Samaung

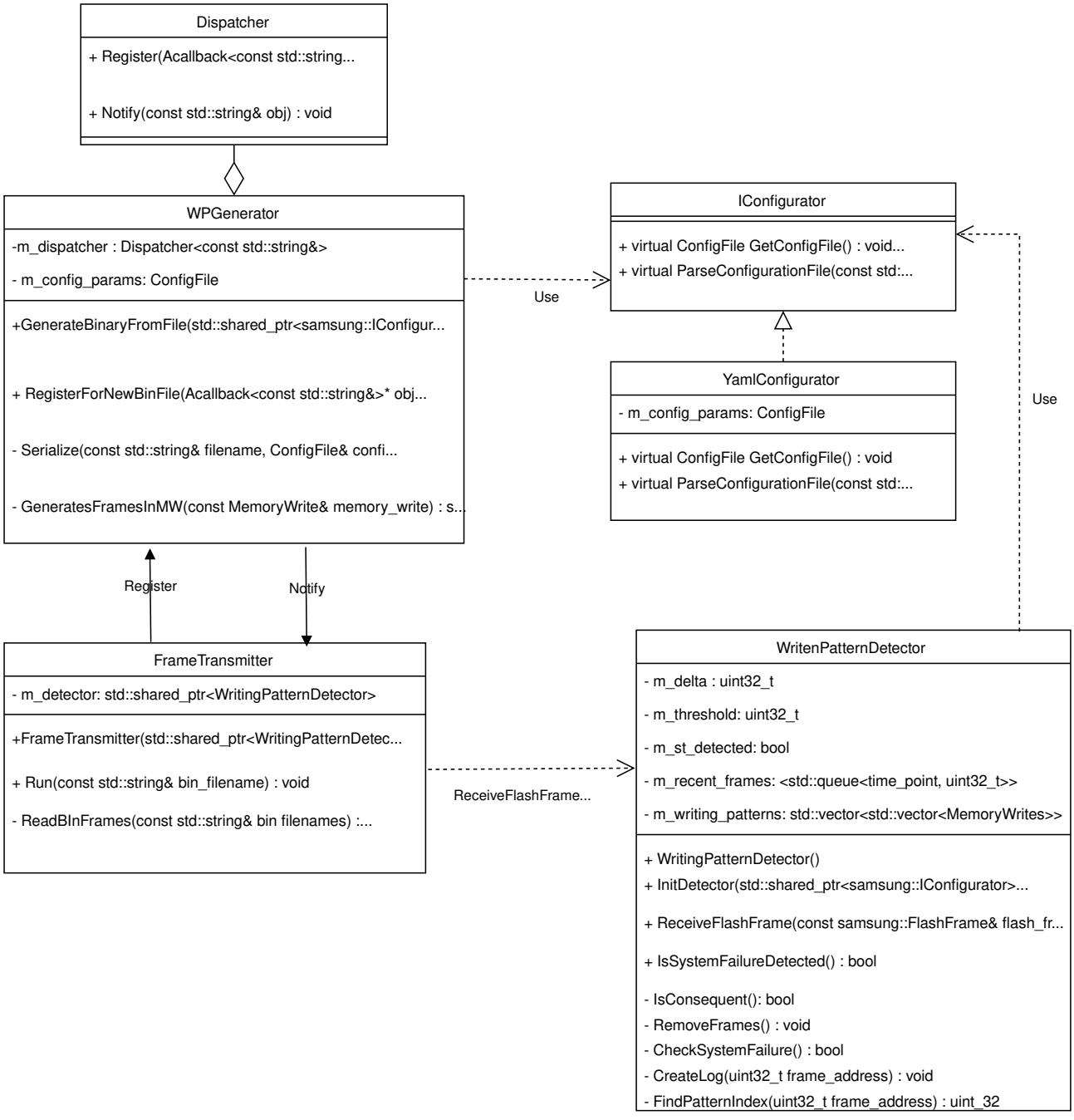
Home assigment

Shani Nativ

System Overview

The system simulates data transmission and reception to FLASH memory over a communication channel. The main focus of this project is early detection of system failures, which occur when multiple memory writes happen within a short, defined time frame. Upon detecting such a condition, the system writes a log file with the details of the relevant writing pattern, and halts further writes to FLASH memory to prevent the system from getting stuck.

UML Diagram



Task 1 – Writing Pattern Generator Module

1. Overview

This module is responsible for converting a user-provided configuration file (in YAML format), which contains one or more writing patterns, into a binary file. This binary file is then passed through the remaining system modules and ultimately written to FLASH memory.

2. Methods

The logic of Generate binary from configuration file:

A YamlConfigurator reads the YAML configuration file and converts its contents into a structured C++ object called ConfigFile. This object contains all relevant configuration data:

```
//definition.hpp
struct MemoryWrite
{
    uint32_t start_time;
    uint32_t duration;
    uint32_t start_address;
    uint32_t frames;
};

//definition.hpp
struct ConfigFile
{
    uint32_t threshold;
    uint32_t delta;
    std::vector<std::vector<MemoryWrite>> writing_patterns;
};
```

Next, the WPGenerator module is used. Its GenerateBinaryFromFile() method takes an IConfigurator instance as input to obtain the concrete ConfigFile. It then creates a new binary file and serializes all the configuration data into it.

The binary file is structured to begin with a metadata section, which ensures that the data can be correctly parsed and interpreted later during transmission or detection phases.

```
//definition.hpp
struct BinMetaData
{
    uint32_t threshold;
    uint32_t delta;
    uint32_t total_frames;
};
```

Immediately following the metadata, the frames themselves are written. In this project, each frame is represented by 4KB of zeroed data.

Once the binary file is generated, a notification is sent to a registered observer — the Frame Transmitter — which is responsible for reading and transmitting the frames.

3. Tests

Functional Testing

- Verify that the binary file is correctly generated by performing a reverse deserialization process and comparing the results with the original YAML input.
- Validate the system's behavior using different YAML structures, including multiple writing patterns and varying complexity. Also tested diffrent edge-case tests, such as: extremely large configurations with thousands of writing patterns, empty configurations, and deliberately malformed or invalid YAML files to evaluate the system's ability to detect and report parsing errors without crashing.

Performance Testing

- Measure runtime and memory consumption across various scenarios, including large configurations with high volumes of writing patterns , to ensure stability and efficiency.
- Using Profiling Tools to measure memory consumption and identify potential performance bottlenecks or memory leaks.

Multi-CPU Consideration

- In a multi-core CPU environment, the program can benefit from parallelism by using multiple threads to handle CPU-bound tasks. For example, the YAML parsing process can be divided among several threads to accelerate configuration file processing and reduce overall execution time.
- Thread Safety: Proper synchronization mechanisms such as mutexes or atomic operations would be essential to avoid race conditions and ensure data consistency.

Task 2 – 'Frame transmitter' and 'Writing Pattern Detector' modules

1. Overview

The modules are responsible for processing the binary file generated by the previous stage and translating it into data writes to FLASH memory. If a system failure is detected during execution, the following actions are performed:

- A log file is generated, containing detailed information about the failure, as specified in the project documentation.
- Further writes to FLASH are blocked in order to prevent the system from freezing or entering an unstable state.

This mechanism ensures both fault detection and system protection during abnormal write behavior

2. Methods

Frame Transmitter

- The Run() method is triggered once the binary file is created — via a callback invoked by the dispatcher.
- This function reads the binary file and parses its content into a queue of FlashFrame structures:

```
//definition.hpp
struct FlashFrame
{
    uint32_t address;
    char data[g_data_size]; // typically 4KB of data (zeros in this project)
};
```

- Each frame is stored in a queue along with its absolute transmission time, which is computed based on the frame's transmission_time metadata.
- After completing the file parsing, the transmitter starts sending frames one by one to the detector module at the appropriate time, using nanosleep to wait between transmissions.

Writing Pattern Detector

- Upon initialization, the detector uses an IConfigurator instance to retrieve configuration parameters, specifically the THRESHOLD and DELTA values. It also prepares internal data structures needed for potential system failure detection and logging.
- The core method is ReceiveFlashFrame, which is invoked whenever a new frame arrives from the transmitter. Its logic is as follows:
    1. If a system failure was already detected, the method exits immediately (no further action).
    2. Otherwise, it enqueues the current arrival time and the frame address into an internal sliding window queue.
    3. It removes entries from the head of the queue that are older than DELTA seconds relative to the current time, ensuring the window only contains recent frames.
    4. It then checks for system failure conditions:
        - If the queue contains at least THRESHOLD frames, and
        - The last THRESHOLD addresses are consecutive in memory (increasing by 1).
    5. If failure conditions are met:
        - A log file is created and populated with failure details.

- During log creation, the detector identifies the specific writing pattern to which the faulty frame belongs, and includes only that writing pattern in the output log for clarity and traceability.
- A flag is set to indicate that a system failure has been detected (to avoid further writes).

6. If no failure is detected, writing to FLASH is considered safe. (the actual FLASH write method was not implemented in this project)

- An additional method, ResetFlashFrames(), is used to reset the internal state of the detector between test runs. This includes clearing the sliding window and resetting the failure detection flag.

3. Tests

Functional Testing

- Verified that the system correctly detects failure conditions by simulating scenarios that exceed the `THRESHOLD` within the specified `DELTA` window and generates the appropriate log file, including all relevant details that caused the failure.
- Verified that the system transmits each FLASH frame at its designated transmission time, ensuring correct timing behavior as defined by the writing pattern configuration.
- Confirmed that the system performs FLASH writes as expected under normal conditions, and accurately halts writing at the right moment, just before a failure would occur.

Performance Testing

- Measured runtime and memory consumption under various workloads, including high-frequency FlashFrame transmission scenarios.
- Using Profiling Tools to measure memory consumption and identify potential performance bottlenecks or memory leaks.

Multi-CPU Consideration

In a multi-core CPU environment, the system can leverage parallelism to improve responsiveness and execution speed by distributing CPU-bound tasks across multiple threads:

- Frame Scheduling: A dedicated thread can handle scheduling of frame transmissions independently, using a scheduler. This allows scheduling to begin before the entire binary file is written, improving overall throughput and minimizing idle time.
- Asynchronous Log Writing: Logging operations can be offloaded to a separate thread, allowing the detector to continue processing incoming frames without being blocked by disk I/O. This improves responsiveness, especially under high load or frequent failure conditions.

- Thread Safety: To maintain correctness in a multi-threaded environment, shared resources such as queues, logs, and state flags must be protected using proper synchronization mechanisms such as mutex or atomic variables. This ensures consistency, avoids race conditions, and preserves data integrity during concurrent access.