QRPp

A QRP Journal, November 2016
Vol 12, Issue 4

QRPGuys.com Digital Counter Kit

Table of Contents

From the Editor - Doug Hendricks, KI6DS

Another issue is in the bank. Lots of construction articles in this one. The cover picture is the QRP Guys Digital Display kit configured as a lab counter. I built my first one in an Altoids tin, but wasn't happy with it. Ken LoCasale, who owns QRPGuys, is a master craftsman. On one of my many trips to Clarkdale he showed me how easy it is to make cases out of pcb stock. You see the results on the cover. These cases are easy to make. If you go to Ken's website, QRPGUYS.com and click on Misc. Files, you will find an in depth article on how to make cases. Much more detail than in my article.

Speaking of Ken, I owe him a huge public thank you for his donating the dummy load kits for the Pacificon building event. The boards for my design had a flaw in them, and I did not have time to fix them before the event. Ken came to the rescue and donated the kits we built. Thank you Ken. Check out his website for some very affordable kits.

Paciificon has come and gone. It was a great time as always, but QRP attendance was down. We haven't done a very good job in the last 4 or 5 years of promoting QRP at Pacificon, but that is about to change. Many thanks to Chuck Adams, Darrel Swenson, Gary Ekker, Steve Smith, Darrel Jones and I am sure there were others for their help. Next year will be bigger and better. Those of us who were there (about 40 - 50) had a great time. The theme was fun, and we sure had some.

Don't forget if you are in the San Jose area on the First Tuesday of the month, we meet at Denny's, 1140 Hillsdale, at 6:30 for QRP Fun. Kurt Kiesow is the guest speaker next month speaking on the Wave Glide. See you there.
Doug, KI6DS

Arduino for the Terrified, Part 5

by Jack Purdum, Ph.D., W8TEE

In this session, I want to discuss adding a very versatile device called a rotary encoder to your Arduino projects. Rotary encoders are used to sense angular motion. In ham radio use, encoders often use that change in angular motion to change something else. For example, in the Forty-9er transceiver, I use the rotation of a rotary encoder to change the VFO frequency. I also used the same encoder for changing menu options in the transceiver. That's one of the nice features of rotary encoders: They can be used to change almost anything. Unlike some device that use rotary motion to affect change, that change is often limited to one result. For example, potentiometers can only change resistance. Rotary encoders, on the other hand, are only limited by your imagination.

While encoders can have minor variations, for the most part there are two basic types: 1) mechanical, and 2) optical. Both rely on a series of pulses that describe the direction of the rotary motion. Optical encoders often use an encased LED shining through a perforated disk to produce a chain of pulses as the encoder is rotated. (A good discussion can be found at: http://machinedesign.com/sensors/basics-rotary-encoders-overview-and-new-technologies-0.) Mechanical encoders have "detents" which causes a pulse chain to result as the shaft is rotated across these detents. With mechanical encoders you can feel the detents as you turn the shaft.

Rotary Encoders

Some expensive optical encoders have extremely fine granularity, where they can send a pulse chain after sensing a shaft movement of less than 1° of rotation. We are going to use an inexpensive mechanical encoder that has only 20 detents per revolution. A

Figure 1. The KY-040 mechanical rotary encoder

little quick math tells us our resolution is one pulse chain per 18° of rotation. Hey...what do you want for about a buck a piece? Specifically, we are going to use the KY-040 encoder. See Figure 1.

The Good News

There are a number of different flavors of this encoder. I prefer the one that comes on the small board seen in Figure 1 for several reasons. First, the pins are clearly marked on the board. Second, the backside of the board usually holds SMD devices that are pull-up resistors on the pins. This means you get “clean” HIGH and LOW because the pin doesn’t float between those values. Third, the shaft is threaded so it is easily attached to an enclosure. BTW, not all KY-040s have threaded shafts, so make sure the ones you order do have threads on them. Finally, the shaft can be pushed to serve as a switch, which means the encoder can do double-duty.

I buy these 10 at a time because I use them all the time. With a little shopping online, you should be able to buy 10 for less than a dollar each.

The Bad News

Unlike optical encoders, as the contact point rotates over the detents, that mechanical contact will vibrate. This vibration sends out a false pulse chain as the mechanical parts of the encoder settle down to a stable state. The Nano is fast enough to read these pulses and they can mess with the real codes being sent by the encoder. Because of this, we need to “debounce” the encoder to remove these false pulse chains.

Basically, we can remove the vibrations in software or in hardware. The software solution simply injects a short time delay in the program until the vibrations stop. 250 milliseconds is a commonly-used time slice for the delay. This actually is an acceptable solution in our use here because humans can easily cope with a quarter-second delay. However, in a time-critical application, that kind of delay might be unacceptable. In that case, we would opt for a hardware solution.

The hardware solution is to connect a small amount of capacitance across the data lines used by the encoder. I found a bargain online where I bought 200 0.1uF capacitors for $0.80 including shipping! I just solder one of these caps to each data line and then to ground. Figure 2 shows how I prepare the two

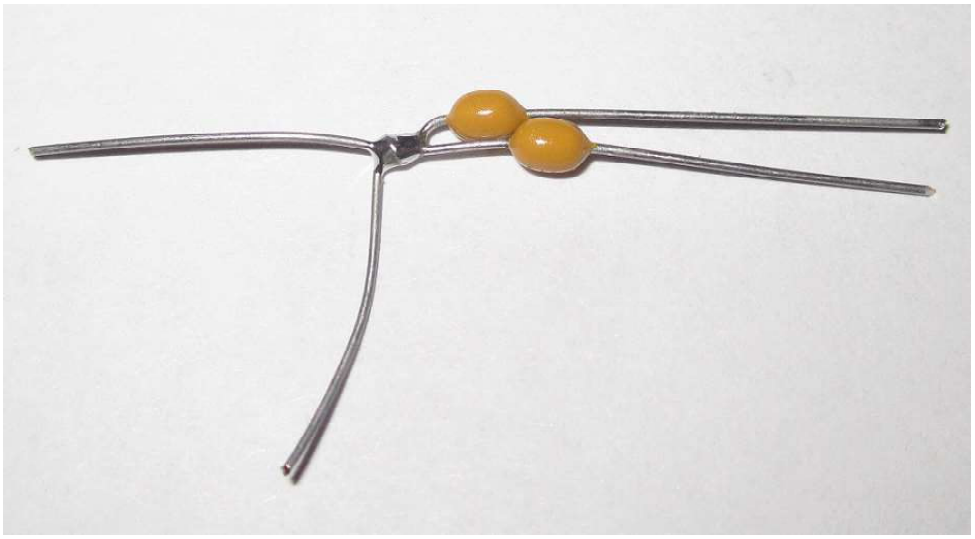


*Figure 2. Soldering two 9.1uF capacitors for debouncing*caps.

Next, I bend the leads in place and solder them to the clock and data lines of the encoder, as shown in Figure 3. As you can see, the clock and data lines are the two right-most pins on the encoder.

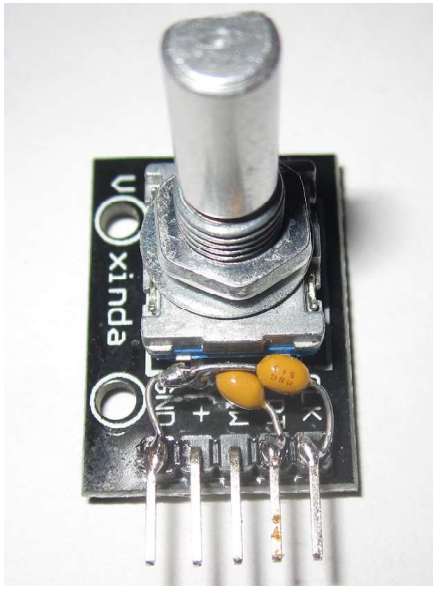


Figure 3. Debounce caps in place

The common lead is soldered to the left-most pin, which is the GND pin. Note how the solder joints are as close to the board as I could get them. This makes it easier to get a good connection using the DuPont jumper wires that will be attached to the pins. The pin just to the right of the GND pin is the 5V pin and the center pin connects to the encoder switch. Simply stated, the two caps suck up the vibrations associated with the mechanical movement of the encoder.

Yeah...I know what some of your are thinking: Those two caps are about the size of a grain of rice, but my eyes are too old and hands too jumpy for this kind of work. Bullcrap! I'm two years younger than dirt and if I can do it, so can you. It may take a little more time than it did 40 years ago, but so what? No excuses, cuz you'll just be backing away from some fun that I know you will enjoy.

Encoder Demo

Okay, we have an encoder with debouncing caps on it...now what? Well, let's use it in a really simple program as a input selection (Step 2) device. In this program, we'll use the LCD display we used earlier, but we really don't need the piezo buzzer, so you can "unplug" that if you want. The encoder is connected with the clock and data pins tied to the Nano's external interrupt pins. We connected the encoder switch pin to pin 4 on the Nano. In *setup()*, we have the statement:

```
pinMode(ROTARYSWITCHPIN, INPUT_PULLUP);
  // Use pullup resistors
```

which activates the pullup resistors built into the Nano. This ensures that the pin doesn't "float". Actually, if your encoder has a small board like the one shown in Figure 3, look an the back side and you will likely see that the switch has 10K SMD resistors already there. Still, our statement won't hurt anything even if they are present.

Encoder Use Choices

There are two basic ways to use an encoder: 1) polling, or 2) interrupts. An example will help explain the difference. Suppose you are writing code to read fire sensors in the Empire State building. Further assume that it takes 1 second to read a sensor to determine its state (e.g., 0 = no fire, 1 = fire). A polling approach means that you have a list of these sensors and you start with sensor 0, visit it, and determine if there's a fire or not. Assuming no fire, you then look to see who's next and proceed to visit sensor 1. You continue doing this until you reach the end of the sensor list. If there's been no fire, you start over again and start polling with sensor 0. Obviously, if you sense a fire, you set of the alarm and execute a "fire" sequence of function calls.

Polling a list of sensors is easy to understand and implement. However, sometimes "easy" doesn't mean good. In our example, suppose each floor in the building has 100 sensors. This means that one polling loop from the ground floor to the top floor will

take 2.8 *hours* to complete! If a fire breaks out right after that sensor was polled, it would be almost three hours before the alarm would go off. Not good.

The alternative is to use interrupts and their associated *interrupt service routine* (ISR). Anyone who has raised a two-year old knows what an ISR is. An interrupt is an event that demands immediate attention. The ISR determines what happens when the interrupt occurs. For example, the fire ISR waits for the sensor to inform it if and when its state changes and "interrupts" it anytime there is a state change. The assumed state is "No fire". Then, instead of visiting each sensor in a linear fashion, each sensor has the ability to immediately inform the controller that its state has changed (i.e., a fire is present). The result is virtually instantaneous notification of a fire at the location of that sensor. Using the more effective ISR approach may cost more because sensors that can send a message to the ISR may be more expensive plus the code to process that message is a little more difficult to write than polling interrupts.

Figure 4 shows my encoder wiring for the demo program. Obviously, we're taking the tricky route, as seen in Listing 1.

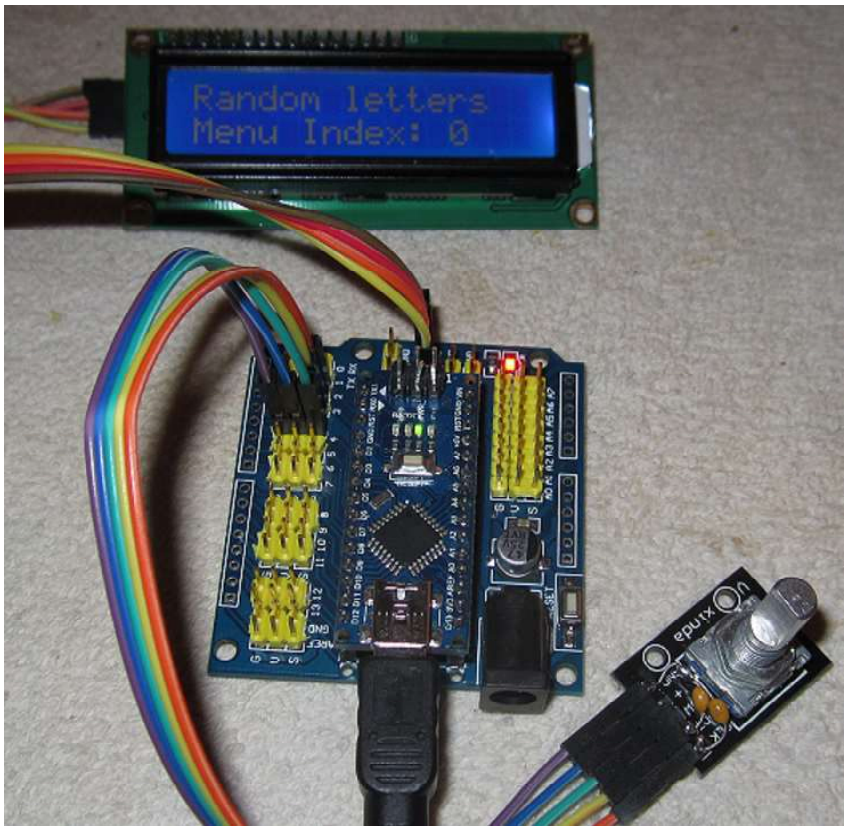


Figure 4. My wiring of the encoder.

A Demo Program

First, note that we are adding another non-standard library, just like we did earlier with the I2C library. This one is for the rotary encoder and uses code written by Brian Low. The URL for the library is in the program source code and the steps to install the library are virtually the same as for the I2C library.

Listing 1. Using an Encoder with an ISR

```
/*
  Rev 1.00: August 21, 2016, Jack Purdum, W8TEE
*/
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
#include <Rotary.h>          // https://github.com/brianlow/Rotary

#define ROTARYCLOCKPIN   2  // Pins 2 and 3 are external interrupt pins on the Nano
#define ROTARYDATAPIN    3
#define ROTARYSWITCHPIN  4  // Used by switch for rotary encoder

#define SCREENWIDTH     16  // Chars on one line of LCD

#define ELEMENTCOUNT(x)  (sizeof(x)/sizeof(x[0])) // Parametized macro for
                                                  // element count

char spaces[SCREENWIDTH + 1];   // Don't forget NULL

// ISR variables
volatile int rotationDirection;    // + is CW, - is CCW

int menuSelection;               // Which menu is active
int menuIndex;                   // Which menu option is active
int elements;                    // Holds menu elements

char *choices[] = {"Random letters", "Call Signs", "Words"};

Rotary r = Rotary(2, 3);       // sets pins of encoder.  Must be interrupt pins.
LiquidCrystal_I2C lcd(0x27, 2, 1, 0, 4, 5, 6, 7, 3, POSITIVE);

//
================================================================================
void setup() {
 lcd.begin(SCREENWIDTH, 2);
```

```
 pinMode(ROTARYSWITCHPIN, INPUT_PULLUP);    // Use pullup resistors

 elements = ELEMENTCOUNT(choices);

 memset(spaces, ' ', SCREENWIDTH);  // Initialize all elements to spaces
 Splash();
 menuIndex = 0;
 ShowCurrentSelection();

 PCICR |= (1 << PCIE2);     // Pin Change Interrupt Enable 2 to set
                    // Pin Change Interrupt Control Register
 PCMSK2 |= (1 << PCINT18) | (1 << PCINT19); // Use pins 2 and 3 for
                                 // the control mask

 sei();                  // Set interrupts flag
 rotationDirection = 0;
}


void loop() {
 static int switchState = 1;

 switchState = digitalRead(ROTARYSWITCHPIN);
 if (switchState == LOW) {                // Switch pressed?
  MyDelay(200);                       // Yep...
  lcd.setCursor(0, 1);
   lcd.print(spaces);
  lcd.setCursor(0, 1);
  lcd.print("Selected: ");
  lcd.print(menuIndex);
 }

 if (rotationDirection != 0) {             // Encoder rotated??
  menuIndex += rotationDirection;          // Yep...

  if (menuIndex < 0)                // Rotated CCW too far, wrap to highest
   menuIndex = elements - 1;

  if (menuIndex == elements)         // Rotated CW too far, wrap to lowest
   menuIndex = 0;

  ShowCurrentSelection();
  rotationDirection = 0;
 }
```

```
}

/*****
 Purpose: to cause a delay in program execution

 Paramter list:
  unsigned long millisWait    // the number of millseconds to wait

 Return value:
  void
*****/
void MyDelay(unsigned long millisWait)
{
 unsigned long now = millis();

 while (millis() - now < millisWait)
   ;     // Twiddle thumbs...
}


/*****
 * This routine displays a simple menu selection sequence on the Serial monitor
 *
 * Parameter List:
 *    void
 *
 * Return value:
 *    void
 *****/
void Splash()
{
 lcd.print(" Rotary Encoder");
 lcd.setCursor(0, 1);                 // Second line
 lcd.print(" Jack, W8TEE");
 MyDelay(2000);
}

/*****
 * This routine displays a menu item from the current menu selection
 *
 * Parameter List:
 *    void
 *
 * Return value:
```

```
 *    void
 *
 * CAUTION: This function assume menuIndex and menuSelection are set
 *         prior to the call.
 *****/
void ShowCurrentSelection()
{
 Erase();
 lcd.setCursor(0, 0);
 lcd.print(choices[menuIndex]);
 lcd.setCursor(0, 1);
 lcd.print("Menu Index: ");
 lcd.print(menuIndex);
}

/*****
 * This erases the LCD screen
 *
 * Parameter List:
 *    void
 *
 * Return value:
 *    void
 *
 * CAUTION: Assumes a spaces[] array is defined
 *****/
void Erase()
{
 lcd.setCursor(0, 0);
  lcd.print(spaces);
 lcd.setCursor(0, 1);
  lcd.print(spaces);

}

/*****
 * This is the Interrupt Service Routine for the rotary encoder
 *
 * Parameter List:
 *    PCINT2_vec      Pin change interrupt request #2, pins D0 to D7
 *
 * Return value:
 *    void
 *****/
```

```
ISR(PCINT2_vect) {
  unsigned char result = r.process();

  switch (result) {
    case 0:                    // Nothing done...
      return;

    case DIR_CW:               // Turning Clockwise
      rotationDirection = 1;
      break;

    case DIR_CCW:              // Turning Counter-Clockwise
      rotationDirection = -1;
      break;

    default:                   // Should never be here
      break;
  }
}
```

Near the top of the program is the following line:

```
#define ELEMENTCOUNT(x) (sizeof(x)/sizeof(x[0]))  // Parametized macro for element
                                                  // count
```

This is what's called a parameterized macro because it is a *#define*, but with a variable name as part of it. The following statement appears in *setup()*:

```
elements = ELEMENTCOUNT(choices);
```

In the macro definition, the 'x' is replaced with the name of an array; *choices[]* in this example. The *choices[]* array is defined as an array of pointers to *char*. Each on of those 3 pointers points to a memory location where the words "Random letters", "Call Signs", and "Words" are stored. Because each pointer takes 2 bytes on an Arduino and there are 3 pointers, the expression *sizeof(x)* returns the number 6, because that the number of bytes in the *choices[]* array. However, we then divide that value by the expression *sizeof(x[0])*. This is asking how many bytes does it take to store one element of the array. We already know each pointer takes 2 bytes, so the expression becomes:

```
elements = ELEMENTCOUNT(choices);
elements = (sizeof(choices)/sizeof(choices[0]));
elements = (6/2);
elements = 3;
```

Big deal, you say. Well, actually it is a big deal for several reasons. First, it allows us to make changes to the *choices[]* array without having to change the code to determine the number of elements in the array *elements*. *elements* would automatically get changed any time you change *choices[]*. Secondly, it works with any type of data. For example, suppose you defined a floating point array as:

```
float temperatures[15];
// ...a bunch of code...
for (k = 0; k < 15; k++) {
  temperatures[k] = ReadSensor(k);
}
```

But, with our macro, you could write the *for* loop as:

```
for (k = 0; k < ELEMENTCOUNT(temperatures); k++) {
  temperatures[k] = ReadSensor(k);
}
```

and, because each *float* variable uses 4 bytes of memory, the math works out to:

```
for (k = 0; k < (sizeof(temperatures)/sizeof(temperatures[0])); k++) {
for (k = 0; k < (60/4); k++) {
for (k = 0; k < 15; k++) {
```

Note how the use of the parameterized macro gets rid of the magic number "15" and actually makes the code easier to understand. Also, if you add 20 more sensors, the *for* loop is automatically adjusted for the new size once you redefine the *temperatures[]* array. Note that the fact that a *float* takes four times as much memory as a *char* does not affect the use of the macro. You can use the macro with any array.

The rest of the code in *setup()* should look pretty familiar, with the exception of:

```
PCICR |= (1 << PCIE2);     // Pin Change Interrupt Enable 2 to set
                           // Pin Change Interrupt Control Register
PCMSK2 |= (1 << PCINT18) | (1 << PCINT19); // Use pins 2 and 3 for
                                           // the control mask

sei();                     // Set interrupts flag
```

These statement are used to establish the ISR for the encoder that we have attached to pins 2 and 3. The first statement sets a flag in the Interrupt Control Register to enable external interrupt 2. If you refer back to Figure 2 of Part 3 of this series, the Nano logical (purple numbered) pins 2 and 3 signify external interrupts 0 (INT0) and interrupt 1 (INT1). (Those two fields are shown with a salmon pink background in the figure.) However, you will also see that those same pins are used for the interrupt control mask (PCINT18 and PCINT19 with white backgrounds), too. How can the same pins be used for more than one task. The reason is because each *bit* associated with each pin has its own meaning, not just the entire byte. The bit masks are logic OR'ed to set the control and mask registers for the interrupt. The *sei()* function call sets the interrupts enabled flag.

I realize this is all Greek to you and I don't want to get bogged down in the details. If you're interested in the details about interrupts, a really good explanation can be found at: http://gammon.com.au/interrupts.

Nick Gammon is a regular contributor to the Arduino Forums and probably has forgotten more about the Arduino than I'll ever know.

Okay...back to the code.

My guess is that all of the program code is easy enough that you can figure it out, but the ISR deserves some comment. I've repeated it here so you don't have to flip back and forth.

```
ISR(PCINT2_vect) {
  unsigned char result = r.process();

  switch (result) {
    case 0:                    // Nothing done...
      return;

    case DIR_CW:               // Turning Clockwise
      rotationDirection = 1;
      break;

    case DIR_CCW:              // Turning Counter-Clockwise
      rotationDirection = -1;
      break;

    default:                   // Should never be here
      break;
  }
}
```

The ISR Function

Because we have compiled our program to include the function signature ISR(PCINT2_vect), any interrupt that can be recognized by our interrupt mask gets immediately routed to the *ISR()* code. The encoder library allows us to define an encoder object, which we have named *r*. (It's defined around line 26 in the program.) That encoder object has a function named *process()* buried within it that we can call to see what has happened. If nothing happened (i.e., a false pulse), the return value of 0 is assigned into *result*. The *switch* block then sends program control back and waits for something to happen. However, if the coder was turn clockwise (DIR_CW) or counter-clockwise (DIR_CCW), we set the *rotationDirection* variable to either 1 (CW) or -1 (CCW). Program control then immediately returns to *loop()*. Because *rotationDirection* is not 0 now, we set the *menuIndex* accordingly and then call ShowCurrentSelection() to display the selected menu item. Once that string is

displayed, we return to *loop()* and set *rotationDirection* back to 0 before the next iteration of *loop()*.

Important ISR Details

A couple of things about ISR's. First, they should be as short as possible. The reason is because, while the Nano is processing your ISR code, nothing else can happen. If you waste a lot of time in your ISR, a fire could break out and, because the Nano is wading through your too-long code, the fire has the opportunity to cause some significant problems. By keeping the ISR as short as possible, you minimize the likelihood of that kind of issue.

Second, look how I defined *rotationDirection* near the top of the file:

```
volatile int rotationDirection;    // + is CW, - is CCW
```

The keyword *volatile* should precede all definitions of variables defined outside of the ISR, but used by the ISR. That keyword forces the compiler to reload that variable from memory any time it is used. The reason is because the compiler is smart enough to leave variables that are used a lot in a program in a CPU register; a process called *caching*. It is possible in an ISR for a cached variable to have a value that is "out-of-date" and may affect the ISR. While that's not likely an issue here, I always use *volatile* for any variable used by the ISR, but defined outside of the ISR. It's a good habit to get into.

Finally, as I mentioned, while the Nano is servicing your code, nothing else can happen, even if it is another interrupt. The Nano must finish your code first. However, since *Serial.print()* and *Serial.println()* also use interrupts, they will not work correctly in an ISR. They may *seem* to work, but they won't. If you need to display some variable's value, wait until control returns from the ISR before you try to use the *Serial* object.

Conclusion

So, what does the program do? Not much. It allows you to advance through a short menu list and then select a menu item by pushing the encoder switch. The code, therefore does show you how to use a rotary encoder to scroll through a list using interrupts. It also shows you how to detect a rotary encoder switch press by looking at the encoder's switch pin. Spend some time studying the code, as encoders are really useful input (Step2) devices. As an exercise, try adding a submenu and use the encoder to select a "main" menu selection and, based on that selection, it presents a "submenu". That will be a great introduction to the next chapter's Morse Code Tutor.

Arduino for the Terrified - Part 6

by Jack Purdum, Ph.D., W8TEE

In this part I want to extend our use of the rotary encoder and have it function both as a menu selection tool and a data input tool, too. What we're going to build is a Morse Code Tutor. Yeah, I know...you already know Morse so what's the point? Actually, there's a lot of "points". First, even if you don't build this unit, you're going to learn something about Arduino interrupts, Interrupt Service Routines (ISR's), and how to use the rotary encoder for more than just a menu option selection device. Second, I honestly believe a lot of my fellow club members are missing out on a lot of enjoyment that knowing CW brings to the table. So, perhaps you can build this for your club and lend it out to members who show an interest in learning Morse code. Third, you might pick up a few good coding habits along the way as this code is fairly long and gives us a chance to see some code that is more than just "learning-type" programming examples. Also, we're going to use a simplified method for using interrupts rather than the port masks we used in Part 5.

The feature set of this Tutor includes:

User-setable code speed
User-setable sidetone frequency
Practice sets using words
Practice set using letters
Practice sets using numbers
Practice set using words and numbers together
Practice Field Day exchanges
Practice CW Sweepstakes exchanges
Practice QSO's with randomly generated segments (that make sense), including random call signs
User-setable encoding using Standard or quasi-Farnsworth encoding
On-off LCD display option—watch letters as they are sent, or turn the display off
Portable or "wall wart" power

In the list above, "user-setable" means that you can change something as the program runs. This is a lot more flexible than forcing choices before the program is compiled and becomes etched in stone.

Figure 1 shows the development state of the Tutor. The display is the same one you've been using all along, as is the encoder which you used in Part 5. The speaker is new. This is a super cheap 8 ohm speaker the I got online for less than $2.00. That red shrink tubing you see going to the speaker is hiding a 1000 ohm resistor. Because of the limited current that can be taken from an Arduino pin, it a good idea to put a current limiting resistor in the line running from the Arduino to the speaker.

Whoa! What's that small blue thingy?

Actually, that is a new version of the Arduino Mega2560 called the Mega2560 Pro Mini. So, what's with the horses-middle-stream thing all about. Well, remember how I said you'd probably run out of SRAM before you run out of program space back in Part 1? Well, that's exactly what happened. Even with the fairly robust feature set presented above, only about 20K or the Arduino Uno or Nano's flash memory is used. However, I am maintaining a lot of data, including all of the state abbreviations, the ARRL section abbreviations, often-used words from a CW QSO, plus a ton of function calls. There's a design flaw in the Arduino architecture that duplicates string constants from flash memory to SRAM at

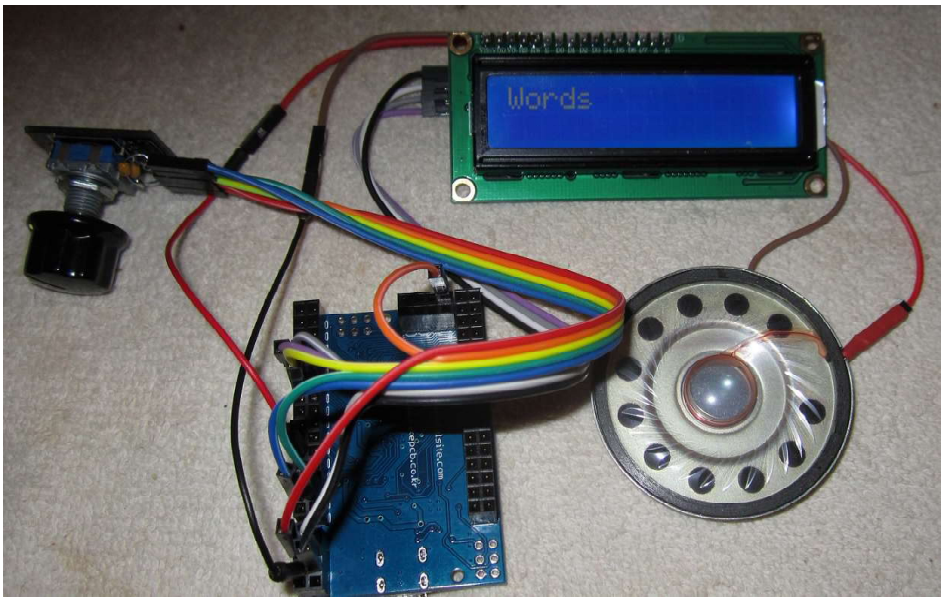


Figure 1. Early version of Tutor

runtime. While I did use a number of memory-saving tricks to conserve SRAM, they still weren't enough to allow me to keep the feature set I wanted. The result is I had to go to the Mega2560. If you'll recall Table 1 from Part 1 of this series, the Mega2560 has 256K or flash, 8K of SRAM, and 4K of EEPROM. It's the additional SRAM that I really needed. The Pro Mini shown in Figure 1 has exactly the same memory resources as its big brother. Alas, as so often happens, the second born is a little spoiled and costs more.

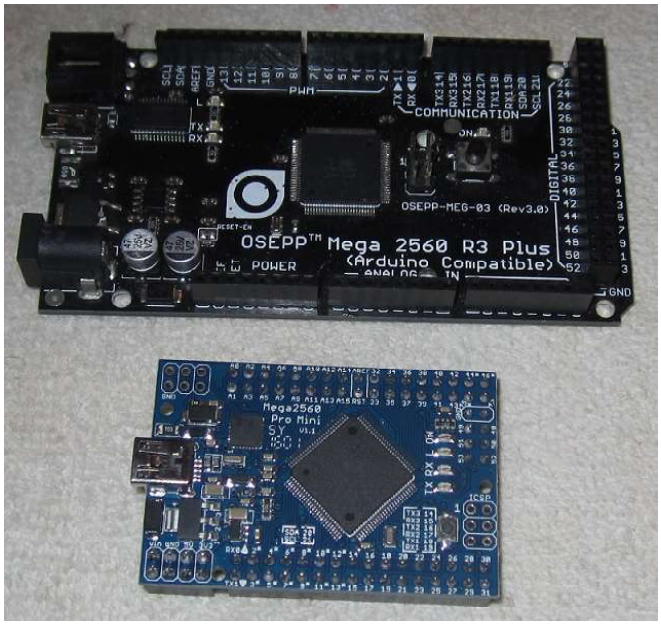


Figure 2. The standard Mega2560 and the Pro Mini

The standard Mega2560 can be found pretty easily for around $10. The Pro Mini is going to cost around $16. Figure 2 shows the two controllers so you can get idea of the relative sizes. So, why spend 60% more for the Pro Mini if they have the same resource base?

Well, size matters.

In this case, the smaller board allows for a smaller enclosure, which means it's easier to sit in the airport and practice Morse code than if you're using the larger board. (No reason why you can't add an earphone jack, right?) Also, it leaves more room in any given case for a larger battery pack. With a few exceptions (e.g., the Arduino Due), most Arduinos can run with a voltage source from 5V to 17V (although the regulator gets a little cranky at higher voltages).

Finally, I suppose you could strip out program features until it runs on an Arduino Uno or Nano. However, that's a crap shoot because SRAM is used extensively during function calls. (If you want some of the ugly details, Google "stack usage during function call" and you'll get about 28 *million* hits!) The problem is that the statistics you see at the bottom of the IDE page are compile-time statistics, not runtime. The compile-time stats show how much SRAM is being detected when your program is being compiled. However, as we mentioned before, the runtime use of SRAM actually ebbs and flows with each function call and data fetch. I had around 65% flash usage and 77% SRAM usage when the Nano version program started to become flaky. My

code ebbed and flowed me right out of the Nano and into the Pro Mini.

In any event, the choice is yours: 1) keep the Nano and strip the fat out of the program until the Nano's ribs start showing, 2) buy a regular Mega, or 3) buy a Pro Mini. Personally, working a few extra days as a Walmart Greeter allowed me to buy the Pro Mini and, to me at least, is well worth it. If you can live with a larger enclosure, the regular Mega is fine. As to the Nano, perhaps there are features you don't need or want, so maybe you can get by without either Mega. Your call.

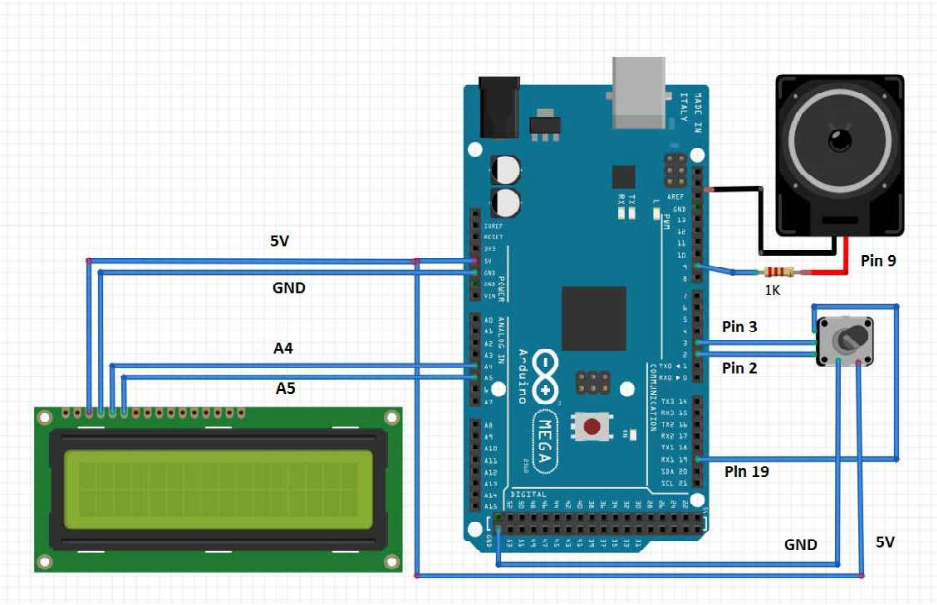


Figure 3. Fritzing diagram of Tutor.

Figure 3 shows a Fritzing diagram of the Tutor, which may help you see how things are connected. Much of the detail is actually also stated in the program code.

The Code

The code for the program is pretty long, so rather than show all of it here, you probably should download the source code file and follow this discussion with the source code displayed in the IDE. The program begins with a series of *#include* preprocessor directives to grab the necessary contents of various library files. The only new library file is the Tone library and the URL for that library is given in the code.

After the *#include* directives, you'll see the following (the source code has a little longer comments; I shortened them here so they each fit on one line):

```
                        // EEPROM offsets
#define WPMOFFSET         10                // WPM offset: 1 byte
#define FARNSWORTHOFFSET  (WPMOFFSET + 1)       // Store Farnsworth speed
#define DISPLAYOFFSET     (FARNSWORTHOFFSET + 1)  // Is display ON or OFF: 1
byte
#define TONEOFFSET        (DISPLAYOFFSET + 1)    // Sidetone freq: 2 byte as int
#define ENCODINGOFFSET    (TONEOFFSET + 2)      // 0 = normal, 1 = Farnsworth
#define CALLOFFSET        (ENCODINGOFFSET + 1)   // User call: up to 10 bytes

#define WEATHEROFFSET     16     // Where WX words start in word list: 1 byte
```

These offsets are used to fetch various data that is stored in EEPROM. Note how each offset is framed in terms of the offset "in front of" it. For example, the *ECODINGOFFSET* is the only entry that has a '2' in it. The reason is because the expected value for tone (e.g., 700Hz) is too large to fit in a single byte (i.e., numeric range of 0-255). This scheme keeps all of the constants together, starting at EEPROM address 10 for the *WPMOFFSET*. If you can receive at more than 255 wpm, you'll need to make the *FARNSWORTH* offset a "+ 2" instead of a "+ 1". (Why?)

A little further down in the code and you'll find this definition (I made the font smaller for a reason):

```
char *words[] = {"DE", "TNX FER", "BT", "WX", "HR", "TEMP", "ES", "RIG", "ANT", "DIPOLE",
                 "VERTICAL", "BEAM", "HW", "CPI", "NAME", "QTH", "WARM", "SUNNY", "COLD", "CLOUDY",
                "RAIN", "SNOW", "FOG", "CALL", "RPT", "UR", "RST", "AGN"
        };
```

This is the definition for the word list that is used to generate a mock QSO. You can easliy place the entire list on a single line, so why didn't I? Well, if you always write perfect code, you don't need to format it this way. However, since elements in this list are used based on a number produced by a random number generator, I may have to do some debugging on that number. The *word[]* array is arranged in rows of 10 elements each. (Recall that arrays in C always start with element 0, so the first 10 elements actually have index values 0-9.) Now when I have to align the value from the random number generator with an element in the *word[]* list, I don't need to count every element in the array to figure out where I am. I've done the same thing with other array lists in the file (e.g., state abbreviations). Like I said, if you always write perfect code, you'll never need this bit of defensive coding. The rest of us mortals may find it useful.

While we're talking about the random number generator, consider these lines that are in the source code:

```
rand = random(WEATHEROFFSET, WEATHEROFFSET + 7);
 strcat(QSO, words[rand]);
 strcat(QSO, " ");
```

If you look at the list of *#define*'s near the top of the source code, you'll see that *WEATHEROFFSET* is 16. Now, count to element 16 of the *word[]* list array and what do you find? Because we've formatted the list the way we did, we can quickly determine that the word at *word[16]* is "WARM". What we are telling the random generator function, *random()*, is only generate numbers with the value 16 to 22. If you look in the *word[]* array, *word[22]* is "FOG". Therefore, *random()* has been told to only produce numbers between 16 and 22 because all of those are "weather" words.

Wait a minute! *WEATHEROFFSET + 7* is 23, not 22. Very astute catch! However, the second argument to *random()* is the cutoff and is *not* included in the values produced. In other words, the range of values is exclusive of the second value used in the function call to *random()*. The first argument is inclusive in the list. Therefore, the "+ 7" tells you that there are 7 weather words in the list.

Obviously, if you want to add more words to the list, you should add them at the end of the *word[]* array. If you want more "weather words", they should be added after "FOG" and the offset extension, '7', would be increased accordingly. A more flexible way might be to use:

```
   rand = random(WEATHEROFFSET, WEATHEROFFSET +
WEATHERWORDCOUNT);
```

How would you implement this so it worked? Think about it...

One more detail: Near the bottom of *setup()* you'll find the following statement:

```
 randomSeed(analogRead(0));
```

The *randomSeed()* function is used to initialize (again, a Step 1 process) the random number generator so that it generates a different series of numbers each time power is reapplied. It does this by reading "noise" on the analog pin 0. If you need to generate a repeatable sequence of random numbers, comment this line out. Sometimes doing so is helpful when debugging code.

In *setup()*

Step 1 in program development is used to initialize things in a way that establishes the environment in which the code is to run. (I placed *setup()* and *loop()* at the bottom of the source file so it's a little easier to find them. With almost 1700 lines of code, this helps locate them more quickly.) In *setup()* you find this code snippet:

```
 c = EEPROM.read(ENCODINGOFFSET);        // Should be 0 or 1

#ifdef DEBUG
```

```
  Serial.print("Tone: ");
  Serial.println(toneFrequency);
  Serial.print("WPM: ");
  Serial.println(WPM);
  Serial.print("display state: ");
  Serial.println(displayState);
  Serial.print("Encoding Type: ");
  Serial.println(encodingType);
  Serial.print("Farnsworth: ");
  Serial.println(farnsworthSpeed);
#endif

   if (c > 1) {                        // If new chip, set defaults
#ifdef DEBUG
  Serial.print("In setup(), writing EEPROM, c =");
  Serial.println(c);
#endif
```

First, what is this *#ifdef* stuff all about? Well, as you might expect, this is a preprocessor directive that says: If my programmer has defined a symbolic constant named *DEBUG*, then compile this program with all of the statements up to the *#endif* directive. Near the top of the source file, just after the *#include*'s you'll see the line (number 6):

```
//#define DEBUG                    // Uncomment when debugging
```

Because the line is presently commented out, the long list of *Serial.print()* function calls between the two preprocessor directives are not compiled into the program. This results in a program that uses less flash and SRAM memory. However, if something goes south and I uncomment line 6, *DEBUG* is now defined and causes all of those print statements to be compiled back into the program after the next compile. This is a simple way to toggling debugging statements into and out of the program without having to erase/retype them each time. This type of defensive coding is often called "scaffolding code" because it helps support the development of the program.

Ignoring the scaffolding code for a moment, commenting out DEBUG leaves us with:

```
  c = EEPROM.read(ENCODINGOFFSET);        // Should be 0 or 1

  if (c > 1) {
```

If you look at these two lines and then look at the comment, the check on *c* to see if it is greater than 1 makes no sense. There are two types of encoding used in the program (more on those later) and they are defined as either 0 or 1. So why check for a value greater than 1?

The reason is because, at least at the present time, EEPROM that has never been written to is initialized at the factory to the value 255 (i.e., 0xFF in hexadecimal, which means all bits are turned on in that byte). Therefore, the first time you run this program, EEPROM is virgin territory. By checking if *c* is greater than 1, either the processor is brand new or it has a bogus value stored in EEPROM. (The *read()* method returns a *byte* data type, which is *unsigned*. This means *c* cannot have a negative value.) In either event, the remaining statements in the *if* statement block writes default values for various variables that are stored in the EEPROM. That way, the next time the user turns the tutor on, it comes up with the proper values in EEPROM.

Setting Up the Interrupts

If you look in *setup()*, you'll find the following statements:

```
attachInterrupt(digitalPinToInterrupt(ROTARYCLOCKPIN), DoClockPin, RISING);


attachInterrupt(digitalPinToInterrupt(ROTARYSWITCHPIN), SwitchInterrupt,
CHANGE);
```

The symbolic constants for *ROTARYCLOCKPIN, ROTARYCLOCKPIN*, and *ROTARYSWITCHPIN* are *#define*'d at the top of the program as pins 2, 3, and 19. The pins shown in Table 1 can be used for external interrupts. (Also see Nick Gammon's discussion on interrupts for more details at, http://gammon.com.au/interrupts).

Table 1. External Interrupt Pins for Arduino Boards

Arduino Board	**External Interrupt Pins**
Uno, Nano, Mini, other 328-based	2, 3
Mega, Mega2560, MegaADK, Pro Mini	2, 3, 18, 19, 20, 21
Micro, Leonardo, other 32u4-based	0, 1, 2, 3, 7
Zero	all digital pins, except 4
MKR1000 Rev.1	0, 1, 4, 5, 6, 7, 8, 9, A1, A2
Due	all digital pins
101	all digital pins

Because of the way that we have defined the symbolic constants, we are using interrupt pins 2, 3, and 19. Let's look at the first *attachInterrupt()* call:

```
(digitalPinToInterrupt(ROTARYCLOCKPIN), DoClockPin, RISING);
```

The *attachInterrupt()* is simply a P"HÊ—wrapper that hides all of the bit masking stuff we did in Part 5. What this call says is: We wish to implement an interrupt service routine (ISR) named *DoClockPin()* on pin 2 (i.e., *ROTARYCLOCKPIN*) that is called

any time the signal on that pin is rising from 0V to 5V. Looking at the code for the ISR, we find:

```
/*****
 Purpose: To determine the direction of rotation of the encoder, -1 = CCW, 1 = CW

 Parameter list:
   int offset

 Return value:
   void

 CAUTION: Assumes rotationDirection is global
*****/
void DoClockPin()
{
 pin3 ? rotationDirection = -1 :  rotationDirection = 1;
}
```

Notice that the *attachInterrupt()* call for the encoder data pin is similar, but that interrupt fires any time there is a change on its pin, regardless of direction. The statement in the *ROTARYCLOCKPIN* ISR uses the *ternary operator* (? :) which is a shorthand version of:

```
if (pin3 == true) {
  rotationDirection = -1;
} else {
  rotationDirection = 1;
}
```

Whenever there is a change on the data pin, the value of *pin3* is changed. So, if there is a rising change on the data pin, its ISR is fired. However, rotating the encoder sends pulses to both interrupt pins. This causes changes in the data pin to ripple into the encoder's clock pin. (Rember from Part 5 that we keep the code in the ISR's as short as possible and we ***never*** put *Serial.print()* calls in an ISR.) If the encoder shaft is being rotated CW, we set the *rotationDirection* value to 1.

Back in *loop()*, we have the following snippet:

```
 if (rotationDirection != 0) {              // Encoder rotated??
#ifdef DEBUG
Serial.print("In loop, rotationDirection = ");
Serial.print(rotationDirection);
#endif
   menuIndex += rotationDirection;          // Yep...
```

```
#ifdef DEBUG
Serial.print("   menuIndex = ");
Serial.println(menuIndex);
#endif
   if (menuIndex < 0)              // Rotated CCW too far, wrap to highest
    menuIndex = elements - 1;

   if (menuIndex == elements)       // Rotated CW too far, wrap to lowest
    menuIndex = 0;

  ShowCurrentSelection(mainMenu, menuIndex);
  rotationDirection = 0;
 }
```

The highlighted line says that, if the user is turning the encoder CW, *rotationDirection* is 1, so the *menuIndex* is increased by 1. The two subsequent *if* statement blocks make sure that the menu index is not advanced or decreased too much so that it becomes out of range of a valid menu item. If the value becomes too large or small, *menuIndex* is "wrapped around" to the start or end of the menu list, as appropriate. The function *ShowCurrentSelection(mainMenu, menuIdex)* displays the proper menu item. We then set *rotationDirection* to 0 in preparation for the next encoder movement.

Double-duty Encoding

Suppose you want to change the code speed the tutor is using. You rotate the encoder until you see "Set Speed" displayed on the screen. You then press the encoder shaft, which activates the encoder switch ISR, *SwitchInterrupt()*. If you follow the code flow, you'll find that this causes the function *SetWPM()* to be called. That code is repeated here:

```
/*****
  Purpose: allow the user to change the wpm speed

 Parameter list:
   void

 Retrun value:
   void
*****/
void SetWPM()
{
 Erase();
```

```
 lcd.setCursor(0, 0);
 lcd.print("Speed: ");
 lcd.print(WPM);

 while (true) {
  switchState = digitalRead(ROTARYSWITCHPIN);
  if (switchState == LOW) {
   MyDelay(ROTARYDELAY);
    WriteWPMToEEPROM();                  // Update EEPROM
    SetDitLength();
     return;
   }

   if (rotationDirection != 0) {        // Encoder rotated??
    MyDelay(ROTARYDELAY);
     WPM += rotationDirection;           // Yep...

    if (WPM < MINSPEED)                  // Rotated CCW too far, wrap to highest
     WPM = MINSPEED;;

    if (WPM > MAXSPEED)                  // Rotated CW too far, wrap to lowest
     WPM = MAXSPEED;

    lcd.setCursor(7, 0);
    lcd.print(WPM);
    rotationDirection = 0;
   }
  }
}
```

The function starts out by displaying "Speed: " on line 1 of the LCD display, followed by the current words per minute (*WPM*) code speed. Note the highlighted line. As the user rotates the shaft, the value *WPM* is changed. Recall that

```
    WPM += rotationDirection;           // Yep...
```

is the same as

```
    WPM = WPM + rotationDirection;      // Yep...
```

If the user is rotating the shaft CCW, *rotationDirection* is assigned the value -1. Therefore, if initial WPM value is 20, after the statement above it becomes 19 with a CCW movement of the encoder. You should be able to convince yourself that rotating the shaft CW increases the code speed.

When the user has the new speed set to the correct value, they once again press the encoder shaft, which sets *switchState* to *LOW* and we call the necessary functions to

write the new speed to EEPROM and change the current dit length to its new value. As you can see here, we can use the same encoder to make menu selections as well as change input values.

Standard versus Farnsworth Encoding

I got my license in 1954. Back then you had to know Morse code to get your Novice, General, Advanced, or Extra license. So at a pretty young age, my friend Charlie McEwen and I spent hours memorizing Morse code for the ASCII characters that could appear on the exam. After we had learned the "dits and dahs", we started sending code to each other. We did the same thing when we upgraded to General class licenses about a year later. (Charlie and I played football together in high school and used Morse code to call cross blocks off and on at the line. Whodathought...) After all those years, the thing I have learned since then is...

...we learned Morse code the wrong way.

We learned code by counting dits and dahs in our head and then writing the letter down on a piece of paper. The approach just about dooms you to a max speed of about 15wpm. Here's why.

The perfect "fist" had a dit length defined by the equation:

ditLength = (1200 / wpm);

The strict definition of "words per minute" is a bit fuzzy since words can vary in length. However, over the years the "timing" word used is PARIS. If you do the math, the equation above determines how long a dit must be for a given speed. Once the *ditLength* is determined, the element spacing within a character is one dit length. Letter spacing is three dit lengths and word spacing is seven dit lengths. For me when I was learning code, those element spacings were etched in stone. This is what I call the Standard method of encoding.

The bad news is that, mentally, I couldn't count dits and dahs and write them down at a pace that exceeded 15wpm using Standard encoding.

Today, we know that approach is not the best way to learn Morse code. A better way is to decide what speed you would ultimately like to be able to send and receive. For me, I want to get to 30 wpm, which is about double my current speed. Let's say you have the same goal but don't know code at all. The new method, however, is to send the Morse at the target speed, ***but*** put a very large space between letters and words. The idea is not to count dits and dahs, but learn to hear the *rhythm* of letters and words.

This new technique for learning code throws the Standard encoding method out the window. The newer method uses what is called Farnsworth encoding. Once again, there is an equation for using a strict interpretation of Farnsworth encoding.

I'm throwing the Farnsworth spacing equations out the window.

Instead, the program allows you to select either Standard or Farnsworth encoding. If you select Standard, you get the 1-3-7 element spacings mentioned earlier. If you select Farnsworth encoding, *you* set what I call the Farnsworth speed. An example will help explain.

If your goal is 30 wpm, use the Set Speed menu option and set it to 30. Now scroll the menu to Set Encoding. It will display the current encoding that's being used (e.g., Standard). If you rotate the encoder, the display changes to "Farnsworth". Since you are just getting started, you probably want an element spacing that is less that 30 wpm. So, dial in whatever speed you think might work. You can go as low as 1 wpm, but you will likely fall asleep at that pace. Try 5 wpm to start with. For someone just starting out, that's a manageable pace. Push the encoder shaft to store that Farnsworth value in EEPROM.

Now go to Send Letters or Send Numbers and start practicing. They are sent in groups of five and horizontal scrolling is active. At first, it may just sound like a bunch of angry bees coming at you, but slowly it will start to makes sense. When you get comfortable receiving Morse letters and numbers, try Send Words. Then move to QSO, which is a mix of everything and elements are randomly changed as you practice. When you start feeling smug, reset the Farnsworth speed to a higher value and start over. Repeat until the Farnsworth speed matches your goal speed. (Then up your goal speed??)

Conclusion

This program is considerably longer than any of the other programs, but it really is no more complex. A little time looking over the code and I'll bet you'll find it all makes sense. Hopefully, buy this time you'll have learned that μc's can be a valuable resource in your home brewing arsenal. If you do start creating some of your own μc-based projects, I hope you'll share them with us.

An End Fed Halfwave Antenna for Portable Ops

by Dave Richards, AA7EE

I'm a very casual operator, and an even more casual portable operator. My main reason for not putting much effort into portable operation is that when I go out into nature, I want to enjoy my surroundings and not be distracted by radios. It sounds like an excuse, but it's true. I spend quite a lot of time hunched over the bench and over my radios at home so when I go out, I don't want to do the same. I'm more the kind of guy who builds small rigs, then operates them from the comfort of my own home. However, I had to take the SST out at least once simply to prove that I can!

The antenna needed to be compact and lightweight, as did the method of matching it. I just didn't feel like carrying lots of boxes and interconnecting cables up the hill, and having to fiddle with them all once up there. An end-fed halfwave, often referred to by the acronym EFHW, seemed to be a good choice, as it only requires a support at the far end. I saw photos that Steve WG0AT had posted on Facebook of his little EFHW, with the matching unit built into a dental floss container, for a light and compact solution. I wanted an antenna that small and lightweight! Steve referenced a blog post by TJ W0EA, in which TJ detailed an EFHW matching unit he had made, based on the one in his Par End Fedz antenna. This little matching unit, that transforms

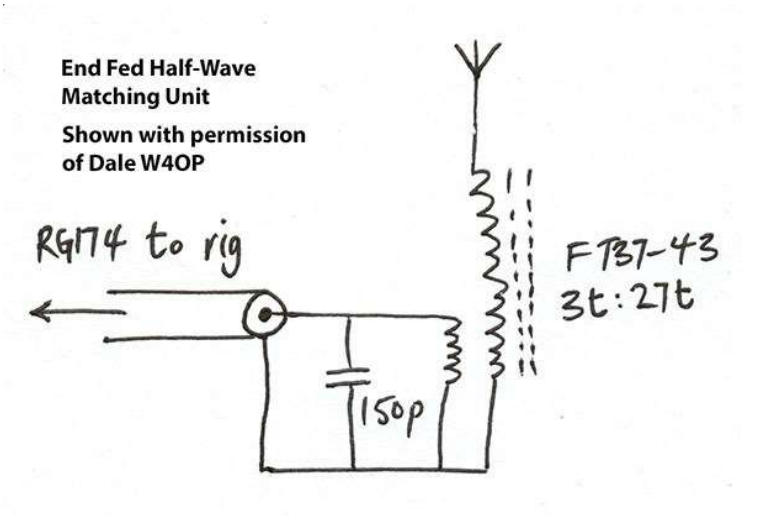


Fig. 1 The Circuit

the high impedance present at the end of a half-wave length of wire into the much lower impedance of 50 ohm coax, consists of a wideband transformer wound on a ferrite core, and a 150pF fixed capacitor. That's it. Simple and compact!

What to put it in, was the big question. I spent several weeks looking in stores for suitable small containers, and finally decided on a Carmex lip-balm tube. Here it is with the lip-balm removed –

The remaining tube still has a corkscrew-like central element that needs removing –

It is a fairly simple matter to grasp the corkscrew with a pair of long and slim needle-nose pliers, and push it until it pops out. You can discard the corkscrew, as it is not needed. The 2 parts on the left of the next picture, the snap-on lid and the main cylinder, are what you want –

The following pictures should show you how it all goes together. A plastic cable tie prevents the RG174 from pulling out of the bottom, and a

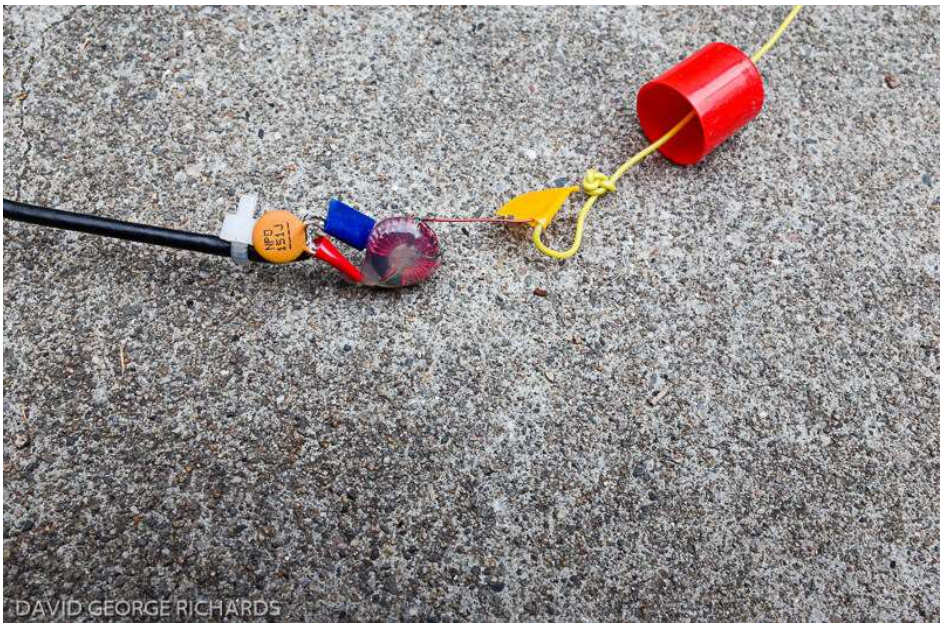
generous squodge of hot glue keeps the toroid in check. If you have a dual temperature glue gun, use the hotter setting –

This matching unit is designed to work with a half-wavelength wire. Some folk build it so that they can change the wire length for different bands. I


DAVID GEORGE RICHARDS


DAVID GEORGE RICHARDS


DAVID GEORGE RICHARDS


DAVID GEORGE RICHARDS

decided to make this a permanent 20M antenna, so started with about 36 feet, and continued to trim it down until the center frequency was close to 14060, at which point the SWR was 1.1:1. Not bad! I'll state the obvious by reminding you that any antenna does need to be reasonably clear of nearby objects, particularly anything conductive, in order to make meaningful measurements. Laying it on the ground isn't going to cut it – you need to suspend one end up in the air and have the antenna clear of obstructions. This is what my final EFHW looked like, all bundled up and ready for the trail, with a 10 foot length of RG174 –

Interestingly, a few days later, I checked the SWR again, only to find that although the center frequency was the same, the SWR at that point was higher, at about 1.4:1. The only thing that had changed was that the first time I measured the SWR, I was powering my MFJ SWR Analyzer from a "wall wart" transformer while the second time, it was powered from internal batteries. I'm thinking that the first time around, the AC wiring in the house was providing a bigger counterpoise and helping to lower the SWR at resonance. It might be interesting to try connecting a counterpoise wire at the rig to see if it reduces SWR any, but I did like the added simplicity of no counterpoise.

How does it work? I bundled the SST, antenna, small sealed lead acid battery, paddle from QRP Guys, and a few other things into my backpack, and cycled up to Vollmer Peak, a local high spot in the Berkeley Hills. I left rather late, had lunch on the way, and by the time I got up to the top, spent about 30 minutes eating trail mix and looking at the view, before realizing that I didn't have much time. I didn't get the antenna very high in the tree, and sat on the ground, listening, finishing off the trail mix, and putting out a few CQ's before heading back down the hill. End result = no QSO's, but I did get spots on the Reverse Beacon Network from Colorado, Arizona, and Alberta. The antenna works – it's the operator who performs better in a cozy

indoor shack.

There is really only one more thing to try with my SST, and that is, as I mentioned in my blog, AA7EE.wordpress.com **"General Harmonic Suppression and a Narrower RX Filter for the SST20"**, to add extra filtering between the TX mixer and the buffer/driver. I think that a lot of harmonic energy is making it to the final and being amplified, before being filtered out by the LPF in the antenna lead. Better to nip all those naughty harmonics earlier in the process, I think. If I do any more work on it, that will be the focus.

Thanks to Ian MW0IAN (great callsign) for clueing me in to the PDF on the G0KYA EFHW antenna. Check out the link below.

http://www.infotechcomms.net/downloads/Endfed_halfwave_dipoles.pdf

Snort's Shorts

by Steve Smith, WB6TNL

Hello, I am Steve "Snort Rosin" Smith WB6TNL and welcome to this month's edition of "Snort's Shorts" column in QRPp. Each month I present Snort's Shorts Hints, Mini Reviews, project ideas, and contributions submitted by QRPp readers.

Reminder!!: Like a QSO, this column is two-way:
Your hints, reviews or project ideas are always welcome for inclusion in one of my columns. So please, share them with your fellow QRPers.

Email to: <sigcom@juno.com>, Thanks!

Snort's Shorts Hints

Prolific brand USB to serial converter chip problem

Everyone knows that the Serial (AKA : COM) port has gone the way of the Dodo Bird. Practically every peripheral these days uses USB for serial data communication. A serious problem has happened because of I.C. counter-feiters who have illegally copied the integrated circuit chips used inside the USB to Serial adapters. These adapters include generic USB to Serial cables, the Kenwood compatable programming cables used by most, if not all, of the Chinese FM handheld radios and the USB to Serial modules used to program Arduino and other micro-controllers not containing on-board adaptors.

The most common convertor chips copied are those manufactured by "Prolific" and"FTDI". Apparently, all of the counterfeited Prolific chips contain the same Vendor ID number (VID_067B) and Product ID number (PID_2303) as contained in the original, genuine Prolific I.C.s. To thwart this, Prolific changed their drivers so that any device using the counterfeit chips would be rendered unusable. Unfortunately, this also causes USB to serial adapters using those chips to cease functioning and you have to go out and buy new adapters containing a genuine, updated Prolific chip.

The work-around for Prolific chips was to download and install the earliest Windows driver (V3.3.2.102) for the Prolific chip. This worked well, that is until Windows 10 came along with its automatic updating. What happens with Win 10 is that the legacy driver installs just fine but almost immediately, the operating system updates the driver, rendering the Prolific chip inoperative.

If one of these Prolific 64-bit drivers gets installed to your Win 10 computer then your legacy device will no longer work and will issue the generic

“This Device Cannot Start (Code 10)” error messaage. Even more aggravating is that sometimes there is no error message at all; the device simply refuses to operate.Fortunately, the solution is simple enough. Family Software

<http://www.ifamilysoftware.com/news37.html>

has published a patch that will not only install the legacy driver but also modify the Windows 10 registry to prevent it from automatically updating the driver without your permission.

Snort’s Mini-Reviews

Velleman Model DVM850BL Mini Digital Multimeter

This month I present the Velleman Model DVM850BL, a compact, toolbox size DVM with a ruggedness making it “field worthy”. I purchased this meter a couple of years ago from Fry’s Electronics. It is often on sale there for $9.00 and that is what I paid. This particular meter was purchased to replace my Craftsman DMM which I had carried around in my toolbox for many years when I was still doing field repair. The Craftsman is still alive but the test leads wore out and I could not find a new set that satisfied me.

L-R: DVM810, Harbor Freight Tools #98025/90899, DVM850BL for size comparison

I like the look and feel of this instrument along with the nice, large LCD display; very easy to read. The flexible plastic “surround” keeps the hard plastic housing from getting gouged if dropped (this happens often in the field).

Basic Features:

Automatic polarity function and 3-½ digit LCD display
Measurements: DC current up to 10A, AC and DC voltage up to 600V, resistance up to 2Mohm
Diode, transistor and continuity test with buzzer
Data-hold function and backlight
With (plastic) protection holster
Insulation rating: CAT. II 600V

Specifications

DC voltage: 200m/2/20/200/600V
Basic D.C. accuracy: ± 0.5% of rdg ± 2 digits for 0.2V range / ± 0.8% of rdg ± 2 digits for 2V~200V range / ± 1.0% of rdg ± 2 digits for 600V range
Input impedance: 1Mohm
Maximum D.C. input: 600V
AC voltage: 200/600V
Basic A.C. accuracy: ± 1.2% of rdg ± 10 digits
Input impedance: 1Mohm
Frequency range: 40 to 400Hz
Maximum A.C. input: 600V
DC current: 200µ/2m/20m/200m/10A
Basic D.C. current accuracy: ± 1.0% of rdg ± 2 digits for 200µA~20mA range / ± 1.5% of rdg ± 2 digits for 0.2A range / ± 3.0% of rdg ± 2 digits for 10A range
Overload protection: Fuse, 500mA/250V - 10A/250V
AC current: No
Resistance: 200/2k/20k/200k/2M Ohms
Basic resistance accuracy: ± 0.8% of rdg ± 2 digits for 200 ohms~200k range / ± 1.0% of rdg ± 2 digits for 2Megohm range
Overload protection: 250VDC or RMS AC on all ranges
Capacitance: - No
Inductance: - No
Frequency measurement: - No
Temperature measurement: - No
Over-range indication: yes = “1”
Continuity buzzer: yes
Transistor test: yes
Diode test: yes
Low-battery indication: yes

Maximum display digits: 3-1/2 (1999)
LCD display size: 1.8" H x 0.7" W (4.6 X 1.8 cm)
Range selection: manual
Data hold: yes
Backlight: yes
Dimensions: 5.4"H x 2.7"W x 1.2"D (13.7 X 6.8 X 3.0 cm)
Weight (with battery): 13.3 oz. (377 Grams)
Power supply: 1 9V 6LR61 battery (incl.)
Accessories: - Red/Black test probes, incl.

Pros:

Compact size; fits nicely in the palm of the hand.
Light weight
Reasonable range of measurements
Reasonable accuracy
10A DC current range
Good display size and contrast
Quality “feel” of range selector switch

Cons:

The socket for transistor HFE testing is almost useless; the socket makes poor contact with the transistor leads.

The continuity “buzzer” is practically inaudible; it really needs some help. Perhaps a date with Mr. Mod.?

Opinion:

For a $9 meter, it is quite good. The test leads have thick insulation and I would not be concerned about operating this meter at the extremes of its voltage ranges (600V A.C. & D.C.).
The blue LCD backlight is nice in low light conditions and the “hold” function is quite handy if one cannot watch the meter and hold the test probes on the DUT (Device Under Test).

Based upon my experience with this inexpensive instrument, I can recommend it for both use on the bench and in the field.

Harbor Freight Tools Six Piece Hook & Pick set, Item #93514

For years I’ve used dental picks for manipulating wires, scraping PC board traces, cleaning corrosion from metallic parts and lifting components while de-

soldering. Some of the handiest tools I've owned. But over the years I've lost, broken or worn out all but two of the stainless-steel dental picks and needed something to use in "The Lair" (my indoor work area).

Overall pick lengths range from 5-1/2 inches (14cM) to
6-5/8 inches (16.8). Two of the picks have knurled shanks which provides a nice feel; the other four have square, polished shanks.

HFT does not indicate the materials used in the manufacture of these picks. My guess is that they are chrome-plated steel. For the price, I don't believe they are stainless-steel. Only time will tell how they hold up under chemicals, solder and high temperatures.

That wraps things up for this month. 73 and Keep Snortin'!…….Steve Smith WB6TNL

The QRP Guys Counter: A Bench Counter For the Lab

By Doug Hendricks, KI6DS

QRPguys is a new qrp kit company that has really good prices on their products. I took one of their through hole counter kits that sells for $15 and uses all through hole parts and made it a Lab Counter. It has 5 digits, and uses a red LED Display which you can see in the pictures. I like to build, and need a counter to set dials, vfo top and bottom, and to check for where a local oscillator is. I like a bnc connector, and I like to power it from a 9V battery to make it portable. The QRPGuys counter meets all of my requirements except one. It doesn't come in a case. That is easily remidied by building a case of PCB stock. The result is shown in Fig. 1. You will note immediately that there is a

Fig. 1 QRPGuys Counter in PCB Case

rectangular cutout for the display. How did I do that? Easy, I used an Adel Nibbling tool. You can order one on Ebay or Amazon. I have had mine for several years but I think one can be had for eight or nine dollars. The BNC connector connected to the cable on the left is for signal input.

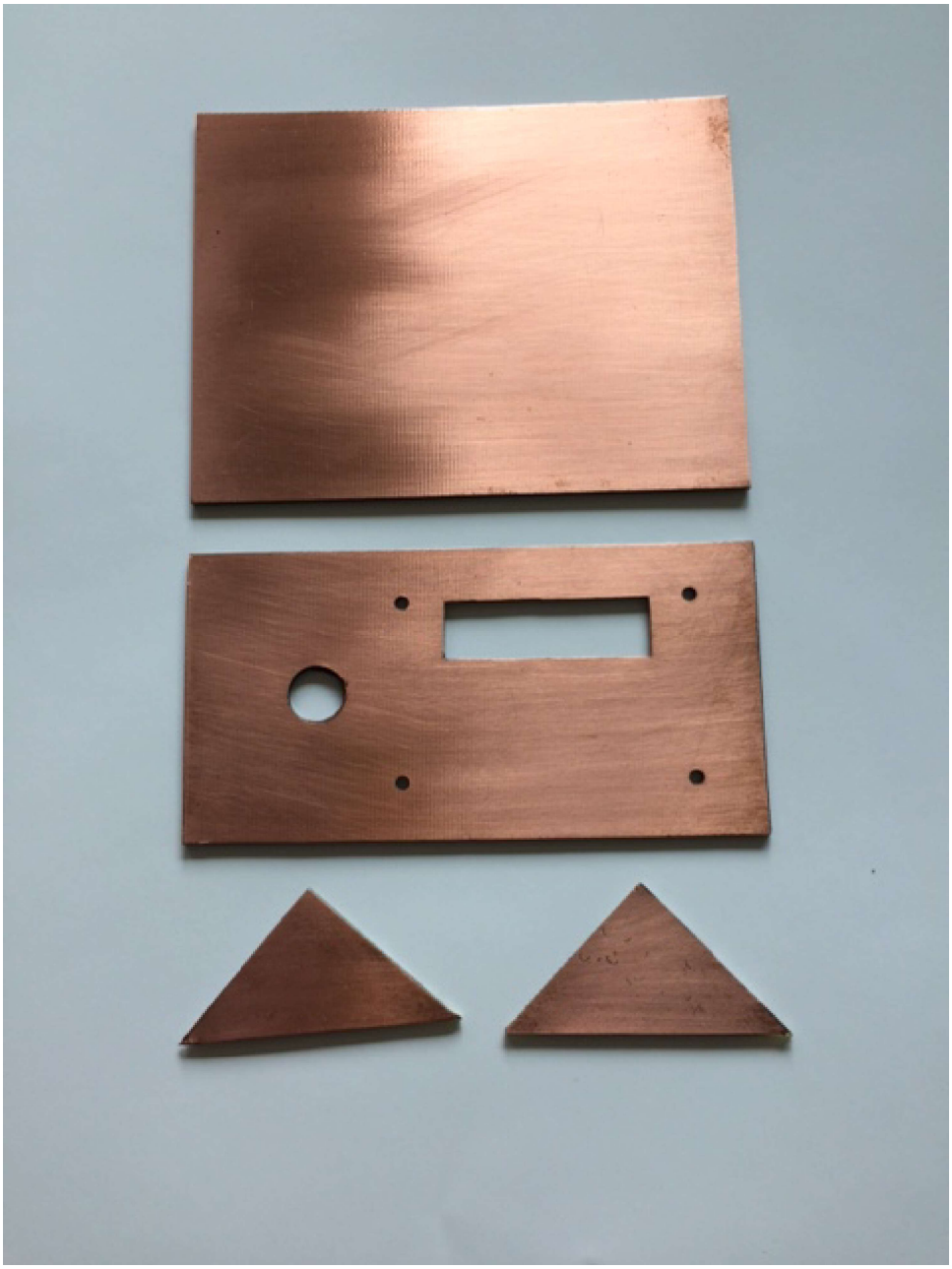


Fig. 2. The four case pieces.

The case is made of 4 pieces of double sided pcb stock. The measurements for my case are 4” wide x 2” high by 3” deep. I use 1/16” thick stock. I cut mine with a pair of tin snips after marking the lines to cut. I cut the front panel and the base out of the same piece of stock so the width is exactly the same. I start with a piece that is 4” x 5” (one piece 2” x 4” and the other 3” x 4”). I make the triangular shaped gussets out of left over pieces.

QRPGuys has a template for cutting a hole for the display on their website, www.qrpguys.com. Do a search on through hole display and you will find the template in the manual. I used the template for my measurements and it worked very well.

Making the case is easy if you use the flollowing procedure. First you need to make a WA4MNT case jig. Go to Home Depot and buy 1 x 4 that is 2 feet long. Cut three pieces 6” long and assemble with screws and glue as shown.

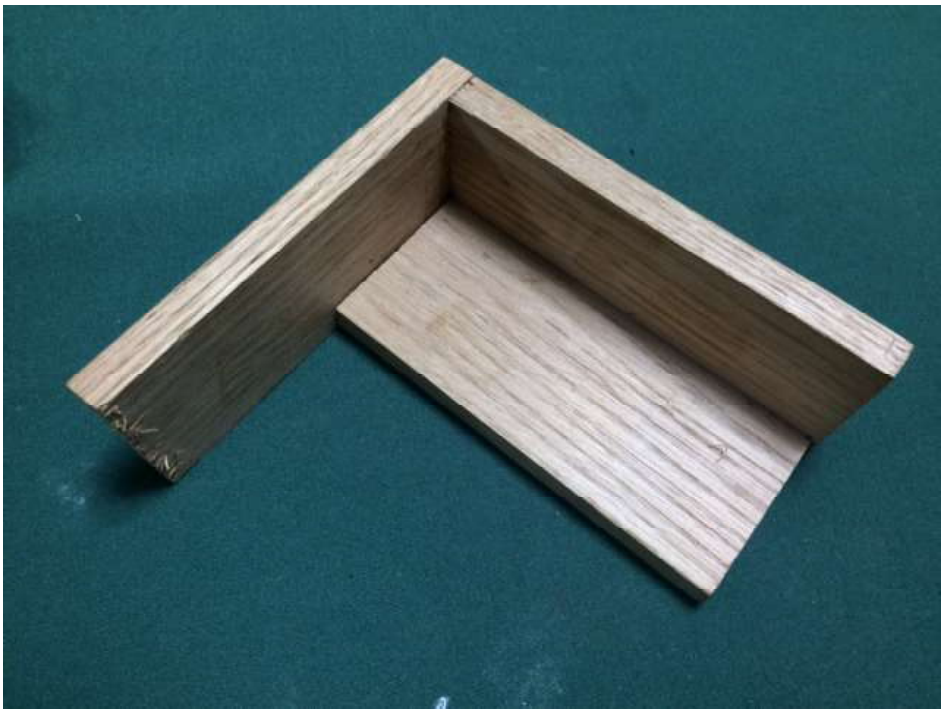


Fig.3 WA4MNT PCB Case Jig

It is extremely important that the ends of the boards that are joined together be cut perfectly square. Check with a square and a test cut. You will also need two hand squeeze clamps. They are available at Home Depot, Lowe’s, etc.

Fig. 4 PCB Case Jig with Clamps

Fig. 5 .030 wire or solder for offset gauge

You will also need a piece of wire or solder that is .030” in diameter to use as an offset gauge. When you solder two pieces of pcb together, it will spring back as it cools, even if clamped. To allow for this, you place an offset gauge between the front panel and the PCB gauge as shown. This will make the front panel sit at an angle and when it cools, it will spring back to be very close to square.

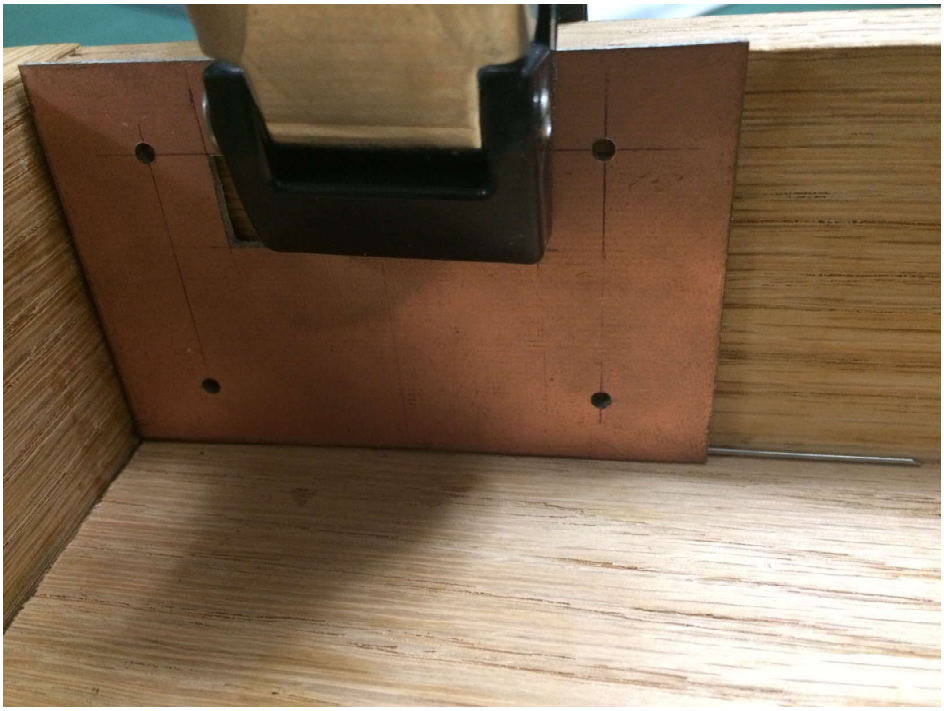


Fig. 6 Front Panel clamped to PCB jig

The first step is to place the offset gauge against the side of the PCB jig. Next, place the front panel and clamp. Make sure that it is tight agains the left edge as shown. You can see the end of the solder offset gauge sticking out from behind the panel. I like to set the front panel first so that when you finish you will not see the edge of the bottom. Looks neater.

The next step is to clamp the bottom against the front panel. Make sure that the left side fits tight against the end of the PCB Jig, and that the bottom edge is flush with the front panel. You are now ready to solder. You don’t need to do a solid solder joint along the whole edge. I usually do 5 spot solders, 1 on either end starting in about 1/2”, and space the other three between the two ends. It is important NOT to solder at the edge because it will get in the way of the two gussets.

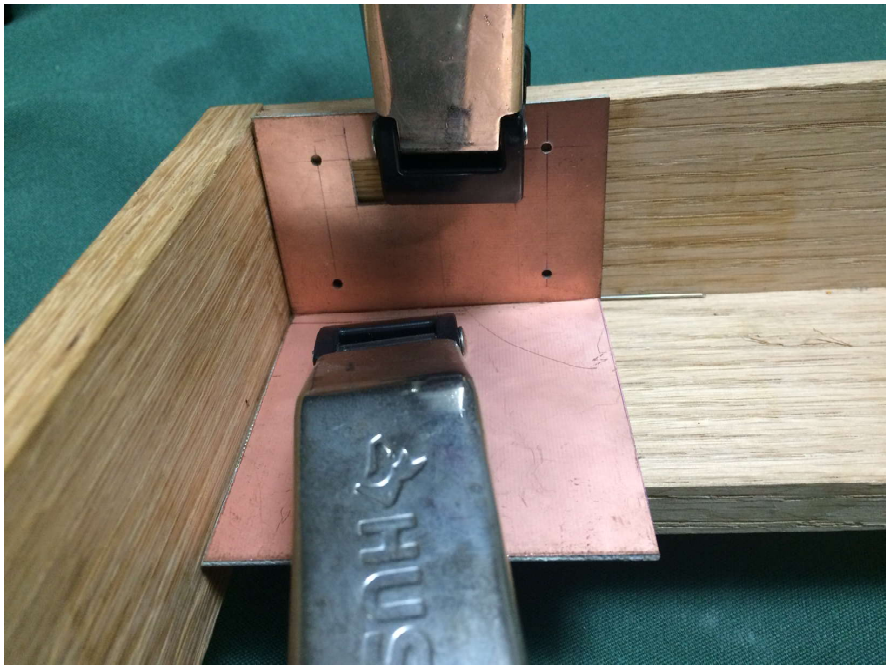


Fig. 7 Bottom Panel clamped in place ready for soldering.

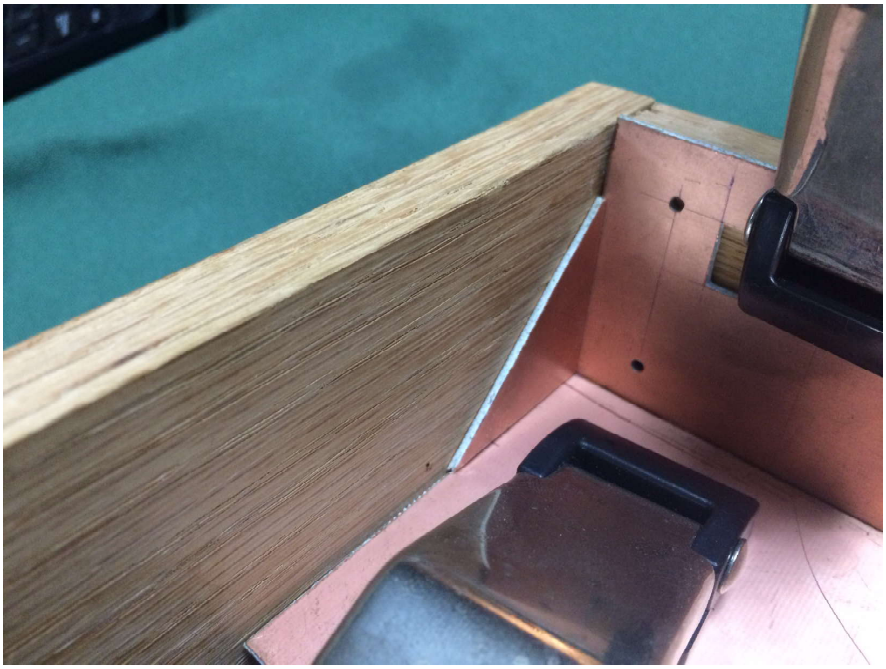


Fig. 8 Detail of Gusset placement

The next step is to solder in the left and right gussets. I make the gussets out of scrap pcb, Usually I use a 1.25” x 1.25” piece cut in half diagonally. Just make sure that the piece is square. Place the gusset as shown, then tac it on either end on the bottom. This will secure it to the bottom. Make sure that it is flush against the bottom and the corner edge is against the front panel at the bottom. After the bottom is soldered, then unclamp the front panel and using your fingers pull the front panel back until it is flush with the front edge of the gusset. While holding, solder the top corner of the gusset. Hold until it is cool, then finish soldering the bottom corner.

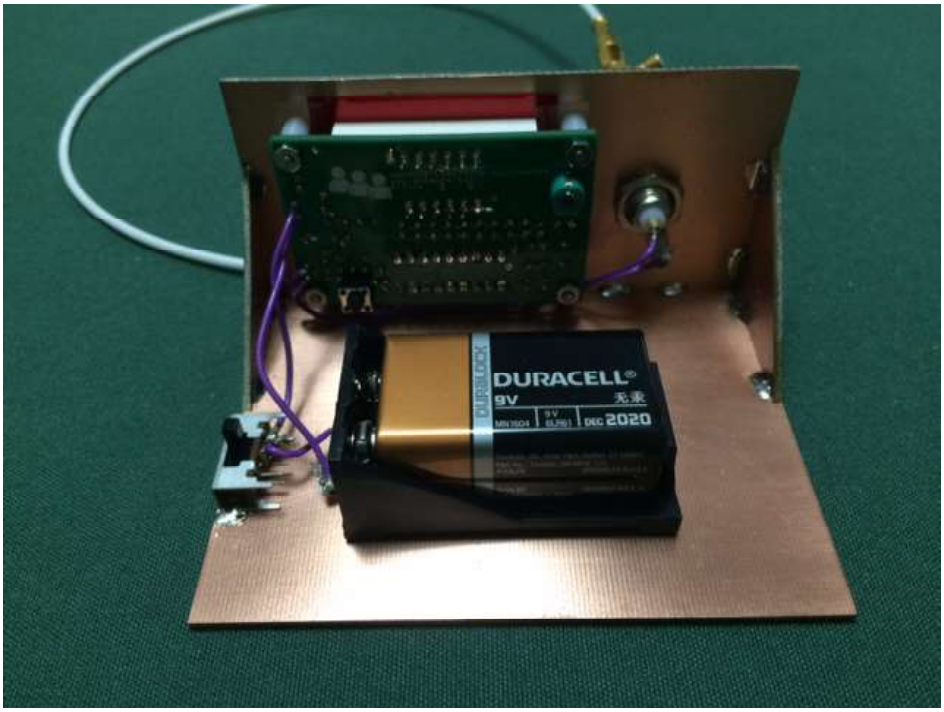


Fig. 9 Back view of the finished case.

When you finish the case, then it is a simple matter of mounting the Digital Display, battery case, switch and bnc connector as shown in the following photos. I really like the convenience of the counter. I had it mounted in an Altoids tin, but think that this method is better, and easier to use. Enjoy.

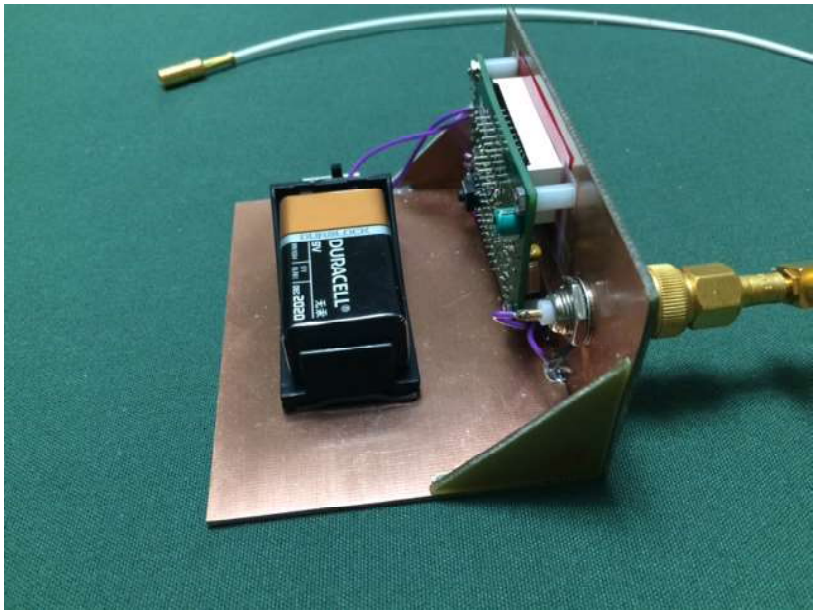


Fig. 10 Left End View

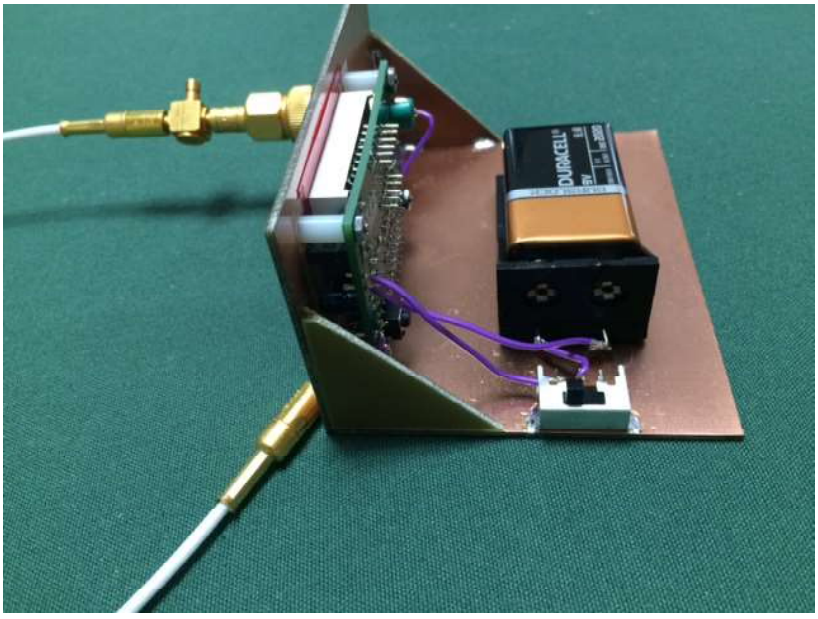


Fig. 11 Right End View

Build the 9-5-3 Power Supply

By Steve Smith, WB6TNL and Doug Hendricks, KI6DS

Several weeks ago I was at the Bay Area Builders Group meeting in Cupertino. Bob Mix hosts the group on the second Sunday of the month. We used to meet at Panera Bread, but Bob has changed meeting places. Contact him for meeting place. One of the guys there, and I did not get his name or call, had a neat little power supply. It consisted of a circuit containing a 5 volt and 3.3 volt regulator, the associated decoupling caps and it was powered by a 9V battery. Pretty neat circuit. I was telling Steve about it on the phone later that day and mentioned that I though it would be neat to buid one. Steve suggested that I could save parts if I used an LM317, some resistors to set the power level, and a jumper system using a berg connector and 2 pin headers. I told him to send me the circuit and I would lay out a board and build it.

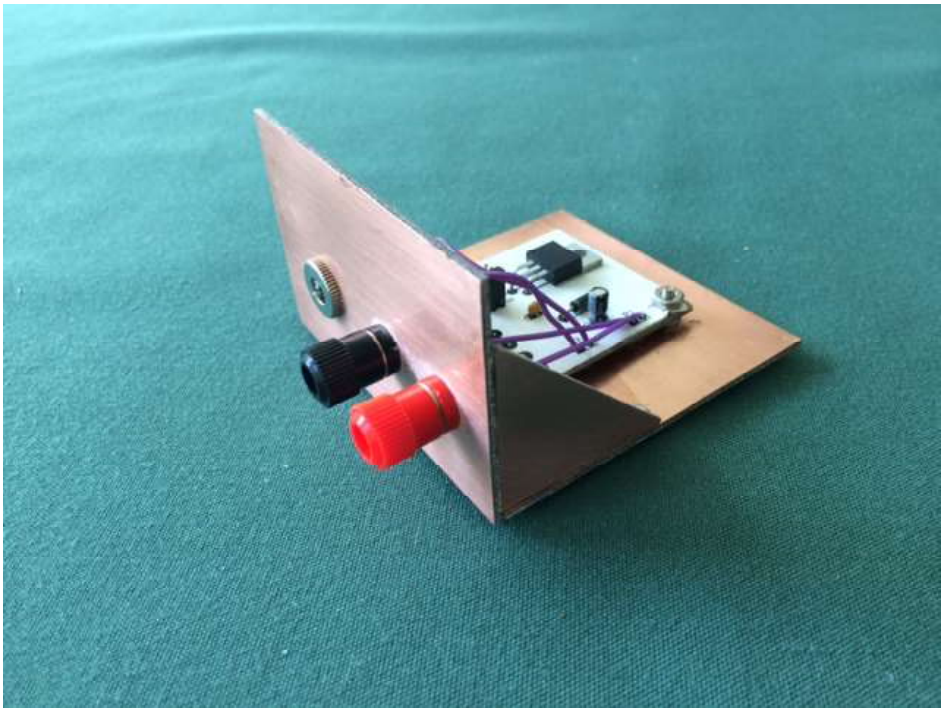


Fig. 1 9-5-3 Power Supply

The next day Steve sent me a schematic and parts list. It is in Fig. 2. I got busy and layed out a board. It was straight forward and easy to do. Orginally I put it in an Altoids tin, but when I did the Lab Frequency Counter found on page 42 of this issue, I liked it so much that I decided to build it in an L-shaped case. Details of making the cases can also be found in the Lab Counter article.

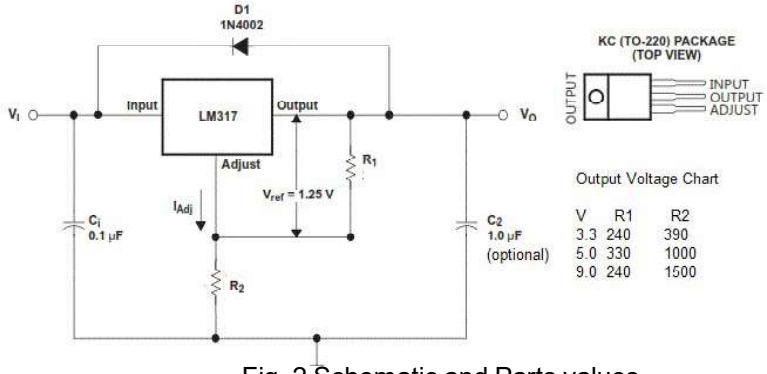


Fig. 2 Schematic and Parts values

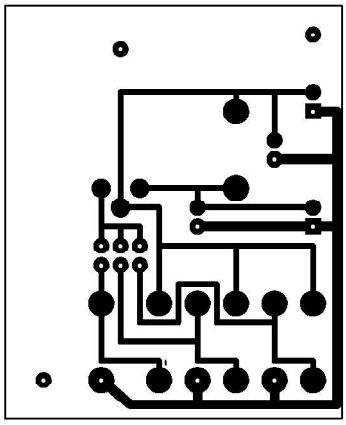


Fig. 3 Print ready artwork

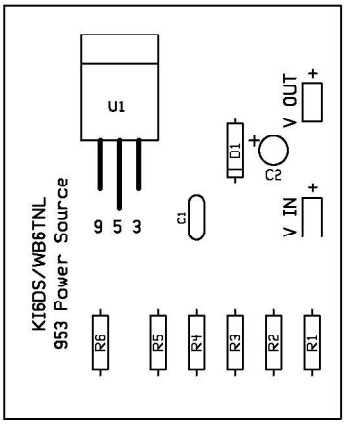


Fig. 4 Silkscreen

Here is the parts list.

U1 - LM317
R1,5 - 240 ohm, 1/4W resistors
R2 - 390 ohm, 1/4W resistor
R3 - 330 ohm, 1/4W resistor
R4 - 1K, 1/4W resistor
R6 - 1.5K, 1/4W resistor
C1 - 0.1uF capacitor
C2 - 1uF/35V electrolytic capacitor
D1 - 1N4001 Diode
1 - Chassis mount 2.1mm power connector
4 - rubber feet
1 - red binding post
4 - ,25" x 4-40 ph screw
1 - black binding post
8 - 4-40 nuts
3 - 2 pin connectors
1 - Berg connector
1 - PCB
1 - Set case parts

One of the problems with making your own boards is that you don't have a silkscreen on the board for parts placement. I was thinking about this and decided to try an experiment. I printed the silk screen on a mailing label, and then put the mailing label on the board. Instand silk screen. I then used a probe to poke holes through the label for the leads. It worked very well, and I will do it again. The picture below shows the result.

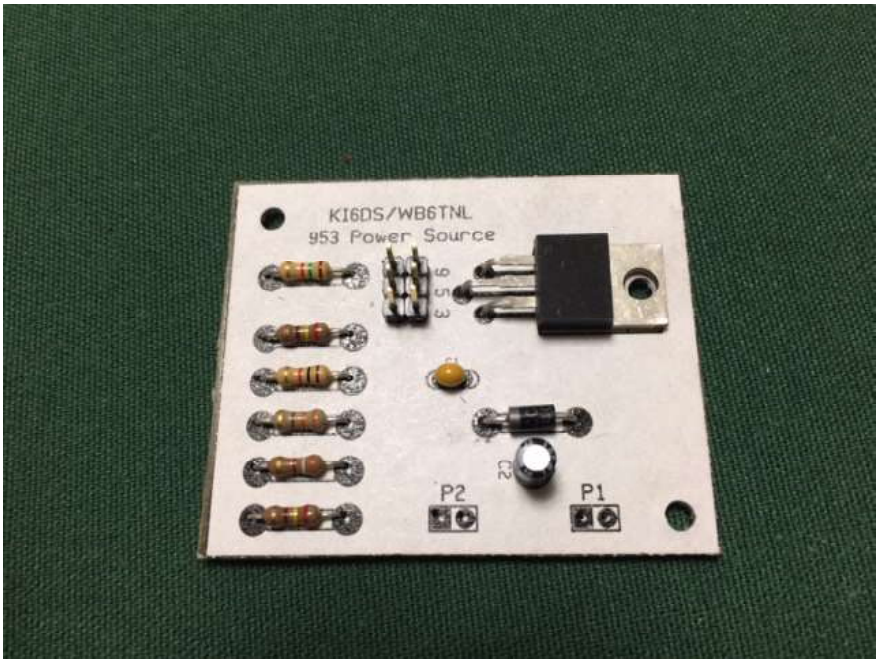


Fig. 5 Board with silkscreen

Building the power suppy is straight fowward. Make your board using the method of your choice. Then drill the board. Put the parts on the board and verify that you have 12V in to the LM317. Place the Berg connector on the 9V position and verify that you have 9V out. If not, check for solder bridges or incorrect parts placement. Do the same for 5V and 3V positions. My supply measured 8.95V, 4.95V and 3.2V. Yours may vary but should be pretty close...

I then drilled the front panel and the base in order to mount the connectors and the board. To mount the board, insert the screws from the bottom, tighten a nut and then put the board on followed by a nut. The first nut will act as a spacer to keep the bottom of the board from shorting out. Install the feet.

I installed the connectors and wired the input V to the 2.1mm connector and the output V to the binding posts. The pictures on the following page show more detail. Steve did a great job with the design of the circuit, and together, I think we came up with a fun and useful project.

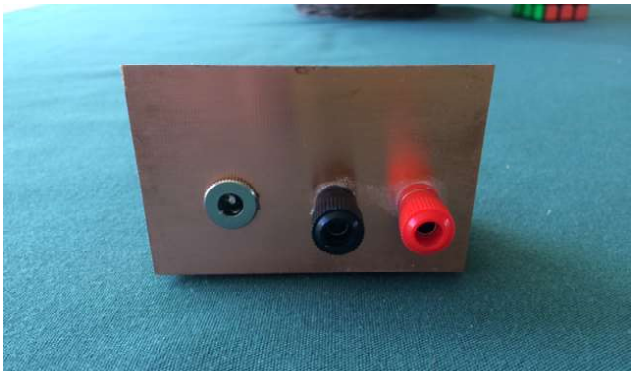


Fig. 6 Front view

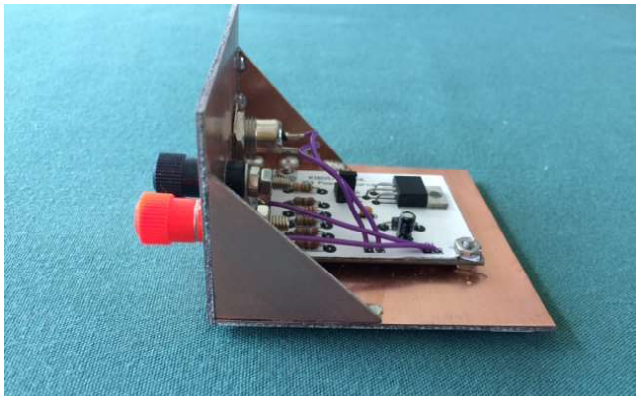


Fig. 7 Side view

Fig. 8 Rear view

Universal Low Pass Filter for Simple Chinese Radios

by Steve Smith, WB6TNL

We live in an exciting time for builders of QRP equipment. The Chinese have flooded the market with cheap, simple cw transceivers. The Pixie, Froggy, Octopus, etc. are all available on ebay, and can be found for as little as $3 delivered to your door. It just doesn't get any better. But, there is a problem. Most, if not all of the kits are not legal to operate here in the US as designed. So, I decided to provide a simple solution for the problem. I layed out the circuit, and had Doug, KI6DS make a board for me. THe result is shown below.

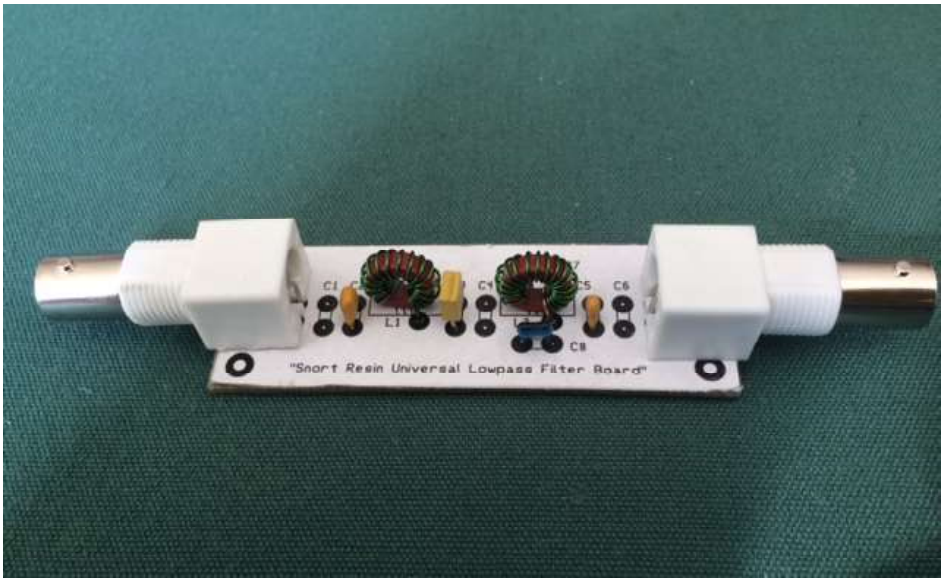
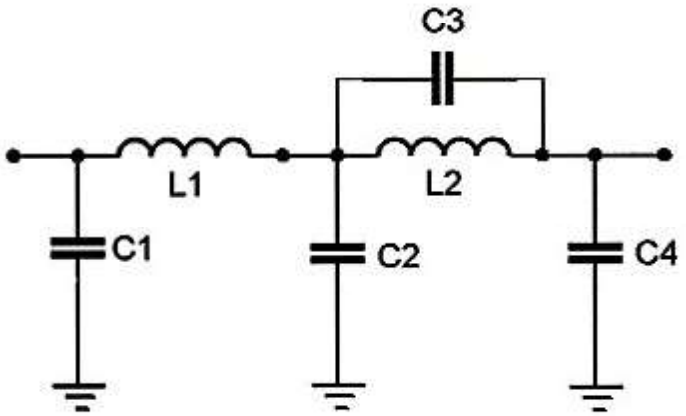


Fig. 1 WB6TNL Cauer Low Pass Filter

The FCC rules say that the second harmonic must be more than 43dB down from the fundamental. To accomplish this I have designed a 6 section filter with a Cauer section. You will note that two positions have been provided for each capacitor position. This is to enable the builder to combine values to achieve the desired result. Say you need a 1500pF capacitor but don't have one in the junk box. But you do have 2 750pF caps. Use both of them. Since they are in parallel their value will be 750 = 750 = 1500, since capacitors in parallel are additive.

Construction is easy. Just pick your band and install the appropriate parts. Board artwork and silk screen pattern are in Fig. 3 and 4. Simply connect the filter between your rig and the antenna, and you will be legal. Disclaimer: I have not measured the filter but according to calculations it should meet or exceed FCC specs for all bands listed.



BAND	C1,C4	C2	C3	L1	L2
80	680	1500	100	2.9-3.0	2.5-2.6
40	330	680	68	1.1-1.2	1.4-1.5
30	220	560	47	0.7	0.9
20	150	330	22	0.6	1.0

All capacitance in pF
All inductance in uH
Powdered-iron cores: 80-30 Meters: -2 material
20 Meters: -6 material

Fig. 2 Schematic

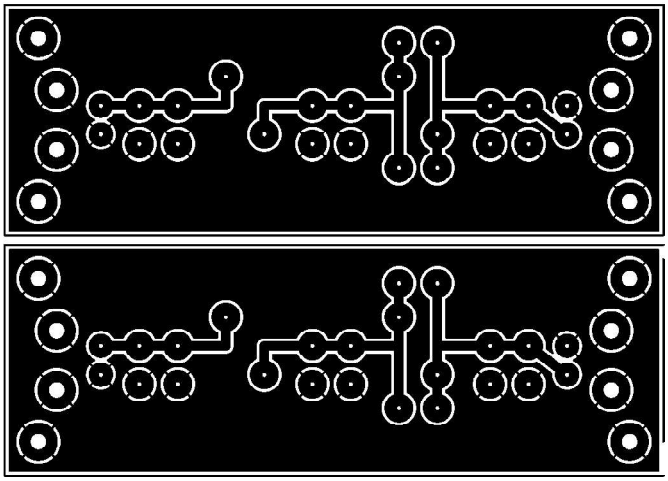


Fig. 3 Print ready artwork

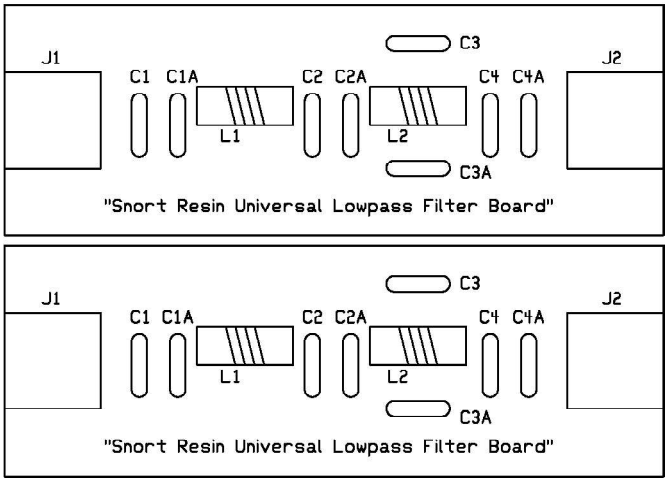


Fig. 4 Silkscreen

Joe Everhart's, N2CX, Drive On Mount

by Doug Hendricks, KI6DS

If you read QRP-L at all, you know Joe Everhart, N2CX. Joe is a QRP Hall of Famer, and an avid QRPer. His current passion is doing National Parks on the Air, and he has done more than 100 sites so far this year in the ARRL sponsored event. The objective is to set up portable and make at least 10 qso's from each National Park Service site. The ARRL has a list, and it is a long one. Joe, out of necessity needed a quick way to get on the air, as he was going to set up and tear down literally hundreds of times. I asked him in an email to describe what he used, and he told me about what he had developed. He also told me that Craig LaBarge told him about it.

The things that I like about it are that it is cheap, parts are all available at Home Depot, only takes one person to set up and tear down, and it is light and NOT bulky. I had seen several drive on mounts that were made from metal, but they were all heavy and bulky. Joe's method really appealed to me. I am going to show you in pictures how to build it. First of all the parts:

Fig. 1 Parts needed to build the Drive On Mount
Top: 1 inch pvc to pipe thread coupler
Middle: 1" x 6" x 24" Popular board and 1" pipe flange
Bottom: 1" x 24" PVC Pipe

All of the above are available at any hardware store, Home Depot, Lowe's, etc. The pipe flange is mounted to the board with 4 flat head machine screws and are counter sunk on the bottome as shown in the next picture.The pictures on the following pages will show you how to reproduce the mount.

Fig. 2 & 3 Detail of mounting the pipe flange


Fig. 4 & 5 The PVC to pipe thread coupler


Fig. 6 & 7 The mount assembled and the assembly under the tire

Fig. 8 The mount with pole in my drive way

My thanks to Joe Everhart for sharing his mount with me. It works great with any of the currently available pushup poles from the various vendors. I use a 33” pole from Sota Beams in England.

Notes