Drone RF Signal Detector

Your Name Here

July 17, 2025

Abstract

This project implements a multi-band drone signal detector capable of identifying activity in the 2.4 GHz and 5.8 GHz bands. Using an STM32 microcontroller, an RF frontend including a CC2500 SPI transceiver and an AD8317 analog power detector, the system provides real-time alerts via LCD display and buzzer output. This report outlines the design, implementation, testing, and career-relevant learning outcomes of the project.

1 Introduction

The ongoing conflict in Ukraine, particularly the widespread use of drones in modern warfare, has underscored the urgent need for early detection systems. Viral footage on social media—especially of FPV (first-person view) drones targeting soldiers and armored vehicles—highlighted the devastating effectiveness of these technologies. In one video, I observed a soldier taking cover as a drone flew overhead. In his hand was a small device with a single antenna, emitting multiple alarms. This moment inspired the idea for a compact drone detection system that could provide early warning to soldiers or civilians. Even a few extra seconds of notice could mean the difference between life and death.

This project focuses on detecting common drone control and telemetry frequencies, including 2.4 GHz, 5.8 GHz, 915 MHz, and 433 MHz bands. The aim is to build a low-cost, portable, and passive device capable of alerting users to nearby drone activity without emitting signals or interfering with surrounding systems. While commercial drone detection systems exist, they are typically expensive and inaccessible for individual or small-scale use. My goal was to keep the total component cost under $100 USD.

To ensure durability, I selected a compact waterproof Pelican case for the enclosure, making the system suitable for field conditions. The user interface was designed with simplicity and discretion in mind; visual and audible alerts can be disabled to prevent revealing the user's location.

This project also served as an opportunity to deepen my understanding of RF signal behavior and detection techniques. The experience gained will serve as a foundation for developing more advanced and effective versions of the system in future iterations.

- **Motivation:** *(e.g., privacy/security awareness, RF experimentation)*
- **Application:** Drone presence detection using passive RF methods, civilian and military use
- **Features:** LCD display, audible buzzer, real-time RSSI monitoring, self-calibration

2 System Overview

2.1 Block Diagram

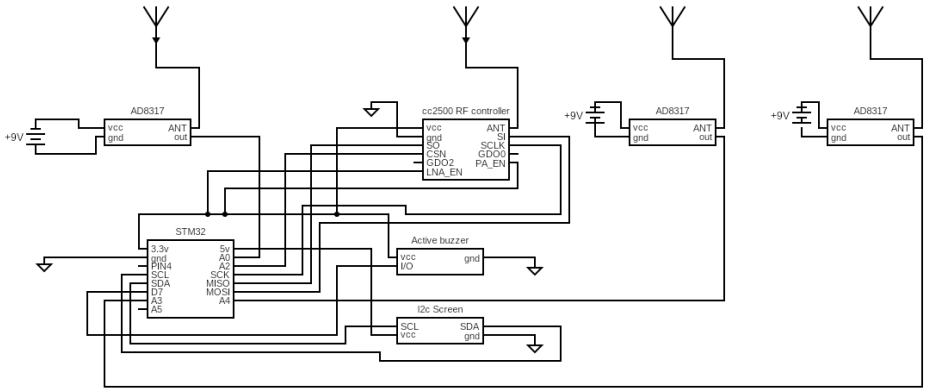

Figure 1: System Block Diagram of RF Detector

2.2 Frequency Bands Covered

- **433 MHz:** Detected using an AD8317 analog RF power detector with tuned antenna
- **915 MHz:** Detected using an AD8317 analog RF power detector with tuned antenna
- **2.4 GHz:** Detected using the CC2500 SPI transceiver (replaced with AD8317 in final build)
- **5.8 GHz:** Detected using the AD8317 logarithmic analog power detector

3 Hardware Design

3.1 Microcontroller and Peripherals

- STM32L4 Nucleo development board
- Passive buzzer (TMB12A05) for tone generation
- QAPASS I2C LCD screen for real-time output

3.2 RF Components

- **433 MHz AD8317**: Analog RF detector with band-specific antenna (SMA)
- **915 MHz AD8317**: Analog RF detector with tuned antenna and SMA to U.FL connector
- **2.4 GHz**: Initially used CC2500 SPI transceiver; later replaced with AD8317 due to reliability

- **5.8 GHz AD8317**: Used for detecting FPV and drone downlink signals in the upper band

3.3 Shielding and Enclosure

- Internally copper-taped Pelican case
- 3D-printed shield box wrapped in copper tape for each AD8317 detector
- All shields grounded to battery negative to ensure effective EMI suppression

4 Software Implementation

4.1 Firmware Overview

- C code developed in STM32CubeIDE
- Peripheral drivers for SPI, ADC, I2C, and GPIO
- Modularized: `cc2500.c`, `char_lcd.c`, `buzzer.c`
- LCD messages for calibration, signal detection, and debug output

4.2 Key Functional Elements

- `CC2500_SweepAndDetect()` — 2.4 GHz scanning logic
- `LogDetector_RecalibrateNoiseFloor()` — ADC averaging for noise baseline
- Button-triggered test via PC13 interrupt (calls `CC2500_RunSignalTest()`) -debugging

4.3 RSSI-Based Detection

- CC2500 RSSI readback from ‘0x34‘ register
- AD8317 voltage sampled via ADC1
- Threshold set as average noise + margin (can be adjusted for accuracy)
- Buzzer and LCD triggered if RSSI or ADC voltage crosses threshold

4.4 Code (C language)

Snippet from main (Includes AD8317 control)

```
int main(void)
{

  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

```

```
  /* MCU Configuration
     ----------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface
     and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */
  RecieveHandles(&hi2c1);
  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_ADC1_Init();
  MX_I2C1_Init();
  MX_SPI1_Init();
  MX_TIM1_Init();
  /* USER CODE BEGIN 2 */

  Buzzer_Init();
  //play tone to signal turn on
  Buzzer_On(440); // Tone 1: A4
  HAL_Delay(100);
  Buzzer_Off();

  Buzzer_On(523); // Tone 2: C5
  HAL_Delay(100);
  Buzzer_Off();

  Buzzer_On(659); // Tone 3: E5
  HAL_Delay(100);
  Buzzer_Off();

  Buzzer_On(784); // Tone 4: G5
  HAL_Delay(100);
  Buzzer_Off();

  CharLCD_Clear();
  CharLCD_Init(); // Initialize the LCD
  CharLCD_Set_Cursor(0,0); // Set cursor to row 0, column 0
  CharLCD_Write_String("INITLZING...");
  CharLCD_Set_Cursor(1,0); // Set cursor to row 1, column 0
  CharLCD_Write_String("CLIBRTING...");

```

```
  //initialize the cc2500 chip
  CharLCD_Clear();
  CC2500_Init(); //includes initial calibration

  //calibrate log noise floor
  LogDetector_RecalibrateNoiseFloor();

  CharLCD_Clear();
  CharLCD_Set_Cursor(0,0); // Set cursor to row 0, column 0
  CharLCD_Write_String("5.8GHZ:  2.4GHZ:");
  CharLCD_Set_Cursor(1,0); // Set cursor to row 1, column 0
  CharLCD_Write_String("0.0V");

  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
          //TODO make timer to re calibrate noise floors.
          //Detect with log detector
          // Start the ADC
          HAL_ADC_Start(&hadc1);
          HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);

          // Read ADC value and convert to voltage
          uint16_t logInputValue = HAL_ADC_GetValue(&hadc1);
          float voltage = (logInputValue / 4095.0f) * 3.3f;

          // Check against calibrated threshold
          if (voltage >= LogDetector_DetectionThreshold) {
              sprintf(logMessage, "DT!:%.1f", voltage);
              CharLCD_Set_Cursor(1, 0);
              CharLCD_Write_String(logMessage);
              //BUZZER ON TILL NEXT CYCLE
              Buzzer_On(440); // Tone 1: A4
              HAL_Delay(300);
          }
          else{
              sprintf(logMessage, "FL:%.2f ", voltage);
              CharLCD_Set_Cursor(1, 0);
              CharLCD_Write_String(logMessage);
              //BUZZER OFF TILL NEXT DETECT
              Buzzer_Off();
          }

          //Sweep and detect with cc2500 chip
          CC2500_SweepAndDetect();

          //TODO add 915 sweep

```

```
            //TODO add 433 sweep

    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
```

Snippet from cc2500 driver

```
void CC2500_ApplyConfig(void) {
    CC2500_WriteRegister(0x00, 0x29);
    CC2500_WriteRegister(0x02, 0x06);
    CC2500_WriteRegister(0x03, 0x07);
    CC2500_WriteRegister(0x06, 0x00);
    CC2500_WriteRegister(0x07, 0x04);
    CC2500_WriteRegister(0x08, 0x05);
    CC2500_WriteRegister(0x0A, 0x00);
    CC2500_WriteRegister(0x0B, 0x06);
    CC2500_WriteRegister(0x0C, 0x00);
    CC2500_WriteRegister(0x0D, 0x5D);
    CC2500_WriteRegister(0x0E, 0x93);
    CC2500_WriteRegister(0x0F, 0xB1);
    CC2500_WriteRegister(0x10, 0x2D);
    CC2500_WriteRegister(0x11, 0x3B);
    CC2500_WriteRegister(0x12, 0x73);
    CC2500_WriteRegister(0x15, 0x01);
    CC2500_WriteRegister(0x18, 0x18);
    CC2500_WriteRegister(0x19, 0x1D);
    CC2500_WriteRegister(0x1A, 0x1C);
    CC2500_WriteRegister(0x21, 0x11);
    CC2500_WriteRegister(0x22, 0xE9);
    CC2500_WriteRegister(0x23, 0x2A);
    CC2500_WriteRegister(0x24, 0x00);
    CC2500_WriteRegister(0x25, 0x1F);
    CC2500_WriteRegister(0x3E, 0xC0);
}

//apply configurations and establish first noise floor
void CC2500_Init(void) {
    HAL_Delay(100);
    CC2500_CS_HIGH(); HAL_Delay(1);
    CC2500_CS_LOW();  HAL_Delay(1);
    CC2500_CS_HIGH(); HAL_Delay(1);

    CC2500_Strobe(CC2500_SRES);
    HAL_Delay(1);

    CC2500_ApplyConfig();
    CC2500_Strobe(CC2500_SRX);
```

```
41
42      // Initial noise floor calibration
43      CC2500_RecalibrateNoiseFloor();
44  }
45
46  //re calibrate average noise floor.
47  void CC2500_RecalibrateNoiseFloor(void) {
48          CharLCD_Set_Cursor(0,7); // Set cursor to row 1, column 0
49          CharLCD_Write_String("CALBRATNG");
50
51          int32_t sum = 0;
52      const uint8_t ch_min = 0;
53      const uint8_t ch_max = 100;
54      const int sweep_count = ch_max - ch_min + 1;
55
56      for (uint8_t ch = ch_min; ch <= ch_max; ch++) {
57          CC2500_SetChannel(ch);
58          HAL_Delay(3);
59          int8_t rssi = CC2500_ReadRSSI();
60          sum += rssi;
61      }
62
63      CC2500_NoiseFloor = sum / sweep_count;
64      //starting threshold value:10 increase or deacrease to
           desired sensitivity. TODO Possibly integrate button to
           change this value.
65      CC2500_DetectionThreshold = CC2500_NoiseFloor + 3;
66
67  }
68
69  // Use sweep mode and read rssi to see if packets are being
       recieved on any channels, read strength, alert on noise floor
       threshold
70  void CC2500_SweepAndDetect(void) {
71          for (uint8_t ch = CC2500_SWEEP_MIN; ch <=
               CC2500_SWEEP_MAX; ch++) {
72          CC2500_SetChannel(ch); //set channel
73          HAL_Delay(3);
74          int8_t rssi = CC2500_ReadRSSI(); //read signal strength
               on channel
75
76
77          //OUTPUT ON DETECTION
78          if (rssi > CC2500_DetectionThreshold) { //only get here
               when spike is detected
79                  CharLCD_Clear();
80                  sprintf(chst,"ch:%d",ch);
81                  CharLCD_Set_Cursor(0,7);
82                  CharLCD_Write_String(chst);
83                  CharLCD_Set_Cursor(1,7); // Set cursor to row 1,
                       column 0
```

```
                    CharLCD_Write_String("DT!");
                    HAL_Delay(3);
                    //Trigger alarm
                    Buzzer_On(523); // Tone 2: C5
                    //scan again
                    CC2500_SetChannel(ch);
                    HAL_Delay(200);
                    rssi = CC2500_ReadRSSI();
                    if (rssi > CC2500_DetectionThreshold){ //second round of detection if spike is detected
                          CharLCD_Clear();
                          sprintf(chst,"ch:%d",ch);
                    CharLCD_Set_Cursor(0,8);
                    CharLCD_Write_String(chst);
                    CharLCD_Set_Cursor(1,8); // Set cursor to row 1, column 0
                    CharLCD_Write_String("DT2!");
                    HAL_Delay(200);
                    }


        }
        else {
                    CharLCD_Set_Cursor(0,8); // Set cursor to row 0, column 0
                    CharLCD_Write_String("2.4GHZ: ");
                    CharLCD_Set_Cursor(1,8); // Set cursor to row 1, column 0
                    sprintf(rssiString, "FL:%d", rssi);
                    CharLCD_Write_String(rssiString);

                    //turn off alarm
                    Buzzer_Off();
        }
    }
}
```

Snippet from buzzer driver

```
#include "buzzer.h"

extern TIM_HandleTypeDef htim1; // Change TIM1 to your timer

void Buzzer_Init(void) {
    HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_1);
}

void Buzzer_On(uint16_t frequency) {
    uint32_t timerClock = 80000000; // 80 MHz default APB2
    uint32_t prescaler = 79;        // Must match CubeMX setting
    uint32_t period = (timerClock / (prescaler + 1)) / frequency
```

```
        - 1;

    __HAL_TIM_SET_AUTORELOAD(&htim1, period);
    __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_1, period / 2); // 50% duty
    __HAL_TIM_SET_COUNTER(&htim1, 0);
}

void Buzzer_Off(void) {
    __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_1, 0);
}
```

Snippet from screen driver

```
void CharLCD_Send_Data(uint8_t data) {
 uint8_t upper_nibble = data >> 4; // Extract upper 4 bits
 uint8_t lower_nibble = data & 0x0F; // Extract lower 4 bits
 CharLCD_Write_Nibble(upper_nibble, 1); // Send upper nibble (DC=1 for data)
 CharLCD_Write_Nibble(lower_nibble, 1); // Send lower nibble (DC=1 for data)
}

/**
 * @brief Initialize LCD in 4-bit mode via I2C
 * @param None
 * @retval None
 */
void CharLCD_Init() {
 HAL_Delay(50); // Wait for LCD power-on reset (>40ms)
 CharLCD_Write_Nibble(0x03, 0); // Function set: 8-bit mode (first attempt)
 HAL_Delay(5); // Wait >4.1ms
 CharLCD_Write_Nibble(0x03, 0); // Function set: 8-bit mode (second attempt)
 HAL_Delay(1); // Wait >100us
 CharLCD_Write_Nibble(0x03, 0); // Function set: 8-bit mode (third attempt)
 HAL_Delay(1); // Wait >100us
 CharLCD_Write_Nibble(0x02, 0); // Function set: switch to 4-bit mode
 CharLCD_Send_Cmd(0x28); // Function set: 4-bit, 2 lines, 5x8 font
 CharLCD_Send_Cmd(0x0C); // Display control: display on/cursor off/blink off
 CharLCD_Send_Cmd(0x06); // Entry mode: increment cursor, no shift
 CharLCD_Send_Cmd(0x01); // Clear display
 HAL_Delay(2); // Wait for clear display command
}

```

```
/**
 * @brief Write string to LCD at current cursor position
 * @param str: Pointer to null-terminated string
 * @retval None
 */
void CharLCD_Write_String(char *str) {
 while (*str) { // Loop until null terminator
 CharLCD_Send_Data(*str++); // Send each character and increment
    pointer
 }
}

/**
 * @brief Set cursor position on LCD
 * @param row: Row number (0 or 1 for 2-line display)
 * @param column: Column number (0 to display width - 1)
 * @retval None
 */
void CharLCD_Set_Cursor(uint8_t row, uint8_t column) {
 uint8_t address;
 switch (row) {
 case 0:
 address = 0x00; break; // First line starts at address 0x00
 case 1:
 address = 0x40; break; // Second line starts at address 0x40
 default:
 address = 0x00; // Default to first line for invalid row
 }
 address += column; // Add column offset
 CharLCD_Send_Cmd(0x80 | address); // Set DDRAM address command
    (0x80 + address)
}
```

5 Testing and Results

5.1 Test Methodology

I began testing by detecting the noise floor, mainly coming from WiFi routers in different buildings on the school campus and town. Upon finding that I consistently measured different but reasonable signal strengths throughout the city, I began the second phase of testing.

In phase 2 of the testing, I will start with a control test. For my purposes, the closest thing to a place with no interference was some caves near St Anthony Idaho. I will verify that the signals i am detecting are coming from outside my Faraday cage casing.

In phase 3 of the testing, i will introduce phones, hotspots, and laptops into the cave. I will check for spikes in the readings and note the strengths of each device. I will introduce a DJI mavic 2 and note the difference in signal strength. Using this information, I will tune the detection thresholds in software and begin phase 4.

In phase 4 I will test the accuracy of detection in the mountains and then the city,

using DJI drones for test subject. I will collaborate with the drone society at my local university to test it on a variety of drones.

5.2 Results Summary

Include screenshots, voltages, or response times. Highlight which bands worked better.

- Phase 1: Phase 1 was successful for the 5.8 GHz and 2.4 GHz frequency range. I used wifi routers to generate the requisite frequencies. The cable i had made for my LCD screen was scrambling the I2C serial data due to interference so i used copper tape to shield the outside of the cable, and grounded the shield to my faraday cage (copper lined casing). I had no reliable way of generating 915MHz and 433MHz for testing And my RF modules (similar to cc2500) were not set up yet. I decided then to replace those obscure RF modules with the now proven AD8317. My cc2500 chip began to malfunction due to poor build quality. It established a noise floor, communicated with my STM board and detected spikes 3 times. It then stopped working due to an unknown failure. For the rest of my testing i will substitute the cc2500 for another AD8317 Analog power detector. Link to video

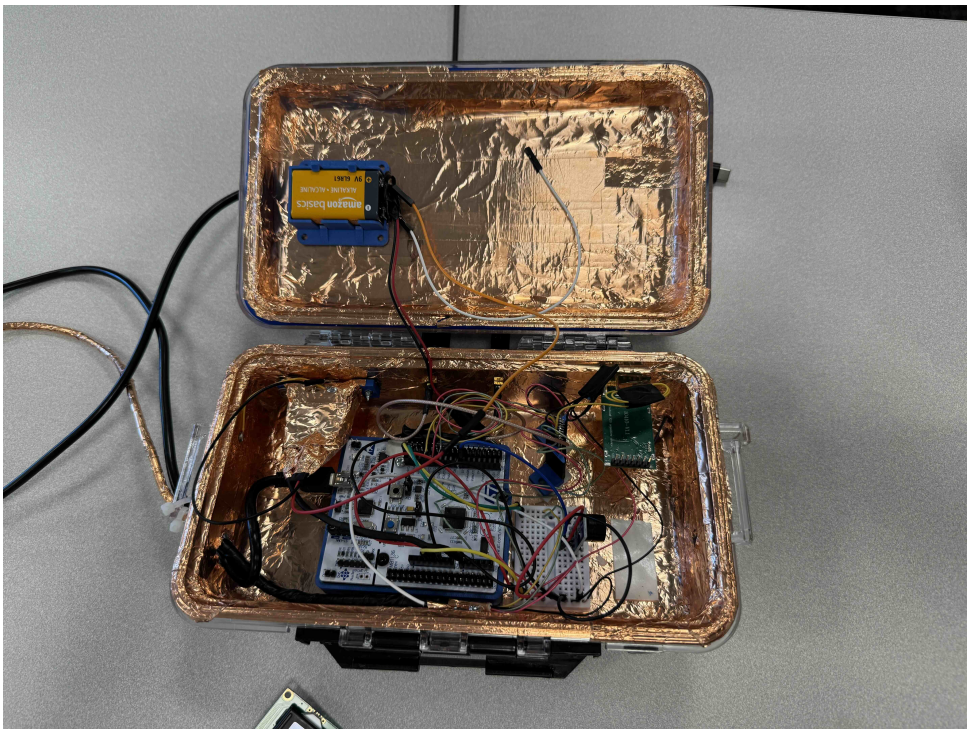

Figure 2: Phase 1 testing

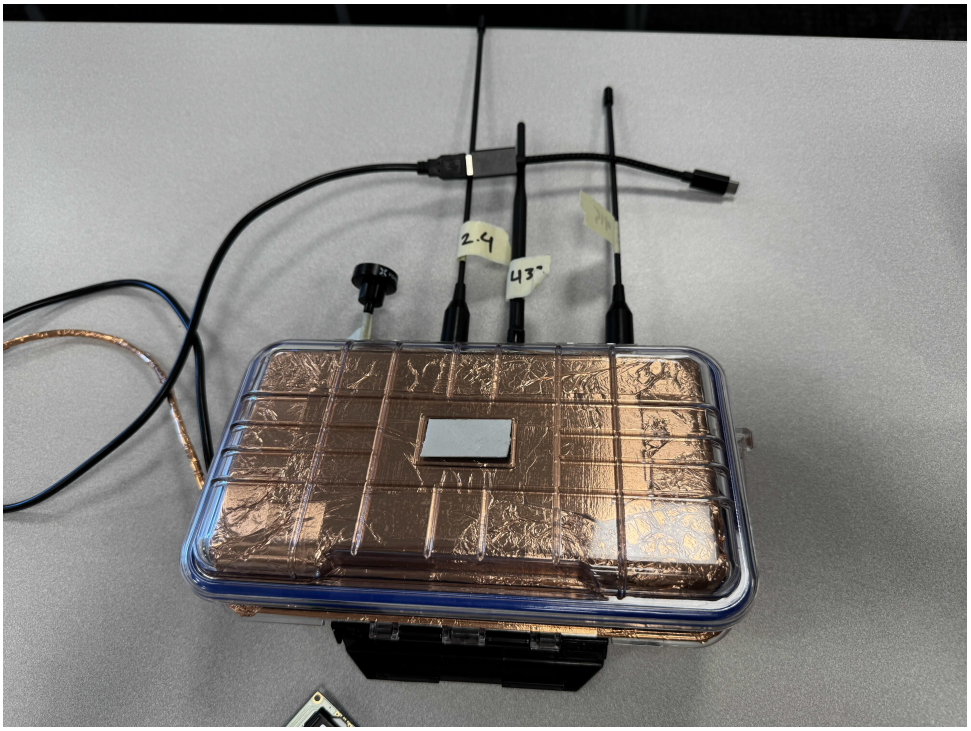

Figure 3: Phase 2 testing

- Phase 2: To be continued...
- Phase 3: To be continued...
- Phase 4: To be continued...

5.3 Challenges

- Ensuring clean ADC signals (AD8317 sensitivity to voltage noise)
- Proper SPI initialization and config for CC2500
- Finding datasheets for cheap chinese clones of reputable RF chips
- Ground loops and shielding effectiveness
- Creating drivers for each I/O device

6 Future Work

- AD8317 for 2.4 GHz frequency
- Code restructuring for updated hardware
- Input buttons for "Dark mode" and "Silent mode"
- Battery power supply
- Phase 2, 3, and 4 testing

7 Career Reflection

I started this project knowing I would be pushing the limits of my knowledge and experience. For weeks I researched and acquired the necessary hardware. For 1.5 months I assembled and tested my system. During that time I changed course twice. In addition to substituting RF modules with logarithmic detectors, I decided that in future versions I would need more processing power and Software Defined Radio to achieve the accuracy I found necessary to reliably isolate drone signals from noise. At the end of this project I have a crude, but functional prototype. I vastly increased my knowledge of:

- Drivers
- I/O
- Serial communication protocols(I2C, SPI)
- ADC(Analog to digital conversion)
- PWM (pulse width modulation)
- Debugging techniques
- RF communication
- Drone technology

8 Appendix

- CC2500 register configuration table

Address (Hex)	Register Name	Value (Hex)	Purpose
`0x00`	IOCFG2	`0x29`	GDO2 output config (optional)
`0x02`	IOCFG0	`0x06`	GDO0 as sync word received
`0x0B`	FSCTRL1	`0x06`	Frequency synthesizer control
`0x0C`	FSCTRL0	`0x00`	
`0x0D`	FREQ2	`0x5D`	2.433 GHz
`0x0E`	FREQ1	`0x93`	
`0x0F`	FREQ0	`0xB1`	
`0x10`	MDMCFG4	`0x2D`	Modem config
`0x11`	MDMCFG3	`0x3B`	
`0x12`	MDMCFG2	`0x73`	
`0x15`	DEVIATN	`0x01`	
`0x18`	MCSM0	`0x18`	Auto-calibrate
`0x19`	FOCCFG	`0x1D`	Frequency Offset Correction
`0x1A`	BSCFG	`0x1C`	Bit synchronization
`0x21`	FREND0	`0x11`	Front end RX configuration
`0x22`	FSCAL3	`0xE9`	Frequency synthesizer cal
`0x23`	FSCAL2	`0x2A`	
`0x24`	FSCAL1	`0x00`	
`0x25`	FSCAL0	`0x1F`	
`0x07`	PKTCTRL1	`0x04`	No address check
`0x08`	PKTCTRL0	`0x05`	Infinite packet length, CRC
`0x3E`	PATABLE	`0xC0`	Max output power

Figure 4: Config table

- Link to GitHub Code