Build yourself a Keyboard Keyer

Bruce E. Hall, W8BH

Introduction

"Life happens to us while we are busy making other plans," said Allen Saunders. In my case, I was planning an LED tester. It's a project that had been waiting for months. I had a rough design in mind when I received an email, asking me if I had seen any modern recreations of Ron Alspaugh's "Computer Keyboard CW Encoder" (QST, Dec 1997). I hadn't. Then it dawned on me: the hardware for my LED tester would be perfect for a keyboard-to-Morse interface. The LED tester would have to wait.

Are you interested in trying typed CW? If so, read on. This article describes the construction of an inexpensive PS/2 keyboard interface. As you type, the characters are displayed on a small OLED screen and converted to Morse Code. This device will key most modern amateur radio transceivers via an onboard 1/8" jack.

I will describe the hardware and software, step-by-step. When finished, you will have a useful piece of equipment and be able to customize it for you needs.

Some features:

- Small and Inexpensive
- Ham-friendly, through-hole PCB construction
- Large CW message memories.
- Variable speed (5-50 WPM) and side-tone audio
- Optically-coupled keyer output, reducing RF and ground-loop issues.

I will describe the project the way I built it, starting with the microcontroller. In this tutorial I assume that you are familiar with the Arduino IDE and know your way around a soldering iron.

Project Files

Part 1 (this document)
Part 2: Builder's Guide
Part 3: User's Guide
Enclosure STL files
Schematic
Source Code
PCB Gerbers
YouTube video

Step 1. The Microcontroller

I have been using Arduino hardware for a long time. The original ATmega328 chip has been slowly supplanted by newer and faster devices. For example, I've used Blue Pill and ESP32 microcontrollers in recent projects. The Blue Pill is hard to beat for the money, but works best with a dedicated programmer. The ESP32 is another great choice, but comes in so many flavors that picking the right one can be a chore. For this project I chose a completely different controller: the Seeeduino (that's not a typo) XIAO. It's a chiclet-shaped board, measuring about 18 x 20 mm, that is quite easy to program.

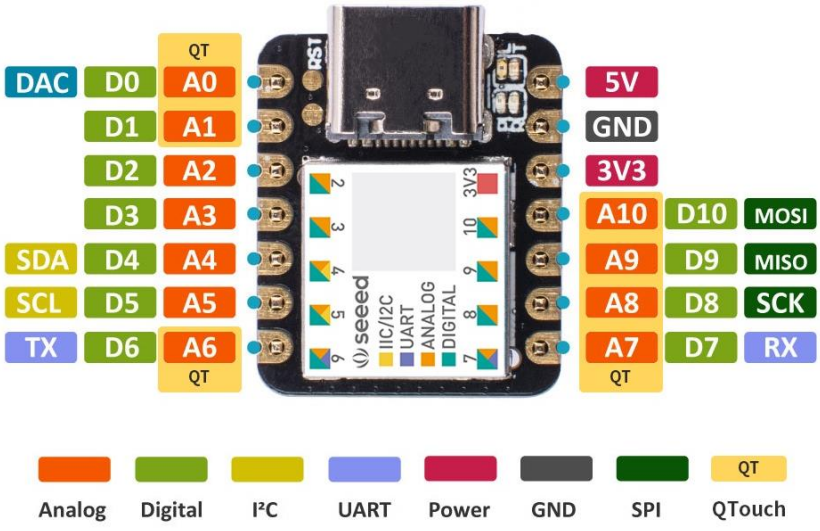
This board is widely available at a cost of $6 each. You can order it on Amazon or directly from Seeed. I ordered mine through Digikey. It is programmed and gets its power through a USB-C cable. So, order one of those, too, if you don't have one already.

The official setup guide is at https://wiki.seeedstudio.com/Seeeduino-XIAO/#hardware. Briefly,

1. Connect the Seeeduino to your computer via a USB-C cable. (My computer does not have a USB-C port, so I use a USB-A to USB-C cable). The green LED on your Seeeduino should light up, indicating that it has power.
2. Start your Aduino IDE application. Go to **File > Preferences** and fill the "Additional Boards Manager URL" with the URL for this board, which is: https://files.seeedstudio.com/arduino/package_seeeduino_boards_index.json
3. Go to **Tools > Board Manager > Boards Manager** and put the words "Seeeduino XIAO" in the search box. You should see an entry for "Seeed SAMD Boards". Click Install.
4. Go to **Tools > Board** and hover over the choice "Seeed SAMD". Select "Seeeduino XIAO".
5. Go to **Tools > Board > Port** and select the com port associated with your device.

Try uploading the blink sketch. The orange LED on the XIAO should flicker as the code is uploaded, then the same LED should blink. During upload the computer will register a USB disconnect followed by a USB reconnect, which is slightly annoying but normal.

For reference, here is the XIAO board pinout:



Step 2. CQ

Time for Morse Code. Let's start by flashing CQ using the onboard LED. We need a short-flash for dit and a long-flash for dah. Do the dits first to send the letter "S":

```
void dit() {
  digitalWrite(LED_BUILTIN,0);      // turn on LED
  delay(120);                       // wait 120 milliseconds
  digitalWrite(LED_BUILTIN,1);      // turn off LED
  delay(120);                       // space between code elements
}

void setup() {
  pinMode(LED_BUILTIN,OUTPUT);
}

void loop() {
  dit(); dit(); dit();              // Send "S"
  delay(2000);                      // repeat every 2 seconds
}
```

This code is downloadable from Github as Step 2a. It looks similar to the blink sketch, doesn't it? Now add a dah() routine. Notice that it differs from dit only in the length of time that the LED is 'on'. Sending CQ now is just a matter is stringing together the correct dits and dahs;

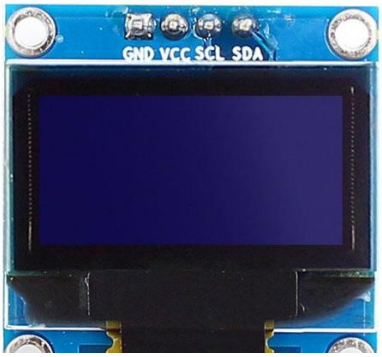
```
void dah() {
  digitalWrite(LED_BUILTIN,0);      // turn on LED
  delay(360);                       // wait 360 milliseconds
  digitalWrite(LED_BUILTIN,1);      // turn off LED
  delay(120);                       // space between code elements
}:

void loop() {
  dah(); dit(); dah(); dit();       // send "C"
  delay(360);                       // space between characters
  dah(); dah(); dit(); dah();       // send "Q"
  delay(2000);                      // repeat every 2 seconds
}
```

Before finishing with CQ, let's add sound. Arduino has a built-in function called tone() that generates a square wave of a specified frequency: tone(pin, frequency). noTone(pin) turns the tone off. Add them to the dit() routine, so that a tone is generated whenever the LED is on:

```
void dit() {
  digitalWrite(LED_BUILTIN,0);      // turn on LED
  tone(9,1200);                     // 1200 Hz tone on pin 9
  delay(120);                       // wait 120 milliseconds
  digitalWrite(LED_BUILTIN,1);      // turn off LED
  noTone(9);                        // turn off tone
  delay(120);                       // space between code elements
}
```

Make the same modifications to the dah() routine. Now add a piezo element between pin 9 of the Seeeduino and ground, and enjoy the sounds of some real Morse Code at 10 words per minute. The full step 2b sketch is downloadable from GitHub.

Step 3. Adding a display



0.96" I2C OLED Display module

You could add a large, color display to this project, and it would look great. I used 2.2", 2.8", and 3.2" ILI9341 displays for my Morse Tutor project to good effect. But I discovered that this keyer does not need a display at all. Just type, and Morse code comes out.

So, I chose a small, monochrome OLED display. It does not take up much space. The 128x64 pixel display is less than an inch diagonal. OLED technology gives it a crisp-looking output that makes smaller fonts readable. And it is cheap: The HiLetGo version on Amazon is currently $6.40. I got mine on Amazon from "Frienda" at 5 for $16. Nice.

Compared to power-hungry ILI9341 boards, an OLED display with SSD1306 controller uses very little power. It has an I2C interface, so only 2 interface pins are required. The downside is that your microcontroller must buffer the entire display in its memory. It is not compatible with older microcontrollers that have limited memory. Fortunately, the XIAO has no problems driving this display.

Here are the connections between the XIAO and the OLED display:

For those of us of a certain age, the pin labels on the XAIO sticker are nearly impossible to read. So, refer to the pinout diagram above. Notice that, with the USB jack oriented away from you, there are 7 pins on the left and 7 on the right. The top two pins on either side are unlabeled. Pin D4 is the fifth pin on the left.

Seeeduino XIAO	OLED Display
3v3	VCC
GND	GND
"5" (D5)	SCL
"4" (D4)	SDA

Connect the hardware according to the table above. We will need to write another sketch to test it out. Adafruit has done the hard work and created a display library for our use. Consider returning the favor and buying from them.

Here is a sketch to test your display, available on Github as Step 3a:

```
#include <Wire.h>                       // built-in I2C library
#include <Adafruit_GFX.h>               // Adafruit graphics library
#include <Adafruit_SSD1306.h>           // Adafruit OLED library

#define SCREEN_WIDTH     128            // OLED display width, in pixels
#define SCREEN_HEIGHT    64             // OLED display height, in pixels
#define SCREEN_ADDRESS   0x3C           // OLED display I2C address, in hex

Adafruit_SSD1306 led(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire);

void setup()   {
  led.begin(SSD1306_SWITCHCAPVCC, SCREEN_ADDRESS);
```

```
  led.display();
}

void loop() {
}
```

Several libraries are required. First, since this display uses I2C for communication, we need to add the Arduino I2C library, called <Wire.h>. Next, we have to add a library for drawing basic graphics and text objects, called <Adafruit_GFX.h>. Finally, we need a library that allows us to communicate with the SSD1306 controller on the display board, <Adafruit_SSD1306.h>.

The Wire library is pre-installed, but you might need to add the Adafruit libraries if you haven't already. In the IDE, go to **Sketch > Include Library > Manage Libraries**. In the search box type "Adafruit SSD1306". If the corresponding entry does not say "INSTALLED", install it now. Do the same for "Adafruit GFX Library".

After the libraries are installed, take note of the screen address define "0x3C". You must enter the I2C address of your display module. Many of these displays use the address 0x3C. Interestingly, the address printed on my display is wrong, and caused me many hours of frustration. The correct address turned out to be 0x3C. If, after looking at the documentation for your screen, you cannot determine its address, download the I2C scanner sketch from GitHub. It will find all of the I2C devices connected to your microcontroller and send their addresses to the serial monitor. Very handy!

After confirming and entering the address of your I2C module, run the sketch. Do you see anything on screen? If so, pat yourself on the back and continue on. If not, check your hardware connections again, try the I2C scanner, and confirm the address.

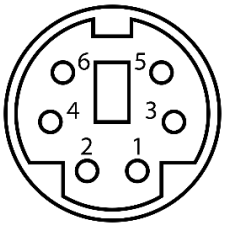
Now that we have a working display, lets add it to our CQ sender from Step 2. Try it! The entire sketch is shown here, and is available on Github as Step 3b:

```
#include <Wire.h>                         // built-in I2C library
#include <Adafruit_GFX.h>                 // Adafruit graphics library
#include <Adafruit_SSD1306.h>             // Adafruit OLED library

#define SCREEN_WIDTH     128              // OLED display width, in pixels
#define SCREEN_HEIGHT    64               // OLED display height, in pixels
#define SCREEN_ADDRESS   0x3C             // OLED display I2C address, in hex

Adafruit_SSD1306 led(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire);

void dit() {
  digitalWrite(LED_BUILTIN,0);            // turn on LED
  tone(9,1200);                           // 1200 Hz tone on pin 9
  delay(120);                             // wait 120 milliseconds
  digitalWrite(LED_BUILTIN,1);            // turn off LED
  noTone(9);                              // turn off tone
  delay(120);                             // space between code elements
}

void dah() {
  digitalWrite(LED_BUILTIN,0);            // turn on LED
  tone(9,1200);                           // 1200 Hz tone on pin 9
```

```
  delay(360);                                  // wait 360 milliseconds
  digitalWrite(LED_BUILTIN,1);                 // turn off LED
  noTone(9);                                   // turn off tone
  delay(120);                                  // space between code elements
}

void setup() {
  pinMode(LED_BUILTIN,OUTPUT);
  led.begin(SSD1306_SWITCHCAPVCC, SCREEN_ADDRESS);
  led.setTextColor(WHITE,BLACK);               // use black on white text
  led.setTextSize(2);                          // medium-sized font
  led.clearDisplay();
}

void loop() {
  led.print('C'); led.display();               // show "C" on display
  dah(); dit(); dah(); dit();                  // and send it in Morse
  delay(360);                                  // space between characters

  led.print('Q'); led.display();               // show "Q" on display
  dah(); dah(); dit(); dah();                  // and send it in Morse

  led.print(' '); led.display();               // space between words
  delay(3000);                                 // repeat every 3 seconds
}
```

Try it! With only 6 additional lines of code, highlighted above, the sketch is becoming useful. The three additional lines in setup() set the font size and color. The three additional lines in loop() print the letters C and Q, synchronized with the Morse Output. Unlike other libraries, a call to the print() routine does not change the appearance of the screen. It only writes the character to a memory buffer. You must call led.display() to make those changes visible.

Step 4. The PS/2 Keyboard

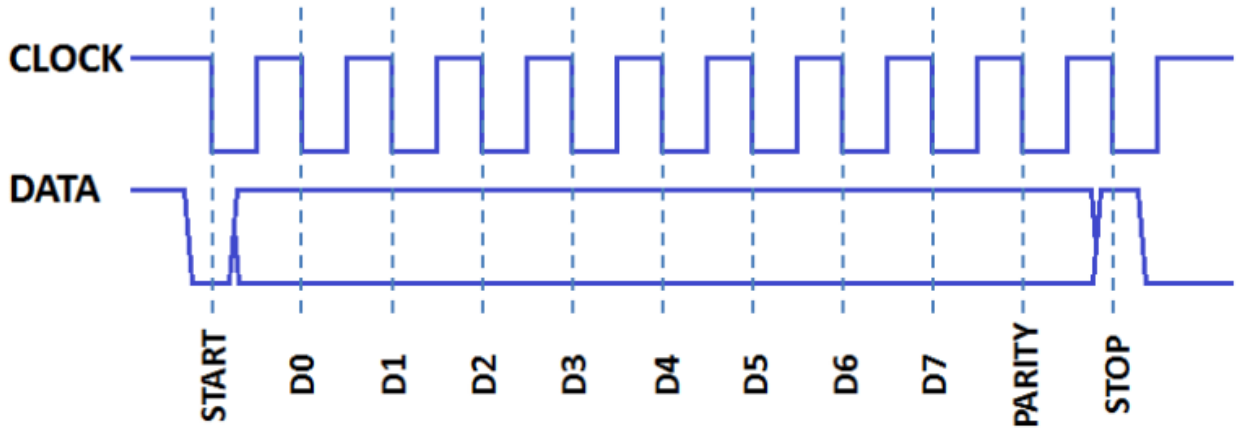


Female Mini-DIN-6

Pin 6	N/C
Pin 5	Clock
Pin 4	+5V
Pin 3	GND
Pin 2	N/C
Pin 1	Data

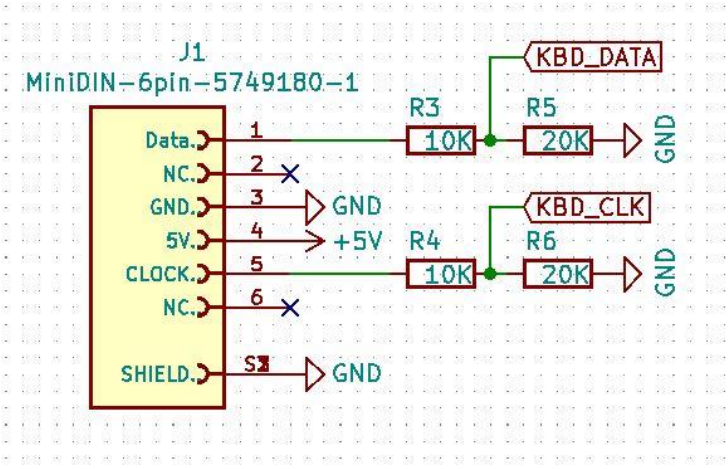
The keyboard is the final and most important hardware to add. In this project we are using the PS/2 keyboard. This style of computer keyboard was introduced with the IBM Personal System/2 series of computers in 1987, hence the name PS/2. The connector for these keyboards is a 6-pin mini-DIN. The PS/2 keyboards were gradually phased out, starting in 2000 with the creation of the USB 2.0 standard, and are now considered legacy devices. So why use these older keyboards? Why not use the more ubiquitous USB variety?

Answer: because it's easier. Modern keyboards require a "USB host controller". The USB specification is complex and requires a significant amount of overhead. While this is not an impossible task for our lowly micro, it is not easy, either. The PS/2 specification, however, is straightforward: the keyboard sends its data using two wires: a clock line and a data line. Data is sent from the keyboard to the micro one byte at a time. It takes 11 clock cycles to send a byte: a start bit, 8 bits of data, a parity bit, and a stop bit:



The data is valid and sampled by the micro on each high-to-low clock transition, as indicated by the vertical dashed lines above.

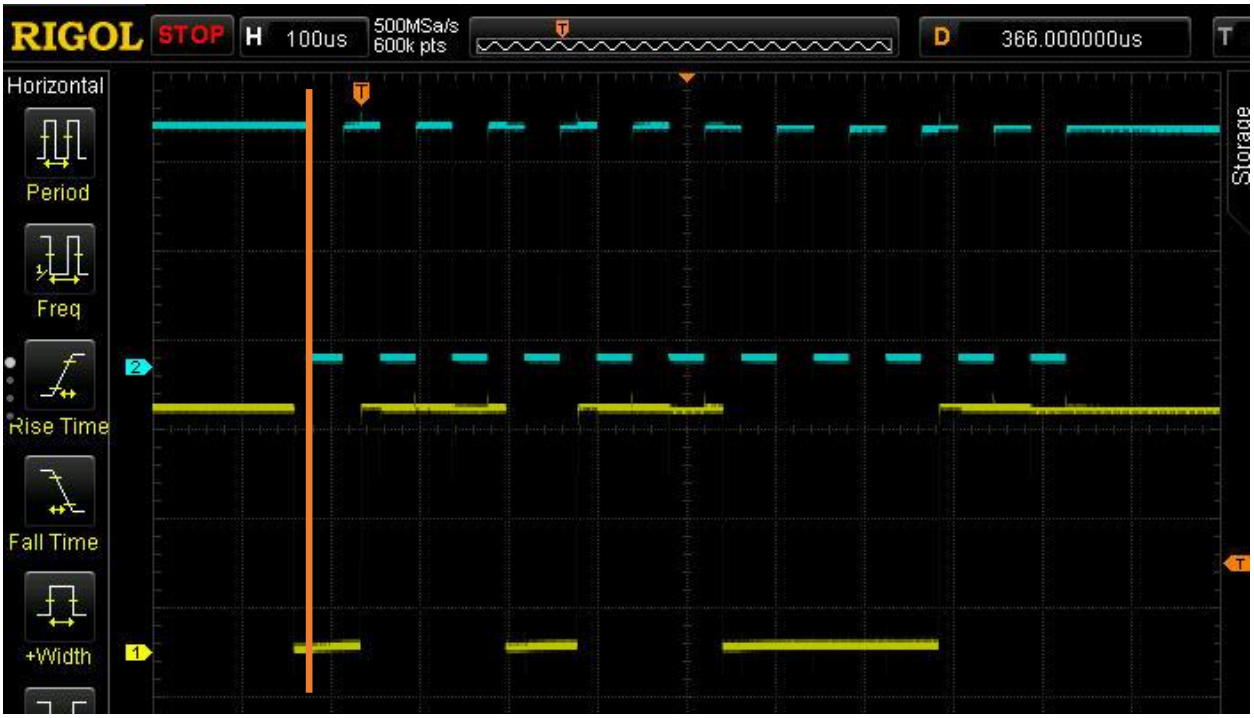
To connect our keyboard, supply +5V to pin4 and ground to pin3. The data and clock lines (pins 1 and 5) are then routed to the micro. Unfortunately, these +5V lines will ruin our microcontroller if connected directly. We will need to convert the 5V to 3.3V logic levels. This can be easily done with dedicated logic level converter chips, or with discrete components like mosfets. I chose a junk-box-friendly approach: two resistors in a voltage-divider circuit. By using common 10K and 20K resistors, we reduce the 5-volt logic signals to 5V * 20K/(20K+10K) = 3.3 volts. Here is the schematic for Bruce's super-simple keyboard interface.



Hook up your keyboard, according to the schematic above, connecting the KBD_DATA line to micro pin 1 and the KBD_CLOCK line to micro pin 0.

If you need a breakout board, here is a link to the one I use: http://w8bh.net/ps2_breakout.pdf

How do you know if your keyboard is working? This is an easy job if you have an oscilloscope and want a reason to use it. Connect the +5V and GND pins to a 5V supply. Put clock on Channel A and Data on Channel 2. The data rate is approximately 10 kHz, so set your horizontal axis for 100uS/div. We are measuring 5V signals so set your vertical axis for 2V/div. Set the mode to single shot. Press the letter 'S' on your keyboard and you should see something like this:



As expected, there are 11 downward-going clock pulses (blue) in the data stream. At the first hi-to-low transition of the clock (orange vertical line), the data line in yellow is low '0'. This is the start bit. At the next falling edge of the clock, the data line is high '1'. Reading left-to-right, the data bits are: 01101100011. Referring to the diagram above:

Start	D0	D1	D2	D3	D4	D5	D6	D7	Parity	Stop
0	**1**	**1**	**0**	**1**	**1**	**0**	**0**	**0**	1	1

Considering only the data bits, D0-D7, and reversing their order, the byte sent was 0001.1011 or 0x1B. The value 0x1B is indeed the scancode for the letter 'S'.

No scope? No problem. Use your microcontroller to probe the data signals. Download Step 4a from GitHub and run it. This simple sketch counts the hi-to-low pulse transitions on pin 0, using an interrupt routine, and shows the count on your display. Connect pin0 to the keyboard clock line (via the resistor divider, of course) and press a key. You should see the count increment in multiples of 11. In fact, each keypress generates 33 pulses: 11 when the key is pressed down, and 22 when the key is released. Before continuing, verify that your keyboard is generating signals on both the clock and data lines.

There are several good libraries available that decode keyboard signals. But many are quite old, and were written long before the XIAO. So, I decided to rewrite one specifically for this project. It is called "EasyKey" and is available on GitHub.

Download and install the EasyKey library. Then, try the following sketch, available on GitHub as Step 4b:

```
#include <EasyKey.h>

#define CLOCK_PIN            0        // pin for keyboard clock line
#define DATA_PIN             1        // pin for keyboard data line

EasyKey kbd;

void setup()   {
  kbd.begin(DATA_PIN, CLOCK_PIN);     // initialize keyboard library
  Serial.begin(115200);               // initialize serial monitor
}

void loop() {
  if (kbd.available()) {              // When a key is pressed,
    char c = kbd.read();              // get the character and
    Serial.print(c);                  // echo it to serial monitor
  }
}
```

The five important lines are highlighted. First, include the EasyKey library and create a keyboard object 'kbd'. Next, in setup(), tell it which pins are connected to the keyboard data and clock lines. In loop(), use available() to see when a keypress has occurred. Finally, read() returns the character pressed.

Run this sketch, open your serial monitor, set its baud rate to 115200, and type a few alpha characters on your PS/2 keyboard. If all goes well, you will see the characters you typed. If not, check your hardware connections as above.

As an aside, check out the step4C sketch to see the raw scancode data generated by each keypress. In addition to the usual alphanumeric characters, the EasyKey library allows you to detect many <shift>key, <ctrl>key, and <alt>key combinations, as defined in EasyKey.h.

Step 5. Keyboard Morse

In Step 2 we created a CQ message. In Step 3 we added the display. And in the last step we added keyboard input. It's time to put all of those together to send CQ from our keyboard. Grab the Step 3b code and add in the following lines for keyboard support:

```
#include <EasyKey.h>

#define CLOCK_PIN            0        // pin for keyboard clock line
#define DATA_PIN             1        // pin for keyboard data line

EasyKey kbd;
```

And add a line to setup() for keyboard initialization:

```
kbd.begin(DATA_PIN, CLOCK_PIN);     // initialize keyboard library
```

Finally, modify the loop() routine, so that it checks the keyboard and displays the characters:

```
void loop() {
  if (kbd.available()) {                  // wait for keypress
    char c = toupper(kbd.read());         // get the character, as uppercase
    led.print(c); led.display();          // show character on display
    if (c=='C') {                         // if it's a "C":
       dah(); dit(); dah(); dit();        // send C in Morse
    } else if (c=='Q') {                  // if it's a "Q":
       dah(); dah(); dit(); dah();        // send Q in Morse
    }
  }
}
```

The keyboard is checked and read just as in Step 4. The only modification is use of the toupper() function, which converts all incoming characters to upper case. Each character is then shown on the display via led.print(c). Finally, if the character is a C or a Q it is sounded out in Morse Code.

The complete sketch is available on GitHub as step 5a. Try it, typing C and Q on your keyboard. None of the other characters will sound out, because we haven't programmed them. Try adding a few more, like this:

```
    if (c=='C') {                         // if it's a "C":
       dah(); dit(); dah(); dit();        // send C in Morse
    } else if (c=='Q') {                  // if it's a "Q":
       dah(); dah(); dit(); dah();        // send Q in Morse
    } else if (c=='A') {                  // if it's a "A":
       dit(); dah();                      // send A in Morse
    } else if (c=='B') {                  // if it's a "A":
       dah(); dit(); dit(); dit();        // send B in Morse
    }
```

You could add the whole alphabet by extending the if..else statement for each letter. It wouldn't be pretty, but it would work. A slightly better arrangement would be to arrange them in a switch..case statement like this:

```
void sendMorse(char c) {
  switch(c) {
    case 'A': dit(); dah();               break;
    case 'B': dah(); dit(); dit(); dit(); break;
    case 'C': dah(); dit(); dah(); dit(); break;
    case 'D': dah(); dit(); dit();        break;
    case 'E': dit();                      break;
    // add a line for each alphanumeric character
  }
}
```

We now have a working keyboard keyer. The full sketch, using switch..case, is available on Github as step 5b. Modify it as you like, adding characters, changing the speed, etc.

Step 6. Surviving without EEPROM

The rest of the keyboard-keyer sketch is window dressing. But there are a few details that merit mentioning. First, for Morse Code conversion, I decided to use the same algorithm – with minor modifications -- that I wrote for my Morse Tutor project. See Part 2 for a full description. Second, it is useful to save user settings, such as speed, pitch, and keyer memories, so that they are not lost when the unit is turned off. These settings are typically stored in a type of non-volatile computer memory called EEPROM. The bad news is that the XIAO and other SAMD21-based devices do not have EEPROM. The good news is that there is a workaround: we can reserve a portion of its non-volatile FLASH memory instead, where the sketch normally resides. There are a few disadvantages of using Flash instead of EEPROM:

1. It requires special programming techniques.
2. The contents are lost whenever a new sketch is uploaded.
3. FLASH memory has much shorter cycle-life, in the 10,000s range.

Since we have no other option for saving data, it will have to do. We minimize programming by using a library, called FlashStorage, which was created to do the heavy lifting. And we minimize the write-cycles by saving data only when necessary.

The FlashStorage library is installed with the XIAO core. The following lines will create a data object that we can store in Flash:

```
#include <FlashStorage.h>                   // loaded with XIAO core
FlashStorage(flash,FlashObject);            // dedicate portion of Flash

typedef struct {
  bool valid;                               // flash contains valid data?
  int  pitch;                               // frequency of audio output
  int  codeSpeed;                           // current morse speed in WPM
} FlashObject;

FlashObject f;                              // instantiate a flash object
```

The nomenclature is a little confusing. 'typedef struct' means that we are creating a new type of data. A struct a collection of data, and we are going to call this new datatype FlashObject. Finally, we create a variable of this type and call it 'f'. f contains 3 pieces of data. We reference them as f.valid, f.pitch, and f.codeSpeed.

We can add as many variables to this data structure as we need. Saving and retrieving the data becomes quite easy. The following code shows how straightforward it is:

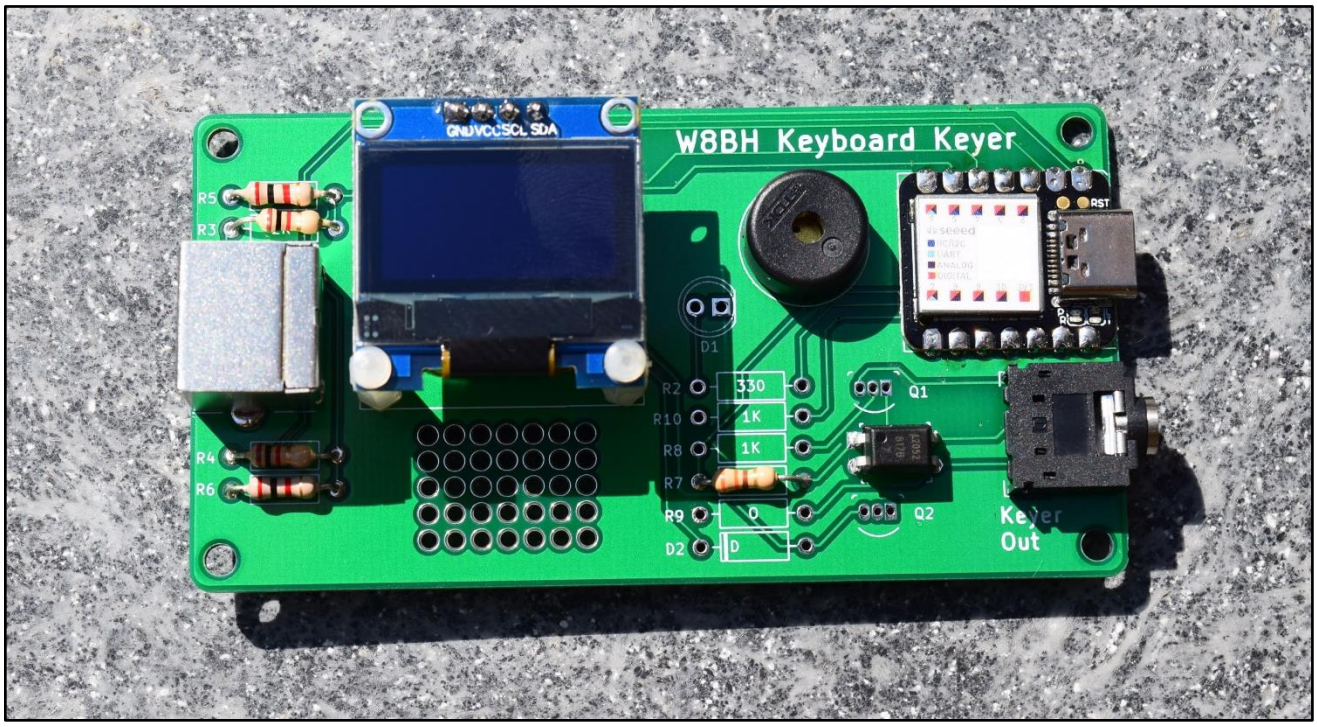
```
void loadParams() {
  f = flash.read();                         // get all data from flash
  if (!f.valid) setDefaults();              // invalid, use defaults instead
}

void saveParams() {
  f.valid = true;                           // flag contents as valid data
  flash.write(f);                           // and save the settings.
}
```

Finishing Touches.

The rest, as they say, is a simple matter of programming. I had fun adding the features I wanted. With the current software, you can:

1. Set up function key macros on your keyboard, F1 through F12, with each one containing up to 99 characters. The macros can be changed at any time without programming.
2. Stop a macro in progress.
3. Add a 'pause' in your macro, allowing you to insert a word or phrase (like an RST report or name) at the appropriate time.
4. Set a default code speed, and then change that speed while in use.
5. Configure the display to use larger or smaller fonts.
6. Set the pitch of the side-tone audio and turn it on or off during operation.

What will *you* add? 73, Bruce.



Reference Links.

This tutorial on w8bh.net
Schematic
Source code and Tutorial Examples on Github
PCB Gerbers
Builder's guide
User's guide
YouTube video
Enclosures

Last updated 5/2/2022