Design and Programming of a Custom Radio-Enabled PCB

Maanas Purushothapu

April 2021

Contents

1 Introduction

This document will discuss a PCB designed to communicate with a robot over radio using a radio module and a microcontroller. The radio board, or Radii as it is colloquially called, was originally designed to easily troubleshoot electrical components and change strategies during competition matches. This particular board was built to interface with a robot participating in the Japanese RobotSumo competition, but this is in no way a restriction, and with a few design modifications, this board can be used to interface with a variety of other robots participating in other competitions.

2 Parts Selection

2.1 Microcontroller

This board uses an AtMega 32u4 microcontroller to control all the devices, which is common among Arduino boards and other designs due to its performance and versatility. The microcontroller was chosen for its capability to interface with the Arduino IDE, simplifying the process of uploading programs. It was also chosen among other microcontrollers used in Arduino products for its ability to natively interface with USB. Its form factor and voltage versatility were also factors that influenced the choice.

2.2 Radio Module

To enable radio communication, the board uses a SparkFun RFM69HCW radio module, which is a miniature circuit that has been packaged to an easily solderable surface-mounted form. This radio module has a variety of frequency configurations, including 433 MHz and 915 MHz; this board uses the 915 MHz version. An important limitation of this board is that its input voltage is limited to 3.3V, which introduces some complications described later in this document. Both the transmitter and the receiver must have this module connected with the same frequency for radio transmission to work, and as such, recent versions of control boards have this same part integrated onto them. Those designs can be found on the same GitHub repository as this document.

2.3 Display

To display data, this board uses a 240 x 240 Wide Angle TFT LCD by Adafruit. This display was chosen for its size, ease of programming (as it uses a standard SPI interface), and the existence of Arduino libraries to make programming the display significantly easier.

2.4 Input Devices

For users to interact with the robot via the radio, the board has built-in slots for five momentary pushbuttons to be connected by soldering the ends of wires connected to them. These buttons are general purpose inputs and can be programmed as such. Each button is connected to a pull-down circuit, so no extra circuit is required when connecting a pushbutton.

3 Design

3.1 Power

Two out of the three main electronic components, the microcontroller and the display, can be used with either 3.3V or 5V. As such, the design for board voltage came down to the limitations of the RFM69HCW radio module, which cannot operate beyond 3.3V. The overall PCB accepts any power voltage above 3.3V - such as a 9V battery - and converts this into 3.3 V using a buck switching regulator arranged with other parts in a circuit shown below.

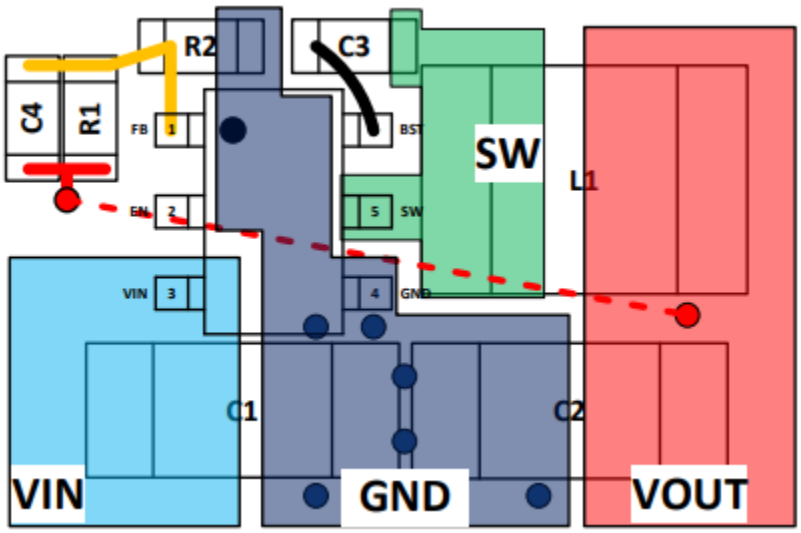

Figure 1: Layout of power regulator, associated parts, and planes

The regulator feeds power to the radio module, the microcontroller, and the display, and is the only power IC on the board.

3.2 Parts Arrangement and Connections

As the microcontroller has to connect to most of the other electronics, the board was designed around it. As such, the microcontroller is in a center position, where it be accessed by the radio module, the display, the input pushbuttons, and the power regulators, which are placed around it. The RFM69HCW is arranged such that the antenna pin is close to an edge or a corner, where a wire is soldered to a connected hole to serve as an antenna. The display, similarly to the pushbuttons, will be externally connected with wires instead of being directly soldered to the board, and so the display connection pins are also on the edge of the board.

To make board layout as simple as possible, all power-related components are organized in one area; this includes the main power connectors, the buck regulator and associated components, and the power LEDs. A reset button for the microcontroller is also placed in the same area, but this placement was a result of the button being added in the late stages of the design process. This arrangement of power components simplifies the arrangement of the power planes.

To limit the number of connections to power and ground for each part, including integrated circuits and capacitors, the top layer of the board has a 3.3V plane that connects all associated pins automatically. The

bottom layer of the board similarly has a ground plane, so to connect a pin to ground there simply needs to be a via connecting that part and the botton plane. Using power planes significantly simplifies routing by preventing complex webs of power connections between all parts.

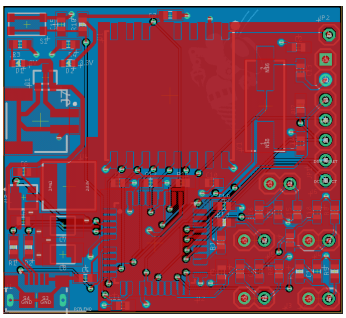

Figure 2: Example of planes. The giant red connection is a 3.3V plane that connects most 3.3V connections easily. The thinner red and blue connections are traces that individually connect pins.

Almost all power pins on the microcontroller, display, and radio module have decoupling capacitors, which are capacitors placed between the connected power line and ground. Using decoupling capacitors reduces noise in the input power lines, whether it be voltage or current spikes or drops. Unlike many connections, which can use routing lines without a significant restriction for length, decoupling capacitors should always be placed as close as possible to their connected pins to maximize their effectiveness.

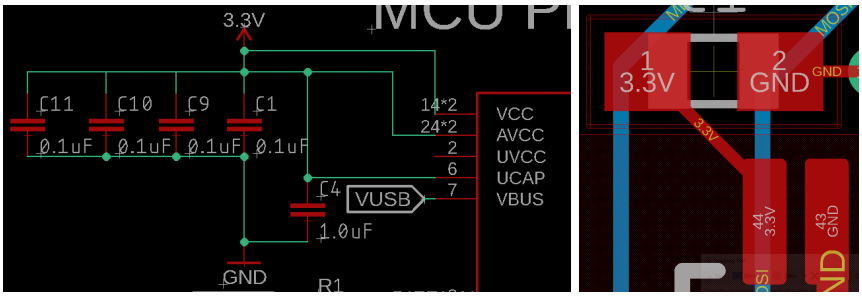

Figure 3: Example of decoupling capacitors. The schematic shows the connections for these capacitors, 1 per pin, and the layout shows one of these capacitors. Notice the proximity of the capacitor to the pin.

3.3 Design Errors

This board served as a good learning tool by not only being an exercise in schematic design and layout but also by serving as a reminder to always consider usage. The main source of these errors was failing to check and confirm which specific pins to connect to in order to upload a bootloader, upload code, and interface with inputs with interrupts correctly.

3.3.1 Lack of Access to Bootloader Pins

As discussed in section 4.1, an uploaded bootloader file allows a microcontroller to accept and run Arduino code. To upload a bootloader file to an initially empty microcontroller, six pins need to be accessed: SPICLK, MOSI, MISO, RESETN, Power, and Ground. During the design phase, it was planned to use the SPI connections for the display to also upload the bootloader; however, the obvious but missed flaw was that the

display connections did not include access to the MISO or RESETN pins. As such, wires had to be soldered to pads on the board to access these pins.

3.3.2 Voltage Choice

As discussed in section 3.1, a voltage of 3.3V was used throughout the entire board to simplify the power connections. However, this brings problems when connecting the circuit for uploading a bootloader file, especially if using another 5V Arduino, such as an Arduino Uno, as the programmer. As such, when uploading a bootloader, using a bidirectional level shifter is critical.

Another limitation of using 3.3V instead of 5V is that the board is not powered by USB and instead requires an external power source even when being programmed by USB, which becomes a hassle when programming and testing the board. While adding this capability would somewhat increase power design requirements, it is overall an easy implenetation (see the Arduino UNO schematic for details).

The choice of using 3.3V as the main voltage also limited the external oscillator frequency to 8 MHz, which served as a severe limitation to which board the design could emulate (i.e. which bootloader can be used). The only such Arduino-capable device is the SparkFun Pro Micro.

3.3.3 Interrupt Pin Connections

However, by far the largest error of this board was not connecting to correct interrupt pins when required. There are 6 inputs that require interrupts to work optimally, the interrupt pin from the RFM69 module (which is required, as it notifies the MCU when a message has been received), and 5 button inputs (which do not strictly require inputs, but manual polling is not optimal). Many of these inputs are not connected to appropriate interrupt pins (including the radio interrupt, which prevented the code to connect to the radio module from working). A significant limitation of the AtMega is that it has only 5 interrupt capable pins, so designs such as these must work around that; as such, this design cannot work solely with interrupts, and polling must occur. However, with the radio, a wire was manually soldered to an interrupt-capable pin, as polling was not an option.

4 Programming

4.1 Bootloading

Before any code can be uploaded to the MCU via USB, the MCU must first be configured as a SparkFun Pro Micro, and this is done by uploading a bootloader, which is a file that runs on startup or reset. This process involves two steps: enabling the Arduino IDE to work with the Pro Micro, and uploading the bootloader.

Instructions to do the first step can be found here.[1]

The second step is the actual bootloading. An Arduino UNO serves as the programmer, while the target is our MCU. Instructions to do so can be found here[2], but some clarifications:

- This design did not have easy access to the MISO and RESET pins, and wires had to be soldered directly for these connections.

[1] https://learn.sparkfun.com/tutorials/pro-micro–fio-v3-hookup-guide/allinstalling-windows
[2] https://learn.sparkfun.com/tutorials/installing-an-arduino-bootloader/all

- The capacitor on the reset line had to be removed.
- There are two versions of the Pro Micro: a 16MHz 5V one and a 8MHz 3.3V one. This design emulated the latter, and this setting had to be changed appropriately in the Arduino toolbar, under Programming.
- When connecting the SPI connections of the PCB to the Arduino UNO, a 3.3V - 5V shifter MUST be used, or the RFM69 radio module, which is connected to the SPI lines, may be damaged.
- The bootloader had to be loaded twice onto the MCU. After each upload, the PCB was connected to a computer via USB and a simple program was uploaded via the Arduino IDE. THe upload failed after the first bootloader upload but succeeded after the second one.

4.2 Code

Code is provided in the appendix to show how to initialize and use the RFM69 radio module. The code is based of the RFM69 library by Felix Rusu of LowPowerLab, and the GitHub repository of the library can be found here.[3], and instructions to add the library to the Arduino IDE can be found here.[4]. Note that the code in this document has modifications from the SparkFun-provided example code that make it more suitable for use with robots.

5 Results

Due to the aforementioned design flaws, getting the PCB to work as intended was a significant challenge. The first main challenge was trying to get a working bootloader on the MCU with an Arduino UNO. After procuring a level shifter, the bootloader had to be uploaded twice for the MCU to work reliably via USB. Also, to address the lack of interrupt connections, a wire had to be physically soldered between the RFM69's interrupt pin and an unused interrupt pin on the MCU, after which the radio module was able to work with the code in the appendix. Additionally, as only two of the five buttons are connected to interrupt pins, a grand total of three out of five buttons are being used in code, with two using interrupts and one using manual polling; a Pushbutton library[5] provided by Polulu is being used for polling, as it provides support for debouncing buttons.

Despite these flaws, the PCB is able to execute its core functionality of sending and receiving radio messages based on user input. To test this functionality, two PCBs were assembled and then programmed to send messages comprised of user-provided Serial input to each other. This message sharing worked with minimal changes to the code provided in the Appendix, both with and without using acknowledgements. As this design still executes its core functionality with a small number of fixes, it can be considered a successful work-in-progress.

6 References

- M. Grusin. "RFM69HCW Hookup Guide." SparkFun.com. https://learn.sparkfun.com/tutorials/rfm69hcw-hookup-guide/all (accessed April 11, 2021).
- "AP63200/AP63201/AP63203/AP63205" Diodes Incorporated, Plano, Texas, USA. Accessed April 11, 2021. [Online]. Available: https://www.diodes.com/assets/Datasheets/AP63200-AP63201-AP63203-AP63205.pdf

[3]https://github.com/LowPowerLab/RFM69
[4]https://learn.sparkfun.com/tutorials/rfm69hcw-hookup-guide/allrunning-the-example-code
[5]https://pololu.github.io/pushbutton-arduino/class$_p ushbutton.html$

Appendix

A.1 Settings initialization - comes before setup()

```
// Include the RFM69 and SPI libraries:
#include <RFM69.h
#include <SPI.h>

// Addresses for this particular node.
// No nodes should have the same ID
#define NETWORKID     0   // Must be the same for all nodes
#define MYNODEID      1   // Node ID
#define TONODEID      2   // Destination node ID

// RFM69 frequency
//#define FREQUENCY   RF69_433MHZ
#define FREQUENCY     RF69_915MHZ // Using 915 MHz version

// Acknowledge messages among receipt:
#define USEACK        true // Request ACKs

// Create a library object for our RFM69HCW module:
int chip_select = A2;
int interrupt_pin = 7;
RFM69 radio(chip_select, interrupt_pin); // First pin is Chip Select, second is Interrupt
// Pin numbers are assigned as per this design and the Pro
// Micro schematic and will probably be different between designs

// Set up basic struct containing Node ID, time since start in ms,
// and some string
typedef struct {
  int nodeId;
  unsigned long uptime;
  char string[40];
} Payload;
Payload message;
```

Figure 4: Code to initialize settings to prepare radio board. Pin settings may be different for different designs.

A.2 Setup - goes in setup()

```
// Resets RFM69 prior to operation. Resets all queues and
// reinitializes settings
// A1 is the MCU pin connected to the reset pin of the RFM69
// Will probably be different for other designs
int reset_pin = A1;
pinMode(reset_pin, OUTPUT);
digitalWrite(reset_pin, LOW);
delay(5);
digitalWrite(reset_pin, HIGH);
delay(250);
digitalWrite(reset_pin, LOW);
delay(4);

Serial.begin(9600);

// Initialize the RFM69HCW:
radio.initialize(FREQUENCY, MYNODEID, NETWORKID);
radio.setHighPower(); // Always use this for RFM69HCW
```

Figure 5: Code to initialize RFM69. Pin settings may vary between designs and bootloaders.

A.3 Sending a struct - goes in loop()

```
// Message string already set with Serial input and not shown
// Fill in values of sample struct
message.nodeId = MYNODEID;
message.uptime = millis();

// Use sendWithRetry() if using acknowledgements
if (USEACK)
{
    // Pass in mem location of message and number of bytes to send
    if (radio.sendWithRetry(TONODEID, (const void*)(&message),
            sizeof(message)));
      Serial.println("ACK received!");
    else
      Serial.println("no ACK received");
}

// If not using acknowledgements, simply use send()
else // don't use ACK
{
    radio.send(TONODEID, (const void*)(&message), sizeof(message));
}
```

Figure 6: Code to send a custom struct.

A.4 Receiving a struct - goes in loop()

```
if (radio.receiveDone()) // Check if transmission is done
  {
    // Print out the source node ID
    Serial.print(radio.SENDERID, DEC);

    // Check if the size of the struct is what is expected
    if (radio.DATALEN == sizeof(Payload))
    {
      // Set radio data to a Payload struct
      message = *(Payload*)radio.DATA;
    }

    // Acknowledge if requested
    if (radio.ACKRequested())
    {
      radio.sendACK();
    }
  }
```

Figure 7: Code to receive a custom struct.