Morse Code Tutor

By

Dr. Jack Purdum, W8TEE

Chuck Ziegler (W8FTQ, SK) gave me my Novice license in 1954. I had a 2 tube 15W transmitter, a 40M dipole, two 7MHz crystals, a Hallicrafters S-40A, and started making contacts. In 1955, Charlie McEwen and I rode the bus to Cleveland, walked to the Federal Building, and took the General Class exam. I told myself: “If I pass, I'll never send another letter of CW again...*ever*!” I passed and, after mowing a bazillion lawns, I upgraded to a Heathkit DX-40 and worked most AM until after I graduated from school and got my first full time job. I got a Galaxy GT550 transceiver and used that until I retired. It was during those first years of retirement that I discovered the fun and challenge of working the world using QRP power levels into a not-so-great wire antenna. To be effective with my limited antenna system, CW pretty much the only choice I had. Even though I was horribly rusty, I quickly found out that I really enjoyed CW. I also found that other CW operators are extremely tolerant of a sloppy fist.

I was saddened when the FCC dropped the code requirement from the licensing requirements, but since I wasn't too active at the time, I didn't complain. Now, I wish everyone could experience the joy of making DX contacts with 1W and the satisfaction listening to a good fist brings to one's self.

I did a survey of my club and found that, of the population who currently do *not* know Morse code, 87% said they would like to learn it. Thus encouraged, at the next meeting I offered to teach a Morse code course and set a date and time during the week when I figured most members would be available. No matter what date/time combination I picked, that was the precise date and time everyone was rearranging their sock drawer. Two club members have attended the very successful CW Academy and are now avid CW operators. Indeed, one of these attendees won North American Rookie of the Year in a recent CW contest. Alas, other than these two members, I simply cannot sell the idea of learning Morse code to the members.

Maybe this project will help some of you to fall over the edge and into the CW pool. The water's great...take the plunge!

Morse Code Tutor Feature Set

First, I learned Morse code the wrong way. I learned by memorizing dits and dahs. The process of learning Morse that way requires a cumbersome translation sequence. First, you count the dits and dahs coming in. Second, your brain does the equivalent of a binary search to find the character that matches that sequence of dits and dahs. Finally, you then write the letter on a piece of paper. Not good.

The better way is to forget about dits and dahs and listen to the rhythm of the characters. Very few high-speed CW operators write anything down. Indeed, I find I'm hard-pressed to copy at 15wpm while writing the message, let alone cruising along at 30WPM. Enviable CW operators simply close their eyes and read the code as it scrolls across the back of their eyelids. If they're in a contest, they might type the log entry as they listen...it's a thing of beauty to watch.

But, if you're just starting out, how can you sense a code rhythm when the sequence is coming at you at

10 characters per minute?

You don't.

The Koch Method

The Koch Method of learning Morse code, developed by Ludwig Koch, has been around since the 1930's. Essentially, the Koch Method says that you learn code by sending/receiving it at a fairly high speed. The Koch Method starts out sending just two letters at a time, but sends them at that relatively high speed. Once you achieve success at copying pairs of letters at that speed, the method adds a new letter to the mix. Given a reasonably high initial speed, you won't have time to count dits and dahs. By necessity, you end up listening to the rhythm of the characters rather than engaging in dit/dah counting and its attendant counting/translating/searching. The spacing between letters is often stretched out somewhat over what would normally be taken to be letter spacing (e.g., 4 times a dit length—more on this below). Also, the spacing between "words" is also stretched out.

Clearly, there's an initial hump to get over before the Koch method produces results. At first, the code will simply sound like a bunch of angry bees. Eventually, however, if you practice and stick with it, things will start to fall into place.

The Farnsworth Method

The Koch Method is not the only suggested method for learning code. The Farnsworth Method starts by sending words in code at your ultimate "goal speed". In this sense, the Farnsworth Method is in perfect agreement with the Koch Method. However, the Farnsworth Method inserts relatively exaggerated delays between words. Again, this forces you to listen to patterns, not dits and dahs. For me, that's easier said than done because of the way I learned Morse code.

Today, one successful method is a combination of the two methods, and that's the approach taken here with the Morse Code Tutor. To illustrate, suppose you'd like to be able to ultimately send/receive code at (a "goal speed") of 30 words per minute (wpm). There's no way that you can start out at 30wpm in the morning and hope to be successful in a 30wpm QSO that evening. So, we throw in a little of the Farnsworth Method and set the inter-word spacing to a much slower speed. Perhaps we set the "Farnsworth Spacing" at 5wpm. When you first try this approach, it will sound like a burst from a chainsaw (i.e., word 1) followed by a slight pause followed the a second burst from the chainsaw (i.e., word 2). Don't get discouraged...it's worth the effort.

Some hypothetical numbers can illustrate this process. For the old FCC code requirement, a "word" was taken to be 5 characters per word (cpw). (Technically, the word "PARIS" was the timing benchmark.) So, if you want to copy at 30 words per minute, that translates to 150 characters per minute, on average. Applying the numbers:

```
150cpm = 30wpm * 5cpw                    // 150 characters per minute (cpm)
  .4spc  = 60seconds/minute / 150cpm     // 0.4 seconds per character (spc)
```

Therefore, the characters are coming at you at rate that is faster than one character every half second. That's the bad news. The good news is:

25cpm = 5wpm * 5cpw
2.4spc = 60seconds/minute / 25cpm // 2.4 seconds per character

This means, if we send at 30 words per minute and use a Farnsworth delay of 5wpm, you have 2.4 seconds between chainsaw bursts! If you chew on those numbers for a few seconds, you'll realize the way to get “up to speed” is to keep collapsing the Farnsworth delay to the point where it, too, is at 30wpm. Simple!

Which One to Use?

People learn at different rates using different methods. The Morse Code Tutor (MCT) allows you to adjust its parameters to use a method that you like best. You can blend the Koch and Farnsworth methods as you see fit. Keep in mind that there are no shortcuts to learning Morse code. It takes time, perseverance, and patience to master Morse code. That said, it is truly a worthwhile investment of your time.

But first, let's see what our MCT looks like. Figure 1 shows the prototype I built. Some of you may

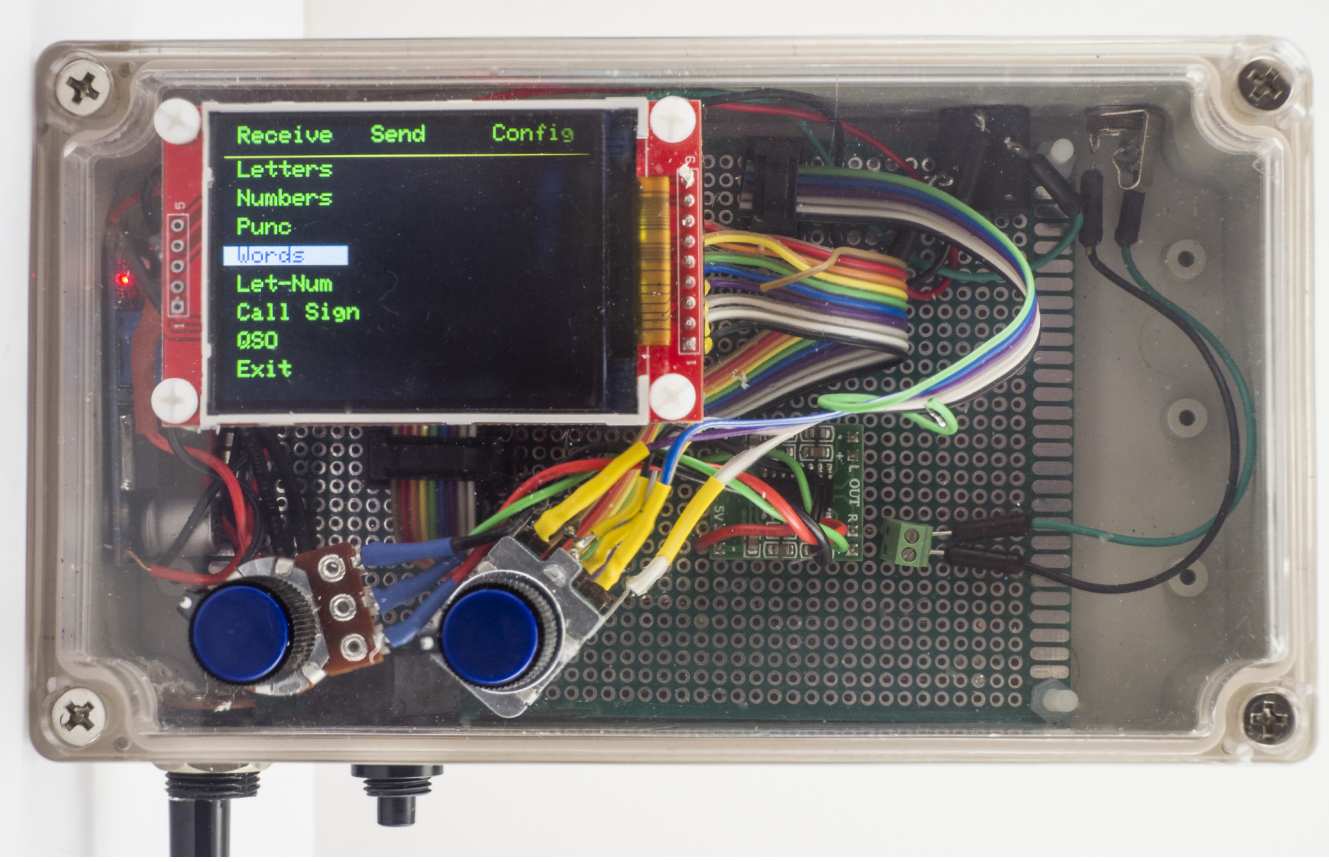

Figure 1. The Morse Code Tutor

think the case looks like the Antenna Analyzer case from November, 2017, *QST*. You'd be right, since I had a few cases left over from a club build. The case is roughly 7”x4” and, as you can see, has lots of room left over. The MCT is based upon the STM32F103 microcontroller, aka the “Blue Pill” (BP from now on). The nice thing about the BP is that you can program it from within the Arduino IDE. So far, I've yet to find a library that doesn't work with the BP, either. (It appears that some kind soul(s) took the time to redo the core libraries for the BP.) By comparison, the BP has 128K of flash memory (Nano is

32K), 20K of SRAM (versus 2K), and is clocked at 74MHz (versus 16MHz). The BP is slightly larger than a Nano, and can be purchased for under $2 in small quantities.

The display is a 2.2”, 320sx240 TFT color display which costs about $7 and uses the SPI interface to communicate with the BP. The volume control on the front panel controls a 3W amplifier. There is a phone jack and key jack in the upper-right back corner of the case. The other control is a rotary encoder with switch that controls the menuing system. I added a DC-DC buck converter which outputs 5.0V. The BP uses 3.3V, but has a regulator on it that can accept 5V. The 5V also feeds the amplifier and the display. Figure 2 shows the MCT parts outside the case.

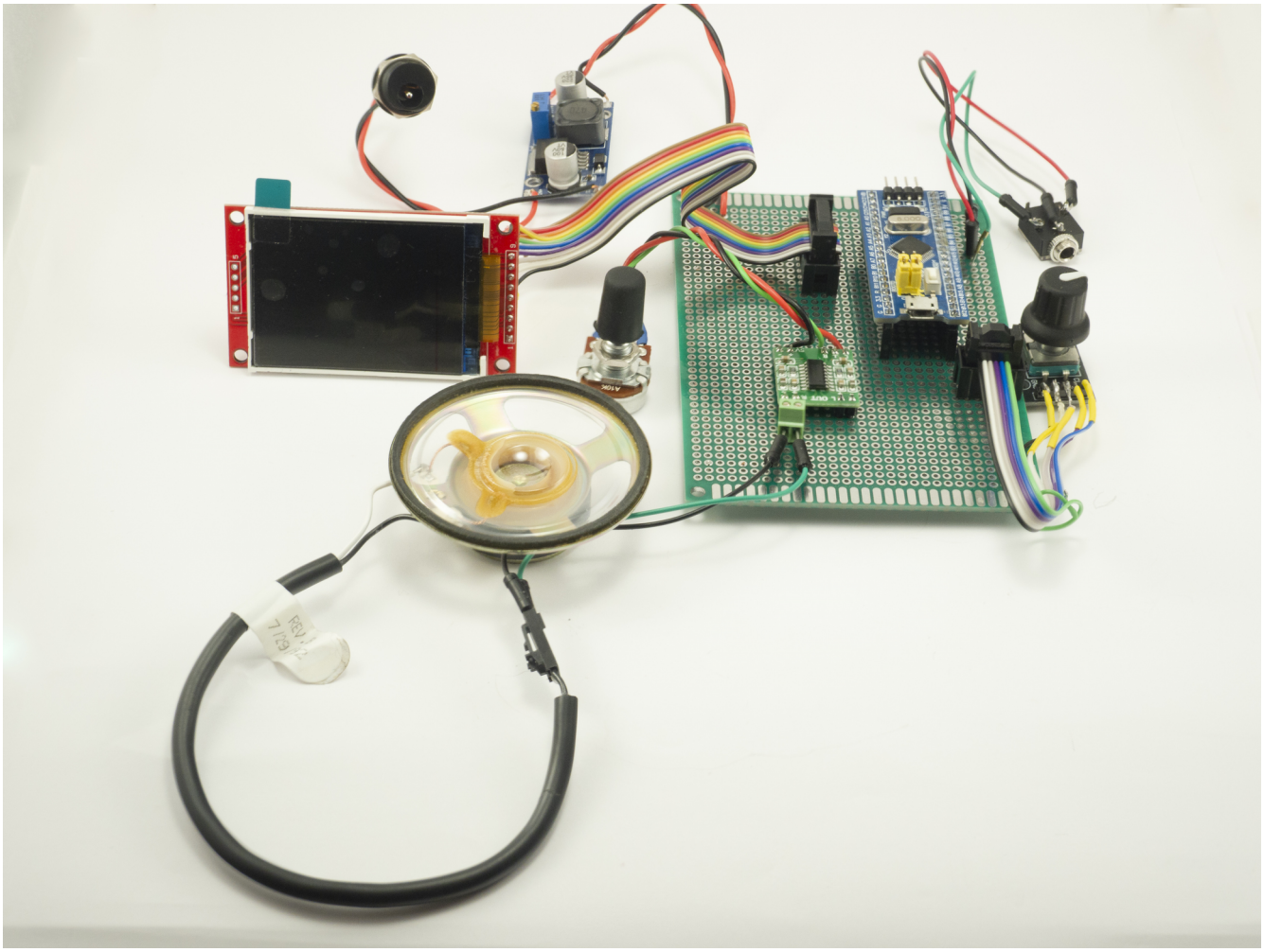

Figure 2. MCT components.

Figure 3 shows the schematic diagram for the MCT. Because the circuit is so simple, I +

used a proto board to build the MCT. The reason for using the buck converter is because I wanted to be able to tolerate fairly large swings on the input voltage, yet not burn the extra energy as heat. While you can power it from a 9V battery, I also wanted to be able to power it from a cigarette lighter socket in a car or a small battery pack.

The case is much larger than it needs to be, but I may add an onboard speaker to it in the space to the right of the display, which leaves the speaker jack for headphones. I'm also considering adding an SD card reader, too. There is an SD socket on the display, but no easy way to get to it without making the display awkward. That is, accessing the SD card would require opening the case unless I figure out a better mounting scheme. Of course, external SD card readers are cheap so I could simply add one. In

short, the MCT is a work in progress.

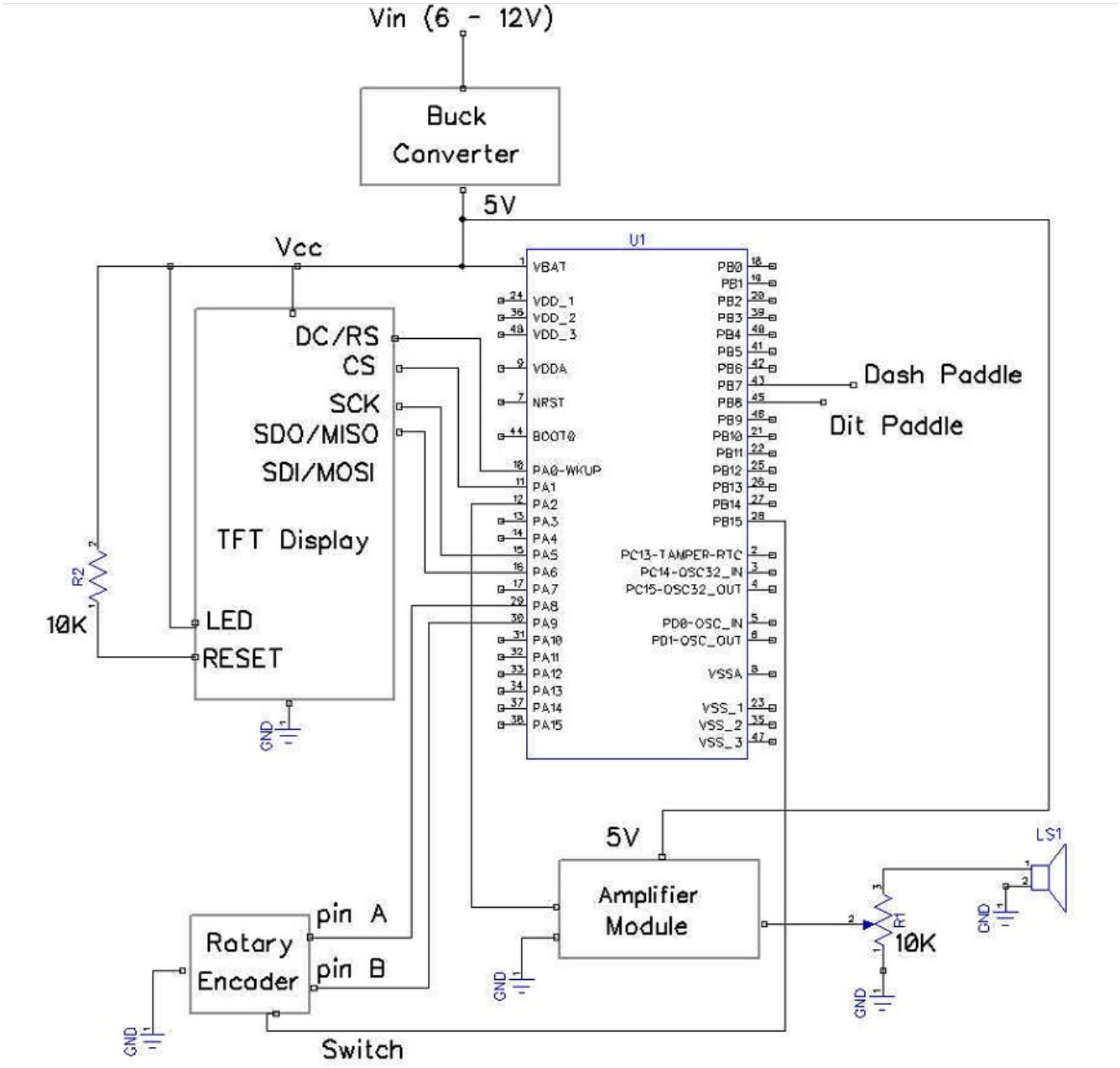

Figure 3. Schematic diagram for MCT.

Given the simplicity of the circuit and it's relatively low cost (under $20), I'm hopeful that some of you will do a build, or perhaps a club build, and get people even more involved with CW.

Morse Code Tutor Feature Set

I'm going to explain the MCT feature set within the framework of the MCT Menuing system. Figure 4 shows the entire menu system at once while Figure 1 shows only the *Receive* submenu options.

Receive	Send	Config	(Main Menu)
Letters	By Two's	Speed	**(Sub-Menus)**
Numbers	Mix	Encoding	
Punc	CopyCat	Tone	
Words	Flashcard	Dit Pad	
Let-Num	Exit	Save	
Call Sign		Exit	
QSO			
Exit			

Figure 4. The Morse Tutor Menuing System

The main menu appears at the top of the page and always uses horizontal scrolling. This means that turning the encoder clockwise while on the *Receive* option would advance the active menu choice to *Send*. The menu “wraps” in either direction (i.e., turning CCW on *Receive* goes to *Config* and CW on *Config* would activate *Receive*.)

Receive Submenu

First, unlike the main menu, all submenu options are vertically oriented. Turning the encoder clockwise advances to the next submenu option listed below it. If you are on the last submenu option in the list (i.e., *Exit*) and turn the encoder CW, the cursor wraps around to the first submenu option. Likewise, if you are on the first submenu option (i.e., *Letters*) and rotate the encoder CCW, you wrap around to the last submenu option (i.e., *Exit*). All submenus operate in this manner.

The *Letters*, *Numbers*, *Punc*(tuation), and *Words* options send random sequences of that option at the currently-set words per minute. If Farnsworth encoding is active, the code inserts an appropriate Farnsworth delay into the stream at the appropriate time (e.g., word or letter end). *Let-Num* is simply sends a random collection of letters and numbers. *Call Sign* sends only randomly-generated call signs. We did this as call signs seem to be a stumbling block in using Morse code in a QSO, probably because it mixes letters and numbers.

The *QSO* option generates what might be called a “typical” sequence of words that you might encounter in a CW contact. If you look near the top of the MCT's *ino* source code file, you will see a number of arrays that are pointers to character arrays. These arrays contain subject-related strings, such as:

```
char *rigNumbers[] = {"7851", "7700", "7600", "7410", "7300", "7100", "718", "78", "9100",// ICOM
                      "990", "480", "2000",                                               // Kenwood
                      "991", "891", "1200", "3000", "2000",                               // Yaesu
                      "1500", "6300", "6400", "6500", "6700"                              // Flex
                         };
```

By using a pseudo-random number generator to generate an array index number, we can make each practice QSO a little different. There are other subject-strings for names, cities, common QSO words, antenna, weather forecast, and a random call sign generator, too. This helps keep the practices varied and to prevent the user from “memorizing” the content of a practice QSO.

Again, looking in the *ino* source code file, you will see other arrays that hold state abbreviations, ARRL section divisions, and Sweepstakes class divisions. We have not used them per se, but they are there if you want to modify the code to use them. (If you are modifying my source code and don't want to use

these, comment them out, which will save you some memory space.)

Practice

Of course, the best way to practice is to listen to CW on the air. Really? Why not just use the MCT for practice? Well, several reasons. First, the MCT sends almost perfect code using the ideal spacing ratios between code atoms, characters, and words. Few operators send perfect code, so practicing with different “fists” makes that type of practice just that much more useful. Second, we can't include all of the words that might pop up in a QSO. The subject matter of QSO's ranges all over the spectrum, and we can't possibly include that depth of word variety. Also, because a QSO is a “real” conversation, it will be more interesting than the random QSO we will generate.

Each submenu ends with an *Exit* option. Selecting the *Exit* option sends program control back to the main menu.

Send Submenu

You're going to find that sending Morse comes quicker and is much easier than receiving. That's why there are relatively few *Send* submenu options. The *By Two's* option is the starting point for using the Koch method of learning Morse. It sends only letters. We encourage you to activate the Farnsworth delay, too, as it makes for a more successful start than using the normal code spacing. You can activate these options using the *Config* menu, setting the Koch target rate (e.g., 30wpm) and implementing a Farnsworth delay (e.g., 10wpm). Our experience is that, if you don't experience some initial success, you will get discouraged and give up. Not good.

The *Mix* option is similar to the *ByTwo's*, except digit characters are also sent. The speed and spacing depends on how you set your *Config* options.

I really like the *CopyCat* submenu option. As it is currently coded, it generates a random call sign and sends it at your target speed. (We use call signs because beginners have a little more difficulty with a mixture of letters and digit characters.) The code then waits for you to send it back to the MCT. If you do not send it correctly, a question mark appears on the display and the same sign is sent again and the program awaits your response. The MCT waits in this pattern until you get it correct, at which time it generates a new call sign. You can terminate the sequence by pressing the encoder switch. Because we use encoders with built-in switches, terminating the sequence simply requires pressing the encoder knob to activate its switch. You could, of course, substitute an off-encoder NO, SPST, switch for the same purpose if you wish. Also, we decided to use polling for the MCT rather than interrupts. There's no real reason for this, it just sorta happened.

Other Learning Tools

I was a professor for forty years and learned that one approach to teaching a concept might yield an ah-ha moment for one student and a deer-in-the-headlights for another. So, it rarely hurts to attack a problem from multiple sides. Something I call the Flashcard approach is one such attempt. The *Flash* option was suggested to me by Joseph Street and I don't know of an app that is using this approach to learn Morse code. Essential, the *Flash* approach uses an electronic equivalent of flash cards to learn code. That is, you hear the letter at high speed (again, it's all about rhythm) and, after the Farnsworth delay, it presents an image of that character on the display. The image is much larger that normal text so it does take on a flash-card feeling. The image of the character remains on the display for FLASHCARDVIEWDELAY milliseconds. (You can change this delay easily by editing it in the

MorseTutor.h header file.)

I added the flashcard option because everyone learns in slightly different ways, so if this works for you, great. If not...well, don't choose it! Like they say, if the only tool you have is a hammer, don't be surprised if all your problems look like a nail. The *Flash* approach is just another tool to hang on your belt if you choose to do so. As it currently is configured, the MCT only sends call signs using the *Flash* option. It would be useful to extend this to include commonly-used words in a QSO. The code includes the top 100 most commonly-used in normal conversation. Feel free to change these to suit your needs.

As I said, there are probably hundreds of different ways to learn Morse code. One I just became aware of by a friend Dr. John Weiner, AB8O, uses mnemonic images to learn, as see in Figure 5. I'm not sure about using such mnemonics because it, once again, forces you to visual dits and dahs. Still, it might work well as a starting point for some people.

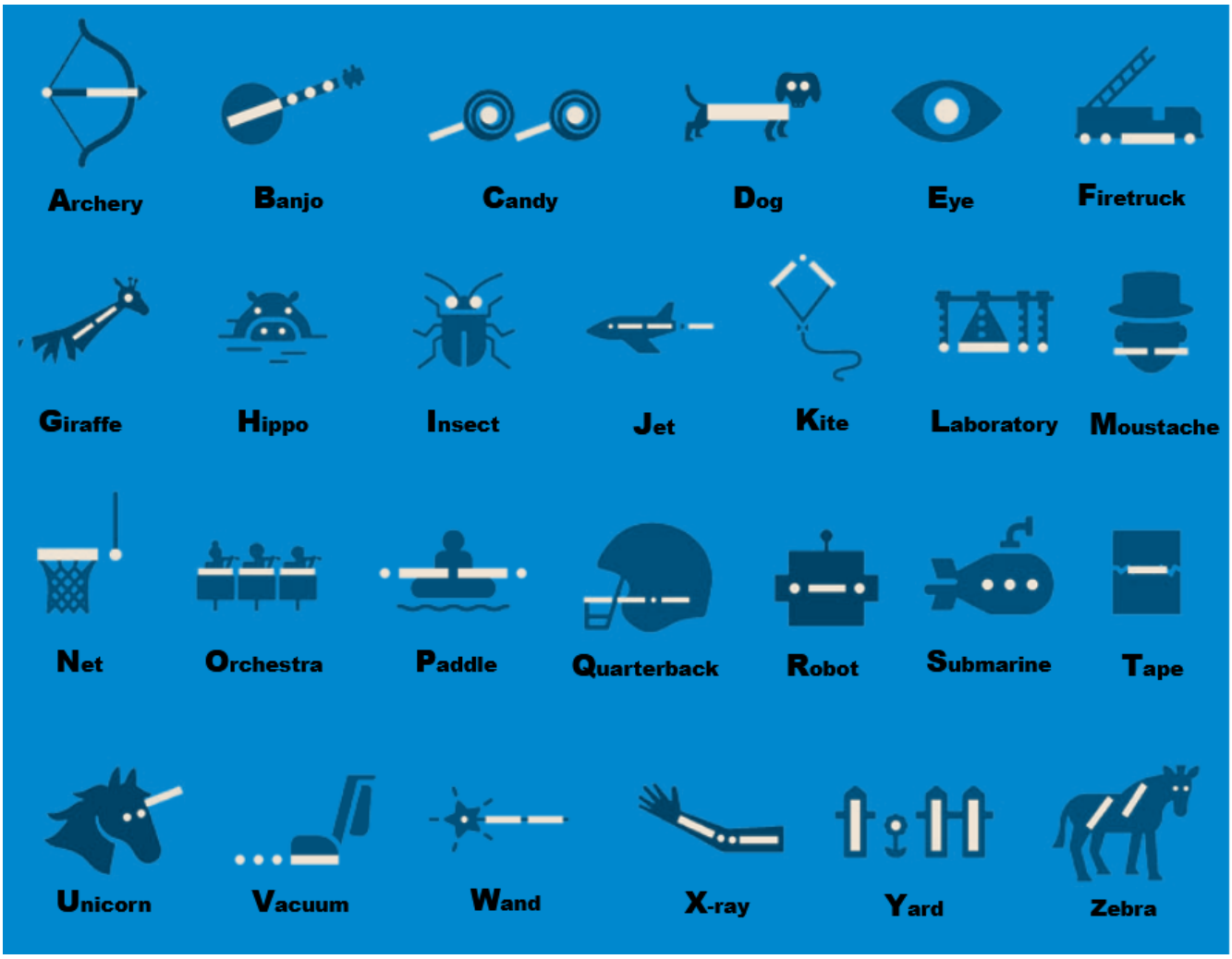

Figure 5. Mnemonics for Morse code.

Config Submenu

This option allows you to set program-wide preferences that can remain in effect even after power is removed from the MCT.

The *Speed* option allows you to set the base words per minute (wpm) that is used for practice sessions.

You should increase this whenever you feel “comfortable” at the current wpm. You want to continually push yourself towards higher speeds. The goal is for you to reach “eyelid speed”. If you plan on using the Koch/Farnsworth methods, I'd like to see you be ambitious and set the wpm parameter fairly high (e.g., 25-35wpm) from the outset.

The *Encoding* option is used to activate/deactivate the Farnsworth encoding scheme. This choice plays a role in how the practice code is sent to you in the *Receive* mode. This option lets you set both the Koch (target) speed (e.g., a goal speed of 30wpm) and the Farnsworth “gap” speed (e.g., perhaps 5wpm if you're just starting out). In the program source code, you will see the goal speed stored in a variable named *targetSpeed* while the gap speed is stored in *farnsworthDelay*. Here again, you should continually advance the *farnsworthDelay* towards faster speeds to push yourself. Your goal is to get the *farnsworthDelay* speed to match the target speed so they are both occurring at the same speed, thus making the gap speed zero.

The *Tone* submenu option simply lets you adjust the audio note used as a sidetone when sending Morse. Most CW operators seem to cluster around 700Hz, but it's a highly personal choice. Once this option is selected, rotating the encoder changes the tone. The tone changes in real time as you turn the encoder. Turning CCW lowers the tone and CW raises it. When you've found one that sounds good to you, simply press the encoder shaft switch to store it in EEPROM. (See note on EEPROM later in this chapter.)

The *Dit Pad* submenu option lets you alter the paddle lever that is used to send a stream of Morse code dits. Yes, we do assume you want to use a paddle, as sending good code with a straight key at more than 20 wpm is not easy. When you're two years younger than dirt, it gets even more difficult. You can also change the dit paddle by changing its assignment in MorseTutor.h header file. Look for these two symbolic constants around line 55 in the header file:

```
#define DASHPADDLE          PB7      // tip
#define DITPADDLE           PA8      // ring
```

and change as needed. You could also reverse the BP pin wiring to accommodate your paddle preference. Doing that might be better so your code remains the same as everyone else's. However, it's a lot easier just to change the symbolic constants.

If you're new to programming, many of the pins on the BP can be used in different ways depending upon their initialization. Figure 6 shows the generic pin assignments. Why the term “generic” pin

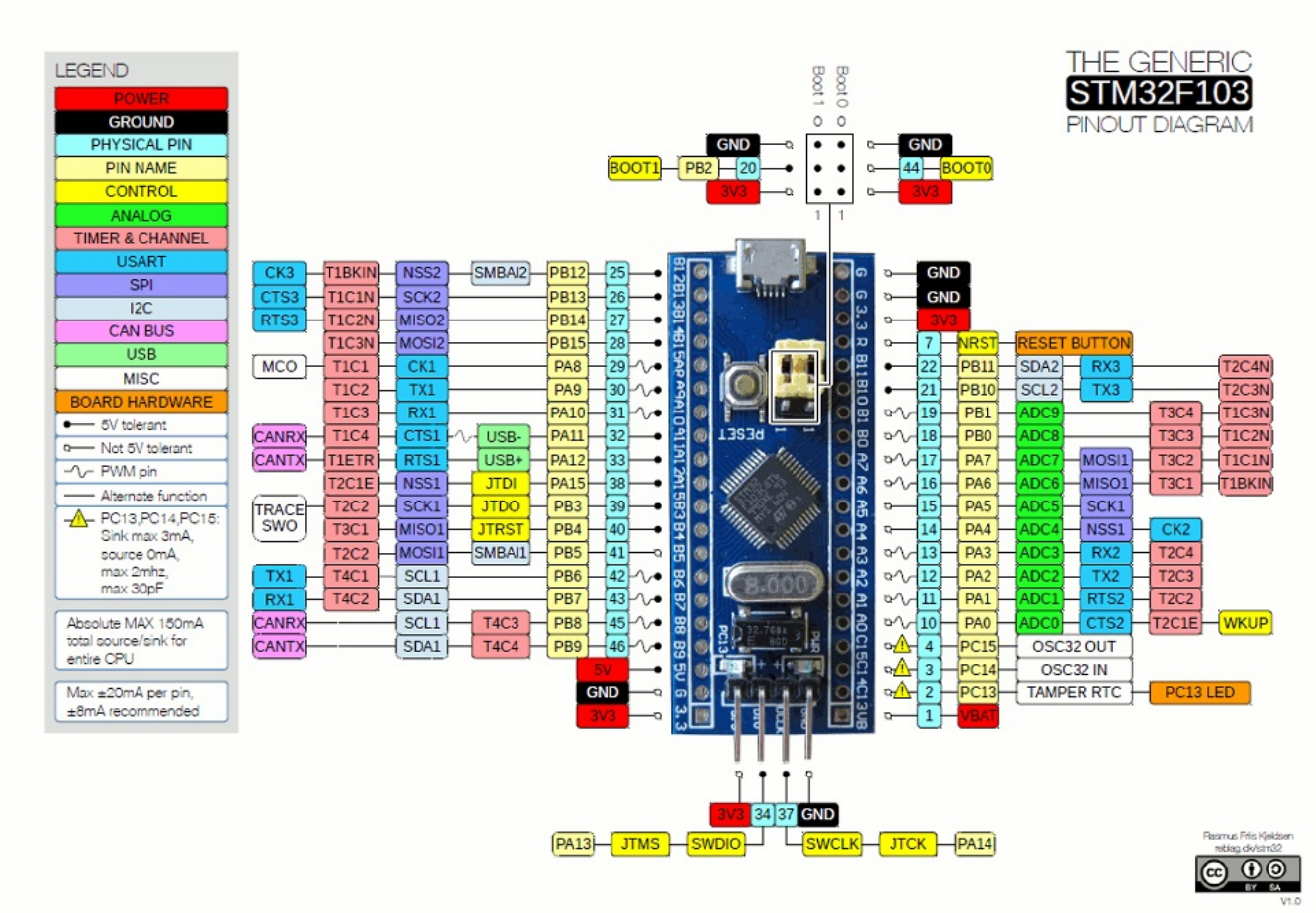

Figure 6. Generic STM32F103 pin assignments

assignments? The reason is because each company that uses the basic chip is free to “dedicate” a given pin to a specific use. For example, I tried to use pin PB9 as a standard I/O pin with one BP and was told it doesn't exist. Well, what really happened is the company producing that version dedicated the pin for another use. Just be forewarned that, if you get a “pin undefined” but it shows up in Figure 6 with a symbolic constant, your BP has likely earmarked that pin for some specific use. Pick another pin...

STM32F103 EEPROM

The *Save* submenu option is used to write the setup information in the *Config* menu option to EEPROM...well...sort of. For this project, we selected the STM32F103 microcontroller when others could have been used. We made this selection mainly for the memory resources, size, and cost reasons. The Arduino family of controllers simply do not have the SRAM memory for all of the strings that are used in the program. The Arduino Mega2560 has enough flash memory, but is still pretty skimpy on SRAM. Clock speed wasn't really an issue, since human reaction times are glacial to these microcontrollers. I really like the Teensy, but at $30 each, it's overkill. Given the relative price points and the resource bases, the choice was a no brainer.

One downside of the STM32F103, however, is that it does not have any EEPROM on board. Actually, neither does the Teensy. Both get around this limitation by emulating EEPROM in the flash memory that is available. The Teensy 3.6 fixes the EEPROM to 4096 bytes and sets the memory pages accordingly. Because the EEPROM memory space is emulated in flash memory, EEPROM writes on the Teensy 3.6 cannot occur at clock speeds in excess of 120MHz. Because the Teensy's native clock speed is 180MHz, the processor slows to the lower clock speed during EEPROM writes. While this is

normally not an issue, it could be if you wanted to do some form of high speed data logging.

The STM32F103 is a little more flexible in that it allows you to determine how much EEPROM space you need. Flash memory starts at 0x0800 0000 and extends upward for 128K. The first thing the program must do is define where the EEPROM is to reside in flash memory. I wrote a function named *DefineEEPROMPage()* to do this. Because the EEPROM demands are very small in this application, we defined only a small 1K block for use as program EEPROM.

<Sidebar Start>

Note: You do *not* want to use EEPROM for data that is frequently changed. Instead, use EEPROM for storing data that does not change that often, like the configuration data we store in EEPROM. The reason is because flash memory (hence, our EEPROM) has a finite write cycle of about 10,000 writes. While that may sound like a lot, it isn't in many applications (e.g., a data logging).

If you want to use the STM32F103 for data logging, consider adding an SD read/write module. They are cheap and can be used to store very large data sets. If that's not fast enough, consider adding “real” EEPROM externally to the BP or Teensy.

With respect to the MCT, an interesting add on would be for you to add an SD card to the program, store *Gone With The Wind* on it in ASCII, and then have it play back the book in Morse. Might help pass the time on those long commutes!

<Sidebar End>

Powering the MCT

The STM32F103 series is a 3.3V device, but the I/O pins are 5V tolerant. In addition, there are 5V and 3.3V regulators on the BP board so you can power the board from either voltage source. However, it's probably a good idea to power the small audio amplifier and the TFT display from an external 5V source, and not draw the 5V from the BP. For that reason, we are adding a buck converter to the circuit so that you can feed it anything from about 6V to 15V and the buck converter will output the 5V needed for the display and also the BP. (The actual input voltage could be larger, depending on the buck converter.)

You have a bunch of choices for the buck converter. Two basic types are shown in Figure 7. The BP

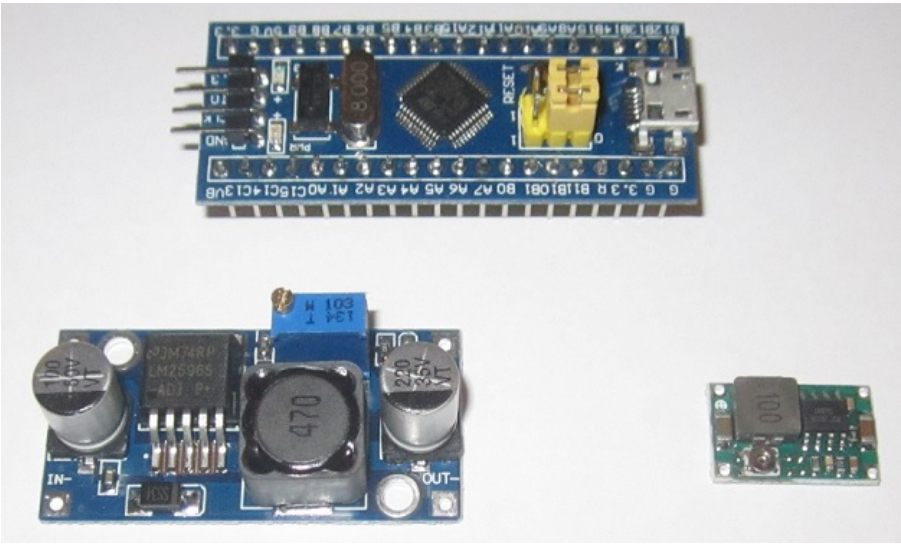

Figure 7. Comparative sizes of two buck converters relative to the BP.

is at the top of Figure 7 and two buck converters are on the bottom. Both have input voltage ranges from about 3.8V to 35V, but the one on the left can handle about 3A of current while the one on the right is limited to about half that level. The cost of the one on the left is under $1.50 and the other is about half that price. Both can handle the project load, but see that little Phillips head screw on the low priced spread? That screw is used to adjust the output voltage and has a tendency to pop off if you look at it wrong. In this case, go big and spring for the more expensive converter.

While the BP has enough power to drive a set of headphones, we added a small audio amp to the project. The audio amp shown on the left in Figure 8 is based on the cockroach of audio amps, the LM386, draws about 2mA at idle, and costs $0.92 online. The audio amp in the middle is more expensive ($0.99), but has a potentiometer that makes it easy to adjust. The amp on the right is the

Figure 8. Small audio amps.

one we selected. It also costs about $1 in small quantities and is more than adequate for the task.

The TFT color display we are using is a 2.2”, 240x320 pixel display and uses the 4-wire SPI interface. It uses the ILI9341 driver chip, for which there are a number of graphics libraries that can be used. We are using the Adafruit ILI9341 library for the graphics routines, which are primarily just textual for the most part. Adafruit makes good products and provides useful software/libraries free. We encourage you to support them as much as you can.

We opted for a 2.2” display because we wanted to keep the project fairly small. There's nothing to keep you from using a larger display other that cost, case size, and perhaps a little more power consumption. Figure 9 shows the backside of our display. We have not utilized the SD card reader that is provided on the back of the display. If you elect to use the SD card, there are plenty of resources you can use to guide you. Simply do an internet search on “using Display SD reader”. The unused pins for the SD card are on the right side of the display in Figure 9.

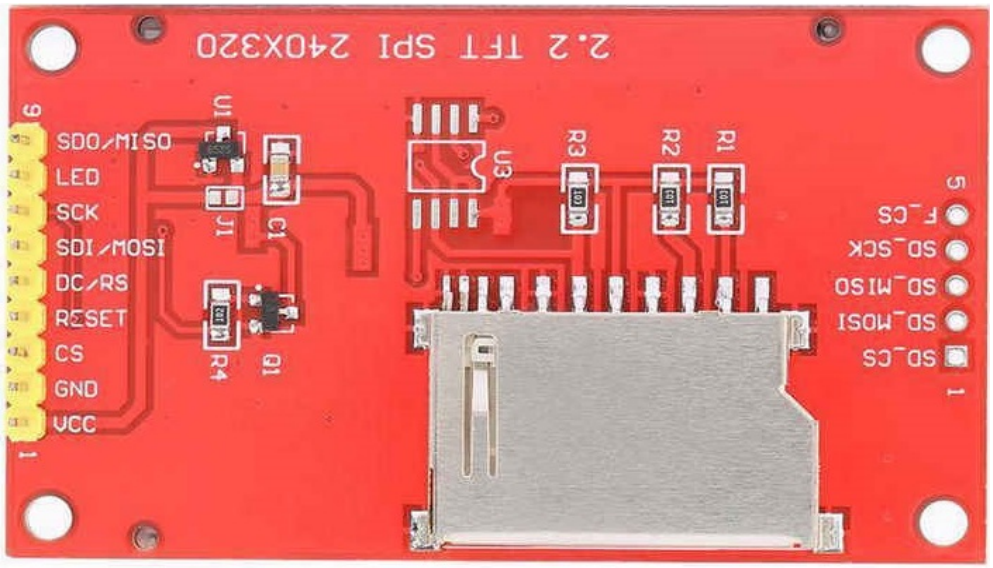

Figure 9. Backside of TFT display.

Table 1 shows the connections between the TFT display pins shown on the left side of the display in Figure 9 and the BP pins.

Table 1. TFT display to BP

TFT display	SDO/MISO	LED	SCK	SDI/MOSI	DC/RS	RESET	CS	GND	Vcc
Blue Pill	A6	N/C*	A5	A7	A0	N/C*	A1	GND	5V

**The LED and RESET pins connects to the Vcc pin. The LED connections is through a 10K resistor. The two pins do not connect directly to the BP.*

Note: The BP connections refer to the pins as labeled on the BP board. (See Figure 6.) Keep in mind that BP images usually use legends instead of the board labels. For example, A5 on the BP board appears several ways in an image: 1) physical pin number (e.g., 16), pin name (e.g., PA5), analog name (e.g., ADC5), SPI name (i.e., SCK1), and sometimes other interface names (e.g., I2C, CAN Bus, USB, etc.) As we said earlier, there are a bunch of ways to use/refer to the same pin. Other books and articles may use some of the other pin names based on the context in which they are referenced. We tend to stick with the pin names that are painted on the board itself. That way you won't need to reference a printed image of the various pin names.

The Software

The software for the MCT is available from our web site, hamradiodesigns.com. Some of you may be puzzled when you download the source code because the ZIP file contains nine source code files. One file is the usual Arduino IDE *.ino file, which always must contain the *setup()* and *loop()* functions. However, there are also two header files: one for the overall project (MorseTutor.h) and one for the menuing system I wrote (Menu.h). That leaves the other six files, which are all *.cpp (i.e., C++) files. True, you could lump all the code into a single *.ino file, but doing so throws away a lot of advantages that multiple files bring to the table.

First, each file has its own tab in the Arduino IDE. This makes it easier to navigate around the project. Using meaningful file names (e.g., Menu.cpp, ProcessCode.cpp, RotaryCode.cpp) makes it easier to test/debug the code. The first tab in the Arduino IDE is always the *.ino file; makes sense, since that's

where program execution begins. All of the remaining files are in alphabetic order; again, making it easy to find stuff.

Second, using separate files allows the compiler to perform “incremental compiles”. For example, suppose you're having problems with one file and you keep changing it, but the other files in the project don't get changed between compiles. With incremental compiling, the Arduino compiler “keeps” an image of the unchanged files and merges them with the newly-compiled file you're working on. While this may not mean much on small files, I've been involved in a project with over 11,000 lines of code spread out over 19 source code files. I figure I save about 20 seconds (on a *very* fast machine) per compile by doing incremental compiles. That doesn't sound like much, but when you do 50 compiles/day, it adds up to more than a half hour saved each day. Given I've been working on the project almost every day for 18 months, or almost 250 *hours* saved. And that's just the time saving on compiles. Add saved debugging time and...who knows how much time I've saved.

Third, my assessment is that the GNU C++ compiler, which is what the Arduino IDE uses, does a better job of type-checking when the files are split up. The compiler seems to catch more mismatched function parameters than when a single file is used.

Finally, I think it actually makes debugging easier. Multiple files makes me more aware of the dangers of using global data in a program. Global data are defined outside of any function or class. As such, it means that every line of code in your program has access and the ability to change that data. To me, it's like placing a hooker at the top of your program and then giving every statement in that program a $100 bill. When a bug is born, where do you start looking for the father of the bug? By defining all data within the confines of a function or method, at least you know where to start looking.

Conclusion

While I know I'm preaching to the choir here, my goal for the MCT is to get more hams interested in using CW. There are a lot of options that can be added to the MCT. For example, use the *Flash* example as a starting point, but have the user send the letter they just heard back before the letter is displayed. *Flash* could also be extended to words, too. Another improvement would be to sense the students success rate when using the *Flash* option, and increase the number of letters sent each time. If the user falters, back down to fewer letters. I'm sure you can think of other extension, too.

Learning Morse code is not an easy task and anything we can do to lower the barriers is a good thing. If any of you have ideas of how I might move people off dead center, please let me know.