Project Documentation

BlackBoxESP

Filip Radojević

September 26, 2025

Contents

Chapter 1

Introduction

The project was initially developed on LPC1768 boards used in the laboratory, where several system modules were implemented, such as `GW_GND`, `GW_SKY`, `INS`, `INS_COST`, `POWER_MANAGEMENT`, `ACTUATOR_MASTER`, and the current `BlackBox`.

However, due to higher performance requirements, the project is now being migrated to the ESP32-S3(R8) ETH Kit, which provides significantly larger memory: both SRAM for network buffers (allowing larger `pcache` and `rcache` sizes in LFS when using larger flash `page_size`), and PSRAM which is intended to serve as a ring buffer for data logging.

Chapter 2

System Architecture

The current implementation uses two tasks: **work** and **log**, both having the same priority.

In the work task, incoming data from Ethernet is processed through the `udp_process` function. These packets are then pushed into queues linked to the log task via a callback function. The log task is blocked on a semaphore, which is released by the work task whenever new data is available.

This ensures that the log task immediately handles data logging without wasting CPU cycles. The log task performs `lfs_write` operations to store data into flash. With this approach, the system has achieved a maximum reliable logging speed of **1.2 Mbit/s**.

However, due to the slow synchronization mechanism of `lfs_sync`, which is necessary to ensure power-loss safety, a larger ring buffer is required to avoid data corruption or loss. Therefore, PSRAM is used as a large intermediate buffer, ensuring that writing speed can exceed reading speed without compromising system integrity.

Chapter 3

Project Structure

The project is organized into the following modules:

- **app/** – main application (e.g., `task_work.c`, `task_log.c`, `main.c`)
- **drivers/** – hardware drivers (Ethernet, flash, SD card, FTP, etc.)
- **middleware/** – protocols and layers (UDP, MAVLink, ULog, LFS, etc.)

3.1 Development Environment

The project is developed in **Visual Studio Code**, where the build system is integrated with the ESP-IDF (Espressif IoT Development Framework) toolchain and `ninja` build system.

- `F7` – Compile (Build process)
- `F3` – Clean Rebuild
- `F5` – Flash the project to the microcontroller

Chapter 4

CMake Build System

The project uses the ESP-IDF build system, which is based on CMake. The build process consists of two levels:

- **Global level:** Defined in the root `CMakeLists.txt`. It specifies the project name, registers extra component directories, adds compiler options, and includes ESP-IDF's `project.cmake`. It also generates build metadata (Git hash, version tags, timestamps) that are injected into `main.h`.
- **Component level:** Each module (e.g., `src/drivers/flash`, `src/middleware/udp`) has its own `CMakeLists.txt` with `idf_component_register`. This defines which source files are compiled, where headers are located, and which other components are required.

When `idf.py build` is executed, ESP-IDF scans all extra components, resolves dependencies, and compiles each module. The result is a firmware binary that can be flashed to the ESP32-S3 device.

Chapter 5

System Diagram

The following diagram illustrates the workflow of the system, showing how the **Work Task**, **Log Task**, PSRAM ring buffer, and Flash interact with each other through semaphores and data pipelines.

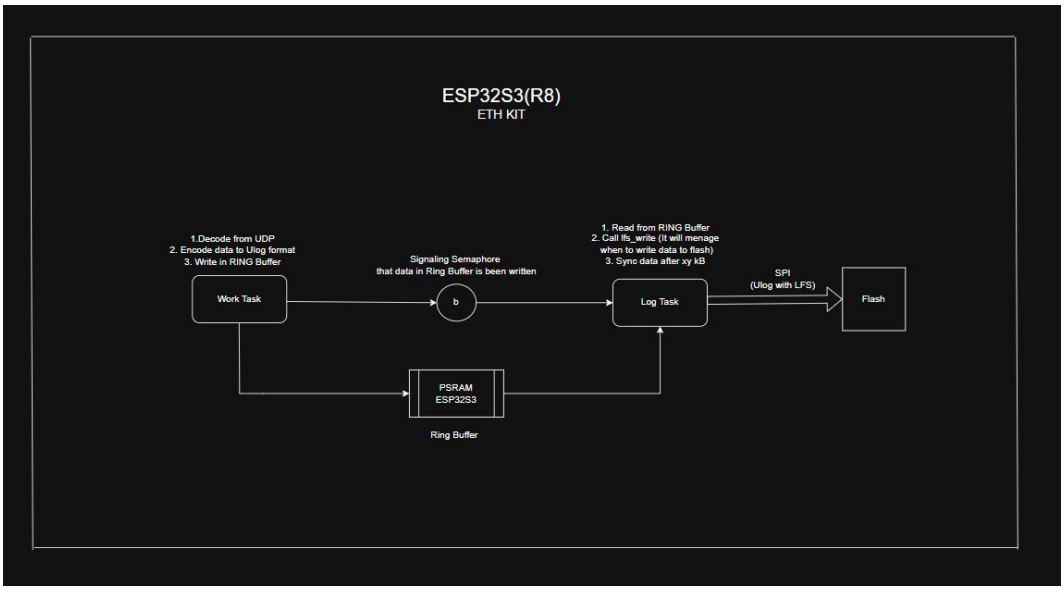

Figure 5.1: System architecture of ESP32-S3(R8) ETH Kit implementation.