Morsewurst Keyer Build Guide

Kasperi Koski

2026-05-26

Contents

1 Morsewurst Keyer

Design: Kasperi Koski

Contents: Morsewurst Python program, Arduino code, and 3D-printable enclosure

1.1 Short description

Morsewurst Keyer is an ESP32-S3-based Morse practice device that reads a straight key or an iambic paddle, produces a sidetone in headphones, shows basic information on an OLED display, and sends raw timing telemetry to Morsewurst running on a computer.

The device can also work as a USB keyboard. In that mode, it types decoded characters directly into the computer. If USB keyboard mode is enabled, it is recommended to use the Finnish keyboard layout on the computer, because some special characters are sent using Finnish keyboard key combinations.

The program code is uploaded with the Arduino IDE from the file:

`Morsewurst_keyer.ino`

The 3D-printable enclosure file is:

`Morsewurst_keyer.stl`

A suitable microcontroller for this version is:

`Adafruit ESP32-S3 Feather with STEMMA QT 8MB`
`https://partco.fi/tuote/adafruit-esp32-s3-feather-with-stemma-qt-8mb-329`

1.2 Main features

- ESP32-S3-based Morse practice device
- Straight key support
- Iambic paddle support
- Headphone sidetone
- OLED display for device settings and status information
- Rotary encoder with push button for adjusting settings
- USB CDC Serial telemetry for Morsewurst
- USB HID Keyboard mode for typing characters into the computer
- Timing data is sent in JSON format
- Measurement uses microsecond resolution, but the realistic practical accuracy is about 50 to 200 microseconds in good conditions

1.3 Required parts

1.3.1 Electronics

- Adafruit ESP32-S3 Feather with STEMMA QT 8MB or a similar ESP32-S3 board with native USB
- 128x64 I2C OLED display, for example SSD1306 or SH1106 compatible
- KY-040 or similar rotary encoder with push button
- 3.5 mm stereo jack for the iambic paddle
- 3.5 mm stereo jack for the straight key
- 3.5 mm stereo headphone jack, preferably a switched model
- 10 kΩ logarithmic mono potentiometer for volume
- 2 pcs 1 kΩ metal film resistors
- 1 pc 2.2 kΩ metal film resistor
- Wire, preferably in several colours
- If needed, Wago connectors, a ground rail, or another neat way to join GND wires

1.3.2 Tools and mechanical parts

- Soldering iron
- Solder
- Wire strippers
- Side cutters
- Small drill or hand drill for cleaning up holes
- 3D printer
- PETG filament
- Screws or other fasteners suitable for the enclosure

1.4 Before final assembly

The circuit should first be tested on a breadboard. Once the keys, display, rotary encoder, sidetone, and USB connection work reliably, the final enclosure version should be wired by soldering directly to the ESP32 board and the connectors.

In a tight enclosure, JST connectors and loose intermediate connections can cause contact problems. For that reason, direct soldering is recommended for the final version.

1.5 Arduino IDE settings

This firmware is intended for an ESP32-S3 board with native USB. The essential Arduino IDE settings are:

```
USB Mode:        USB-OTG or TinyUSB
USB CDC On Boot: Enabled
Upload Mode:     USB-OTG CDC or TinyUSB
```

If USB CDC On Boot is not enabled, Morsewurst may not find the telemetry serial port correctly. If USB-OTG or TinyUSB mode is not enabled, USB HID Keyboard mode may not work.

Use a USB data cable, not a charge-only cable.

If the computer does not recognise the ESP32-S3 board for programming, try bootloader mode:

1. Hold the BOOT button down
2. Press the RESET button
3. Release RESET
4. Release BOOT

After this, the computer should see a drive or serial port for programming.

1.6 Display

This project used the 0.96-inch I2C OLED display that came with an Elcrown plant watering kit. In practice, almost any 128x64-resolution I2C OLED display can work as long as the correct U8g2 driver is selected for the display controller chip.

In this setup, the following configuration worked well:

```
U8G2_SH1106_128X64_NONAME_F_HW_I2C u8g2(
  U8G2_R2,
  U8X8_PIN_NONE
);
```

Although the display may be sold as an SSD1306 display, in this project the `SH1106` driver worked better in practice and prevented glitches and artefacts from appearing on the right edge of the display.

If the image appears wrong, is partly shifted, or the display behaves strangely, it is worth trying different U8g2 drivers. The controller chip used in OLED displays does not always fully match the name under which the display is marketed.

`U8G2_R2` rotates the image 180 degrees. This is needed because the display is physically installed upside down in the enclosure to save space.

The display uses 3.3 volt logic and is connected to the ESP32 I2C bus.

Typical I2C wiring:

Display	ESP32-S3 Feather
GND	GND
VCC	3.3V
SCL	SCL
SDA	SDA

If you use the STEMMA QT connector, the wiring is effectively the same, but the connector handles the wire order automatically.

It is worth noting that even a small difference in the physical dimensions of the display can prevent it from fitting properly into the `Morsewurst_keyer.stl` enclosure. If you use a different display model, the enclosure may need to be modified.

The enclosure prints relatively quickly. A typical PETG print usually finishes within a few hours, so testing different display versions is fairly easy. Many libraries and makerspaces also offer access to 3D printing.

1.7 Pinout

Function	ESP32 pin
Sidetone audio out	GPIO11
Straight key	GPIO12
Iambic DIT	GPIO9
Iambic DAH	GPIO10
Rotary encoder CLK or A	GPIO5
Rotary encoder DT or B	GPIO6
Rotary encoder pushbutton	GPIO13
Headphone detect	A3

All key and encoder inputs use the internal `INPUT_PULLUP` resistor. This means that the button or key connects the signal pin to ground when pressed.

1.8 Straight key

The straight key is connected to a 3.5 mm stereo jack as follows:

Jack part	Connection
TIP	GPIO12
RING	Not used
SLEEVE	GND

In the code, the straight key is:

```
const int STRAIGHT_KEY_PIN = 12;
```

1.9 Iambic key

The iambic key is connected to a 3.5 mm stereo jack as follows:

Jack part	Connection
TIP	GPIO9, DIT
RING	GPIO10, DAH
SLEEVE	GND

In the code, the pins are:

```
const int DIT_PIN = 9;
const int DAH_PIN = 10;
```

If DIT and DAH are reversed, you can fix it either by swapping the wires in the jack or by using the program's `swapPaddles` setting.

1.10 Rotary encoder

The recommended encoder is a KY-040 or similar model with a push button. The wiring is shown below.

1.10.1 Top row, two pins, push button

Encoder pin	Connection
One pushbutton pin	GPIO13
Other pushbutton pin	GND

1.10.2 Bottom row, three pins

Encoder pin	Connection
Left	GPIO5, CLK or A
Middle	GND
Right	GPIO6, DT or B

If the rotation direction is reversed, swap GPIO5 and GPIO6.

In the code, the encoder pins are:

```
const int ENC_CLK_PIN = 5;
const int ENC_DT_PIN = 6;
const int ENC_SW_PIN = 13;
```

1.11 Sidetone and headphone output

The sidetone is produced from ESP32 GPIO11 as a PWM square wave. It is not intended to drive a speaker directly, but to provide a headphone-level signal.

In the code, audio is:

```
const int AUDIO_PIN = 11;
```

Volume is adjusted with a 10 kΩ logarithmic mono potentiometer.

Potentiometer wiring:

Potentiometer terminal	Connection
Left outer terminal	GPIO11
Right outer terminal	GND
Middle terminal, wiper	Headphone output junction

If the volume control works backwards, meaning the sound gets louder when it should get quieter, swap the wires on the two outer potentiometer terminals.

Resistors from the middle terminal:

From the middle terminal	To
1 kΩ resistor	Headphone jack TIP
1 kΩ resistor	Headphone jack RING
2.2 kΩ or 10 kΩ resistor*	GND

The two 1 kΩ resistors are separate. They are not in series with each other.

The headphone jack SLEEVE or GND goes directly to ground without a resistor.

* If the sidetone can still be heard in the headphones even when the potentiometer is fully at zero, try a larger resistor to ground.

For example:

- 2.2 kΩ may allow a little sound to leak through in some setups
- 10 kΩ may reduce leakage more effectively

The suitable value depends on the potentiometer, headphones, and headphone jack structure being used.

Simplified wiring:

```
GPIO11 -> potentiometer outer terminal
GND    -> other potentiometer outer terminal

Potentiometer middle terminal
  ├── 1 kΩ -> headphone jack TIP
  ├── 1 kΩ -> headphone jack RING
  └── 2.2 kΩ or 10 kΩ -> GND

Headphone jack SLEEVE -> GND
```

1.12 Headphone detection, optional

The project reserves support for a switched 3.5 mm headphone jack. The idea is that the headphone jack can pull the A3 pin to ground when headphones are connected.

In the code, this appears for example as:

```
const int HEADPHONE_DETECT_PIN = A3;

bool headphonesConnected() {
  return digitalRead(HEADPHONE_DETECT_PIN) == LOW;
}
```

The purpose of the feature was to allow behaviour such as playing the sidetone only when headphones are plugged in.

However, this feature is not currently in active use in the project. It is mainly a reserved extension option for future use.

If you use a standard 3.5 mm headphone jack without a detection switch, you can leave the whole detection feature unused by commenting out the pin definition:

```
// const int HEADPHONE_DETECT_PIN = A3;
```

In that case, a normal headphone jack works normally without the detection circuit.

1.13 3D-printable enclosure

Enclosure file:

`Morsewurst_keyer.stl`

Recommendation:

- Material: PETG
- Layer height: about 0.2 mm
- Infill: about 20 to 30 %
- Walls: 2 to 3 perimeters
- Printer: for example Prusa i3 MK3S Plus or similar

If the printer accuracy is not good enough for all holes directly, the holes should be drilled or cleaned up carefully by hand. Do not use too much force, or the enclosure may crack.

The display is physically installed upside down in the enclosure because the internal space is limited. The image is rotated back to the correct orientation in software with the `U8G2_R2` setting.

1.14 Telemetry for Morsewurst

The device sends timing events over USB CDC Serial in JSON format. Morsewurst reads these lines and uses them to analyse timing, rhythm, and decoding during practice.

Example tone event:

```
{
  "v": 1,
  "type": "tone",
  "src": "straight",
  "t0": 123456789,
  "t1": 123556789,
  "dur": 100000
}
```

In iambic mode, the element and unit length may also be included:

```
{
  "v": 1,
  "type": "tone",
  "src": "iambic",
  "el": ".",
  "t0": 123456789,
  "t1": 123516789,
  "dur": 60000,
  "unit": 60000,
  "wpm": 20.0
}
```

The important point is that `t0`, `t1`, and `dur` are values measured by the ESP32. USB and Python latency affects when the event appears on the computer, but it does not change these values already measured by the ESP32.

1.15 Measurement accuracy

The code uses the ESP32's microsecond-resolution timing system:

```
uint64_t nowTime() {
  return (uint64_t)esp_timer_get_time();
}
```

Timestamps are stored in microseconds, but the practical accuracy of the whole system is not one microsecond. With the current polling-based implementation, the realistic practical accuracy in good conditions is about:

```
50 to 200 microseconds
```

This is more than accurate enough for Morse practice, because the timing variation in human sending is usually much larger than the measurement error of the device.

If even more accurate edge timing is wanted for the straight key in the future, the next development step would be GPIO interrupt-based measurement. In that case, mechanical key contact bounce must also be handled.

1.16 USB HID Keyboard mode

The device can send decoded characters to the computer as a USB keyboard.

This is convenient if the device is to be used directly for text input. For special characters, the code assumes the Finnish keyboard layout. If the computer uses, for example, an English keyboard layout, some characters may be produced incorrectly.

The mode can be enabled or disabled from the device settings.

1.17 Build order

1. Load `Morsewurst_keyer.ino` into the Arduino IDE
2. Select the correct ESP32-S3 board and USB settings
3. Test ESP32 programming with a USB-C cable
4. Connect the OLED display and make sure the image appears correctly
5. Connect the rotary encoder and test menu control
6. Connect the straight key to GPIO12 and GND
7. Connect the iambic paddle to GPIO9, GPIO10, and GND
8. Build the sidetone circuit with the potentiometer and resistors
9. Test the headphone output at low volume
10. Test USB serial telemetry in Morsewurst
11. Test USB keyboard mode if needed
12. Print the enclosure from PETG
13. Clean up the holes carefully
14. Solder the final wiring
15. Install the parts in the enclosure
16. Do a final test before closing the enclosure

1.18 Safety and reliability notes

- Use 3.3 volt logic
- Do not feed 5 volts into ESP32 GPIO pins
- Do not drive a speaker directly from GPIO11
- Start headphone tests at low volume
- Make the final wiring as short and mechanically solid as possible

- Avoid loose connectors in a tight enclosure
- Clearly mark GND wires
- Test each part separately before final assembly

1.19 Files

File	Purpose
morsewurst_keyer_*_*.ino	Arduino firmware
Morsewurst_keyer.stl	3D-printable enclosure
Morsewurst	Practice program

1.20 Summary

I originally made the Morsewurst Keyer for my own Morse practice. The whole design philosophy of the project was that I wanted to make my own sending as measurable, analysable, and visualisable as possible instead of practice being deadly boring tapping without any information about how well I was doing.

The ESP32-S3 measures key press timings in microseconds and sends the events as JSON telemetry to Morsewurst. Morsewurst can then analyse rhythm, timing, errors, and other details much more precisely than one would notice by listening alone.

As a nice bonus, the device can also work as a USB keyboard if the setting is enabled. In that case, the sent Morse text is typed directly into the computer.

I strongly recommend building and testing the whole device on a breadboard before doing any final soldering. Once the display, keys, rotary encoder, sidetone, and USB connections work reliably, the final version should be made so that all wires are soldered directly to the ESP32-S3 board. In a tight enclosure, loose wires and loose connectors can easily cause contact problems.

If you want to use the ready-made enclosure, the `Morsewurst_keyer.stl` file is included. The enclosure is designed for PETG printing. In practice, however, its fit requires that the components used are very close to the same ones as in this project. Even small differences in the dimensions of the display or connectors may prevent the parts from fitting correctly.

Fortunately, designing a similar enclosure yourself is quite easy today with tools such as Fusion 360 or Blender.

In this enclosure solution, the display is physically installed upside down to save space. The display image is rotated back to the correct orientation in software with this U8g2 setting:

```
U8G2_SH1106_128X64_NONAME_F_HW_I2C u8g2(
  U8G2_R2,
  U8X8_PIN_NONE
);
```

`U8G2_R2` rotates the image 180 degrees.

If you make your own enclosure and wonder why the display appears upside down, change that line for example to:

```
U8G2_SH1106_128X64_NONAME_F_HW_I2C u8g2(
  U8G2_R0,
  U8X8_PIN_NONE
);
```

Then the display is no longer rotated in software.

1.21 Serial API for compatible devices

Morsewurst's most important hardware interface is line-based JSON telemetry sent over a USB CDC Serial connection.

This means that a Morsewurst-compatible device does not need to be the original ESP32-S3-based Morsewurst Keyer. Any microcontroller, Arduino-compatible device, or other custom hardware solution can work if it can send correctly formatted JSON lines over a serial port.

The interface is effectively one-way. The device sends timing data to Morsewurst. Morsewurst does not need to send commands back to the device.

1.21.1 Serial connection

The recommended baud rate is:

`115200 baud`

The device sends one JSON object per line.

Each message is terminated with a newline:

`\n`

In practice, Morsewurst reads the serial port line by line and tries to parse each line as a JSON object.

1.21.2 Device identification

Morsewurst can identify the correct serial port based on Morsewurst-compatible JSON messages arriving from the port.

Useful messages for identification are:

```
hello
heartbeat
tone
```

A good compatible device sends a `hello` message after startup or after the serial connection is opened, and sends a `heartbeat` message regularly even when the key is not being pressed.

This helps Morsewurst find the correct COM port or serial port even when the user is not sending Morse at that moment.

1.21.3 Hello message

`hello` indicates that the port contains a Morsewurst-compatible device.

Example:

```
{"v":1,"type":"hello","app":"morsewurst","device":"Morsewurst","fw":"1.0","mode":"raw_timing"}
```

Fields:

Field	Type	Meaning
`v`	number	Protocol version. The current version is `1`.
`type`	string	Message type. Here, `hello`.
`app`	string	Application identifier. Recommended value is `morsewurst`.
`device`	string	Device name.
`fw`	string	Firmware or software version.
`mode`	string	Telemetry mode. Recommended value is `raw_timing`.

1.21.4 Heartbeat message

`heartbeat` indicates that the device is still connected, working, and sending raw timing telemetry.

The recommended sending interval is about 5 seconds.

Example:

```
{"v":1,"type":"heartbeat","app":"morsewurst","device":"morsewurst","fw":"1.0","mode":"raw_timin
```

Fields:

Field	Type	Meaning
`v`	number	Protocol version.
`type`	string	Message type. Here, `heartbeat`.
`app`	string	Application identifier. Morsewurst expects the value `morsewurst`.
`device`	string	Device name.
`fw`	string	Firmware or software version.
`mode`	string	Telemetry mode. Recommended value is `raw_timing`.
`uptime`	number	Device uptime in microseconds.
`wpm`	number	The device's current WPM setting, if one exists.
`telemetry`	boolean	`true` when raw timing telemetry is enabled.

`heartbeat` does not replace actual timing events, but it is important for practical compatibility. Without it, Morsewurst may only find the device when the first `tone` event arrives.

1.21.5 Tone message

The most important message for Morsewurst's timing analysis is `tone`.

`tone` reports the timing of one sound, key press, or Morse element.

Minimum form:

```
{"v":1,"type":"tone","src":"straight","t0":1000000,"t1":1100000,"dur":100000}
```

Fields:

Field	Type	Required	Meaning
`v`	number	yes	Protocol version. The current version is `1`.
`type`	string	yes	Message type. For timing events, the value is `tone`.
`src`	string	yes	Event source, for example `straight` or `iambic`.
`t0`	number	yes	Start time of the key press or sound in microseconds.
`t1`	number	yes	End time of the key press or sound in microseconds.
`dur`	number	yes	Duration in microseconds. Usually `t1 - t0`.

If the device only sends raw timing data from a straight key, this minimum form is enough.

1.21.6 Timestamps

Timestamps must be microsecond timestamps measured by the device itself.

They do not need to be real clock time. They can be microseconds since the device started.

The important point is that the values are:

- microseconds
- integers

- from the same device's own clock
- monotonically increasing
- mutually comparable during the same practice session

If the compatible device is made with an ESP32, a good way to get the time is `esp_timer_get_time()`, because it returns microsecond time.

Example of an ESP32-style implementation:

```
uint64_t nowTime() {
  return (uint64_t)esp_timer_get_time();
}
```

If the device is made with another microcontroller, it does not need to use the same function. It is enough that it can send reliable and monotonically increasing microsecond timestamps.

If possible, timestamps should be sent as 64-bit integers so that overflow does not occur quickly.

1.21.7 Straight key events

For a straight key, the recommended source value is:

`straight`

For a straight key, the device does not need to decide whether the press was a dot or a dash. It is enough to send the start time, end time, and duration of the press.

Example:

```
{"v":1,"type":"tone","src":"straight","t0":123456789,"t1":123556789,"dur":100000}
```

Morsewurst can then estimate from the timing whether the event was a dot, dash, letter gap, or word gap.

1.21.8 Iambic key events

If the device implements the iambic keyer logic itself, it can also send information about whether the produced element was a dot or a dash.

For an iambic key, the recommended source value is:

`iambic`

Example of a dot:

```
{"v":1,"type":"tone","src":"iambic","el":".","t0":1000000,"t1":1060000,"dur":60000,"unit":60000
```

Example of a dash:

```
{"v":1,"type":"tone","src":"iambic","el":"-","t0":1200000,"t1":1380000,"dur":180000,"unit":6000
```

Additional fields:

Field	Type	Meaning
`el`	string	Morse element, either `.` or `-`.
`unit`	number	Length of one dit unit in microseconds.
`wpm`	number	Speed in words per minute.

In iambic mode, `unit` can be calculated like this:

```
unit = 1200000 / wpm
```

At 20 WPM, the length of one dit unit is:

`60000 µs`

1.21.9 JSON format

Numeric values are sent as JSON numbers, not strings.

Correct:

```
{"t0":123456789,"dur":100000}
```

Not like this:

```
{"t0":"123456789","dur":"100000"}
```

Strings are sent as normal JSON strings.

Correct:

```
{"type":"tone","src":"straight"}
```

Each JSON object must be sent on its own line.

Correct:

```
{"v":1,"type":"heartbeat","app":"morsewurst","device":"Morsewurst","fw":"1.0","mode":"raw_timin
{"v":1,"type":"tone","src":"straight","t0":6000000,"t1":6100000,"dur":100000}
```

1.21.10 Practical minimum implementation

The simplest compatible device does this:

1. Opens a USB CDC Serial connection at 115200 baud
2. Sends a `hello` message when the connection is available
3. Sends a `heartbeat` message regularly
4. Measures the start time of the key press in microseconds
5. Measures the release time of the key press in microseconds
6. Calculates the duration of the press in microseconds
7. Sends one `tone` message for each press

Minimum example:

```
{"v":1,"type":"hello","app":"morsewurst","device":"Morsewurst","fw":"1.0","mode":"raw_timing"}
{"v":1,"type":"heartbeat","app":"morsewurst","device":"Morsewurst","fw":"1.0","mode":"raw_timin
{"v":1,"type":"tone","src":"straight","t0":6000000,"t1":6100000,"dur":100000}
```

1.21.11 Plain text serial output is not enough

Plain text serial output alone is not enough for Morsewurst's raw timing analysis.

For example, this is not enough:

```
HELLO WORLD
```

Nor is sending only one decoded character at a time:

```
H
E
L
L
O
```

Raw timing analysis needs the press timings:

```
t0
t1
dur
```

Without these, Morsewurst cannot reliably analyse keying rhythm, dot and dash durations, letter gaps, or word gaps.

1.21.12 USB HID Keyboard mode

The original Morsewurst ESP32 firmware also includes a USB HID Keyboard mode. It allows the device to type decoded characters directly into the computer like a keyboard.

This is mainly an extra fun feature. It should not be treated as a serious Morse decoder, because the device's own real-time decoding can make mistakes and keyboard text input is not as useful as Morsewurst's raw timing analysis.

For Morsewurst, USB HID Keyboard mode is not required. The actual compatibility is based on USB CDC Serial telemetry.

1.21.13 Recommended API-compatible message sequence

A good Morsewurst-compatible device sends, for example, a sequence like this:


```
{"v":1,"type":"hello","app":"morsewurst","device":"Morsewurst","fw":"1.0","mode":"raw_timing"}
{"v":1,"type":"heartbeat","app":"morsewurst","device":"Morsewurst","fw":"1.0","mode":"raw_timin
{"v":1,"type":"tone","src":"straight","t0":6000000,"t1":6060000,"dur":60000}
{"v":1,"type":"tone","src":"straight","t0":6120000,"t1":6300000,"dur":180000}
{"v":1,"type":"tone","src":"straight","t0":6360000,"t1":6420000,"dur":60000}
{"v":1,"type":"heartbeat","app":"morsewurst","device":"Morsewurst","fw":"1.0","mode":"raw_timin
```


When a device sends lines like these over a USB CDC Serial connection, Morsewurst can read the events, identify the device, and analyse Morse timing regardless of which microcontroller or electronics the device is built with.