DSPExplorer – A Learning Environment for Digital Signal Processing

Ward Harriman - AE6TY ae6ty@arrl.net

Introduction

Several years ago I embarked on a self-guided attempt to learn about Software Defined Radios. I was comfortable with simple RF design so I could implement the RF front end and I was comfortable with programming so I expected I could 'write the code'. My biggest hurdle was in understanding and applying the principles of Digital Signal Processing. This was a completely new discipline for me as I had only the most tenuous memory of learning about it some 30 years before. The memories were not good but I was convinced that I now had the time and the interest to gain at least a passing understanding of this important technology.

As I read about the principles and techniques of DSP I found myself wanting to try my hand at actually writing code. For me, writing functional code was a kind of trial by fire. If I could write functional code then I believed I had a passing understanding of the principles involved. Unfortunately, in order to prove a program was functional I had to run it and to run a program, I had to have some input and I had to be able to verify the output. At this point, my progress fell precipitously because I could find no easy way to generate input and verify output and hook everything together.

My first attempt at writing programs came in the form of Excel[1] spreadsheets. Excel is very good at performing computations and graphing the outputs. Unfortunately, it is not so good at generating input and it is a very poor programming language for DSP applications.

My second attempt was to use C and write programs to generate input and to graph the output. This approach allowed me to generate sophisticated input, write my test program efficiently and graph the output… more or less. While functional, this approach did not lend itself to general utility. C is a great programming language for DSP work but without self-discipline the programs quickly degrade to a heap of spaghetti. I was beginning to consider the idea of developing a methodology for teaching DSP principles and unadorned C simply wasn't a good vehicle. I believed the newcomer would quickly be lost in the nuances of C programming rather than in the topic of interest: DSP programming.

What I needed was a kind of "Integrated Development Environment" or IDE which would provide a complete environment for demonstrating DSP principles. The requirements I envisioned were:

- Provide a simple way to generate input of many forms

[1] Excel is Microsoft Spreadsheet calculator much like Numbers from Apple. Both Excel and Numbers are, undoubtedly, trademarked.

- Allow the user to write programs in a C like language
- Provide a simple way to connect segments of code together
- Display both inputs and outputs and any intermediate 'signals'
- Run on three different platforms: Windows, OS-X and Linux (of some form).
- Be free to anyone who wanted to use it.

I looked around the web for some time but could not find any applications of this form. This is not to say they aren't there, I just failed to find one. The engineering applications typically used for these functions failed on several counts. First, they were far from free. Second, there was a substantial learning curve involved with using them. And so, I set off to make an application of my own.

What follows is an introduction to my IDE I named "DSPExplorer". Unfortunately, there is no getting around it: to get started in Digital Signal Processing, you have to read and eventually write programs. Fortunately, the programs are generally quite simple and certainly do NOT require anything even remotely sophisticated. All the programming skill you'll need to get through this paper is generally taught in the first week of a middle-school course on programming. If you aren't a programmer, here's your chance to start!

NOTE: This short paper does not discuss how to download or invoke DSPExplorer. That information is best conveyed using short video tutorials on my website, www.ae6ty.com, where you can also download DSPExplorer. Here, I will try to show the basic capabilities of DSPExplorer and, hopefully, pique the reader's interest enough to download it and give it a try.

DSP Explorer

When I started to write DSPExplorer I had recently finished up my Smith Chart software (SimSmith, also available at www.ae6ty.com) and was flush with confidence. After some casting about, I ultimately decided that there would be four fundamental tools in my IDE. First, I would need a text editor. The text editor didn't need to be very sophisticated since most DSP program segments are quite short.

Second, I would need an interpreter or compiler. Unfortunately, many computers do not come with C compilers already installed. Rather than require the user to go and fetch and install a compiler, I decided to provide one built in. My compiler is not exactly C, more of a 'C-lite' but it is adequate for these purposes.

Third, I would need to provide a way for the user to structure the program and to reuse blocks. I decided that representing modularity in DSP code was best done using schematics. Thus, I would need a simple schematic capture capability where the user would 'wire together' blocks of code.

Fourth, I would need a familiar output mechanism. Here, I decided to provide a very 'oscilloscope' like function. In addition to basic signal traces, the output subsystem should also provide 'spectrum analyzer' type functions.

Picture 1 shows the default screen layout of DSPExplorer. In this screen I have typed a program into the upper right hand window. Note that I have purposefully introduced an error. When the compiler encounters an error it endeavors to highlight (make red and underscore) the item it finds confusing. It doesn't always get it right, but it gets close. Additionally, the compiler attempts to describe the nature of the error in the 'error' window.

The window on the upper left hand side is the schematic capture window. There is only one block in our example so far and so only one block is shown. Note that there is a red X drawn through the box because there is an error in the associated program. The block also shows a single output called 's'.

The lower left hand window is the oscilloscope (and spectrum analyzer) window. In this case I have chosen the variable 's' as the trigger. This window can display any variable in any module. There is no limit on the number of traces which can be shown. Triggering can take the familiar forms of positive or negative edges and the 'level' can be adjusted using the 'L>' slider. The default is positive edges passing through 0.

The lower right hand window displays text output. There are three different 'panes' in this window. The 'error' pane shows error messages from the compiler. The 'console' pane shows messages which come from the program. Messages can be sent to the 'console' using the "println()" function call from the written program. Finally, the 'log' pane is used by DSPExplorer to inform the user of various internal

events. Most DSPExplorer functions take place quietly in the background and the 'log' pane is rarely used.

Picture 2 shows the screen after I have corrected the error in the 'first' module, selected the "Simulate" menu and then clicked on "Run". Notice that there is now a trace on the oscilloscope and that the 'console' pane has been displayed in the lower right hand corner.

This simple example shows how the four basic windows are used in concert to develop a program using DSPExplorer. Everything that follows is really fleshing out the capabilities provided.

Module Creation

Modules are created by simply clicking on 'Module' and then 'New'. DSPExplorer will ask you for a name. Once you have provided the name, DSPExplorer will create a schematic module of the same name and open the text editor so that you can write your code.

Every time you make a modification to a module, the schematic, or the 'scope, DSPExplorer updates the associated files and attempts to compile the code. When you exit DSPExplorer and then restart it, DSPExplorer endeavors to restore the design to the same state as when you exited. It doesn't do everything perfectly, but it tries!

Variables

DSPExplorer provides two kinds of variables: double and complex. A 'double' is represented internally as a 64 bit, IEEE-754 floating-point number. A 'complex' is two doubles, one used to hold the 'real' part and one to hold the 'imaginary' part. All variables must be declared before use and are initialized to zero by the compiler. All variables are 'static' in that they maintain their values between calls.

DSPExplorer also provides for single dimensional arrays. Array indexes are always positive and always start with 0; the '0'th element is the 'first'. Essentially all programming languages implement arrays in this way. DSPExplorer allow for indexes outside the obvious range. For example, element "-1" is the last element in the array. This is useful when implementing ring buffers. Sample declarations are:

```
double d;
complex c;
double  da[3];
complex  ca[4];
```

Basic variables and arrays can be accessed only in the modules in which they are declared and only after they have been declared.

It is almost always the case that a module will have some inputs and some outputs. There is no 'equivalent' concept in C so I invented my own syntax. Inputs and outputs can be arrays as well as basic variables. Some examples are:

```
input double ind;
input complex inc;
input double inda[12];
output complex[5];
```

It is often the case that when you write a module you will have an input which you know will be an array but you won't know what size until it is wired to another module. In this case you can write:

```
input double inda[];
```

There is one final variable type, the 'parameter', which can be used to initialize a variable via a value in the schematic. Picture 3 shows how we can use parameters to provide some constants to the compiler. Notice how the 'parameter' initial values are reflected on the schematic. Of course, the value on the schematic can be altered… that is the whole point!

This variable can only be a 'double' and is declared as:

```
parameter double aparam;
```

Note that a 'parameter' is a compile time variable. Changing the value on the schematic will halt simulation and cause the module to be recompiled.

One final thing about variables… variable declaration can be combined with assignment. This is really just a typing shortcut but can prove very useful. You already saw this in the 'first.mod' program shown in pictures 1 and 2. This assignment should not be confused with 'initialization', the assignment is executed every time the module is called.

Hold on here, we'll write a program in a moment.

Multiple Modules

You can create new modules at will. DSPExplorer will maintain text files corresponding to each of the modules. Additionally, DSPExplorer will maintain a text file which contains the state of the schematic and scope panes. Let's create the two modules shown in picture 3. The first module will simply take two parameters and construct a 'complex' output. The second module will compute the magnitude complex number and print out the result.

To run the program in Picture 3 simply click on “Simulation” and “Single”. We use “Single” because we want to execute the program only once. If you hit “Run”, DSPExplorer will continuously execute the schematic until you hit “Abort” or “Halt”.

The source code for “second.mod” is:

```
parameter double r = 1;
parameter double i = 0;

output complex o = complex(r,i);
```

Note how the complex output is constructed using a call to the function “complex(r,i);”. No magic here, DSPExplorer simply takes two doubles and creates a complex number, then assigns the number to the ‘o’.

In ‘magphz.mod’ we show how to take apart complex numbers. The ‘real()’ and ‘imag()’ functions do the obvious thing. We then use a ‘println’ to show the arguments and the results.

One very important characteristic of DSPExplorer is shown in the result of the println. Specifically, the result of the sqrt() function is a complex number which is denoted by the “{5,0}”. Indeed, essentially every function provided by DSPExplorer returns a complex value. When you assign a complex number to a double, the double takes on the value of the ‘real’ part of the complex number. Thus:

```
double r = real(in);
```

could have been written:

```
double r = in;
```

A Simple DSP Example

We’ve now seen enough of DSPExplorer to write a useful demonstration program. You can follow along in Figure 4.

First, let’s write a more sophisticated signal generator which will create an exactly periodic signal. Let’s modify ‘first’ so that it will take two parameters which will control the frequency of the output sin wave. We’ll call this module ‘exact’:

```
parameter cycles = 1;          //how many cycles in a period.
parameter period = 128;        // how long  the period is.

double phase;
phase = phase + cycles/period;
output double s = sin(phase*2*3.14159);
```

Then, let's write a program which will mix two inputs. This module will be called 'mix'. Mixing, of course, is simple multiplication:

```
input double a;
input double b;
output double p = a*b;
```

Picture 4 shows the result of the mix. Look closely at the 'scope' section of the picture. I have displayed the two inputs and the mix signal as normal oscilloscope type traces. Additionally, I show the spectrum of the 'mix' signal. As expected, there is a spectral component at the sum and at the difference of the two input signals. Note that because the signals are 'real' signals, there is a 'positive' and a 'negative' frequency component to each signal… a total of four peaks.

As another DSP example, let us see what happens when replace all the variables in 'exact' and 'mix' with complex numbers. Exact is now:

```
parameter cycles = 1;
parameter period = 128;
double phase;
phase = phase + cycles/period;
double r = sin(phase*2*3.14159);
double i = cos(phase*2*3.14159);
output complex s = complex(r,i);
```

In Picture 5 the exact.s and exact0.s signals are displayed as 'complex' traces; the real part is a solid line and the imaginary part is a dotted line. Note now that the spectrum of the 'mix' signal now has just a single peak. This is what happens when we mix 'complex' or 'analytic' signals rather than 'real' signals. (An 'analytic' signal is the same as the familiar 'I/Q' signal seen in most SDR down converters.)

The raggedness seen on the traces in Picture5 are the result of digital sampling of the ideal waveform. To see what the 'software' sees, DSPExplorer allows you to display a signal as 'bars' rather than as 'lines'. The 'mix.p' signal in Picture 5 shows just such a display.

A Complete DSP Example

A complete exploration of all the capabilities of DSPExplorer is far beyond the scope of this short paper. Still, it is useful to see an example most of us can relate to. For this application I chose to show the components of a simple phasing rig. The DSPExplorer screen is shown in picture 6.

In this example, my input is a more generic signal generator called 'sigGen'. SigGen generates an I/Q pair. It continuously sweeps the frequency starting at the 'lwr'

frequency and working its way up to the 'upr' frequency and then back down. By adjusting the 'inc' value, the sweep rate can be adjusted. The 'ramp' signal allows one to shape the amplitude of the signal as a function of frequency. I don't use this feature here which is why it is '0'.

The output of the sigGen is essentially what would come from the familiar I/Q down converting mixer. As you can see, the 'Q' channel passes through a shift register (shiftReg). The effect of the shift register is to simply delay the signal by 64 samples. This is used to compensate for the delay in the 'I' path.

The 'I' path is where most of the processing takes place. The I channel is buffered using the shiftReg module. The shift register is 128 entries long to match the length of the Hilbert Transform. The 'mem' output of shiftReg is the array of samples.

The Hilbert block generates a Hilbert Transform kernel which is 128 samples long. The purpose of the Hilbert Transform is to provide a phase shift of 90 degrees for every frequency. This kernel is passed through the Hamming window function. The Hamming window will reduce the ripple introduced by Hilbert Transform.

The work is done in the 'convolve' block. In essence, the convolve block computes the dot product of the two input vectors. The code is shown here:

```
input a[];
input b[];
output o;

double sum = 0;
for (double n = 0; n<sizeof(a);n=n+1)
   sum = sum + a[n]*b[n];
o = sum;
```

This function is such a common operation that DSPExplorer provides a builtin in function to simplify the coding. The for loop above could be written "o = dot(a,b);".

The output of the convolve block is the I channel with a 90 degree phase shift applied. The upper and lower side bands are then easily computed. The upper sideband is the sum of the delayed Q and Hilbert Transform of I. The lower sideband is the difference.

The scope in picture 6 shows four traces. The topmost trace is shiftReg0.t which is simply a delayed version of the Q channel. (There was no particular reason for choosing this signal.) The next trace down is the spectrum of sigGen.s. When the peak is to the right of center, the frequency is positive and when the peak is to the left, the frequency is 'negative'.

The next trace down is the math.add or upper sideband. As you can see, there is a signal there since the peak in sigGen.s is on the right hand side of the screen. The bottom trace is math.sub which is the lower sideband. Math.sub is a flat line indicating that there is no signal on the lower sideband of sigGen.s.

As sigGen sweeps the frequency up and down, the upper and lower sideband signals alternate indicating that the phasing rig is working properly. For completeness, picture 7 shows a lower sideband signal.

Other Experiments

DSPExplorer has been used to examine a wide range of DSP applications. A few are listed here:

- Demonstrate the effect and utility of various windowing functions.
- Show how Quadrature Error Correction can be done.
- Demonstrate alternate phasing architectures which don't use the Hilbert Transform.
- Explore implementations of Digital Automatic Gain Control.
- Demonstrate different ways Window kernels can be generated.
- Show how using the FFT and inverse FFT can replace the Hilbert Transform and the convolve function.
- Demonstrate how to use the FFT and the IFFT to perform more precise spectrum analysis.
- Demonstrate up and down conversion using decimation rather than multiplication.
- Demonstrate other filter techniques such as Frequency Sampling, Comb, and Infinite Impulse Response filters.
- Show how the Sliding DFT and single frequency detectors function.

Summary

When I started to try to learn about Software Defined Radio I found that there was no easy way to verify my understanding without actually writing programs. But when I started to write programs I found myself lost in the intricacies of programming inputs, tests, and graphing outputs. When I went looking for a better way, I found none and I was driven to develop an Integrated Development Environment. The hope was that the IDE would provide a generic framework for DSP program development and that this generic framework would allow me to concentrate on my programs. DSPExplorer was born.

DSPExplorer is a fairly comprehensive framework for developing DSP software techniques. It allows the user to generate nearly any test signal simply by writing a program to generate the signal. Blocks of DSP code can be written in a C-like language and connected together using a simple schematic capture tool. All program variables can be displayed in 'time' or in 'frequency' using a simple

'oscilloscope'. DSPExplorer is completely self contained and requires no external hardware or software to operate. It will operate on any system which supports Java.

My hope is that by providing a self contained development environment I can entice the reader to try writing a few DSP programs. Nothing teaches like doing, and DSPExplorer makes doing so much easier! I urge you to download it at www.ae6ty.com and give it a try.