

AES Cambridge Seminar Series  
27 October 2010

## Audio Signal Processing and Rapid Prototyping with the ARM mbed

Dr Rob Toulson

Director of The Sound and Audio Engineering Research Group  
Anglia Ruskin University, Cambridge

[rob.toulson@anglia.ac.uk](mailto:rob.toulson@anglia.ac.uk)

# Contents

- Introducing the ARM mbed
- Audio filtering example with the ARM mbed
  - Analogue signal input
  - Simple input/output code
  - Signal reconstruction
  - Adding a digital low-pass filter
  - Filter coefficients
  - Filter Implementation
  - Assigning the filters to pushbuttons
- Delay / echo effect
  - Delay effect code
  - Delay effect waveforms
- Images
- References

# Introducing the ARM mbed

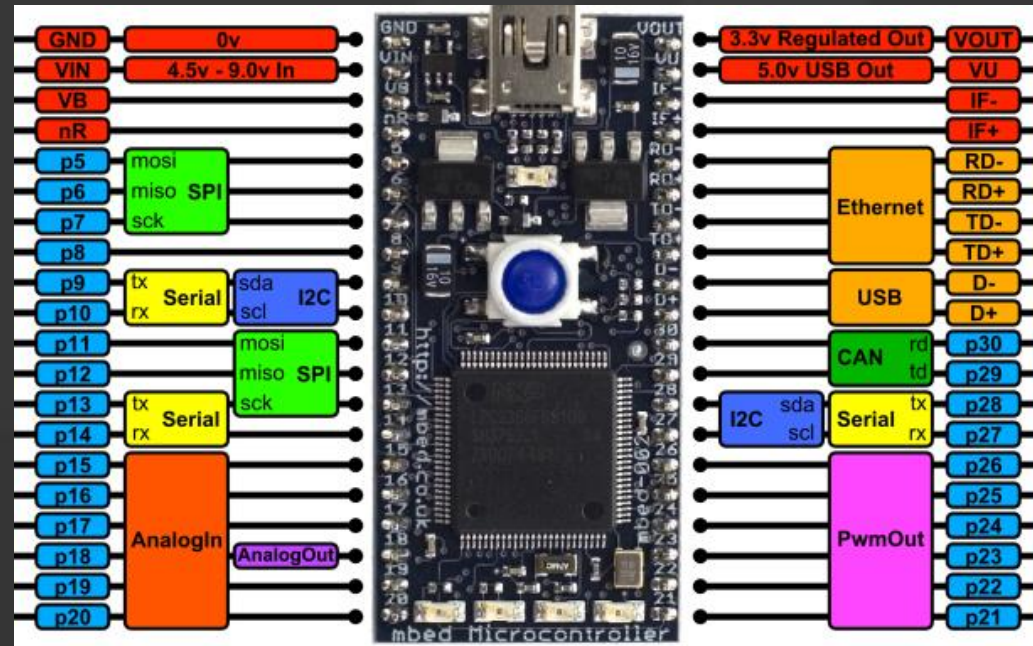
The mbed Microcontroller is a 32-bit ARM processor with a comprehensive set of peripherals and a built-in USB interface, all provided in a small and practical DIP package. It is designed specifically to make ARM microcontrollers easily accessible for rapid prototyping and experimentation.



The mbed Microcontroller is based on the NXP LPC1768 with an ARM Cortex-M3 Core running at 96MHz, 512KB FLASH, 64KB RAM and lots of interfaces including Ethernet, USB Device and Host, CAN, SPI, I2C and other I/O.

Cost £39.00 from Farnell

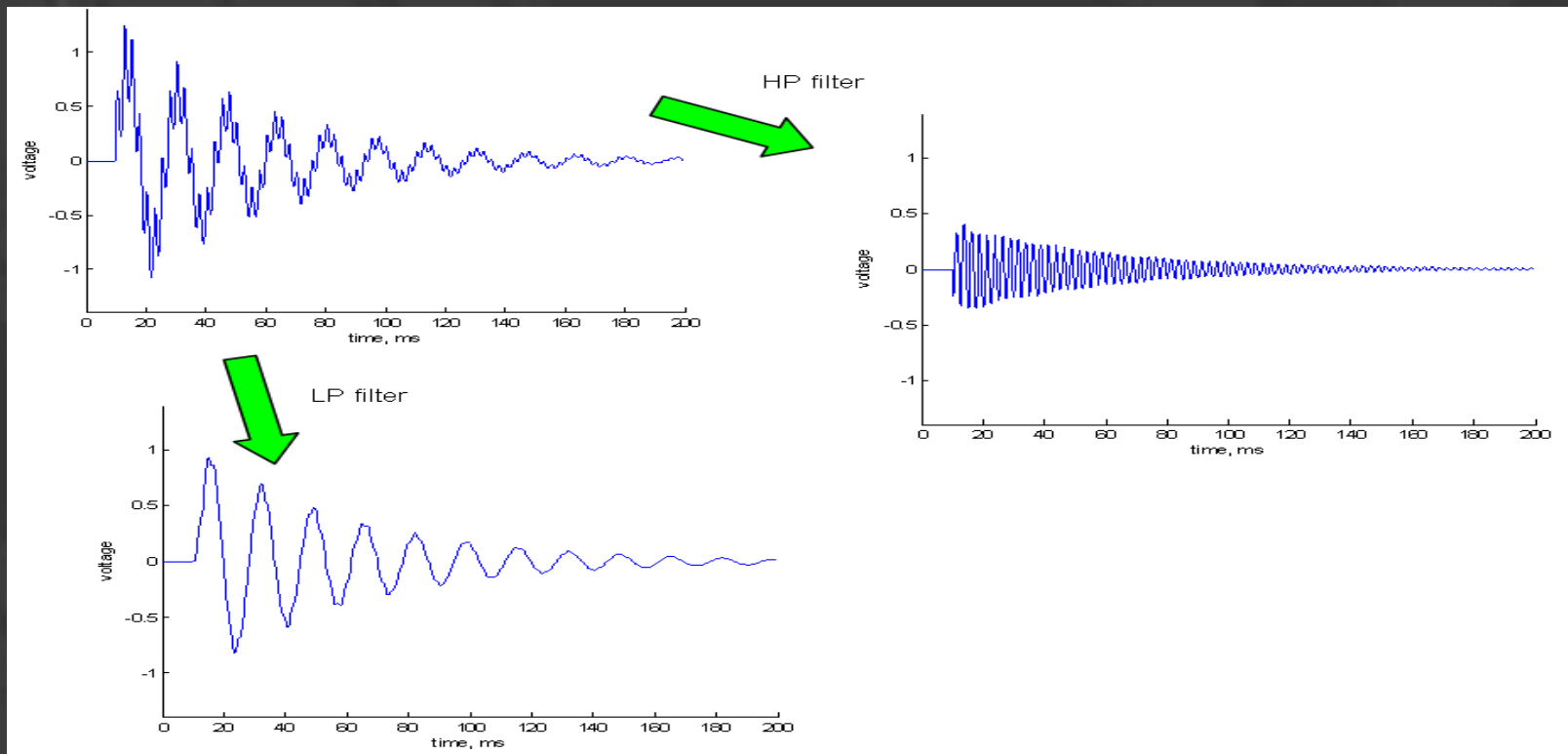
See [www.mbed.org](http://www.mbed.org)



# Audio filtering example with the mbed

We can use the mbed's built in ADC and DAC to process audio in real-time. So we sample audio, process it in software and then output the processed analogue signal.

This example will only use 12-bit AD conversion and run at 20kHz, but it is possible to use an external ADC/DAC chip to process data at higher rates and resolution



# Analogue signal input

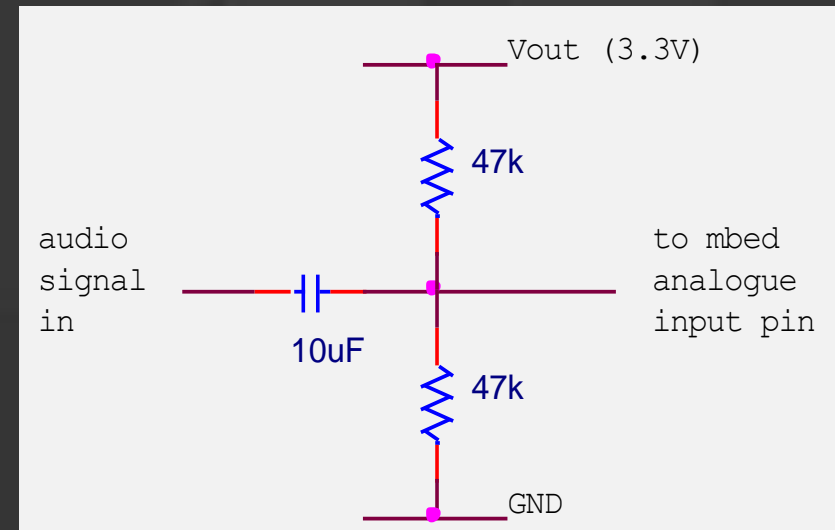
First we need a signal source to input to the mbed. A simple way to create a signal source is to use a host PC's audio output while playing an audio file of the desired signal data. Here we will use three test audio files as follows:

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

We can also analyse the source signal with a scope. The signal oscillates positive and negative about 0 Volts, which isn't much use for the mbed, as we can only read analogue data between 0-3 Volts on the mbed

We therefore need to add a small coupling and biasing circuit to achieve a signal midpoint of approximately 1.65 Volts.

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



# Simple input/output code

First we can write mbed code in C++ to sample and output the audio data with no processing at all – this will show us that the system is working. The mbed compiler allows us to use pre-written library functions to simplify the hardware setup and code design. Here we will define

- an analogue input on pin 20
- an analogue output on pin 18
- A ticker feature to call a regularly repeating function

```
#include "mbed.h" // include mbed library header
AnalogIn Ain(p20); // analogue input on pin 20
AnalogOut Aout(p18); // analogue output on pin 18
Ticker s20khz_tick; // ticker feature to call a regularly repeating
function

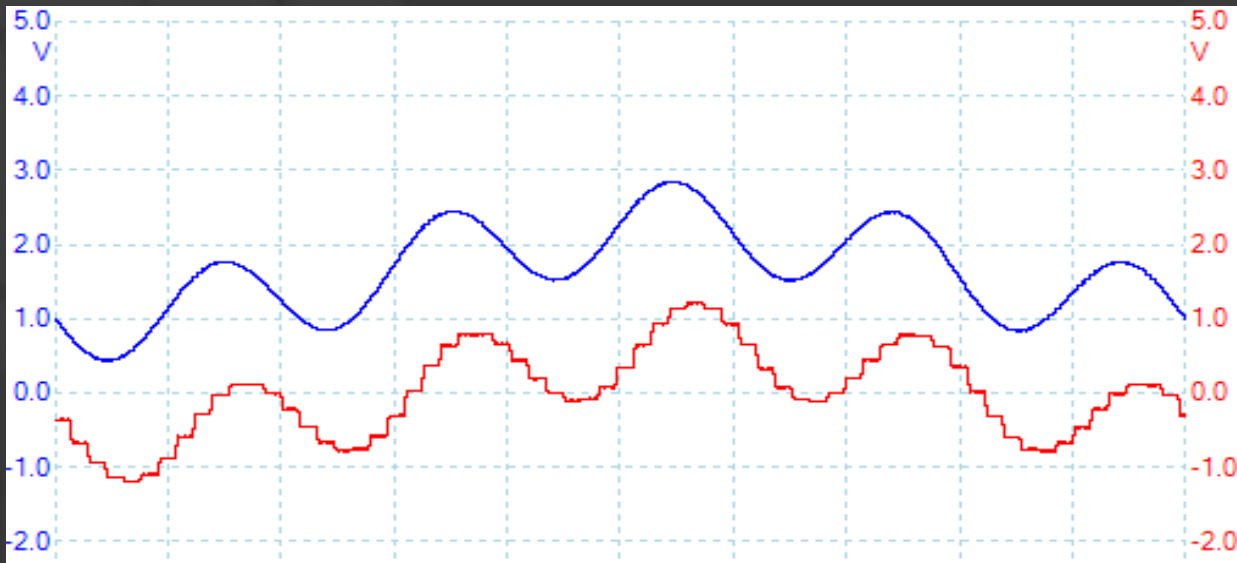
void s20khz_task(void); // function prototype definition
float data_in, data_out; // floating point variable declarations

int main() { // start main software function
    s20khz_tick.attach_us(&s20khz_task,50); // attach 20khz task to 50us tick
}

void s20khz_task(void){ // task function running 20,000 times per second
    data_in=Ain; // variable data_in=Analogue input value (pin 20)
    data_out=data_in; // set variable data_out=data_in
    Aout=data_out; // set analogue output (pin 18) to value data_out
}
```

# 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.

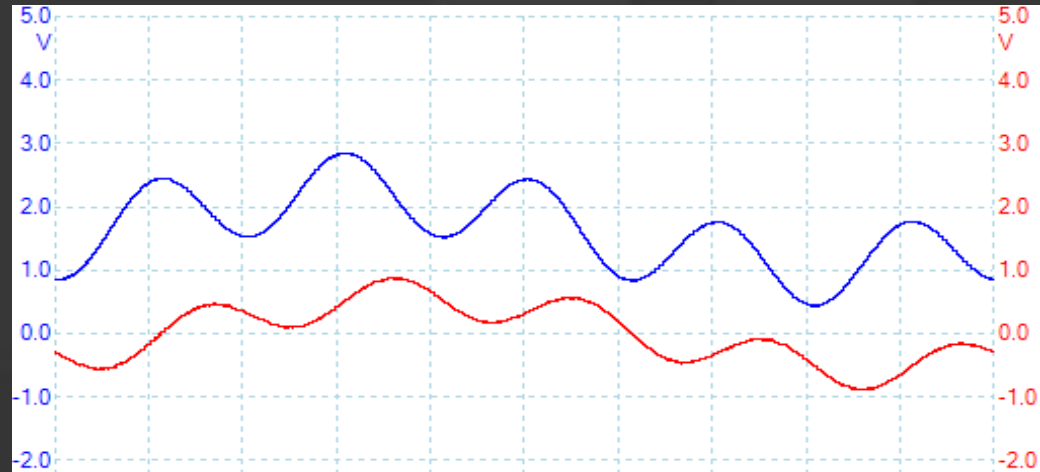
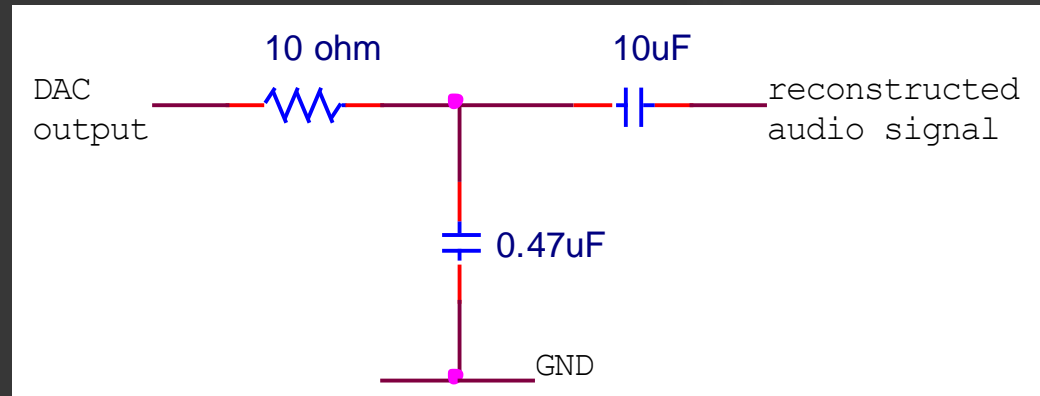


The discrete audio data needs to be converted to a 'reconstructed' audio signal by implementing an analogue 'reconstruction filter' straight after the DAC, this removes all steps from the signal. In audio applications, our reconstruction filter is usually designed to be a low pass filter with a cut-off at 20 kHz, this is because the human audible range doesn't exceed 20 kHz. We therefore add a reconstruction filter to smooth the DAC output.

# Signal reconstruction

The reconstruction filter shown below can be implemented with the current project to smooth the DAC output.

After the low-pass filter, 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 or set of headphones to monitor the processed DAC output.



Note: the mathematical theory associated with digital sampling and reconstruction is complex and beyond the scope of this demonstration. For example, in audio sampling systems it is necessary to add a further 'anti-aliasing' filter prior to the analogue to digital-conversion. For simplicity we have not implemented this extra filter, for those interested the theory of sampling, aliasing and reconstruction is described well and in detail by a number of authors including Marven and Ewers (1996) and Proakis and Manolakis (1992).

# Adding a digital low pass filter

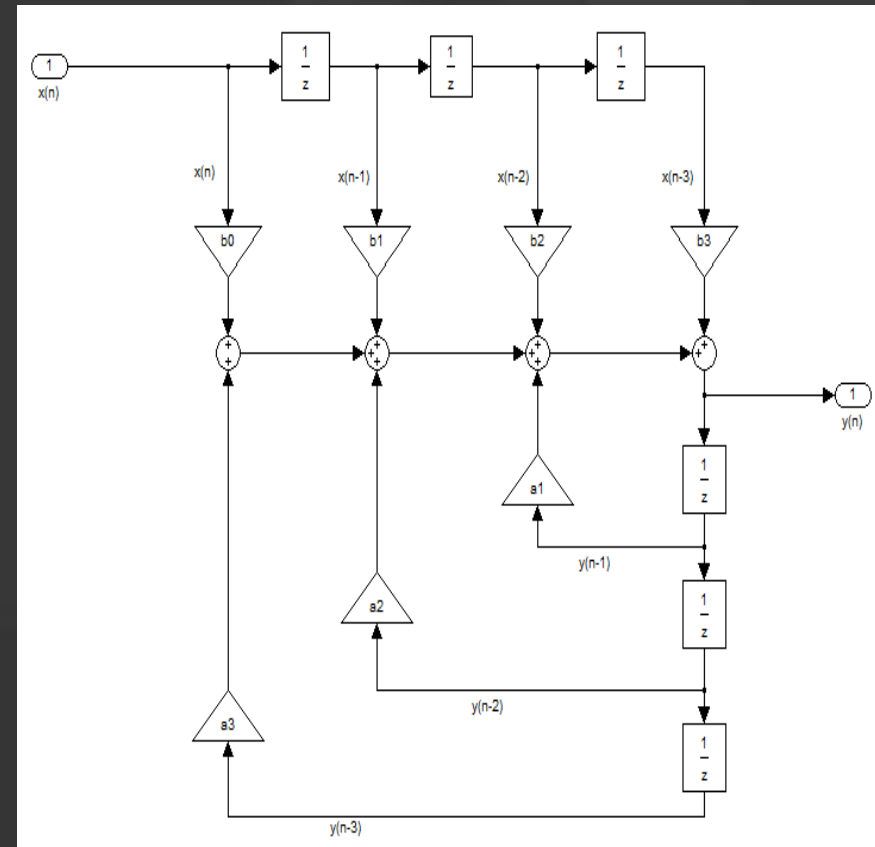
We will add a digital low pass filter routine to filter out the 1000 Hz frequency component. This can be assigned to a switch input so that the filter is implemented in real-time when a push button is pressed. In this example we will use a 3<sup>rd</sup> order Infinite Impulse Response (IIR) filter.

This 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:

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

Where:

$x(n)$  is the current data input value  
 $x(n-1)$  is the previous data input value (applies similarly for  $n-2$ ,  $n-3$  etc)  
 $y(n)$  is the calculated current output value  
 $y(n-1)$  is the previous data output value (applies similarly for  $n-2$ ,  $n-3$  etc)  
 $a_{0-3}$  and  $b_{0-3}$  are coefficients which define the filter's performance



# Digital filter coefficients

We can therefore implement this equation to calculate 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 and online calculation routines, for example that provide by Fisher (2010) or with the Matlab filter design and analysis tool (Mathworks, 2010)

The screenshot displays the Filter Design & Analysis Tool (FDA) interface. The main window shows the 'Current Filter Information' panel on the left, which includes details such as Structure (Direct-Form II, Second-Order Sections), Order (3), Sections (2), Stable (Yes), and Source (Designed). The central plot shows the Magnitude Response (dB) versus Frequency (kHz), illustrating a low-pass filter characteristic. The bottom panel contains design specifications, including Response Type (Lowpass), Filter Order (Specify order: 3), Frequency Specifications (Fs: 20000 Hz, Fc: 600 Hz), and Magnitude Specifications (The attenuation at cutoff frequencies is fixed at 3 dB). A 'Design Filter' button is visible at the bottom.

Overlaid on the right side of the screenshot is a 'Filter Coefficients' window. It displays the following coefficients:

```
Numerator:  
0.00069934964990100491  
0.0020980489497030149  
0.0020980489497030149  
0.00069934964990100491  
Denominator:  
1  
-2.6235518066052381  
2.3146825810891105  
-0.68553597728466409
```

# Filter implementation

The low pass filter designed with a Butterworth algorithm for a 600 Hz cut-off frequency can be implemented as a C function as follows:

```
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  
    y[3] = y[2]; y[2] = y[1]; y[1] = y[0];           // move y values by one  
  
    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  
}
```

A high pass filter can be designed in a similar way and implemented with just the coefficients changed as follows:

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

# Assigning the filters to pushbuttons

You can now assign an if statement to a digital input, allowing the filters to be switched in and out in real time. So we attach digital switches to pins 21 and 22 and define a DigitalIn objects as follows:

```
DigitalIn LPFswitch(p21);  
DigitalIn HPFswitch(p22);
```

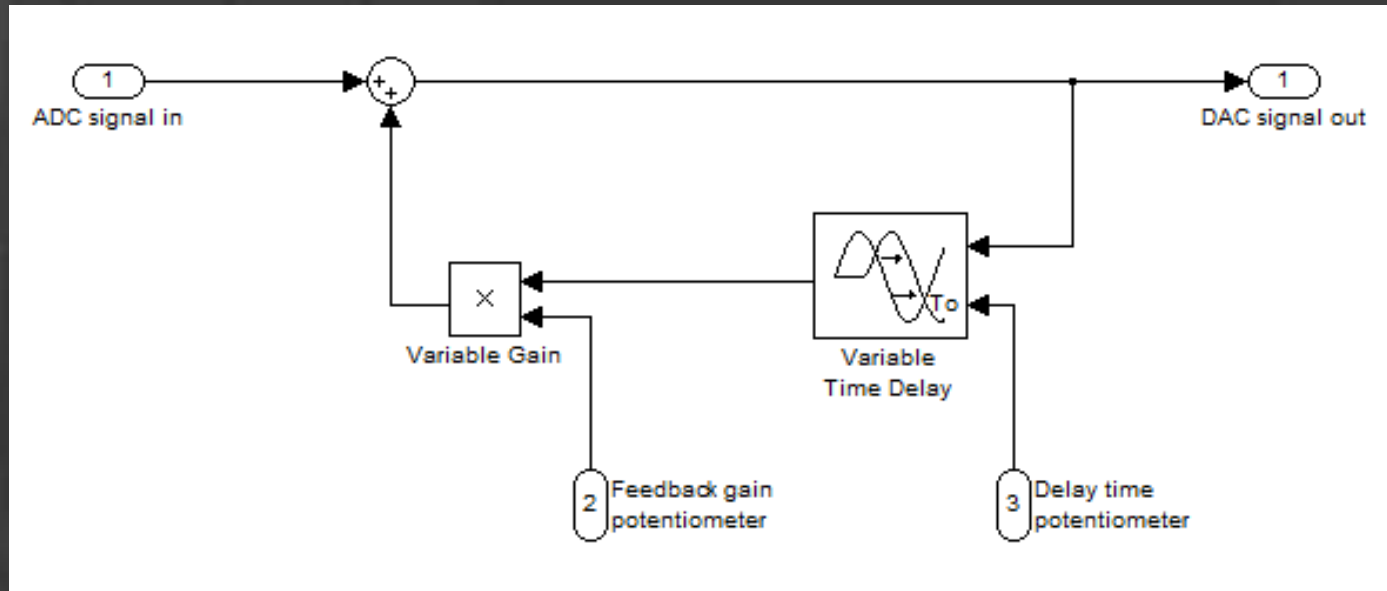
We can now use conditional statements to activate the HPF or LPF when pushbuttons are pressed, as shown.

Before performing the calculation, the mean value of the signal is subtracted ( $\text{data\_in} = \text{Ain} - 0.5$ ). This is to normalise the signal about 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 output is floating point data in the range 0.0-1.0, we must also add the mean (halfway) offset back to the data before we output to the DAC ( $\text{Aout} = \text{data\_out} + 0.5$ ).

```
data_in=Ain-0.5;  
if (LPFswitch==1){  
    data_1=LPF(data_in);  
}  
else {  
    data_1=data_in;  
}  
  
if (HPFswitch==1){  
    data_out=HPF(data_1);  
}  
else {  
    data_out=data_1;  
}  
Aout=data_out+0.5;  
}
```

# Delay / echo effect

We can design a feedback delay which has variable delay time and variable feedback gain



We need to load digital data into a buffer (a large array) and then mix this stored data with the immediate data.

The feedback gain determines how much buffer data is mixed with the sampled data.

The delay time variable resets the buffer counter after a small or large number of iterations.

# Delay effect code

```
#include "mbed.h"

AnalogIn Ain(p20);
AnalogOut Aout(p18);
AnalogIn speed_pot(p16);
AnalogIn feedback_pot(p17);
Ticker s20khz_tick;

void s20khz_task(void); //function prototypes

#define MAX_BUFFER 14000 // max data samples

//variables and data
signed short data_in;
unsigned short data_out;
float speed=0;
float feedback=0;
signed short buffer[MAX_BUFFER]={0};
int i=0;

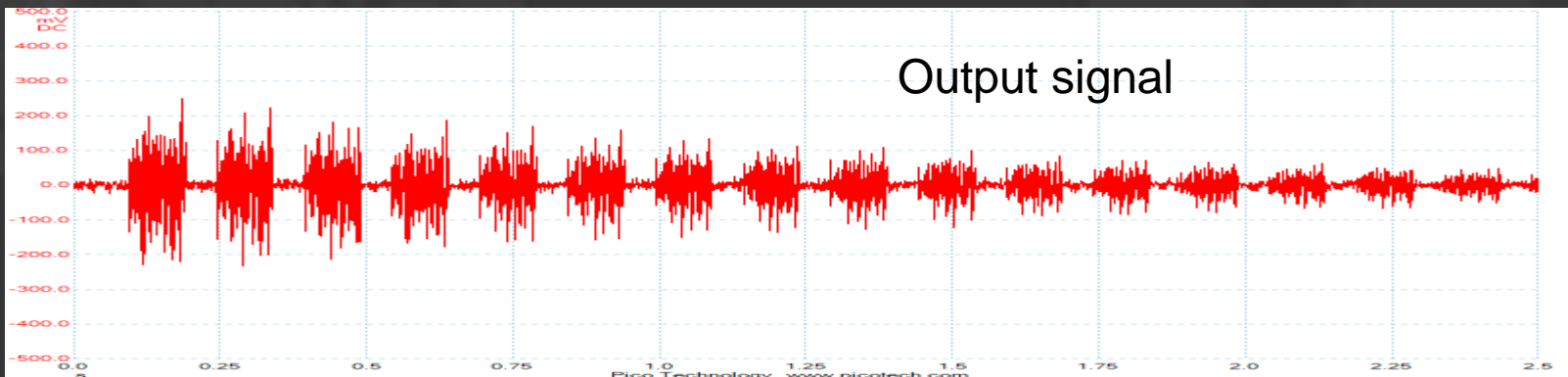
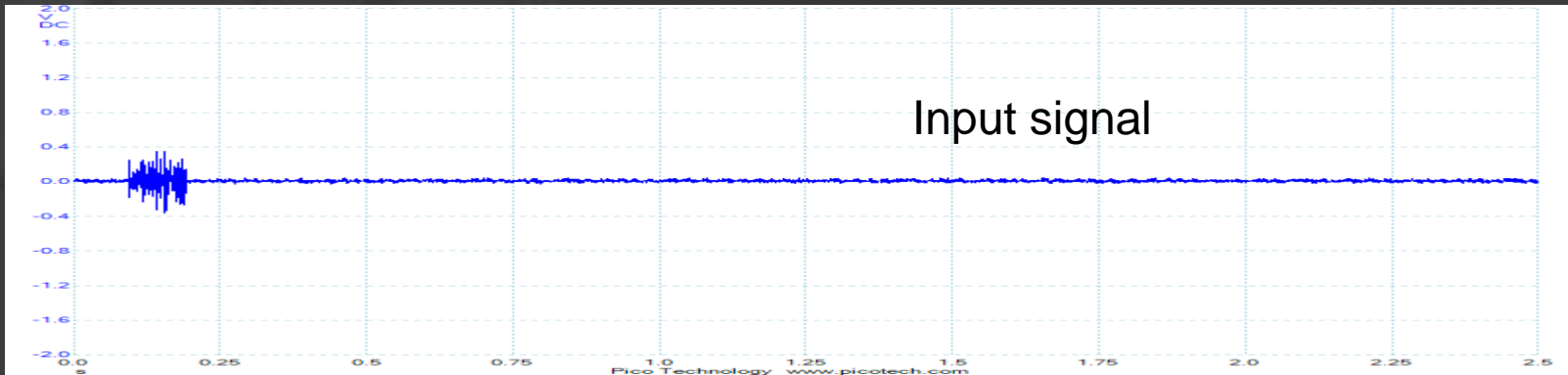
//main program start here
int main() {
    s20khz_tick.attach_us(&s20khz_task,50);
}
```

```
// function 20khz_task
void s20khz_task(void){

    data_in=Ain.read_ul6()-0x7FFF;
    buffer[i]=data_in+(buffer[i]*feedback);
    data_out=buffer[i]+0x7FFF;
    Aout.write_ul6(data_out);

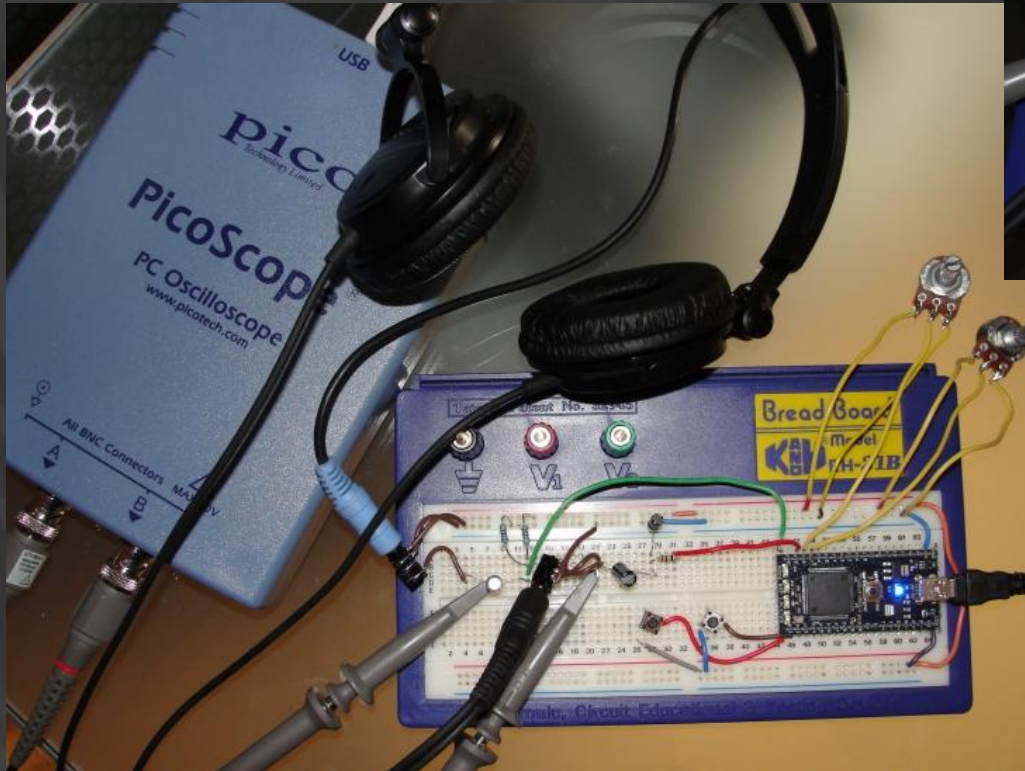
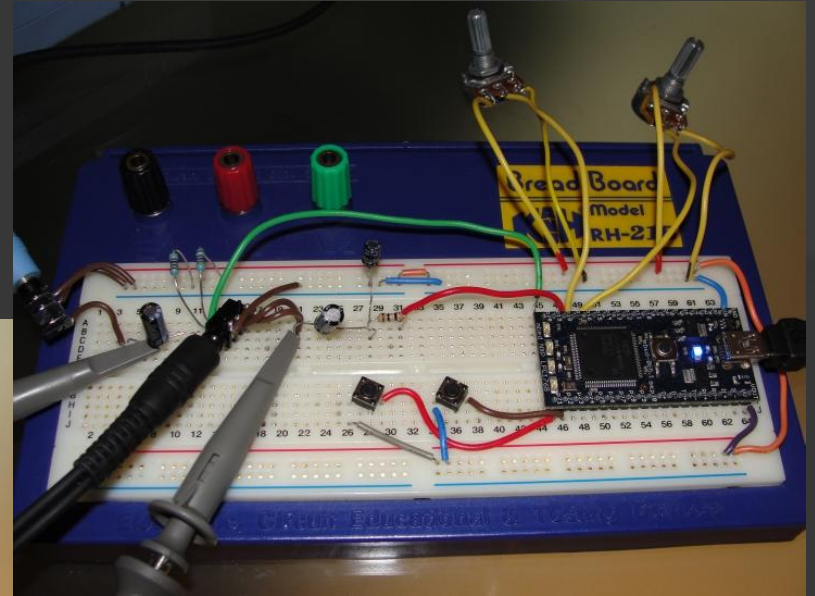
    if (i>(speed)){
        i=0;
        speed=speed_pot*MAX_BUFFER;
        feedback=(1-feedback_pot)*0.9;
    }
    else
    {
        i=i+1;
    }
}
```

# Delay effect waveforms



This project can be developed as a guitar effect unit by adding extra signal conditioning, variable amplification stages and enhanced output conditioning similar to circuits described by Sergeev (2010).

# Images



# References

**Fisher, T. (2010).** Interactive Digital Filter Design (online calculator), accessed from <http://www-users.cs.york.ac.uk/~fisher/mkfilter/>

**Marven, C. and Ewers, G. (1996).** A Simple Approach to Digital Signal Processing, Wiley Blackwell.

**Mbed.org (2010).** Mbed – Rapid Prototyping for Microcontrollers, accessed from <http://mbed.org>

**Proakis, J. G. and Manolakis, D. K (1992).** Digital Signal Processing: Principles, Algorithms and Applications, Prentice Hall.

**Ravet, S. (2010).** Fun with mbed – Billy Bass gets a brain implant, accessed from <http://www.youtube.com/watch?v=Y6kECR7T4LY>

**Sergeev, I. (2010).** Audio Echo Effect., accessed from [http://dev.frozeneskimo.com/embedded\\_projects/audio\\_echo\\_effect](http://dev.frozeneskimo.com/embedded_projects/audio_echo_effect)

**The Mathworks (2010).** FDATool – open filter design and analysis tool, accessed from <http://www.mathworks.com/help/toolbox/signal/fdatool.html>

[rob.toulson@anglia.ac.uk](mailto:rob.toulson@anglia.ac.uk)  
[www.robtoulson.com](http://www.robtoulson.com)  
[www.rt60.co.uk](http://www.rt60.co.uk)  
[www.pleasurizemusic.com](http://www.pleasurizemusic.com)