VSIDE DSP LIBRARY

VS1005, VS1063, VS1053, VS1003, VS1103

Project Code:
Project Name: VSMPG

Revision History			
Rev.	**Date**	**Author**	**Description**
1.00	2015-04-30	HH	Initial release.

Contents

List of Figures

1 Introduction

This is an introduction, tutorial and manual for VLSI Solution's VSDSP VSIDE DSP Library. It explains how to use the library with VS1005/VSOS3 (hereforth called just VS1005), as well as with VS1063, VS1053, VS1003, and VS1103 (hereforth called VS10xx). In addition, it provides some basic information on digital signals.

The document is arranged as follows.

Chapter 2, *Definitions*, defines shorthands and terms.

Chapter 3, *Overview of the VSIDE DSP Library*, presents on overview of the VSIDE DSP Library.

Chapter 4, *Requirements*, tells the prerequirements for using the VSIDE DSP Library.

Chapter 5, *Digital Audio Basics*, while being far from comprehensive, provides some basic information of what digital audio systems are, what digital filters are, and what you can do with them.

Chapter 6, *Tutorial: Designing and Using an FIR Filter*, is a tutorial that shows how to design an FIR filter using Octave, and how to convert that filter into VSDSP code.

Chapter 7, *Tutorial: Compiling And Linking Your Filter with VSIDE*, continues the tutorial tour by showing how to compile and run your code with VSIDE.

A reference to the DSP Octave functions is presented in Chapter 8, *Function Reference*.

Installation to Microsoft Windows operating systems is shown in Chapter 9, *Installing GNU Octave + VSIDE DSP Library, Windows 8.1*.

Chapter 10, *Troubleshooting*, provides troubleshooting.

Chapter 11, *How to Load a Plugin*, explains how to load a plugin to a VS10xx IC (not VS1005).

Finally, Chapter 12, *Contact Information*, contains VLSI Solution's contact information.

2 Definitions

ADC Analog-to-digital converter

Aliasing Aliasing occurs almost always when a real signal is interpolated or decimated. It causes incorrect frequency components to the signal. To avoid aliasing, anti-aliasing filters need to be used. See also *anti-aliasing filter*.

Amplitude / **Peak amplitude** Maximum absolute value of the signal. See also *peak-to-peak amplitude*.

Anti-aliasing filter A filter that modifies the signal in such a way that interpolating or decimation doesn't cause more aliasing that what is acceptable.

A-weighting A frequency response curve defined in IEC 61672:2003 for frequencies between 20 and 20000 Hz. It can be applied to sound in order to approximate how human hearing responds to different frequencies. The A-weighting curve is shown in Figure 1 on page 8.

B Byte, 8 bits

b Bit

DAC Digital-to-analog converter

dB Decibel, a logarithmic unit that indicates the ratio of two powers. 1 bel, which equals 10 decibels, is a power ratio of 10. A power ratio of 4:1, which in audio equals an amplitude ratio of 2:1, is approximately 6.02 dB, and is often rounded to 6 dB. With additional qualifiers (like dBfs, or dbV), the decibel scale can also be used to quantify absolute signal levels.

dB(A) / **dBA** Decibel, but with the signals to be compared run through an A-weighting filter before calculation. A-weighting usually gives results more relevant to how humans hear differences than measurements done without a weighting filter. See also *A-weighting*.

dBfs Decibel full scale. Decibel scale where the zero point is bound to a digital full scale sine wave.

dBV Decibel volt. Decibel scale where the zero point is bound to 0 dBV = 1 Vrms. See also *Vrms*.

Decimation The act of discarding values so that only a given fraction of original values are left. E.g. Decimating a signal by two removes every other sample, and this also halves its sample rate. To prevent aliasing, a decimation filter is usually applied before decimation. See also *aliasing*.

FIR Finite Impulse Response. With FIR filters each input sample affects only a certain number of output samples, and the end result is always stable.

IIR Infinite Impulse Response. With IIR filters each input sample affects (even if in a very minor way) the output for infinity.

Interpolation The act of adding zeroes between existing values to increase their amount. E.g. Interpolating a signal by two doubles the amount of samples, and thus also doubles its sample rate. Interpolation is usually followed by running an interpolation filter to prevent aliasing. See also *aliasing*.

Ki "Kibi" = 2^{10} = 1'024 (IEC 60027-2)

Mi "Mebi" = 2^{20} = 1'048'576 (IEC 60027-2)

Gi "Gibi" = 2^{30} = 1'073'741'824 (IEC 60027-2)

Pass-band The part of a filter's frequency response where the filter shouldn't change the amplitude of the signal. See Chapter 5.4, *Linear Digital Filters*, for more details.

Peak-to-peak amplitude The difference between highest and lowest signal value. See also *amplitude*.

% Per cent. Sometimes used to present THD instead of the dB scale. To convert from dB scale to per cent scale, use the following formula: $pc = 100 \times 10^{-THD/20}$.

SNR Signal-to-Noise Ratio. Usually given in dB, or dB(A) if A-weighting curve is used. See also *dB(A)*.

Stop-band The part of a filter's frequency response where the signal is fully attenuated. See Chapter 5.4, *Linear Digital Filters*, for more details.

THD Total Harmonic Distortion. Usually given in dB or as a percentage. 1 % = -40 dB, 0.1 % = -60 dB, 0.01 % = -80 dB, etc. Given in dB(A) if A-weighting curve is used. See also *dB(A)*.

THD+N Total Harmonic Distortion plus Noise. Usually given in dB, or dB(A) if A-weighting curve is used. See also *THD*, *SNR*, *dB(A)*.

Transition band The part of a filter's frequency response where it is transitioning from pass-band to stop-band. See Chapter 5.4, *Linear Digital Filters*, for more details.

Vpp Volts, peak-to-peak. A sine signal of 1 Vpp = $\frac{1}{2\sqrt{2}}$ Vrms. See also *Vrms*.

Vrms Volts, root mean square. A sine signal of 1 Vrms, or 0 dBV, presented in Figure 2 on page 8, has an amplitude of $\sqrt{2}$ V, and a peak-to-peak amplitude of $2\sqrt{2}$ Vpp. Note that this relationship between Vrms and V/Vpp is only valid for a single sinusoidal signal. It doesn't apply for more complex waveforms. See also *dBV*, *Vpp*.

VSDSP VLSI Solution's DSP core

VSOS VLSI Solution's Operating System

VSIDE VLSI Solution's Integrated Development Environment, available at *http://www.vlsi.fi/en/support/software/vside.html*

W Word. In VSDSP, instruction words are 32 bits and data words 16 bits wide.

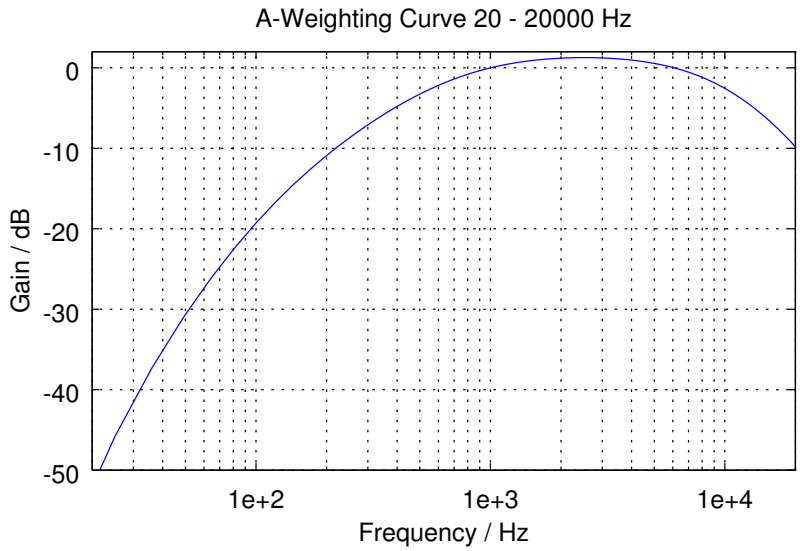

Figure 1: A-weighting curve 20...20000 Hz

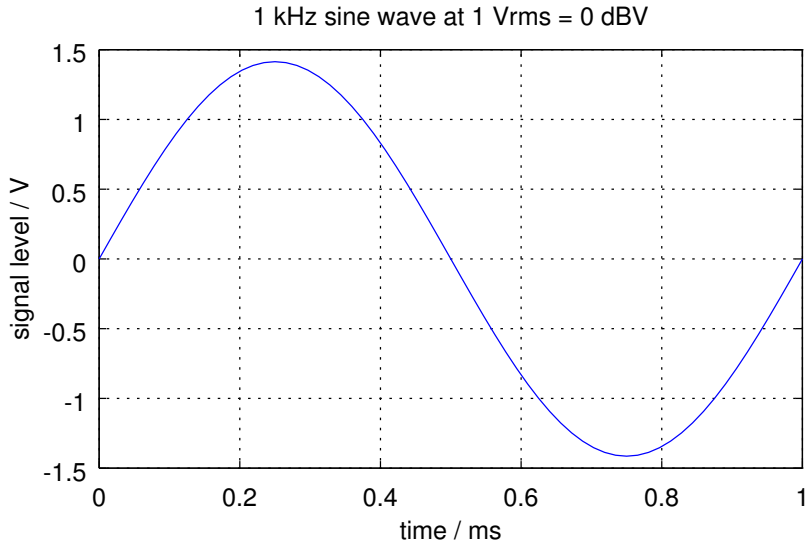

Figure 2: 1 kHz sine wave at 1 Vrms = 0 dBV, amplitude ≈ 1.41 V ≈ 2.83 Vpp

3 Overview of the VSIDE DSP Library

The VSIDE DSP Library is a library that offers digital signal processing services to the user. It contains filters highly optimized to take the most out of the VSDSP hardware.

At this stage, The VSDSP Library supports Finite Impulse Response (FIR) filters. It automatically supports many several optimized FIR filter special cases, like half-band filters, bandwidth splitting filters, as well as upsampling and downsampling filters.

To make filter design easier, help functions for the free GNU Octave numerical computations interpreter are provided.

3.1 VSIDE DSP Library Package Files and Folders

VSIDE DSP Library package folders	
Folder Name	**Contains**
audio/	Example audio files (Chapter 6.1.2).
docs/	This documentation.
octavefiles/	Files needed for GNU Octave (Chapter 9).
solutions/	VSIDE example DSP solutions (Chapter 7).

4 Requirements

These are the requirements for creating and implementing filters using this document.

- VLSI Solution's Integrated Development Environment VSIDE must be preinstalled. VSIDE is available for free for Windows at
  http://www.vlsi.fi/en/support/software/vside.html
  VSIDE also partially works under Linux' Wine.
- The GNU Octave numerical computations interpreter needs to be installed. GNU Octave is available at
  http://www.gnu.org/software/octave/
  For details on how to install GNU Octave and the VSIDE DSP Library package to a computer with a Microsoft Windows operating system, see Chapter 9, *Installing GNU Octave + VSIDE DSP Library, Windows 8.1*. For Linux variants, Octave is typically directly available in the Operating System's package management system.
  - You need to install the signal processing functions package to Octave. To check if the package is installed, start Octave, and type on the command line:
    `help remez`
    If you get an error message of a missing function, you don't have the package installed. See Chapter 9, *Installing GNU Octave + VSIDE DSP Library, Windows 8.1*, for hints on how to get the function working.
  - You need to install VLSI Solution's fir2vsdsp.m function file to your Octave path or working directory. To check if the file is in your path, start Octave, and type on the command line:
    `help fir2vsdsp`
    If you get an error message of a missing function, you don't have the package installed.

5 Digital Audio Basics

Digital audio consists of audio samples that are gathered at a given frequency (called the sampling frequency, or sample rate), and a given accuracy (usually given by the number of bits used to encode the signal).

5.1 Sample Rate and Bandwidth

According to the Nyquist-Shannin sampling thoerem, the maximum potential bandwidth of a digital system is exactly half of the sample rate. So, according to the theory a CD, which has a sample rate of 44100 Hz, can represent audio upto 22050 Hz.

For practical reasons, though, in any real high-fidelity digital system, bandwidth needs to be limited by filters to be slightly lower than the theoretical maximum. This makes it possible to implement the reconstruction filters that are required to make the necessary sample rate upconversions before audio can be sent to an analog amplifier. For instance, in CD audio, bandwidth has been limited to 20 kHz for exactly this reason.

High sample rates makes it easier to retain a good frequency response in the 20 kHz human hearing frequency area. For instance, at 48 kHz, the required filters for perfect playback can be much more relaxed than in a 44.1 kHz system.

In double blind tests there have been no cases where increasing the sample rate above 48 kHz, or the frequency response above 20 kHz, have proven beneficial for high fidelity audio playback. Increasing the sample rate does, however, make digital filtering easier upto a point.

Perhaps a bit surprisingly, using higher sample rates can also have adverse effects to sound quality. Inaudible ultrasound frequencies that cannot be handled cleanly by an amplifier, may lead to audible intermodulation distortion.

5.2 Bits, SNR, and Dynamic Range

When digitizing audio, the bit-accuracy of the digitizing process sets the absolute upper limit to how large a linear signal-to-noise ratio (SNR) the signal can have.

The formula to calculating the SNR that you can get for a single, maximum amplitude sine wave over the whole signal bandwisth is approximately 1.76 dB$+b \times 6.02$ dB, where b is the number of bits. So, a 16-bit system is capable of at most an SNR of 98.1 dB.

While the SNR of a 16-bit system can be 98 dB, in real systems it is usually not that high. With a full-scale signal there may be some noise in the signal that does not appear when the signal is near silent. Although measurable, this noise can often not be heard because the strong signal is masking the noise or other distortions.

Dynamic range tells the difference between the loudest representable single sine signal

and the background noise on an otherwise empty channel (or, in some cases, the difference between the loudest and softest representable signal). Typically it is larger than the best SNR in the system.

Linear SNR is, though, not the whole story. For an audio signal, A-weighting is closer to how a human ear hears. For a 16-bit 48 kHz system with uniformly white noise, the maximum theoretical A-weighted SNR for a 1 kHz sine signal is 101.8 dB(A) (the (A) after the dB number means that the measurement has been performed with A-weighting).

The issue of dynamic range becomes very different if an audio system is capable of using a much higher sample rate than is required for human hearing. In this case dynamic range and SNR in the audible 20 kHz bandwidth can be increased to significantly higher numbers than could be expected from the number of bits alone. This makes it possible to create analog-to-digital and digital-to-analog converters that use just one or a few bits running at upto several megahertz frequencies, and which still give very good audio performance. Examples of this are VLSI Solution's VS10xx audio ICs which had their ADCs and DACs always running at a fixed rate of 6.144 MHz.

5.3 Good Digital Audio Is Not Stair-Stepped

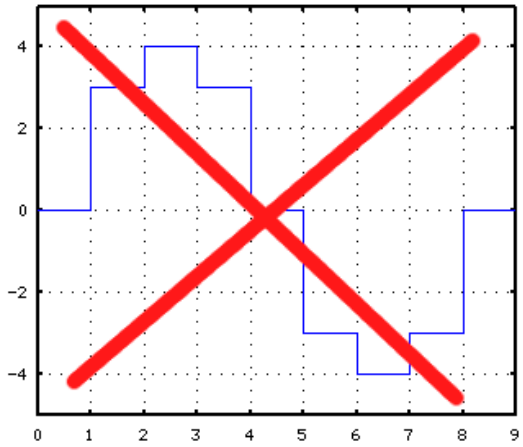

Figure 3: Good digital audio does *not* look (nor sound) like this!

A common misconception of digital audio is that, as shown in Figure 3, it is stair-stepped. For properly filtered digital systems, this is not true. A major task of digital filtering included in the hardware of VLSI Solution's Digital-to-Analog converters is to upsample the signal to such a high frequency (6.144 MHz), that the audio in the audible band and close to it is very smooth and "rounded". The rest of the noise, way out of the audible band, is handled by very simple low-order analog RC filters. After this, the waveform that can be measured from e.g. the line output of a VS10xx IC is extremely smooth, and without any jaggies.

An example for how audio upconversion is handled in a digital system so that resulting audio is not stair-stepped can be seen in Chapter 6.1.2, *Designing a Sample Rate Upconversion Filter*.

5.4 Linear Digital Filters

Linear digital filters are numerical algorithms which manipulate the frequency (and phase response) of a signal. As an example, they can be used to remove unwanted or incorrect information, like background noise or aliasing distortion.

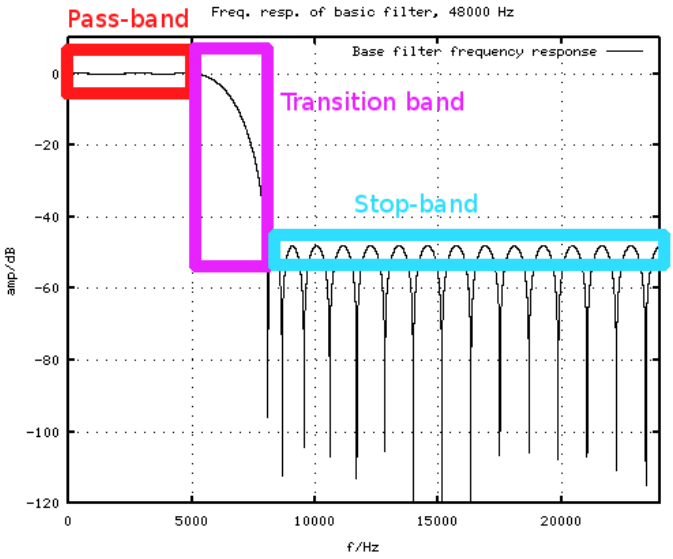

Figure 4: Frequency response of a low-pass FIR filter

Before designing a filter we need to familiriaze ourselves with a few basic terms. Figure 4 shows the frequency response of a low-pass filter which has a pass-band of 0-5 kHz, a transition band from 5 to 8 kHz, and a stop-band rejection of about 48 dB. What this means is that low frequencies under 5000 Hz, essential for speech, are preserved with good fidelity, while frequencies higher than that are progressively muted, until at 8000 Hz and higher they are attenuated so much that they are almost inaudible.

The X-axis in the figure shows the frequency in Hertz (Hz), and the Y-axis shows the gain of the filter in decibels (dB).

The pass-band box shows the pass-band portion of the frequency response. By magnifying the pass-band area (shown in Figure 5), we can see that the pass-band ripple, or in other words the unwanted fluctuation of frequency response in the pass-band, is about ± 0.04 dB, which is very low.

The transition band box shows the transition band, which lies between the pass-band and stop-band. In this are, the signal is partially attenuated, but not as much as required from the stop-band.

The stop-band box shows the stop-band portion of the frequency response. Here we can see that the worst stop-band gain is about -48 dB. This is enough for many applications, though it would be insufficient for a 16-bit sample-rate conversion filter.

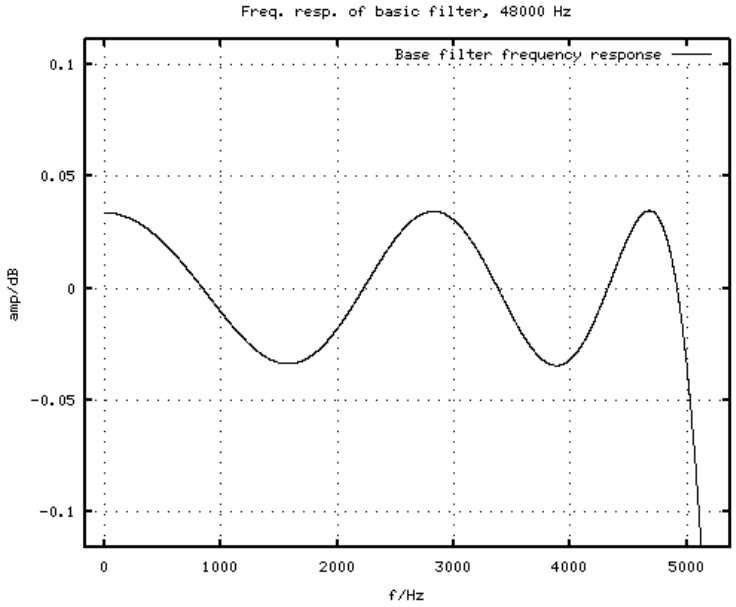

Figure 5: Pass-band frequency response of a low-pass FIR filter

So, before designing a filter we need to think of a few important parameters; namely:

- Largest acceptable pass-band ripple. The lower the ripple, the longer the filter.
- The minimum required stop-band attenuation. The higher the attenuation required, the larger the filter.
- Transition band width. The narrower the transition band, the longer the filter.
- The order/length of the filter. The higher the order, the better the performance, but also the higher the requirements for memory and processing power.

There are two basic filter types:

- Finite Impulse Response (FIR) filters where each input sample affects only a certain number of output samples, and the end result is always stable.
- Infinite Impulse Response (IIR) filters where each input sample affects the output for infinity.

FIR filters are generally easier to handle, and they are easy to design to be linear phase. They also by definition always stay stable. However, IIR filters have an advantage particularly when handling steep filters at low frequencies.

5.5 Further Reading

For more basics of digital audio, have a look at *http://en.wikipedia.org/wiki/Digital_audio*

Further information on some of common digital audio misconceptions can be found at *http://xiph.org/ xiphmont/demo/neil-young.html*

6 Tutorial: Designing and Using an FIR Filter

In this tutorial, we will use GNU Octave to design a few FIR filters for various applications. Then we will convert those filters to VSDSP code.

6.1 Designing an FIR Filter with GNU Octave

Two basic functions for designing low-pass, high-pass, and band-pass filters using Octave are REMEZ() and FIRLS(). Though pretty similar, they are not exactly the same, and depending on the application, one might be superior to the other.

6.1.1 Designing a Low-Pass Filter

Let's say we are sampling audio at 48 kHz. In this case, we have a theoretical audio bandwidth upto 24 kHz, though in practise the ADCs are in this case limited to a bandwidth of a little over 20 kHz.

Now let's say that, for some reason, we want to limit the audio bandwidth from the original audio 20 kHz bandwidth to around 6 kHz. To do this, we need to design a low-pass filter.

To design a low-pass filter, we can use the Octave REMEZ() function, along with VLSI Solution's FIR2VSDSP() which plots the frequency response. So, start up Octave, and enter the following commands:

```
fstart=48000;up=1;down=1;fup=fstart*up;fdown=fup/down;fsp2=fup/2;
lowpass=remez(30,[0 5000/fsp2 10000/fsp2 1],[1 1 0 0],[1 100]);
fir2vsdsp(lowpass,16,1,1,"lowpass","-rate",fstart);
```

On the first line we tell the sample rate is 48 kHz, and define some help variables that are going to be useful for us later.

On the second line, we create a low-pass filter with the following features:

- 30th degree (we get a 31-tap filter, with 31 coefficients). By changing this number you can affect the quality of the filter.
- Pass-band from 0 to 5000 Hz, stop band from 10000 Hz to fs/2.
- [1 1 0 0] tells that the amplitude for the pass-band should be multiplied by 1, and the stop-band by 0. If you replace the vector with [0 0 1 1], you will get a high-pass filter. Numbers other than 0 and 1 should be used with caution.
- The last parameter, [1 100] tells the weights for the quality of pass-band and stop-band, respectively. For instance, if you replace it with [1 1], or omit it, pass-band ripple (which is now about $\pm$0.1 dB if you zoom on the pass-band) will decrease, but stop-band attenuation will get worse. Weighting needs to be optimized separately for each filter.

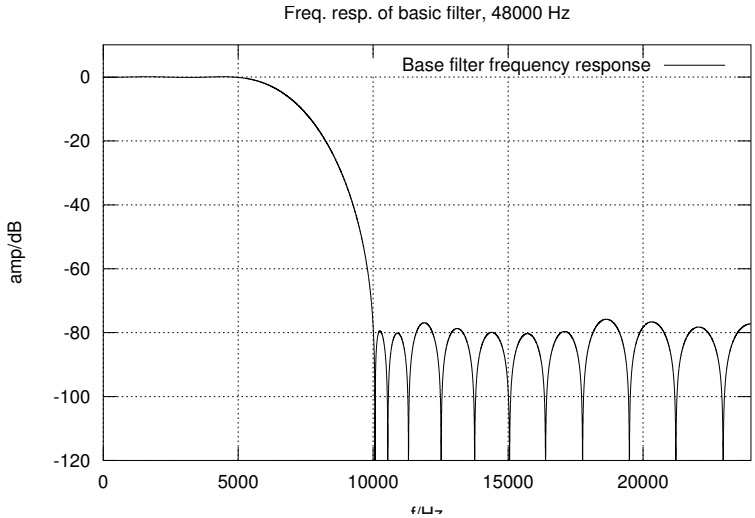

Figure 6: Low-pass filter frequency response

Finally, on the third line we get the VSDSP C source code for the filter (more on that later, in Chapter 6.2), and a plot of the frequency response of the filter, as shown in Figure 6. The parameters to FIR2VSDSP() are interpreted as follows:

- The filter to be converted to VSDSP code is in variable *lowpass*.
- 16-bit accuracy.
- Upsampled by a multiplier of 1 (i.e. *not* upsampled).
- Downsampled by a multiplier of 1 (i.e. *not* downsampled).
- The name of the filter C implementation will become “lowpass”.
- The sample rate of the source signal is 48 kHz.

If you want to graphically see the coefficients of the filter you have just designed, you can type:

```
plot(lowpass); grid on;
```

Congratulations! You have just designed your first filter! Now try around with the parameters provided and see how they affect your end result.

6.1.2 Designing a Sample Rate Upconversion Filter

Let's say we have audio encoded at 8 kHz, but we want to play it back on a system that can only play at 48 kHz. The naive way to convert would be to just copy each sample 6 times. However, this sample-and-hold filter would lead to exactly that kind of stair-stepped audio that Chapter 5.3 preached against.

A second idea would be to smooth the waveform by doing linear interpolation between the new sample points. In other words, make weighted averages of the samples so that new samples are formed by "drawing straight lines" between original sample values. Alas, the signal processing properties of that approach is not much improved over the sample-and-hold filter.

The correct way is to generate an upsampling, or interpolation filter. To generate such a filter, we can run the following commands:

```
fstart=8000;up=6;down=1;fup=fstart*up;fdown=fup/down;fsp2=fup/2;
up6=remez(82,[0 3200/fsp2 5500/fsp2 1],[1 1 0 0],[1 200]);
fir2vsdsp(up6,16,up,down,"upBy6","-rate",fstart);
```

Here, we have defined a low-pass filter which has a 3.2 kHz pass-band, and a stop-band that starts from 5.5 kHz. More emphasis is laid on the stop-band than on the pass-band.

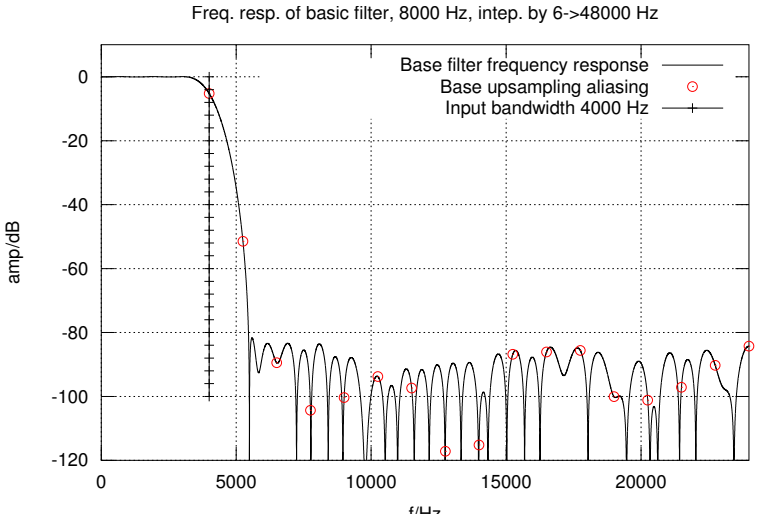

Figure 7: High-quality 8000 Hz to 48000 Hz interpolation filter frequency response

As a result we get a frequency response image roughly as shown in Figure 7. Let's see how to read this picture.

First of all, the original signal was sampled at 8 kHz. Thus, the theoretical upper limit for bandwidth is 4 kHz, indicated by a vertical black line. The part of the frequency response to the left of that line is the part we want to preserve as well as possible. Anything to the right of the 4 kHz line is aliasing distortion, or what the "stair-stepped" audio would sound like, so we want to remove that as well as we can.

As can be seen from the image, the stop-band rejection is over 80 dB, which makes

all aliasing distortion above 5.5 kHz inaudible, though not immeasureable. With 16-bit filters it is difficult to get much better image rejection figures without hand-optimizing the filter coefficients.

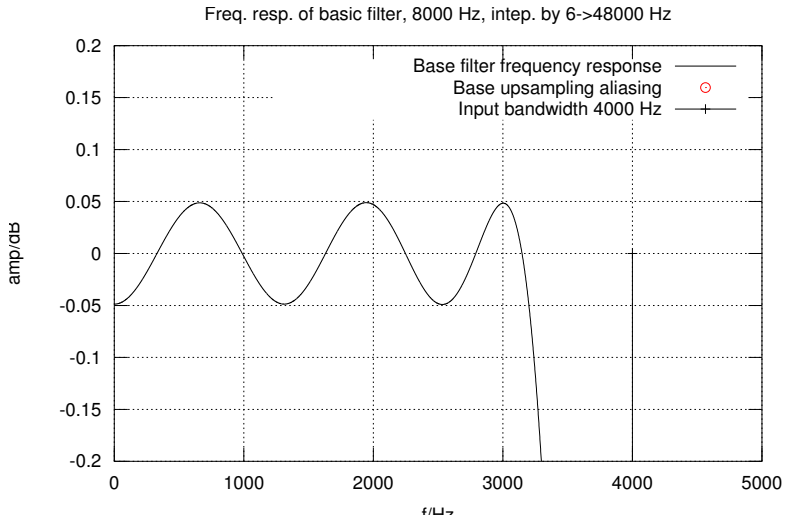

Figure 8: High-quality 8000 Hz to 48000 Hz interpolation filter pass-band ripple

If we zoom into the pass-band area, as shown in Figure 8, we can see that out filter has a pass-band ripple of about ± 0.05 dB. This is way beyond the human hearing threshold for frequency response fluctuations.

All in all, for many applications this would be a very agreeable interpolation anti-aliasing filter.

Oh, but how about those trivial methods mentioned earlier, the sample-and-hold, and linear interpolation methods? As a matter of fact, they are linear filters, and their frequency response can be plotted in just the same way as for the filter we designed.

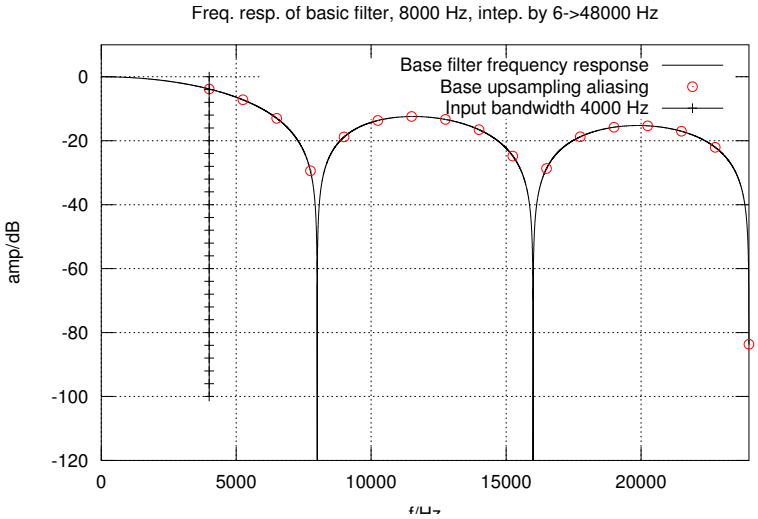

Figure 9: Sample-and-hold 8000 Hz to 48000 Hz interpolation filter

Figure 9 shows the frequency response of a sample-and-hold upsampling filter. The

pass-band is warped (2 kHz gain is -2 dB), and the worst stop-band rejection (at 12 kHz) is only 12.5 dB.

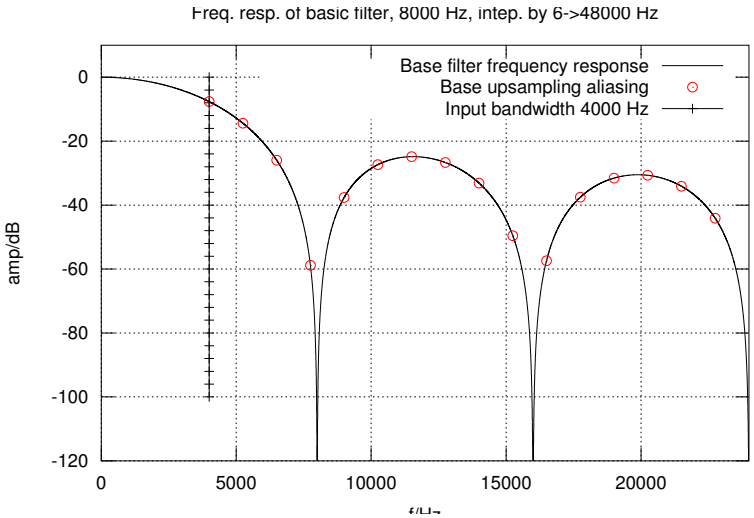

Figure 10: Linear interpolation 8000 Hz to 48000 Hz interpolation filter

The linear interpolation filter frequency response in Figure 10 is not much better. The pass-band frequency response is even worse (the gain at 3 kHz is -4 dB, against the $\pm$0.05 dB ripple of our proper filter), and the worst stop-band rejection at 12 kHz is less than 25 dB. Speech audio fed through this filter will certainly have “that digital sound”.

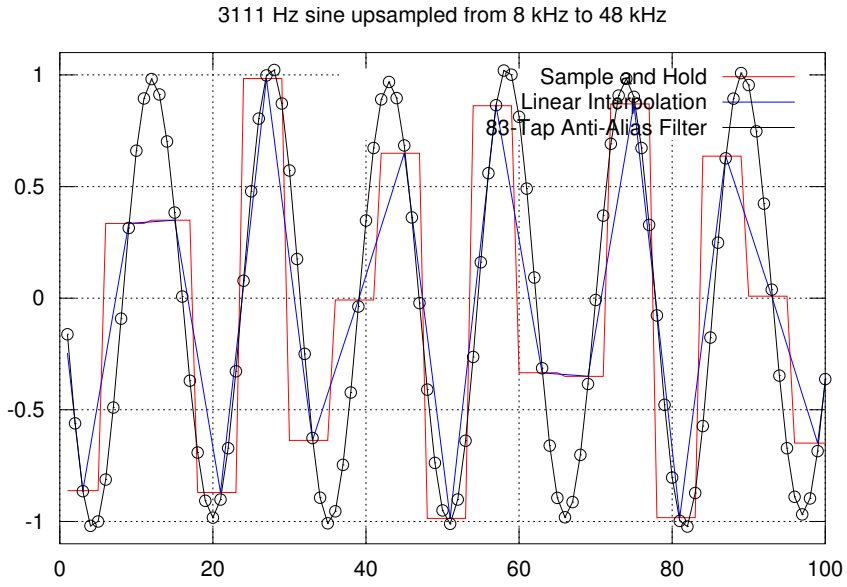

Figure 11: 3111 Hz signal waveform from three 8000 Hz to 48000 Hz interpolation filters

But what does all this mean in practice? Figure 11 shows a 3111 Hz sine signal that has been sampled originally at 8 kHz, then upsampled to 48 kHz using the three methods presented in this section: sample-end-hold (copy the same value 6 times), linear interpolation (draw a straight line between samples), and using the “proper” anti-aliasing filter

we just designed. As you can see, our anti-alias filter has done a great job of retaining the original waveform, whereas both the sample-and-hold and linear interpolation methods have failed miserably.

Yes, you might ask, this all looks good on paper, but can you actually hear these differences? To demonstrate how they sound, this package contains example audio files, generated with the three upsampling methods mentioned.

Example files in the audio/ folder	
File Name	**Description**
Upsample_08kHz_Original.wav	Original speech audio sample. Playback quality may vary depending on the filters of the playback device.
Upsample_48kHz_SampleAndHold.wav	Upsampled by duplicating sample values.
Upsample_48kHz_LinearInterpolation.wav	Upsampled by “drawing lines” between samples.
Upsample_48kHz_ProperFilter.wav	Upsampled using the filter designed in this chapter.

6.1.3 Designing a Sample Rate Downconversion Filter

When we designed a sample rate upconversion filter, or interpolation filter, we first interpolated audio to a new sample rate, and then filtered away the frequencies that would end up as aliasing distortion.

Designing a downconversion filter, or decimation filter, is pretty similar. However, here we have to first remove all the audio frequencies that could not be presented by our destination sample rate, and then just drop samples until we get to that rate.

In this case, let's start with a sample rate of 24 kHz, and downsample to 12 kHz, in other words by a ratio of 2. One way to get there would be as follows:

```
fstart=24000;up=1;down=2;fup=fstart*up;fdown=fup/down;fsp2=fup/2;
down2=remez(34,[0 4800/fsp2 7200/fsp2 1],[1 1 0 0],[1 100]);
fir2vsdsp(down2,16,up,down,"downBy2","-rate",fstart);
```

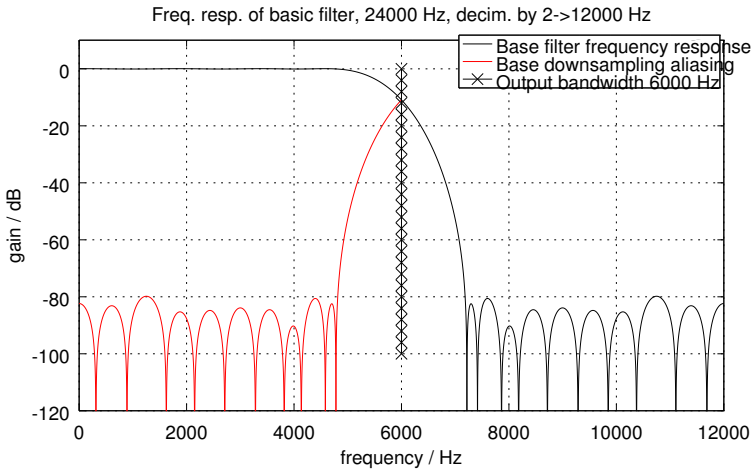

Figure 12: 24000 Hz to 12000 Hz decimation filter

Figure 12 shows the frequency response of the downsample filter. This looks similar, but is interpreted somewhat differently from the upsample filter frequency response of Figure 7.

The vertical line at 6 kHz shows the highest frequency that can properly be represented after resampling from 24 kHz to 12 kHz. Anything that is on the right side of that frequency after filtering will become aliasing distortion to the left side after resampling. This aliasing after downsampling is shown by the red line. If we zoom in the image, we can see that all aliasing above the level of -80 dB is going to end up at the frequency range of 4800-6000 Hz, which in many cases is acceptable.

6.1.4 Designing a Band-Pass Filter

A band-pass filter is a filter that lets certain frequency bands through while rejecting others.

As an example we are going to design a “telephony” band-pass filter that lets through frequencies between 300 Hz and 3400 Hz, while rejecting other frequencies. We will design this filter to operate at a sample rate of 12 kHz.

The filter design code could look like this:

```
fstart=12000;up=1;down=1;fup=fstart*up;fdown=fup/down;fsp2=fup/2;
bp=remez(90,[0 20/fsp2 400/fsp2 3200/fsp2 3600/fsp2 1],[0 0 1 1 0 0],[20 1 12]);
fir2vsdsp(bp,16,up,down,"bandPass","-rate",fstart);
```

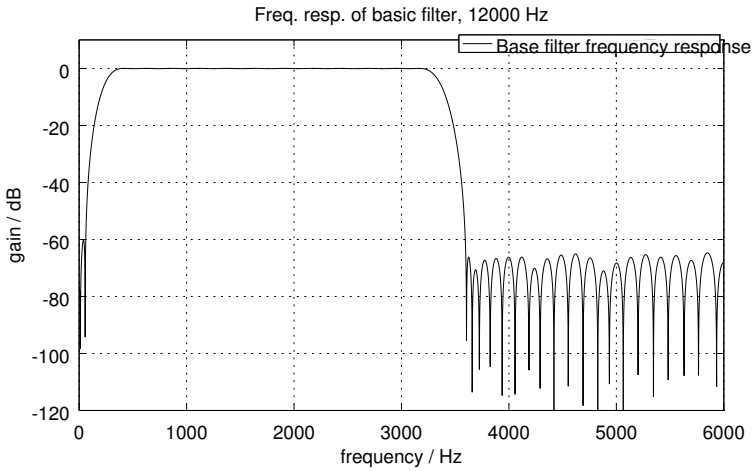

Figure 13: 300-3400 Hz band-pass filter running at 12 kHz

Figure 13 shows the frequency response of the band-pass filter. Stop-band rejection is a little over 60 dB. If we zoom closer, we will again see that the pass-band ripple is just below ± 0.05 dB.

6.1.5 Designing a Half-Band Filter

If a low-pass or a high-pass filter is designed in a certain way, it will become a half-band filter. A half-band filter is a filter where, with exception of the center coefficient, half of the coefficients are zeroes. In many cases, VLSI Solution's DSP library can take advantage of half-band filters and create filters that operate close to twice as fast as ordinary filters of similar performance.

The rules to get a half-band filter are as follows:

- The filter must have even order, in other words, odd length.
- The filter transition band must start and stop at equal distances of fs/4. Example: if sample rate fs = 8 kHz, transition band could e.g. be from 2.0-0.4 kHz = 1.6 kHz to 2.0+0.4 kHz = 2.4 kHz.
- The pass-band and stop-band must have the same weight.

An example filter design code is shown below:

```
fstart=8000;up=2;down=1;fup=fstart*up;fdown=fup/down;fsp2=fup/2;
transtart=3400;
hb=remez(62,[0 transtart/fsp2 1-transtart/fsp2 1],[1 1 0 0],[1 1],256);
fir2vsdsp(hb,16,up,down,"halfBand","-rate",fstart);
```

If you plot the filter coefficients like this, or just type out the values, you will see that half of the coefficients are zeroes (or very close to zero), with the exception of the center coefficient, which is 0.5.

```
plot(hb,'-o');grid on
hb
```

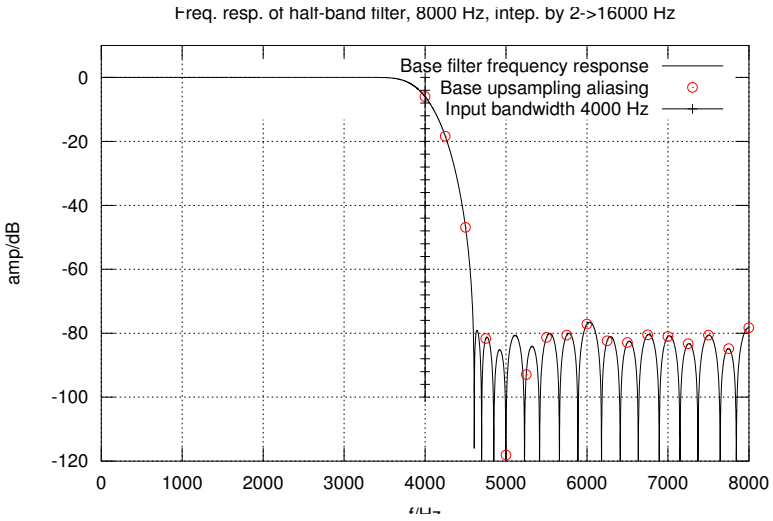

Figure 14: Half-band interpolation filter from 8 kHz to 16 kHz

Figure 14 shows the frequency response of the half-band 8 kHz to 16 kHz interpolation filter.

6.1.6 Designing a Bandwidth Splitting Filter

When you design a low-pass, high-pass, or a band-pass filter, and if your filter length is odd (the order is even), it is possible to split the audio in two parts with negligible computational cost. The first part is the filter you designed, while the second part is the “inverse” of your filter: the stop-band becomes pass-band, and vice versa.

An example filter design code for a split filter is shown below. The basis for it is a low-pass filter with a transition band from 1 kHz to 2 kHz, and a stop-band attenuation of about 70 dB.

```
fstart=8000;up=1;down=1;fup=fstart*up;fdown=fup/down;fsp2=fup/2;
split=remez(30,[0 1000/fsp2 2000/fsp2 1],[1 1 0 0],[1 1],256);
fir2vsdsp(split,16,up,down,"split","-rate",fstart,"-split");
```

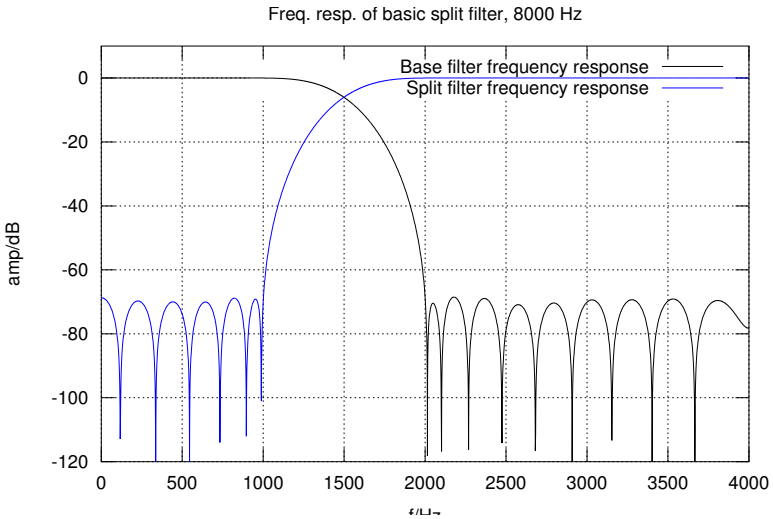

Figure 15: Split filter running at 8 kHz

Figure 14 shows the frequency response of the example split filter. Two frequency responses have been plotted: one for the filter you designed, and one for its “inverse” filter.

A useful property of a split filter is that if you add the two halves, you will get back exactly your original signal, though with some delay that depends on the length of the filter.

6.2 Converting Your Filters to VSDSP Source Code

As you have seen, whenever you design a filter using the Octave FIR2VSDSP() function, it outputs some C code to the screen. You can either copy this code to your source code, or let the function automatically prepare .c and .h files for you.

To generate C code to a file, you need to add two parameters to your call to FIR2VSDSP(). The parameters are “-file”,”fileNameBase”, where fileNameBase is the base to which the .c and .h suffixes are added.

If you want to add more than one filter in your source code, you may add the parameter “-append” to your parameters for all but the first filter. In this case, the .c and .h files are not removed. Instead the new code is appended to them.

If you need more than one channel for your filter, like for instance with stereo sound, you may define more than one filter name.

Example: we want to make a C source code for a stereo audio path that will take 24 kHz audio, resample it to 12 kHz (Chapter 6.1.3), and then band-pass filter it to the telephone bandwidth of 300-3400 Hz (Chapter 6.1.4). It is done as follows:

```
fir2vsdsp(down2,16,1,2,"downBy2L","downBy2R","-rate",24000,
          "-file","telephony");
fir2vsdsp(bp,16,1,1,"bandPassL","bandPassR","-rate",12000,
          "-file","telephony","-append");
```

Now you should have two files, telephony.c and telephony.h in your current folder (under Cygwin the home folder may be e.g. *C:\cygwin64\home\tardis3*). Of these two files, telephony.h should look much like this:

```
/*
   This is an automatically generated file by VLSI Solution's
   Octave program fir2vsdsp.m.

   DO NOT MODIFY unless you ABSOLUTELY know what you are doing!
*/

#ifndef __FIR_TELEPHONY_H__
#define __FIR_TELEPHONY_H__

#include <vstypes.h>
#include <fir.h>

extern struct FirFilter __mem_y downBy3L;
extern struct FirFilter __mem_y downBy3R;

extern struct FirFilter __mem_y bandPassL;
extern struct FirFilter __mem_y bandPassR;
```

```
#endif
```

Congratulations! You have now designed several filters and converted them into VSDSP C source code.

But how to use the code? For an explanation, continue to Chapter 7 of this tutorial.

7 Tutorial: Compiling And Linking Your Filter with VSIDE

Now it's time to take the code we created in Chapter 6, and compile it into a real project.

Before starting, make sure you have the latest version of VSIDE!

In this package, there is an example VSIDE Solution for each VS10xx IC, using the downsampling-by-2 + bandpass *telephony* example built in Chapter 6.2. The associated files are *telephony.c* and *telephony.h*. *main.c* contains the main body of the solution, including input and output routines, and interleaving / deinterleaving stereo samples. Other files offer external support. By modifying this example, you may build your own signal processing software.

To test the example solution, you will need to connect a line input or a microphone to your VS10xx analog inputs, and earphones to VS10xx analog outputs.

7.1 Compiling and Linking for VS1005 VSOS3

This package contains the folder SigPros1005 which contains the example as an .AP3 application for VS1005 / VSOS3.

Copy the solution to your VSIDE solutions folder, then open it with VSIDE.

Compile the solution, copy the resulting .AP3 file to your VSOS system disk, then run it.

You should now hear audio as if it was coming through a telephone.

7.2 Compiling and Linking for Other VS10XX ICs

This package contains folders SigPros1063, SigPros1053, SigPros1000, SigPros1003, and SigPros1103, which contain the example as a solution compatible with that IC.

Copy the solution for your IC to your VSIDE solutions folder, then open it with VSIDE.

Compile the solution. It will create .plg and .cmd loading tables to the folder Emulation-Debug. You can now load and run the solution on your VS10xx IC, either using VSIDE and UART, or by loading the .plg file through SCI / SPI using your system microcontroller. To see how to load .plg files, read Chapter 11, *How to Load a Plugin*.

You should now hear audio as if it was coming through a telephone. Look for main.c for details on how to toggle bypass mode.

8 Function Reference

8.1 fir2vsdsp.m

B_ROUNDED = fir2vsdsp(B, BITS, UPSAMPLEBY, DOWNSAMPLEBY,...);

Creates C code for a BITS-bit FIR filter NAME that first upsamples data by UPSAMPLEBY, then filters with Fir filter B, then downsamples by DOWNSAMPLEBY. Prints the filter structure as C code. VS1005 header file <fir.h> contains definitions required for running the filters.

The function returns the vector B rounded to the accuracy that could be represented by its internal integer accuracy.

Parameters:

- B - Fir filter
- BITS - Number of bits. Currently only 16 is supported.
- UPSAMPLEBY - How many times to upsample before filtering
- DOWNSAMPLEBY - How many times to downsample after filtering
- ... - Varargs options may contain some of the following fields:
  - NAME - Name of the filter. If more than 1 channels are required, many names can be specified. There must be at least one name.
  - "-split" - If defined in the argument list, make a split filter that outputs twice the input data (e.g. low-pass and high-pass path).
  - "-rate",FS - If defined, followed by FS, the samplerate of the source signal. Example: fir2vsdsp(b,16,2,3,"myFilter","-rate",48000). In this case a frequency response of the filter is plotted. If a split filter, frequency response of both paths is plotted.
  - "-magnify" - Show Y scale of -2 to 2 dB instead of the default.
  - "-file",NAME - Write source code to file NAME.c and NAME.h.
  - "-append" - If writing to a file, append the code the files instead of creating the files.

Example:

```
fir2vsdsp(b, 16, 1, 2, "myFilterLeft", "myFilterRight",
          "-rate", 48000, "-file", "MyFilter");
```

9 Installing GNU Octave + VSIDE DSP Library, Windows 8.1

This Chapter explains how to install GNU Octave under Windows 8.1 so that it will be ready for use by the VSIDE DSP Library functions.

Note that there are at least two Windows installation options for GNU Octave: Cygwin, or MinGW. These instructions have been tested with the Cygwin version on 2015-04-29.

1. Go to *https://cygwin.com/install.html*. There, depending on your Windows version, select either the 32-bit or 64-bit installer.
2. When the installer asks to *Select packages for install*, select the following packages:
   (a) Math/gnuplot
   (b) Math/octave
   (c) Math/octave-control
   (d) Math/octave-general
   (e) Math/octave-signal
   (f) Text/texinfo
   (g) X11/xinit
   (h) X11/xlaunch
3. Accept all dependencies that the installer asks from you and proceed with the install. At the time of writing (2015-04-29), this will install Octave 3.8.2-1.
4. After the install, you will have a new Cygwin (or Cygwin64) Terminal icon / start menu item.
5. Open the Cygwin home folder (e.g. *C:\cygwin64\home\tardis3*). Copy the VSIDE DSP Library package files/folders *.octaverc* and *octave* to the home folder.
6. Start the Cygwin Terminal.
7. In the Cygwin Terminal, type *xlaunch*.
8. In XLaunch, usually the *Multiple windows* option is the best. In the next question screen, select *Start a Program*, and in the next one, select the program to be *xterm*. Accept all other question pages.
9. You will be presented with an *XTerm* terminal. Here, type *octave*.
10. Test that Octave and GnuPlot work. Enter e.g. command *plot(sin(0:0.1:2*pi))*. You should get a sine waveform plotted in a new window.
11. Test if you have the signal processing package installed properly by entering command *help remez*. If you get help, all is ok.
12. Test if you have the VSIDE DSP Library package installed properly by entering command *help fir2vsdsp*. If you get help, all is ok.
13. You are now ready to run the exercises in this document!

10 Troubleshooting

10.1 VS1005 voplinkg Linker Error

Symptoms:
A linker error message that looks like below:

```
ERROR: "_FirFilter16x16HBUp1Dn2" is needed but not found anywhere.
```

Reason:
The DSP Library or General Function Library is missing from the linker directives.

Solution:
Right-click on your project, then select Properties (See Figure 16) -> Linker -> Libraries. Add “-ldsp -lgeneral” to the end of that line.

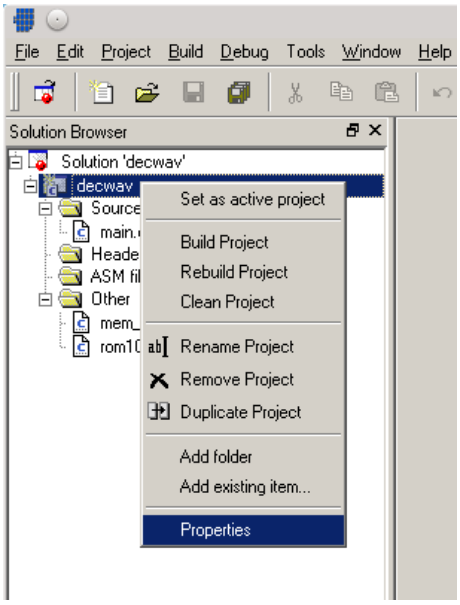

Figure 16: VSIDE project Properties menu selection

10.2 VS10xx vslink Linker Error

Symptoms:
A linker error message that looks like below:

```
ERROR: vslink: Undefined symbol _FirFilter16x16NUpDn in init_y
D:\VSIDE\bin\make.exe: *** [Emulation-Debug/Dsp1063.coff] Error 10

Build failed!
```

Reason:
The DSP Library or General Function Library is missing from the linker directives.

Solution:
Right-click on your project, then select Properties (See Figure 16) -> Linker -> Libraries. Add “-ldsp -lgeneral” to the end of that line.

10.3 Missing Octave Function Error when Calling ELLIP()

Symptoms:
When calling function ELLIP, an error message that looks like below:

```
error: ‘fminbnd’ undefined near line 37 column 4
error: called from:
[... additional lines cut ...]
```

Reason:
In the current Octave FMINBND has been moved into the core. Some Linux installations are running the old version of Octave but with the new libraries that are missing the function. Happens at least in Kubuntu 12.04 LTS.

Solution:
Downgrade your Octave libraries or upgrade your Octave. May require upgrading the whole OS to newer. (E.g. Kubuntu 14.04 LTS contains a working version.)

10.4 Missing Octave Function REMEZ() or ELLIP()

Symptoms:
When running Octave under Cygwin, and When calling function REMEZ() or ELLIP(), the function is not found.

Reason:
If running Octave under Cygwin, the signal processing package is not automatically loaded, and you have not copied the example .octaverc contained in this package to the correct location.

Solution:
To load all packages, enter the following command in Octave:

```
pkg load all
```

11 How to Load a Plugin

11.1 How to Load a .PLG File

A plugin file (.plg) contains a data file that contains one unsigned 16-bit vector called plugin. The file is in an interleaved and RLE compressed format. An example of a plugin vector is:

```
const unsigned short plugin[10] = { /* Compressed plugin */
  0x0007, 0x0001, 0x8260,
  0x0006, 0x0002, 0x1234, 0x5678,
  0x0006, 0x8004, 0xabcd,
};
```

The vector is decoded as follows:

1. Read register address number `addr` and repeat number `n`.
2. If `n` & 0x8000U, write the next word `n` times to register `addr`.
3. Else write next `n` words to register `addr`.
4. Continue until table has been exhausted.

The example vector first tells to write 0x8260 to register 7. Then write 2 words, 0x1234 and 0x5678, to register 6. Finally, write 0xabcd 4 times to register 6.

Assuming the vector is in vector `plugin[]`, a full decoder in C language is provided below:

```
void WriteVS10xxRegister(unsigned short addr, unsigned short value);

void LoadUserCode(void) {
  int i = 0;

  while (i<sizeof(plugin)/sizeof(plugin[0])) {
    unsigned short addr, n, val;
    addr = plugin[i++];
    n = plugin[i++];
    if (n & 0x8000U) { /* RLE run, replicate n samples */
      n &= 0x7FFF;
      val = plugin[i++];
      while (n--) {
        WriteVS10xxRegister(addr, val); /* You need to implement this func. */
      }
    } else {           /* Copy run, copy n samples */
      while (n--) {
        val = plugin[i++];
        WriteVS10xxRegister(addr, val); /* You need to implement this func. */
      }
    }
  }
}
```

12 Contact Information

VLSI Solution Oy
Entrance G, 2nd floor
Hermiankatu 8
FI-33720 Tampere
FINLAND

URL: http://www.vlsi.fi/
Phone: +358-50-462-3200
Commercial e-mail: sales@vlsi.fi

For technical support or suggestions regarding this document, please participate at
http://www.vsdsp-forum.com/
For confidential technical discussions, contact
support@vlsi.fi

Note: for discussions of the VSIDE DSP Library, using the “DSP and Audio Software” section of VSDSP Forum is recommended.