Space Systems Lab Firmware and Mission Requirements Document

Hersch Nathan

November 2024

0.1 General Requirements

This software is to support the KRUPS missions. The current mission is KRUPS Aboard Norwegian GhostSat (KANGS). The hardware paradim is based around an ESP32-S3.

This work is porting Matt Ruffner's firmware for Adafruit Feather M0/M4 to the ESP32-S3. This including switching from the Arduino IDE to ESP-IDF.

The goal of this project is to make the firmware more modular and configurable for each mission and hardware.

The first chapter of this document outlines the overall architecture design and background information for this project. The second chapter provides information for writing firmware for this project. The third chapter details each library, driver, wrapper, and other software components that we wrote for this project. It provides design decisions and the reasoning behind them as well as how to interface with them. The fourth chapter and beyond detail the software requirements for each hardware component for each mission.

Contents

Chapter 1

Architechure Design

1.1 Background Information

1.1.1 freeRTOS

freeRTOS is a real time operating system. A real time operating system works by having tasks that run at different priorities. The tasks are scheduled by the operating system and those tasks occur through a determanistic prosess. The tasks can communicate with each other using message queues. The tasks can also synchronize with each other using semaphores.

1.1.2 ESP-IDF

ESP-IDF is the Espressif IoT Development Framework. It is the official development framework for the ESP32 and ESP32-S3. It is a set of libraries and tools that are used to develop software for the ESP32 and ESP32-S3.

1.2 Overall Architecture Design

The firmware's framework is designed to be modular and configerable thmission and each hardware. The backbone of the firmware is using freeRTOS tasks to handle the different modules. Each taks is its own module that encapsulates the relevant hardware. The tasks communicate with each other using a publisher/subscriber model.

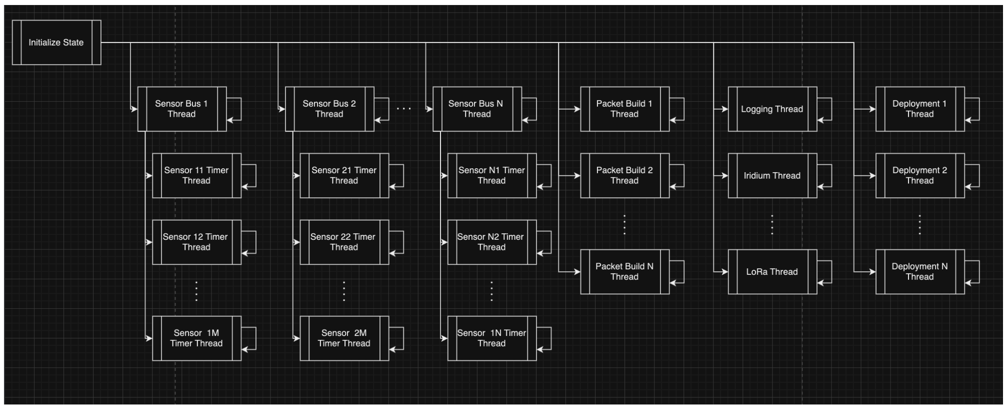


Figure 1.1: Firmware Architecture

In the figure 1.1, the main tasks there are four types of tasks.

1.2.1 Task

Sensor Bus Tasks

Each sensor bus task manages a single hardware bus (I2C, SPI, UART, etc). At startup, it configures the sensors with the necessary parameters, then it calls a looping task. That task takes the semaphore for the used serial communication interface. It sends the command to the sensor to get the data, reads the response, and then releases the semaphore. The task then sends the data to the packet build tasks queues.

The sensor manager works by creating a configuration for each bus, including the sensors connected to it, their intervals, and their handlers. It uses FreeRTOS timers to periodically trigger sensor events, which are added to a queue. The bus task processes these events, taking a mutex to ensure exclusive access to the bus, and calls the appropriate sensor handler from a lookup table. The handlers are responsible for reading data from the sensors and performing any necessary processing.

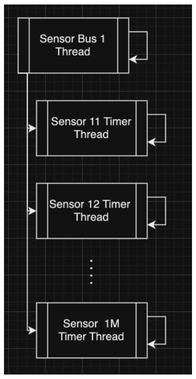


Figure 1.2: Sensor Task Architecture

Packet Build Tasks

Wireless Communication Tasks

Deployment Tasks

1.2.2 Middleware

Chapter 2

Writing Firmware

2.1 Adding Components

The current file stucture of SLLFirmware is as follows. It has 3 compoents the main firmware, the sensor drivers, and the manager drivers. If another compoent is to be added make sure to append it to the SSLFirmware/CMakeLists.txt file. Then create a folder for the component in the SSLFirmware folder. In that folder create a CMakeLists.txt file and the source files for the component.

```
SSLFirmware/
    - CMakeLists.txt
    - sdkconfig
    - main/
        - CMakeLists.txt
        - SSLFirmware.c
        - config.h
    - Manager/
        - i2c_manager/
            - CMakeLists.txt
            - i2c_manager.c
            - i2c_manager.h
    - Sensors/
        - CMakeLists.txt
        - BME280_Sensor/
            - CMakeLists.txt
            - BME280_Sensor.c
            - BME280_Sensor.h
            - BME280_API/
        - Fake_Sensor/
            - CMakeLists.txt
            - Fake_Sensor.c
            - Fake_Sensor.h
```

The CMakeLists.txt file for the project should look like the following. The EXTRA_COMPONENT_DIRS variable is used to add the components to the project. The include statement is used to include the project.cmake file from the ESP-IDF tools. The project name is set to SSLFirmware.

```
cmake_minimum_required(VERSION 3.16)
```

```
list(APPEND EXTRA_COMPONENT_DIRS ${CMAKE_SOURCE_DIR}/Manager)
list(APPEND EXTRA_COMPONENT_DIRS ${CMAKE_SOURCE_DIR}/Sensors)

include($ENV{IDF_PATH}/tools/cmake/project.cmake)
```

```
project(SSLFirmware)
```

The CMakeList.txt file for each componet should look like the following. It specifies the source files for the component and the include directories for the component. The idf_component_register function is used to register the component with the project. The SRCS variable is used to specify the source files for the component. The INCLUDE_DIRS variable is used to specify the include directories for the component. The PRIV_REQUIRES variable is used to specify the components that the component requires. The REQUIRES variable is used to specify the components that the component depends on.

```
idf_component_register(SRCS "i2c_manager/i2c_manager"
                       INCLUDE_DIRS "i2c_manager"
                       PRIV_REQUIRES driver Sensors
                       REQUIRES main)
```

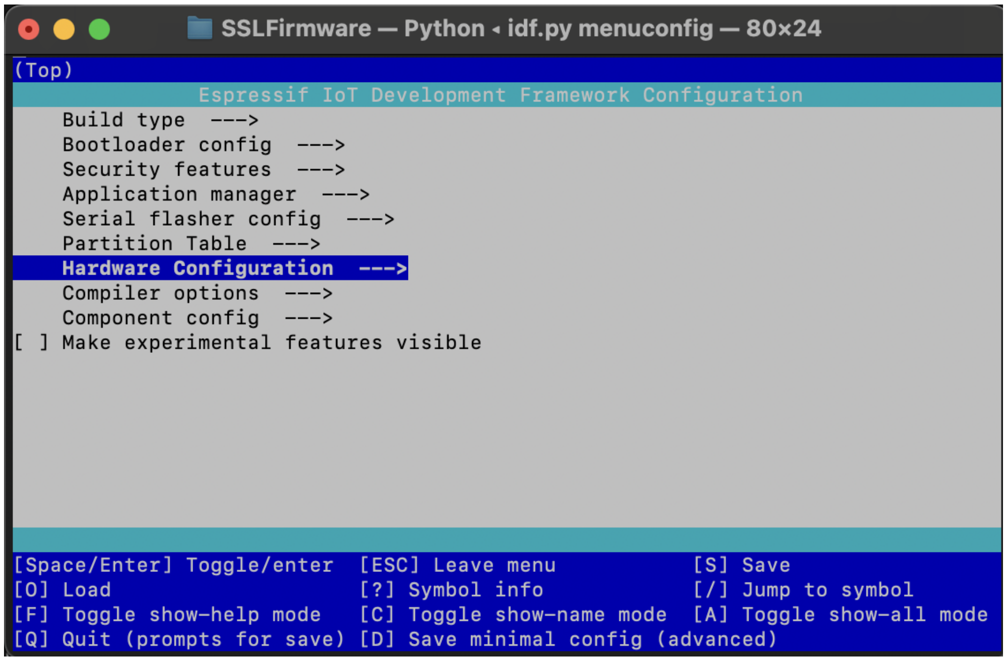
Chapter 3

Software Components

3.1 Configuration

To select the hardware that the firmware will use, use the menuconfig tool. To access the menuconfig tool, run the following command in the terminal. The options is under Hardware Configuration see figure 3.1. Then select the hardware that you want to use and save the configeration see figure 3.2. Later additions will have a selection for the mission then for each hardware component if this list gets too long.

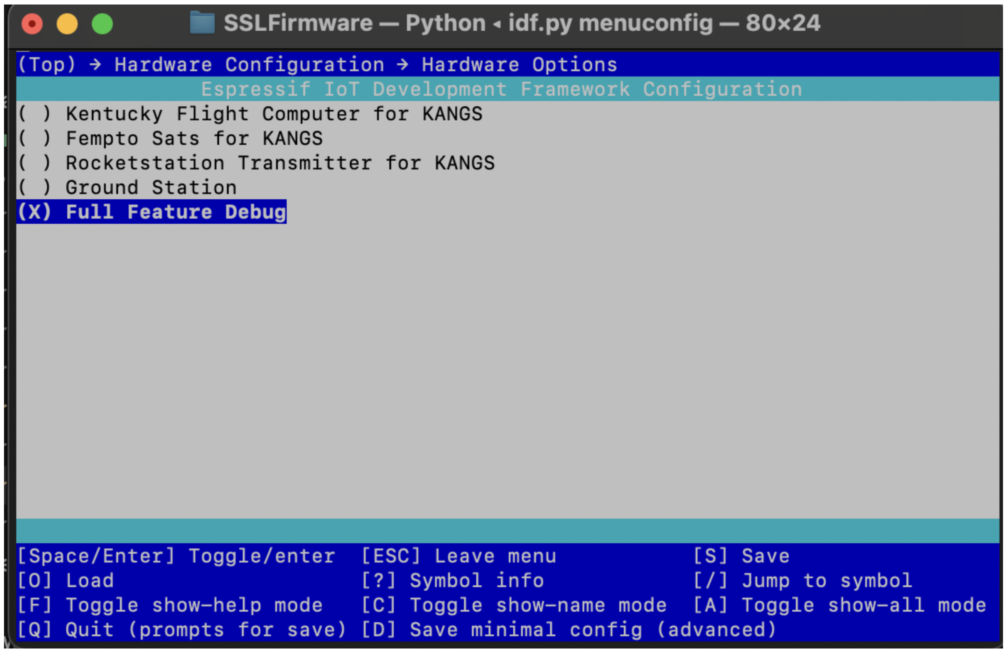
```
idf.py menuconfig
```



```
(Top)
                   Espressif IoT Development Framework Configuration
    Build type  --->
    Bootloader config  --->
    Security features  --->
    Application manager  --->
    Serial flasher config  --->
    Partition Table  --->
    Hardware Configuration  --->
    Compiler options  --->
    Component config  --->
[ ] Make experimental features visible

[Space/Enter] Toggle/enter  [ESC] Leave menu           [S] Save
[O] Load                    [?] Symbol info            [/] Jump to symbol
[F] Toggle show-help mode   [C] Toggle show-name mode  [A] Toggle show-all mode
[Q] Quit (prompts for save) [D] Save minimal config (advanced)
```

Figure 3.1: IDF Menuconfig



```
SSLFirmware — Python ◂ idf.py menuconfig — 80×24
(Top) → Hardware Configuration → Hardware Options
                  Espressif IoT Development Framework Configuration
( ) Kentucky Flight Computer for KANGS
( ) Fempto Sats for KANGS
( ) Rocketstation Transmitter for KANGS
( ) Ground Station
(X) Full Feature Debug

[Space/Enter] Toggle/enter  [ESC] Leave menu           [S] Save
[O] Load                    [?] Symbol info            [/] Jump to symbol
[F] Toggle show-help mode   [C] Toggle show-name mode  [A] Toggle show-all mode
[Q] Quit (prompts for save) [D] Save minimal config (advanced)
```

Figure 3.2: Hardware options

the specific mission parameters (pinout, sensor configuration, etc) are set in the config.h file.

3.2 Managers

3.2.1 I2C Manager

The I2C manager is responsible for managing the I2C bus and interfacing with sensors connected to it. It provides a high-level interface for configuring sensors, setting up timers, and handling sensor events. The I2C manager is designed to be modular and extensible, allowing for easy integration of new sensors and bus configurations.

The I2C manager works as follows:

- **Initialization**: The I2C manager initializes the I2C bus with the specified configurations. It sets up the I2C port, creates a queue for sensor events, and initializes a mutex for bus access control.

- **Sensor Configuration**: Before starting the timers, the I2C manager configures each sensor connected to the bus. This involves calling the sensor-specific configuration functions.

- **Timer Creation**: The I2C manager creates FreeRTOS timers for each sensor. These timers are set to trigger at specified intervals, defined in the sensor configuration. When a timer expires, it adds a sensor event to the event queue.

- **Event Handling**: The I2C manager runs a task that continuously waits for sensor events from the queue. When an event is received, it takes the mutex to gain exclusive access to the I2C bus, calls the appropriate sensor handler from the lookup table, and then releases the mutex.

Note that the timeout ticks are hard-defined to 1000 in the I2C manager header file. This value can be adjusted based on the specific requirements of the application.

The ‘i2cBusTask‘ is created as follows:

- **I2CBusConfig**: This structure is used to configure the I2C bus. It includes the I2C port, sensor events, sensor intervals, sensor configuration functions, and the number of sensors. For example:

  ```
  I2CBusConfig bus1Config = {
      .i2cPort = I2C_NUM_0,
      .sensorEvents = {SENSOR1_EVENT, SENSOR2_EVENT},
      .sensorIntervals = {sensorIntervals[SENSOR1_EVENT], sensorIntervals[SENSOR2_EVENT]},
      .sensorConfigs = {SENSOR1_config, SENSOR2_config},
      .sensorCount = 2
  };
  ```

- **Event Queue**: An event queue is created for each I2C bus to hold sensor events. This queue is used to pass events from the timers to the I2C bus task. For example:

  ```
  bus1Config.eventQueue = xQueueCreate(10, sizeof(SensorEventType));
  ```

- **Mutex**: A mutex is created for each I2C bus to ensure exclusive access to the bus. This prevents conflicts when multiple tasks try to access the bus simultaneously. For example:

  ```
  bus1Config.mutex = xSemaphoreCreateMutex();
  ```

- **Task Creation**: The 'i2cBusTask' is created using the 'xTaskCreate' function. This function takes the task function, task name, stack size, task parameters, task priority, and task handle as arguments. For example:

  ```
  xTaskCreate(i2cBusTask, "I2CBusTask1", 2048, &bus1Config, 5, NULL);
  ```

Chapter 4

KRUPS Aboard Norwegian GhostSat (KANG)

4.1 Kentucky Flight Computer (KFC) Requirements

4.2 FemptoSats Requirements

The FemptoSats submission is to test the viability of using Wifi/LoRa for intercapsuole communication.

4.2.1 Sensors Requirements

The FemptoSats will have the following sensors:

- BME280 Temperature/pressure/humidity sensor
- BNO086 9 - Axis IMU

BME280 Temperature/pressure/humidity sensor

The BME280 will be run with the following configuration settings:

BNO086 9 - Axis IMU

The BNO086 will be run with the following configuration settings:

4.2.2 Wireless Communcation Requirements

the FemptoSat will use the following Wireless Communcation modules:

- Integrated Wifi
- LoRa

The Wifi will fail over to the LoRa when the Wifi gets out of range

4.3 Rocketstation Transmitter (RST) Requirements

uart to raspi to shut off need

4.4 Rocketstation Requirements

uart to raspi to shut off need

4.5 Groundstation Requirements