Application Note: AN01008

Adding DSP to the USB Audio 2.0 L1 Reference Design

This application note shows how to integrate DSP processing into the XMOS USB Audio solution which enables the single tile L8 device to implement both the USB interface together with audio processing in a single chip. The DSP core utilizes the 64bit single cycle MAC instruction to provide audio processing functions such as EQ.

The DSP is provided by the open source sc_dsp_filters module from github to supply the Biquad filters[1]. This module can generate arbitrary cascaded biquad filters with suitable coefficients across a wide dB range. These coefficients are generated at compile time into one of the header files used to build the final executable. This application note is based on the biquad_xta_appnote_1v0 tag in the above repository.

The USB XUD core requires a minimum of 80 MIPS, which limits this example to 6 cores. The design has one core available so we will use all of the available performance for DSP. For simplicity, it requires the device to operate at 500MHz, and will not support MIDI or SPDIF.

A typical use of this application would be in USB active speakers where DSP is required to compensate for physical limitations of the speaker enclosure or in a higher performance device where a digital crossover might be required.

This application note assumes the reader is familiar with the XMOS XS1-L1 DSP Performance Application Note[2] which describes how performance of the biquad filter library can be measured.

[1] https://github.com/xcore/sc_dsp_filters
[2] http://www.xmos.com/published/an01011

1 DSP functionality and performance overview

For audio processing, a single biquad filter can be used to provide a single frequency transformation function - e.g. a high shelf gain (treble boost), or a single peaking frequency function. Use of multiple cascaded filters can be used to produce more complex functions - such as a multi-band equalizer or a digital crossover.

The XS1-L architecture supports a single cycle 32*32 MACC instruction producing a true 64 bit result which is ideal for fixed point DSP operation.

A single biquad filter requires 5 MACC operations. Around this are operations to load the coefficients to be applied and to check for overflow. From the open source XCore repository on Github, an optimised implementation in assembly is available in the sc_dsp_filters repository.

1.1 Building the DSP into the application

The DSP core can be added to the USB Audio reference design by splicing a new core between the decouple and audio cores

```
int main()
{
  chan c_sof;
  chan c_xud_out[NUM_EP_OUT];  /* Endpoint channels for XUD */
  chan c_xud_in[NUM_EP_IN];
  chan c_aud_ctl;
  chan c_aud_out;
  chan c_mix_out;
  .....
  {
    set_thread_fast_mode_on();
    dsp(c_mix_out, c_aud_out, p_button_a, p_button_b);
  }

  {
      thread_speed();
    /* Audio I/O (pars additional S/PDIF TX core) */
    audio(c_aud_out, null, null);
  }
  ....
```

Button A and B are the ports connected to the buttons on the USB Audio L1 Audio board.

The DSP function itself can be written in a separate file - example source code is in the appendix.

The core diagram representation of the application is shown in Figure 1:

The DSP core implementation is based on the mixer functionality from the USB Audio Multi-Channel design to buffer data between the decouple and audio cores. The USB Audio design has a very simple design for buffering data between decouple and output cores based on the audio data rate.

The audio core outputs a word to indicate it is ready to receive samples. The core talking to it responds with either a control token to indicate sample rate change, or with a handshake word back to indicate normal audio data. The audio core then sends 1 sample per input channel and receives one sample per output channel before doing the i/o operations with the ports.

As the DSP core sits between decouple and audio, it has to look like the audio core to decouple, and the decouple core to audio. This behaviour is very similar to the mixer in the USB Audio Multi-Channel design,

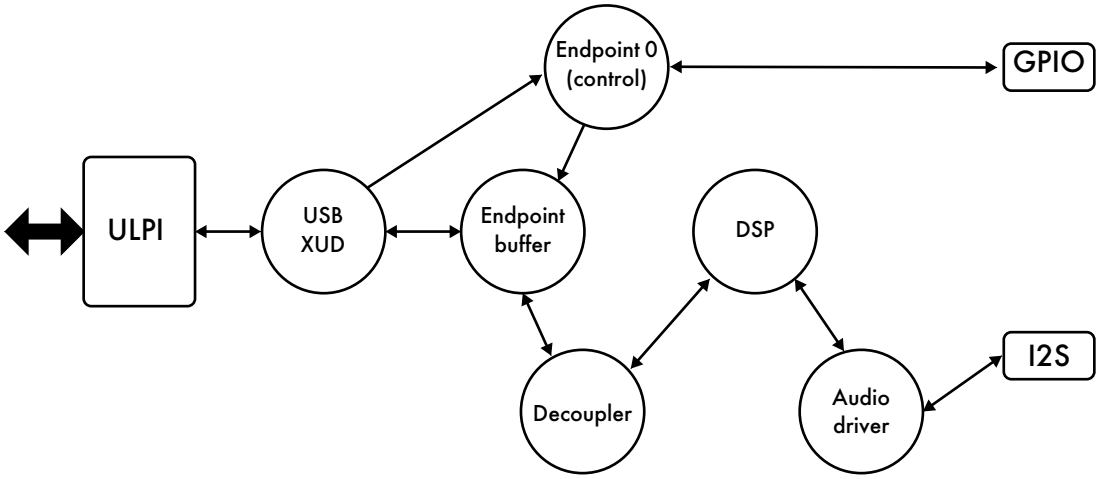

Figure 1: L1 USB Audio software core diagram with DSP core

so the DSP core was based on this functionality.

In turn, it accepts the request for data from the audio core and passes it to decouple. It then handles the handshake from decouple and passes it back to audio. Samples are then passed and received as required, with the core buffering one sample per channel in each direction at any time. The DSP is applied to samples as they are received from decouple before being stored in the local core buffer.

DSP is only applied to data from host to the audio outputs (i.e. output channels), so only two channels have DSP applied. This affects how many filters will fit in a single core. There are a few instructions consumed for handling the incoming audio data, but this is minimal.

The DSP Application note[3] indicates the performance available in a device running 6 cores at 500MHz - with 80 MIPS per core and approx 2.5 MIPS required per biquad filter (or 5MIPS per stereo pair of channels), we could support 16 cascaded biquads per channel for a 2 channel design. This would allow very fine grained EQ or accurate tuning of the output response to suit a particular speaker application.

The example code below is based on using 4 cascaded biquad filters - this is plenty to demonstrate the effects clearly without overlap between the filters which may cause distortion.

[3] http://www.xmos.com/published/an01011

2 Sample application

As mentioned above, the appendix to this document includes the full source code for the DSP core to be included in the USB Audio L1 reference design. For this example some extra features have been added which make the effects of DSP clearly audible.

The coefficients used in the open source repository are computed using very basic algorithms - however custom coefficients created by a DSP expert could just as easily be used.

This code has been written to show what's possible with the XMOS L1 device. This includes live changing of EQ settings either through presets or changing the coefficients for a single filter. The DSP is configured always to be enabled, but with the following settings:

- Notional "off" 0dB response
- Bass boost (shelf filter below "min" frequency in Makefile)
- Treble boost (shelf filter above "max" frequency in Makefile)
- Bass boost, treble boost and peaking filters (as specified in Makefile)

The selection is switched using Button A which iterates over the above presets. Additionally, Button B can be used to independently select the amount of Bass boost (3 settings, cut, normal and boost).

For more details on the Biquad filters please consult the documentation supplied with the biquad module. (https://github.com/xcore/sc_dsp_filters)

The values chosen in the above example are designed to create an easily audible effect so may cause distortion when playing loud tracks at full scale volume.

3 Measuring audio performance

Having built the application and flashed it onto the board, some measurements can be made using a spectrum analysis tool (such as the Spectra PC application). For the plots below, a test PC was setup playing a sweep sine wave (20Hz - 20kHz) with a baseline output level at -40dB through the XMOS USB audio device configured with a binary running the DSP code. The output from the USB Audio board was connected to an analogue input of a recording PC with an EMU-24k soundcard. Spectra was used to record the waveform using peak hold FFT to plot the frequency response. The output level set to -40dB to minimise any distortion introduced by clipping. The red line is a "peak hold" line from the swept input sine wave.

Throughout these tests the volume levels were left unchanged.

Firstly, let's check the frequency response in the default position. Although the biquad filters are enabled, they are configured to produce a flat frequency response:

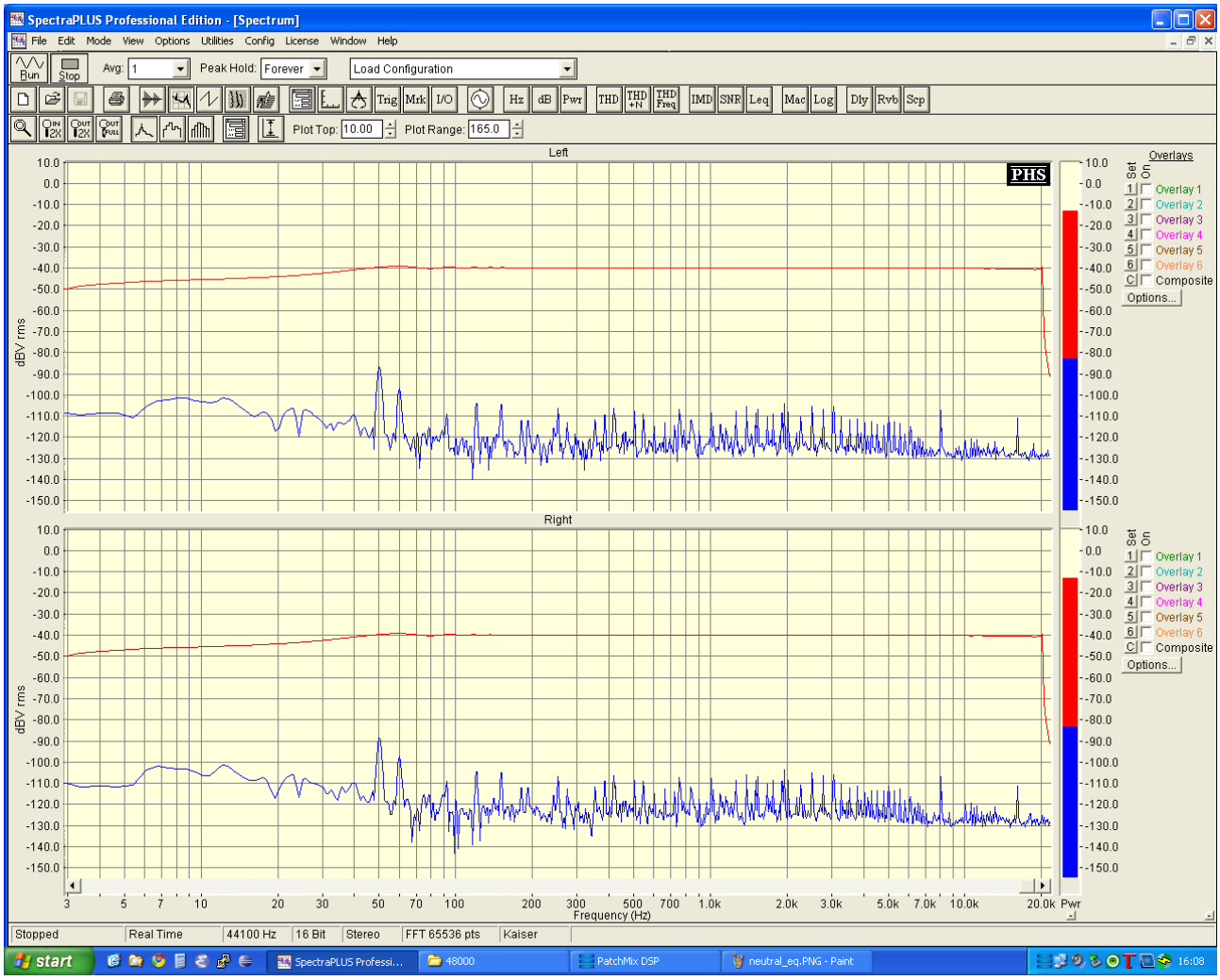

Figure 2: Frequency response for neutral EQ setting

This looks pretty good, with a baseline of -40dB, although there is a rolloff below 50Hz. Next, pressing Button A once selects the Bass boost shelf filter, with a corner at 250Hz and a relative level of +10dB. This

produces the following response:

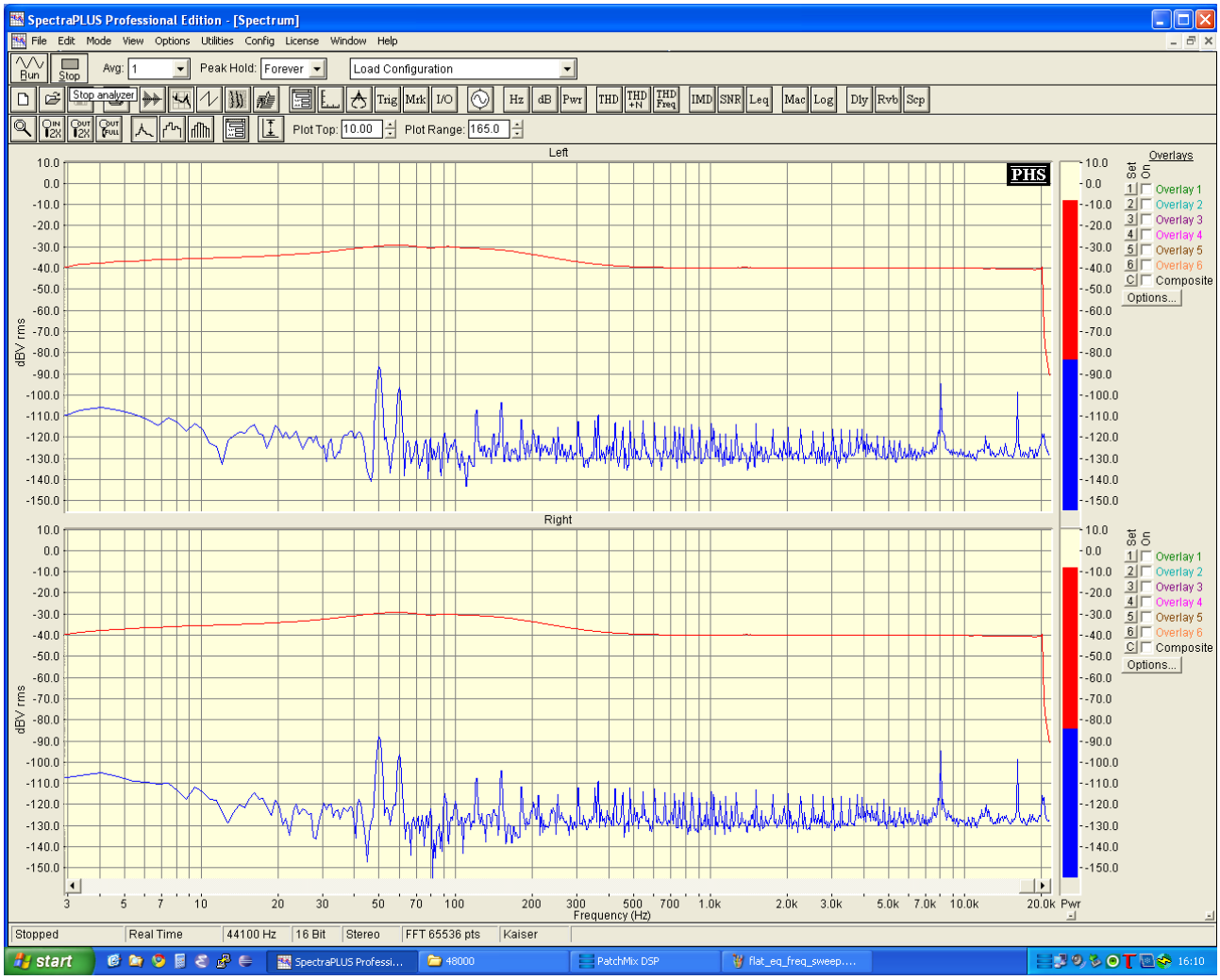

Figure 3: Frequency response for bass boost EQ setting

Again, this looks a good match for our expectations with a boost below 250Hz and the same rolloff below 50Hz as the original trace. The third setting is treble boost only with a corner frequency of 2kHz and again a relative level of +10dB. This produces the following:

A further press of button A selects bass boost, treble boost and the peaking filters at 500Hz and 1000Hz:

The overlap of multiple filters makes calculating the overall response less straightforward.

3.1 Makefile

The following directories from sc_dsp_filters should be copied/cloned into the USB Audio source code directory:

- module_cascading_biquad
- build_biquad_coefficients

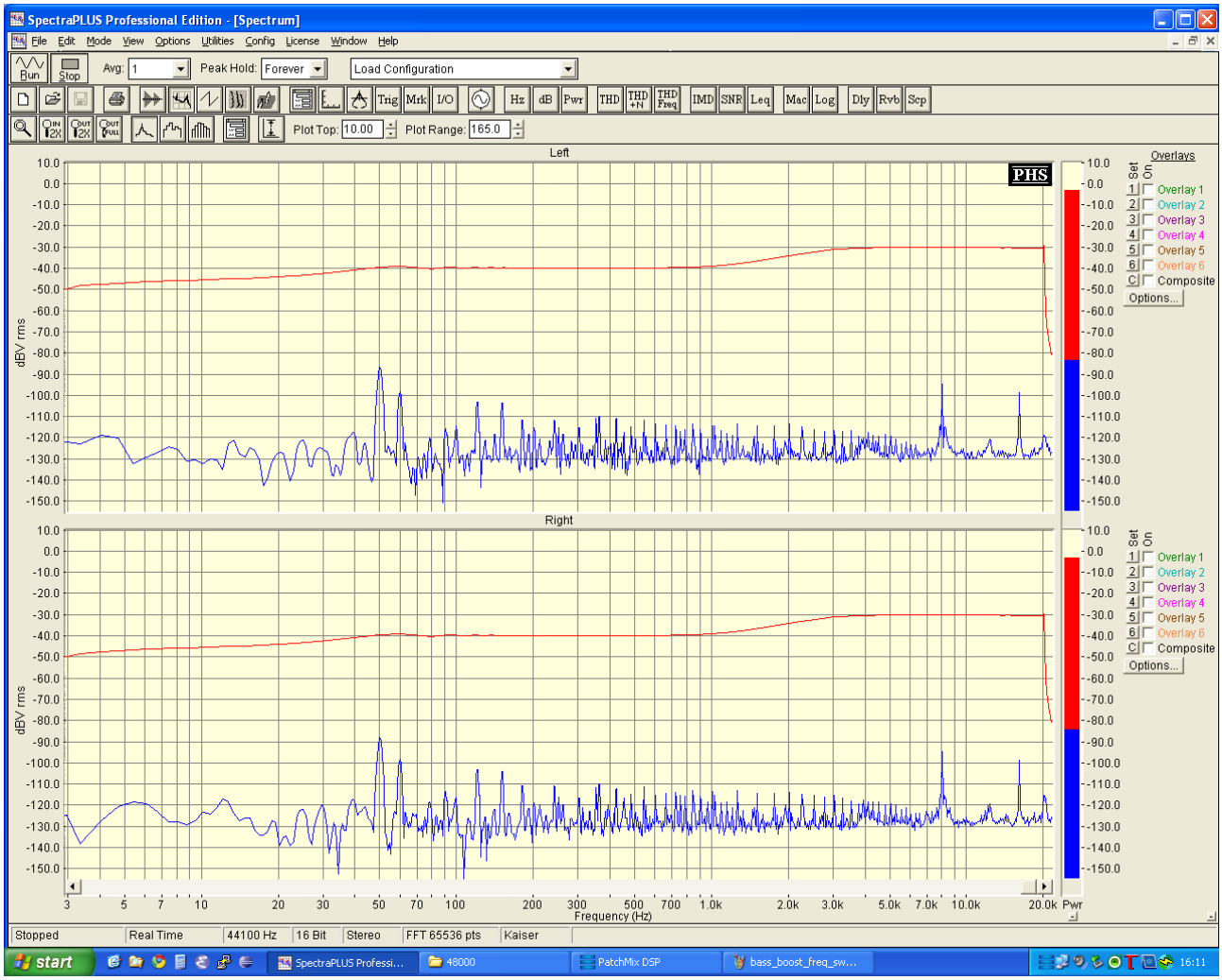

Figure 4: Frequency response for bass boost EQ setting

The Makefile in app_usb_aud_l1 needs to be modified to generate the required coefficient files and to include the biquad module in the build.

The following section should be added, with “coefficients” also added to the “all” build targets:

```
coefficients:
    make -f ../build_biquad_coefficients/Makefile \
             FILTER='-min -20 -max 20 -step 1 -bits 27 -low 250 -high 2000
             -peaking 500 1 -peaking 1000 1' \
             INCLUDEFILE=src/coeffs.h \
             XCFILE=src/coeffs.xc \
             CSVFILE=bin/response.csv

all: coefficients $(BIN_DIR)/usb_audio.xe
```

The above parameters will build a biquad cascade with 4 filters. More can be included by adding further parameters as required. For details of what the parameters describe, please consult the biquad.rst file included in the sc_dsp_filters/doc directory.

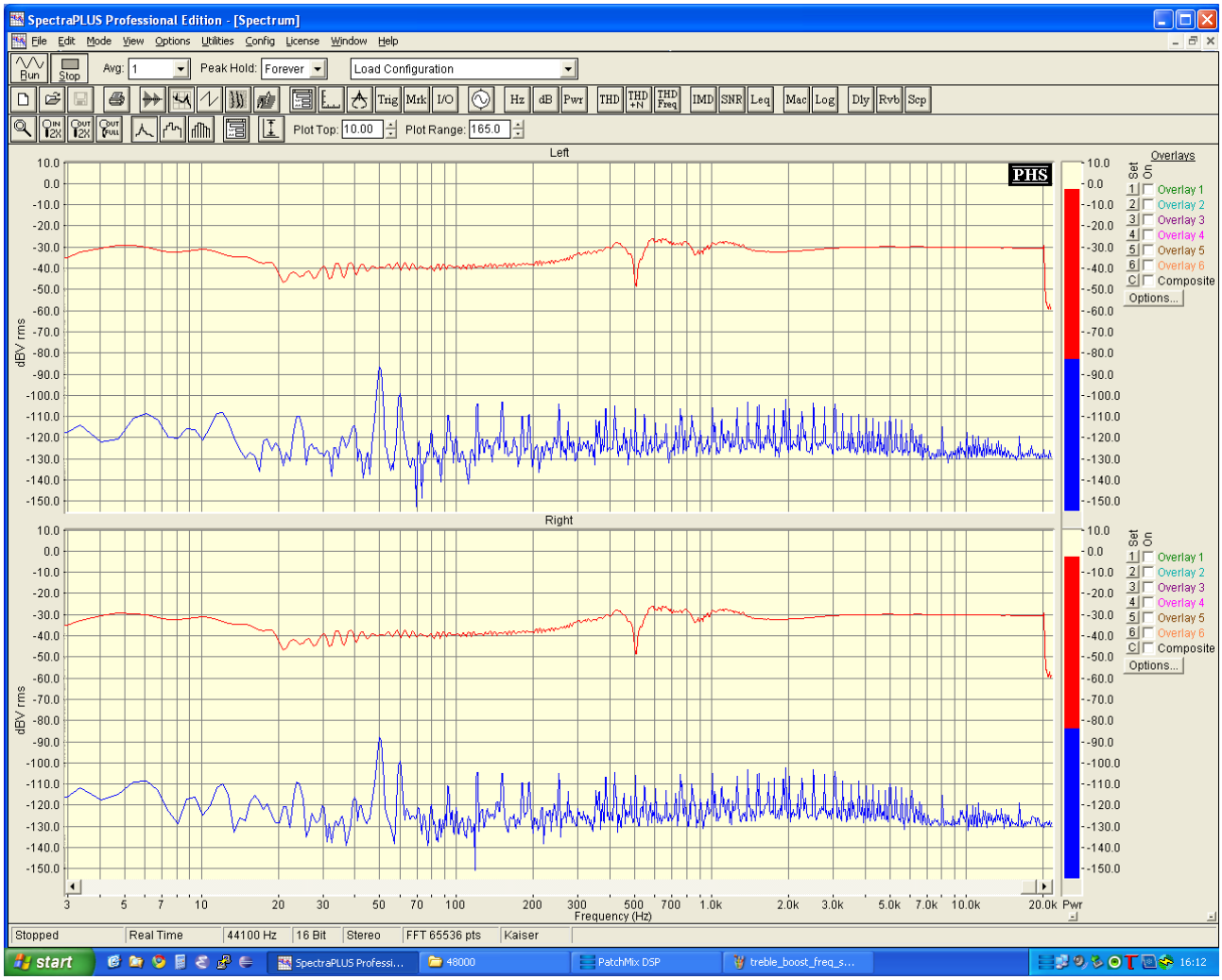

Figure 5: Frequency response for all filters

3.2 Source

From the timing analysis in the DSP component documentation, the rough rule of thumb is 5 MIPS/biquad filter for 48kHz, 2 channel audio. With a core speed of 80MIPS, this suggests up to 16 biquads per channel would be possible. The example Makefile above uses 4 to keep the effects easily audible while trying to minimise any distortion.

The core diagram Figure 1 shows how the DSP core sits between the decouple and audio cores.

4 Conclusion

This application note demonstrates how it is straightforward to integrate code from the open source DSP libraries into a reference design and shows how quickly you can iterate through changes in DSP parameters with only a simple and quick recompilation of the code.

APPENDIX A - DSP function source code

```
// DSP core.

#include <xs1.h>
#include "devicedefines.h"
#include <print.h>
#include "coeffs.h"

extern int biquadAsm(int xn, biquadState &state);

// BANKS is defined in coeffs.h and generated by makefile.

// The change_dsp funcion simply iterates over some presets to demonstrate
// the sorts of possible effects.
// These are chosen for being easily noticed, rather than providing
// subtle sound quality improvements.
// The program assumes that the Bass shelf filter is the first value in the Makefile generation
// and the treble shelf is the second value. Putting the filters in a different order in the makefile
// will change the effects here.
#pragma unsafe arrays
void change_dsp(biquadState &bq, int eq_select){
  switch(eq_select) {
  case 0: // "Off"
#pragma loop unroll
    for (int i =0; i < BANKS; i++) {
      bq.desiredDb[i] = 20 ; //zeroDb value
    }
    break;
  case 1: // Bass boost only
#pragma loop unroll
    for (int i =1; i < BANKS; i++) {
      bq.desiredDb[i] = 20; // zeroDb value
    }
    bq.desiredDb[0] = 30; // Set bass filter to boost
    break;
  case 2: // Treble boost only
#pragma loop unroll
    for (int i =0; i < BANKS; i++) {
      bq.desiredDb[i] = 20; // zeroDb value
    }
    //    bq.desiredDb[1] = 30; // set treble filter to boost
    break;
  case 3: // bass boost, treble boost + all peaking filters (All filters turned on).
    // As the number of filters defined in the makefile increase, it is
    // suggeted you reduce desiredDb towards 20 (no change) to minimise
    // distortion.
    // With 5 banks, a setting of 30 is okay for most music types
    // With 10 banks, a setting of 25 is okay for most music types
#pragma loop unroll
    for (int i =1; i < BANKS; i++) {
      bq.desiredDb[i] = 25;
    }
    break;
  default:
    break;
  }
}
```

```
//give/get functions based on mixer.xc then simplified.

#pragma unsafe arrays
void giveSamplesToHost(chanend c, const int samples[])
{

#pragma loop unroll
  for (int i=0;i<NUM_USB_CHAN_IN;i++)
  {
        int sample;
        sample = samples[i + NUM_USB_CHAN_OUT];
        outuint(c,sample);

  }
}

#pragma unsafe arrays
static void getSamplesFromHost(chanend c, int samples[], biquadState bs[])
{
  unsigned int l_samp[NUM_USB_CHAN_OUT];
#pragma loop unroll
  for (int i=0;i<NUM_USB_CHAN_OUT;i++)
    {
      int sample, x;
      /* Receive sample from decouple */
      l_samp[i] = inuint(c);
    }

  // timing fails if we do dsp in between receiving for both channels.
  // Instead receive into a local buffer, then do the dsp on both channels.
  // XTA times a single call to biquadAsm as 234 core cycles with 4 banks.
  // 48000 sample freq * 2 channels => ~100000 operations/second or 10us
  // allowance between samples.
  // With an 80 MIPS core speed (500MHz/6 cores), this gives ~36 cascaded biquads as the
  // maximum number possible at a max sample freq of 48kHz (ie BANKS=18).
#pragma loop unroll
  for (int i=0;i<NUM_USB_CHAN_OUT;i++)
    {
      int sample;
      sample = biquadAsm(l_samp[i],bs[i]);
          samples[i] = sample;
    }
}

#pragma unsafe arrays
void giveSamplesToDevice(chanend c, const int samples[])
{
#pragma loop unroll
    for (int i=0;i<NUM_USB_CHAN_OUT;i++)
    {
        int sample;
        sample = samples[i];
        outuint(c, sample);
    }
}

#pragma unsafe arrays
void getSamplesFromDevice(chanend c, int samples[])
{
#pragma loop unroll
    for (int i=0;i<NUM_USB_CHAN_IN;i++)
    {
        int sample;
        sample = inuint(c);
                samples[NUM_USB_CHAN_OUT+i] = sample;
  }
}
```

```
void dsp ( chanend c_audio,
           chanend c_decouple,
           in port p_button_a,
           in port p_button_b) {
   /* One larger for an "off" channel for mixer sources" */
  int samples[NUM_USB_CHAN_OUT + NUM_USB_CHAN_IN + MAX_MIX_COUNT + 1];

  timer ta, tb;
  unsigned time_a, time_b;
  unsigned button_a_val = 0, button_a_active = 1;
  unsigned biquad_offset = 0, button_b_active = 1, button_b_val = 0;
  unsigned bass_filter = 10;
  biquadState bs[NUM_USB_CHAN_OUT];

  // 20 is the central value (approx no-eq).
  // As values tend to the extremes, distortion will become more prevalent.
  // The spread of available values is defined in the Makefile where
  // the filter coefficients are set.
  // If a smaller coefficient array is used, these values will need to be adjusted.
  initBiquads(bs[0], 20);
  initBiquads(bs[1], 20);
  set_port_inv (p_button_a);
  set_port_inv (p_button_b);
  // zero samples buffer.
  for (int i=0;i<NUM_USB_CHAN_OUT + NUM_USB_CHAN_IN + MAX_MIX_COUNT;i++)
    {
      samples[i] = 0;
    }

  while(1){ //Implements channel protocol between decouple and audio cores
      inuint(c_audio);                    //Get sample request from audio
      outuint(c_decouple, 0);             //Send sample request to decouple
      if(testct(c_decouple)){             //Test for sample frequency change .
          command = inct(c_decouple);     //Get the CT and command - See commands.h
          value = inuint(c_decouple);     // Get command value from decouple core - normally SR value
          outct (c_audio, command);       //Send control token to audio (SR change)
          outuint (c_audio, value);       //Now send value to audio
          chkct (c_audio, XS1_CT_END);    //wait for handshake
          outct (c_decouple, XS1_CT_END); //Forward handshake to decouple
      }

      else{  // Normal audio loop
          underflow = inuint(c_decouple);   // Get confirmation from decouple indicating we're ready
          outuint (c_audio, underflow);     // Pass it on to audio

          getSamplesFromDevice (c_audio, samples);//Always get input samples from device and send to audio
          if (!underflow) giveSamplesToDevice (c_audio, samples); //Only send audio if not in underflow
          giveSamplesToHost (c_decouple, samples);//Always send samples to decouple
          if (!underflow) getSamplesFromHost (c_decouple, samples, bs);//Only get samples from decouple if no
            ↪ underflow
      }
    }
  }
```

```
    // The section below demonstrates how a user interface can
    // be used to change the EQ settings on the fly using the A and B buttons
    // on the L1 reference board.

    // Sample the buttons values with a timeout for debounce.
    // Button A changes the Bass shelf filter through 3 settings (cut, normal, boost)
    p_button_a :> button_a_val;
    if (button_a_val && button_a_active) {
      bass_filter = bs[0].desiredDb[0];
      bass_filter += 10;
      if (bass_filter > 30) {
        bass_filter = 10;
      }
      bs[0].desiredDb[0] = bass_filter;
      bs[1].desiredDb[0] = bass_filter;
      ta:>time_a;
      button_a_active = 0;
    }

    // Button B iterates over the 4 DSP presets
    // B will clear any changes made with button A.
    p_button_b :> button_b_val;
    if (button_b_val && button_b_active) {
      biquad_offset ++;
      biquad_offset &= 3;
      change_dsp(bs[0], biquad_offset);
      change_dsp(bs[1], biquad_offset);
      tb:>time_b;
      button_b_active = 0;
    }

    // 0.5s timeout on button press for B, 0.3s on A
    // Holding the buttons down will then iterate over the possible settings with this time period
    // between changes.
    select {
    case (button_b_active == 0 ) => tb when timerafter (time_b + 50000000) :> int _:
      button_b_active = 1;
      break;
    case (button_a_active == 0 ) => ta when timerafter (time_a + 30000000) :> int _:
      button_a_active = 1;
      break;
    default:
      break;
    }
  }
}
```