SAOGNR - A Ham Radio SAO (Supercon Add On)

By Al Williams WD5GNR

Rev 0.1 1 October 2024

Introduction

For Supercon 2024 we threw down a challenge: bring your best SAO designs and – for a change – try to use I2C. While I'm not very artistic when it comes to board design, I decided I better set an example, and so SAOGNR was born.

What does it do? Well, it sends little messages via Morse code on a built-in speaker and on some intelligent LEDs. It draws power from the SAO connector on the host badge, or you can power it via USB C. In addition to sending little Morse code snippets, it can also act as an I2C peripheral so that a computer on the main badge can command it to send Morse code.

There are several ways you might use SAOGNR. If you just want it to beep and blink, you'll want to read the **User's Guide**, below. Fair warning: if you want to change the messages, you will have to edit files on the built-in RP2040 CPU. However, they are just text files and however you send them to the RP2040 is fair game (e.g., mpremote, for example).

If you just want to customize certain things, you can easily change a single file using Micropython (config.py) and change many things about the code without having to know anything about Python. You can find out about these in the section **Simple Configuration**.

If you want to write code on the badge CPU and talk to SAOGNR via I2C, that's another section (**Using I2C**). In addition to sending normal characters, you can operate the same menus available to the user and commands that may not be available to the user.

Finally, if you really want to hack on the code, there's the **Hacking** section. Want to make your own board, check out the **Resources and Schematic** section, below.

FAQ

Q: Why is there so much wasted space on the board?
A: I wanted room for silkscreening the logos and I wanted the biggest badge allowed by the SAO "spec."

Q: How did you shrink the Pi Pico?
A: It is actually a product called the RP2040-Zero. Unlike a real Pico, it has an addressable LED onboard, although – at least on the ones I have – they use BGR instead of RGB.

Q: Can I plug the USB cable in while the board is powered through the SAO connector?
A: In theory, yes. There is a FET that switches the SAO power off when it detects 5V from the USB connector.

Q: What is P2 for? Why isn't it populated?
A: P2 is just an area with some signals where you can prototype or put an SMD 2x6 header for connecting things.

Q: Can it run Doom?
A: No

User's Guide

When you plug the board into a powered SAO or USB connector, it should start going. By default, it will play message 0 at 13 WPM and 800 Hz. It will delay 60 seconds between sendings. However, you can change any of this and save it so that your defaults will be used.

Everything you do (other than watch and listen) revolves around the BOOTSEL button on the RP2040-Zero board in the SAO. If you press it quickly, it will either start the message playing or stop it, depending on the current state of the SAO.

However, you can also long press the button for more than a second. When you release the button, you will enter menu mode and any playback will abort. The top left LED will turn purple to indicate it is waiting for a command. The top right LED will turn dull purple. This indicates a number (which is also sent in Morse code) that is almost the resistor color code. Each time you short press the BOOTSEL button, the number will advance. So:

Color	Digit	Command
Dull Purple	0	Exit menu
Brown (sort of)	1	Select message (0-9)
Red	2	Select delay (0-9)
Orange (sort of)	3	Select WPM (0-4)
Yellow	4	Select tone (0-3)
Green	5	Save/delete configuration (0=exit, 1=save,

		2=delete file and restore default)
Blue	6	Temporary restore default (0=no, 1=yes)
Violet	7	Reset message number counter

When you long press to accept a command, the top left LED turns blue to indicate it is now waiting for you to enter the second part of the command. By default, the menu will start with the current value, so if you don't want to change anything, just long press again.

Commands, 2, 3, and 4 select among a few options:

Number	Command 2	Command 3	Command 4
0 (dark purple)	30 seconds	13 WPM	800 Hz
1 (brown)	60 seconds	5 WPM	440 Hz
2 (red)	90 seconds	20 WPM	1000 Hz
3 (orange)	300 seconds	25 WPM	1200 Hz
4 (yellow)	600 seconds	50 WPM	
5 (green)	900 seconds		
6 (blue)	1800 seconds		
7 (violet)	3600 seconds		
8 (gray/dim white)	7200 seconds		
9 (white)	0 (don't repeat automatically)		

Once you select a second value, the board returns to normal operations.

Simple Configuration

If you want to change messages or do other simple configuration, you'll need to change files on the RP2040-Zero via a PC with a USB C connector.

Each message is stored in a file named MessageN.txt where N is a number from 0 to 9. In general, any upper or lower case letters, numbers, spaces, and some punctuation will be sent. Illegal characters are ignored. If you want to see what characters are available, see the table in

Morse.py. Note that + is AR and = is BT. Spaces cause a word space and a new line is treated as a space.

You can add comments using the # character and they go through the end of the line.

If you want a longer delay, you can use a ~ character in any position but the first character. Each ~ provides a 1 second delay. If the very first character is a ~ you will get a 1 second delay, but you'll also get a constantly repeating message with no extra delay between sending. You can, of course, program in whatever delay you like. So if Message5.txt reads:

```
~73~~~
```

The board will pause a second, send 73, pause three more seconds and then repeat (including the initial pause). If you really just want a pause at the start (why?) you can use a comment:

```
# don't use ~ at the first character
~I play after a 1 second pause without autorepeat
```

If you put an underscore (_) in a message, the message number will replace it. The number rolls over at 9999 and you can reset it to 1 using command 7 from the menu.

More advanced configuration requires editing config.py. This file contains many options such as DEFAULT_WPM, the tables for the menus, and some hardware things you would only change if you were adapting the software to a different board.

The comments in the file will tell you what each item does. Note that indentation is important. So to change the LED breathing color, you can't write:

```
     CPU_LED_PIN = 16  # Pin controlling onboard NeoPixel
CPU_LED_BASE_COLOR=(255,0,0)  # Base color for the NeoPixel (R, G, B)
     CPU_LED_BREATHE_TIME = 60 # Rate at which the onboard Neopixel "breathes" in
```

Instead, you should write:

```
     CPU_LED_PIN = 16  # Pin controlling onboard NeoPixel
     CPU_LED_BASE_COLOR=(255,0,0)  # Base color for the NeoPixel (R, G, B)
     CPU_LED_BREATHE_TIME = 60 # Rate at which the onboard Neopixel "breathes" in
```

Using I2C

The SAOGNR uses an I2C address of 0x73, of course. You send bytes one at a time to this address to control it. Any byte that is less than 0x7F will be treated as a letter. Note that the underscore and autorepeat characters don't work from I2C. Any character that is unknown just does nothing.

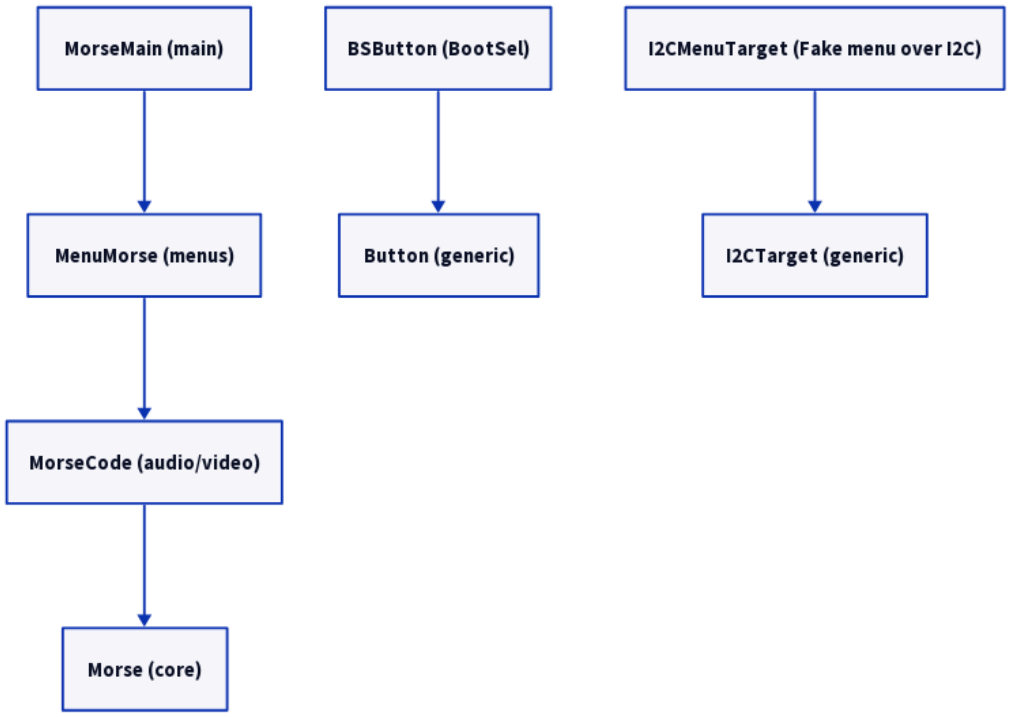
To operate a menu item you will send one or two bytes. The first byte will be 0x80 + the command number. So to send command 3, you'd send 0x83. Then to set the next menu entry

you will send 0xC0 + the number. So, for example, to set the message (command 1) to message 8, you'd send 0x81 0xC8.

There is one special command that is only available from I2C and only requires one byte. Command 0x8E will stop any output currently being sent. You should not send a second byte for this command.

If you send text while the SAO is doing something else, you might not get the results you expect. One way to work around this is to set the delay to 0 so that the SAO doesn't automatically send anything. You can also set the message to a message file that is blank so that there is never anything sent automatically. Note that if you set the delay to 0, the user can still trigger a message with the button.

Hacking



The class hierarchy appears above. The main code is in MorseMain.py and the core code-generation is in Morse.py.

The MenuMorse object also uses MenuSystem as a helper and several classes use BSButton.

A few notes:

- Morse doesn't care how you send dots and dashes. That's up to the subclass.
- The corner LEDs are RGB but the RP2040-Zero one is BRG. The code hides this from you
- The I2C receiver implementation is not very complete. Oddly, Micropython does not provide support for I2C targets.

Resources and Schematic

- Hackaday post: https://wp.me/paBn4l-32z2
- Hackaday.io: https://hackaday.io/project/198144-saognr-an-sao-for-morse-code
- GitHub: https://github.com/wd5gnr/saognr
- YouTube: https://www.youtube.com/watch?v=DIxf_U3MRIw&feature=youtu.be