**A Python Powered ESP32 Based CW Keyer**

This article describes a single chip iambic keyer with remote terminal wifi access, pre-programmed messages for contesting, a display screen, and built in code practice capability. The project can be easily be built in a weekend and costs less than $20. All the software will be provided and available at the ARRL website.

In addition describing how to build and operate the keyer, the article will also describe how the project was designed implemented in Python and how it might be a good candidate for your next ham radio project.

**The Hardware**

The hardware (and software) for this project has been designed with reusability in mind. Depending on your desires and interests you can add the components you need. Lets start with the basics and add pieces.

Figure 1 shows the most basic configuration, the ESP32, a sonic sounder, and power supply. With this configuration, you will be able to access the keyer from your wifi network. This will provide a CW terminal with built in canned messages, random code practice sequences. From the terminal you can control the sending speed and sidetone volume. This setup might be useful if you are just learning the code and want a practice mechanism.

Figure 2 shows the addition of a key, either be straight key or iambic type . This arrangement would be useful for CW contesting, where code from the key can be interspersed with keyboard input.

In figure 3, we add a keying circuit to interface with our transmitter. Several options are given for a keying circuit so you can choose the one that best matches your transmitter.

And finally, in figure 4 shows how some other options you might want to add to the keyer

1. we a speed potentiometer to control the speed from the unit, rather than using the remote terminal
2. Push buttons to interspurse pre-canned messages, rather than using the remote terminal. These first 2 options make the keyer more like a traditional iambic keyer.
3. A TFT display to show status info for the keyer

Table list all the components and where I purchased them.

This next section will describe the software design. If you are not interested in learning about Python, you could just jump to the construction section and skip this part of the article. But I would encourage you to “take a look under the hood” at Python and see if it might be useful for your next ham radio project.

**The Software… Why Python?? And Why Not Arduino?**

Back in the 60s when I started experimenting in Ham Radio it was not necessary to learn any digital techniques. Ham radio was solely about designing RF circuits and designing antenna. Filters were large combinations of capacitors, inductors, and resistors. Power supplies were in the hundreds of volts. Any displays were mechanical meters are dial arrangements.

But all that has changed. Ham radio now consists of equipment built around software defined radios, digital signal processing, and internet connectivity. Any user input and output is via liquid crystal displays and touch screens. Any serious Ham radio experimenter will need have some programming skills to comprehend and built upon the body of knowledge.

The introduction of the Arduino in ? has shown up in countless ham radio projects in the last few years. It has provided a great C language platform to generate and process signals to control our ham radio projects.

So why make the shift to Python? C language has been with us for a long time. I first started programming in C in 1975. Since that time it has been enhanced but is still missing some the advanced primitives seen in modern languages. Python offers these advantages over C that can greatly simplify ham radio project:

* It does not require a separate compiler or integrated development environment. Code is edited by a standard editor (I use Gleany). The text is copied to the target’s file system which is maintained inherently by Python.
* It is extremely portable and can easily be moved from machine to machine.
* It is interpreted rather than compiled. The code is interpreted during run time and converted to byte codes that can be executed on .
* It is easier to debug. line by line debugging capability is provided by an REPL.
* It has enhanced error handling capability. Operations and be tried (using the try and except) and errors handled or raised to another level.
* It has some very powerful data structures that simplify complex processing(for example, dictionary lookup).
* It has powerful inherent flow control mechanisms. Threading and asynchronous I/O are built into the Python.

The Python interpreter now comes standard on most computing systems. Is available on Windows, Linux, and microprocessor systems. For this article, we are going to use a microcontroller called the ESP32 which comes it micropython installed. A description of capabilities is shown in the inset.

**The Software Architecture**

In designing software with Python (or even in Arduino C) it useful to think of your project in terms or its objects. The code to implement each object can then be created and tested as a separate entity, called a class. This provides the possibility of reusing the software on your next ham radio project. Python excels at making it easy to create classes and test them using the REPL.

So, what are the “objects” in our keyer? There are several ways to abstract the keyer. Here is the objects I have chosen:

* A I/O pin class (called Pin) which knows how to interface with the ESP32 input and output pins. Pin is provided for us in microPython so we can use it directly.
* A keyer class, which has methods and attributes to generate signals based on which of the paddles of the key are depressed.
* A transmitter class, which knows how to key the transmitter and generate the side tone
* A remote terminal class, which knows how to accept connections from the internet and process commands. This class reuses the WIFI class, which is provided by micropython.
* A cwConverter that accepts accepts from the remote terminal and translates the ascii to CW
* A Button class, which is triggered when a button is pressed and inserts a predefined CW message. The button class uses an instance of the Pin class to get I/O input.
* A SpeedController class, which reads a potentiometer and sets the CW speed. This class makes use of the Analog to Digital class (ADC)

Each of these classes can be written separately and easily tested with Python using the REPL. So next step is to actually build our keyer.

**Construction of the Keyer**

Figure x shows all the parts for the keyer. Note that the ESP32 D32 Pro comes with pin headers to be soldered to the board. I personally prefer using these and then soldering them to a perf board, as opposed to soldering directly to the ESP32. You can then as many of the optional components as you desire. Figure x shows a basic unit for remote keyboard access. Figure y shows a unit I built with keyer, message buttons, and display.

**Loading the Software**

Your ESP32 should come pre-loaded with micropython. To make sure your unit is working, connect it to your PC with a usb cable. Then open up a terminal program (I use PTTY) and hit the return key. You should see message like the one shown in figure z. You are now in the Python REPL and could start programming.

But we want to load all of our software files without retyping. So we just need to copy the files from the PC to the Python filesystem on the ESP32. There are several ways of doing this but I prefer using a tool call RSHELL which is available here. But before we do the actual load, we need to set some tuneable parameters, (viz wifi information, stored canned messages, etc). Here are the steps:

1. Extract all the software into a directory on your PC
2. Open the tuneableParameters.py file in the text editor of your choice. I used Gleany since it especially designed for programmers.
3. Edit the following fields:
   1. If you are using Wifi, put in your network name and password. If you don’t want wifi access, leave the ssid field as null. If you want a password for access to your terminal, edit the remotepassword field, otherwise leave it as null.
   2. Edit the pre-canned messages to fit your needs.
4. Open RSHELL in the directory where your program files are loaded and type the following commands:
   1. Connect serial yourPortNo
   2. Cp -r \*

**Powering UP Your Unit**

If the load was successful, you are ready to use your keyer. Let start it up. First type REPL to connect you into the ESP32. Then type a control-c which will reboot the keyer. You will get some status messages printed from the REPL(and in the display, if provided) that the Wifi was connected. The blue led on the D32 will also light. The messages will also include the wifi address that has been assigned to your unit by your router.

If you plan on using the remote terminal, lets test it next. Open a terminal program on the PC (or tablet). There are lots of choices. I prefer PUTTY which is available on many platforms. For whatever terminal you use, set the following options:

1. Set the wifi address and port in the options.
2. Also set the mode to RAW, and
3. turn on local line editing. We want the terminal to send one character at a time.

Figure x shows the options I use for PUTTY. After you have set the options ,save them so you won’t have to type in them again. Hit connect and you should see a screen on your PC as shown in figure zz.

Now that everything has been tested, you can disconnect the PC connection and plug into a direct powersupply. After hitting the reset button, your keyer is all ready to go. You can set the speed and sidetone volume directly from the terminal using the (ctrl+s) and (ctrl+v) options.

**Using the Keyer For CW Practice**

If you are just learning CW, you might want to use the practice option (ctrl+p) option. It sends random groups of characters for your practice session.

You might want to also use the keyer for practice with a friend. To do this, build 2 units and share your IP and port number with your friend (see figure aaa). Open a PUTTY terminal and connect to your friends unit. Now everything you type will come out as CW on the other unit. It is a good way to practice actual QSOs before you actually get your ham ticket. This will work if your friend is on the same wifi network but could also work across the internet with port forwarding. There are security concerns with this arrangement (see below).

**Keying your Transmitter**

Providing the proper interface between your keyer and your transmitter and your newly built keyer is not only important to the longevity of your ESP32 but also to your safety as an operator. Is is important to be able to key the transmitter but also provide good electrical isolation. We are dealing with 3.3 volt logic in our keyer and are connecting it to transmitters that can potentially produce hundreds of volts. High SWR currents or high plate voltage from older equipment can quickly wipe out electronics and give you a shock. How much isolation you will need will depend on the transmitter you are keying.

Figure hh shows 3 possible keying arrangements that might be used. The circuit used in (a) is commonly used and might be sufficient for new rigs with electronic switching. Figures (b) and (c) are more robust. The relay switching in figure c provides the best isolation but does introduce about 10 msec of delay. Depending on the transmitter, this could be significant at higher CW speeds.

So with all that as background… lets create our keyer. I have found it useful on ham projects to create it in small pieces, test each piece, and then integrate them together in the final product.

**The first steps…. Connect Your Controller and Using the REPL, test the input/output pins**

Micropython has a built in debugger known as the REPL, which stands for ????. As I mentioned, Python is an interpreted language so it possible to write a line of code and execute just that line of code. For our first lets hook up a piezo buzzer as shown in fig x. We will then:

* connect the esp32 to a USB connection on our computer
* Open a terminal window and connect it to the USB port. There are several options for this, I use PUTTY(1). This will power the ESP32
* Hit Control C which should connect you to the REPL and you should see this…

Now type the following:

Import Pin

buzzer = Pin(5,Output,0)

You are now ready to control the buzzer from the REPL

Typing

Buzzer.value(1) will sound the buzzer buzzer.value(0) will silence the buzzer. Give it a try.

Now… out key will need some input from our key. Type in the following

Key = Pin(4,INPUT,1)

Now we can test the input and use it in our program.

Type in the following in the REPL

Key

So we can easily get the value of the key.

How about analog input? That is easy in python also. We will need this for a speed control for

Our keyer. Connect a 10k potentiometer as shown in ?.

Then, type the following command into the repl

Potentiometer = anolog read

Print(pot)

Now, we can put this all together to build a simple iambic keyer.

Now change the pot and type Print(pot) again. The value shown should change

**Generate some decent quality audio**

**Lets do some threading**

**Lets add a Touch Display**

**Using the DACs**

**Build an IAMBIC keyer**

**Using the File System – A voice CQ caller**

**Add some Networking -**

There are many great ways to learn Morse code. There are apps to be purchased, on -line sites, and on the air broadcast like those provided by the league. These can be great to get you started but at sometime everyone need to transition to the real world of making on the air contacts. It takes a while to learn the techniques of listening on a frequence, responding and ?.

This CW simulator described in this article provides a real life chance for several operators to operate cw just like a real world. You can make contacts and use all the appropriate Q signals, long before you have received that license.

The project also provides a simple construction project that provides a great introduction to a common microcontroller and the Python programming language.

Project Description

A diagram of the simulator is show in figure 1. It operates on a local WiFi network and has 1) one access node and 2) a station node for each CW operator on the network. The access node is only used to initially set up the connections for the station nodes. The station nodes have a key and speaker connected so the operator can send and receive CW signals that are heard simultaneously by all the other stations. This simulates an environment just like a real world environment where stations need to develop skills to …

The simulator does all the communication on the 2.4 GHz wifi frequencies but operates on a separate subnet to insure separation from other local wifi traffic. The operation of network is fairly simple:

1. Upon initiation, the station mode stations are given an address by the access node.
2. Every time interval, call it T, the station node takes a sample of the operator’s key (either up or down) and broadcasts that state to the all the nodes on the subnet.
3. Each station nodes receives all of these broadcast messages. If any of the received messages indicate that key was down, a tone is played on the speaker.

So you might be asking, what should the value be for T ? ie how often does the station node need to sample the CW key to capture in dots and dashes. To answer that question, one needs to look at the timing of a CW signal.

A diagram of a typical CW signal is shown in figure 2. It consists of a dash timing that is 3 times longer than the dot timing. If we are taking samples and want to capture all the information, we need to sample fast enough to not miss any transitions. The shortest interval for this would be a nodes sending all dots or the letter ‘e’. It turns out for a station sending CW at 10 words/minute this is about 10 milliseconds.

To capture all the transitions at 10 words per minute, our nodes must sample much faster. It turns out there a basic law of information transfer that one would need to sample at least twice the sampled frequency to capture all the information. So, taking this all into account, the value of T has been set to 10 mseconds, which should adequately capture all the transitions.

Hardware

The microcontroller I used in this project is an ESP32. This controller is available in all kinds of configurations. The one being used here is the WEMOS D32 pro and is readily available for about $8 on the internet. This board has:

1. An extremely fast dual 240 mhz 32 bit processor
2. Built in Wifi with internal antenna
3. Built in Bluetooth capability
4. All the required circuitry for a plug in and go TFT display
5. An I2C peripheral interface
6. An SDA interface
7. Numerous input output pins
8. Ram and ROM

The Software

When I was learning code in the 1960s and experimenting with ham radio, there was no need to know any programming. While many ham projects involve the use of radio frequency design, many hams are making use of digital techniques to support the design. A novice transmitter from the 1960s was likely to make of a crystal oscillator. That same transmitter today is likely has a microcontroller to support the RF circuitry. So has become necessary for the ham operator to learn some programming skills.

Building the Project

Construction of this project is very simple and requires only a few connections. The nodes are powered by standard usb cables and power supplies. There is no additional wiring to do for the one access node…. Just load the software and go. The station nodes require the connection of a standard key and a piezo speaker. The units can be mounted in a box (non-metal) due to the wifi antenna or just placed on breadboards. The Wemos units are shipped with pin headers but they are not soldered to the board. If you are not experienced at soldering you might want to view some of the available information on soldering techniques and equipment.

Loading the Software

As I mentioned, Python does not require a separate compiler on your PC or Linux system. Most ESP32 and ESP8266 boards come pre-loaded with the Python interpreter and built in debugger (called REPL). So all that is needed is to copy the program(in straight ascii format) to the board and start it up. There are several programs available to copy the code from the PC to the board. I used one call rshell. The details can be found here.