IMP Documentation

Tomáš Brablec (xbrabl04)

Contents

1 Introduction

The goal is to create an FM radio based on the ESP32 microcontroller, RDA5807M tuner module, PAM8403 audio amplifier, SSD1306 display module and a rotary encoder for user input. The radio should be able to tune to different stations in the range of 76-108MHz, seek stations automatically, display RDS information about the current station, have configurable volume, and persistent preset slots for storing known stations.

1.1 Link to Video

The video presentation of the complete project is hosted on YouTube at the following address: https://youtu.be/Rc2GACCjUhw

2 Platform and Technology

I decided to create the microcontroller software in Rust using the ESP-IDF platform [1]. This choice is based on my familiarity with the Rust programming language and the fact that Espressif provides first-party support for building Rust projects on their microcontrollers. Also, Rust provides memory safety and better developer experience overall than platforms based on C, which is crucial in embedded projects, since the debugging capabilites of the microcontroller are limited.

I chose to utilize existing drivers for the RDA5807 module [2] and for the SSD1306 display module [3]. I had to add a few functions to the tuner driver to get access to RDS data, the forked driver is available at [4].

I also used the `embedded_graphics` package [5], which provides function for drawing simple shapes to a buffer, such as rectangles, circles, or monospace text. These are used to draw every element of the GUI.

3 Physical Construction

All wiring is displayed in the diagram below. In short, the tuner is connected to ESP32 via I2C, the display is connected via SPI and the rotary encoder is connected simply to GPIO pins. All pins are labeled according to the real components.

The amplifier is not the same chip as the one included in the kit, since the one included in the kit had only one functional channel and I was not able to find an exact replacement. This amplifier does not have configurable volume, but we can configure volume on the tuner itself.

All components except the amplifier are connected to the same power source (that is the USB powering the ESP32 itself). The amplifier is powered from a separate source to avoid interference. When testing the project, the audio would contain loud noise whenever the CPU on ESP32 was transmitting data over the serial interface, therefore I separated the amplifier on a different circuit. However, the audio ground input on the amplifier is still connected to the ESP32 ground.

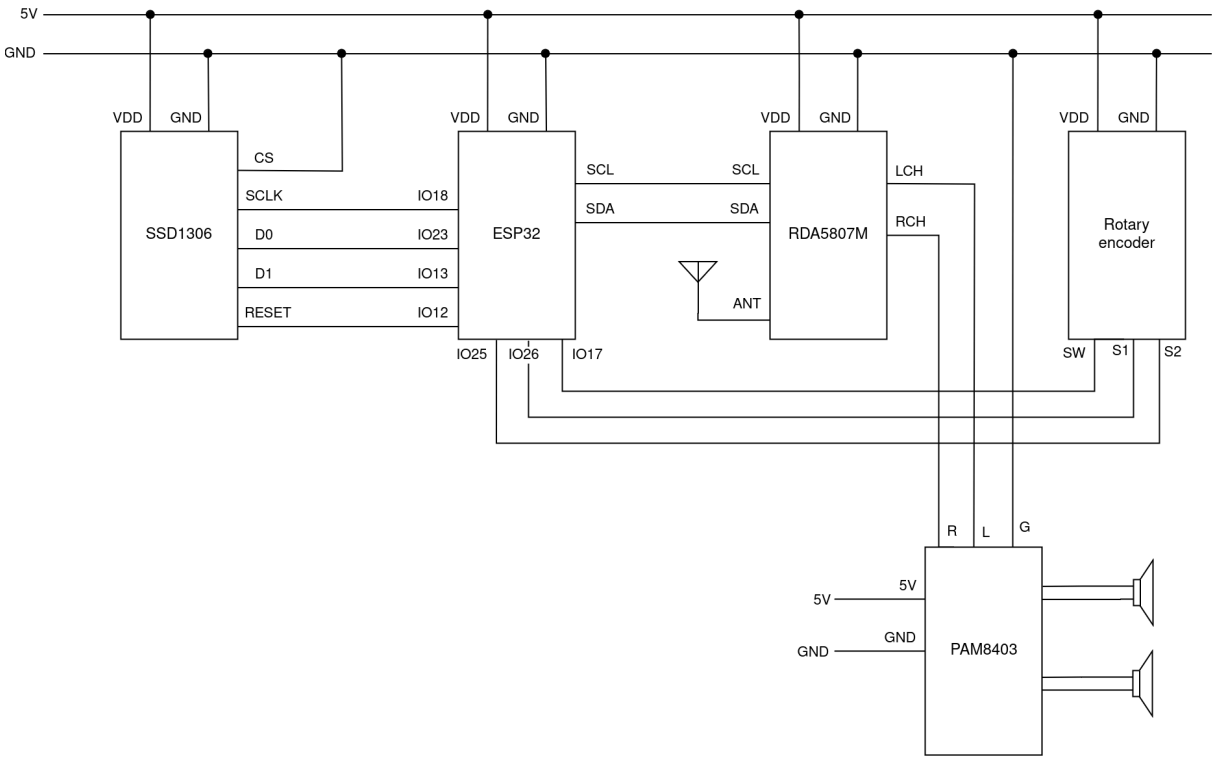

Figure 1: Wiring diagram of the radio

4 Software Architecture

4.1 Threads

The project requires to continually watch for events on different inputs. In a C project, this would best be handled by using interrupts on all the inputs. However, this would require using unsafe code, and the ESP-IDF toolchain provides a better abstraction anyway.

There are threads (`std::thread`) (internally represented as RTOS tasks), and a wait function (`button.wait_for_rising_edge`) which puts a thread to sleep until a rising/falling edge on a GPIO pin is detected. This function internally uses an interrupt handler to wake up the task, and therefore does not waste CPU resources by busy-waiting in a loop.

The application must have an event loop, which handles input events, performs actions and updates the GUI accordingly. I decided to implement the software according to the MVC design pattern, where the application state, event handling logic, and UI rendering logic are clearly separated.

The program will use 4 threads:

- Main thread, with the application state, event handling and rendering logic
- IO thread for rotary encoder
- IO thread for rotary encoder button
- Tuner thread, which polls the tuner for updates of RDS information

4.2 Communication

Communication between threads is handled using channels (`std::sync::mpsc::channel`) to send input events and output commands to and from the main thread. A channel is a synchronization primitive in Rust which allows us to send data between threads. The data arrives in-order, therefore the channels also function as event and command queues.

The *input events* (`enum InputEvent` in `main.rs`) are sent from IO and tuner threads to the main thread, and encode messages about user inputs and updates from the tuner:

- rotary encoder turns
- rotary encoder button presses
- change of tuner frequency during automatic seek
- change of RDS information

Output commands (`enum OutputCommand` in `main.rs`) are sent from the event handling logic on the main thread to the tuner, and include:

- command to automatically seek stations
- command to change frequency
- command to change volume

4.3 User Interface

The user interface consists of distinct UI elements, each of which can be selected by a cursor. The cursor is a rounded rectangle, which appears around the element when it is selected. The cursor can be moved across all the UI elements by turning the rotary encoder.

UI elements can be either buttons, or value setters. Buttons (seek buttons, presets) are activated simply with a press of the rotary encoder button. Value setters (frequency and volume selection) are activated with the press of the button, then the value is set by turning the rotary encoder, and with a second press, the element is deactivated. Active element is displayed with a thick border.

4.4 Application State

The application stores the state in `struct AppState` (`main.rs`), which includes the following data:

- current frequency and volume settings
- current RSSI and station information
- which UI element is currently selected by the cursor
- whether the element is activated

4.5 Non-Volatile Storage

To store preset stations across device resets, the program uses Non-Volatile Storage (NVS). NVS uses a partition on the ESP32 flash to permanently store data. It is accessed through ESP-IDF API (`esp_idf_svc::nvs`).

5 Implementation

The code is divided into six modules. The program entrypoint is in `main.rs`.

5.1 Main module (`main.rs`)

The main module uses other modules to initialize all hardware, spawn other threads, and contains the event loop. It also creates the input event and output command channels. The event loop looks like this:

```
while let Ok(event) = event_receiver.recv() {
    state.process_event(event, &command_sender, &mut nvs);
    state.update_ui(&mut display).unwrap();
}
```

- `event_receiver.recv()` waits until an input event arrives (the thread is parked until that moment)
- `state` is the `AppState` struct holding the program state, and the method `process_event` implements the event handling logic. It gets access to the sending end of output command channel, and to the NVS.
- the `update_ui` method redraws the UI on the display, it gets access to the display driver only.

5.2 Display module (`display.rs`)

This module implements the function `setup_display`, which returns the display driver struct. It handles the initialization of the SPI driver, the display driver itself, and the reset of the display module using the RESET pin connected to GPIO12. Without this, the display would not work. The display driver object itself is boxed (stored on the heap), because it contains a buffer with the content of the display (writes to the display are buffered). Without boxing the driver, the program would overflow the stack. After driver initialization, the display is cleared.

5.3 Input module (`input.rs`)

The module implements two functions. The function `spawn_button_listener` spawns a new thread, where it waits for falling edge on the GPIO input connected to the rotary encoder button. Then it waits for rising edge (the button is pulled high), measures the press length, and then either discards the press as button bounce (<50 ms), sends a `ShortPress` input event (50 to 600 ms), or a `LongPress` input event (>600 ms).

The second function, `spawn_encoder_listener`, spawns a new thread that waits for rising edge on the `S1` pin of the rotary encoder. Then it compares the levels of `S1` and `S2`, and sends a `ScrollUp` event if they are the same, and `ScrollDown` event if they differ.

Every second rising edge is discarded. this is done by the physical construction of the encoder, not by any bouncing or similar effect. In my testing, the rotary encoder worked well with this setup. Note that I used a different model of rotary encoder than the one included in the kit.

5.4 GUI module (`gui.rs`)

This module implements the `update_ui` method on the `AppState` struct, which draws the UI according to the state. It uses the `embedded_graphics` package for drawing primitives. The font for displaying RDS information is used in the ISO-8859-2 variant, which should be compatible with Czech radio stations. However, I was not able to find a single station which would transmit RDS information in anything other than plain ASCII.

5.5 Event processing (`state.rs`)

This module implements the `process_event` method on `AppState`, which updates the application state based on the input event, and optionally sends output commands to the tuner. It is implemented using pattern matching on relevant data.

5.6 Tuner control (`tuner.rs`)

The module implements the function `spawn_tuner_thread`, which initializes the I2C driver, tuner driver and the tuner module itself.

Then, it continually reads data from the tuner and when they change, sends the update as an input event. This polling is done in a loop with a 100ms sleep at the end, therefore the CPU is not busy all the time. The observed data include:

- current frequency

- current RSSI value
- current RDS information, more precisely the Radio Text (RT) value, which commonly contains the station name, or the name of the current programme.

Reading the RDS data is done by first reading the RDSA to RDSD registers, which contain the raw decoded blocks of RDS data. Then the group type is parsed from block B, and if the value is 2 (Radio Text), the characters from blocks C and D are inserted into an array of characters according to the offset given in block B. The parsing logic is implemented according to [6].

At the beginning of the loop, the output command queue is checked without any waiting (`try_recv`), and if a command is available, it is processed.

6 Auto-evaluation

- F - Functionality - 4 points (the amplifier does not have configurable volume, implemented using tuner instead)
- E - Approach to solution - 2 points (code architecture is based on established design patterns, implemented using a modern toolchain)
- D - Documentation - 3 points (documentation covers all points of implementation)
- P - Presentation - 2 points (video shows off all implemented functionality)
- Q - Quality of solution - 2 points (user interface is comfortable to use, code uses established design patterns and is well commented)

Bibliography

[1] "The Rust on ESP Book." [Online]. Available: https://docs.esp-rs.org/book/introduction.html

[2] "rda5807m driver for Rust.." [Online]. Available: https://crates.io/crates/rda5807m

[3] "I2C/SPI driver for the SSD1306 OLED display controller." [Online]. Available: https://crates.io/crates/ssd1306

[4] "rda5807m driver fork." [Online]. Available: https://github.com/pepega007xd/rda5807m

[5] "Embedded graphics library for small hardware displays." [Online]. Available: https://crates.io/crates/embedded-graphics

[6] "Radio Data System — Wikipedia, The Free Encyclopedia." [Online]. Available: https://en.wikipedia.org/w/index.php?title=Radio_Data_System&oldid=1263044522