Making a Software Defined Radio for the QRP Enthusiast—Part II

Ward Harriman—AE6TY ae6ty@arrl.net

In the previous issue we discussed development of the hardware for a self-contained (no PC) SDR radio. Here, we'll explore the development of the first workable software to be used on this hardware. I'm going to do something a little unusual in the article. I'm going to describe some things that worked AND some things that DIDN'T work. I do this for two reasons. First, the goal of this paper is to encourage the reader to learn through experimentation and even failed experiments teach us something. Second, I hope to save you some grief by showing where "obvious" approaches turn out to be harder than I expected.

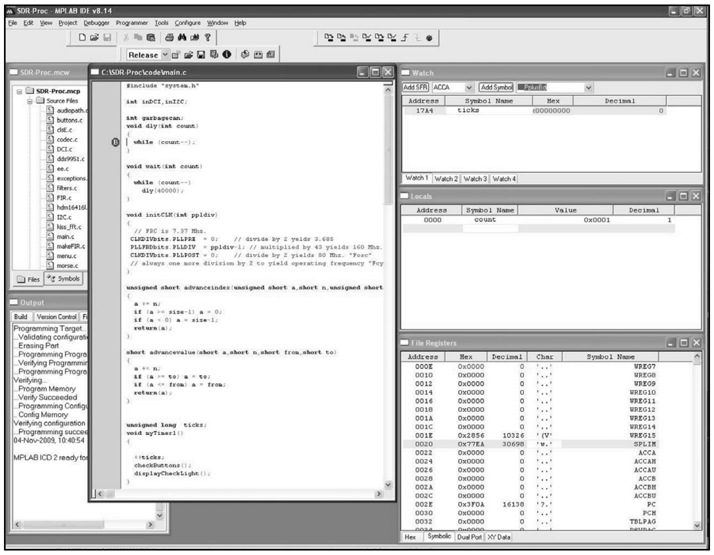
Before delving in, I'd like to point out that my SDR project is a work in progress. If you go look at my schematics or my PC board layouts or my source code (all of which are on the QRP Quarterly website), things are rather haphazard. You are not going to be seeing a completed project with assembly instructions and parts lists and code documented to commercial standards. Indeed, the software in particular is the coding equivalent of "ugly construction." My hope is that by providing all this information you will be able to poke around and see how I solved any particular problem. Please recognize that I didn't set out to build a world-class rig, I set out to build a usable rig and to explore an emerging technology. I hope what follows will encourage you to do the same.

As for the rest of this article, I won't be describing all of the inner workings of my software. Rather, I hope to explain some of the main problems I ran into and how I went about solving them. Some of these problems were logistical and needed only a modest effort to solve once identified. Other problems took me a while to understand and solve in my own modest ways. I expect this project to continue more or less forever. I guess it is my own version of "the Unfinished."

Microchip IDE

As described in the previous article, this project uses a dsPIC processor from Microchip. One of the big reasons for choosing this processor family was the software development environment provided (free) by Microchip. This environment runs on a standard windows platform. It is called an "Integrated Development Environment" because it provides every tool necessary to write and debug programs for the entire microchip processor family. It provides a basic text editor and an assembler and C compiler. Once the code has been written and compiled, the IDE provides a linker and a comprehensive "standard library" of subroutines familiar to C programmers. (For those less familiar with programming, it is common practice to provide calls within a program to common subroutines. The linker and compiler ensure that the code for these common subroutines is included as a part of the software package, thus saving the programmer from having to write these common functions into his/her code.)

Once a program has compiled and linked it is downloaded to the target hardware. Once downloaded, the IDE allows the developer to monitor and modify the various program variables and



```
Programming Target...
...Validating configurati
...Erasing Part
...Programming Progra
...Verifying Programmin
...Programming Progra
Verifying...
...Program Memory
...Verify Succeeded
...Programming Configu
.. Config Memory
Verifying configuration
...Programming succee
04-Nov-2009, 10:40:54

MPLAB ICD 2 ready for
```

```c
#include "system.h"

int inDCI,inI2C;

int garbagecan;
void dly(int count)
{
  while (count--);
}

void wait(int count)
{
  while (count--)
    dly(40000);
}

void initCLK(int ppldiv)
{
  // FRC is 7.37 Mhz.
 CLKDIVbits.PLLPRE  = 0;    // divide by 2 yelds 3.685
 PLLFBDbits.PLLDIV  = ppldiv-1; // multiplied by 43 yields 160 Mhz.
 CLKDIVbits.PLLPOST = 0;    // divide by 2 yields 80 Mhz. "Fosc"
 // always one more division by 2 to yield operating frequency "Fcy
}

unsigned short advanceindex(unsigned short a,short n,unsigned short
{
  a += n;
  if (a >= size-1) a = 0;
  if (a < 0) a = size-1;
  return(a);
}

short advancevalue(short a,short n,short from,short to)
{
  a += n;
  if (a >= to) a = to;
  if (a <= from) a = from;
  return(a);
}


unsigned long  ticks;
void myTimer1()
{

  ++ticks;
  checkButtons();
  displayCheckLight();
}
```

Address	Symbol Name	Hex	Decimal
17A4	ticks	x00000000	0

Address	Symbol Name	Value	Decimal
0000	count	0x0001	1

Address	Hex	Decimal	Char	Symbol Name
000E	0x0000	0	'..'	WREG7
0010	0x0000	0	'..'	WREG8
0012	0x0000	0	'..'	WREG9
0014	0x0000	0	'..'	WREG10
0016	0x0000	0	'..'	WREG11
0018	0x0000	0	'..'	WREG12
001A	0x0000	0	'..'	WREG13
001C	0x0000	0	'..'	WREG14
001E	0x2856	10326	'(V'	WREG15
0020	0x77EA	30698	'w.'	SPLIM
0022	0x0000	0	'..'	ACCA
0024	0x0000	0	'..'	ACCAH
0026	0x0000	0	'..'	ACCAU
0028	0x0000	0	'..'	ACCB
002A	0x0000	0	'..'	ACCBH
002C	0x0000	0	'..'	ACCBU
002E	0x3F0A	16138	'?.'	PC
0030	0x0000	0	'..'	PCH
0032	0x0000	0	'..'	TBLPAG

Figure 1—The Integrated Development Environment display screen.

hardware registers using symbolic names. Program execution can be controlled using breakpoints and single stepping at either the C or assembly code level.

During programming and debug, the IDE controls the processor using hardware called an "In Circuit Debugger"; Microchip calls their debugger the ICD3. The ICD3 connects to the computer using a USB interface and to the target processor using a four-wire interface. This ICD3 allows the IDE to control the processor directly on a clock tick by clock tick basis. The ICD3 is designed specifically for the Microchip product family and supports essentially any Microchip processor. (It is useless for other vendors' products.) The ICD3 costs about $190 from Microchip but there are compatible modules available on the web for less. The ICD3 represents an investment which can be used in a large variety of projects. I consider it as essential to my workbench as my 'scope and signal generator.

A very quick glance at the IDE screen may prove informative. Figure 1 shows a screen shot of the IDE which was taken during a typical debugging session. The largest window is called "main.c" and shows the actual source code. The dot on the left hand side of the window shows that a breakpoint is installed at that line. When the processor reaches that line it will halt and repaint all the windows.

To the right of the "main.c" window are three smaller windows. One monitors program variables which are allocated in memory, one monitors "local" variables which are available only in that specific subroutine and one monitors processor hardware registers. After hitting the breakpoint the developer can change variables, single step, proceed to the next breakpoint or even change the code and recompile; all without invoking any tools outside the IDE. It is truly an *Integrated* Development

Environment.

The Microchip tutorials for the IDE are fairly comprehensive and demonstrate the basic capabilities quite well. Those interested in exploring how the IDE can be used can download it, again for free, from www.microchip.com. The basic IDE download does not include the C compiler tool suite; that is an independent (but still free) download.

Getting Started: Divide and Conquer

The first step in writing any fairly ambitious piece of code is to divide the problem into manageable pieces. For this project I ended up dividing the effort into three more or less distinct pieces that interact in very limited and well-defined ways. Just as with hardware, software modules require specified interfaces so that the modification of one module does not impact the operation of any other modules. To the extent possible, each module should "hide" its internal complexity and provide a simple to understand and use interface.

Also, just as with hardware designers, programmers develop particular styles which are followed unthinkingly. There are obvious "style" characteristics such as how they format their programs, where they put their comments, how they name their variables and how they perform trivial tasks. There are hidden characteristics as well, things such as how the source code is arranged and how the programming style simplifies or complicates the debugging process.

I mention all these style issues because they actually make a huge difference to the developer in the long run. Remember: software, almost by definition, is never finished. You may think you'll remember how you structured all the code but in a few months you will probably forget a great deal. Being consistent about how you perform routine tasks will allow you to "come up to speed"much faster once you decide to add a feature or debug a problem several months from now.

A final note: when writing code one should remember most programs are not critical and need only provide basic functionality. Most code should not be clever because "clever" often translates to "obtuse." Generally speaking, code which was difficult to write will have more bugs and be more difficult to repair. Straightforward and consistent coding techniques yield working code in a shorter time and with fewer bugs along the way.

So, without further ado, let's get started. The three major pieces of code I had to write I call:

- The board support package which directly controls the hardware.
- The user interface which fields button pushes and controls the display.
- The SDR piece which processes data from and delivers data to the CODEC.

Board Support

The board support package is the very first piece of software written for any project. The various subroutines are responsible for initializing the hardware and providing low level "idealized" interfaces to the rest of the software. Let's use the I2C (or I^2C) bus master inside the dsPIC as an example:

The board support package must initialize the I2C interface that is used to monitor and control the CODEC and non-volatile memory chips. The initialization of this interface is quite straightforward and requires only a few lines of code.

Once the I2C interface is initialized it is ready to write control commands and read status from the various devices connected to the I2C bus. The I2C bus protocol is implemented in software and requires a few dozen lines of code. The board support package hides these low level details by providing two subroutines that can be called by other programs: write and read. Because there can be multiple devices connected to an I2C bus, the write and read subroutines require a device ID. Additionally, the I2C bus protocol provides for the writing and reading of multiple bytes of data in a single operation. This means that the write and read routines will need to accept a length argument as well. Here is an example of how a call to the I2CWrite routine might be appear:

```
I2CWrite(50,10,"sample string",13);
```

This call requests that 13 ASCII characters be written to device 50 starting at address 10. In this example, the 13 characters are specified using a quoted string "sample string." (The space between the words in quotes counts as one ascii character.) A possible read string routine might look like this:

```
I2CRead(50,10,buffer,12);
```

As expected, this routine would read 12 bytes of data from device 50, starting at address 10 and place the string into "buffer." So you get the idea; each piece of hardware will require some minimal initialization and a few other subroutines which allow other programs to access the hardware. Once these routines have been written they can be used without regard to their inner workings. If there is one word which describes the purpose of these low level subroutines it is this: Abstraction!

Let's return to the general discussion. A quick overview of this project's hardware yields the following list. Some of these elements are pretty obvious while others are less so. For example, it is clear that the DDS will need some kind of control. It is less clear that the processor clock rate will need setting. In most cases a review of the appropriate datasheets will yield a more or less complete list of the subsystems which need attention. In many cases, the vendor will even provide sample code for setting up and exercising a particular feature.

- Processor stack and memory initialization (provided by Microchip).
- Processor clock rate
- Processor Timers
- I2C interface
- I2S interface
- DDS control interface (no status from DDS)
- Keypad polling interface
- Knob 'interrupt on change' interface
- CODEC control/status interface
- Nonvolatile memory (NV ram)
- QSD control register support
- Power Amplifier control register support
- CW Key input

The amount of work involved with each of the above hardware subsystems varied widely. Some subsystems required a few minutes while others required an hour or two. Please note that I am a fairly experienced programmer and these time estimates are for me. For those beginning the journey that is programming, the time estimates are surely short. However, programming these types of routines is a skill quickly learned and after an evening or two the newcomer will be writing initialization code fluently.

The writing of the initialization code is often done more or less in parallel with debugging the hardware. Each part of the hardware is initialized and small test programs are written to verify proper hardware initialization, verify proper operation and verify proper understanding of the hardware functionality. Generally, these test programs morph into the subroutines provided for general use by the upper level software.

Each of the above subsystems was addressed as time and desire dictated. The majority of the board support package was written in a few weeks of spare time. Of course, as with all programs, the board support package continued to evolve as understanding and usage grew. Often times a subroutine needed to be rewritten when it became clear that it did not provide the correct functionality. For example, in the I2C routine above, the initial subroutines provided reading and writing single bytes. While programming other routines it became clear that reads and writes always required multiple bytes. Rather than call "I2CReadByte" multiple times it was easier to change I2CReadByte into I2CReadBlock.

Once all the subsystems have been initialized and debugged it was time to move on to the next phase, the user interface. But first…

Details of Microchip C

All this user interface stuff was going to be written in a high level language called "C." C is very widespread and there are countless books concerning how to program using "C." While C is a "standard," there are differences in how C works on different processors. Table 1 shows the differences in the version used by Microchip for the dsPIC product family, particularly as it applies to describing the menus I used with the SDR. Those readers not familiar with C programming (or perhaps a bit rusty) are encouraged to read the sidebar to this article, "A 'C' Primer: Describing a Menu." [see Notes]. The Microchip tutorial, available on their website, also shows how C can be used to program the dsPIC.

The User Interface

I really didn't pay much attention to the user interface early in this project. However, after only a few evenings it became clear that the user interface was going to be a big effort and would require a certain amount of planning. This was a bit discouraging because I was really trying to get some Digital Signal Processing work done. Still, a little planning and a good foundation would make subsequent development move faster and be less frustrating.

A few programming sessions trying to control the CODEC and DDS quickly taught me most of what I needed to have in a user interface. First, I expected that I would need to control, at most, a few dozen parameters so a simple list was adequate; no menus and sub-menus to complicate things. Second, in most cases, the parameter was simply a number which needed to be adjusted.

Command	Description	Example
long	Declares a variable as a 32 bit integer	long frequency;
char	Declares a variable as a 8 bit integer	char achar;
[ ]	Used in conjunction with declaration statements to declare arrays; may be empty to declare an array of unknown size (pointer)	char name[8];
" "	Used to specify string constants.	char name[4] = "freq";
*	Indicates that the following variable is a pointer rather than a regular variable.	char *p2c;
&	Indicates that the address of the following variable should be used rather than the value.	char *p2c = &foo;
[*n*]	Using the variable as an array, access the 'n'th element of that array.	p2c[1];
struct *name* { }	'Struct declares 'name' as a group of variables.	struct parameter{ char *name; long value; long minimum; long maximum; long increment; Function afunc; };
Function	User defined 'type' that I use to indicate the following variable specifies a function rather than a value.	Function aFunction;

Table 1.

Also, that number always had a "minimum" value and a "maximum" value. I decided that all parameters would be adjusted using a knob and so each parameter would need an "increment" value; how much should the parameter change when the knob is turned. And of course, other portions of the software needed to be told that the parameter had been changed. For example, here is the declaration describing the parameter freq:

```
struct parameter freq =
  { "freq",
    14000000,
    14000000,
    14349999,
    50,
    ddsUpdateFreq
  };
```

Of course, there will be many parameters. I decided to group the parameters together into an array. C allows me to declare an array of parameters in much the same way as a simple parameter and initialize that array. Let's jump to the first real menu I had to write. In this menu I had to control the frequency of the DDS and the gain of the amplifiers in the CODEC chip. The CODEC chip has an amplifier before the A->D converter and I call this "rfgain." It also has an amplifier after the D->A converter and I call this "afgain." The registers inside the CODEC that control the gain of these amplifiers can take on values from -63 to 63. Notice that I don't specify the size of the uMenu array (empty []). C figures out how big uMenu is by counting the size of the initializing data!

```
struct parameter uMenu[ ] =
  {
    {"freq",  14000000,14000000,14349999,50,ddsUpdateFreq},
    {"afgain", 0, -63,  63, 1,codecUpdateAFGain},
    {"rfgain", 0, -63,  63, 1,codecUpdateRFGain}
  };
```

After all this work I had the framework for describing the menu in my SDR. With a small amount of effort, I could now introduce a new menu item and have the user interface software display the parameter values, modify them, and let the rest of the software know that the parameter value had changed. But so far I had only described the menu, I had not yet written any code.

Implementing the User Interface

Once the structure for the menu had been defined it was time to start writing some code. The first code I had to write was a program which would control the ascii display. This display is 16 characters wide and 4 lines long. Since there would be more than 4 parameters in the menu I had to have a way to specify which menu item should be displayed on each line. To keep track of this I introduced an array called:

```
// which menu element is on which line.
long menuLines[ ] = {0,1,2,3};
```

Thus, to start out, menu element 0 would be displayed on the top line of the ascii display and menu element 3 would be on the bottom.

Next, I had to write a routine which would detect knob movements and modify the appropriate menu items. I chose to have four knobs, one for each line of the display. Each knob would control whichever parameter was displayed on the associated line. Each time a knob was turned the software would have to perform a series of actions. For example, when knob 0 was turned clockwise the software would examine the menuLines[0] entry to find out which parameter was being modified. It would then look into the menu array and find that parameter and update the value by adding the increment. It would then make sure the new value was between the minimum and maximum. Finally, it would call the callOnUpdate function and show the new parameter value on the display.

Whew!

There was one final requirement before I could get started on the real SDR software: there had to be a way to choose which parameters would be displayed on which lines of the display. In the first instance of this code I tried to make this simple. The knobs have built in buttons. I changed the knob code a little to check to see if the button was being pushed while it was turned. If the knob was being pushed then the software changed the value in the menuLines array rather than the parameter itself. Thus, by pushing knob 0 while turning it clockwise I would increment the variable menuLines[0] rather than the parameter. In that way I could select the parameter to be displayed rather than the parameter itself.

Enhancements

At this point I had a more or less workable user interface and could get back to work on my real goal which was the digital signal processing code. Before discussing that, though, a few more things need to be said about the user interface.

Above I described the FIRST cut at the user interface. It has been much enhanced since those humble beginnings but almost all enhancements have been cosmetic; better formatting of displayed lines and the ability to display the value as a word (like "USB" or "LSB") were significant efforts. I changed around how the parameters for display are chosen and how the knob operates providing dynamic increments instead of fixed increments.

After using the rig for some time I added two more important features. The first was to introduce the keypad so I could key in numbers directly; frequency is by far the most common parameter directly keyed. At first it seemed a waste to have a keypad for one parameter but other uses have been found.

The second feature I added to the user interface was full support for the non-volatile (NV) ram, a small, 8 pin memory device connected to the I2C bus. The NV ram is updated automatically every time a parameter is changed. "Non-volatile" implies that this memory holds the values in it when power is lost. It is difficult to describe how pivotal this was without experiencing it. Consider what life was like before the NV ram was supported. During operation I would make various parameter changes and occasionally I'd find a problem. I would go modify the program and recompile and download the new version. However, the parameters that I had chosen which caused the problem would be reset by this reload of the program. Then I would have to re-adjust the parameters hoping that I could remember how they had been set before I started fixing the problem. It was a very tedious process.

Once I had provided support for the NV ram, life was much easier. Upon encountering a problem I would simply fix the program (or hardware), reload the program and restart it. The parameters would be restored to their previous values. It was much easier to determine if the fix had indeed fixed the problem. At this point, I was truly ready to start the DSP part of the project.

Digital Signal Processing

It is finally time to get to work on the Signal Processing code. In the first cut at this software, the signal processing code runs "at the interrupt level" which means that each time the CODEC completes an A-to-D conversion the processor will be interrupted. The processor will stop what it is doing, read in the A-to-D conversion data, process that data and deliver new data to the D-to-A converter which drives the headphones.

In this project, samples to and from the CODEC are 16 bits numbers. In dsPIC C, a 16 bit number is called a "short." The CODEC samples represent fixed point, fractional numbers in the range $-1<\text{sample}<1$. Here's a new C trick. You can tell C that when you say "A" you really mean "B." Now there's no real reason to do this except that we'd like to keep track of when we're talking about a CODEC sample and when we're talking about a 16 bit integer. We do this by telling C that "fractional" really means "short":

```
typedef short Fractional;
```

So from now on, we can write "Fractional" and mean "short."

Of course, the CODEC provides a sample of the I and a sample of the Q at the same time and we're probably going to want to deal with I/Q pairs. In mathematics, an I/Q pair would be called a complex number and we'd like to deal with complex numbers as single units. We've already seen how this is done; the "struct." We can combine the two forms and write:

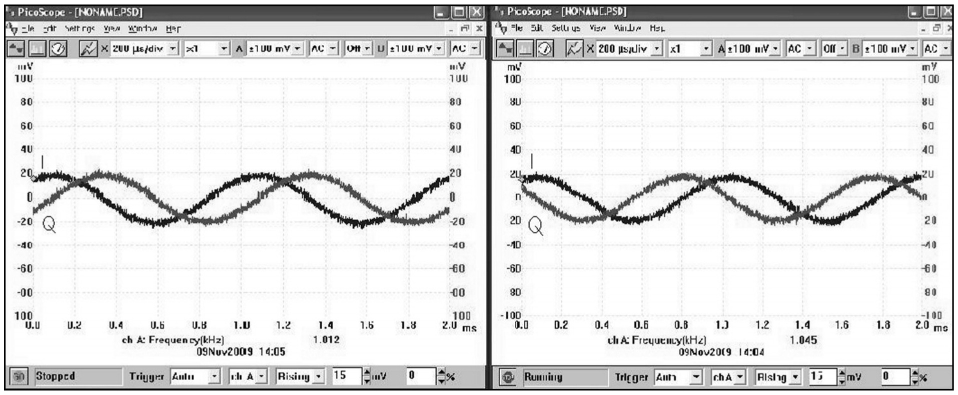


Figure 2—A 1 kHz signal passing through the SDR (no processing).

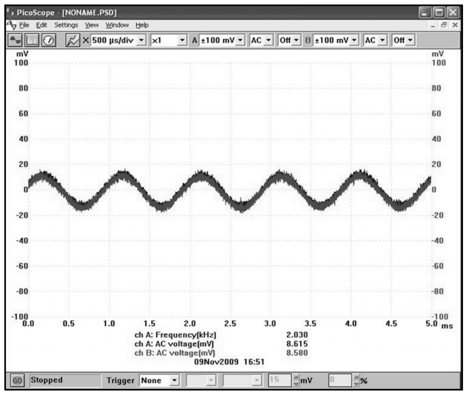


Figure 4—I and the derivative (slope) of Q for 1 kHz signal.

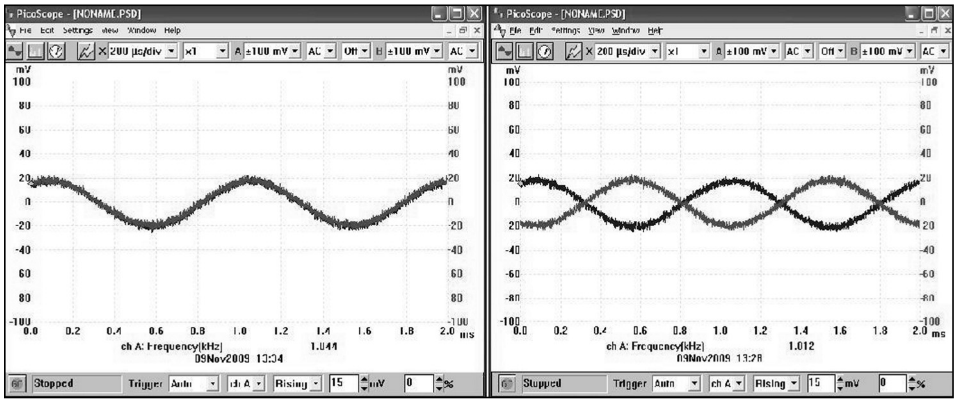


Figure 3—A 1 kHz signal with one channel delayed 900.

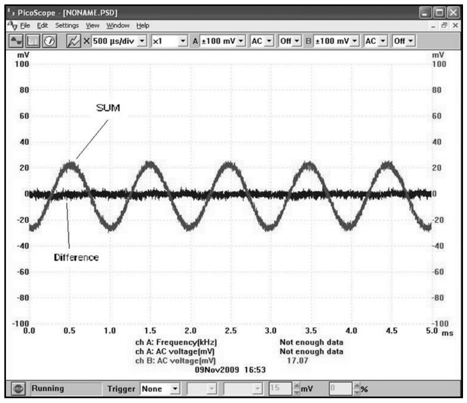


Figure 5—Sum and difference of I and slope of Q for 1 kHz.

```
typedef struct
  {
    Fractional i;
    Fractional q;
  } Complex;
```

In other words, when I declare a variable as "Complex" I mean it is a bucket with two Fractionals which are, of course, really shorts. So I can write:

```
Complex FromCODEC;
```

Oh, and of course, I can write the outgoing variables:

```
Complex ToCODEC;
```

We're getting close here. In C, I need to write a subroutine which will be called each time the CODEC produces a new sample. I'll brush over exactly how the routine gets called and just call it "CODECinterrupt." We will now write our first C program.

Without undue explanation, the following is the declaration of a subroutine which is called every time the CODEC generates a pair of new samples. This routine will simply take the data from the CODEC and hand it back. This is the equivalent of connecting the headphones directly to the QSD down converter. Figure 2 shows a 1 kHz signal passing through the SDR using this code. (Please note that there is lots of "hash" on the signals you will see here. My workbench environment is very noisy and the daemons have been too numerous to exorcise completely.)

```
void CODECinterrupt()   //"void" means this subroutine
                                  returns nothing.
  {
     Complex Tmp;    //a temporary holding place.

     Tmp = FromCODEC;        //get data from CODEC
     ToCODEC = Tmp;          //send data on to CODEC
  }
```

Please take a look at Figure 2 on the left hand side. (We'll see lots of dual trace pictures from now on.) On the left hand side of Figure 2, the one trace is the "I" channel and one trace is the "Q" channel. Note that the "I" channel is ahead of the "Q" channel. In this system, this indicates a time when the frequency of interest is above the local oscillator. If the local oscillator were 14.002 MHz then the "Q" channel would lead the "I" channel as seen on the right hand side of FIgure 2. Some people would call this a "negative frequency," but don't get hung up on that term just yet. There are other things to explore right now.

Step One: Opposite Side Band Suppression

One of the true joys of any hobby is the chance to sit back and

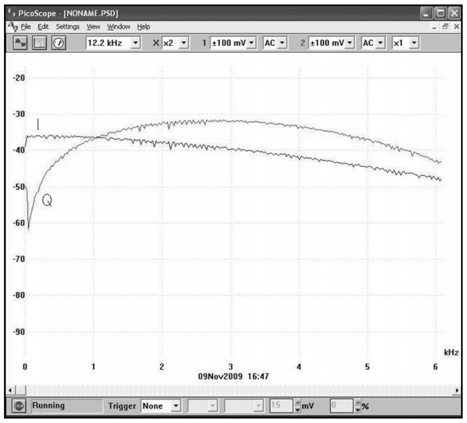


Figure 6—I and slope of Q as a function of frequency.

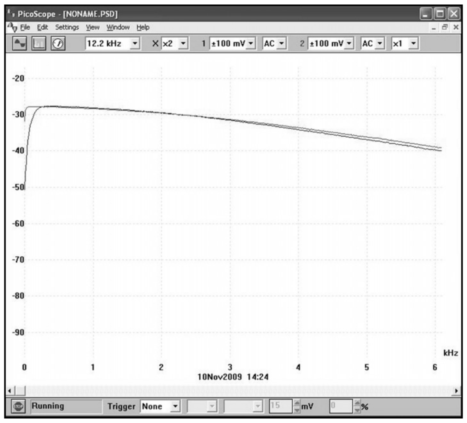


Figure 7—I and Hilbert Transform of Q as a function of frequency.

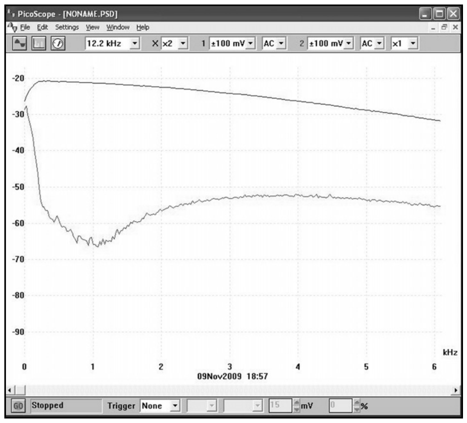


Figure 8—Sum and difference of I and Hilbert Transform of Q.

try some things and ponder the results. Unlike a practicing engineer plagued with deadlines and budgets, the amateur can try some things and learn from the experiments. Even a failure is a lesson because it shows us when our understanding is incomplete. Knowing when we don't understand is important because then we won't proceed in false confidence. One reason for this whole project was to provide a way to try some experiments. Let's try one now.

In order to increase the usefulness of the rig and to reduce the amount of QRM and QRN, I decided to eliminate the "opposite sideband." To do this, the literature suggests using a "Hilbert Transform" because forming the Hilbert transform of a signal is equivalent to adding a 90 degree phase shift to all components of the signal, thus setting the groundwork for generating an SSB signal via the phasing method. I didn't really want to just do things "by the book"; I wanted to try some of my own ideas. So I ignored the literature and did a little thinking. In looking at Figure 2, I saw that what I really wanted to do was to delay one of the two channels by another 90 degrees. Figure 3 shows the results. Note that when the signal frequency is below the local oscillator the I and Q channels are essentially the same and when the signal is above the local oscillator the two channels are opposite.

Now if the signals are added together the "lower" sideband would be selected because the two channels are in phase. The "upper" sideband, however, would have no signal because the two channels are opposite. This is the essence of "opposite sideband rejection using the phasing technique."

The question is how to delay a signal by 90 degrees. For simple sine waves, delaying a signal by 90 degrees is just like "taking the slope" or "taking the derivative." So I wrote a little program like this: I keep three copies of the incoming samples. Then I simply computed the "slope" of one of the channels; here I take the slope of the Q channel. (NOTE: I needed to keep the delay of the two channels the same so I had to delay I by one and take the slope using the "fromCODEC" and an "oldoldCODEC." It is worth playing around with this to convince oneself that this is correct.)

```
    Complex oldCODEC;
    Complex oldoldCODEC;

void CODECinterrupt
  {
      toCODEC.i = oldCODEC.i;           //match delay.
      toCODEC.q = (oldoldCODEC.q - fromCODEC.q)*scale;

      oldoldCODEC = oldCODEC;
      oldCODEC = fromCODEC;
  };
```

Note that I slipped in a "scale" factor. In my experiment, I simply declared a new parameter in the uMenu and viola! Figure 4 shows the results. The two traces are essentially co-incident. Honestly, this looks pretty good. Figure 5 shows the result of the "addition"; one trace being the sum and the other being the difference. I confess I had to tune the "scale" variable to give the best answer.

What happens if I change the frequency? Figure 6 shows a scan of "I" and "Qslope" amplitude as a function of frequency. Notice how the two traces cross at 1000 Hz. That is where I tuned "scale" to eliminate the opposite sideband. At frequencies other than 1000 Hz, though, this technique really won"t eliminate the opposite sideband well at all. If I had a really narrow filter right after this simple circuit, it is possible this might work. Although not shown, I'd like to point out that at other frequencies the "Qslope" phase is still correct, it is only the "scale" that is wrong because it needs to change with frequency.

An Unexpected Imperfection

OK; back to the Hilbert Transform; no real surprise. Figure 7 shows the I and the Hilbert Transform of Q as a function of frequency. (In Part III of this article, we will talk about how the Hilbert Transform is computed. For now, just assume that we can generate the transform and apply it as shown to our incoming signal samples.) Again, I did some tuning for 1000 Hz. Note that the Hilbert Transform does much better over a wider range but still has a few problems. Examine the two traces closely. At the very left margin, the trace heading down is the Hilbert Transform. This is a well-known shortcoming; The Hilbert Transform has trouble at frequencies near zero. Now look toward the right hand side of Figure 7. Notice that the two traces are diverging. As it turns out, this is unexpected; the Hilbert transform should work well at the

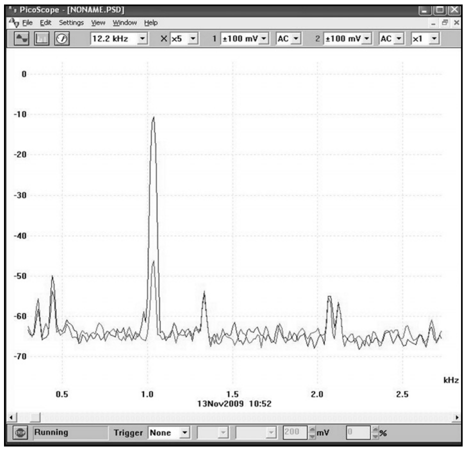


Figure 9—Testing the Hilbert Transform implementation.

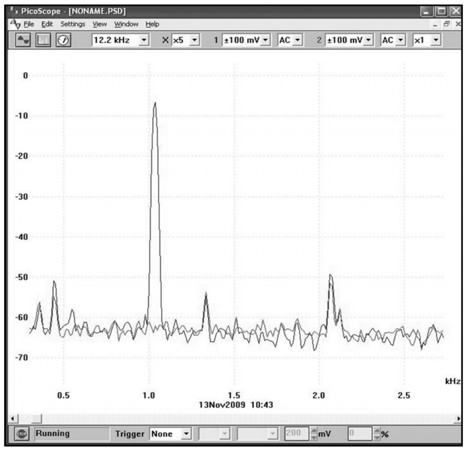


Figure 10—Sum and difference without adjustment factors.

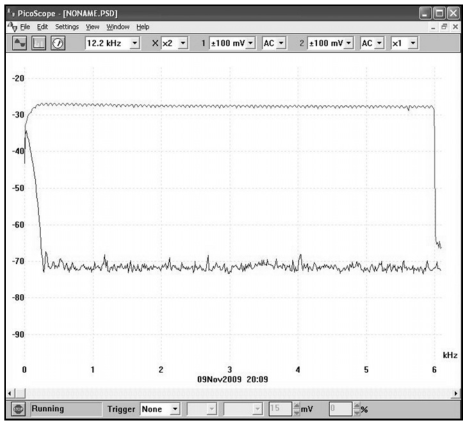


Figure 11—Sum and difference with adjustment factors.

higher frequencies. It often helps to look at the problem in a different light. Figure 8 shows the same data but with the one trace as the addition and the other trace as the difference.

Figure 8 shows that the Hilbert transform does much better than my simple slope detector but problems remain. Near the 1000 Hz, point the Hilbert transform performs better than my equipment's ability to measure. Away from 1000 Hz though, things are different. If you were to listen to this rig on the air you would hear opposite sideband signals. They would be faint but you would definitely hear them. This rig, so far, is suppressing the opposite sideband by only 20 to 25 dB well away from the tuned frequency.

This is clearly unacceptable. The question I asked was WHY this was happening. There were four options: the Hilbert transform did not work as I expected, I coded the transform incorrectly, the CODEC was not digitizing correctly or finally, the QSD board wasn't working correctly.

I decided to find out if the problem was my implementation of the Hilbert Transform. My first step was to write a program which would generate a sine wave of a chosen frequency. This routine would be needed later anyway so now was a good time. I decided to write a routine called "ejm" which would return a complex number with a real part=cos(m) and an imaginary part=-sin(m). I could then use this "ejm" routine to generate a (more or less) perfect input signal with which to test my Hilbert transform. (The interested reader can find the code for "ejm" in my source code.) Here was my test code:

```
long BeepInc;
long BeepPhase;
Complex In, Tmp, S;

BeepPhase = BeepPhase + BeepInc;      //advance generated
                                        phase
In = ejm(BeepPhase);             // get an 'I' and a 'Q'
Tmp.i = Hilbert(In.i);           //Hilbert transform of I
Tmp.q = MatchHilbertDelay(In.q);      //"Hilbert" delays
                                        'i'… delay 'Q' to
                                        match.
S.i = Tmp.i + Tmp.q;             //sum
S.q = Tmp.i - Tmp.q;             //difference.
ToCODEC = S;                     //send it out!
```

Figure 9 shows the result. It is clear that the Hilbert Transform I had written was not the problem.

So the first step in examining this problem was to provide for adjusting the relative amplitudes and phases of the incoming I & Q channels. Common wisdom says this isn't necessary for the Quadrature Sampling Detector I was using but I decided to put it in anyway. The routine for adjusting amplitude and phase is fairly simple. (Unfortunately, dsPIC C does not have a built in way to multiply two Fractionals so I had to write a routine "fxf(fa,fb)" which would do that multiplication.) The first step is to add the menu elements:

```
{"AmpAdj",  0,-32000,32000,1,NULL},//no 'update routine'
                                    so "NULL"
{"PhzAdj",  0,-32000,32000,1,NULL},//no 'update routine'
                                    so "NULL"
```

Then, just after I get the incoming CODEC sample I wrote:

```
In = FromCODEC;
If (AmpAdj > 0)        //should I reduce magnitude of 'I'
  In.i = In.i + fxf(In.i,AmpAdj);
else
  In.q = In.q + fxf(In.q,AmpAdj);        //if not, reduce the
                                          magnitude of 'Q'

In.i = In.i + fxf(In.q,PhzAdj);   //adjust phase if necessary.
```

This code is pretty simple and playing with it is HUGELY educational. Take a look at Figure 10. This picture shows the sum and differences as we've seen earlier. The goal is to have the bottom trace be "down in the noise." As you can see, the lower trace is down 35 dB or so. Pretty good. BUT by adjusting the AmpAdj and PhzAdj parameters you can produce Figure 11; down around 60 dB! It is most educational to play with these adjustments. You quickly find that you MUST adjust both amplitude AND phase to maximize performance; no amount of adjusting phase will

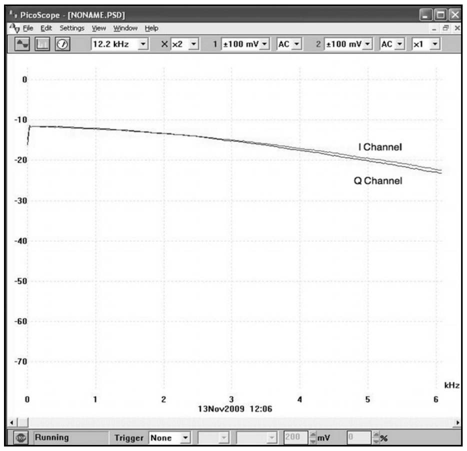


Figure 12—Gain of the I and Q channels vs. frequency.

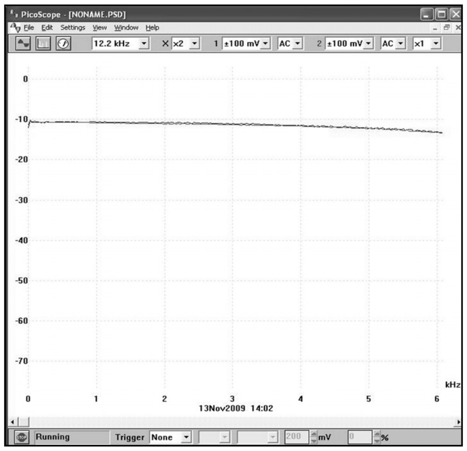


Figure 13—Gain of the I and Q channels with new op amps.

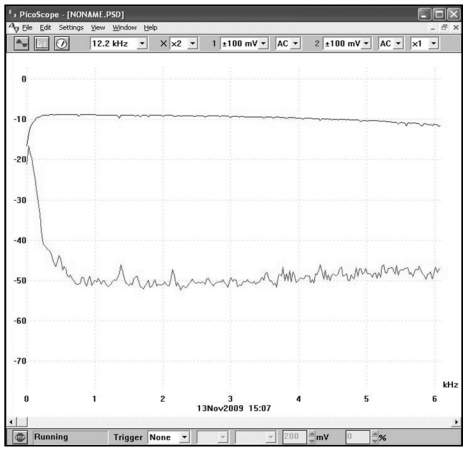


Figure 14—Sideband rejection after new op amps and adjustment.

account for differences in magnitude, nor vice-versa.

Having written that code I went and optimized the performance at a half a dozen frequencies. Here's what I found; as the frequency went up the required AmpAdj and PhzAdj went down. This represented a critical clue. If the AmpAdj was constant then this would indicate a difference in gains of the two channels would be the problem. If PhzAdj was constant then this would represent a problem with the QSD mixer. (Note, I have since learned that this view was overly simplistic.) Since both adjustments changed as frequency changed, I was forced to conclude there was a worse problem. Perhaps there was a difference in the roll-off of the two channels either in the op amps or the QSD integrating capacitors. If one channel rolled off faster than another then this would account for my observations.

Yes, I should have gone and looked earlier, but I had read in several places that differing rolloffs between channels was never a problem. Well, I went and looked and saw what is shown in Figure 12. On the left hand side, Figure 12 shows that the DC gain of the two channels are essentially equal. Because of this, I concluded that resistors used in conjunction with the amplifiers are reasonably well matched. However, as the frequency increases the gains diverge. Thus, the problem is in the roll-off of the two amplifiers. I then went and looked up the data on the LT1636 op amp I was using. It turns out the open loop gain of this device is only about 30 dB at 6 kHz. My QSD board had set the op amp gain to 40 dB. Thus, the op amp itself is rolling off roughly 10 dB from 0 to 6 kHz; almost exactly what is showing in Figure 12. Yes, I had to go change my op amps. Figure 13 shows the results after I had replaced them. As you can see, things are much improved but not yet perfect; notice how the bottom signal rises toward the right of the trace. I suspect that this is an intrinsic property of QSD mixers, their "conversion gain" rolls off with increased frequency; something to look into later.

In the end, I reached a fairly flat sideband rejection. Figure 14 shows a frequency sweep of the upper and lower sidebands. With the exception of low frequencies, the opposite sideband rejection is about 40 dB across a fairly wide band. About what one might expect from fairly casual construction and coding.

Summary

So now my project has achieved a significant milestone. I have a working hardware platform, a working user interface, I can tune the receiver and I can select a sideband. However, the receiver is still "wide open." There is no narrow band filtering yet implemented. When I started this journey, this is where I expected to be out of my element and where I would really start researching and discovering. I was right.

In Part III of this article, I will show how that research and discovery turned out. At that point, the software will be fully described and the transceiver functional with an output of about 200 mV peak/peak. Part IV of this series will describe the class E power amplifier used to boost this 200 mV signal to 5 watts.

Notes

The referenced sidebar, "A 'C' Primer: Describing a Menu," will be made available on the QRP ARCI web site when this article is published.

References

1. Lyons, Richard G., *Understanding Digital Signal Processing,* Second Edition, Prentice Hall, Upper Saddle River, New Jersey, 2004.

2. Hayward, Campbell, Larkin, *Experimental Methods in RF Design,* The American Radio Relay League, 2003.

3. Smith, Steven W., *Digital Signal Processing—A Practical Guide for Engineers and Scientists,* Newnes, 2003.