

Adding a Keyer to the Arduino “Freq-Mite” for the NC40A

Mike WA8BXN

Jan 2018

As mentioned in my previous project implementing a frequency readout in CW for the Norcal NC40A similar to the functionality of the K1SWL “Freq-mite” using an Arduino, the program used there spends most of its time waiting for the button to be pressed telling it to measure and report the dial frequency. That leaves a lot of processing power available to perform the logic needed to implement a keyer. This project provides such an implementation.

The basic logic for an iambic keyer is simple:

```
repeat forever:
  if dit paddle pressed then key transmitter for 1 dit time
  if dah paddle pressed then key transmitter for 3 dit times
```

If just one paddle is pressed the corresponding dit or dah will be sent. If a given paddle is pressed and held a string of dits or dahs will be sent (we do have to provide a dit time pause between them). If both paddles are pressed and held a sequence of dit dah dit dah etc or dah dit dah diit etc will be sent, depending on which paddle was pressed first.

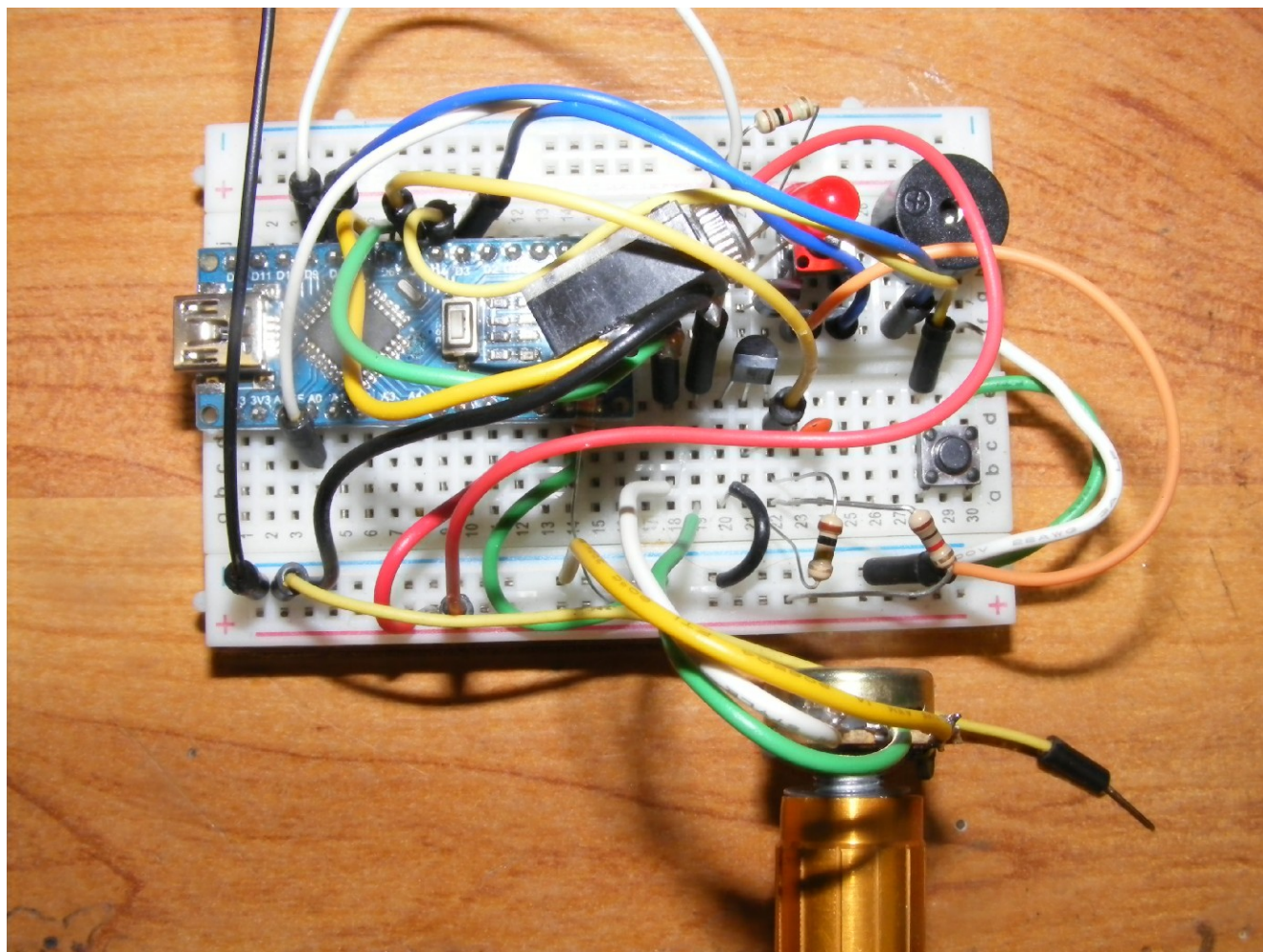
In the Arduino environment testing for the press of a paddle uses the digitalRead function. Transmitter keying can be done with the digitalWrite function and there is a delay function that can be used to pause in the code for the number of mS that corresponds to the dit time. A keyer could be written in very few lines of code.

If you look at the program below you will see that more than a few lines of code have been added to the previous program that just did the frequency counting. There are a number of reasons for that. Included is more than just a very basic keyer, such as being able to send a stored message (optionally repeated after a few seconds) and the ability to adjust speed during operation with a potentiometer.

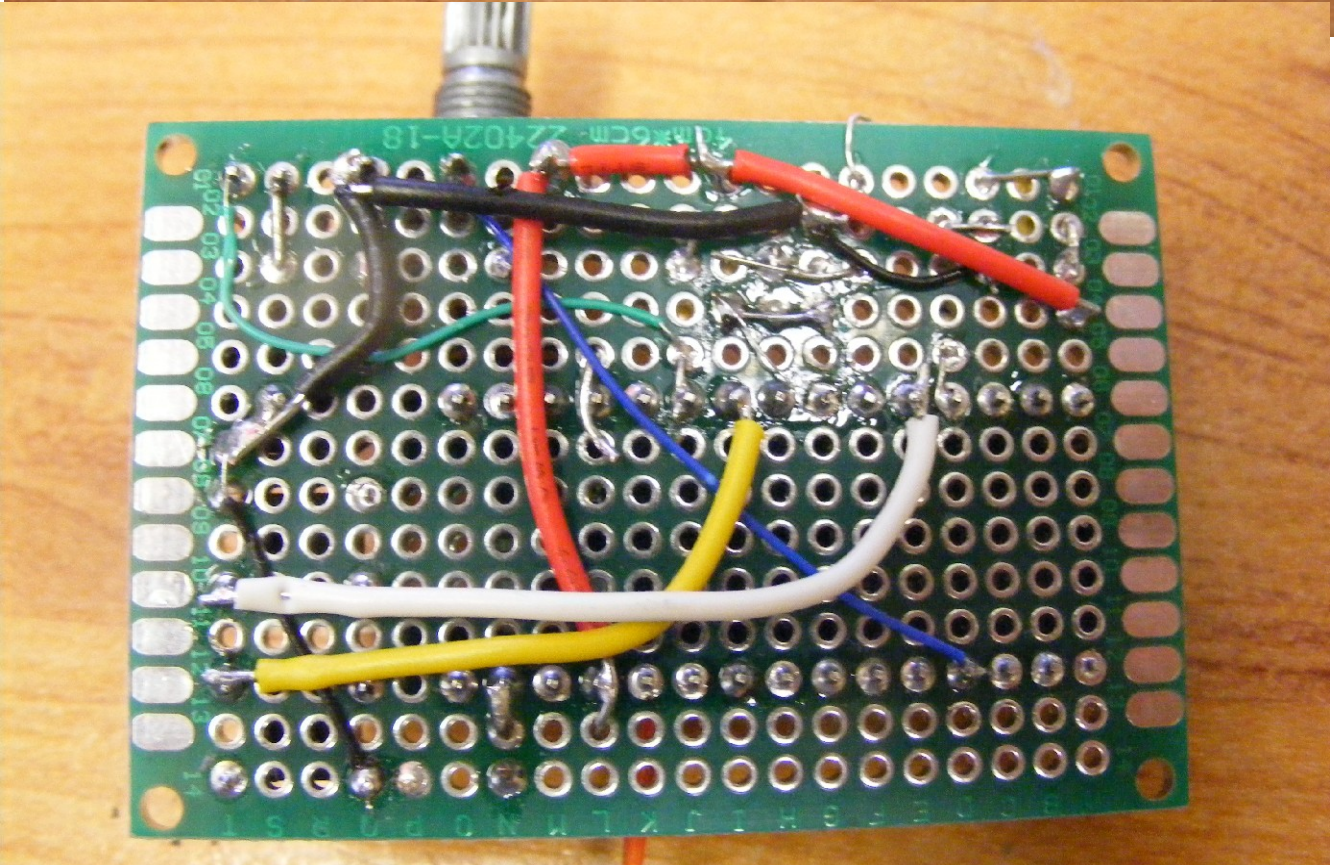
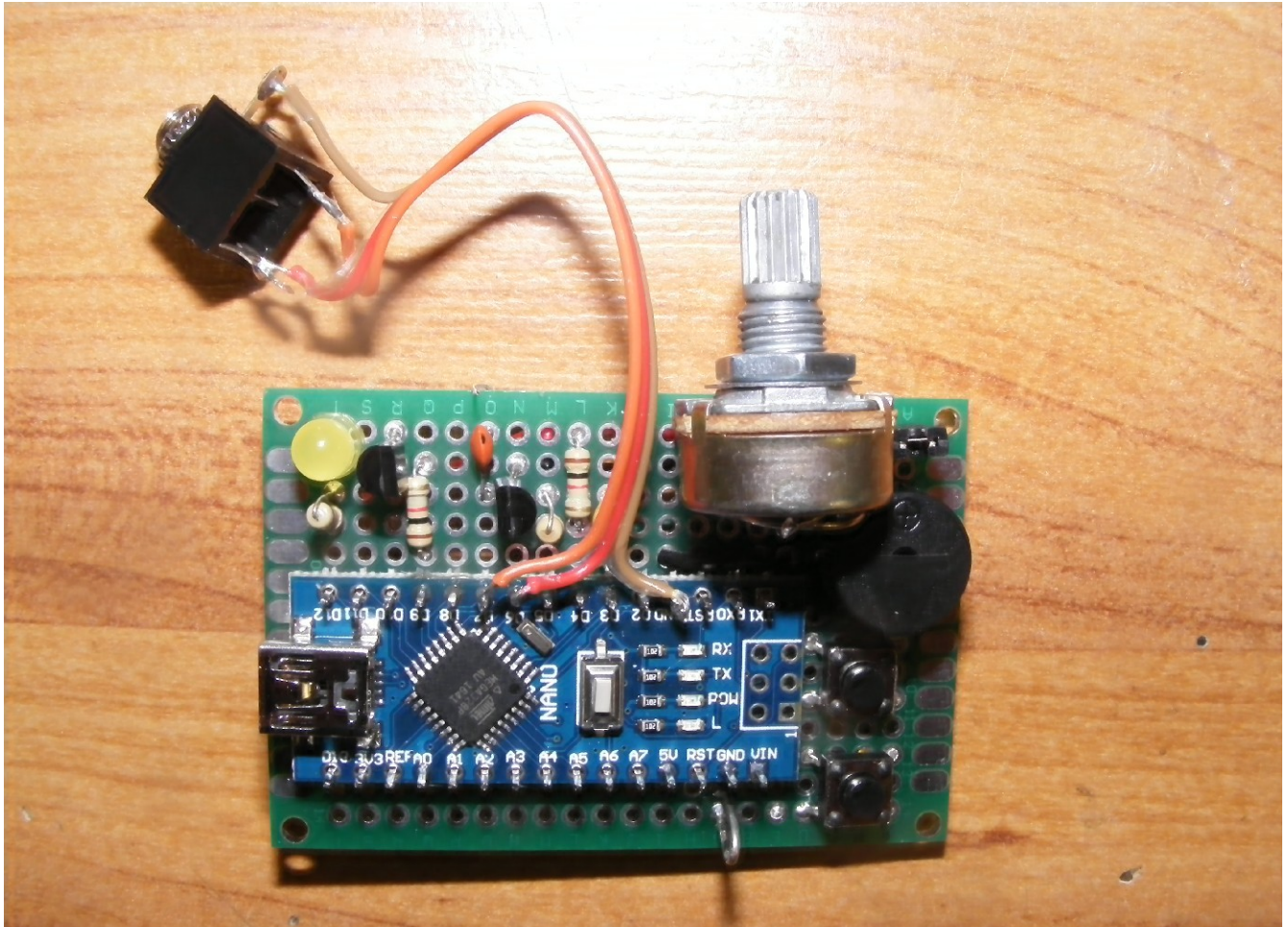
Since the same program must still include the frequency counting ability there are some complications in getting it just right as to when the transmitter needs keyed and sidetone needs generated. We don't want to send the frequency out over the air but we must hear it on the sidetone. There are also little issues like being able to interrupt the transmission of a stored message.

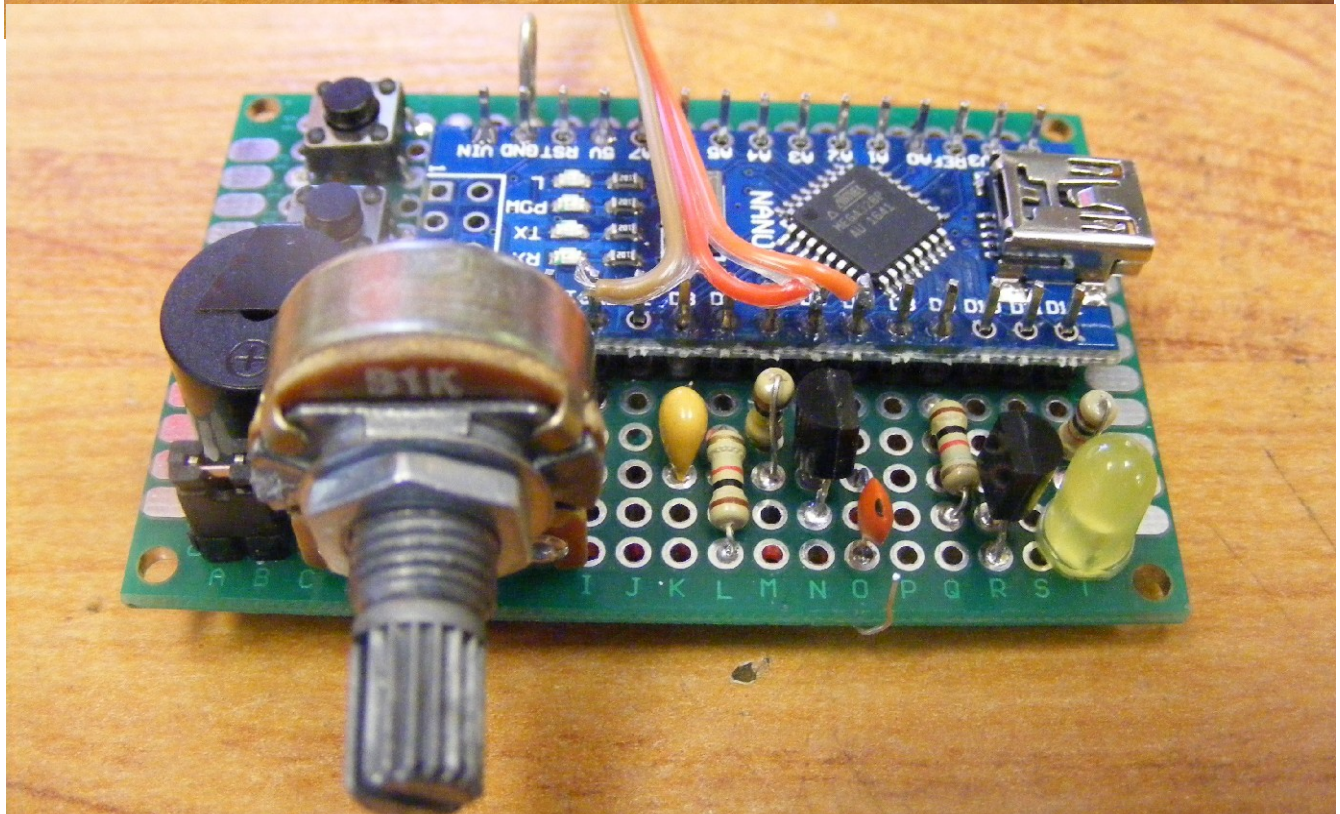
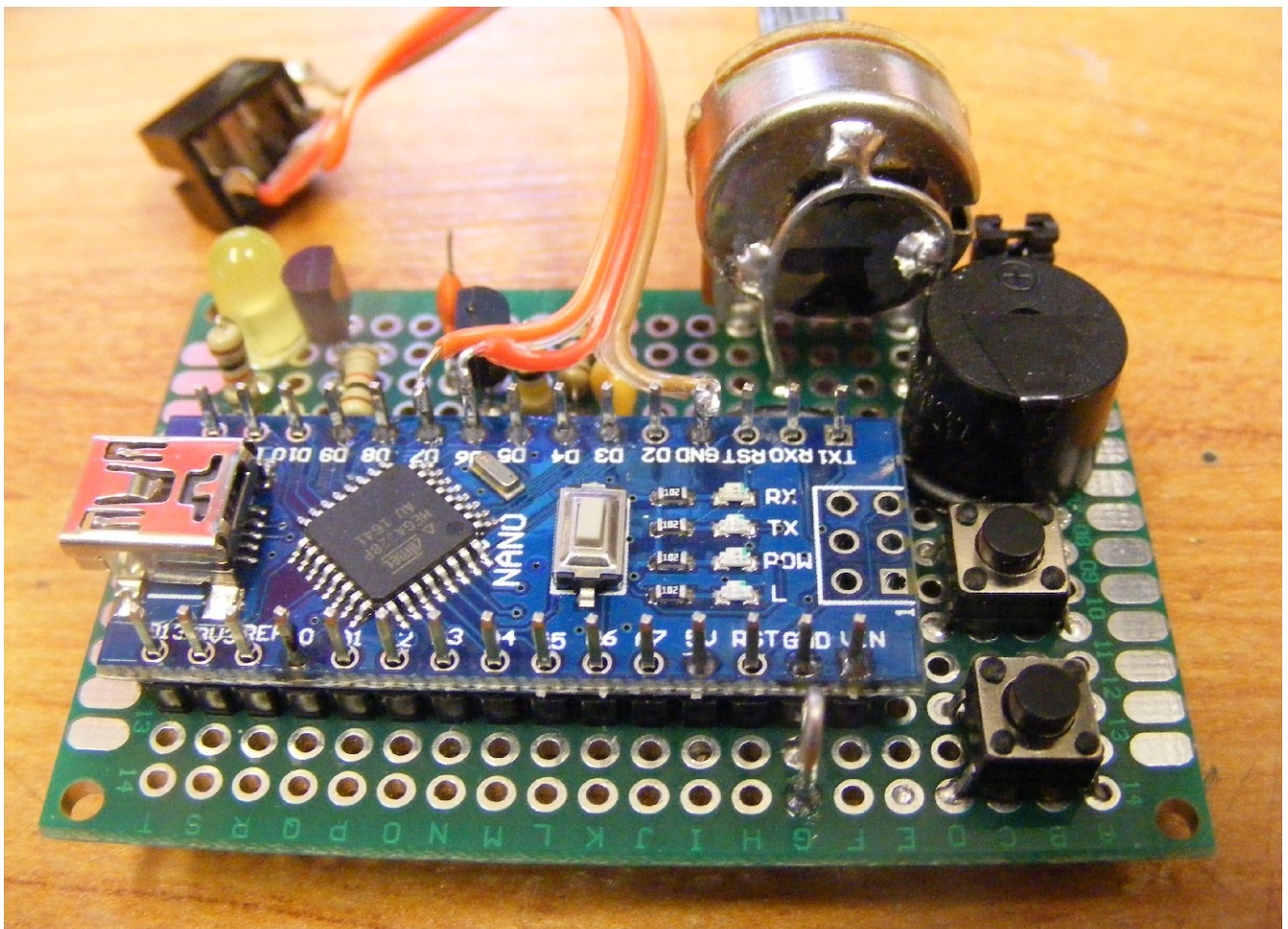
Giving the user access to the program itself allows some simplification in designing the program. What might otherwise need implemented in a set of menus (and requiring a menu button) can be done with the user changing lines of code (like constants) in the program. Entering the particular message text can be done in the program code as well.

Many construction methods can be used to build the hardware. Minimally only three components need added to the previous version of this project: a stereo jack into which to plug paddles and a resistor and transistor to drive the key input of the transmitter. These could be easily added to the previous build if desired.



A solderless breadboard could also be used. My choice this time was to use a perfboard with plated through holes.





The resulting board shown includes many options and makes testing easy before mounting in a rig. That mounting could be accomplished using the shaft of the potentiometer or the corner mounting holes.

Let's go on an tour of various things visible in the pictures above. Not shown are off board connections. For testing purposes the USB connector on the Arduino Nano can supply power from the computer to which it is connected or a little USB charging cube would work as well. The Nano is mounted to the perfboard using header strips with the longer side pointing up so connections can be made to various pins on the processor board (the paddle stereo jack for example). 12 volt power can be connected to the VIN and ground pins on the processor board.

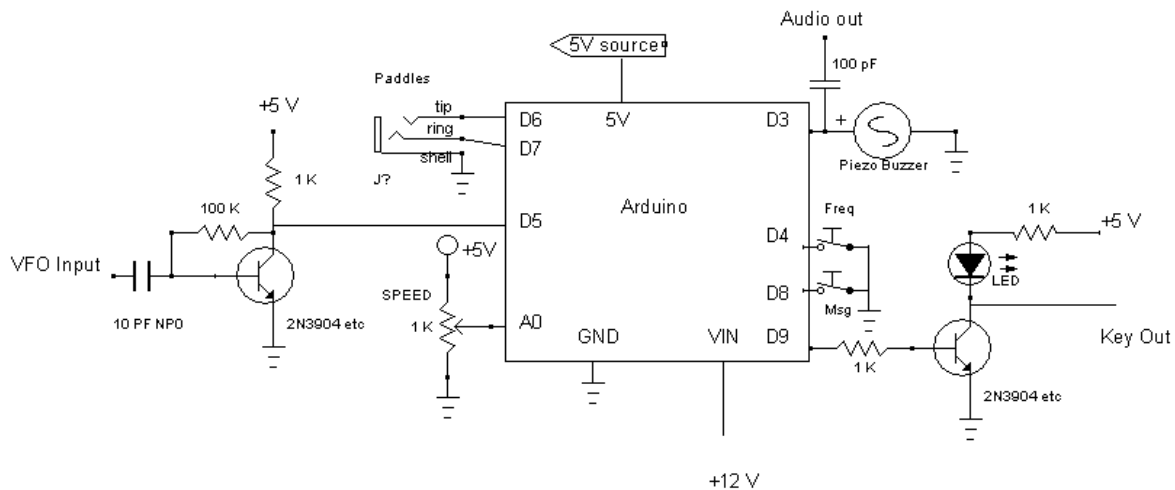
Wiring was done on the bottom of the board using solid hook up wire as well as some wire wrap wire. The speed potentiometer is anchored to the board using a U shaped loop of 20 gauge solid wire soldered on the back of the pot. Its important here to cleaning any coating off the metal so it can be soldered, I used a Dremel tool.

Again mostly for testing an LED is included that lights as the transmitter output is keyed. Its the yellow LED near a corner. At the other end of the board a piezo buzzer can be seen. Next to it is a pair of header pins with a jumper block in place. This is used to listen to the sidetone without having to connect the board to a radio. The jumper allows tuning off that buzzer. Note also the use of a bit of electrical tape over part of the hole in the buzzer used as a volume control.

On the edge of the board across from the potentiometer a loop of bus wire is included which is connected to board ground. This serves as a convenient ground connection for test leads.

The two push buttons on the board are also there for testing. One is pressed to read out the frequency in CW as in the previous version of this project. The second button is either pressed or long pressed to send the stored message. The long press sends it repeatedly until a paddle is pressed. The buttons are normally open to ground, allowing them to be left on the board with connections made of off board buttons mounted in the radio case.

Counter input from the VFO is connected to the orange capacitor lead sticking up at the edge of the board. The yellow capacitor provides sidetone audio output that connects to the audio chain of the receiver if the piezo buzzer is not used. Transmitter keying is found at the edge of the board near the LED at the holes labeled R and S on the board (they are jumpered together on the bottom side).



The schematic for this project is still fairly simple. The only necessary additional components compared to the previous project are just the stereo jack for the paddles and the 1 K resistor and transistor for keyer output. Transistors can be many common NPN parts including 2N2222 for example. I used an Arduino Nano board but an Arduino UNO will work as well. Not shown is the USB connector on the processor board used for programming. The 100 pF capacitor for audio output can be changed to adjust audio level. Connection points to the NC40A board are discussed in the original frequency only project included below.

The optional speed potentiometer can be any value from 1K to 10K. If the 10 pF capacitor needs changed be sure to use an NP0 capacitor and keep it as small as possible consistent with reliable frequency readout.

A note on the output keying circuit is appropriate. The keying of most radios these days grounds a 12 volt or so input with low current (a few mA). Most transmitters using tubes do not meet this requirement and a different keying arrangement will be required. Some QRP transmitters key a significant amount of current and/or don't key a low voltage to ground. Problems can arise here too.

The NC40A keying circuit potentially could be troublesome as it includes the emitter circuit of the driver transistor Q6. Using a hand key one expects a pretty close to zero ohm closure. With a keyer there are two possible issues. One is the current level that must be keyed. The other is how the resistance of the keyer at keydown will possibly change biasing of the driver transistor. Experimentation indicates that neither of these issues is an actual problem with this keyer and the NC40A in terms of power output.

Program details on the frequency measuring functionality are described in the previous project. Constant declarations for the keyer addition include these:

```
#define SPEED          0      // if 0 then use speed pot

#define SIDETONE      true

#define MIN_SPEED     10

#define MAX_SPEED     25
```

Previously SPEED specified the fixed speed value at which CW would be generated. Now this value can be set to 0 which enables the potentiometer to control the speed. If the potentiometer is not used a fixed speed can still be used. If the potentiometer is used, MIN_SPEED and MAX_SPEED control the range of speeds that will be used as the names suggest.

The SIDETONE specifies if a sidetone will be output by the keyer. For testing the “true” value that is the given default is useful. It is also useful for radios that do not themselves have sidetone provisions. When set to “false” only the frequency announcement will generate sidetone output.

The default stored message in the program below is this:

```
char msg[]=
    "-.-. --.- -... ." // CQ DE
    "-.-. .... -.- -.- " // put your call here
    "-.-";              // K
```

It takes the form of a string of characters that include ., - and spaces. Yes you have to spell out the dits and dahs to be used for the message yourself. Although it would not be that hard to have added the capability to enter the message text in regular ASCII characters and let the program translate to the appropriate dits and dahs, this is a bit simpler. Entering the text with the paddles could also have been incorporated, again at additional complexity and requiring a menu system or additional buttons. Storing a message entered with the paddles could be done in EEPROM which is found in the Arduino UNO but not the Nano. With the source program available for easy modification the approach used here is reasonable.

The statements that determine if the msg button is tapped or long pressed are these:

```
if (digitalRead(MSG)==LOW)
{
    while (digitalRead(MSG)==LOW) ; // wait button release
    sendMsg((millis()-startTime)>250);
}
```

A tap results in a parameter of false being passed to function sendMessage or true if long pressed. Function sendMessage uses that to determine if it should repeat the message. The constant 250 is the minimum time in mS for a long press. When the message is to be repeated, a paddle press terminates that repetition. It also can abort sending the complete

message as well. This is handled in the sendMessage function with calls to paddlePressed function. As written here more than one tap of a paddle might be needed to be seen by the program. This could be improved at the cost of more program complexity.

Sidetone generation makes use of library tone() and noTone() functions. As written it is usable with either a piezo buzzer or sending audio into the audio chain of the receiver. If just using a piezo buzzer and a bad sounding sound results, try replacing the tone and noTone calls with digitalWrite(BUZZER,HIGH) and digitalWrite(BUZZER,LOW) respectively.

The program is intended to be useful as it is and also be a starting point for those that might want to add or change functionality. For example it would not be difficult to have two (or more) messages that could be sent. Additional buttons could be used, or the distinction between tap and long press of the message button could be used to select between two message if the repeat function were not needed.

An LCD display could also be added but timing issues could make that very tricky. An I2C hardware interface would probably be required since there aren't a lot of data pins available. Code examples of formatting frequency are included in the debug output from the frequency counting portion of the program.

With an LCD display one might want to continuously display frequency and CW speed. This could give the timing problems. When measuring frequency we really don't want to deal with running the keyer which would affect accuracy and/or greatly increase code complexity. If the frequency display were only updated when the frequency button is pressed there would not be the timing problem but the resulting display may not be so desirable. The program implementation as it is here is a compromise between usefulness and code complexity.

A more reasonable approach would be to use a single Arduino processor to both provide a keyer and control a DDS replacing the free running VFO. Then we don't need to measure frequency but will rather accurately control it. Design of such a program should start from scratch rather than just modifying the one included here. Useful concepts from this one could help with the coding. Of course there would be many hardware design considerations in using a DDS so as to put out a clean signal from the transmitter.

The original project description is included after the program listing.

```

/*
NC40A frequency in CW
and a simple memory keyer
Mike WA8BXN Jan 2018
*/

#define KEYOUT          9
#define MSG             8
#define DAH_PADDLE     7
#define DIT_PADDLE     6
#define FREQ_IN        5          //Do NOT change!
#define BUTTON         4
#define BUZZER         3
#define SPEED_POT      A0

#define SPEED           0          // if 0 then use speed pot
#define ditTIME        (1200/wpm)
#define IF_OFFSET      49150
#define CPU_FREQ_ADJUST 9943
#define SIDETONE       true
#define MIN_SPEED      10
#define MAX_SPEED      25

volatile unsigned int overflowCounter = 0;
unsigned int frequency = 0;
int          wpm=SPEED;

char *code[10]=
    {"-----", ".-----", "..---", "...--", "....-",
     ".....", "-.....", "--...", "---..", "----."};

char msg[]=
    "-.-. ---. -.. ." // CQ DE
    "-... ..-. --- -.- " // put your call here
    "-.-"; // K

// Timer 1 is our counter
// 16-bit counter overflows after 65536 counts
// overflowCounter will keep track of how many times we overflow
ISR(TIMER1_OVF_vect)
{
overflowCounter++;
}

void setup()
{
// Timer 1 will be setup as a counter
// Maximum frequency is Fclk_io/2
// (recommended to be < Fclk_io/2.5)

```

```

// Fclk_io is 16MHz
TCCR1A = 0;
// External clock source on D5, trigger on rising edge:
TCCR1B = (1<<CS12) | (1<<CS11) | (1<<CS10);
// Enable overflow interrupt
// Will jump into ISR(TIMER1_OVF_vect) when overflowed:
TIMSK1 = (1<<TOIE1);

Serial.begin(9600);
Serial.println("Begin AFA");
pinMode(FREQ_IN, INPUT);
pinMode(BUTTON, INPUT_PULLUP);
pinMode(BUZZER, OUTPUT);
pinMode(DIT_PADDLE, INPUT_PULLUP);
pinMode(DAH_PADDLE, INPUT_PULLUP);
pinMode(MSG, INPUT_PULLUP);
pinMode(KEYOUT, OUTPUT);
setSpeed();
OK();
}

void loop()
{
long startTime=millis();
setSpeed();
if (digitalRead(BUTTON)==LOW)
    {
    freqCount();
    while (digitalRead(BUTTON)==LOW) ;
    }
if (digitalRead(MSG)==LOW)
    {
    while (digitalRead(MSG)==LOW) ; // wait button release
    sendMsg((millis()-startTime)>250);
    }
if (digitalRead(DIT_PADDLE)==LOW) ditKey();
if (digitalRead(DAH_PADDLE)==LOW) dahKey();
}

void freqCount()
{
char buf[20];
// Delay 10 mS. While we're delaying Counter 1 is still
// reading the input on D5, and also keeping track of how
// many times it's overflowed
TCNT1=0;
overflowCounter = 0;
delayMicroseconds(CPU_FREQ_ADJUST);
frequency = TCNT1 + 65536 * (unsigned long) overflowCounter;
}

```

```
sprintf(buf,"%d %5d.%d kHz",frequency,frequency/10,frequency%10);
Serial.println(buf);
sendFreq();
}
```

```
void ditKey()          // key transmitter, optional sidetone
{
if (SIDETONE) tone(BUZZER,600);
digitalWrite(KEYOUT,HIGH);
delay(ditTIME);
noTone(BUZZER);
digitalWrite(KEYOUT,LOW);
delay(ditTIME);
}
```

```
void ditTone()        // sidetone always, don't key transmitter
{
tone(BUZZER,600);
delay(ditTIME);
noTone(BUZZER);
delay(ditTIME);
}
```

```
void dahKey()
{
if (SIDETONE) tone(BUZZER,600);
digitalWrite(KEYOUT,HIGH);
delay(3*ditTIME);
noTone(BUZZER);
digitalWrite(KEYOUT,LOW);
delay(ditTIME);
}
```

```
void dahTone()
{
tone(BUZZER,600);
delay(3*ditTIME);
noTone(BUZZER);
delay(ditTIME);
}
```

```
void OK()            // Startup message
{
dahTone();dahTone();dahTone(); delay(3*ditTIME);
dahTone();ditTone();dahTone();  delay(3*ditTIME);
}
```

```
void sendDigit(int n) // single digit to CW
```

```

{
char *p=code[n];
while (*p)
    if (*p++=='.') ditTone(); else dahTone();
delay(ditTIME);
}

void sendFreq()          // send freq in CW sidetone only
{
long f=frequency+IF_OFFSET;
Serial.println(f);
int n=0,tenths;
int digits[10];
tenths=f%10;
f=f/10;
while (f)
    {
    digits[n++]=f%10;
    f=f/10;
    }
n--;
while (n>=0)
    {
    sendDigit( digits[n--]);
    }
/*    uncomment to send tenths of kHz
ditTone();dahTone();ditTone();delay(3*ditTIME); // "R"
sendDigit(tenths);
*/
}

void sendMsg(bool repeat) // stored message
{
char *p;
long delayOver;
do
    // loop for repeating msg
    {
    p=msg;
    while (*p)          // for each character in msg
        {
        if ( paddlePressed())
            return;
        switch (*p++) {
            case '-': dahKey(); break;
            case '.': ditKey(); break;
            default: delay(3*ditTIME);
        }
        }
    }
}

```

```

if (repeat)          // delay between sneding msg
{
  delayOver=millis()+5000;  // delay in mS
  while (delayOver>millis())
    if (paddlePressed()) return;
}
} while (repeat);    // repeat until paddle pressed
}

bool paddlePressed()
{
return (digitalRead(DIT_PADDLE)==LOW) ||
        (digitalRead(DAH_PADDLE)==LOW) ;
}

void setSpeed()      // use pot to set speed
{
int potVal;
if (SPEED!=0) return;    // no pot, constant speed
potVal=analogRead(SPEED_POT);
wpm= MIN_SPEED+((MAX_SPEED-MIN_SPEED+1)*potVal/1023);
}

```

Arduino "Freq-Mite" for Norcal NC40A

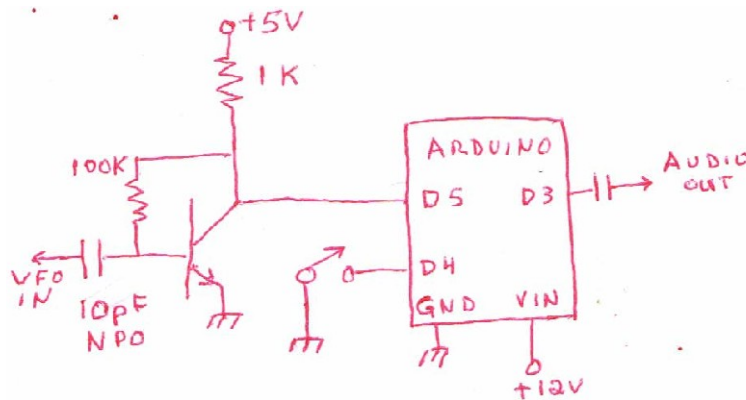
Mike WA8BXN

Jan 2018

Dave Benson's (K1SWL) Freq-Mite is a popular frequency counter used as a digital readout in CW of the operating frequency of QRP transceivers. No longer produced by Dave himself it is currently available from <http://www.4sqr.com/freq-mite.php>

I present here similar functionality using an Arduino processor intended primarily for use with the Norcal NC40A QRP transceiver. Within its limitations noted below it could be used otherwise. Besides being a useful device it should serve as an example of how the functionality might be implemented using an Arduino.

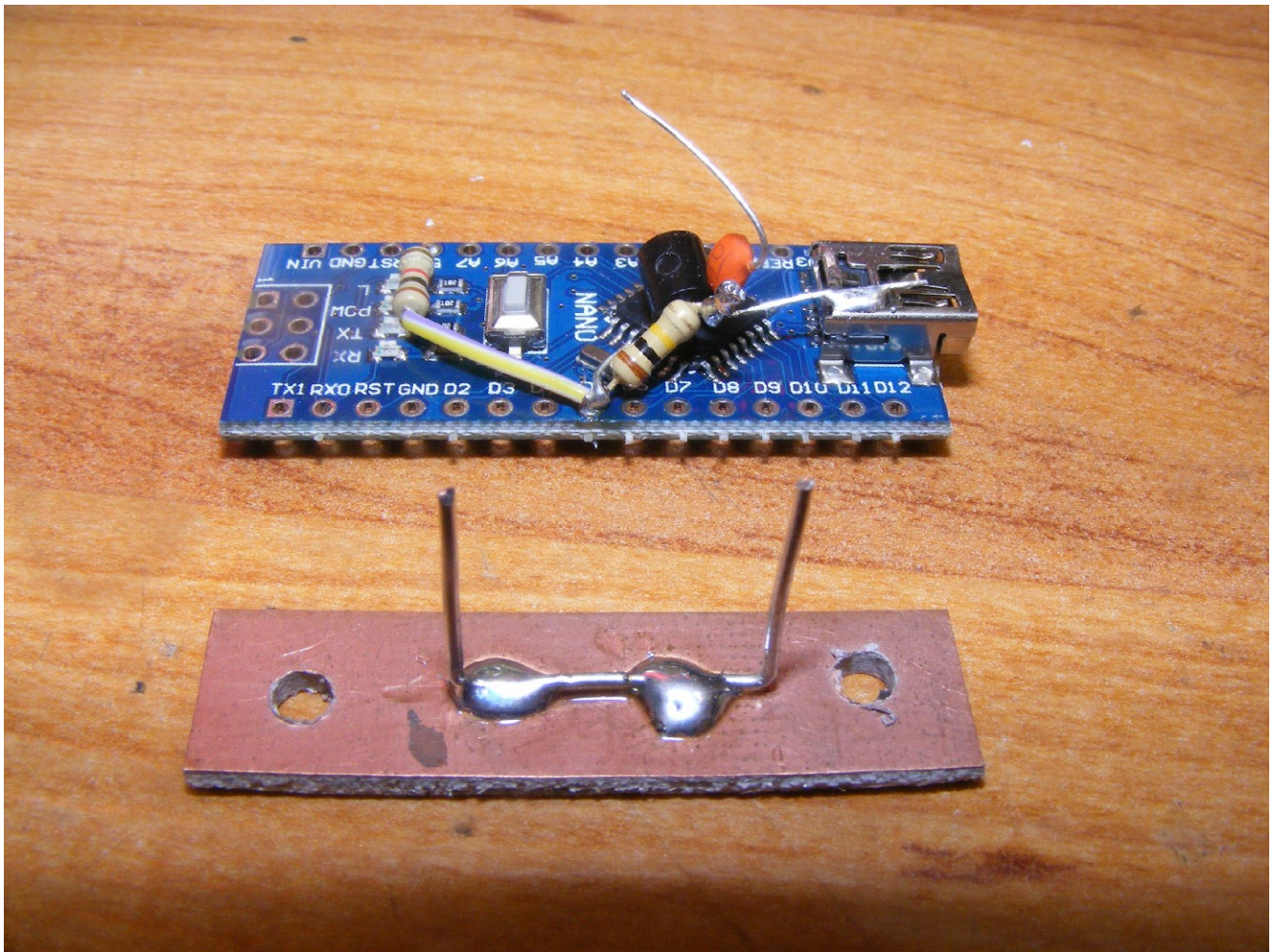
The hardware components are quite simple, shown in the schematic below:



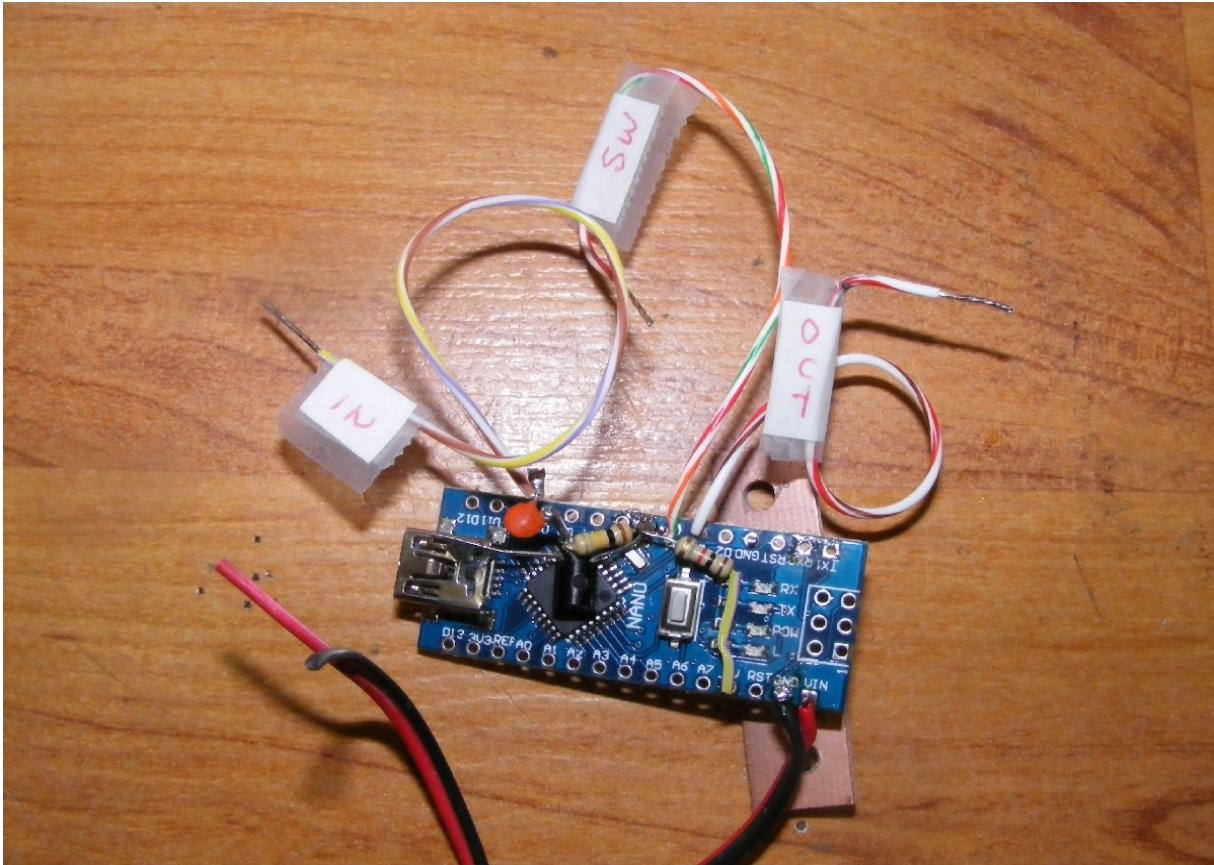
The switch is a push button that when pressed announces the frequency in CW on the audio output. The rest of the circuit on the left is a simple preamp that allows light coupling to the VFO. I used a 2N3904 transistor, many others (such as a 2N2222) should work as well. I used an Arduino Nano processor board, an Arduino UNO board will work as well but is of course larger.

+5 volts is obtained from the Arduino board itself, which is powered by the supply to the NC40A which I indicate as +12 V. VFO input is obtained at the junction of R23 and C7 on the NC40A board and audio output is connected to the R7 and U3 pin 2 junction. For the capacitor shown above connected to D3 of the Arduino I used 100 pF. This value can be adjusted to give desired audio level. The 10 pF capacitor going to the VFO from the transistor could also be changed as needed. Keep it as small as possible to limit its effect on the VFO itself. It should be a NP0 so as to not introduce frequency drift in the VFO.

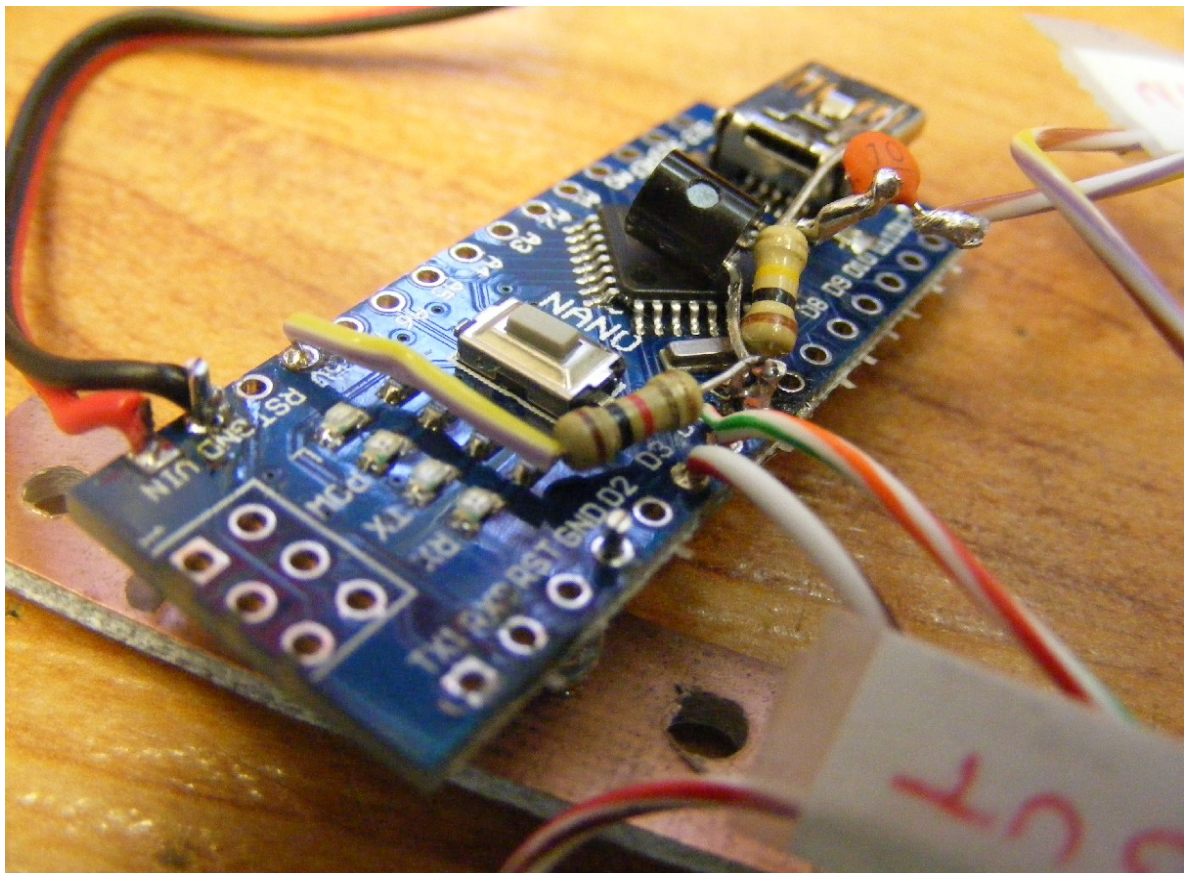
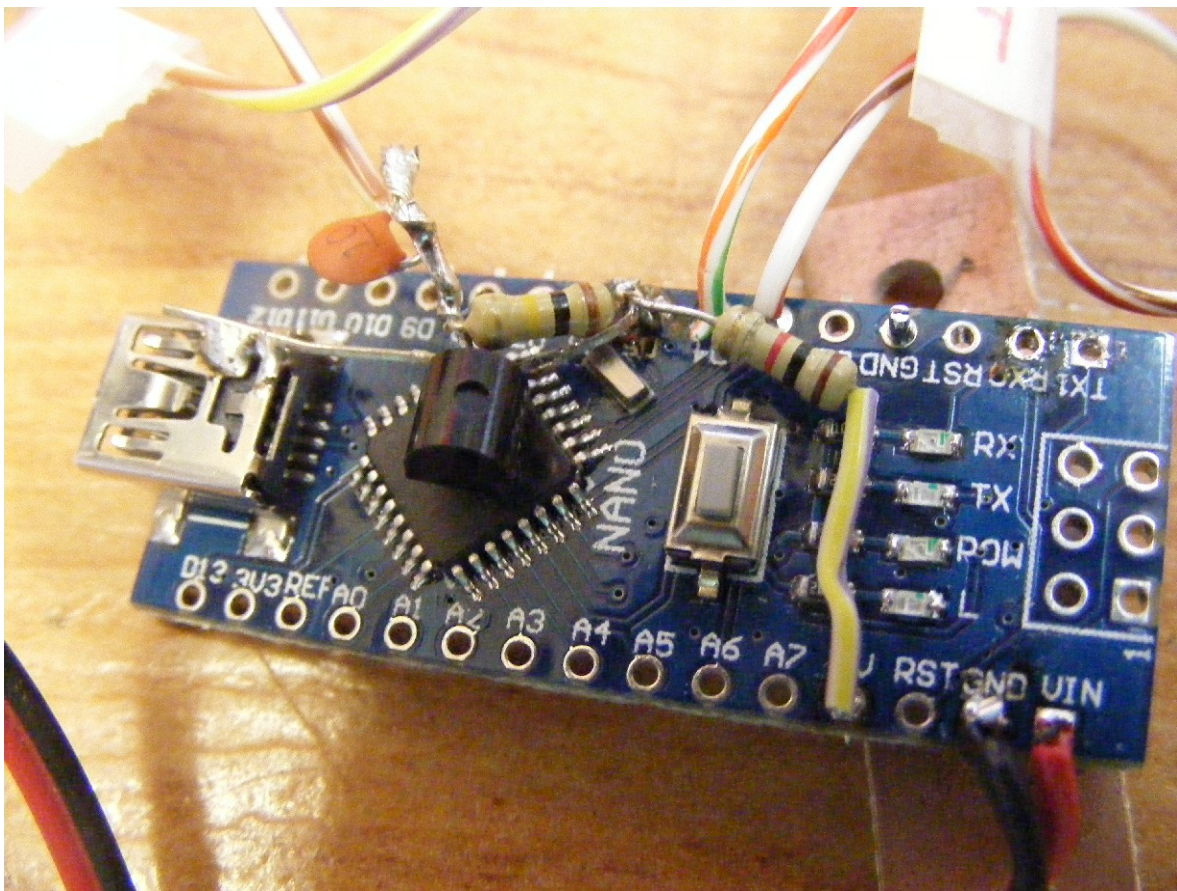
If desired a piezo buzzer could be connected directly to D3 instead of feeding audio into the NC40A for the CW frequency announcement.



Many Arduino Nano boards don't provide much in the way of mounting the board itself. I used a scrap of circuit board and a U shaped piece of wire soldered to it for mounting. The ends of the wire were soldered to the two ground pins on the Nano board. Other components were soldered to the Nano board with the shell of the USB connector used as a ground point for the emitter of the transistor. Other construction methods could be used as well.



Above is the complete board. The output capacitor connected to D3 is not shown as it was connected to the NC40A board. Below are a couple other close ups of construction.



The original Freq-Mite had jumpers to select the IF offset frequency. Since the source code for the program is provided here that offset value is simply specified in one of the lines of code. Similarly adjustment of the CW speed is possible by changing a line of code instead of pressing the button at power up.

Before looking at the program below some general background information is necessary for its understanding. Basically the program functions as a frequency counter that measures the frequency of the VFO in the NC40A, adds the IF offset frequency (around 4.915 MHz) and then generates CW for the operating frequency. When the button is pressed this happens and the frequency in KHz is heard.

To measure frequency one could watch the input signal in the program and count the number of cycles that occur in a given period of time. Testing the input to see when each cycle of input begins in code would greatly limit the frequency limit of what we could count. This would probably limit us to audio frequencies not the around 2 MHz at which the VFO operates.

The trick used that allows us to count frequencies in the MHz range makes use of the fact that most of the processor chips are designed with a counter in the chip that can be used to count cycles in hardware without executing program instructions to recognize each cycle and do the counting. Access to this is not apparent in the normal description of programming in the Arduino environment but does exist and is documented somewhere.

The hardware counter in the processor on the Arduino board is only 16 bits meaning it can only count up to a value of 65535. Beyond that value it just start over counting at zero again. Fortunately an interrupt is generated when the counter overflows and we can in our program count the number of times that occurs. From this we can compute larger counts. If you don't fully understand this detail its OK.

Now our basic approach to frequency counting will be to use that hardware counter to count input cycles for us for some period of time. If we count for one second the resulting count would be the frequency in Hz. We don't need that much accuracy nor do we want to wait a whole second before hearing the frequency so the program only counts for 10 mS. The resulting value we will get will be in terms of hundreds of Hz. By default the program will only report the frequency in KHz. Un-commenting a few lines at the end of the program will allow it to report the frequency including tenths of KHz.

Accuracy of the frequency we get from doing the cycle counting will depend on how accurately we can determine the period of time we do the counting. The Arduino processor board does have a crystal controlled timing oscillator running at 16 MHz. However there is no trimmer to adjust to get it exactly on frequency. As a result our timing should be fairly stable but not absolutely accurate. Provision is made in the program to compensate for that.

Another issue is the value for the IF frequency to add to our frequency count. The crystals used in the filter are nominally 4.915 MHz but where the center of the passband we get is a bit different. Another variable to this is the adjustment of trimmers C34 and C17. A constant in the program can be adjusted for these issues.

The program below can be used as is most likely. Comments about modifications are included following the listing. Note that if you do use a Nano processor and have never done

so before you are likely going to need to install drivers found at <http://bit.ly/2pMF4in>
If you use an Arduino UNO board needed drivers are included with installation of the Arduino programming environment (IDE).

Installation of the Arduino IDE, programming in C and related topics are of course beyond the scope of this discussion but well covered elsewhere.

```

/*
Arduino frequency counter for Norcal NC-40a with CW output
Can be used with VFOs up to around 5 MHz
Runs on Nano or UNO processor boards

Mike WA8BXN Jan 2018
*/

// Port definitions
#define FREQ_IN 5           // Do NOT change this
#define BUTTON 4
#define BUZZER 3

#define SPEED 15
#define ditTIME 1200/SPEED
#define IF_OFFSET 49155
#define CPU_FREQ_ADJ 9951

volatile unsigned int overflowCounter;
unsigned int frequency;

char *code[10]=
  {"-----", ".----", "..---", "...--", "....-",
  ".....", "-....", "--...", "---..", "----."};

// Timer 1 is our counter
// 16-bit counter overflows after 65536 counts
// overflowCounter will keep track of how many times we overflow
ISR(TIMER1_OVF_vect) // Interrupt handler
{
overflowCounter++;
}

void setup()
{
// Timer 1 will be setup as a counter
// Maximum frequency is Fclk_io/2
// (recommended to be < Fclk_io/2.5)
// Fclk_io is 16MHz
TCCR1A = 0;
// External clock source on D5, trigger on rising edge:
TCCR1B = (1<<CS12) | (1<<CS11) | (1<<CS10);
// Enable overflow interrupt
// Will jump into ISR(TIMER1_OVF_vect) when overflowed:
TIMSK1 = (1<<TOIE1);

pinMode(FREQ_IN, INPUT); // This is the frequency input
pinMode(BUTTON, INPUT_PULLUP);
pinMode(BUZZER, OUTPUT);
Serial.begin(9600); // Console debug output
Serial.println("Begin AFA");
OK(); // Startup msg in CW
}

void loop()
{
if (digitalRead(BUTTON)==LOW)
freqCount();
}

```

```

void freqCount ()
{
// Delay 10 mS. While we're delaying Counter 1 is still
// reading the input on D5, and also keeping track of how
// many times it's overflowed
char buf[20];
TCNT1=0;
overflowCounter = 0;
delayMicroseconds (CPU_FREQ_ADJ);
frequency = TCNT1+65536* (unsigned long) overflowCounter; // freq in 100s of Hz
sprintf (buf, "VFO %5d.%d kHz", frequency/10, frequency%10);
Serial.println (buf); // Console debug output
sendFreq ();
}

void dit ()
{
tone (BUZZER, 600);
delay (ditTIME);
noTone (BUZZER);
delay (ditTIME);
}

void dah ()
{
tone (BUZZER, 600);
delay (3*ditTIME);
noTone (BUZZER);
delay (ditTIME);
}

void OK () // send power up "OK"
{
dah ();dah ();dah (); delay (3*ditTIME); dah ();dit ();dah (); delay (3*ditTIME);
}

void sendDigit (int n)
{
char *p=code[n]; // string for this digit
while (*p) // for each character in string
if (*p++=='.') dit (); else dah (); // send dit or dah
delay (2*ditTIME); // pause between digits
}

void sendFreq () // send frequency in CW to audio
{
long f;
int n=0, tenths;
int digits[10];

f=frequency+IF_OFFSET;
Serial.print ("Mixed freq "); Serial.println (f); // Debug output
tenths=f%10;
f=f/10;
while (f) // get individual digits right to left
{
digits [n++]=f%10;
}
}

```

```
        f=f/10;
    }
n--;
while (n>=0)                // send digits left to right
    {
    sendDigit( digits[n--]);
    }
/*                          uncomment for tenths of KHz
dit();dah();dit();delay(3*ditTIME); // send letter R for decimal place
sendDigit(tenths);
*/
}
```

For those that want to fine tune the program, there are 3 lines of code that are easy to change. They are found near the beginning of the listing:

```
#define SPEED          15
#define IF_OFFSET      49155
#define CPU_FREQ_ADJ  9951
```

The value for speed is the simplest. Change 15 to the CW speed you prefer, in WPM.

The other two require experimentation to determine the best values. The following procedure should be used.

The value CPU_FREQ_ADJ allows us to compensate for variations in the 16 MHz CPU clock frequency. Connect a known frequency input to the circuit (1 MHz would be a good value). Start the Arduino IDE and connect the processor board to the computer with a USB cable to get set up for programming the board. Load the source program into the IDE if its not already there.

In the programming IDE, open the debugging monitor window. Press the button to read frequency. In the debug monitor window you will see something like this:

```
Begin AFA
VFO 2123.7 kHz
Mixed freq 70392
```

I just used the VFO rather than a 1 MHz input. With 1 MHz you want to get it to show VFO 1000.0 kHz or what ever your known frequency input actually is. It must be less than around 5 MHz!

Change the CPU_FREQ_ADJ value and recompile and upload the program and run again until you get as close as you can to the correct display.

To adjust the number for IF_OFFSET connect the input to the VFO as would be used for normal operation. Tune the VFO to a know frequency using some independent standard. You could measure the transmit frequency of the NC40A or tune the receiver to hear a signal at a know frequency. Press the button and note the value displayed in the debug monitor window for Mixed freq. Change the number for IF_OFFSET to get it to display the right Mixed freq when the button is pushed.

Concluding remarks

I hope you find this project description useful. I think its fairly easy to actually build and get working, particularly if you have any Arduino experience already. You may have noticed that most of the time the program just sits waiting for that button to be pressed. I heard it begging for more to do and with a few more hardware parts and some more lines of code added a simple iambic keyer with a stored message as well but that is a story for next time!