

An Inexpensive Arduino-Based Morse CW Keyer

John Silzel, N6HCN

December 8, 2020



Abstract

No doubt dozens of hams have homebrewed keyers using the popular Arduino microcontroller boards. For those who are interested in such endeavors, this article describes a design built by N6HCN for under \$30 USD. The goal here is not to encourage reproduction the author's unit, but to jump start other hams interested in building similar projects by providing design suggestions, recommending parts and learning materials, and giving programming tips for implementing increasingly deluxe features, including memory and a Morse keyboard function.

1 Does the World Need Another Homebrew Keyer?

In a world of cheap, reliable ham gear, available off the shelf, why homebrew? And if one does homebrew, why spend hours designing and debugging a device that someone else has already made available in kit form, perhaps even 90% assembled? Why restore a classic automobile? Why refinish a nice old coffee table? We are meant to work with our hands, and the pleasure of building something that you, or others, can really use and enjoy is part of the reward of craftsmanship. Life is better if you have a project going.

Another reason to homebrew is that you can build an excellent keyer. You can include only the features you want. You can build something that is brutally simple, or you can build a real monster. For the price of a hamburger, you can add gigabytes of memory, enough to send "War and Peace" for months of code practice when the bands are closed. You can provide inputs for multiple paddles, your entire bug collection, and that straight key on your shelf, and switch from key to key at will. Connecting a mechanical key to a micro-processor lets you use firmware tricks to tame troublesome contacts. Since most of a keyer's brain amounts to its programming, if a new idea comes to you, in many cases you'll be able to implement it without soldering.

Table 1: Digital devices and related parts used in the N6HCN keyer. All were sourced online from All Electronics, but these parts are widely available. The author finds that Arduino Uno clones such as SeeDuino are equivalent in performance.

All Electronics Cat No	Description	Price
ARD-21	Arduino Uno R3	\$14.00
ST-1113	16x2 LCD Display W/ Keypad	\$6.00
HDR-07	Arduino Stackable Header Kit	\$1.50
SDR-7	Micro SD Card Reader W/ Header	\$2.85
TOTAL (\$US,2020)		\$24.35

In this article I will help you get started and supply some basic ideas. The rest is up to you, your needs, and your imagination. I'll start by giving you a list of simple, widely available parts, and encourage you to get started. Along the way you'll need to learn some simple C programming, but since we'll be using the Arduino microcontrollers, you'll find that good examples and how-tos abound. You'll be sending CQ in no time.

2 A Keyer is Born

I was shopping for Christmas gifts in December, 2019, unaware, of course, that COVID-19 would make 2020 a very strange year, indeed. My daughter Faith was a college sophomore, fond of building electronic kits, so I was looking for a project for her on the All Electronics website. I wound up buying her an Arduino "Inventor Kit", the kind with motors and LEDs and sensors enough to keep anyone busy on a rainy day. As long as I was shopping, I decided to get myself a "stocking stuffer" as well, as I'd been thinking about building a keyer to go with "The Wren", a 10W dual band transceiver I had just homebrewed for myself. The idea of an entirely homebuilt station (key, keyer, rig, and antenna) was appealing.

Some projects are meant to be. In ten minutes I had found nearly everything (Table 1) needed for the digital part of the project. All of these items can be found elsewhere. I just found it encouraging that All Electronics had these items all in one place and at prices that didn't trigger a conscience attack for blowing \$25 on myself when I was supposed to be shopping for others. The SD card reader was a whim. For under \$3 I figured I could do something with it. After all, the fancy keyers on the market all have memory, and our household seems to have lots of old SD cards lying around. Nowadays a 4Gb card is too small to use in a camera, but overkill for even the fanciest keyer.

Although I had not started designing anything, and this parts list was pure impulse buying, these parts turned out to be pretty good choices. Let's walk down the list so you'll know what to look for and what options you might choose for your own keyer design.

2.1 Why Arduino?

The Arduino Uno is the keyer's brain. Nothing more than an Italian-made demo board for a commercial microcontroller IC, the Arduino family of products and their knockoffs have been developed for the so-called maker community (which seems to be the new term for what we used to call homebrewers, students, and general geeks). Lots of leading keyers like the one by W7QC use PIC microcontroller ICs instead of the Arduino, and I would too if I were designing a commercial product. That's because professional PICs themselves are dirt cheap, feature a wide range of capabilities, and have very sophisticated interactive development environments (IDEs) that feature simulators and debugging tools. But they also require buying a program-

mer device, cannot be reprogrammed in-circuit, and can present a steep learning curve to newcomers. For simplicity and low cost, it's hard to beat the Arduino, which features easy on-board programming, serial USB, on-board voltage regulation, a free IDE, and vast amount of learning material on the web, much of it aimed at non-technical types or students.

Businesses have flooded the internet with dozens of circuit boards that make expansion of an Arduino device easy. Take the second item on my list, for example. Google "LCD Display with Keypad" and you will find this board available from many places. Mine was made by DFRobot (<http://dfrobot.com>), whose website features extensive documentation, dimensional drawings, sample code, and example projects. The Arduino is dimensionally and pin-pin compatible with these *shield boards* by means of a standard stackable header interface, making it a no-brainer to use the Arduino if the goal is building yourself a custom keyer while learning digital project skills.

Out of the Arduino lineup, I picked the Uno because it is widely available, and dimensionally compatible with the LCD display. If you're a bargain hunter, you will find knockoffs of the Uno for under \$10 that are, in my experience, just as good as the "genuino". And if your knockoff ever fails, replacing it is a no-solder affair: just plug a new one into your keyer, reload your program, and you're back on the air. If you're a SOTA hiker counting grams, or trying to fit a keyer into an existing rig, check out the Arduino Nano, which is tiny and might make for a very compact, but Spartan, custom keyer.

3 From Arduino to Keyer

The parts above will give you the brain of a keyer, and an LCD display and buttons to let you interact with it. What remains is to enable the Arduino to key your transmitter, produce audible sidetones so you can hear your sending, and let you connect keys or paddles. To do all this, you will program the Arduino to control output pins or read input pins. The Arduino's pins are numbered (Figure 1), and in many cases a given pin may be defined as either an input or an output, depending on what you have connected to it.

An easy way to manage the connection of pins to the outside world is by building your own keyer shield board. A "shield board" is the Arduino term for a printed circuit board (PCB) that has pin headers that align with the Arduino main board. These headers feature sockets on one side of the board, and pins on the other, so that shield boards may be stacked vertically above the Arduino in any number necessary to provide the needed functions. The LCD keypad has to be on the top of our stack, so that the display and buttons will be accessible. The keyer shield will be sandwiched between the LCD shield on the top and the Arduino on the bottom (Figure 6). You'll have to pay attention to which numbered pins are used by the LCD and keypad, and which pins are available to your keyer shield board to work CW magic.

You can build a keyer shield board without having to do your own PCB art and such. Google "Arduino Proto Shield" and you will find pre-drilled perfboards of exactly the correct size, complete with the pin headers, ready for you to solder on the necessary parts. In my own homebrewing, I like to use the free Eagle PCB software to draw schematics and create PCB art. I have used professional PCB fabricators, but I can produce a finished single-sided PCB in about 2 hours using the "Toner Transfer Method". But that's a story for another day.

Whether a proto-board or PCB, your Keyer Shield board will feature Arduino through-headers positioned to plug into the Arduino on the bottom and the LCD Shield on the top. The Keyer Shield will also have connectors for key inputs (two wires for a bug or straight key, three for iambic paddles), and two-wire outputs for things like headphones, or a speaker or piezo buzzer so you can hear your CW. The Keyer Shield board can be simple, consisting of just the parts necessary to key a transmitter, or it can be elaborate. You can stack up more shields to add ethernet, bluetooth, and such. I'll discuss my own Keyer Shield next, but I hope you will design your own for the satisfaction of doing so.

Table 2: *Parts list for the keyer shield board, part designators refer to the schematic in Appendix I. Indicated part numbers are available from All Electronics (AE). The others may be sourced easily from Jameco, Marlon P Jones, DigiKey, Mouser, etc. Prices are author's estimates.*

P/N (Vendor)	PCB Ref	Description	Price
	R1,R2	470 Ω Resistor	\$0.10
	R3,R4,R5,R7	10K Ω Resistor	\$0.20
	R6	1K Ω Resistor	\$0.05
	C1-C4	0.01 μ F Capacitor	\$0.50
	D1,D2	General Purpose Si Diode 1N914 or equiv	\$0.50
	Q1,Q2	General Purpose NPN BJT 2N3904 or equiv	\$0.50
	Q3	General Purpose PNP BJT 2N3906 or equiv	\$0.25
CON-242 (AE)	X5,X6,X9	2-Pin Connector w/ Header	\$0.85
CON-243 (AE)	X2	3-Pin Connector w/ Header	\$1.10
	N/A	9VDC Wall Adapter, 5.5 x 2.5mm Center Positive	\$3.95
		TOTAL	\$8.00

4 A Basic Keyer Shield Board

My keyer shield board is very simple. I've included the parts I used and my schematic here to get you started. It is easy to have a small run of PCBs made professionally for a group or club build, but the goal here is for you to have the fun and flexibility of owning your own design rather than producing yet another electronic kit for hams. Besides, since you are connecting it to your transmitter, you must take responsibility for ensuring that connecting the keying circuit will not damage your rig or expose anyone to hazardous voltages. See the Appendix for some tips.

Table 2 lists all the remaining parts I used to build my keyer shield. I've also included a 9VDC wall adapter with the connector dimensions to power the Arduino. Power for the LCD display and Keyer Shield comes from the Arduino, so the entire keyer is powered as a unit in that manner. Note that if you will have a computer or powered USB port handy at your station, the keyer can be operated just by plugging the Arduino's USB cable into the PC, no AC adapter would then be required. For remote operation, see any of the online articles on correctly powering an Arduino from battery or solar. The main consideration is not to feed too high a DC voltage to the Arduino's online regulator, which must reduce whatever you provide down to 5VDC. Feeding 9VDC produces less heating than 12VDC, but my keyer seems to operate fine from a 12VDC source without the Arduino's voltage regulator getting hot to the touch.

4.1 Connecting your Key

To turn an Arduino into a keyer, you have to connect a key. Iambic paddles have three wires: a ground, a dot contact, and a dash contact. Pressing the dot lever connects the dot wire to ground, and pressing the dash lever connects the dash wire to ground. Squeeze, and both wires get grounded. All we need to do is connect the dash and dot wires to pins on the Arduino.

Digital inputs 2 and 3 (D2 and D3) on the Arduino Uno are uniquely suited to being key inputs, because they function as *interrupts* (INT0 and INT1, see Figure 1). Interrupts on the Arduino do exactly what their name says- whenever the voltage on D2 or D3 changes, the Arduino will interrupt whatever else it's doing and do a specific task associated with interrupt INT0 or INT1. In our case, when a key contact closes, the Arduino immediately notes your request, for responsive iambic operation.

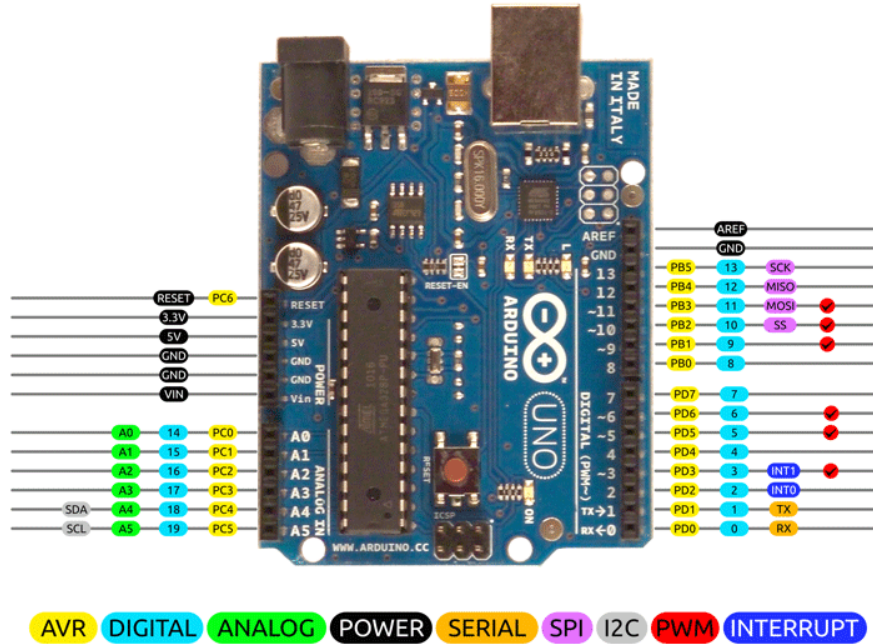


Figure 1: *Arduino Uno Pin Header Definitions. Most pins can be configured as Inputs or Outputs, Digital or Analog. Some have special purposes, especially D2 (INT0) and D3 (INT1), which are very handy for a keyer, see text.*

When deciding what pins to use for a given purpose, we must be aware of what pins are already in use by other parts of the project we’re building. The LCD Keypad shield (Figure 4) uses pins D4 through D10, as well as A0/D14 (notice that 6 of the Arduino’s pins can be either Analog or Digital in nature). Since D2 and D3 are available, and perfect for keyer operation, let’s connect those pins (through 470 ohm current limiting resistors) to the 3-pin connector that will go to our paddles. Which paddle goes where? It doesn’t matter right now! Dots, dashes, bugs or straight keys can connect so as to ground either D2 or D3, we’ll be able to sort out which in the Arduino’s programming, later. You have a lot of freedom when building projects with microcontrollers like the Arduino.

Figure 2 shows how I connected the key to the Arduino. Simple, right? The connector marked JP1 is the header shown at the lower right edge of the Arduino board in figure 1. See how the signals correspond? And don’t just copy my 470Ω resistor! I came up with this value using Ohm’s Law: $V/I = R$. Dividing the (normal) maximum voltage coming from the Arduino (5V) by a safe 10 mA current gives 500Ω which I rounded down to 470Ω. And the 0.01μF capacitor presents an impedance of $\frac{1}{2\pi \times f \times C} = 16\Omega$ at the lowest HF frequency ($f = 1$ MHz) a ham would likely operate. The rule of thumb is that the bypass capacitor impedance should be 10% or less of the series resistor (R1 and R2). Do you have some capacitors of different values in your junk box? Grab a calculator and see if you can substitute yours instead! Congratulations, with that simple act you have graduated from kit assembler to circuit designer and home brewer!

The circuit of Figure 2 behaves pretty well. But if I route the wire between the keyer jack X2 and my key or bug across the back of my transmitter, the stray RF causes mis-keying. Routing the key’s wire away from the keyer so that it does not come near the transmitter eliminates this problem. But I do not currently run an amplifier. If you do, consider adding a ferrite toroid choke in series with the resistors. Many good commercial keyers do not include a choke in this position, however when you homebrew, you are in control!

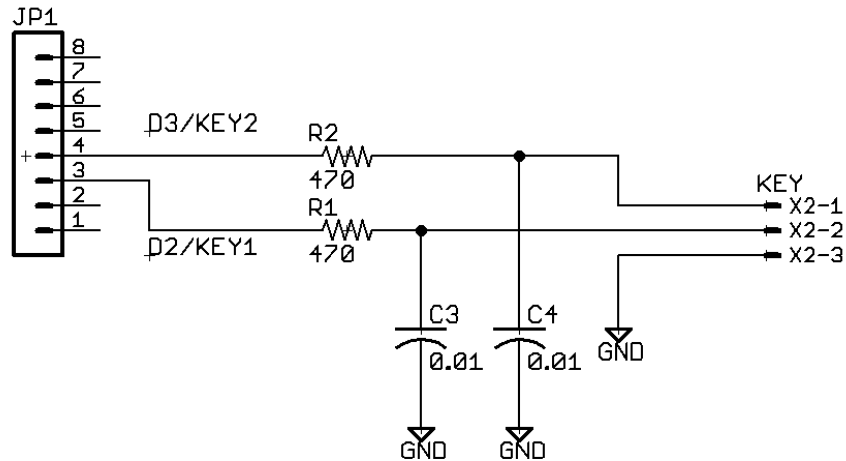


Figure 2: Connecting iambic paddles (or a bug or straight key) to the INT0 and INT1 pins of the Arduino Uno. The resistors limit current, and the capacitors help short stray RF signals to ground. See the text for details.

5 Connecting Your Transmitter

WARNING!

Making ANY external connection to a radio transmitter can result in potential exposure to dangerous voltages and brings the risk of burns, shock, electrocution, including death. Consult a qualified radio service technician before making such connections.

Never rely on ANY key or keyer as the sole means of preventing unintentional energizing of a transmitter.

CAUTION!

Making ANY external connection to radio equipment can result in potential damage to that equipment, and may void manufacturer's warranties. Consult the manufacturer and/or a qualified radio service technician before making such connections.

The other primary function of the keyer shield PCB is to connect an appropriate Arduino pin to the key input on your rig. Our goal is to be able to give a command that changes the level of one of the digital output pins to key the transmitter. Most modern solid state rigs are keyed by grounding their key input. A simple way to do this is to use an NPN transistor, connected in "open collector" fashion (Q3 in Figure 3). On my shield board, programming a positive voltage on pin D18 biases the base of transistor Q3 positively relative to the grounded emitter. This causes the transistor to "turn on" effectively connecting the transmitter's keying input (SS Rig) to ground. Resistor R7 limits the current into Q3's base, and capacitor C1 helps prevent RF entering the Arduino from the transmitter's keyer cable, as discussed above. Diode D1 prevents Q3 from being damaged by reversed keyer wires.

Some older rigs (like my Kenwood TS-520S) that use tube finals are keyed differently. In these rigs, the keyer must send a positive voltage to the radio. This is sometimes called "grid block" keying, and it can be accomplished by the circuit consisting of R3-R6, Q1, Q2, C2, and D2. Since all my homebrew rigs are solid state, but my commercial rig uses grid-block keying, I have both keying circuits on my shield board. Raising Arduino pin D16 to "high" puts a positive bias on the base of Q1, which turns the transistor on. When Q1 turns on, it grounds R5, lowering the base voltage of Q2 below the supply voltage. Because Q2 is a PNP transistor, it operates differently than Q1: lowering its base voltage turns it on, connecting the supply voltage to the GRID BLK output, keying the transmitter.

Notice that all power for the keyer shield board comes from the Arduino header JP4, which corresponds to the six POWER pins on the header in the upper left edge of the Arduino in Figure 1. Connector X1 is a 3-pin header that uses an old-fashioned jumper to select either regulated 5VDC from the Arduino or unregulated DC from the AC adapter. I use the jumper between X1-2 and X1-3, so that the grid block keying voltage on the collector of Q2 is unregulated. This works and avoids making the Arduino's voltage regulator do the work of keying. We want the Arduino's digital brain to be powered as glitch-free as possible.

Not sure how your rig's keying circuit works? Check its schematic, or try the tips in the Appendix.

6 Gotta Have Beep!

You will want your keyer to be able to make sounds. This is nice for monitoring your keying if your rig doesn't provide a sidetone, but it is also very nice for debugging operation of your station, and for offline sending practice. It's nice to have audio cues to tell you what is happening by sending responses. When the keyer powers up, for example, it beeps "OK"... in CW, of course! Making your Arduino beep is easy by connecting an available pin (I chose D19) to a suitable audio sounder device. A great choice is a passive piezo element, which lets you change the frequency of the beeping by programming the Arduino to send pulses at the audio frequency of your choice. But you could choose a mini PCB speaker or such. Whatever you connect, use Ohm's Law again to make sure the maximum current rating of the Arduino pin (40 mA) is not exceeded. The Mallory PT-4175WQ Piezo (DigiKey has them) is bulky but makes a nice loud sidetone.

7 Adding an SD Card

The rest of the wiring in Figure 3 simply connects the Arduino's available SPI interface pins to the corresponding ones on the SD Card Reader board listed in Table 1. I will let you figure this part out, but you will see it is just a matter of matching the MOSI, MISO, SCK, and power pins from the Arduino to the corresponding ones on the SD card board. Excellent instructions for using the SD card reader with an Arduino are found in the Arduino instruction materials for the SD Library. One bit of trickery is that I used D15 as the "Chip Select" wire for my SD card. I did this because the usual one (D10) was already in use by the LCD Keypad shield. Fortunately, the SD programming library on the Arduino allows us to select a different Chip Select wire, so we can use the LCD Keypad and the SD Card in the same project without conflicts.

8 Cases and Other Finishing Touches

That's it for the keyer's innards. All that remains is to add connectors for your key(s), rig(s), and power. I used 1/4" phone jacks for my keys (stereo jacks for paddles), and RCA jacks for the keying lines to the rig(s). My unit has a bug input, a paddle input, with the bug input connected in parallel with the dash paddle line. (I have a menu item in my unit that chooses "paddle" or "bug" so that I can go from one key to another in the same QSO.) Multiple bugs, keys or paddles can be enabled by just connecting all keys in parallel. The key and rig jacks were wired to pin connectors that plug onto the shield board so I can rapidly disassemble the whole keyer for modification without desoldering. For DC power, I use the AC adapter directly to the Arduino. That's it.

A custom case adds fun and style to your project. I chose an unconventional approach, a box glued up from 1/8" thick wood sheet, purchased from Cherokee Wood Products. The result is a steam-punk marriage of old-world woodcraft and modern digital style. Wood doesn't provide shielding, but you could line a wooden case with conductive metal foil, or use an aluminum box. I used brass brads, traveling in drilled holes, to operate the keypad buttons through the case.

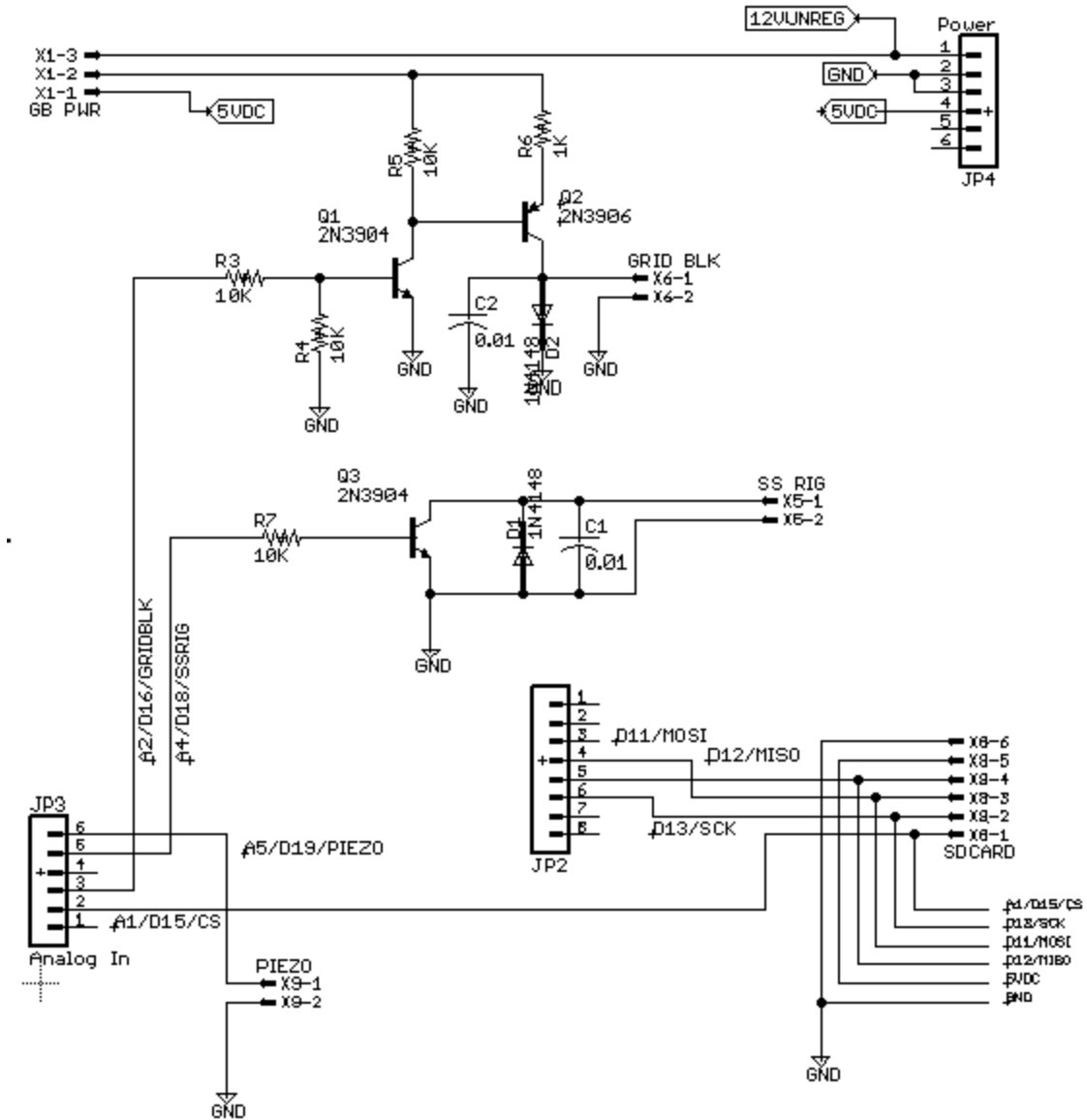
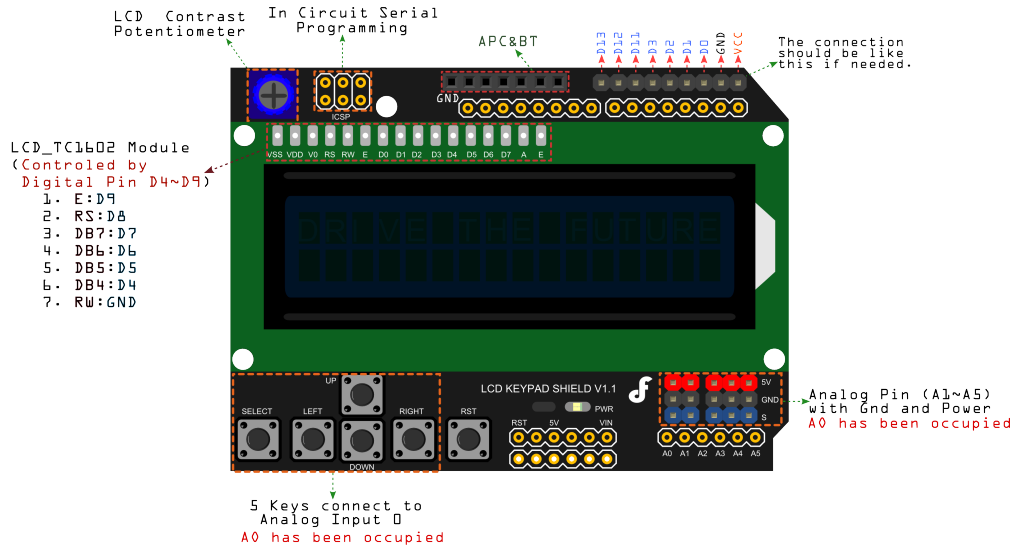


Figure 3: Remainder of the Shield Board Schematic showing outputs from the Arduino to key solid state or grid block rigs. Also shown is the optional SPI port wiring for the SD card reader, the piezo/speaker output. Connector X1 (GB PWR) is a jumper for selecting keying power for grid block rigs. I connect X1-2 to X1-3 and leave X1-1 open, see text.



Instruction for D4 To D10 and Analog Pin 0

Pin	Function	Instruction
Digital 4(D4)		
Digital 5(D5)	D4~D7 are used as DB4~DB7	Four high order bidirectional tristate data bus pins. Used for data transfer and receive between the MPU and the LCD.
Digital 6(D6)		
Digital 7(D7)		
Digital 8(D8)		
Digital 9(D9)	Enable	Starts data read/write
Digital 10(D10)	LCD Backlight Control	
Analog 0(A0)	Button select	Select, up, right, down and left

Figure 4: The DFRobot LCD Keypad Shield layout and pin usage. There may be clones of this board on the market, so check yours in case its pinout varies from this figure. The 6 keypad buttons short resistors in a voltage divider to communicate to the Arduino using only one analog input pin (A0/D14). The rightmost button reboots the Arduino.

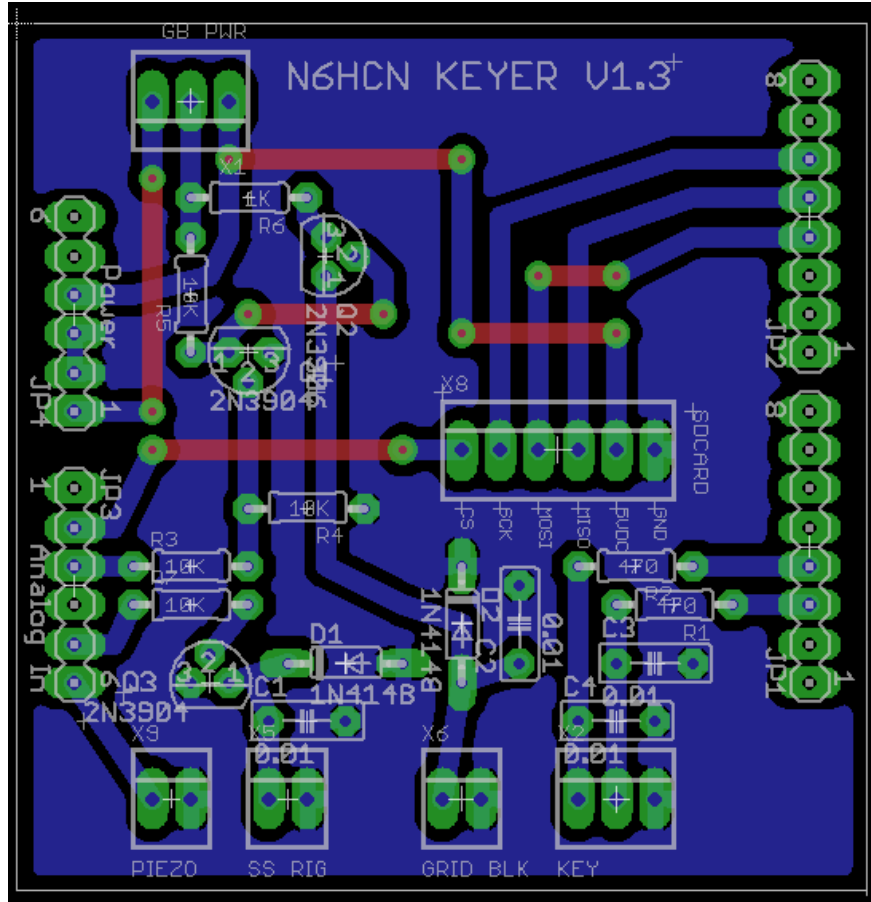


Figure 5: Screen shot from Eagle PCB software shows the layout of my keyer shield board. Actual size is 2.3 inches square. I fabricate single boards by ironing laserjet toner off of photo paper onto the copper clad, then etching, drilling, stuffing and soldering. But you could use one of the many Arduino Shield Proto Boards available online.

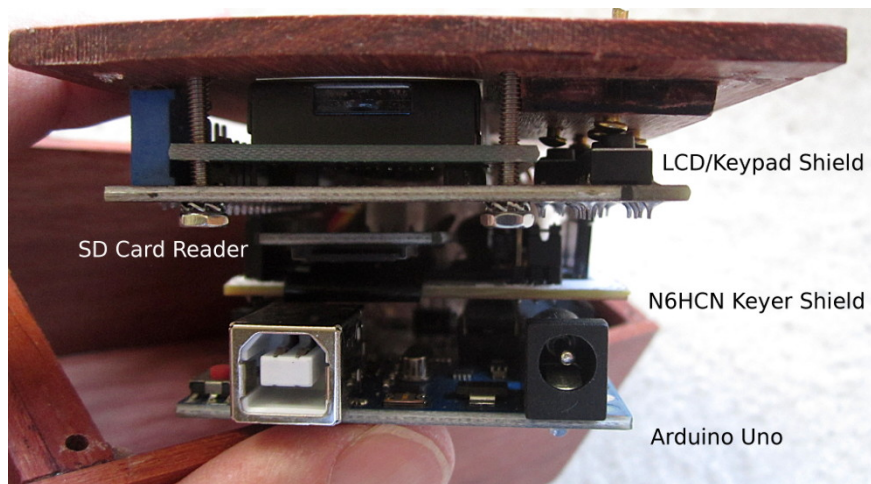


Figure 6: An end-on view of the assembled keyer, looking at the USB and power connector end of the Arduino Uno. All boards stack and plug together using the Arduino pin header system. The SD sub board is soldered to the keyer shield PCB. Notice that the whole assembly is mounted to an 1/8 inch wooden front panel.

Table 3: Arduino Uno pin assignments used by the keyer. Power pins are not shown but are referenced in schematic. LCD and SD Card interface pins are easily managed for us by Arduino LCD and SD Card libraries.

Arduino Pin	Configuration	Function	Comment
D0	Unused		
D1	Unused		
D2 (INT0)	Digital Input	Key/Paddle 1	Paddle or Key actuation connects pin to GND.
D3 (INT1)	Digital Input	Key/Paddle 2	Paddle or Key actuation connects pin to GND.
D4	Bidirectional	LCD Display Data	Handled by Arduino LCD Library
D5	Bidirectional	LCD Display Data	Handled by Arduino LCD Library
D6	Bidirectional	LCD Display Data	Handled by Arduino LCD Library
D7	Bidirectional	LCD Display Data	Handled by Arduino LCD Library
D8	Digital Output	RS	LCD Display Function (LCD Library)
D9	Digital Output	LCD Enable	LCD Display Updates (LCD Library)
D10	Analog Output	LCD Backlight	Controls LCD brightness.
D11	SPI	MOSI	SD Card Comms (SD Library)
D12	SPI	MISO	SD Card Comms (SD Library)
D13	SPI	SCK	SD Card Comms (SD Library)
D14 (A0)	Analog Input	Keypad Button Data	Voltage indicates button pushed.
D15	SPI	Chip Select	SD Card Communications (SD Library)
D16	Digital Output	Key GB Rig	Logical "high" output keys rig.
D17	Unused		
D18	Digital Output	Key SS Rig	Logical "high" output keys rig.
D19 (A5)	Analog Output	Piezo/Speaker	Arduino "Tone" function gives audible sound from keyer.

Other than to remove or replace the SD card, there isn't much need to access the keyer's insides. But I highly advise making the Arduino's USB connector available from the outside. This will allow you to reload new versions of your control firmware as you dream up new features, without even disconnecting the keyer from your station. (I've done this mid-QSO, in fact.) One of the joys of this kind of project is that you can tweak and customize your keyer without soldering or disassembly, just by uploading new code to the unit from a computer via the USB.

You can also use the USB and the Arduino Terminal application to send text from your computer keyboard directly to the keyer. In this way, the keyer can also be used as a Morse sending Keyboard. There is a small amount of noise in my receiver when I am using the keyer as a CW Keyboard. But this noise is only present when I am sending, and of course disappears during receive when the keyer is idle and no data is being passed over the USB.

9 Programming Hints

Of course your keyer will not do anything when you assemble it and power up! You must teach the Arduino how to be a keyer, by programming it in the C language. Since "programming" sounds scary, the Arduino community refers to programs as "sketches", though this is a gilt-edged distinction. It is not hard to learn C if you have never programmed. If you are a programmer, you will find that the Arduino supports many C++ objects, allowing for some real sophistication. If you're new to programming, the code examples below might look hopelessly confusing. Please don't be discouraged! The Arduino community has done a wonderful job of producing step-by-step tutorials that will get you started quickly.

No matter your prior experience, this section is designed to get you started. But before you begin, grab just your Arduino board and work through some of the introductory Arduino programming exercises available on the Arduino website. You should soon be able to flash the onboard LED of the Arduino using the "Blink" demo code. Study that example until you understand how to use the Setup and Loop code blocks, and also how to set any of the digital pins high or low. If you can blink the LED, you can "blink" the keying lines on the shield board, which will also "blink" your transmitter! All we have to do is make the blinking into CW. Fortunately it is not hard to do.

9.1 Testing the Hardware using Arduino Example Sketches

I recommend that you begin by testing the keyer hardware, one function at a time. To test the keying outputs, you could use the "blink" sketch from the Arduino tutorials, but change the pin to the one that should key your transmitter. Check function using a voltmeter and suitable current limiting resistors. Only when you are sure all is well should you try your rig's key input. The piezo output can be tested using the "tone" function in the Arduino libraries. Look this up and study the example code. You should be beeping in no time. Use the "digitalread" function to test the key inputs. The LCD display is also easy to get working. Look at the LCD library and load an example sketch like the "Hello World!" demo. Be sure to change the pins in the example sketch to match your circuit. When you load that sketch, your LCD should come to life. For some reason, the LCD display as shipped from the factory usually requires a screwdriver adjustment of contrast when you first power it up—I find that until this is done, even a good unit will not display anything.

9.2 The Secret of Iambic Keying

I was worried at first about how to program the Arduino for iambic keying. There seems to be a lot of mystique around the famous "Curtis Keyer IC" of my youth, and I thought this might be a real Pandora's box to homebrew. It's not.

The iambic journey begins with learning about Arduino interrupts. This is considered a moderately advanced topic, but it's really simple. The INT0 and INT1 pins on the Arduino Uno (Figure 1) have a special

power. You can program these pins as a special kind of input: if the voltage on one of these pins changes in a particular way, such as going from logic "high" to logic "low", then *no matter what the Arduino may be busy doing*, processing will *immediately* be *interrupted*, and the Arduino will execute a special function associated with that pin. (You get to provide that function.) Once the special function is finished, the Arduino goes back to what it was doing before the interrupt happened. The special function is called an "Interrupt Service Routine" or ISR for short.

Interrupts on the Arduino are easy to set up. The function "attachInterrupt(a,b,c)" takes three parameters, (a, b, and c) and establishes an interrupt triggered whenever pin (a) changes in a way described by (c). When this happens, the ISR given by parameter (b) is executed. After the ISR executes, the Arduino goes immediately back to what it was doing before the interrupt occurred. Below is a bare-bones example of how you might set up interrupts for iambic keying. If you have not programmed before it might look like Greek, but after some Arduino tutorials the fog should begin to clear. Note that text after "//" on any line are comments to help you understand what is going on).

```

//-----
// Define pins used by the keyer shield PCB
//-----
const int key = LED_BUILTIN; //a debug help.
const int ditPin = 2; //one of two paddle pins.
const int dahPin = 3; //the other paddle pin.
const int piezo = 19; //pin used for piezo buzzer
const int grid_block_out = 16; //keys a grid block rig
const int ss_rig_out = 18; //keys SS rig

//-----
// Variables Related to CW
//-----
int wpm = 25; //words per minute
// PARIS has 50 dots, CODEX 60 dots.
const unsigned int dot_per_word = 50;
float dash_over_dot = 3.0; //element ratio
float dot_msec, dash_msec;
volatile bool ditflag = LOW;
volatile bool dahflag = LOW;

//-----
// The "setup" function runs at power-up.
//-----
void setup(){

    // Initiate Keyer Pins
    pinMode(ditPin,INPUT_PULLUP);
    pinMode(dahPin,INPUT_PULLUP);
    digitalWrite(key,LOW);
    digitalWrite(ss_rig_out,LOW);
    digitalWrite(grid_block_out,LOW);

    // Calculate lengths of dits and dahs in milliseconds
    dot_msec = 1.0 / wpm / dot_per_word * 60.0 *1000.0;
    dash_msec = dash_over_dot * dot_msec;

    // Assign interrupts to paddle pins
    attachInterrupt(digitalPinToInterrupt(ditPin),dit_isr,LOW);
    attachInterrupt(digitalPinToInterrupt(dahPin),dah_isr,LOW);
}

```

Now you see why the key contacts are connected to pins INT0 and INT1. When you close the key contacts, those pins go from HIGH to LOW. The result is that every paddle push or contact closure on your bug triggers an interrupt. This means the Arduino will be very responsive to the key. When the dit contact closes, the Arduino will see the keyer pins transition from high to low, and execute the ISR named “dit_isr” once. When the dah contact closes, the Arduino quickly executes the “dah_isr” once before going back to whatever it was doing. All of the names for these variables, pin names, and user-written functions are entirely up to you, of course.

During iambic keying, we send letters like Q by holding down the dah paddle, then “punching” in a dit in by tapping the dit paddle. Conversely we send R by holding the dit paddle and tapping the dah paddle to inject a dah. Notice that we cannot write the dah_isr function so that it sends a dah! That’s because the dah has to be precisely timed depending on the WPM in use. We don’t want to interrupt

sending a dah to send a dit! But that would happen if our dit_isr sent a dit and the dah_isr sent a dah. The keyer has to always be in one of three states: doing nothing, sending a dit, or sending a dah. We need to use interrupts but we cannot alter the timing of our CW. Ideally, our ISRs must execute so quickly that they make no noticeable change in CW timing, even at 50 WPM! We want our ISRs to be lean and mean.

One solution is to use *flags*. These are usually *boolean* variables, meaning they take on the value true or false, 0 or 1. Let's call these ditflag and dahflag. Normally, these flags will be set to "false" (or 0 or LOW). When you close the dot contact, the dit_isr will set the ditflag HIGH (or 1 or "true"). Conversely the dah_isr will assert dahflag. (Notice that for arcane reasons, variables changed by an ISR must be declared as "volatile".) Now our ISRs only need to set a flag that indicates a dah or a dit has been keyed by the paddles. We can be even more economical in our ISRs by avoiding repeated execution if the paddle contacts bounce: this would cause multiple interrupts and needless setting of the (already set!) flags. We can prevent this by having the ISRs also disconnect their own interrupt! Now when the dit paddle is closed, the flag is set and the keyer ignores any more dit paddle closures. Of course we need a way to actually send the dits and dahs, and re-enable the interrupts. We'll get to that, but let's take stock of the code so far. (I have repeated the declarations of ditflag and dahflag again here for clarity.) Using flags, a pair of lean, mean ISRs for dits and dahs might look like this:

```
volatile bool ditflag = LOW;
volatile bool dahflag = LOW;

void dah_isr () {
// flag dah paddle closure
dahflag = HIGH; //set the dah flag
detachInterrupt(digitalPinToInterrupt(dahs)); // ignore ints during character
}

void dit_isr () {
// flag dit paddle closure
ditflag = HIGH; //set the dit flag
detachInterrupt(digitalPinToInterrupt(dits)); // ignore ints during character
}
```

At this point, closing a paddle contact quickly sets a flag and causes the keyer to ignore future closures of that paddle. It doesn't seem like this is very useful! But really, we are almost there. Let's write two more functions, one to send dits (doDot), and one to send dahs (doDash). We'll key and un-key the transmitter using two other functions (keyDown and keyUp). Notice that when a dah or dit is sent, the appropriate flag (dahflag or ditflag) is cleared, and the appropriate interrupts are re-established. Notice that all of these functions, and the ISRs themselves, are not placed inside the main Arduino "loop" function that executes repeatedly over and over. We want all of these functions to only execute when we actually move a paddle. Notice also that the variables dot_msec and dash_msec, establish the length of dots and dashes in time. We computed these values from our desired WPM in the setup code example above. Since setup only runs once when we power up our keyer, the WPM will be "fixed" in this example. To make code speed adjustable, you will want to let the user modify this variable (and recalculate the element durations) using the keypad. As college professors say, we'll leave that as an exercise. But rest assured that the instructional materials and examples on the internet for the LCD Keypad will get you going in no time.

```

void doDot() {
// send one dot and interelement space, managing dit interrupt/flag.
  keyDown(); //start the dot
  delay(dot_msec); //this is the length of the dot
  keyUp(); //end of the dot
  delay(dot_msec); //pause one dot space
  ditflag = LOW; // clear the dit flag
  attachInterrupt(digitalPinToInterrupt(dits),dit_isr,LOW); // back to normal
}

```

```

void doDash() {
// send one dash and space, managing dah interrupt/flag
  keyDown(); //start the dah
  delay(dash_msec); //this is the length of the dah
  keyUp(); //end of the dah
  delay(dot_msec); //wait one dot space
  dahflag = LOW; //clear the dah flag
  attachInterrupt(digitalPinToInterrupt(dahs),dah_isr,LOW); // back to normal
}

```

```

void keyDown() {
// Key the transmitter
  digitalWrite(key,HIGH); //this lights an LED
  digitalWrite(ss_rig_out,HIGH); //Key a SS rig
  digitalWrite(grid_block_out,HIGH); //Key a GB rig
}

```

```

void keyUp() {
// Unkey the transmitter
  digitalWrite(key,LOW); //turn off the LED
  digitalWrite(ss_rig_out,LOW); //Unkey a SS rig
  digitalWrite(grid_block_out,LOW); //Unkey a GB rig
}

```

With all the above pieces in place, full iambic keying (type B) is easy to make happen. All it takes is this code in the Arduino's "loop" function, which the Arduino slavishly runs over and over following a single execution of "setup":

```

void loop() {
//The whole, glorious iambic keyer function:
  if (ditflag) {
    doDot();
  }
  if (dahflag) {
    doDash();
  }
} //void loop

```

The simplicity of this code surprised me, but you will find if you trace through the logic that things like dot injection and dash injection all happen very naturally. Squeezing the paddles results in alternating dits and dahs, just as we want. Basically, the Arduino spends its time watching ditflag and dahflag to detect a paddle closure. Its attention alternates between these flags, and it sends whichever element, be it a dah or a dit, is triggered first.

10 Beyond the Basics

You are now in a position to add a nearly unlimited number of features to your keyer. Really the only limits are your needs, your imagination, and the limitations of memory and available pins on the hardware. Be aware that the Arduino has rather limited dynamic RAM for variable storage. I find that the Uno has plenty of memory for nearly any imaginable feature so long as you are clever and do not use unnecessary variables. But an annoying fact of life with the Arduino is that exhausting the available memory will lead to your code operating unpredictably. In one case my keyer would spontaneously reboot itself for no apparent reason. The culprit turned out to be my use of the SD card library, which is a real memory hog. I was able to solve that problem by reducing the number of variables in use. Like living on a boat, programming microcontrollers can be a healthy exercise in minimalism.

10.1 Morse Keyboard

You can make your keyer into a Morse keyboard by putting this code into the “loop” function:

```
//  
// This code implements morse keyboard very economically.  
//  
if (Serial.available()) {  
    CWchar = Serial.read(); //get a character to send  
    sendInCW(CWchar); //send it!  
    Serial.write(CWchar); //echo sent CW back to computer  
}
```

You’ll need to initialize the serial port with a particular baud rate (in the setup function), and you’ll need to write a function `sendInCW` to send Morse characters. But this is not hard, you already have `doDash` and `doDot`, right? If you are new to C programming it might take you some weeks to do this, but it really isn’t hard. One suggestion is to store the Morse alphabet in a look up table of static arrays. Don’t fall into the beginner’s trap of trying to use `if/else` structures.

10.2 Storing the Morse Alphabet

My keyer stores the Morse characters like this:

```
//-----  
// A Morse Code Lookup System:  
// the array elemCW stores dot/dash patterns  
// the array lenCW stores the length of each character  
//-----  
// chars below are 0 = dot 1 = dash, from R to L.  
static const byte elemCW[41] =  
{  
B0, // space, 32 <- this is the ASCII code for space.  
B110011, //comma, 44  
B101010, //period, 46  
B01001, // slash/, 47  
B11111, // 0 48  
B11110, // 1 49  
B11100, // 2 50  
B11000, // 3 51  
B10000, // 4 52  
B00000, // 5 53  
B00001, // 6 54  
B00011, // 7 55  
B00111, // 8 56  
B01111, // 9 57  
B001100, //?, 63  
B10, //A 65  
B0001, //B 66  
... etc, see the pattern? ...};  
  
// Number of elements in characters above.  
static const byte lenCW[41] =  
{  
0, // space  
6, //comma  
6, //period  
5, //slash  
5,5,5,5,5,5,5,5,5,5, //numbers 0-9  
6, //?  
2, //A 65  
4, //B 66  
4, //C 67  
3, //D 68  
1, //E 69  
4, //F 70  
3, //G 71  
4, //H 72  
... etc type the rest yourself! .};
```

Once you have these arrays defined (static, because they never change), you can send CW by just looking up the characters in the arrays. Notice that the ASCII codes from the keyboard need to be converted into the proper index into your Morse arrays. Nothing in this code is very tricky, but if you are new to programming you might have to study how the shift operator (\gg) and the bitwise AND ($\&$) work in this code. (Hint: \gg and $\&$ let you look at each binary bit in the elemCW array to decide whether to send a dah or a dit.)

And consider that to send a character we have to know when to stop– that is determined by the lenCW array. It's important to realize that my example here is just one way of sending automatic CW. You may well improve on my code, and learn and have fun by doing so. So please experiment, it's the tradition of ham radio.

```
void sendInCW(char ichar)
{
  // sends a CW character
  int iint = convAscii(ichar);
  for (int indx = 0; indx < lenCW[iint]; indx++) {
    int idd = (1 & (elemCW[iint] >> indx));
    if (idd) {
      doDash();
    } else {
      doDot();
    }
  }
  delay(dash_msec); // inter-character space.
}

int convAscii(char iascii)
{
  // converts ascii codes to index into Morse lookup
  switch (iascii) {
    case 32: //space
      return 0;
      break;
    case 44: //comma
      return 1;
      break;
    case 46: //period
      return 2;
      break;
    case 47: //slash
      return 3;
      break;
    case 63: // question mark
      return 14;
      break;
    default:
      if (iascii > 47 && iascii < 58) {
        return iascii - 44; //digits
      } else
      if (iascii > 64 && iascii < 91) {
        return iascii - 50; //letters
      } else
      if (iascii > 96 && iascii < 123){
        return iascii - 32 - 50; //lowercase
      } else return 0;
      break;
  }
}
```



Figure 7: Closeup of the 16x2 LCD Display/Keypad in use. This display indicates operation at 25 WPM, iambic keying mode, QSO serial number set to 001, and sidetone on. The cursor (>) indicates that up/down buttons will adjust WPM. The right/left buttons scroll the whole display to reveal other parameters like sidetone frequency, etc.

10.3 User Interface/Display

Obviously the intent of the LCD display and keypad is to allow you to interact with the keyer during operation. This means you will need to read the keypad buttons, and print text and numbers to the display. These input and output operations are very well described on the internet in conjunction with the LCD Keypad shield, so the basics are easy. All that is required is to check for a button press in the Arduino loop function. This means that between dots and dashes the Arduino will be making quick checks to see if the keypad has been pressed. This checking takes very little time (one AnalogRead operation requires a few microseconds, to be exact) and will not impact your keying performance. Actually *responding* to button presses may take a bit longer, and may lead to hesitation in keying, but generally you're either sending or pressing buttons, but not both. But where to begin?

Here is where you get to design a user interface (UI) for your keyer, and it can be as complex or as simple as you wish. A good approach to this task is to diagram the 16x2 character LCD on a sheet of paper and think about how you want to interact with it. I chose to have a “parameter + value” paradigm. Each of my keyer variables has a display name, and an associated variable. On the LCD, I print the name and the current variable value, such as “WPM” and “25”. You might use the up/down buttons to change what parameter/value pair is displayed, and the left/right buttons to change the parameter value up/down. Or vice-versa. Don't forget, by the way, to add a splash screen that proudly displays your call for a few seconds during power-up!

11 Features, from the Sublime to the Ridiculous

This is just a list of possible features and some thoughts on how to implement them. For more ideas, download and read the manuals for high-end keyers, or maybe you'll want to emulate the operation of a favorite keyer from the 1970s. The choice is up to you!

- **Paddle Swap:** If your paddles happen to be wired wrong for the way you built your keyer, it's handy to have a boolean variable that assigns the keyer pins "the other way 'round". Voila— your keyer will work with any paddles, no matter how they're wired.
- **Dah to Dit Ratio:** In the code examples above I set the dash length to $3 \times$ the dot length. This is conventional, but you can vary it. A ratio of 2:1 is hard to copy, and high values (above 5:1) sound like a bug operator with "swing". Honestly I am not sure why anyone would want this feature. But if you're really eccentric, you could use the `rand()` function to simulate a human fist.
- **Iambic/Manual/Tune Select:** Most of us use paddles and bugs or even straight keys from time to time, and it's nice not to have to plug and unplug things. A UI feature that allows you to choose iambic operation, manual keying, or continuous key-down can be helpful in daily operation and transmitter tuning. It seems anachronistic to key a microprocessor with an antique key, but the low voltage and current in the Arduino keying line (and clever use of interrupts) can also smooth out dot contact chatter and other keying problems associated with bugs designed for tube gear, or rigs that have keying quirks.
- **Multiple Jacks:** My keyer features input jacks for paddles, bugs, and straight keys, paralleled so that all can be live at once. I also have different outputs for the two keying circuits shown in Figure 3 so that my TS520S and a solid state rig can both be keyed without rewiring.
- **Farnsworth Spacing:** This is as simple as letting the UI change the duration of the character space in the function `sendInCW`. Obviously it applies only to keyboard-sent CW.
- **Keypad Dynamics:** As you program your UI, you will find you must add a debounce delay each time a button press is detected. This will prevent one button push from giving you multiple responses. It is nice to reduce this delay by, say, 10 % each time the same button press is detected on consecutive loops. This will let you hold the button down to make faster and faster changes to a parameter. (If this is confusing, think about how you set the time on a digital clock.)
- **Audio Feedback:** The ability to send characters in CW using a function like `sendInCW` is very handy. Your keyer can send "OK" when it powers up, or provide other audio feedback to you using the piezo buzzer. Be sure to disable keying the transmitter for these messages!
- **Type A or B Iambic:** Most of us use one particular form of iambic keying. While the example here gives the more common "B" keying, you may enjoy being able to select either mode. The logic to do this is simple and I'll leave it for you to discover and implement it so the UI can select one or the other.
- **Persistence of Settings:** It's nice if your keyer remembers how it was set the last time it was used. You can save and reload any parameter using the SD card or the EEPROM library, which puts data in your Arduino's non-volatile memory from which you can recall it at will. This EEPROM memory can only be written a few hundred thousand times, but figure it out— the Arduino is likely to last a lifetime, and the chip is socket mounted on the Uno anyway...
- **Sidetone Parameters:** The piezo buzzer output on the shield board can be used by adding a call to the Arduino "tone" function in the `keyDown` function, and a call to "noTone" in the `keyUp` function. You can turn sidetone on or off, or change its frequency to any desired pitch by adding UI controls.
- **Contest Serial Numbers:** If you're an avid contester, it isn't hard to use `sendInCW` to send a serial number at the press of a button, and to increment or reset the serial number with the LCD Keypad. With a little experimenting you can send leading zeros as "cut", too!

- **LCD Backlight Control:** Don't forget about pin D10 in Table 3. You can have the LCD display "sleep" to save power, adjust its brightness, or even flash the display to signify something important. On my keyer, the display goes off if the keyer is idle, is dim if I am in a QSO (see the ID timer below), and is bright if I press a button.
- **ID Timer:** I am a ragchewer and long-winded enough to sometimes wonder if I have failed to ID properly. If this worries you, or you wish to time your transmissions for other reasons, consider adding a timer. My keyer starts the timer when I first send a character, and resets the timer if I am silent for 10 seconds. If the timer ever reaches an (adjustable) number of minutes, the display starts to flash. Pressing any button resets it, as does any ten second rest.
- **Morse Keyboard:** I enjoy ragchewing with Fred, K1NVY who is very good with his Morse keyboard. The type-ahead allows for very good sentence construction and nearly no spelling errors! The Arduino Terminal Monitor application is free and part of the IDE, and can easily be used to send your end of the CW. You'll want to add the ability to send prosigns, perhaps by adding to the example lookup tables included above.
- **Memory Keyer Goodies:** You can implement SD Card storage and recovery of large amounts of text if you want to send the ARRL bulletin or even War and Peace. But unless you want to do all this in a very self-contained package, consider using USB and a laptop terminal program to send characters to the keyer. You'll have all the function keys and logging capability that goes with a PC in the bargain. (I like the Raspberry Pi...)
- **Solar Powering** Solar powering the keyer is easy, just watch the voltage as indicated in the text. Very inexpensive 12VDC to USB adapters are available. You can homebrew a very smart solar charge controller using another Arduino, LCD display, and a custom shield board, but that is a topic for another project guide!
- **Head Copy Trainer:** Your keyer can be a speed-adjustable CW player that will stream entire e-books in Morse.¹ Store the book text (from Gutenberg.org, perhaps) on the SD card and then feed it to sendInCW one character at a time. Lose that pencil and your head copy speed will grow quickly!
- **Rotary Encoder Control:** It's hard to beat a rotary knob control for keyer WPM. A knob is faster than a keypad button to rapidly set WPM to match a calling station's speed. Using a free Arduino pin and a digital rotary encoder, you can provide your keyer with this capability.
- **Morse Decoder:** Don't even think about it. Use your brain for this! (See Head Copy Trainer, above.)

12 What are you Waiting For?

I hope this guide inspires you to build your own keyer, either on your own or as a club project. The hardware is inexpensive and simple enough to be assembled by young folk in one session, and the programming can be as simple or as complex as you wish. Once the soldering is done, you'll find the project stretching out for weeks (enjoyable weeks!) as you tweak this and that. And if there's a C programmer in your club, this project could be a good foundation for some in-depth how-to Elmering. Bottom line: you really can design, construct, and program a unit capable of rivaling any commercial keyer available. And by doing so you will find enjoyment and skills that will carry over into other projects and enrich your time on the air.

Next, we'll build a key to go with your new keyer. But that's another article!
See you on the air, or I'm good on QRZ.

73 ES ZUT DE N6HCN (dit dit)

¹See my article on this pastime in the May 2018 newsletter on the CWOps website: cwops.org

13 Appendix: Connect Your Rig Safely

All of my radios are homebrew or vintage ones I would take a soldering iron to. Likely your rig is more modern, and possibly under warranty. Modern rigs are sensitive to ESD (electrostatic discharge) and require care in their interfacing. Consult your manufacturer before connecting any homebrew projects to valuable rigs lest you void a warranty or cause damage.

In most cases, though, the keying line to a rig is designed to allow connection to an uninsulated straight key or bug. This implies that any voltages present are safe, and that a reasonable tolerance for ESD and electrical noise is designed into the gear. **If you are keying a vintage tube rig, however, the keying line voltage could be hazardous. Consult an expert.**

A good way to gauge your rig's needs is to measure the Thevenin DC equivalent circuit for its keying input. This is quite easy, all you need is a digital multimeter and no disassembly is involved. Here's how.

1. Connect a straight key or bug to your rig.
2. Set your rig up to transmit into a dummy load and confirm that it is keying correctly using the manual key.
3. Set your multimeter to read DC voltage. With the key up but the rig ready to transmit, measure the DC voltage across the key. This is the Thevenin open circuit voltage, V_{OC} .
4. Make sure to note which terminal on the key is the positive potential.
5. With the multimeter still set for DC Voltage, measure the potential of each key terminal versus the rig's ground connection. Write these down, call them V_1 and V_2 .
6. Set your multimeter to read current.
7. Connect the multimeter across the key contacts. The transmitter should key. Record the current, this is the short-circuit current I_{SC} that flows in your key during operation.
8. Calculate your rig's Thevenin equivalent resistance R_T in ohms by dividing V_{OC}/I_{SC} and write this down.

If either V_1 or V_2 are negative relative to the rig's ground, then you probably have grid block keying, and you would connect the rig to X6 in Figure 3. On most modern rigs, one of the keying lines will have a positive voltage equal to V_{OC} and the other will be tied to the rig's ground. In this case, the positive line would connect to X5-1 on Figure 3, and the other line to the keyer's ground, connector pin X5-2.

If you use a 2N3904 transistor as I did at Q3 in Figure 3, and a 2N3906 at Q2, make sure that your rig's V_{OC} is less than 20 Volts, and that its I_{SC} is less than 100 mA. You might get away with up to 40V and 200 mA, but watch for heating of Q2 or Q3 during key-down. Honestly, if your rig's values are this high, you probably should redesign the shield board.

14 Attribution

For instructional purposes only. No warranty of suitability or performance is made or implied for any of the design ideas described in this article.

Figure 1, Arduino Uno Pin Diagram by Bouni (<https://github.com/Bouni/Arduino-Pinout>) is used under Creative Commons license By-SA (<https://creativecommons.org/licenses/>), no endorsement by the licensor is implied.

Figure 4 by DFRobot (<http://dfrobot.com>), used with permission. See original link: (https://wiki.dfrobot.com/LCD_KeyPad_Shield_For_Arduino_SKU__DFR0009)