# QRPp

**Journal of the NorCal QRP Club, September 2016**
**Vol 12, Issue 2**


## The Portable Pixie Power Pack

Table of Contents



## From the Editor
## By Doug Hendricks, KI6DS

I know I said that when I announced the return of QRPp that it would be about 12 to 16 pages. You have probably noticed that this issue is far bigger, 60 pages. How and why did that happen? I got some great articles, and it was easy to do. I was talking with Jack Purdum, and we agreed that we needed to speed up the delivery of his tutorial series on the ham radio uses of Arduinos. The readers would be anxious to get started. So we combined parts 1 & 2. I realized that the reason I used to limit the size of QRPp was the cost of printing and distribution. But I don't have those issues any more. So it seems natural to make QRPp whatever size is necessary to do the job. Chuck and I discussed this, and he said that he could do a better job with no page restrictions. You will note that all of the articles have lots of pictures, which I think is a good thing. I like the articles in this issue, as they are interactive. You can build the projects in the articles. All of the information is there, including artwork to make boards where needed. We live in an amazing world. Please read the Pacificon Announcement. It will give you information that you will need to attend. Pacificon 2016 is going to be fun. Chuck Adams, K7QO is the keynote QRP speaker, show and tell, Chinese kit display, QSO Party, Free Dummy Load building event, qrp vendor night. All of this at Pacificon 2016. See you there.

Don't forget, If you have an article, send it to ki6ds1@gmail.com. 72, Doug,

## The PPPP or P^4 Power Supply, Sept. QRP Meeting Kit
## by Doug Hendricks, KI6DS

This month's kit was inspired by Jon Kim, who showed me a version that he had made to charge his smart phone using a 9V battery as the power source. I decided that I wanted to build a variable power supply that would be adjustable from 3 to 30V. I searched Ebay and came up with the XL6009 Buck/Booster module. Here are the specifications:

**Product Description**

The module uses the second generation of high-frequency switching technology XL6009E1 core chip performance than the first generation technology LM2577. XL6009 boost module has superior performance than LM2577 based modules.

1. Wide input voltage 3V ~ 32V, optimum operating voltage range is 5 ~ 32V
2. Wide Output voltage 5V ~ 35V
3. Built-4A efficient MOSFET switches enable efficiency up to 94%; (LM2577 current is 3A)
4. High switching frequency 400KHz, can use a small-capacity filter capacitors that can achieve very good results, the ripple is smaller. (LM2577 switching frequency is only 50KHz)

**Technical Specifications**

- Input voltage :3.2V-32V
- Output voltage: 4V-38V (Must ensure that set the output voltage is higher than the input voltage)
- Modules Type: Non-isolated step-up (Boost)
- Rectification: Non-synchronous rectification
- Rated Output Current: 3.0A
- Peak Output Current: 4.0A
- Module size: 43mm x 21mm x 14mm (L x W x H)
- Input mode: IN + input positive level, IN-input negative
- Conversion efficiency: Upto 94%

As you can see, it is just what I was looking for. The cost is very affordable, in late August 2016, they are priced at $1.06 each including shipping. What a deal. When I first thought of this project, I was going to include a 3 digit digital readout. But the problem is that the 3 digit volt meters are not very accurate. I hadn't even thought of verifying the reading until Steve Smith, WB6TNL, asked me on the phone if I had verified the output voltage. I suppose I had a deer in the headlights look, but the answer was no. So, I got my trusty Harbor Freight Red VOM and checked it out. I got my variable power supply ready to go and set it at 6.00 Volts. Read it on the Harbor Freight, and it read

6.0 Volts. Then I read it with the 3 digit volt meter. Uh oh. 6.5 volts. That is an error of .5 Volts or 8.3%. Too much error. Then I set the power supply at 12.0 Volts and read it with the Harbor Freight. 12.0 Volts. Good. Thne I used the 3 Digit Voltmeter. Oh my, 11.4 Volts. That's an error of 0.6 volts, 5.0%. But the problem is that at the low end it was measuring high, and at the high end it was measuring low. I asked Pete Rowe, of Rowe Labs, what he thought of that, and he said well, somewhere between 6 and 12 volts, it is correct. But it is not very accurate everywhere else. So, I decided not to use the digital readout.

Building the kit is straight forward. Below you see the parts included in the kit.

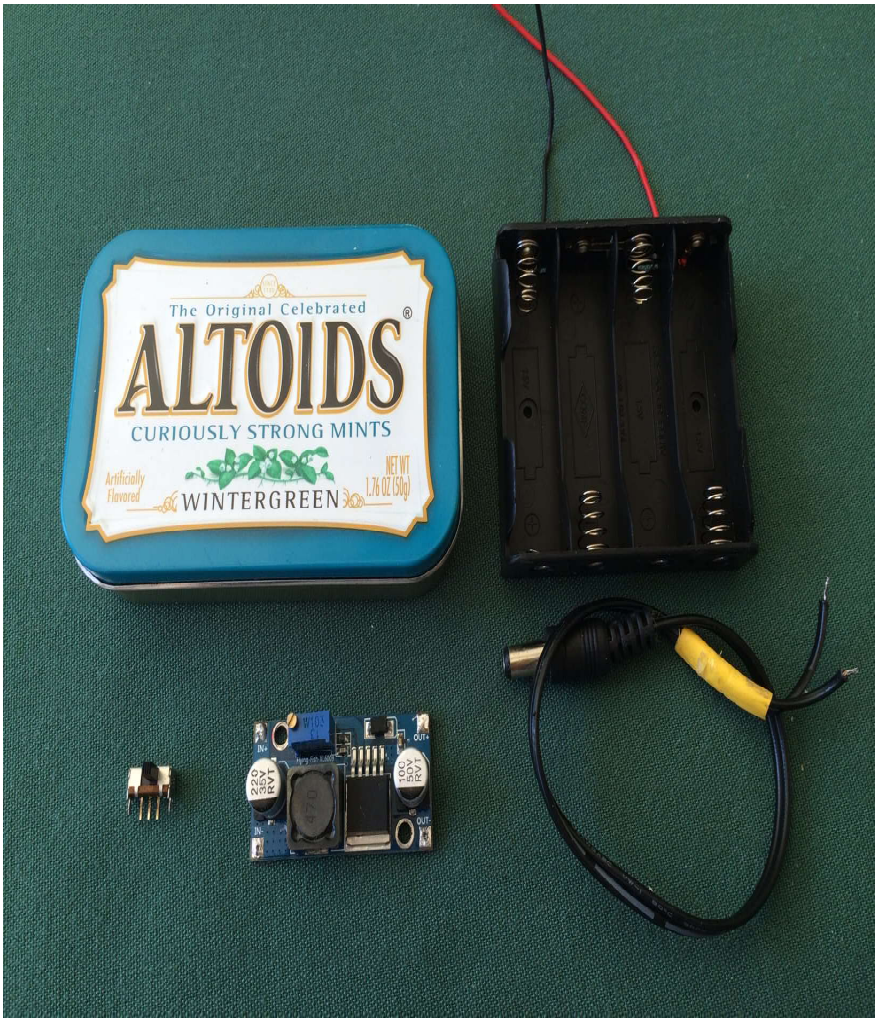



Fig. 1 Kit Parts

Note that the Altoids Tin is not included in the kit. Parts clockwise starting with the Altoids Tin, 4 cell AA battery case, 2.1 mm x 5.5 mm power lead with 2 pieces of shrink tubing (yellow part), XL6009 module, power switch. The picture below shows the hardware. You will have two of each item pictured.

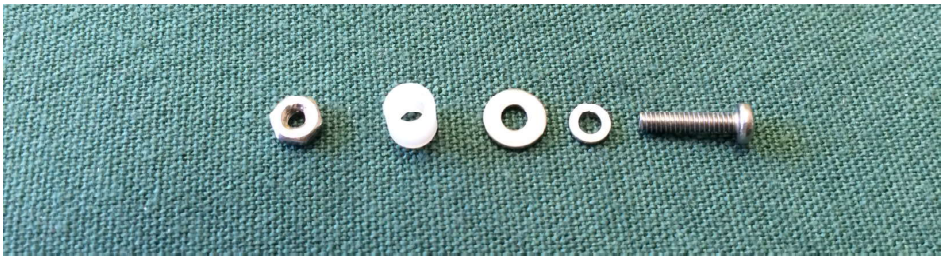


Fig. 2 Hardware

The first thing to do is to reroute the wire on the battery case. You want it to come out the bottom, not the side. Simply work the wire back through the holes until your battery holder looks like the example below.

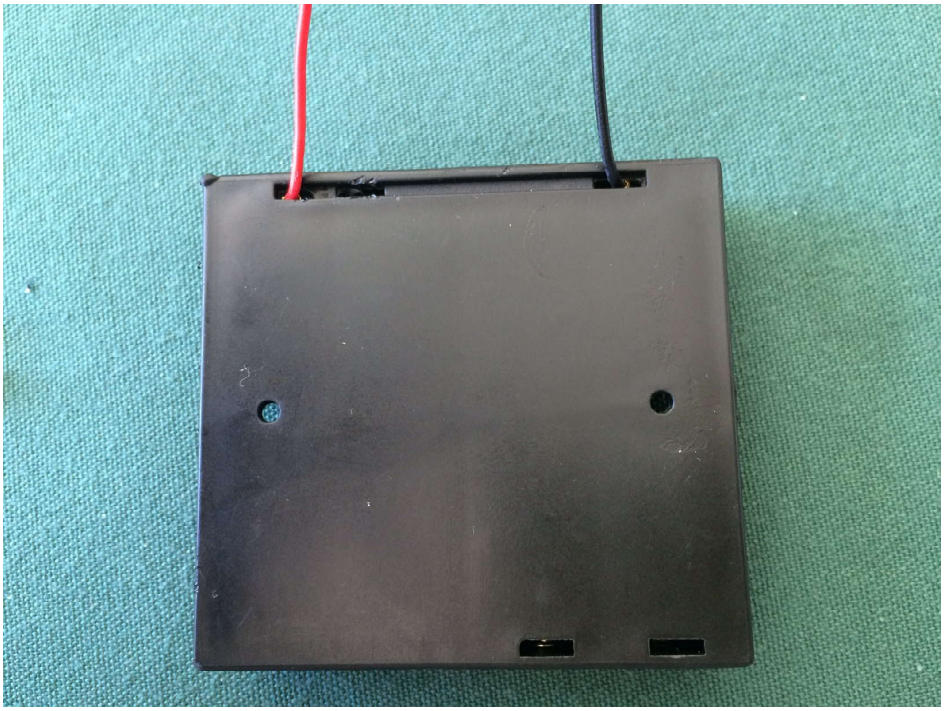


Fig. 3, Battery holder with wires rerouted.

Next we will wire the switch, battery holder and booster module. The booster module has an input and an output. It is important that you wire the battery to the input side, or your project will not work.

The first thing you want to do is drill the Altoids tin. The way that you do this is to take the module and put it in the tin where it will go. It must go as close to the end as possible for everything to fit. Mark the hole closest to the middle with a fine point marker. Use a center punch to mark the hole placement, and drill it with a 3/32 bit. Now, put one of the machine screws provide through the hole and mount the module so you can mark the other hole. This one is tough. I used a small nail that just fit in the hole to mark it. I had to do that because I could not get the marker in vertical. Remove the module. Mark with the center punch and drill. Next, mark the front of the panel where the power cord will go. Make sure you don't put the hole too high, you need clearance for the lid to close. Use the pictures as a guide. Start with a 1/8" bit and work your way up to a 3/16. I use a step drill.

Start the wiring process by preparing the battery case. Cut the red wire in half. Wire the red wire from the battery case to the middle pin of the slide switch. Now take another piece of hook up wire that you supply, and measure it to be 2" longer than the length of the Altoids tin. Run this wire from one of the end pins of the switch to the input + connection on the XL6009. Next solder the black wire to the input - connection of the XL6009. Prepare the power cord by putting on the two pieces of heat shrink tubing one at a time. The tubing should be about 1" from the end of the insulation. Shrink the first one, followed by putting on the second and shrinking it. This will act as a good insulator for your power cord. Thread the cord through the hole. Be sure to do this step before you solder the cord to the module or you won't be able to get it though the hole. Been there, done that. Solder the cord to the plus and minus output connections of the module. Make sure you match polarity. Inside of the connector is plus.

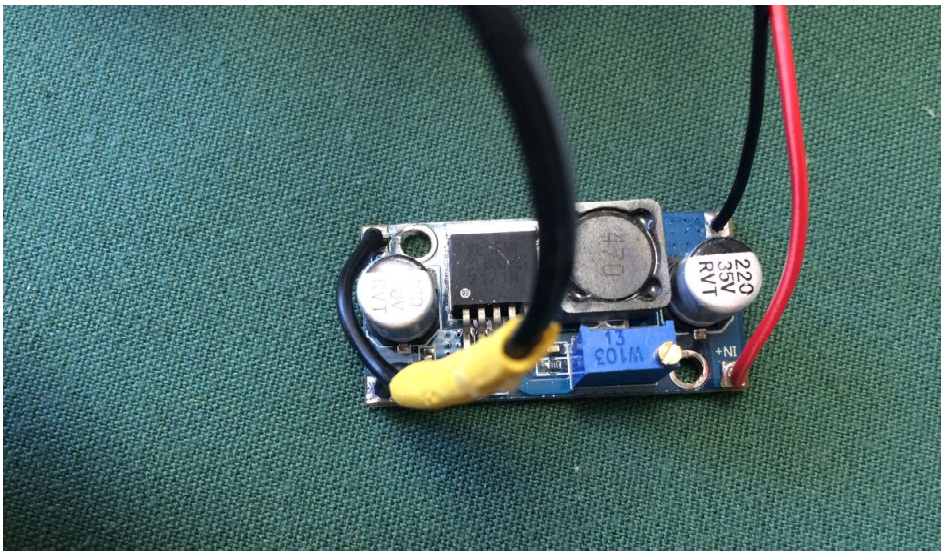



Fig. 4 Module shown wired. Note input on the right

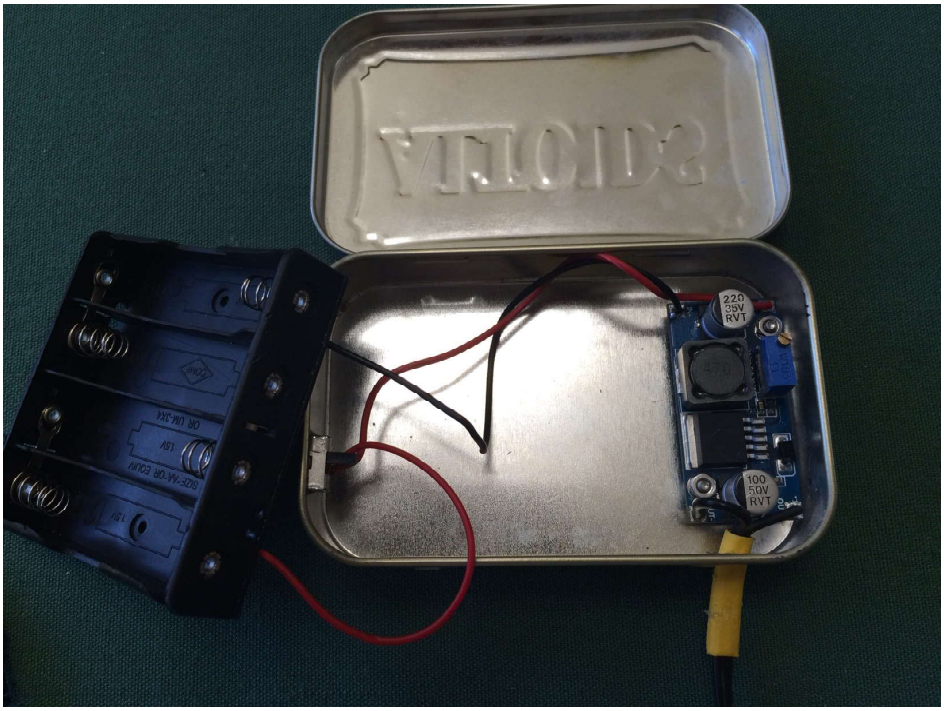



Fig. 5 Module, battery case and switch assembly after wiring

In the picture above the module has been mounted in the tin. Mounting the module is easy, if you follow this procedure.

1. Put a lockwasher on the screw.
2. Place a screw in the hole nearest the edge.
3. Put the nylon spacer on the screw.
4. Mount the module.
5. Put a nut on the screw but don't tighten.
6. Take tweezers and put the other spacer under the module and align.
7. Put lockwasher on screw, and insert from the bottom.
8. Put the nut on and tighten both screws.

Easy wasn't it!! Now pull the power cord through the hole. Seal with hot glue. Do a neat job, but apply glue both inside and outside the case. This will provide strain relief. Don't forget to do it. Otherwise, the wires will break with the flexing of the wire. Next, solder the switch onto the end of the case. Do not apply heat very long. The switch will melt. I used an alligator clip to hold the switch in place while soldering. Check the switch when you finish to make sure it works.

Fig. 7 Hot Glue on power cord

You are just about done. Next put 4 AA batteries in the battery case. Use your VOM to check the voltage at the input of the module. You should have 6 volts or so. If you don't, check to make sure the switch is on. After verifying 6 volts input, we will set the voltage of the output. Pick any value between 6 and 38 volts. Remember the output voltage must be higher than the input voltage. If you want 5 volts for a smart phone charger, then you will need a dummy battery for one of the battery slots, so you can have a 4.5V supply. To set the voltage use a VOM to measure output voltage as you adjust the trimpot on the board. Simple. Once it is set, you are good to go. I hope you enjoyed this project. For those of you who did not attend the meeting and get a kit, the module is available on Ebay. Do a search for a XL6009 Buck Booster. The battery case is available from Mouser, but much cheaper on Ebay. Look for a 4 cell AA case that is the same as the one in the photo. That does it for the September QRP Meeting Kit.

If you are in San Jose on the first Tuesday of the month, come to the meeting at Denny's, 1140 Hillsdale, San Jose starting at 6:30PM. We would love to have you, and you will get a free kit too!!

72, Doug, KI6DS


Fig. 8 Inside the finished Power Supply

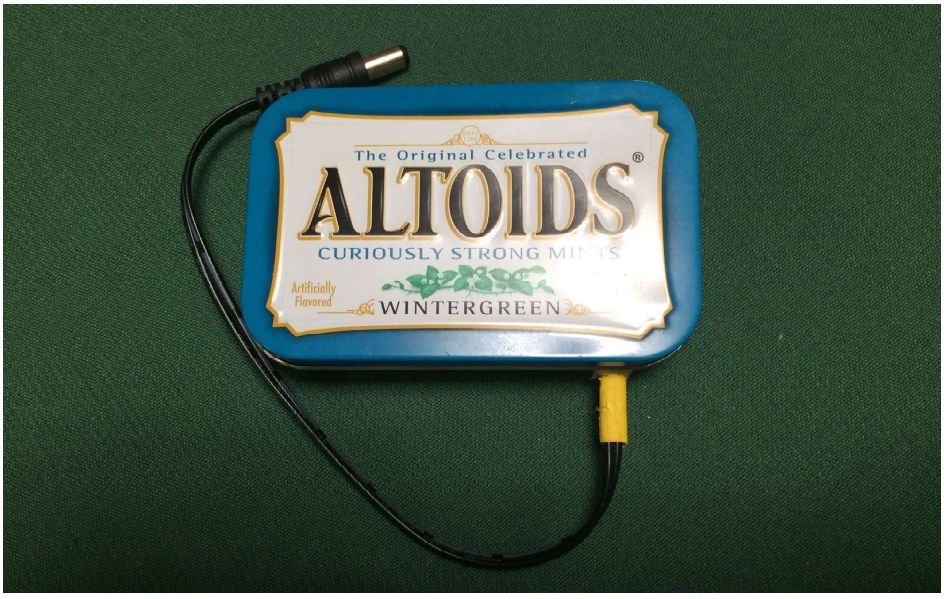



Fig. 9 The P^4 Power Supply

## Arduino for the Terrified, Part 1
## by
## Jack Purdum, Ph.D. W8TEE

Strange title, isn't it? Yet, that's often the impression I get when I mention using an Arduino microcontroller (uC from now on) in a ham radio project. Distilled to its essence, the most common reason I hear for not investigating and using an Arduino is: "But I don't know how to program!" Yeah...so what?

You've already proved that you are smart enough to get a ham license. Couple that with the ability to fog a mirror and you pretty much have the qualifications necessary for using uCs in your projects. When you started preparing for your license exam, did you know how to draw the schematic for a Colpitts oscillator? Could you calculate the length of a half wave dipole on 40M? For many of us, probably not, but you took the time and made the effort to learn what you needed to know to pass the exam. What you need to know about an Arduino is probably less difficult than what you went through to get your license. How so?

First, the Arduino family of uCs is built upon the concept of Open Source resources. Open Source is a movement/philosophy which means that virtually everything from the software applications you use to write your own programs to the details of the Arduino chip design is openly available to anyone. As a result, there's about a bazillion lines of program source code out there just waiting for you to use it. Program source code is a collection of English-like program statements, often written in the C programming language, that cause the controller to do your bidding. In addition, there are hundred of task-specific mini applications, called libraries, that are the basic building blocks of program development. There are libraries to use LCD displays, touch screen displays, stepper motors, finger print readers, motion-heat-fire sensors, GPS coordinate libraries, proximity sensors, laser controllers...you name it and there's probably a support library for it. As a result, program development is like building an object with Legos. Find the right Lego, snap it into place, and repeat until done.

Second, mistakes are normally non-destructive. Unlike hardware mistakes where, once the white smoke gets out, that chunk of hardware is done, software mistakes manifest themselves as simply producing a result that doesn't match what you want. Fixing mistakes simply means tracking the error, or program bug, down and correcting it. Even if you'd make a really BIG flat-forehead mistake (you know, the kind where, once understanding the mistake, you slam the heel of your hand into your forehead and ask "How could I make such a mistake!") and that mistake does "brick" (i.e., destroy) an Arduino board, the Arduino that we'll be learning on can be replaced for about $3.

Third, the tools to write programs are completely free. Most popular platforms are supported (e.g., Windows, Mac, and Linux). There are dozens of

good books that provide more information if you need it, plus there are literally thousands of free programming tutorials on line. In this series, we will give you enough to get you started. If you have a particularly nettlesome question, there is an Arduino user group Forum that is designed to answer those questions. (See: http://forum.arduino.cc/index.php?board=4.0. Make sure you read the post titled *How to Use This Forum* before you post a question.) Simply stated, there's a lot of help out there if you need it.

Finally, many of you are going to discover a new sense of accomplishment when your program does what *you* want it to do. It's not too different from making your first QSO with a piece of homebrew equipment. I spend a lot of time on the above-mentioned Forum simply because I truly enjoy programming. You might discover that you enjoy it as well.

I guess what I'm saying is: Give it a shot. For an investment of a couple of dollars and a little of your time, you may discover a new venue for augmenting the enjoyment you get from ham radio. Like it or not, software is a growing element of our hobby...just look at the number of rigs that have a least some element of SDR (Software Defined Radio) in them. You may as well stick a toe in and test the waters.

**Which Arduino to Buy?**

Actually, there is a whole family of uCs that find a home under the heading "Arduino". Table 1 shows some of them, plus a number of key elements of each. All of the members of the Arduino family have three types of memory,

| Micro controller | Flash memory (bytes) | SRAM (bytes) | EEPROM (bytes) | Clock speed | Digital I/O Pins | Analog input pins | Voltage | Price (Approx.) |
|---|---|---|---|---|---|---|---|---|
| Arduino Uno | 32K | 2K | 1K | 16Mhz | 14 | 6 | 5V | $8.00 |
| Arduino Nano | 32K | 2K | 1K | 16Mhz | 14 | 8 | 5V | $4.00 |
| Digispark | 16K | 2K | 1K | 16Mhz | 14 | 10 | 5V | $2.00 |
| RoboRED | 32K | 2K | 1K | 16Mhz | 14 | 6 | 5 or 3.3V | $15.00 |
| ATmega1280 | 128K | 8K | 4K | 16Mhz | 54 | 16 | 5V | $15.00 |
| ATmega2560 | 256K | 8K | 4K | 16Mhz | 54 | 16 | 5V | $10.00 |
| Arduino Leonardo | 32K | 2.5K | 1K | 16Mhz | 20 | 12 | 5V | $8.00 |
| Arduino Due | 512K | 96K | - | 84Mhz | 54 | 12/2[1] | 3.3V | $16.00 |

***Table 1.***

have a clock speed of 16MHz (except the Due), and can operate from a 5V source. Actually, they can tolerate voltages between 5V and 17V, but need 5V to function properly. As you can see, you have quite a few choices, but none are prohibitively expensive. Also, they all use the same programming tools to write programs. So, which one should you buy? It depends on what you want to do with it. Recently I've been working on an antenna analyzer and started out using a Nano, mainly because of its small size. (See Fig. 1, the Nano is the

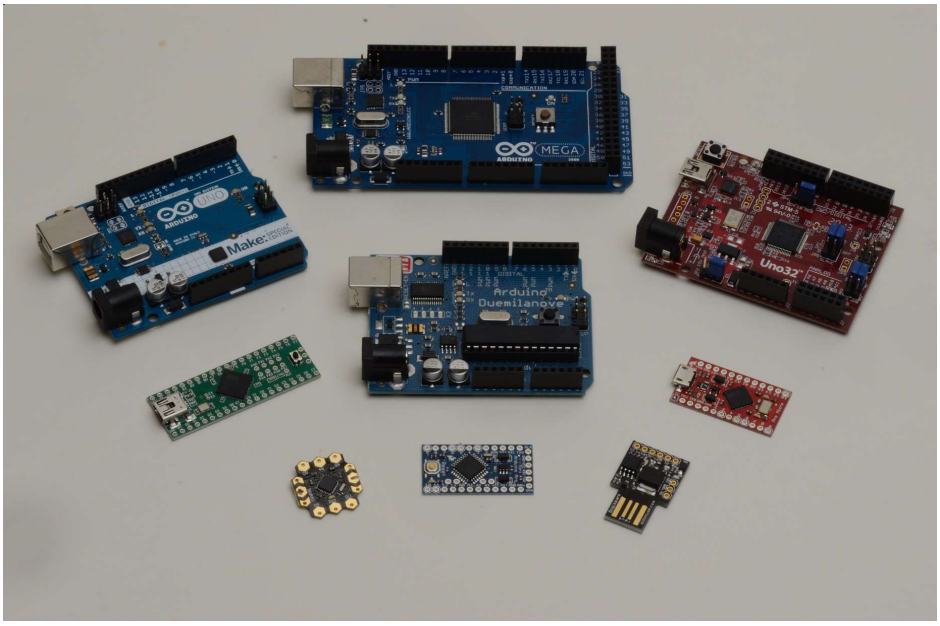



***Figure 1. Some Arduino boards.***

thumbsized board at the center-bottom of the figure while the Uno is in the top-left of the figure.) However, as I started adding more and more features, I exceeded the amount of memory needed for the program, so I upgraded to the Atmega2560 (center-top of Figure 1). The great thing about the Arduino family is that most code is upward/downward compatible. Typically, your choice is most influenced by available memory and the number of Input/Output (I/O) pins available. Let's look at the memory types first.

***Flash Memory***

Flash memory is the memory that is used to store your program code. If you look at Table 1, you'll see that the most popular Arduino boards (Uno and Nano) only have 32K of memory for your program. In an age where PC's have a mega-munch of memory, how can you do anything in 32K? You'd be surprised! My article in the March, 2016, issue of *QST* describes building a 40M, 3W, CW transceiver with VFO and LCD display and the code that drives the rig uses less than 9K of flash memory...plenty of room left over for adding more features (e.g., a keyer?).

Another feature of flash memory is that it is non-volatile. That is, when power is removed, the content of flash memory remains unchanged. When power is reapplied, the application restarts. (There are also some tricks you can use to stretch the flash memory a bit if you start to impinge on its upper limits.)

***SRAM Memory***

All of the program's data resides in SRAM memory. Indeed, you are more likely to run out of SRAM memory than flash memory in most applications. When you compile a program, the compiler reports the amount of flash and SRAM your program is using so it's pretty easy to see when things might be getting tight. (Look at the bottom of Figure 7 below.)
Unlike flash memory, SRAM is volatile memory, which means it goes stupid when power is lost. Any values that variables may have had when the power is removed are lost. When power is reapplied, those variables will assume whatever values you gave them at program start. If you didn't assign the variables an initial value, they will have random values.

***EEPROM Memory***

Electrically Erasable Programmable Read Only Memory (EEPROM) is an area of nonvolatile memory where one often stores data that need to be retrievable each time the program is started. Like flash memory, data values stored in EEPROM survive power removal.

However, EEPROM has two drawbacks when compared to flash memory: 1) it is a little slower to access than flash memory, and 2) it has a finite number of write cycles (i.e., 100,000) before it becomes unreliable. Because of these factors, EEPROM memory is often used to store configuration or other types of information that are needed when the system powers up, but are not often changed. For example, you might have some sensors that need to have certain values sent to them before they can be used, or other devices that need to be initialized with specific data. EEPROM would be a likely candidate for storing such configuration data. I used the EEPROM memory to store the band edges for the CW transceiver mentioned earlier because they are unlikely to change that often.

***Input/Output (I/O) Pins***

As you might expect, a uC with more memory and I/O pins costs a little more. Note that there are numerous clones available on the Internet for each member of the Arduino family. As a general rule, buy the "biggest" you can comfortably afford that is consistent with the project(s) you have in mind. If you buy a Nano (a good choice for this series), make sure you don't get the Pro Mini, as it does not have the USB connector on it which makes it less convenient to program. Hardware projects are often subject to "feature creep" where more and more functionality is requested as the project goes forward. "Buying bigger than you need" is often a good idea if you can afford it.
There is one advantage to the Uno and Mega boards: They support plug-in boards that can be directly plugged into the board. These plug-in boards, called *shields*, are hardware devices for everything from GPS sensing to finger print readers. The Nano and other smaller boards do not directly support these plug-in boards. You can still use such boards, but they are connected to the I/O pins

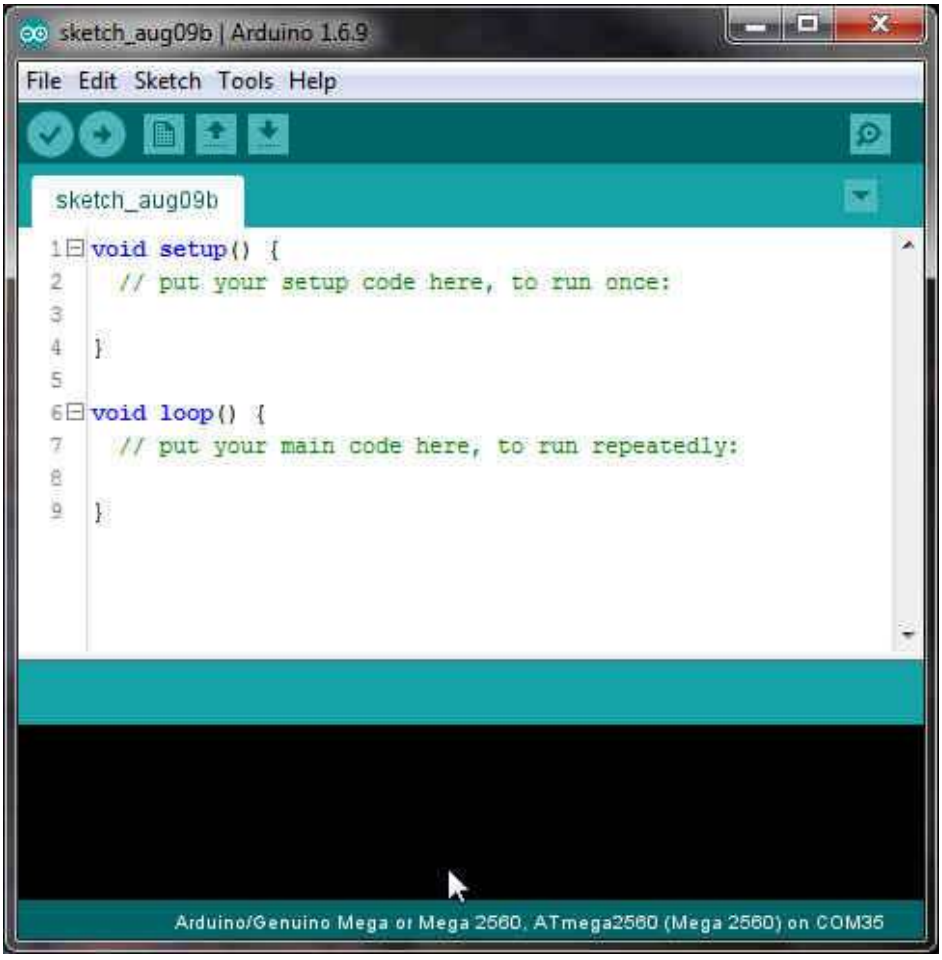



***Fig. 2, The IDE Startup Screen***

via jumper wires. BTW, the RobRed board makes it easy to have off-board connections because it brings all of the I/O pins out to individual header pins. My suggestion: Buy a Nano to start with. They can be purchased online for less than $3 and they have the same hardware resources as the Uno.

**Arduino Software**

A uC without software is about as useful as a bicycle without wheels. Like any other computer, a uC needs program instructions for it to do something

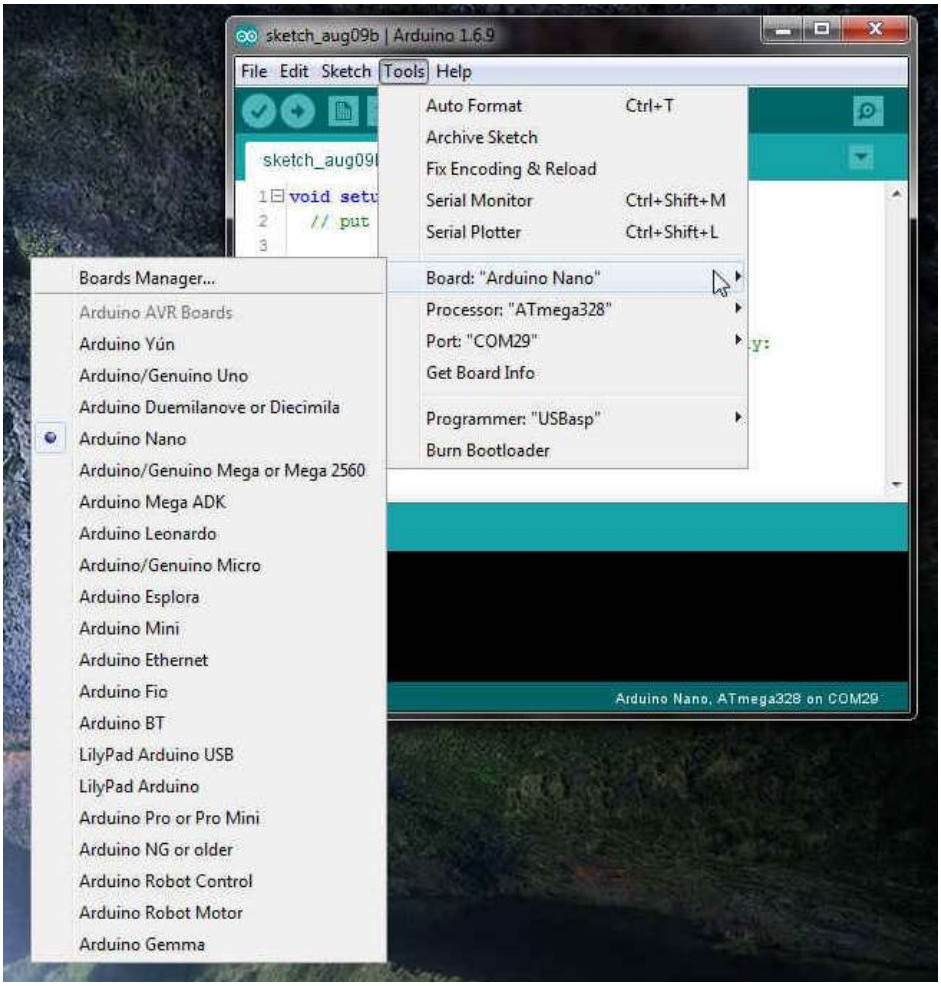



***Figure 3. Setting the Arduino board.***

useful. Arduino has provided all the software tools within their (free) Integrated Development Environment (IDE) that you need to write program code. The remainder of this article discusses downloading, installing, and testing the software you need. Start your Internet browser and go to:

http://arduino.cc/en/Main/Software

All that remains is to select the proper Arduino board and port. Figure 3 shows how to set the IDE to recognize your board. As you can see, the one Isupports a large number of the Arduino family. Click on the board that you are using.

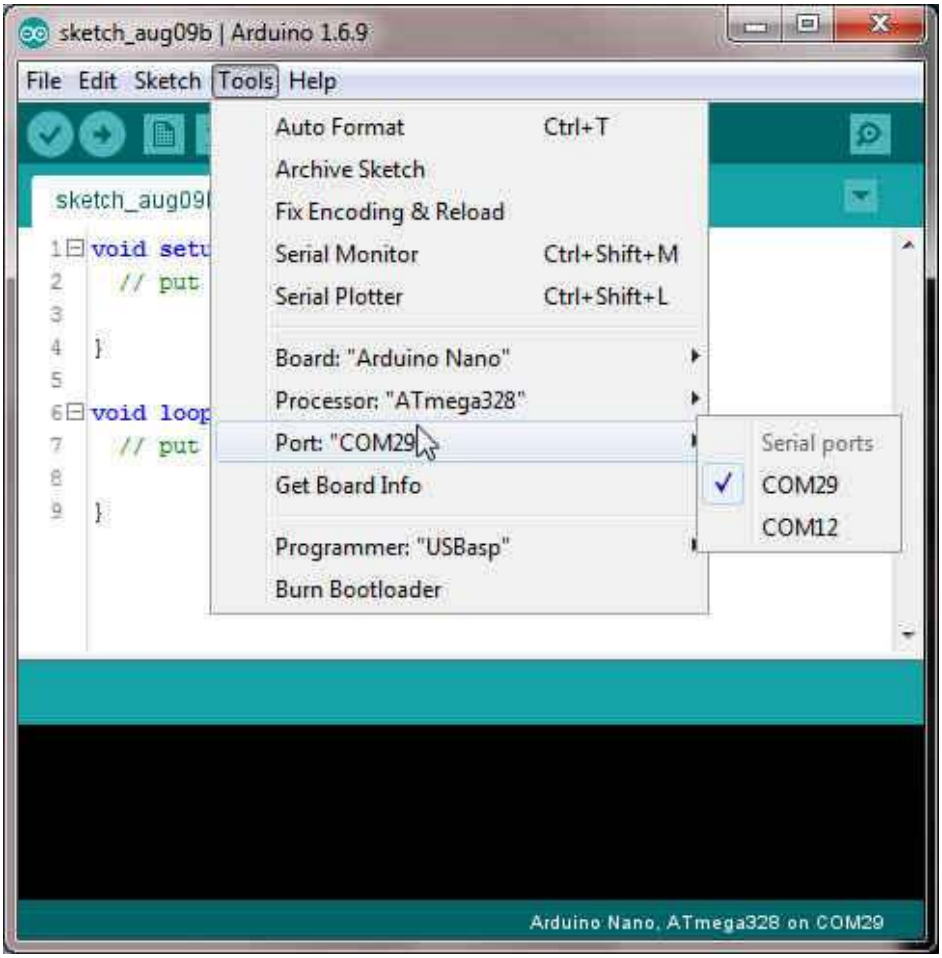



***Figure 4. Setting the communications port for the Arduino.***

Now set the port number for the USB port that is being used to communicate between your Arduino and the PC. This is shown in Figure 4.

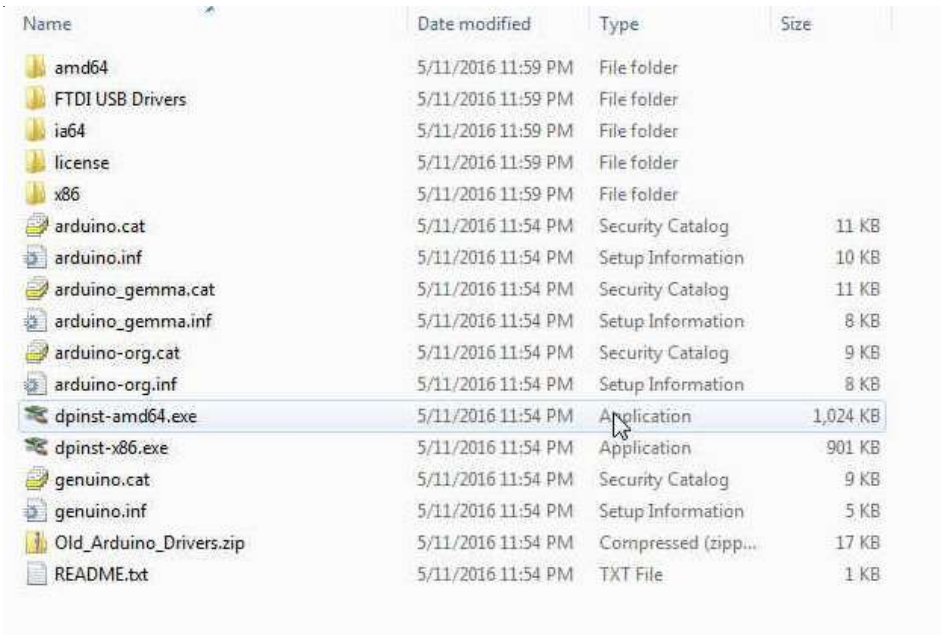



***Figure 5. The drivers subdirectory***

Sometimes the Windows environment does not find the proper port. This is usually because the device driver for the port isn't found. If this happens, go to your installation directory and into the *drivers* subdirectory (e.g., C:\Arduino1.6.9\drivers) and run the driver installation program that's appropriate for your system. (Figure 5 shows the *drivers* subdirectory contents.)

Once the driver is installed, the program should now find your Arduino COM port.

**The Integrated Development Environment (IDE)**

The easiest way to test that everything installed correctly is to run a program. Your IDE has a program called Blink. You can find it using the menu sequence: File --> Examples --> IO. Basics --> Blink. You can see this sequence in Figure 6. Once you click on Blink, the IDE loads it into the Source Code window.

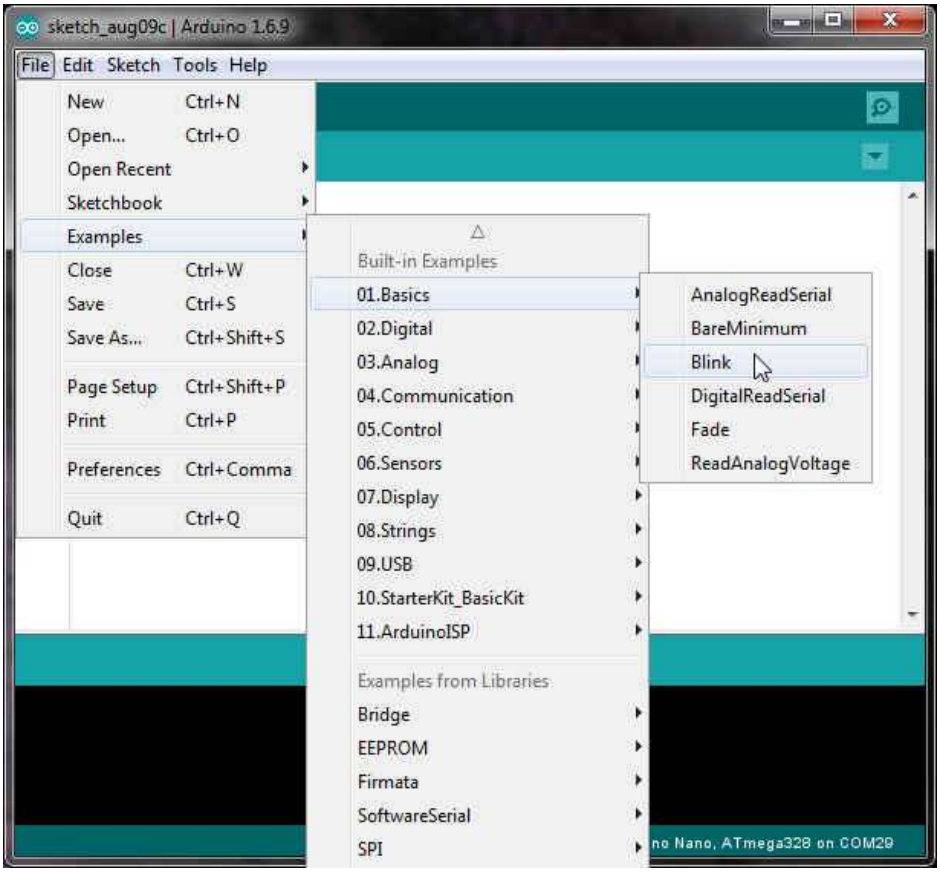



***Figure 6. Loading the Blink program.***

Once the Blink program is loaded, the IDE looks similar to Figure 7. I've marked some of the more important elements of the IDE in the figure. Starting near the top is the Serial Monitor icon. You click on this to see any print statements you might be using in the program. We'll show an example of such statements in a moment.

The large white space is where you will write your program source code. Program source code consists of English-like instructions that tell the compiler what code to generate. Because you already loaded the Blink program, you can see the Blink source code in Figure 7.

The Compile Icon is exactly that: It compiles your program. It does not, however, link all the parts of the program together to form an executable program. Using the Compile Icon is a quick way to see if your have the code syntax correct.

The Compile-Upload icon not only compiles your program, but it links it

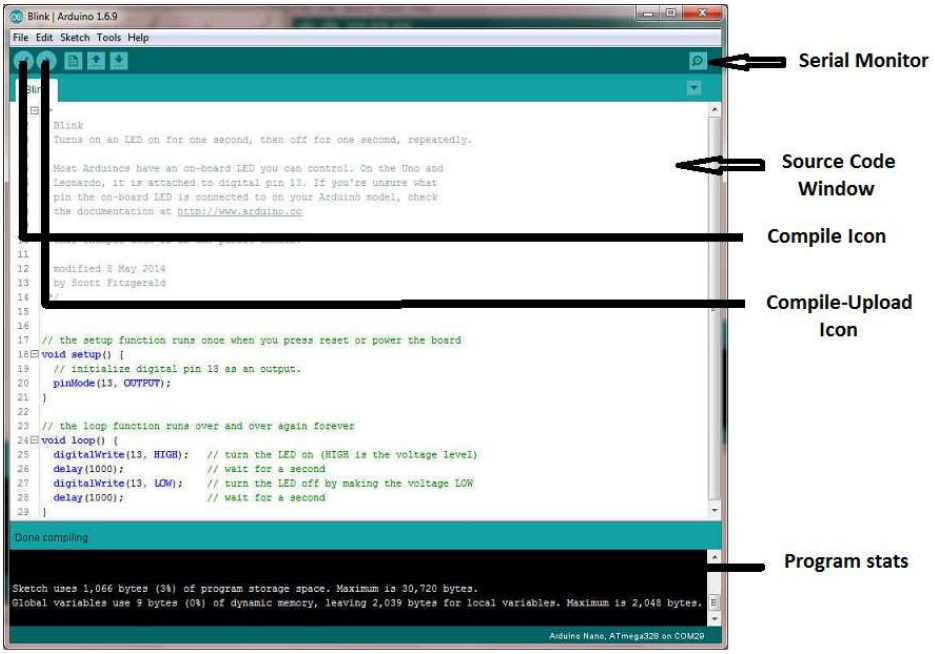



***Figure 7. The IDE.***

into an executable program and then transfers the program code to the Arduino via the USB cable. In other words, your PC is used to write, test, compile, and debug your program, but the code actually runs on the Arduino. The Arduino has a small program, called a bootloader, that manages all of the communications between your PC and the Arduino. You don't need to worry about it. (If you compile a program on a Nano, it will tell you that you have about 30K of program space even though it's a 32K memory bank. The "missing" 2K is the bootloader.)

The Program stats window tells you how much flash and SRAM memory your program is using. The window is also used to display error messages if the IDE detects something wrong with your program.

**Your First Program**

Let's look at the Blink program and make a couple of simple changes to it. The code is reproduced in Listing 1, with all of the comments stripped away. Look at the element called *setup()*. The element is

***Listing 1-1. Blink source code.***

```
void setup() {
  // initialize digital pin 13 as an output.
 pinMode(13, OUTPUT);
```

```
}

/
 the loop function runs over and over again forever
void loop() {
  digitalWrite(13, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);              // wait for a second
}
```

actually called a function. A *function* is a group of one or more program statements designed to perform a specific task. The purpose of the *setup()* function is to define the environment in which the program is to operate. Without getting too deep right now, *pinMode()* is also a function which has been predefined for you to establish how a given Arduino I/O pin is to operate. In this case, we are saying that we want to use pin 13 to output data. However, every Arduino happens to have a small LED available for use on pin 13. In this program, therefore, we are going to use pin 13 to output data. As it turns out, the output data consists of blinking the LED.

One thing that it unique about *setup()* is that it is only called once, and that is when you first apply power to the Arduino or you press its Reset button. Once it has defined its operating environment, its task is complete and it is not recalled.

The *loop()* function, however, is called continuously. If you look at the code, the first statement is a call to the *digitalWrite()* function. It has two function arguments, 13 and HIGH. These two pieces of information (i.e., the pin number and its state) are needed by the *digitalWrite()* function to perform its task. The function's task is to set the state of pin 13 to HIGH. This has the effect of turning the LED on. The next program statement, the *delay()* function call, has a number between its parentheses. This number (i.e., 1000) is called a *function argument* which is information that is passed to the *delay()* function code which it needs to complete its task. In this case, we are telling the program to delay executing the next program statement for 1000 milliseconds, or one second.

After one second, another *digitalWrite()* function call is made, only this time it sets the state of pin 13 to LOW. This turns off the LED tied to pin 13. Once the LED is turned off, the program again calls delay() with the same 1000 millisecond delay. So, collectively, the four program statements in *loop()* are designed to turn the LED on and off with a one second interval. That almost sounds like blinking the LED, right?

Now here's the cool part. After the last *delay()* is finished, the program loops back up to the top of the *loop()* function statements and re-executes the first program statement, *digitalWrite(13, HIGH)*. (The semicolon at the end of the line

marks the end of a program statement.) Once the LED is turned on, *delay()* keeps it one for once second and the third program statement is again executed.

Now press the Compile-Upload icon and after a few seconds you will see a message saying the upload is done. If you look at your Arduino, it's now sitting there blinking its LED for you.

Your program repeats this sequence until: 1) you turn off the power, 2) you press the reset button (which simply restarts the program), or 3) there is a component failure. If your program is performing as described here, you have successfully installed the IDE and compiled your first program.

**A Simple Modification**

Let's add two lines to make this "your" program. Move the arrow cursor into the Source Code window. The cursor will change from an arrow to an I-bar cursor. Now type in the new lines shown in Listing 2.

***Listing 1-2. Blink modifications.***

```
void setup() {
  // initialize digital pin 13 as an output.
  Serial.begin(9600);
 Serial.print("This is Jack, W8TEE);
 pinMode(13, OUTPUT);
}
```

Obviously, you should type in your name and call. The first line says we want to use the IDE's Serial object to talk to your PC using a 9600 baud rate. The Serial object has a function imbedded within itself named *begin()*, which expects you to supply the correct baud rate as its function argument. The second line simply sends a message to your PC at 9600 baud over the Serial communication link (i.e., your USB cable). If you click on the Serial Monitor icon (see Figure 7), you will see the message displayed in the Serial monitor dialog box. At the bottom of the box are other baud rates that are supported by the IDE. The *begin()* baud rate and the rate shown at the bottom of the box must match. If they don't, your PC will blow up! Naw...just kidding. Actually, you may not see anything at all or you may get characters that look a lot like Mandarin.

**Conclusion**

That's enough for now. Try adding some more *print()* function calls using the Serial object. Put one inside *loop()*. What happens? What happens if you use *println()* instead of *print()*? Experiment and enjoy!

## **Arduino for the Terrified**, Part 2

by Jack Purdum, Ph.D., W8TEE

By the end of the first installment of this series, you had the Arduino IDE installed and had modified the Blink program. Hopefully, you also played around with the *Serial.print()* and *Serial.println()* functions and found out how easy it is to communicate between the Arduino and your PC. Now what?

Well, what do you want to do? My guess is that some of you started reading this series because you have something in mind that you'd like to build. Great! Hopefully, a good number of you just might be curious enough about uC's to spend $3 on a Nano, but don't have any project in mind at the moment. Also great! Still, a lot of people stall out at this point because they don't know how to move from point A to point D. They know there are steps in between, but they are not sure what those steps are. Rest assured, you're not alone. What we need to do is provide a general framework in which you can view all programs, regardless of their complexity. That framework is what I call The Five Program Steps.

### The Five Program Steps

It doesn't matter how simple or complex an application is, it can be simplified into five steps. These five steps allow you to organize your thoughts about the program you want and shows you how to move from point A (i.e., program design) to point D (i.e., a program that performs a task).

#### *Step 1. Initialization*

This step performs all those tasks that might be necessary to create the environment in which the program will operate. If you are working with sensors, perhaps they need certain tasks performed before they can be used. If you're working with the random number generator, it should be "seeded" before it is used. If you're working with stepper motors, perhaps the stepper needs to be moved to the "home" position. If you're going to use your phone to control a transceiver, perhaps some port needs to be initialized in some way. If you want to print something, perhaps a printer port or database connection needs to be activated. Note that all of these things likely take place before the user sees anything visible on a display screen.

If you have used a word processor or a presentation program, it's common for the four or five most-recently accessed files to be appended to the bottom of the File menu of the application. Clearly, that program had to read that file list from somewhere and tack it onto the File menu. This is a typical Step 1 task.

In Part 1 you saw how we had a statement in the *setup()* function that set the Serial object's baud rate to 9600 before we could use it. That was a Step 1 task. We also established that we wanted pin 13 to function as an output device. That, too, was a Step 1 task. We also mentioned in Part 1 that the

*setup()* function is only executed once, when the program first begins execution. Hmmm. Sounds like *setup()* would be a good place to put all of your Step 1 tasks. While there are exceptions to that rule, for now, let's think of all initialization tasks as being placed in the *setup()* function.

**Step 2. Input**

Reduced to its simplest terms, most programs take data in one form, chew on it, and output new data based on the input data. It makes sense, therefore, that Step 2 be associated with the step that gets the raw data into the program. While we often think of input as coming from the keyboard, with uC programs, the data source can be anything from a heat sensor to a retinal scanner. The input can also be from another program. Your transceiver may well have software in it that sends commands out over a USB port and that data may well be your input data.

Input data tends to fall into one of two catagories: 1) textual, 2) numeric. Textual data, like you pressing the letter 'A' on a keyboard, actually is also numeric, but it needs to be decoded. For example, if you send a capital 'A' from the keyboard, what your PC actually receives is the number 65. When the number 65 arrives from the keyboard in your PC, your operating system (e.g., Windows, Linux, etc.) looks up the character code associated with 65 and displays an 'A' on your display. The character codes come from the American Standard Code for Information Interchange, or ASCII. Table 2-1 shows the ASCII character set. Note how 65 (decimal) is the letter 'A'. What character do you get if you add 32 to the ASCII codes for 'A', 'B', and 'C'?

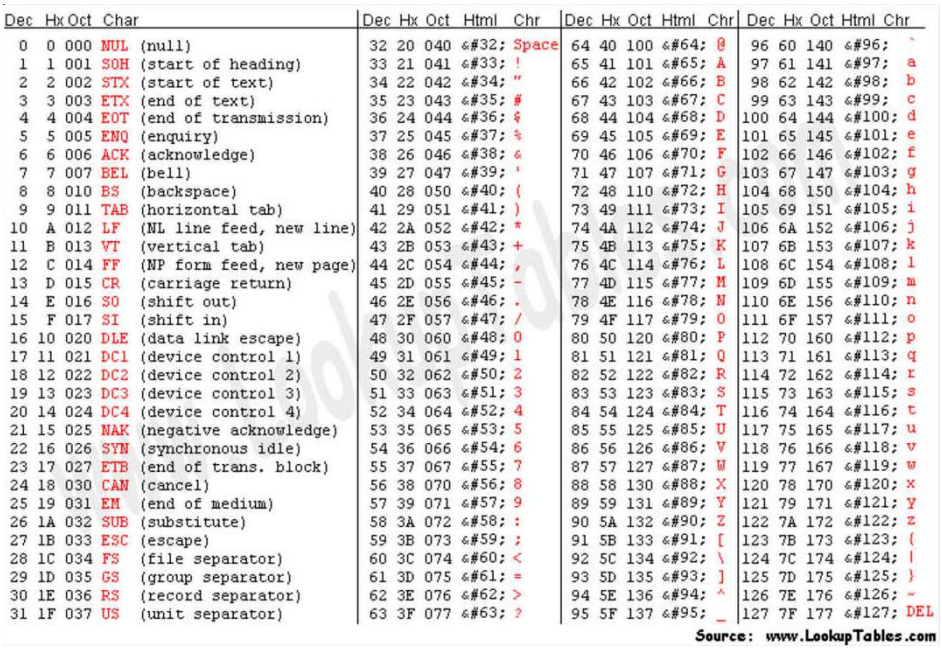
It's important to note that digit characters are different than "pure" numbers. That is, when you see a '0' (i.e., zero) pop up on your screen, that is actually the digit character 48. If you see a '9', it's ASCII 57. If you want to perform some kind of math operation on data coming from the keyboard, you need to convert it from ASCII to a "pure" number.

Also, keep in mind that data sources can mess up. People mistype things, they push the wrong buttons, sensors fail, and people answer questions wrong. Twice I had data where the respondent was pregnant, but stated that her gender was 'M'. The point is, just getting the input data probably isn't enough to actually use it. You probably need to check it to ensure that it is "correct". This usually involves range checking (Really? You're 212 years old?), or consistency checks (you're male, but pregnant...really?)

Once you have the raw data and have performed whatever checks you think are needed, you're ready for Step 3.

**Step 3. Process**

Once you have collected and verified the data, the program transforms that data into something new. That "something" is what you've designed the program to do. Perhaps you're reading two temperature sensors and when their



| Dec | Hx | Oct | Char | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 000 | NUL (null) | 32 | 20 | 040 |   | Space | 64 | 40 | 100 | @ | @ | 96 | 60 | 140 | ` | ` |
| 1 | 1 | 001 | SOH (start of heading) | 33 | 21 | 041 | ! | ! | 65 | 41 | 101 | A | A | 97 | 61 | 141 | a | a |
| 2 | 2 | 002 | STX (start of text) | 34 | 22 | 042 | " | " | 66 | 42 | 102 | B | B | 98 | 62 | 142 | b | b |
| 3 | 3 | 003 | ETX (end of text) | 35 | 23 | 043 | # | # | 67 | 43 | 103 | C | C | 99 | 63 | 143 | c | c |
| 4 | 4 | 004 | EOT (end of transmission) | 36 | 24 | 044 | $ | $ | 68 | 44 | 104 | D | D | 100 | 64 | 144 | d | d |
| 5 | 5 | 005 | ENQ (enquiry) | 37 | 25 | 045 | % | % | 69 | 45 | 105 | E | E | 101 | 65 | 145 | e | e |
| 6 | 6 | 006 | ACK (acknowledge) | 38 | 26 | 046 | & | & | 70 | 46 | 106 | F | F | 102 | 66 | 146 | f | f |
| 7 | 7 | 007 | BEL (bell) | 39 | 27 | 047 | ' | ' | 71 | 47 | 107 | G | G | 103 | 67 | 147 | g | g |
| 8 | 8 | 010 | BS (backspace) | 40 | 28 | 050 | ( | ( | 72 | 48 | 110 | H | H | 104 | 68 | 150 | h | h |
| 9 | 9 | 011 | TAB (horizontal tab) | 41 | 29 | 051 | ) | ) | 73 | 49 | 111 | I | I | 105 | 69 | 151 | i | i |
| 10 | A | 012 | LF (NL line feed, new line) | 42 | 2A | 052 | * | * | 74 | 4A | 112 | J | J | 106 | 6A | 152 | j | j |
| 11 | B | 013 | VT (vertical tab) | 43 | 2B | 053 | + | + | 75 | 4B | 113 | K | K | 107 | 6B | 153 | k | k |
| 12 | C | 014 | FF (NP form feed, new page) | 44 | 2C | 054 | , | , | 76 | 4C | 114 | L | L | 108 | 6C | 154 | l | l |
| 13 | D | 015 | CR (carriage return) | 45 | 2D | 055 | - | - | 77 | 4D | 115 | M | M | 109 | 6D | 155 | m | m |
| 14 | E | 016 | SO (shift out) | 46 | 2E | 056 | . | . | 78 | 4E | 116 | N | N | 110 | 6E | 156 | n | n |
| 15 | F | 017 | SI (shift in) | 47 | 2F | 057 | / | / | 79 | 4F | 117 | O | O | 111 | 6F | 157 | o | o |
| 16 | 10 | 020 | DLE (data link escape) | 48 | 30 | 060 | 0 | 0 | 80 | 50 | 120 | P | P | 112 | 70 | 160 | p | p |
| 17 | 11 | 021 | DC1 (device control 1) | 49 | 31 | 061 | 1 | 1 | 81 | 51 | 121 | Q | Q | 113 | 71 | 161 | q | q |
| 18 | 12 | 022 | DC2 (device control 2) | 50 | 32 | 062 | 2 | 2 | 82 | 52 | 122 | R | R | 114 | 72 | 162 | r | r |
| 19 | 13 | 023 | DC3 (device control 3) | 51 | 33 | 063 | 3 | 3 | 83 | 53 | 123 | S | S | 115 | 73 | 163 | s | s |
| 20 | 14 | 024 | DC4 (device control 4) | 52 | 34 | 064 | 4 | 4 | 84 | 54 | 124 | T | T | 116 | 74 | 164 | t | t |
| 21 | 15 | 025 | NAK (negative acknowledge) | 53 | 35 | 065 | 5 | 5 | 85 | 55 | 125 | U | U | 117 | 75 | 165 | u | u |
| 22 | 16 | 026 | SYN (synchronous idle) | 54 | 36 | 066 | 6 | 6 | 86 | 56 | 126 | V | V | 118 | 76 | 166 | v | v |
| 23 | 17 | 027 | ETB (end of trans. block) | 55 | 37 | 067 | 7 | 7 | 87 | 57 | 127 | W | W | 119 | 77 | 167 | w | w |
| 24 | 18 | 030 | CAN (cancel) | 56 | 38 | 070 | 8 | 8 | 88 | 58 | 130 | X | X | 120 | 78 | 170 | x | x |
| 25 | 19 | 031 | EM (end of medium) | 57 | 39 | 071 | 9 | 9 | 89 | 59 | 131 | Y | Y | 121 | 79 | 171 | y | y |
| 26 | 1A | 032 | SUB (substitute) | 58 | 3A | 072 | : | : | 90 | 5A | 132 | Z | Z | 122 | 7A | 172 | z | z |
| 27 | 1B | 033 | ESC (escape) | 59 | 3B | 073 | ; | ; | 91 | 5B | 133 | [ | [ | 123 | 7B | 173 | { | { |
| 28 | 1C | 034 | FS (file separator) | 60 | 3C | 074 | < | < | 92 | 5C | 134 | \ | \ | 124 | 7C | 174 | | | \| |
| 29 | 1D | 035 | GS (group separator) | 61 | 3D | 075 | = | = | 93 | 5D | 135 | ] | ] | 125 | 7D | 175 | } | } |
| 30 | 1E | 036 | RS (record separator) | 62 | 3E | 076 | > | > | 94 | 5E | 136 | ^ | ^ | 126 | 7E | 176 | ~ | ~ |
| 31 | 1F | 037 | US (unit separator) | 63 | 3F | 077 | ? | ? | 95 | 5F | 137 | _ | _ | 127 | 7F | 177 | &#127; | DEL |

Source: www.LookupTables.com

sums equal a certain value, you open valves to mix the two vats. Perhaps the

*Table **2-1. The ASCII character set.***

data is the amount of sunlight falling on a photoresistor and you rotate a solar collector based on the interpretation of the resistance coming from the photoresistor. Perhaps your program reads button presses and, if they are in the right sequence (e.g., a password), you open a lock.

Some form of plan, or *algorithm*, directs the process. If you're reading temperature sensors that return Celsius temperatures and you want to display in Fahrenheit, an equation becomes the algorithm that directs the solution. In all cases, some form of algorithm, or plan, directs the Process Step. When completed, the Process Step should provide a solution to whatever the program was designed to address.

**Step 4. Output**

A solution that no one knows about it worthless. You need some way to convey the solution to the outside world. Your first program blinked an LED as its output. You then modified the program slightly to output a short message to the Serial monitor. However, keep in mind that the output of one program can (and often is) the input to another program. Outputting a patients name to a database may be your program's goal, but that name become an input for the hospital billing cycle. The output of a photoresistor can become the input for another program that controls a stepper motor.

More commonly, your program's output to the Serial monitor or some

kind of display device. Two and four line LCD devices are cheap and provide a mean of displaying output without having to tie into a PC. (We're going to do that in Part 3.) Usually, the nature of the problem will dictate the type of output device that is needed.

**Step 5. Termination**

My beginning programming students thought the Termination Step simply meant turning the computer off. Wrong! Programs should end gracefully and that often means "undoing" what was done in the Initialization Step. If you opened a printer port, you should release that port. If you opened a file system or a database, you should close those files. If your application shows the last five files the user accessed when the program starts, that list needs to be updated before the program ends.

That said, most uC programs are different: They are often designed to run forever. If the Empire State building has 10,000 fire sensors, you don't turn those off at 5:00PM. Traffic lights run all day and all night. Refrigerators and many other appliances are designed to run forever...even if they don't. For most of the programs we will write, there is no formal Termination Step. So, even though we won't be using it, we need to keep it in mind.

**A Complete Program**

Let's write a complete program of our own. By complete, I mean that it requires input from an external source and output that is a little more sophisticated that an LED. Let's write a simple game where the uC tries to guess a number that you are thinking of, and that number must be within the range of 0 through 100. The uC presents you with a guess, and you must respond with 'H' when the guess is too high, 'L' when its guess is too low, and 'C' when it nails it. The uC's object is to minimize the number of guesses it needs to determine the number.

**Step 3,**

Process is where we have all kinds of choices. After all, the Input Step is simply going to read a letter that you enter into the program via the Serial monitor. The Display Process will display the uC's guess and you will respond with 'H', 'L', or 'C'. The Initialization Step needs to establish a Serial object so you can enter a letter from the keyboard and the uC can display a guess. If you want to, how about flashing the LED when the uC gets the correct answer?

**Step 3, Process, But Using Which Algorithm?**

So, the idiot's way to approach this is for the uC to start with 0 and ask it that's the number. If not, advance by one and ask if 1 is the correct number, and so on. On average, it will take 50 guesses to get the number correct. It works, but it's as dumb as a bag of hammers.

This type of problem is well-suited to using what is called a binary

search algorithm. Simply stated, it involves continually dividing the list to search in half with each guess until the number is isolated. For example, let's say your number is 73. Using a binary search, the uC's first guess should be 50. Since the guess is too low, you would enter L. Immediately, the uC can throw away half the list; 0-50. The uC then takes the remainder of the list, 51-100, divides it in half and guesses 75 (i.e., 51 + 100 = 151 / 2 = 75.5 = 75, integer math drops, not rounds, the fractional component). Now you would respond with H. Note that after only two guesses, all values from 0-50 and 75-100 have been eliminated. As a general rule, the binary search algorithm will find the number in about 7 guesses.

So, let's start with Step 1, Initialization.

**Initialization Step**

All Arduino programs ***MUST*** have a *setup()* and a *loop()* function. You already know that *setup()* only executes once, so that would be the proper place to initialize our Serial communication object. So, our code looks like:

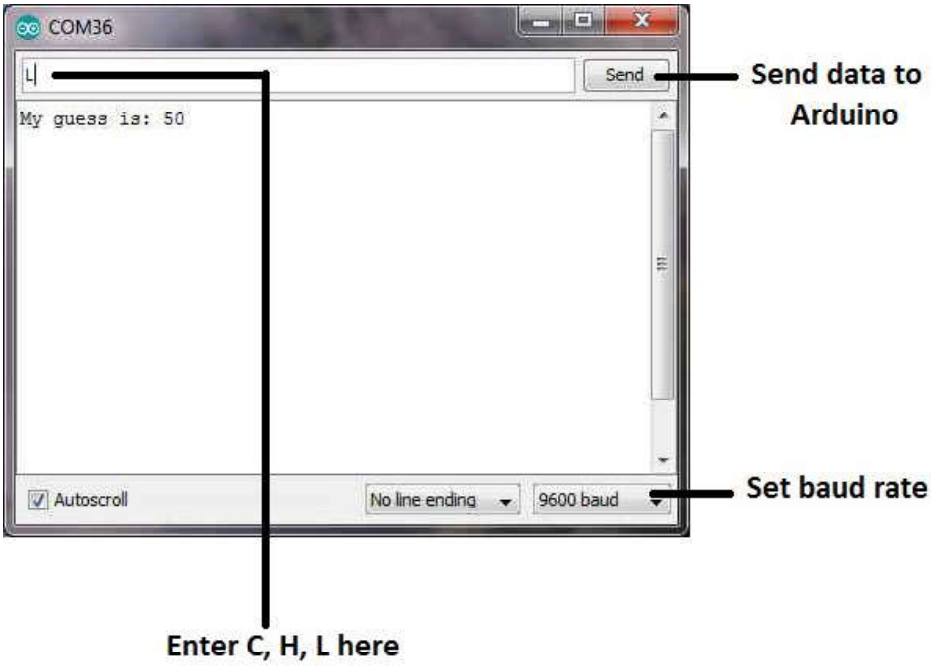
```
void setup()
{
  Serial.begin(9600);           // Serial com initialization
  pinMode(13, OUTPUT);          // If you want to flash the LED when correct
}
```

You'll notice some comments on each of the lines. The double-slash characters (//) mark the start of a comment to the end of that line. You can write anything you want after the comment characters, but usually comments are used to provide details about what is going on. Comments have no effect on executable code size or speed. That's it for the Initialization Step.

**Input Step**

We need to be able to collect a keystroke from the PC's keyboard to make our program work. Because communicating with the keyboard is such a fundamental task, the Arduino gives us the Serial object already built in to the IDE. Figure 2-1 shows how the Serial Monitor looks on your PC. The top textbox is where you enter the data you want to send to the uC. Once your data is entered, you can click the Send button, or just press the Enter key on your keyboard. The data are then moved to the uC at the baud rate seen at the bottom of the monitor. You also have several options for the way you want the data terminated using the dropdown box. I've selected No line ending. So, if I press the Enter key, my L in the input box is sent to the uC at 9600 baud. Then what?

Well, surely the Arduino people provided some means by which to read characters coming from the keyboard. If you look at:




https://www.arduino.cc/en/Reference/Serial

***Fig. 2-1 Serial Moniter on Screen***

you will find documentation on the functions that are part of the Serial object. One of those functions is named *available()*. Simply stated, when you touch a key on the PC's keyboard and the Serial object is active via the USB cable, the key you touched sends the ASCII code into a temporary holding spot, called a *buffer*, in the uC's memory. The *available()* function continuously monitors that buffer and returns a positive number if something is in it, or a zero if it is empty. Sounds promising. Therefore, we write:

```
void loop() {
  char letter;

  if (Serial.available() > 0) {  // Anything sitting in the buffer?
    letter = Serial.read();          // Yep! Read the ASCII character, and
                                     // stuff into letter
  }
}
```

The *if* statement block you see above in *loop()* is a C programming language keyword that tests to see if an expression is logic True or logic False. In the code above, the *if* statement is asking if *available()* is sending us a non-

zero value. If it is True that something is sitting in the buffer, *available()* will send us a non-zero value, making the *if* test True. If that *if* test is True, we execute all statements that fall between the opening ({) and closing (}) braces. This means that we call another function, *read()*, that is part of the Serial object. The *read()* function removes the first character it finds in the Serial buffer and assigns that ASCII character into the variable we define named *letter*.

In the C programming language, you must define each variable before you use it, which is what the

***char letter;***

statement does. A *char* data type occupies 1 byte (i.e., eight bits) of memory, so one byte of memory is set aside for use as *letter*. A *char* is capable of holding the numeric values between 0 and 127. Some of you might be saying: "Wait a minute! $2^8$ is 256, not 128." Well, that's true, but C reserves the high bit for use as a sign bit, which means a *char* can have negative values. A a result, we really only have 7 bits, or 128 values. Since zero is a valid number, the range is 0-127. Wait a minute...that looks familiar! Look again at Table 1. What's the largest value in that table? You don't suppose...

**RDC**

So far, we have what I call RDC...Really Dumb Code. The reason is because we are talking about reading input from the Serial object, which means the user must be responding to the uC's guess. The problem is I haven't written any code to show them my guess. Duh...

Listing 1 shows a fairly complete, albeit RDC, iteration of the program. At the top of the program I've defined three variables: *guess*, which will hold the computer's guess for the number the user is thinking of, *rangeStart*, which is the lowest number used to construct the guess, and *rangeEnd*, which is the highest number used to construct the guess.

***Listing 1. Guess Game***

```
int guess;       // Current guess by computer
int rangeStart;  // lower guess limit — may move as game progresses
int rangeEnd;    // upper          "

void setup() {
 Serial.begin(9600);
 rangeStart = 0;
 rangeEnd = 100;
 guess = (rangeEnd - rangeStart) / 2;        // My starting guess
 Serial.print("My guess is: ");                    // Show it to the user
 Serial.println(guess);
}
```

```
void loop() {
  char letter;

  if (Serial.available() > 0) {
    letter = Serial.read();
    if (letter == 'H') {              // Too high
       rangeEnd = guess - 1;
       guess = (rangeEnd - rangeStart) / 2 + rangeStart;
    }

    if (letter == 'L') {              // Too low
       rangeStart = guess + 1;
       guess = ((rangeEnd + rangeStart) / 2);
    }

    if (letter == 'C') {              // Bingo!
       Serial.println("Correct. Start new game.");
       rangeStart = 0;
       rangeEnd = 100;
       guess = (rangeEnd - rangeStart) / 2;
    }

    Serial.print("My guess is: ");          // Show new guess
    Serial.println(guess);
  }
}
```

The *setup()* function performs the Initialization Step by establishing the Serial communications link with the PC at 9600 baud. It also initializes the three variables with numeric values used to construct the guess. We have made these variables integer (*int*) variables because we are only using whole numbers. Because an ***int*** uses 2 bytes of memory, we have 15 bits for the number and the high bit for the sign. So, using $2^{15}$, the range of an *int* variable is -32,768 to 32,767. Since our range of numbers is 0 through 100, an *int* will do just fine. (We could have used a *byte* or a *char* data type, too, since they cover our numeric range of values.) The last thing *setup()* does is display its guess on the Serial monitor.

Once the last statement in *setup()* is executed, program control immediately begins to process the *loop()* code. This is unchanging behavior: *setup()* is run once, and then *loop()* runs...forever! Indeed, the statements in *loop()* are run over and over until: 1) power is lost, 2) a reset is performed, or 3) a component fails. I think you understand the power thing. A reset can be done by pressing the small pushbutton you see on the Arduino board or by starting (or

restarting) the Serial monitor. A component failure is also obvious (e.g., look for white smoke!).

**SDC**

The program behaves as expected and compiled to 2444 bytes of flash memory, and used 242 bytes of SRAM. Not bad. Looking at the code, I label this SDC...Sorta Dumb Code. Why dumb? After all, it works and it doesn't use anywhere near all of the resources we have.

***Cleaning Up SDC***

Let's turn a critical eye to *setup()*. Pretty simple, so there's not much to go wrong. However, look at the last two lines in *setup()* and the last two lines in *loop()*. In both cases, they are the same. As you gain some programming experience, this kind of thing will cause a spot somewhere on your person to twitch. The reason is because we can avoid duplicating the statements by moving them into a function and calling that function where needed. So, let's write a new function and put it just after *loop()*:

```
/*
 Purpose: To display the current guess on the Serial monitor

 Paramter list:
  int g        the guess

 Return value:
  void
*/
void ShowGuess(int g)
{
  Serial.print("My guess is: ");
  Serial.println(g);
}
```

In C, the character pairs "/*" and "*/" mark the start and end of a *multi-line* comment. While we could have stuck five single-line comment characters "//" at the beginning of each line, multi-line comments are quicker to use in this case. Everything between the /* and */ is ignored by the compiler. The comments I made are not necessary, but are useful when I come back to this at some later time. Given enough time, (for me, a lunch break), even you will forget what the function does. The function header comment block tells me the purpose of this function, any data that must be passed to the function (g), and if any value is returned from the function. (More on that later.) The keyword *void* means nothing is returned from this function.

Once the function is written, we can use it in *setup()*:

```
void setup() {
 Serial.begin(9600);
 rangeStart = 0;
 rangeEnd = 100;
 guess = (rangeEnd - rangeStart) / 2;        // My starting guess
 ShowGuess(guess);
}
```

**Removing More SDC**

Did the program run correctly the first time you ran it, or did you have to press the Shift key to make sure you sent an 'H', 'L', or 'C'. We shouldn't require the user to worry about this, so add a new line as follows:

```
letter = Serial.read();
letter = toupper(letter);
```

The *toupper()* function takes the *letter* you pass to it, checks it and, if it is not a capital letter, it converts it to a capital letter and returns that new value to the program. For example, suppose the computer guess is too high and you press 'h' instead of 'H'. The lowercase 'h' will cause all of the if statement tests to fail and the computer will assume nothing's changed so it will repeat the same guess. Not good. The *toupper()* function looks at the letter you passed to it and, seeing the ASCII value 104, it knows that it was give a lowercase 'h'. So, the function subtracts 32 from the ASCII code, and returns 72 back to the program, which is then assigned back into letter. Now *letter* is an uppercase 'H'. Now, if the user enters 'I', 'h', or 'c', the program will work correctly.

Finally, we now execute a bunch of *if* statements to see what the user's response is. Suppose letter is an 'H'. Because the *if* test is True, we execute the statements within that *if*'s statement braces:

```
if (letter == 'H') {
   rangeEnd = guess - 1;
   guess = (rangeEnd - rangeStart) / 2 + rangeStart;
}
```

Since the first computer guess is 50, the first statement sets the *rangeEnd* variable to 49. After all, when *guess* was 50 it's too high, we can throw everything from 50 on up out the window. The next line calculates the next value for *guess*. (You can wade through the statement on your own to figure out the next guess.)

Now, here comes the really dumb part: If we know letter was an 'H', why

bother checking to see if *letter* is an 'L' or a 'C'? Both of those *if* tests are unnecessary because we know it's an 'H'. Since neither of those *if* statement tests can be True, why bother to check? So, let's fix that.

Now we could use what's called a cascading *if-else* statement block in the C programming language. It is a commonly use C construct, but... I hate cascading *if-else* statement blocks.

Simply stated, in terms of execution speed, they are not as efficient as what I am about to show you plus I think they are harder to read. So, if you want to know about cascading *if-else* statement blocks, you can look it up online. Instead, we're going to use a *switch/case* statement block. Our new code to replace the series of dumb *if* statement blocks is as follows:

```
switch (letter) {
  case 'H':      // guess was too high
    rangeEnd = guess - 1;
    guess = (rangeEnd - rangeStart) / 2 + rangeStart;
     break;

  case 'L':      // guess was too low
    rangeStart = guess + 1;
    guess = ((rangeEnd + rangeStart) / 2);
     break;

  case 'C':      // guess was correct
    Serial.println("Correct. Start new game.");
    InitializeData();    // Reset the data
     break;

  default:       // catch input from stupid players
     Serial.println("Illegal input");
     break;
}
```

Let's suppose that *guess* is correct, so *letter* equals 'C'. At the top of the statement block, the code checks the value of *letter* ('C') against each *case* parameter (i.e., 'H', 'L', and 'C'). Rather than messing around with the 'H' and 'L' cases, program execution jumps over the code associated with those parameters and immediately starts executing the statements in the 'C' *case* statement block. The program can behave this way because the compiler created what's known as a jump table of memory addresses where each *case* statement block lives in memory. If letter had been an 'L', program control would have immediately jumped to the memory address where the 'L' *case* code is stored.

I once saw some commercial banking software that had 31 "stupid" *if*

statement blocks; one for each day of the month. If it was the first day of the month, the code proceeded to perform 30 unnecessary (i.e., stupid) *if* tests. I was hired as a consultant by the bank and pointed this out in a code walk-through meeting, only to be told later by one of the programmers that the person who hired me wrote that stupid code. I was fired the next day. (Great programmers have really thick skin.) So, Listing 2-2 shows how our final program looks:

***Listing 2-2. PGC...Pretty Good Code***

```
int guess;       // Current guess by computer
int rangeStart;  // lower guess limit — may move as game progresses
int rangeEnd;     // upper              "

void setup() {
 Serial.begin(9600);
 InitializeData();
 ShowGuess(guess);
}

void loop() {
 char letter;

 if (Serial.available() > 0) {
  letter = Serial.read();
  letter = toupper(letter);
  switch (letter) {
    case 'H':      // guess was too high
      rangeEnd = guess - 1;
      guess = (rangeEnd - rangeStart) / 2 + rangeStart;
       break;

    case 'L':      // guess was too low
      rangeStart = guess + 1;
      guess = ((rangeEnd + rangeStart) / 2);
       break;

    case 'C':      // guess was correct
      Serial.println("Correct. Start new game.");
      InitializeData();     // Reset the data
       break;

    default:       // catch input from stupid players
      Serial.println("Illegal input");
       break;
```

```
   }
   ShowGuess(guess);
  }
}

/*****
 Purpose: To initialize the intial values for the data

 Paramter list:
  void

 Return value:
  void
*****/
void InitializeData()
{
  rangeStart = 0;                    // Lower guess limit
  rangeEnd = 100;                      // Upper    "
  guess = (rangeEnd - rangeStart) / 2;       // Starting guess
}

/*****
 Purpose: To display the current guess on the Serial monitor

 Paramter list:
  int g        the guess

 Return value:
  void
*****/
void ShowGuess(int g)
{
  Serial.print("My guess is: ");
  Serial.println(g);
}
```

You'll notice I added another function, too, named *InitiailzeData()*. See if you can figure out why on your own. Notice that the code size increased to 2504 bytes and the SRAM use went to 254 bytes. A good part of this is because I added a *default* statement block in the *switch* statement block. This gives the user a clue why *guess* doesn't change if they didn't enter one of the three letters.

**Conclusion**

Enough programming. Next time we are going to hang an LCD display on to your Arduino so it can be used to display information rather than displaying output on the Serial monitor on your PC. If you don't have an LCD display, I would suggest getting this one:

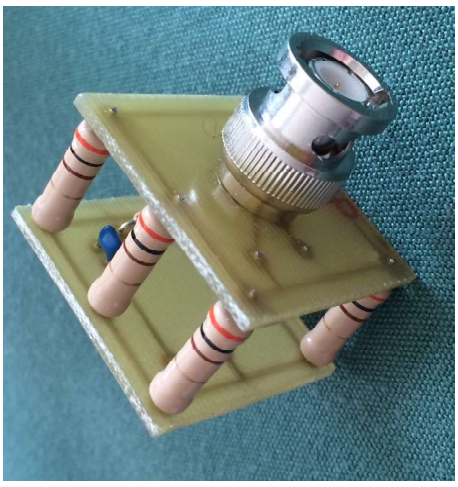
http://yourduino.com/sunshop2/index.php?l=product_detail&p=170

It's a 16x2 display with an I2C interface, which means you only need to tie up 2 I/O pins instead of 8 for displays that don't use the I2C interface. It's price is $5.50, which is pretty good, too.

## Pacificon 2016 Announcement
## Doug Hendricks, KI6DS

Pacificon will again be held in San Ramon, California at the Marriott Hotel on Oct. 15-17. We have lots of fun QRP Activities Scheduled. Here is the list:

Friday, Oct. 15
There are two events scheduled for Friday. First we will be distributing a free



**Prototype for Pacificon Dummy Load Kit.**
**Loops on back are for RF, RFProbe and Ground.**

Pacificon Dummy Load kit that will be used in the QSO Party scheduled for

Saturday night. All that you have to do to get your free kit is go to the Kit Building Booth located across the hall from the Vendor area. It is at the same location as always. If you are new to Pacificon, just walk down the hall from the main lobby. As you turn to your right you will run right into it. Ask at the booth for a free Dummy Load kit. You must agree to build it at Pacificon, and you must agree to participate in the QSO party Saturday night (if you didn't bring a rig, then you can participate by attending and observing). The kit is a 12W dummy load with rf probe built in. We will be attaching a clip lead for a radiator and using it for our antennas during the contest. It was designed by me and is provided by Hendricks Consutlting as a gift to you for attending Pacificon. Kits will be availabe Friday from 1-5 PM. Ask for Doug, Steve or Darrel.

The kit will be easy to build, as it consists of 7 resistors, 1 cap, 1 diode, 1 BNC connector and 2 pcb's. When you finish, you have a very nice QRP Dummy Load that will handle 12 Watts easily. Plus with the built in rf probe you have a power meter. The rf probe has been optimized to go with the Harbor Freight VOM Meter that they give away for free.

Friday night we will have the first night of our QRP Open House. This will be for Vendors, Show and Tell, and as a new attraction this year, a Chinese Kit Show and display.

Vendor night will work as usual. Everyone is welcome to come and set up goods for sale, whether you are a Commercial Vendor or not, it doesn't matter. The swap tables are free, first come first serve. We will have the tables arranged around the perimeter of the room with chairs in the middle.

Show and Tell will be for those of you who have built projects in the past year or so. I will be bringing my 1 Watter, as I want to show the case and paddles that I built. Many people say that the best part of the open house is the opportunity to see what others have built. Even if it is just a simple circuit that you have built on a home brew circuit board, we want to see it. Many ideas are generated by viewing the work of others. So, gather up the fruits of your labors over the past year and share it with us.

The Chinese Kit Show is an idea that I had. There are so many kits available from China at an unbelievable price. I have ordered at least 5 of them because I have seen them at qrp events. Steve Smith showed me a great power supply module, Mike Herr was responsible for my Transistor Checker, Chuck Adams showed me his capacitance meter, Bob Mix showed me the 49er, and Jon Kim shared the XL6009 adjustable buck booster. I can't explain it, but there is just something about seeing and holding a kit in your hand that makes it much more attractive to buy.

Here is what we want you to do to participate. Bring your built kits, along with a 1 page description for each kit of what it is, what it does, how well it works, item number on ebay or other web address, how much it costs. Tell us the pros and cons. Don't be afraid to use a larger size font so we can read it.

Saturday:
9:00 AM - 4:00PM Pacificon Dummy Load kits availabel from at the Kit Building Booth.
1:00 QRP Keynote Forum featuring Chuck Adams, K7QO. Check the forum schedule for room numbers. Try to be there early as seats will go fast.
7:00 - 10:00 The Great Pacificon QSO Party and Open House

The feature event Saturday night will be the QSO Party. Rigs must be a kit that you have purchased from China and built. The rigs use crystals, so we will be using 7.023 MHz, 7.030 MHz, 10.116 MHz and 14.058 MHz. The 7.023 crystal is in your kit, and I will supply the rest of them. You will need to modify your kit to have a sip socket where the crystal goes in order to be able to change frequencies. If the radio is in a case, there will be 5 bonus points added to the score. You may work each station on each frequency. Keyers are allowed. The antenna will be a Pacificon Dummy Load that you have built here at Pacificon. We will supply radiator wires. You must supply everything to get your rig on the air. The contest will last for 30 minutes.

We will also have the swap tables available after the contest if anyone wants to set up and sell. The rest of the time will be spent listening to the winners brag, the losers whine, and the observors telling us how they would have won.

The QRP Events are designed to get QRPers together to talk and visit. We used to have the building event during the open house, but realized that too much time was taken away from the opportunity to get to know each other by the event. So we decided to try something new this year. The Mount Diablo Amateur Radio Club has been very kind to provide us the meeting rooms at no charge again this year. They have shared the kit building booth with us. Supplying soldering stations and a place to work. That is a very kind gesture and is appreciated.

It is because of donations like the above that there is no charge for QRP Events at Pacificon. They are included in your ticket for admission. I like this format and plan on continuing using it. If you live in the bay area, and would like to get together with like minded QRPers, there is a monthly meeting held at Denny's Restaurant, 1140 Hillsdale Ave., in San Jose. The meeting starts at 6:30 PM with a no-host dinner, and we usually start the activities at 7:00. We usually have one forum per meeting, usually lasting about 30 minutes. We have an LCD projector and screen and a private room so presentations are easily done.

We also have access to a patio outside where we can set up antennas and do demonstrations. Ned Tully brings his vertical delta loop to provide HF coverage. I bring 20 kits to each meeting that are simple projects to build and have fun with. Members bring their completed kits to the next meeting to pass around for kudos. The meeting adjourns about 9 or 9:30. There is no business meeting or cost to attend, other than your meal. All are welcome. Please join us. 72, Doug

## Noise Generators, Part 2
## by Chuck Adams, K7QO

Last time we constructed a general purpose noise generator. It practically generated white noise from DC to daylight. In this article I'm going to show you just a few uses for the instrument and then talk about an S9 signal generator.

Visualize noise from below the AM broadcast band up to the 2m amateur ham band. Think of it being the same signal strength at any segment of the HF frequency spectrum.

I showed you last time what the signal looks like in a 2MHz bandwidth using the SDR Play receiver. Let's now look at what the signal looks like through some common QRP transceivers that you may already own.

I'll start with the popular Dave Benson, K1SWL,SWL-40+ transceiver for 40 meters. I will take one off the shelf that hasn't been fired up in years. Just to demo how a noise generator works and to show how some alignment and tweaking can be done to a rig that has not been used in a long, long time. I know you've had some rigs that have set for sometime. It is most likely they will not be exactly up to par than when they were first built.

You can also use the same techniques here to checkout used equipment that you have purchased for some great price at a swap meet. Been there. Done that. Have the rigs to prove it.

The noise generator will be used to check the receiver only. Be sure, and I do mean be sure, to remove any keying mechanism from the transmitter. Any straight key, external keyer and if the transceiver has a built in keyer, then by all means remove the dual lever paddle. Oh. And you bug users disconnect the bug. For this experiment you will need the following.

1. Transceiver or Receiver to be checked.
2. Noise Generator.
3. Stereo audio cable with 3.5mm male connectors on each end.
4. Computer
5. Audio spectrum software to analyze the output from the receiver.

I will be using a program called baudline. You can search the Internet for audio spectral display software compatible with your operating system and hardware. You want a visual display that shows the amplitude of the signal at each audio frequency from zero to 20KHz. Since the sound cards typically sample at 44.1KHz, then the Nyquist frequency is half that or 22.05KHz and that will be the upper limit unless you happen to be sampling at a higher rate. You can't hear that high any way.

A warning. Use ESD (anti-static care) procedures so that you do not zap your computer internals with a 20KV or higher voltage spike with static electricity.

First, with the software running and nothing on the microphone input take a screen snapshot of the display as a base line. This is to determine what the minimum noise level is for the soundcard.

I plug the cable into the audio out of the receiver first and then into the computer microphone jack. With receiver off and gain on minimum for the software. Just to be safe. The computer is probably worth more than the transceiver. The audio cable is the one used for connecting MP3 players and such to AUX jacks on your car stereo or into externally powered speakers for room filling volume.

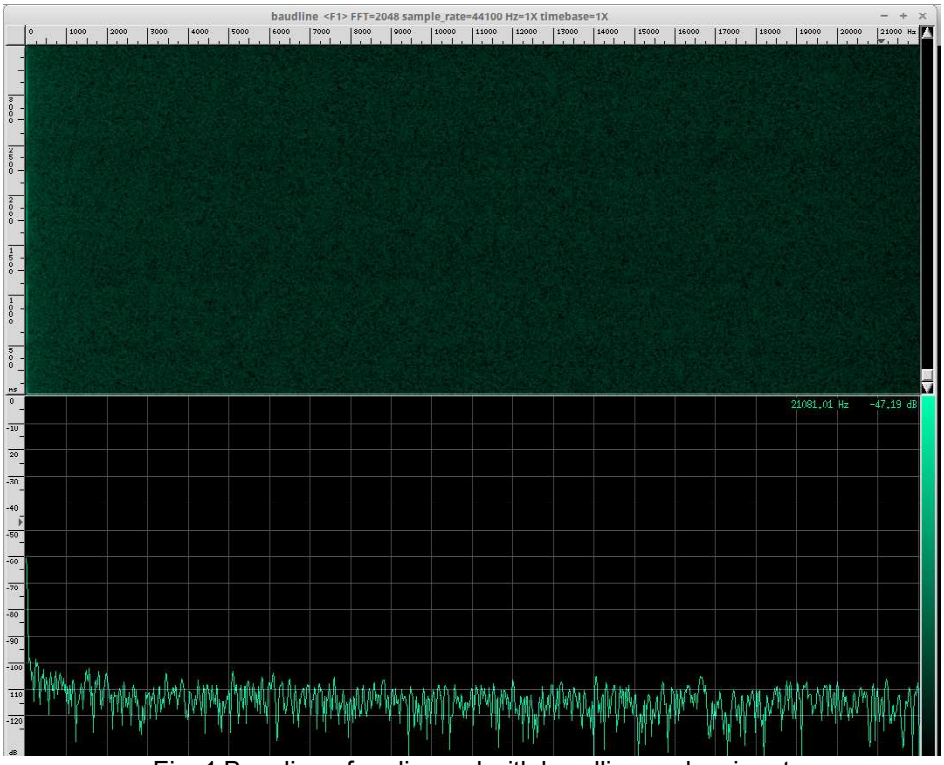



Fig. 1 Baseline of audio card with baudline and no input

You can see that the signal level is pretty flat just below the -100dB line with a spike at 0Hz due to fft calculations. So this is what a quiet sound card has as sound frequencies that produce the slight hiss, if audible with sound cranked up. By the way. Never put on headphones or use speakers where you do not have the sound at a minimum when turning on the power. How many sets of ears do you get in a lifetime?

I took two old SW-40+ 40 meter transceivers off the shelf. Don't have the date I put them together, but I guarantee you it is a long time ago. I connected up the first one and powered it up with volume on minimum. Plugged in

the audio connecting cable into the computer and here is the image with no signal and minimum gain on the audio. All noise is from the internal thermal noise of the components. Everything. You see a hint of where the audio bandpass and IF filtering is centered around 1KHz. This may be too high for your CW tastes, but because of my QRQ trainingand preference I find I copy better at the higher audio tone. It isn't for everyone. That's OK. I can listen to the lower tones also.

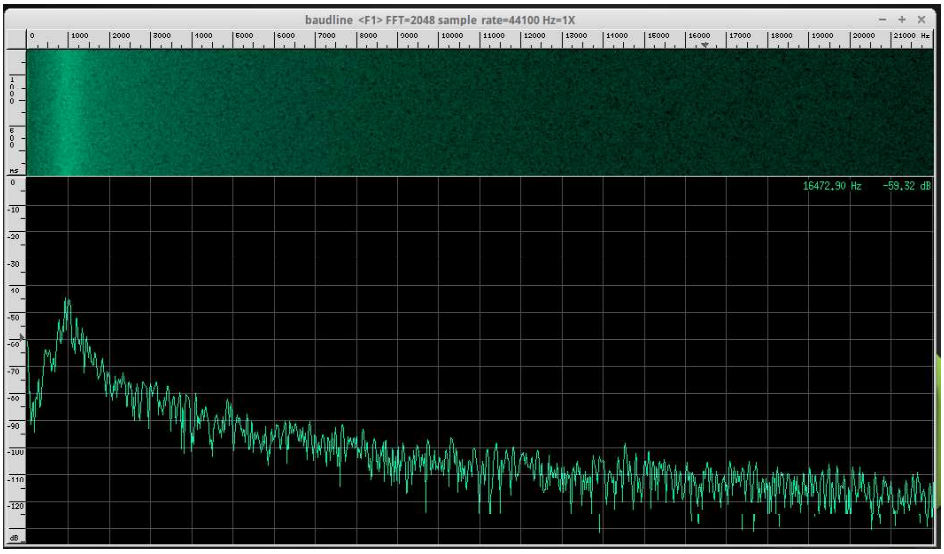



Figure 2. Base line for a SW-40+ transceiver from Small Wonder Labs.

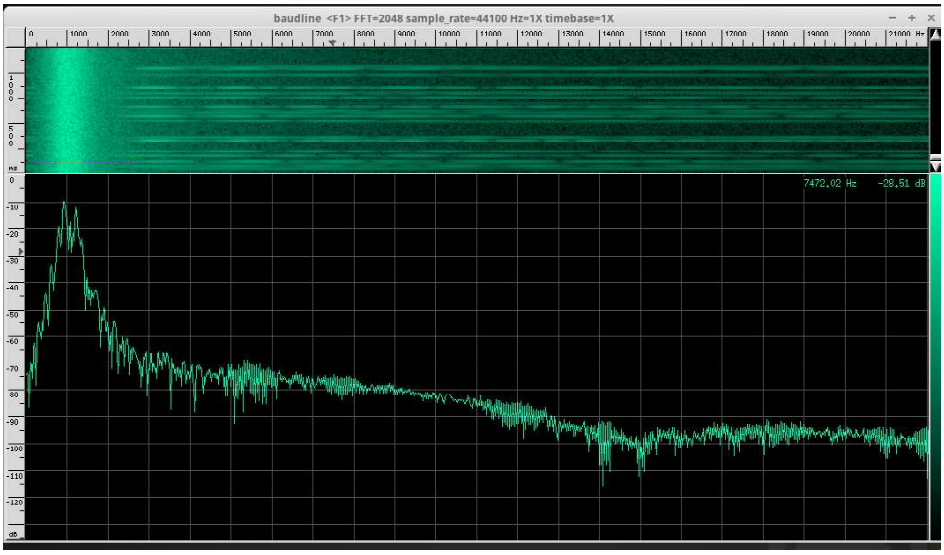



Figure 3. SW-40+ transceiver with +30dB/S9 noise level input at antenna.

The next figure shows the audio output with the 30db/S9 noise level into the receiver. Quite a difference, wouldn't you say? Looking at the -30dB points on the left hand scale and the signal level I'm guessing at around 750Hz band pass on the IF. Not bad for a transceiver that costs easily less than $100 back in the day.

The second SW-40+ transceiver with the same noise level shows that this one has a lower center frequency for the bandpass and about the same bandwidth.

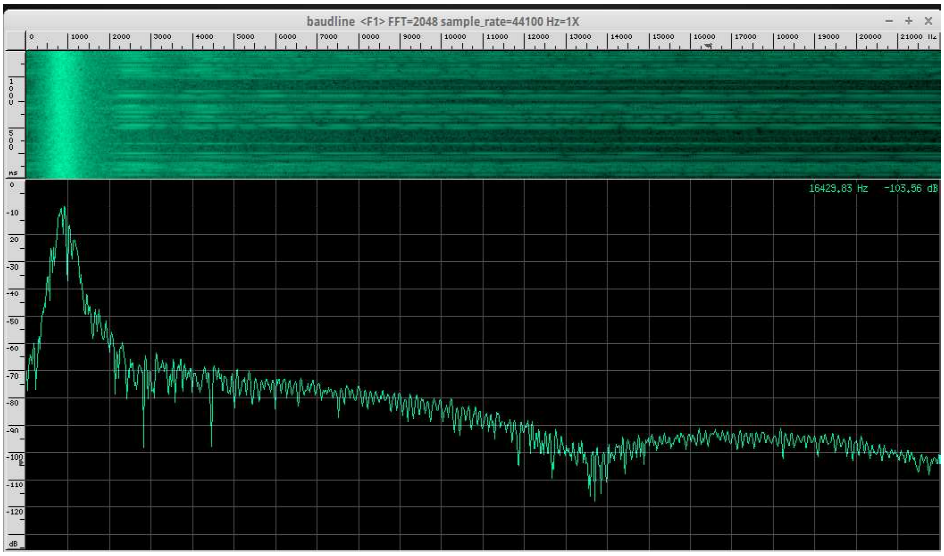



Figure 4. Second SW-40+ transceiver with
+30dB/S9 noise level input at antenna.

Since Doug, KI6DS, is looking for articles. Here is something that you can do to become famous. Build a K7QO Noise Generator. Get some audio analysis software, preferably freeware for your computer and take some of your receivers and do the same measurements. Write up your results and email them to Doug, KI6DS at KI6DS1@gmail.com

Consider this a high school science experiment. Show us your stuff. Doug will clean up any issues with writing.

And, like a kid with a new toy, I just had to do two more transceivers for grins. I took an old Flying Pig Version 2 transceiver for 7.122MHz from Diz, W8DIZ, at kitsandparts.com and did the baseline. Quieter receiver due to some component values on the LM386. Yet another science experiment for a writeup.

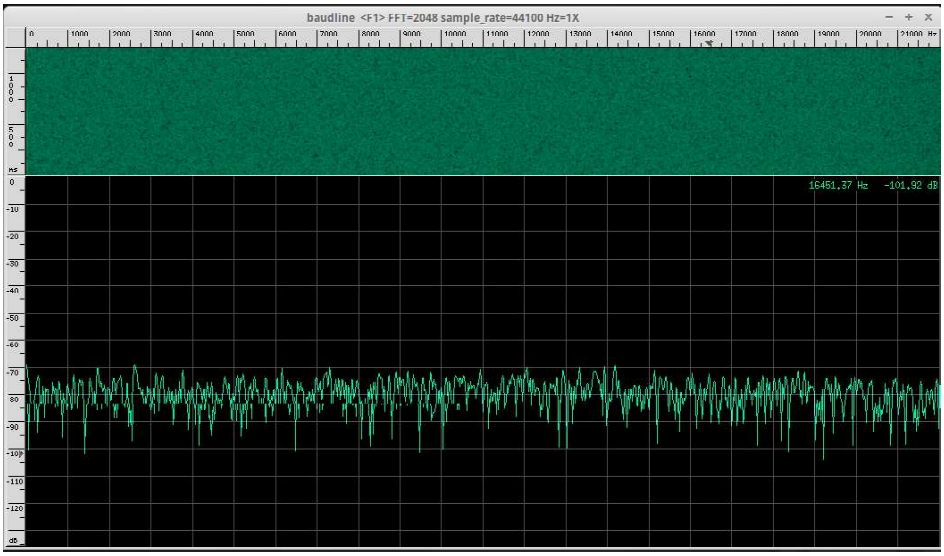



Figure 5. Flying Pig Version 2 7.122MHz transceiver baseline.

Feeding it the noise source we see that it has a very narrow bandpass. I'll do another experiment in next months column.

And here is the rig with the noise input.

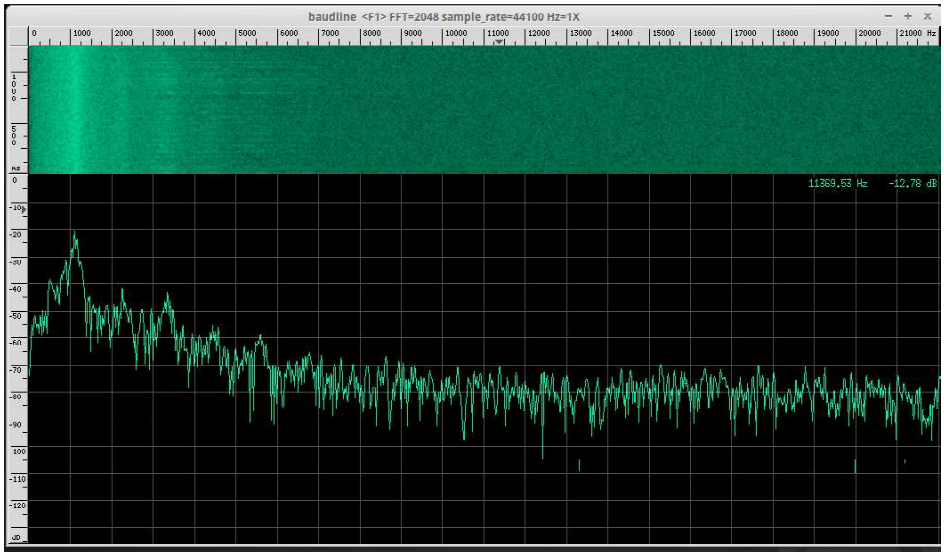



Figure 6. Flying Pig Version 2 7.122MHz transceiver with noise.

And, last but not least is an old TenTec 1340 40 meter transceiver. I consider this rig to have the nicest audio of all. And you can see why. It is low in the frequency bandpass. All below 1KHz, but I'm not doing QRQ work with this rig or any QRP rig for that matter. :-)

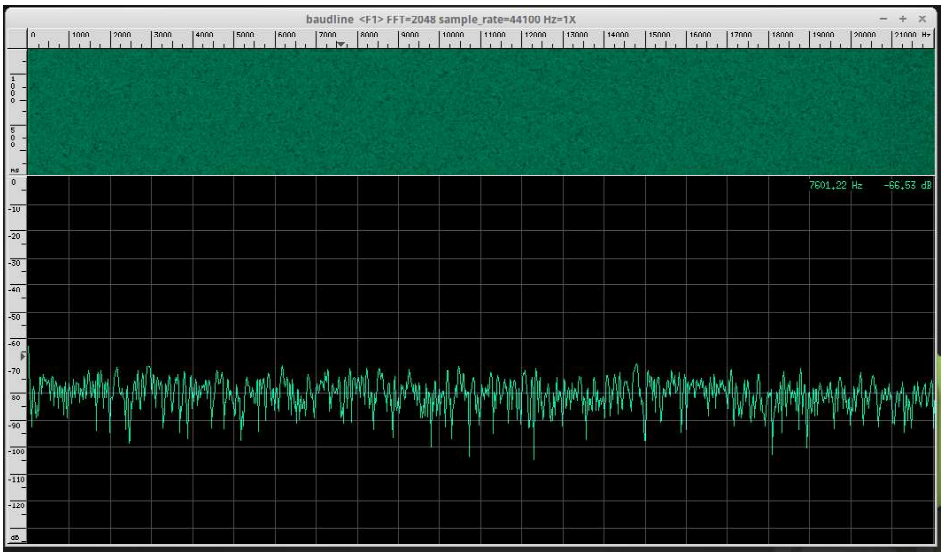



Figure 7. TenTec 1340 Transceiver baseline audio spectrum.

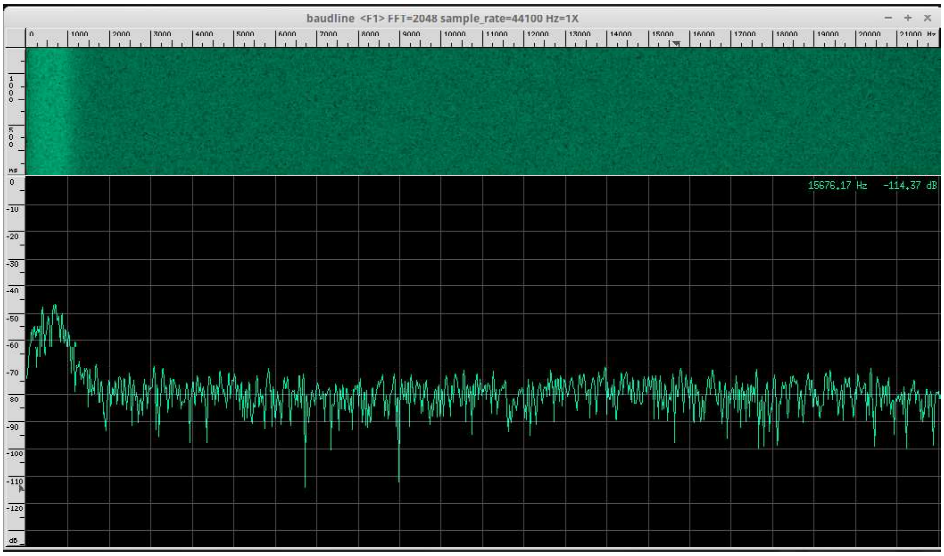



Figure 8. TenTec 1340 Transceiver with noise input.

This months material was a start at what you can do with a noise generator. I did not show tweaking tuning along the receiver chain to peak the signal level. With the visual display of the audio on a computer screen, you can do a much better job than just using your ears, although the trained ear can do pretty darn good.

Next month I will take several S9 generators and show how to use them. May even dig out the old VE3DNL marker generator and demonstrate what I was doing back in Dallas at two o'clock in the morning when I built a 20m SST NorCal kit for Doug. He talks about it often. Good times.

So much to cover and so little time. Enjoy and don't forget your home-work. This is not a spectator sport.

Oh. One last photo. Doug, on qrp-tech, posted about a small drill press that was recommended by Ken, WA4MNT, for use in drilling PCB material. I bought one immediately, just on their recommendation. To show you that I can do PTH boards as well as muppet boards, here are two K7QO noise generators. After the photo is the PCB layout. This one you do not reverse to laminate and etch. Use as is. The spacing is 0.10" on the transistor pads, 0.40" on the resistor pads but it is not critical since you can bend the leads to fit. I used 2N3904s on one board and 2N2222As on the second. The 2N2222As do a little better, but your mileage may vary. I use a high intensity green LED from China with a 4.7K resistor to remind me I have the critter plugged in and turned on. I use 0.10" headers for power and BNC connections to switch between project boards in the lab. Saves on hardware for me. Thanks for your time and atten-tion. Chuck, K7QO, just east of CA.

Figure 9. Two K7QO Noise Generator boards assembled.

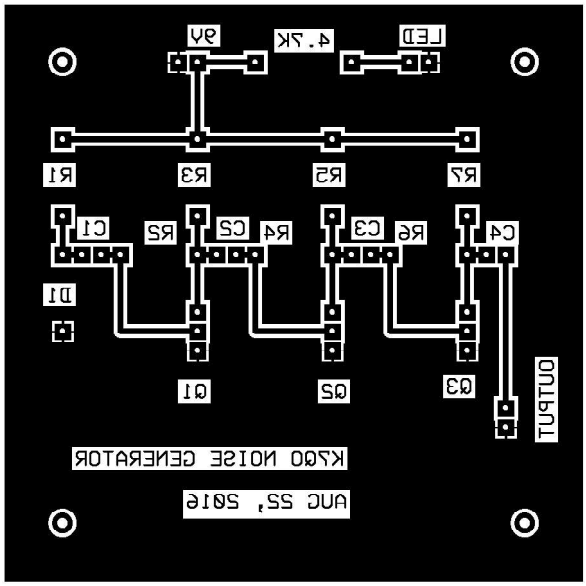



Figure 10. ExpressPCB layout.

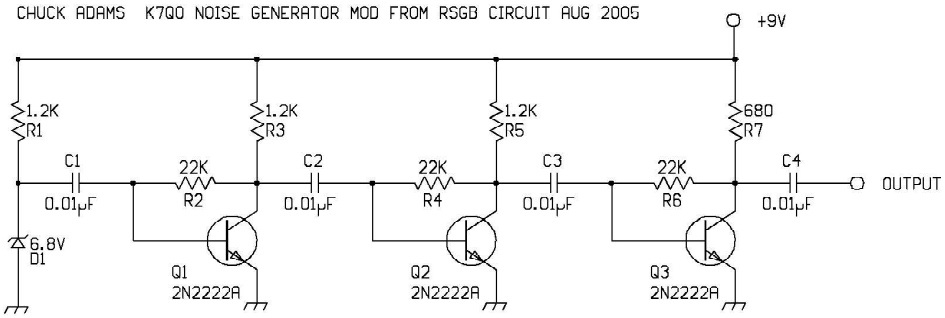



Figure 11. K7QO Noise Generator Schematic