

ECE-486, Spring 2008

Instructions for Real-Time Audio Signal Processing Under Linux

Don Hummels

January 30, 2008

1 Introduction

This document describes the use of the “dsp486” command to develop and implement real-time DSP algorithms which access the soundcard in a Linux environment. The interface has been developed to support ECE-486 Labs at the University of Maine.

2 Overview

In general, you are expected to develop “plugins” which are loaded and executed by the “dsp486” command. The “dsp486” command continually captures sampled data from the “line-in” terminal of the soundcard, passes the data through your plugin, and writes your calculated results back to the soundcard to drive the analog “line-out” terminal.

Essentially, a “plugin” is a set of subroutines (which you write) that are loaded when the “dsp486” command is executed. Your code should define the required subroutines — It should *not* define the `main()` function. “dsp486” retrieves the sampled data from the soundcard in consecutive blocks, and calls your plugin routine once per data block. For example, for a sampling frequency of $F_s = 48000$ samples/sec, if “dsp486” selects a blocksize of 4800 samples, your plugin function will be called 10 times/sec. Your plugin routine must complete processing each block of 4800 samples within 100 msec, or input data blocks will be missed, and discontinuities in the output data stream will result.

3 Cautions...

- Please be careful in connecting any audio source or function generator to the line-in connector of the soundcard. If you’re not sure which connector to use, please ask.

- Keep your input voltage levels reasonable. You should not need to drive the line-in terminal with anything over ± 1 volt, and generally one-tenth of this should be adequate for this class. Your input signal should not contain any DC offset.
- Be aware that the soundcard may alter your input waveform to some extent. Some DC blocking is apparent (either on the input or output signal paths), so you should not expect to measure system responses right down to DC. Also, the sampler on the input signal path includes an anti-aliasing filter which begins to cut off as the folding frequency of the converter is approached.
- This interface utilizes large buffers of data, and so introduces significant time delays between the system input and the system output. Delays of 1/3 second and above are common. As a result,
 - It will be difficult to measure the phase characteristics of your discrete-time system designs. The phase response tends to be masked by the large phase shifts associated with the system delays.
 - Care should be used in characterizing the system response using some instruments. For example, a dynamic signal analyzer set to sweep through a range of frequencies may have difficulty, since it tends to move to the next frequency step before the response from the current frequency has arrived at the output. You may need to use the random input setting, instead of the swept input frequency.

4 Compiling and Using the Plugin

The following steps are needed to compile and run a real time DSP algorithm using the “dsp486” interface:

1. Download the “dsp.c”, “dsp.h”, and “Makefile” template files from the course WWW site.
2. “dsp.c” is a C file that you will modify to include your processing. **Be careful not to change the function interfaces.** Section 5 describes the functions which must be present in your file.
3. Compile the plugin by typing “make”. If successful, this should create the plugin (shared object file) “libdsp.so”.
4. Type “dsp486”. This should run the “dsp486” program, which will look for your plugin in the current directory. The program will begin reading from the soundcard, calling your plugin routines, and writing results back to the soundcard. Some status messages should be displayed on the terminal, including estimates of the CPU usage as the plugin is called, the average input/output buffer size, and the maximum allowed time for processing each buffer of data (which depends on the F_s used).

Options may be passed to “dsp486” (and to your plugin) by changing the command in step 4. A collection of integer parameters may be passed to your plugin routine using the `-dsp` argument. For example,

```
dsp486 -dsp=500,3,2
```

results in an array of integers (500, 3, and 2) being passed to the plugin initialization routine (Section 5). In addition, any parameters interpreted by the “arecord” or “aplay” functions may be specified. For example, to run your program using a soundcard sample frequency of 8 ksp/s, use

```
dsp486 -dsp=500,3,2 -r 8000
```

If no rate is specified, a default value of $F_s = 48000$ samples/sec is assumed.

5 Plugin Interface

Your plugin must implement the `dsp_construct()`, `dsp_process()`, and `dsp_destroy()` functions. The required interface to these functions is described in the following subsections.

5.1 System Initialization: `dsp_construct()`

The `dsp_construct()` function will be called once, at the beginning of “dsp486” execution. You should use this function to perform any required initialization or memory allocation. The interface is defined by the function prototype

```
void dsp_construct(int count, int * vals, size_t c_buf_size);
```

The input parameters `count` and `vals` are used by the “dsp486” command to provide the array of integers specified on the “dsp486” command line to your plugin routine. For example, if “dsp486 -dsp=500,3,2” is typed on the command-line, then `dsp_construct()` will be called with `count` set to 3, and `vals` will be a pointer to the array of integers (500,3,2). The `c_buf_size` argument will contain the *maximum* block size of samples which will be handed to your routine. This value will depend upon the particular soundcard and sample rate (among other things).

5.2 Signal Processing: `dsp_process()`

The `dsp_process()` function will be repeatedly called by “dsp486” once for every new block of input data. All “real-time” code resides in this function. The function prototype is

```
int dsp_process(short * ibuf, size_t isize,
               short * obuf, size_t * osize);
```

Table 1: Input sample array provided to `dsp_process()`

Array Entry	Value
<code>ibuf[0]</code>	1 st Sample, Left Channel
<code>ibuf[1]</code>	1 st Sample, Right Channel
<code>ibuf[2]</code>	2 nd Sample, Left Channel
<code>ibuf[3]</code>	2 nd Sample, Right Channel
<code>ibuf[4]</code>	3 rd Sample, Left Channel
<code>ibuf[5]</code>	3 rd Sample, Right Channel
⋮	
<code>ibuf[isize-2]</code>	Last Sample, Left Channel
<code>ibuf[isize-1]</code>	Last Sample, Right Channel

Sampled data from the ADC is represented by signed 16-bit (short) integers. The input parameter `ibuf` points to the array containing the samples from the left and right channels of soundcard “line-in” connector. These samples are interleaved, so that even indexes provide the left-channel samples, while odd indexes provide the right-channel samples.

The *total* number of samples (left and right) passed to the `dsp_process()` function is provided by the `isize` parameter. For example, for `isize=1024`, there are 512 samples from left channel, interleaved with 512 samples from the right channel.

The structure of input sample array is shown in Table 1.

Your plugin should write the resulting output sample sequence to the `obuf` array, and should set `*osize` to the number of output samples generated (usually the same value as `isize`). Once again, the output array is assumed to be interleaved left/right samples, with exactly the same structure as for `ibuf`. The output samples are also assumed to be 16-bit signed (short) integers.

To continue processing blocks of data, `dsp_process()` should return the number of output samples generated. To end execution of “dsp486” a value of “-1” should be returned.

The “`dsp.c`” code from the ECE-486 web site contains example code that separates samples into left and right channels. The data from the right channel is scaled and passed directly to the right channel of the output buffer. The absolute value of each sample of the left channel is written to the left-channel outputs, resulting in a full-wave rectified output signal.

5.3 Cleanup: `dsp_destroy()`

The `dsp_destroy()` function is called once, as the “dsp486” process terminates. It provides an opportunity for your plugin to free any allocated system resources. In particular, you should free any allocated memory, and close any open files. There are no input or output arguments to the function.

6 Mixer Settings

The “dsp486” program accesses the PC soundcard using the open-source ALSA audio interface. The “alsamixer” program must be used to configure the soundcard to allow recording the line-in terminal while driving the line-out terminal. An appropriate soundcard setting state for the ECE cluster computers has been created and saved to the `asound.state` file available on the course web site. By downloading this file, the soundcard settings can be restored by opening a terminal window, changing directories to the location of the file, and typing

```
/sbin/alsactl restore -f asound.state
```

7 Running “dsp486” on Your own Computer

The “dsp486” interface was developed under Fedora Core, and should be compatible with the corresponding linux kernel and ALSA driver versions. We can provide the source code with a Makefile for the interface. Compile the source code by typing “make”. Either directly run “dsp486” in the directory in which you compiled it, or copy the executable to a location included in your path.

Your mileage for may vary, depending on your sound card and kernel version. You’ll probably have to experiment with the ALSA mixer settings for your sound card. But it’s worth a try. Let us know how it works out!

8 If things go wrong...

If things are not working, try the following:

- Double check the cables connected to line-in and line-out.
- Check that the soundcard interface is working (without using the “dsp486” program). An easy way to send random data to the soundcard is to use a command like

```
cat /bin/ls > /dev/audio
```

A random noise signal should be showing up on the line-out terminal. If this does not work, don’t waste time checking your plugin routine. Double check the cables and the mixer settings. Try another computer (in case the soundcard is dead).

- `ddd` is a useful symbolic debugger. By default, “Makefile” is set up to enable debugging. To start the debugger for your compiled shared object file, use

```
ddd libdsp.so
```

Then, within the `ddd` environment, set the desired breakpoints and type “run dsp486 -dsp=...”.

- Starting “dsp486” actually starts multiple processes which communicate with each other. If (when?) your program “crashes”, it’s possible that some of these processes will be left running, even though the “dsp486” command has apparently terminated. Eventually, these stranded processes can become a drain on the system. To clean out any stranded processes, type

```
killall dsp486
```