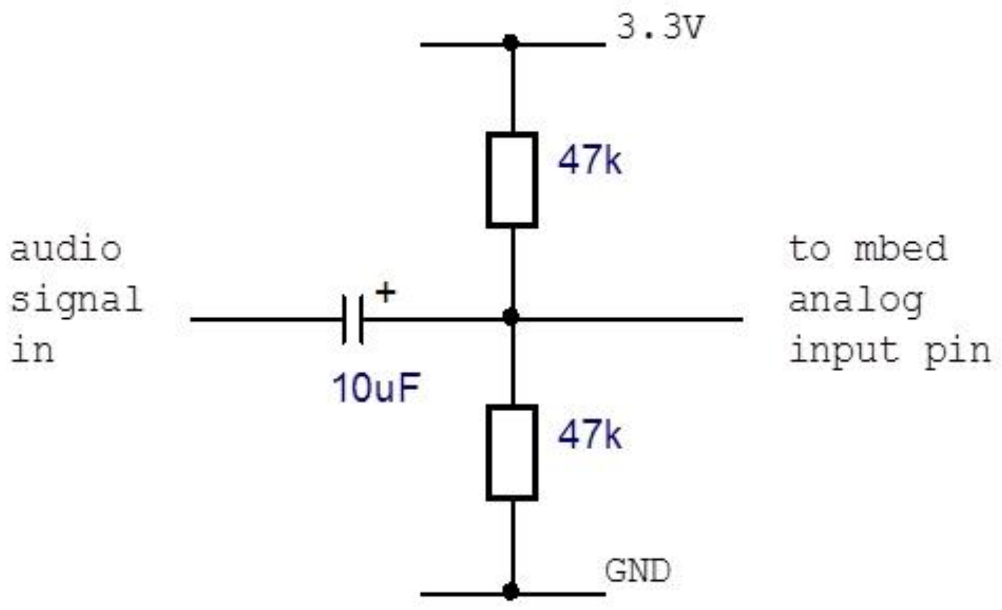
The audio files can be played directly from a host PC to an mbed analog input pin.

The signal can also be viewed on an oscilloscope, which shows 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.

It is therefore necessary to add a small coupling and biasing circuit to offset the signal to a midpoint of approximately 1.65 Volts.

The circuit shown effectively couples the host PC’s audio output to the mbed.

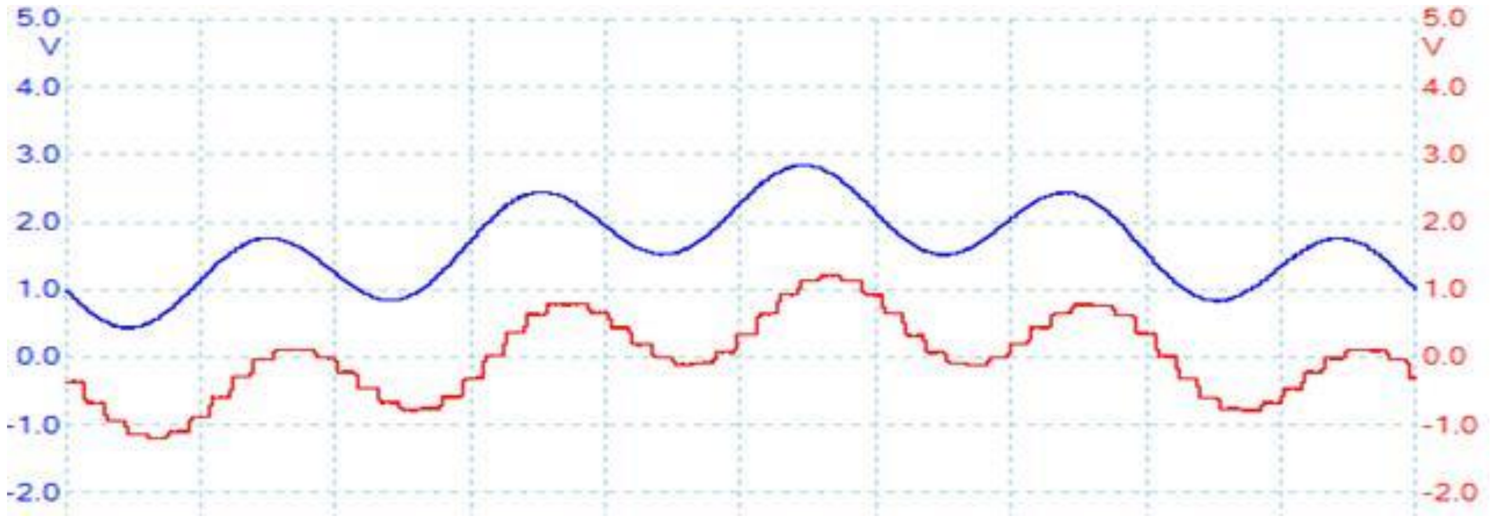


An mbed DSP example

```
/* Program Example 13.5 DSP input and output using a 20 kHz ticker object
                                                          */
#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;
}
```

Signal reconstruction

If you look closely at the audio signals input to and ouput from the mbed, 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.



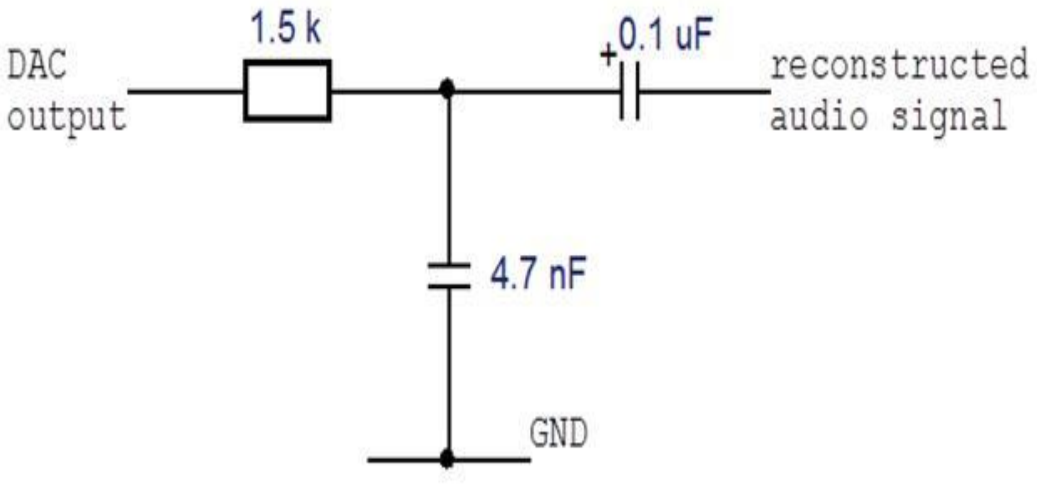
With many audio DSP systems, the analog output from the DAC is converted to a reconstructed signal by implementing an analog *reconstruction filter*, this removes all steps from the signal leaving a smooth output.

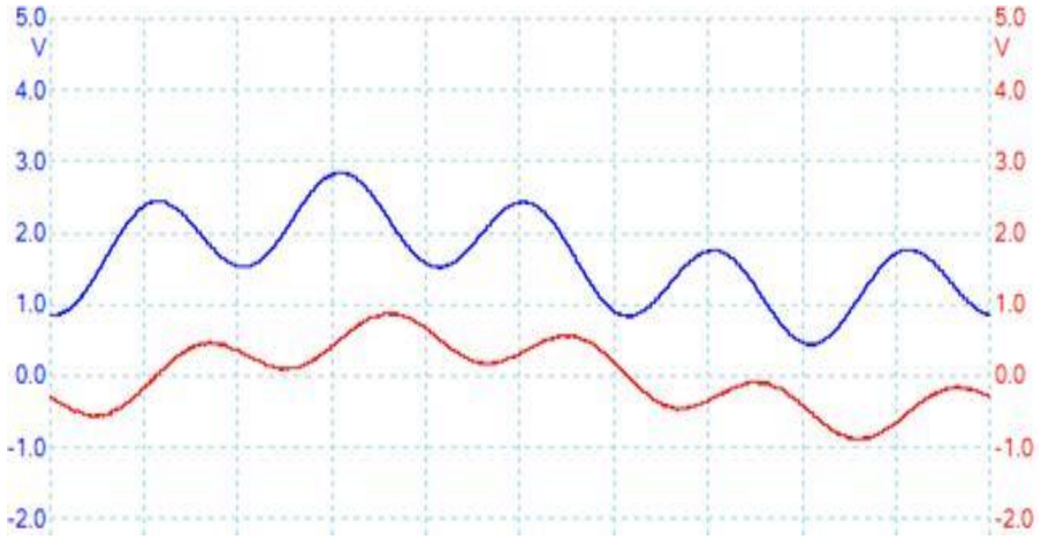
The reconstruction filter

In audio applications, a reconstruction filter is usually designed to be a low pass filter with a cut-off frequency at around 20 kHz, because the human hearing range doesn't exceed 20 kHz.

The reconstruction filter shown can be implemented with the current example to give a smooth output signal.

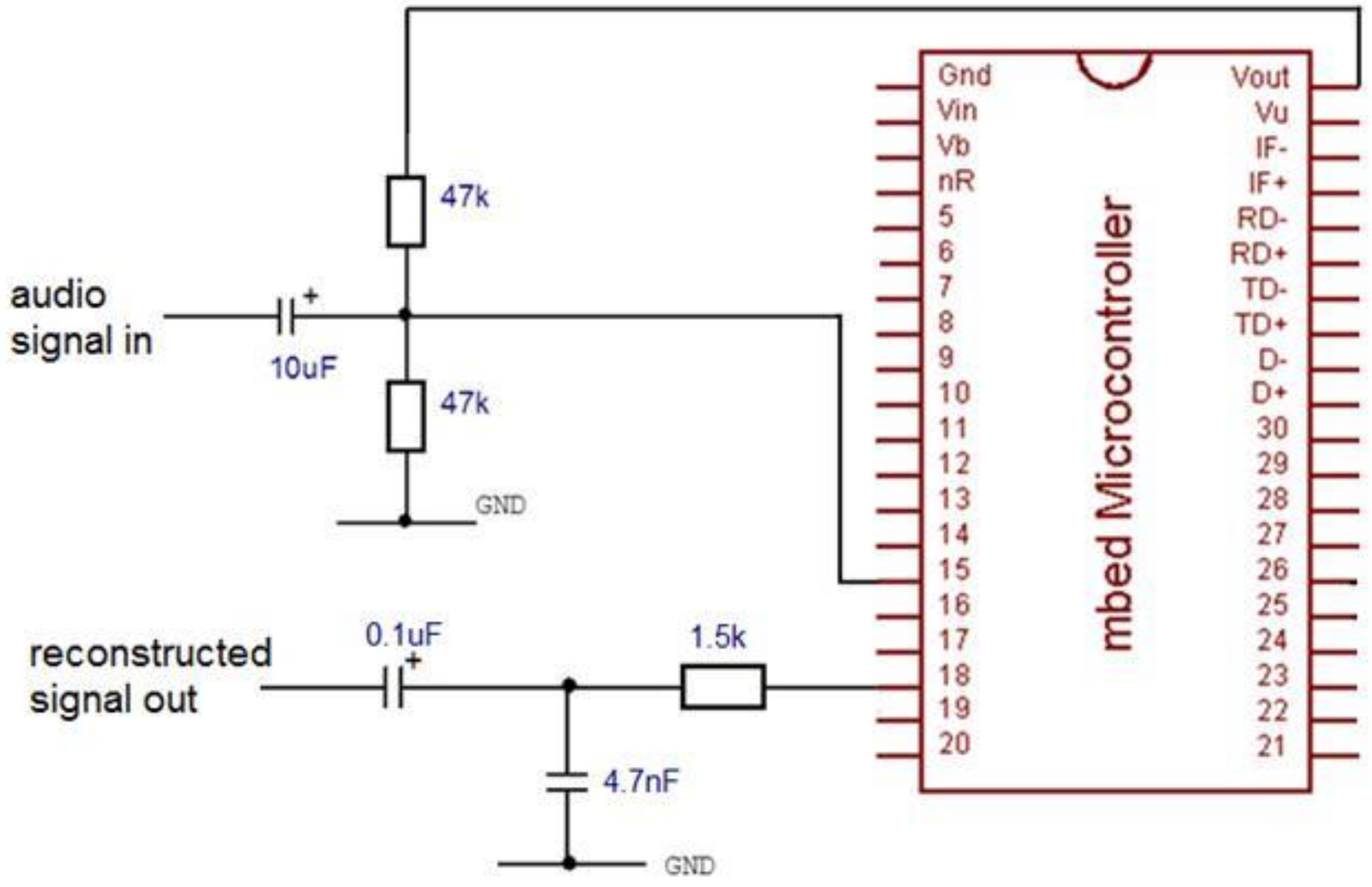
Note that after the low-pass filter, a *decoupling capacitor* is also added to remove the 1.65 V DC offset from the signal.





An mbed DSP example

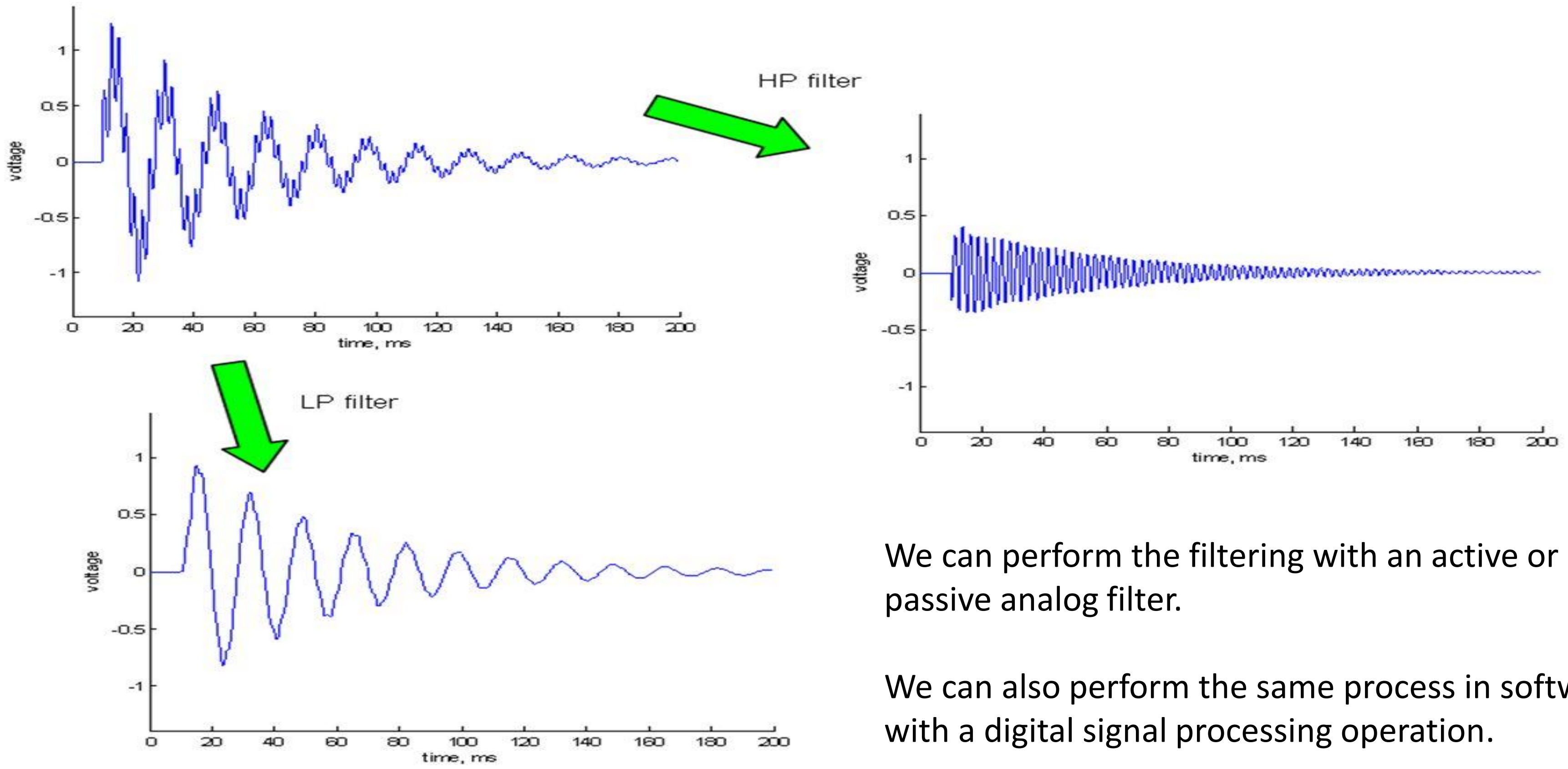
The complete circuit diagram for our mbed DSP example is therefore as follows:



Digital filtering example

Filters are used to remove chosen frequencies from a signal. The diagram shows 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 a *low-pass filter* (LPF).



We can perform the filtering with an active or passive analog filter.

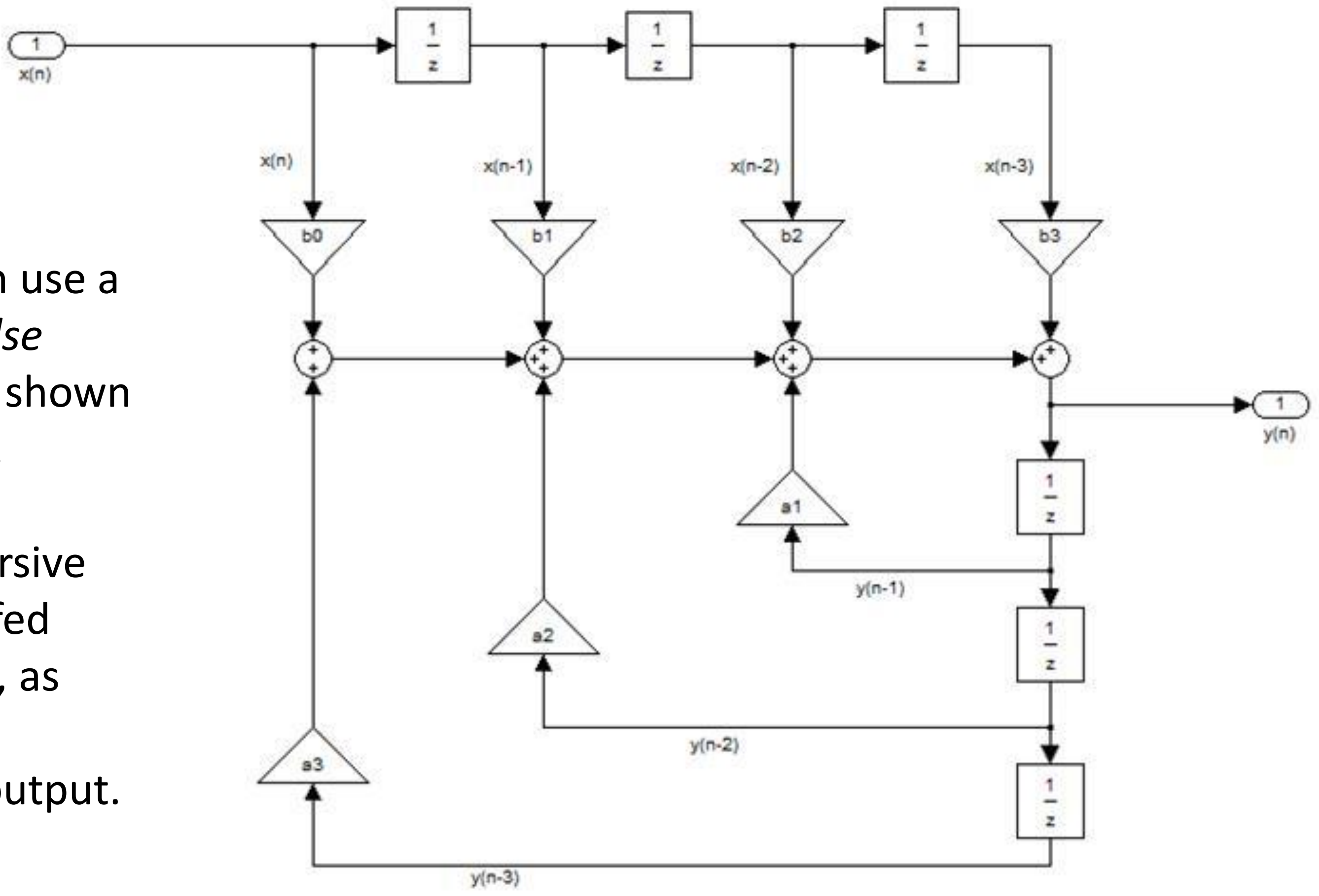
We can also perform the same process in software with a digital signal processing operation.

Adding a digital low-pass filter

We can 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.

In this example we can use a 3rd order *infinite impulse response* (IIR) filter, as shown in block diagram form.

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.



Adding a digital low-pass filter

The filter results in the following equation for calculating the filtered value given the current input, the previous 3 input values and the previous 3 output values:

$$y(n) = b_0 x(n) + b_1 x(n-1) + b_2 x(n-2) + b_3 x(n-3) +$$
$$+ a_1 y(n-1) + a_2 y(n-2) + a_3 y(n-3)$$

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,
- $a_{0\text{-}3}$ and $b_{0\text{-}3}$ are coefficients (filter taps) which 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 or online at

http://www-users.cs.york.ac.uk/~fisher/mkfilter/

Adding a digital low-pass filter

```
/*Program Example 13.6 Low pass filter function for 3rd order IIR with 20 kHz
sample frequency and 600 Hz cutoff frequency
                                               */
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
}
```

Here we can see the calculated filter values for the **a** and **b** coefficients.

Note that we have used the **static** variable type for the internally calculated data values, which allows specific values to be retained in the function during program execution.

Adding a digital low-pass filter (with pushbutton 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;
```

Notice that we *normalise* 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, with the line

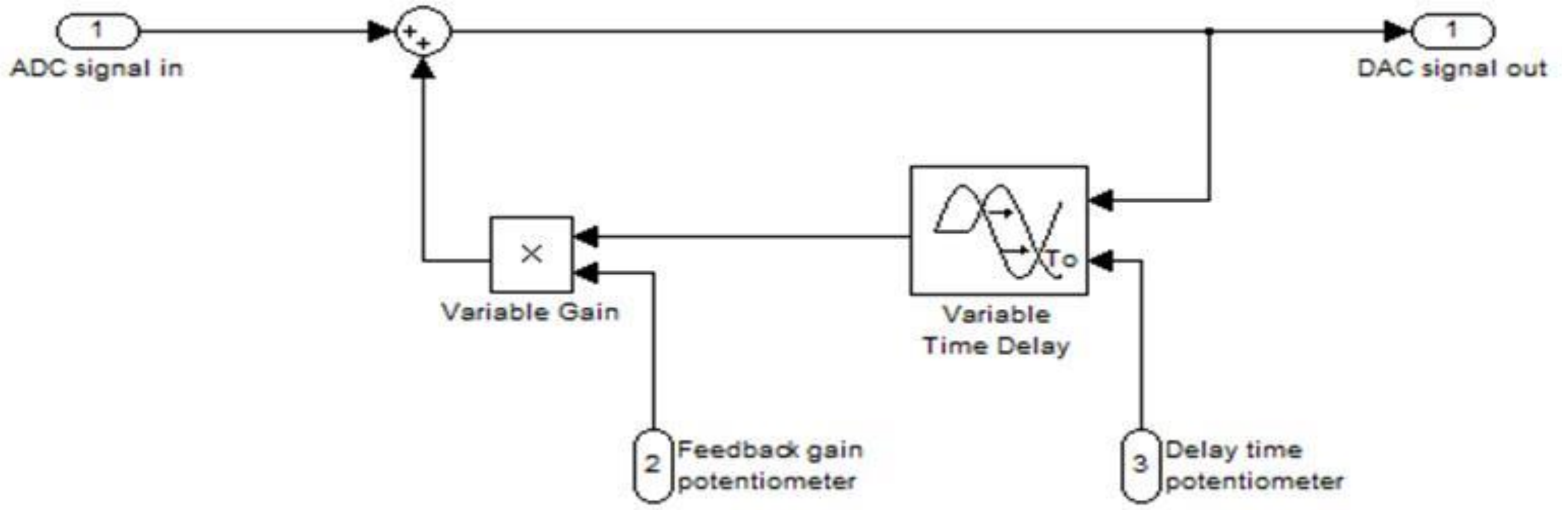
```
data_in=Ain-0.5;
```

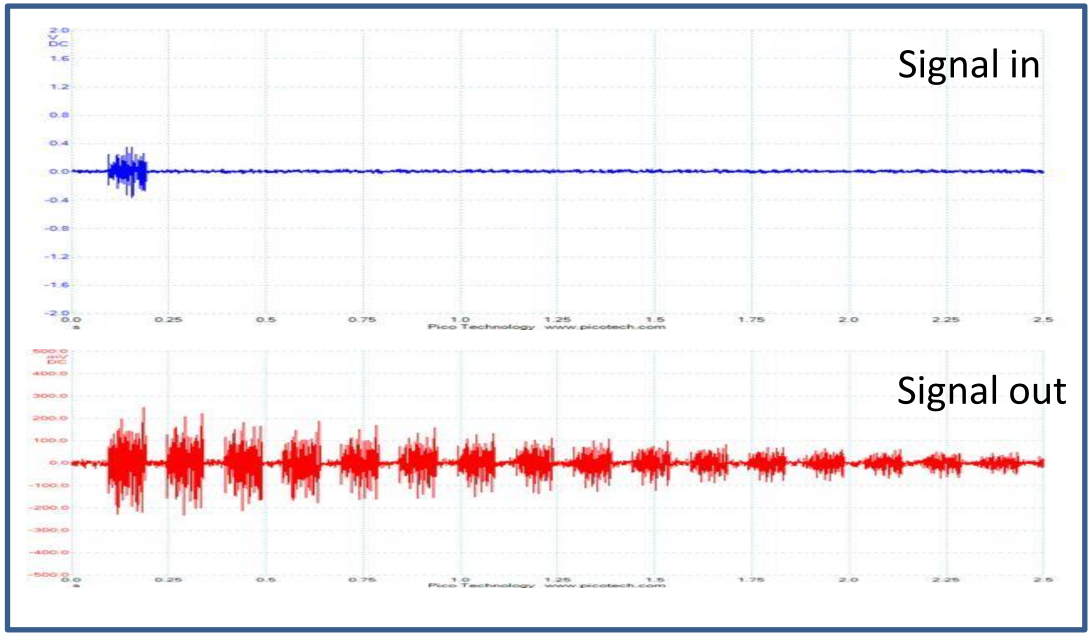
The mbed DAC anticipates floating point data in the range 0.0 to 1.0, so we also add the mean offset back to the data before we output, in the line

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

Read further: Digital high-pass filter example in Section 13.4.2

Audio delay and echo effect





Read further: Audio delay and echo effect project in Section 13.5

Working with wave audio files

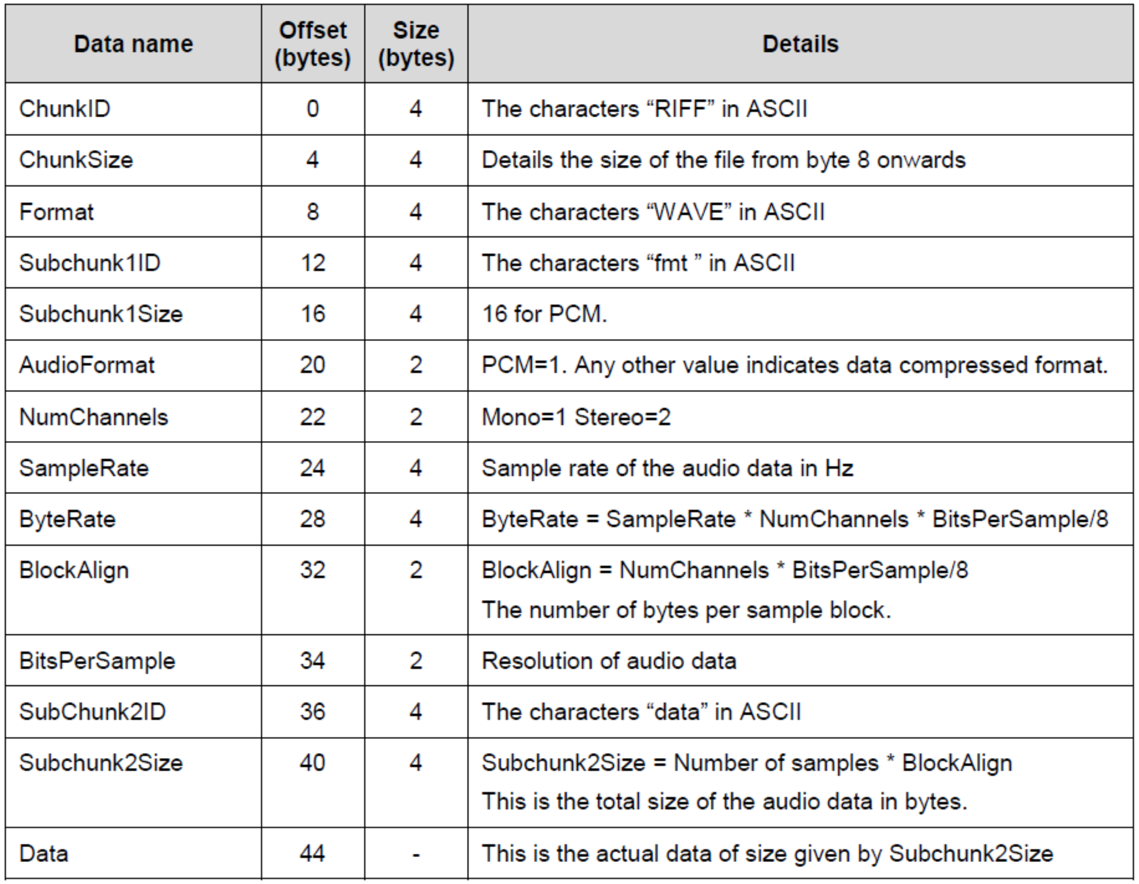
A number of signal processing applications rely on data which has previously been captured and stored in data memory.

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

There are many types of audio data file, of which the wave (.wav) type is one of the most widely used.

The wave header information details, for example, the resolution and sample frequency of the audio, as shown.

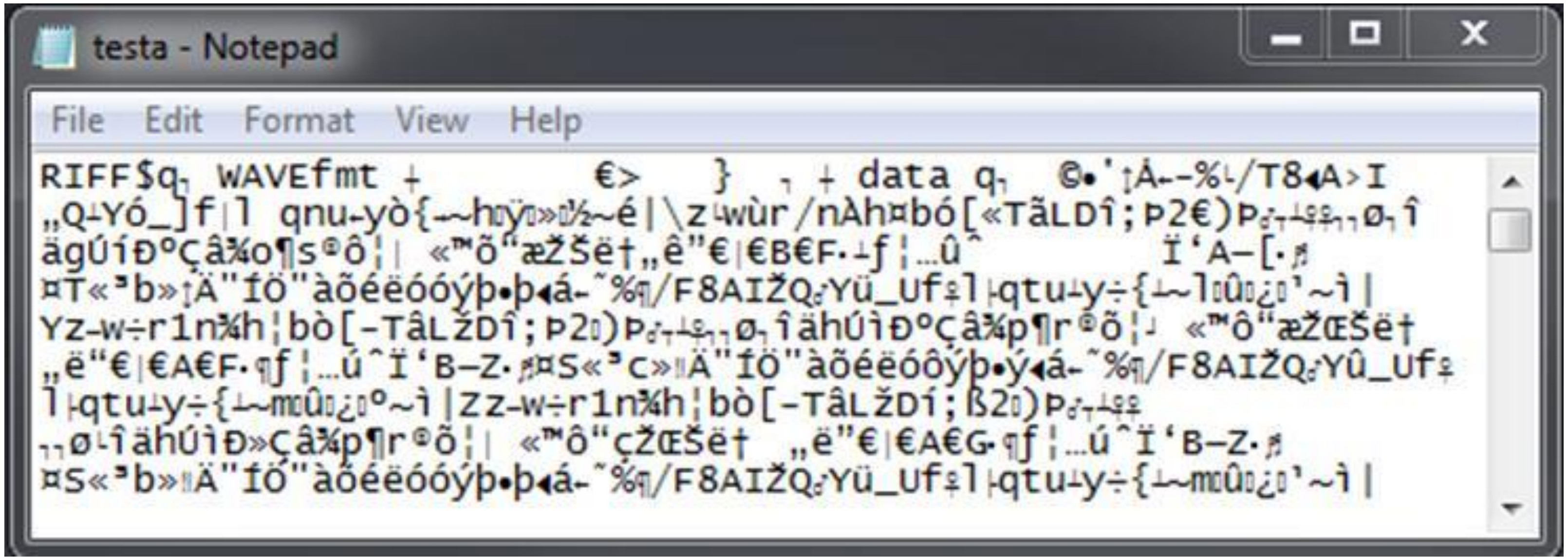
By reading this header it is possible to accurately decode and process the contained audio data.



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

Working with wave audio files

It is possible to identify much of the wave header information simply by opening a .wav file with a text editor application, as shown.

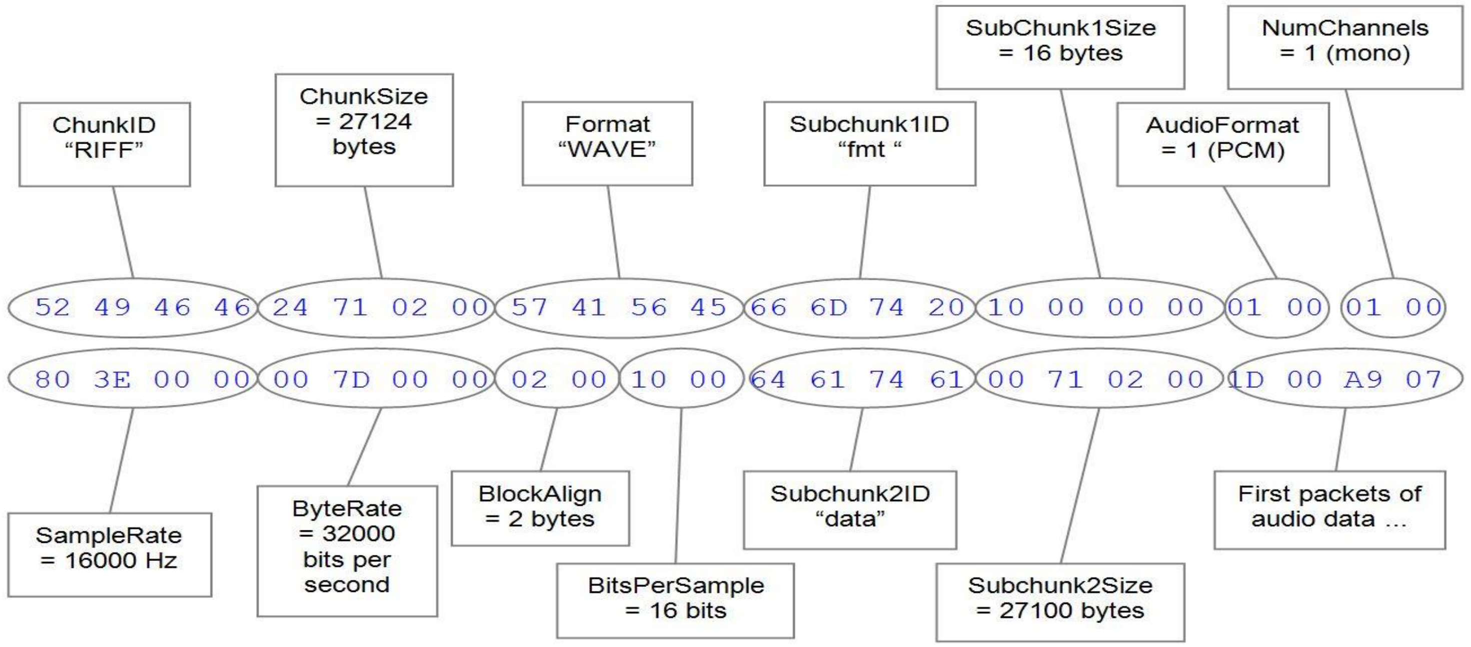


```
RIFF$q┐ WAVEfmt ┼         €>    }  ┐ ┼ data q┐   ©•'↕Å←-%└/T8◄A>I
„Q┴Yó_]f│l qnu┬yò{┴~hüÿü»ü½~é|\z└wùr/nÀh¤bó[«TãLDî;Þ2€)Þ♂┬┴♀♀┐┐ø┐î
ägÚíÐºÇâ%o¶s®ô¦| «™õ"æŽŠë†„ê"€|€B€F·┴ƒ¦…û^          Ï'A–[·♬
¤T«³b»↕Ä"ÍÖ"àõéëóóýþ•þ◄á←˜%¶/F8AIŽQ♂Yü_Uf♀l├qtu┴y÷{┴~lüûü¿ü'~ì|
Yz-w÷r1n%h¦bò[-TâLžDí;Þ2ü)Þ♂┬┴♀┐┐ø┐îähÚìÐºÇâ%p¶r®õ¦┘ «™ô"æŽŒŠë†
„ë"€|€A€F·¶ƒ¦…ú^Ï'B–Z·♬¤S«³c»‼Ä"ÍÖ"àõéëóôýþ•ý◄á←˜%¶/F8AIŽQ♂Yû_Uf♀
l├qtu┴y÷{┴~müûü¿üº~ì|Zz-w÷r1n%h¦bò[-TâLžDí;ß2ü)Þ♂┬┴♀♀
┐┐ø└îähÚìÐ»Çâ%p¶r®õ¦| «™ô"çŽŒŠë†  „ë"€|€A€G·¶ƒ¦…ú^Ï'B–Z·♬
¤S«³b»‼Ä"ÍÖ"àõéëóóýþ•þ◄á←˜%¶/F8AIŽQ♂Yü_Uf♀l├qtu┴y÷{┴~müûü¿ü'~ì|
```

Here we see (the ASCII characters for ChunkID ("RIFF"), Format ("WAVE"), Subchunk1ID ("fmt") and Subchunk2ID ("data"). These are followed by the ASCII data which makes up the raw audio.

Working with wave audio files

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.


ChunkID "RIFF"
ChunkSize = 27124 bytes
Format "WAVE"
Subchunk1ID "fmt "
SubChunk1Size = 16 bytes
AudioFormat = 1 (PCM)
NumChannels = 1 (mono)
52 49 46 46 24 71 02 00 57 41 56 45 66 6D 74 20 10 00 00 00 01 00 01 00
80 3E 00 00 00 7D 00 00 02 00 10 00 64 61 74 61 00 71 02 00 1D 00 A9 07
SampleRate = 16000 Hz
ByteRate = 32000 bits per second
BlockAlign = 2 bytes
BitsPerSample = 16 bits
Subchunk2ID "data"
Subchunk2Size = 27100 bytes
First packets of audio data ...

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.

Reading the wave file header with the mbed

```
/* Program Example 13.8 Wave file header reader (wav data stored on SD card)
                                                                              */
#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");                  // open SD card wav file
    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);
}
```

Reading and outputting mono wave data

For this example we will use a 16-bit mono wave file of a pure sine wave of 200 Hz.

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.

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.

We therefore need to use a buffered system to enable accurate timing control.

Reading and outputting mono wave data

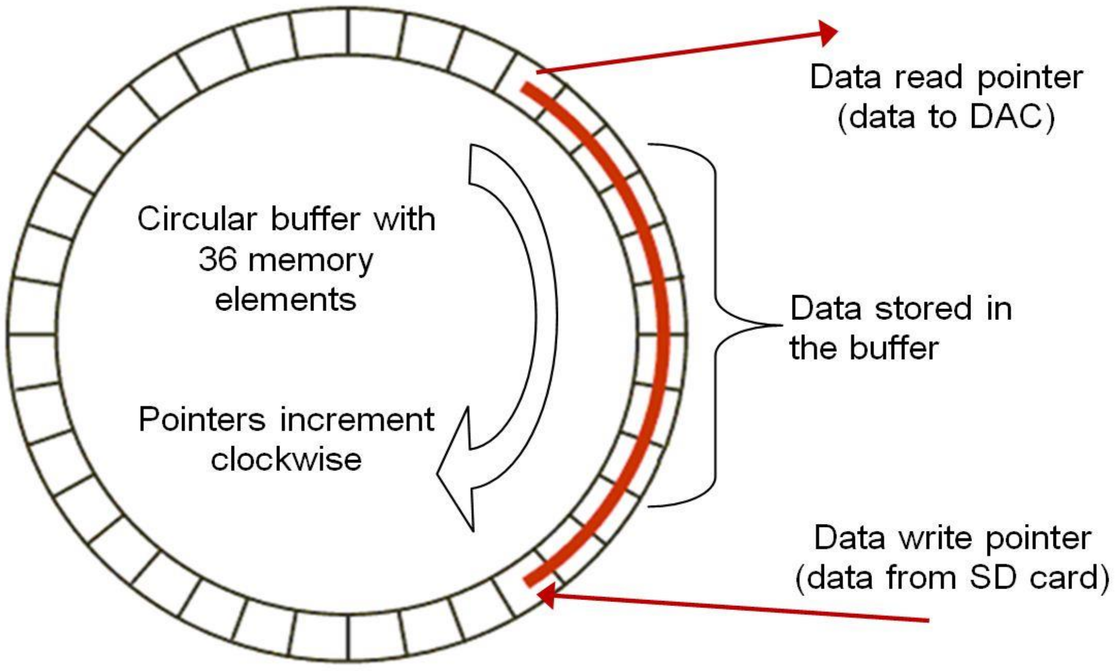
A good method to ensure accurate timing control when working with audio data is to use a *circular buffer*.

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.


Data read pointer (data to DAC)
Circular buffer with 36 memory elements
Pointers increment clockwise
Data stored in the buffer
Data write pointer (data from SD card)

Reading and outputting mono wave data

```
/* Program Example 13.9: 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);
}
```

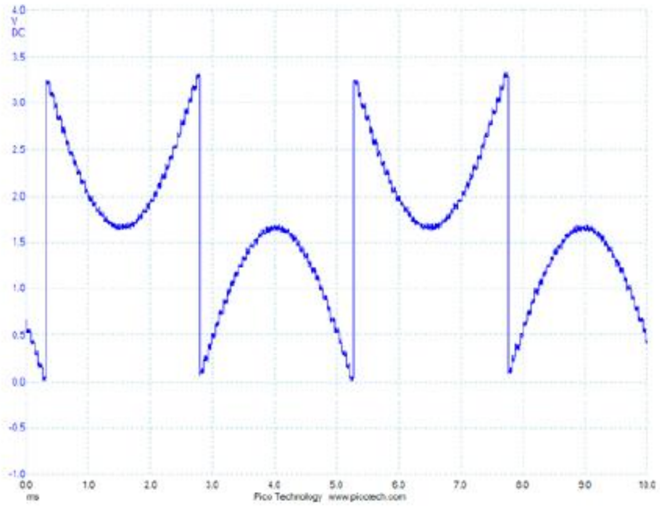
Reading and outputting mono wave data

```
// Program Example 13.9 Continued

// 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
    }
  }
}
```

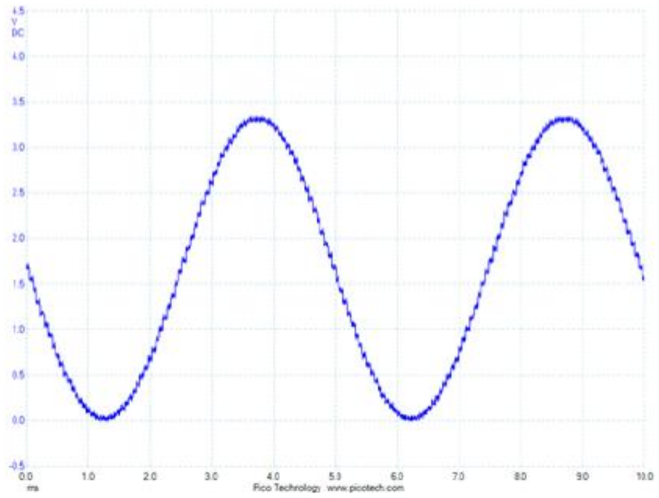
Reading and outputting mono wave data

When Program Example 13.9 is implemented with a pure mono sine wave audio file, initially the results will not be correct. Figure a below 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. A simple adjustment of the data is therefore required in order to output the correct waveform as shown in Figure b.



a: raw data output



b: two's-compliment corrected output

Read further:
Two's complement arithmetic covered in Appendix A

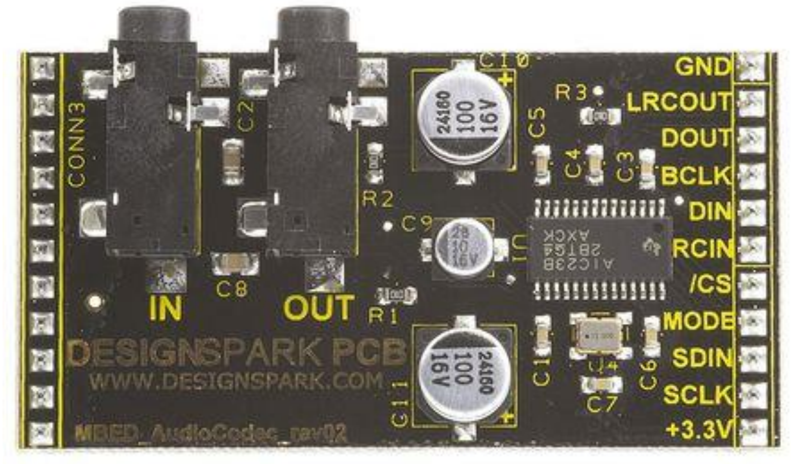
High-fidelity digital audio with the mbed

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.

The Texas Instruments TLV320AIC23B chip is a high-performance stereo audio convertor with integrated analog functionality.

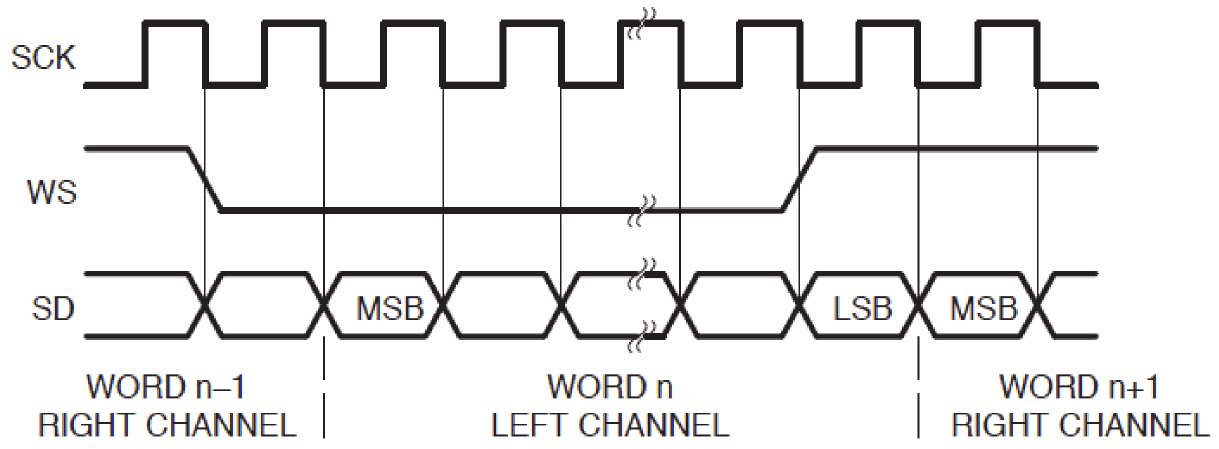
The TLV320's ADCs and DACs can process 2 channels of 16, 20, 24 or 32 bit data at sample rates up to 96 kHz.



Conveniently, the Synergy AudioCODEC by DesignSpark is designed to easily interface the TLV320 to the mbed, with PCB mounted 3.5mm stereo audio sockets and on-board analog signal conditioning

The Inter-IC Sound (I^{2}S) serial protocol

- The TLV320 uses a serial digital audio protocol called *I^{2}S* (*Inter-IC Sound*) to communicate with a microcontroller. The I^{2}S protocol uses three signal lines:
    - continuous serial clock (SCK)
    - word select (WS)
    - 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).



Read further:
mbed I^{2}S connection to the Synergy AudioCODEC is shown in Table 13.3

Outputting audio data from the TLV320

```
// 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
```

Read further: Description of Program Example 13.10 in textbook Section 13.7.2

Outputting audio data from the TLV320

```
// 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
}
```

Outputting audio data from the TLV320

```
// Program Example 13.12 Function to load square wave 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
  }
}
```

You can combine program examples 13.10, 13.11 and 13.11 to generate a high fidelity square wave from AudioCODEC. The frequency of the square wave will depend on the value of **halfperiod**. The square wave frequency in Hz is given by

$$square\ wave\ frequency = \frac{audio\ sample\ rate}{2 \times halfperiod}$$

Read further:

The textbook covers further examples of high fidelity digital audio, including:

- High-fidelity wave file player (Section 13.7.3)
- High-fidelity audio recording (Section 13.7.4)

Chapter quiz questions

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 anti-aliasing 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 I^2S communications protocol and how it is utilised 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?

Chapter review

- The MIDI protocol allows music systems to be connected together and communicate musical data, such as pitch and velocity
- 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 audio.
- A DSP system communicates with the external world through analog-to-digital convertors and digital-to-analog convertors, 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 in order 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.