

QEXFiles
Companion Material for
A Pocket-Sized Wi-Fi CW Trainer with Farnsworth Timing and Custom Text Playback

Bob Fontana, AK3Y

A Few Design Considerations

1. Battery Choice

I wanted the unit to be portable and battery-operated. However, the ESP8266 requires an input voltage of 3.0 – 3.6 V (3.3 V typical) and may require instantaneous peak currents of up to 500 mA or so during Wi-Fi transmissions. A series combination of AA batteries (1.5 V nominal) or AA rechargeables (NiMH 1.2 V nominal) seemed possibilities. However, directly connecting more than one AA without some form of regulation has been known to physically damage the chip.

Three AAs (either alkaline or NiMH) with a low drop out regulator (LDO), such as the HT7333 chip, would provide a simple and cheap solution. Alternatively, two AAs with a boost converter such as the MT3608 or Adafruit PowerBoost 1000 provide better efficiency, although that would be more complex and need additional board space.

A third option is the use of a single Li-ion or LiPo battery, again with an LDO regulator. This ended up being my preferred choice because of the significantly higher efficiency and the capability for high current draw — needed upon boot and Wi-Fi operations. A single 18650 (3.6 V) cell with an HT7333 LDO ended up in the design.

Typically, average current draw for the unit is around 70 mA with peaks of up to 200 mA or so during Wi-Fi access, which only occurs infrequently. A 2600 mAh Li-ion such as the Fenix Flashlight works perfectly, with an estimated work time of better than 36 hours. The Fenix Flashlight has the additional advantage of having an onboard USB charging port, obviating the need for a separate charger.

2. Problems with Starting

A known issue with powering ESP8266 modules from batteries is the potential failure to boot due to high inrush current requirements, particularly when Wi-Fi starts. As indicated above, the microcontroller can momentarily draw several hundred milliamps, which can exceed the current capability of the HT7333 LDO regulator (max 250 mA continuous). This can cause a voltage sag which leads the ESP8266 to either crash or fail to boot entirely.

The solution was actually straightforward. First, a high value (470 – 1,000 μ F) low-ESR tantalum or electrolytic capacitor was added directly across the output of the HT7333 chip. This helped buffer the inrush currents during Wi-Fi startup. Secondly, an RC delay circuit was added to the RESET pin of the ESP8266 (here, 10 k Ω to Vcc feeding RESET with a 10 μ F capacitor to ground). This prevents the chip from booting until sufficient

charge is stored in the 1,000 μF cap to handle surge current requirements. Finally, thick power and ground traces (low resistance) were used in the PCB design to further prevent unwanted voltage drops.

3. High-Fidelity Audio

While the onboard buzzer was adequate for casual use, for longer training sessions the somewhat raspy sound from the buzzer can be a bit unpleasant or even misleading for Morse practice. Thus, a high-fidelity (sinusoidal) output was also desired.

One common method of providing high-quality audio is to use the I2S (Inter-IC Sound) serial bus protocol that is available on the ESP8266. However, as the chip lacks a built-in DAC (digital-to-analog converter), to output audio you would need to add an external I2S codec chip and amplifier. As we are only looking to output a sinusoidal signal and not complex audio, a much simpler solution exists, one which also takes much less circuit board real estate.

The `tone()` function generates a square wave of a specific frequency on a specified digital output pin. Of course, a square wave is rich in odd harmonic content. By filtering out the third and higher order harmonics, a nice sinusoidal output can be generated. To accomplish this, a Sallen-Key¹ low-pass filter was used. Its design is simple, compact, and provides a well-controlled frequency response. Using two cascaded second-order Sallen-Key stages enables very effective suppression of third and higher harmonics, delivering a low distortion tone to external speakers or headphones (see **Figure 1** below). Furthermore, the design operates from a single 3.3 V rail, matching the ESP8266's supply voltage, and uses a pair of very low power op-amps within a single DIP package, very suitable for battery-powered operation.

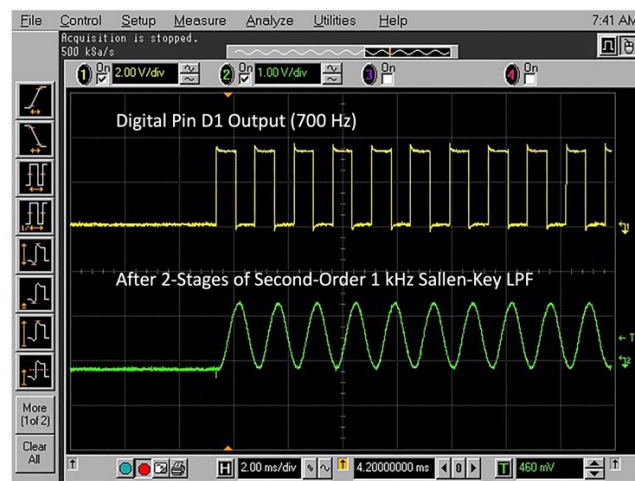


Figure 1. Result of Sallen-Key Low-pass Filtering

¹ Sallen, R. P. and E. L. Key, "A Practical Method of Designing RC Active Filters," *IRE Transactions on Circuit Theory*, Volume 2, Issue 1, March 1955.

This high-fidelity audio is output to either headphones or to an external device (e.g., audio input on a VOX-controlled foxhunt radio). Design templates are available² for designing a specific Sallen-Key filter.

Programming the ESP8266

The ESP8266 can be programmed using the Arduino *Integrated Development Environment (IDE)*. This can be downloaded from

<https://www.arduino.cc/en/software>

Versions of the *IDE* are available for Windows, Linux, and the Macintosh OS.

Once the *IDE* is installed, we need to allow it to recognize the ESP8266 NodeMCU microcontroller:

- (a) Select “File... Preferences...” and add the following line to the dialog box following “Additional boards manager URLs”:

http://arduino.esp8266.com/stable/package_esp8266com_index.json

This sets up the *IDE* to recognize the ESP8266 hardware.

- (b) Select “Tools... Board... Boards Manager...” and look for “NodeMCU.” The entry “esp8266 by ESP8266 Community” will appear. Install this code — this process may take several minutes.
- (c) Now select “Tools... Board... esp8266...” and select “NodeMCU 0.9 (ESP-12 Module).” The Arduino *IDE* will install the necessary code to manage this particular microcontroller.
- (d) You are now ready to compile and upload the code to your board. Make sure that the USB cable is attached to the ESP8266 module and that you have the USB port selected in the Arduino *IDE* (“Tools... menu”³:

Click on the compile and upload arrow (right arrow next to check mark in *IDE*).

2 E.g., <http://sim.okawa-denshi.jp/en/OPseikiLowkeisan.htm>

3 The ESP8266 uses the FTDI virtual COM port driver. If your operating system does not have this driver installed, you can download it from <http://ftdichip.com/drivers/vcp-drivers>.

Hardware Construction Notes

The actual assembly of this unit is very straightforward, requiring only minimal soldering skills. A few points, however, are worth noting:

1. The ESP8266 module can either be soldered directly to the PCB or plugged into a pair of 8-pin female headers (as shown in the article). If mounting directly to the board, a low-profile USB cable may be needed for any future reprogramming to avoid physical interference between the connector shield and the PCB.
2. The SD Card Reader module can be supported on the board solely by its 6-pin connector. However, for additional support, a small dab of hot wax glue can be placed in the gap between the front of the overhanging board and PCB.
3. The TL2285 SPDT pushbutton switch is symmetric, so its orientation on the board is unimportant (despite the asymmetrical silk screen pattern).
4. To securely solder and attach the battery holder to the board, apply a thin layer of solder to the PCB pads as well as to the top and bottom surfaces of each holder tab. Position the holder and apply heat to the top of the pad, adding a bit of extra solder as needed to form a solid connection to the PCB.
5. The SD Card *must* be formatted as FAT32. All data files should have the extension .TXT in order to be recognized by the interface.

CW Tone Frequency

The CW tone generated by the onboard buzzer is set by internal circuitry and cannot be changed. It produces a tone with a frequency of approximately 1 kHz. However, the frequency of the tone sent to the headset can be changed, and this can be done in software.

To change the tone frequency, modify line 24 in the library file Morse3.cpp. It is currently set to 700 Hz:

```
const int _morseToneFreq = 700; // Frequency of Morse analog tone (Hz)
```

Recompile with your change if desired, and upload the modified code to the processor as described above. 700 Hz was chosen as it is an offset commonly used in many transceivers.

In this design, I used a cascade of two second-order 1 kHz Sallen-Key low-pass filters. This combination actually has a -3 dB cutoff of approximately 800 Hz. I chose this tighter filter bandwidth to provide additional attenuation for the third and higher harmonics. With a 700 Hz tone frequency, the third harmonic at 2.1 kHz is attenuated by 24.6 dB. The chart below shows the calculated attenuation of the third harmonic for a given CW tone frequency.

2-Stage Sallen-Key Filter Harmonic Suppression vs CW Tone Frequency

<u>CW Tone Frequency (Hz)</u>	<u>Third Harmonic Suppression (dB)</u>
300	-4.4
400	-9.7
500	-15.6
600	-20.4
700	-24.6
800	-28.0
900	-31.7
1,000	-32.6

For phone quality audio, third harmonic suppression of at least 15 dB is desirable. So, keeping CW tones to 500 Hz or higher will work well given this Sallen-Key filter design. The most common BFO frequency range for CW decoding is 500 – 700 Hz.

Generating Practice Material

One of the biggest challenges in learning Morse is to find appropriate material to actually listen to and learn from.

“On-the-air” contacts are certainly one way, but these are often filled with abbreviations (important to eventually learn, but not necessarily conducive to learning all the letters and numbers well); usually subject to noise, fading, QRM, etc. making the learning process difficult at best; and have no way of checking that what you decoded is actually what was sent.

The ARRL website references a number of great online sources: **www.arrl.org/learning-morse-code**.

However, as this Morse Trainer was designed to be portable and operate “off-line,” it is necessary to come up with .TXT files that can aide in the learning process. Uploading excerpts of text from online or other digital sources can be useful, but again do not really create the “muscle memory” needed for long term retention.

In the mid-1930s, German psychologist Ludwig Koch developed a method for learning Morse⁴. Rather than starting with the entire Morse code alphabet or memorizing dots and dashes visually, Koch's method introduces just two characters at full target speed, typically around 20 WPM. The learner practices decoding these two characters until reaching a 90% accuracy rate. Once this level of proficiency is achieved, a third character is added. Practice continues with the three characters until accuracy stabilizes, and then a fourth is introduced. This incremental process continues until the entire alphabet, numerals, and common punctuation marks are mastered.

By introducing characters gradually and maintaining full-speed practice from the beginning, the Koch method helps learners develop instant recognition by sound, rather than relying on slow, conscious decoding. This strategy reduces the risk of developing bad habits like counting dits and dahs or relying on visual cues. The method has proven effective for radio operators, hams, and military personnel, and forms the basis of many modern Morse training programs and software tools. One of the more popular Koch-method training tools is the Windows app from G4FON⁵.

In his original paper, the specific ordering of letters to introduce to the student was left flexible and not standardized by Koch himself. Since Koch's time, various trainers and Morse programs have adopted their own character orders, generally based on (a) avoiding easily confused characters early on; (b) introducing more frequently used characters earlier; and (c) spacing out similar-sounding letters (e.g., E and I, or S and H).

I thought it would be interesting to ask an Artificial Intelligence (AI) app for help in picking out the best sequence of letters to use for the Koch method. I posed the question to *ChatGPT* and it came up with the following letter order. Its philosophy in picking this order was:

- The first few characters (K, M, R, S, U, A) are rhythmically distinct and commonly used;
- Similar patterns (e.g., E, I, S, H) are delayed to reduce early confusion; and,
- Punctuation and numbers are added only after mastering a solid base of letters.

ChatGPT indicated that it reviewed the work of both G4FON and **www.LCWO.net** ("Learn CW Online") in putting this list together.

⁴ Koch, Ludwig, "Der praktische Weg zum Hörenlernen des Morsealphabets" [The Practical Way to Learn Morse Code by Ear], *Nachrichtentechnische Zeitschrift (NTZ)*, Germany, 1936.

⁵ Morse Trainer by Ray Burlingame-Goff, G4FON (SK), **www.g4fon.net**

Suggested Koch Letter Order (Optimized for Reduced Confusion)

1. K (-.-)	20. G (-.-)
2. M (-)	21. 5 (.....)
3. R (-.-)	22. 4 (....-)
4. S (...)	23. 9 (-----)
5. U (-.-)	24. 2 (-..---)
6. A (-)	25. Q (-.-.)
7. P (-.-)	26. 3 (-...-)
8. T (-)	27. Z (-.-.)
9. L (-.-)	28. 8 (-----)
10. O (-.-)	29. B (-...)
11. W (-.-)	30. ? (-...-)
12. I (-.)	31. C (-.-.)
13. J (-.-)	32. , (-.-.-)
14. N (-.)	33. X (-.-.)
15. E (-.)	34. . (-.-.-)
16. F (-.-)	35. D (-..)
17. 0 (-----)	36. 6 (-....)
18. Y (-.-)	37. 7 (-....)
19. V (-.-)	38. 1 (-....)

The next step is to generate .TXT files which can be uploaded to the Morse Trainer's SD Card. Toward this end, I wrote a C program (*generate.c*), included in the **QEXFiles** material for this article, which allows the user to select any combination of letters, numbers, and symbols, and generate a .TXT file containing a specified number of characters randomly chosen from this list.

To run this program:

Under Mac OS (using Terminal)

Install Xcode command line tools (includes clang):

```
xcode-select - install
```

Compile:

```
clang generate.c -o generate
```

Run:

```
./generate
```

Under Windows

Download and install MinGW (www.mingw.org)

Install gcc compiler using MinGW Installer

Add c:\MinGW\bin to your PATH environment variable

Compile:

```
gcc generate.c -o generate.exe
```

Run:

```
generate.exe
```

Under Linux (Ubuntu, Debian, Fedora, etc.)

Install gcc if not already installed:

```
sudo apt update  
sudo apt install build-essential
```

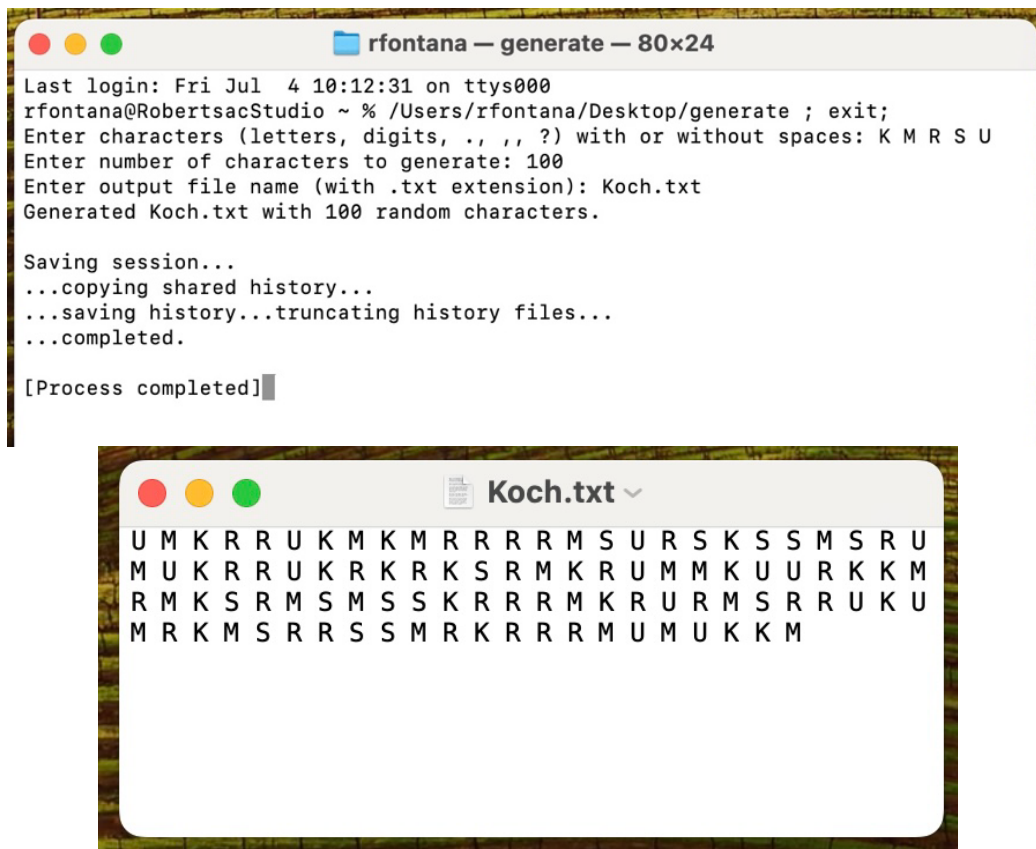
Compile:

```
gcc generate.c -o generate
```

Run:

```
./generate
```

Example Output:



The image shows two screenshots from a macOS environment. The top screenshot is a terminal window titled 'rfontana — generate — 80x24'. It displays the following text: 'Last login: Fri Jul 4 10:12:31 on ttys000', 'rfontana@RobertsacStudio ~ % /Users/rfontana/Desktop/generate ; exit;', 'Enter characters (letters, digits, ., ,, ?) with or without spaces: K M R S U', 'Enter number of characters to generate: 100', 'Enter output file name (with .txt extension): Koch.txt', 'Generated Koch.txt with 100 random characters.', 'Saving session...', '...copying shared history...', '...saving history...truncating history files...', '...completed.', and '[Process completed]'. The bottom screenshot is a text editor window titled 'Koch.txt'. It displays a 4x20 grid of random characters generated from the set 'K M R S U'. The characters are: Row 1: U M K R R U K M K M R R R R M S U R S K S S M S R U; Row 2: M U K R R U K R K R K S R M K R U M M K U U R K K M; Row 3: R M K S R M S M S S K R R R M K R U R M S R R U K U; Row 4: M R K M S R R S S M R K R R R M U M U K K M.

```
rfontana — generate — 80x24  
Last login: Fri Jul 4 10:12:31 on ttys000  
rfontana@RobertsacStudio ~ % /Users/rfontana/Desktop/generate ; exit;  
Enter characters (letters, digits, ., ,, ?) with or without spaces: K M R S U  
Enter number of characters to generate: 100  
Enter output file name (with .txt extension): Koch.txt  
Generated Koch.txt with 100 random characters.  
  
Saving session...  
...copying shared history...  
...saving history...truncating history files...  
...completed.  
  
[Process completed]
```

```
Koch.txt  
U M K R R U K M K M R R R R M S U R S K S S M S R U  
M U K R R U K R K R K S R M K R U M M K U U R K K M  
R M K S R M S M S S K R R R M K R U R M S R R U K U  
M R K M S R R S S M R K R R R M U M U K K M
```

In the above example, the letters K M R S and U were selected, and a file of 100 random exemplars from that set were generated (Koch.txt file). Koch.txt can now be uploaded to the SD Card for future play.

Other Applications

Foxhunt Transmitter

The audio output from the Morse trainer can be applied directly to the microphone input on many handheld radios. As an example, here is a Baofeng UV-5R being keyed (in VOX mode) by the Morse trainer. (The VOX menu on the Baofeng is set to a nonzero value, and the audio output on the Morse trainer is adjusted, if necessary, to prevent overmodulation.)

With its infinite looping capability, the ability to use an arbitrary text file for transmission, and the use of inter-transmission delay, the Morse trainer provides a versatile and compact foxhunt controller.



Beacon Mode

The opto-isolated output from the Morse trainer can be used to directly trigger the keying circuitry on most transceivers. (Some of the older, tube-style units may require an additional, high voltage transistor for keying; however, the PC817X optoisolator can handle voltages of up to 80 V.)

For Beacon operation, the unit can be used to repeatedly send an arbitrary text file. An external 3.5 – 12 V supply may be preferred, in this case, over battery operation.

Appendix

Generate.c Code (Koch Method)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <time.h>

#define MAX_INPUT 200
#define MAX_ALLOWED_CHARS 100
#define MAX_FILENAME 100

// Check if character is valid (letter, digit, or one of ., ,, ?)
int is_valid_char(char c) {
    return (isalpha(c) || isdigit(c) || c == '.' || c == ',' || c == '?');
}

// Clean input: extract valid characters, convert letters to uppercase
void clean_input(char *input, char *valid_chars, int *count) {
    *count = 0;
    for (int i = 0; input[i] != '\0'; i++) {
        char c = input[i];
        if (is_valid_char(c)) {
            if (isalpha(c))
                c = toupper(c);
            valid_chars[*count] = c;
            (*count)++;
        }
    }
}

int main() {
    char input[MAX_INPUT];
    char allowed_chars[MAX_ALLOWED_CHARS];
    char filename[MAX_FILENAME];
    int char_count = 0;
    int num_to_generate;

    // Seed the random number generator
    srand((unsigned int)time(NULL));

    // Get valid characters from user
    printf("Enter characters (letters, digits, ., ,, ?) with or without spaces: ");
    fgets(input, sizeof(input), stdin);
    clean_input(input, allowed_chars, &char_count);

    if (char_count == 0) {
```

```

        printf("No valid characters provided.\n");
        return 1;
    }

    // Get number of characters to generate
    printf("Enter number of characters to generate: ");
    if (scanf("%d", &num_to_generate) != 1 || num_to_generate <= 0) {
        printf("Invalid number.\n");
        return 1;
    }

    // Get output file name
    printf("Enter output file name (with .txt extension): ");
    scanf("%s", filename);

    // Open output file
    FILE *fp = fopen(filename, "w");
    if (fp == NULL) {
        perror("Error opening file");
        return 1;
    }

    // Generate and write characters
    for (int i = 0; i < num_to_generate; i++) {
        char c = allowed_chars[rand() % char_count];
        fprintf(fp, "%c", c);
        if (i < num_to_generate - 1) {
            fprintf(fp, " ");
        }
    }

    fclose(fp);
    printf("Generated %s with %d random characters.\n", filename,
num_to_generate);
    return 0;
}

```