Samaung

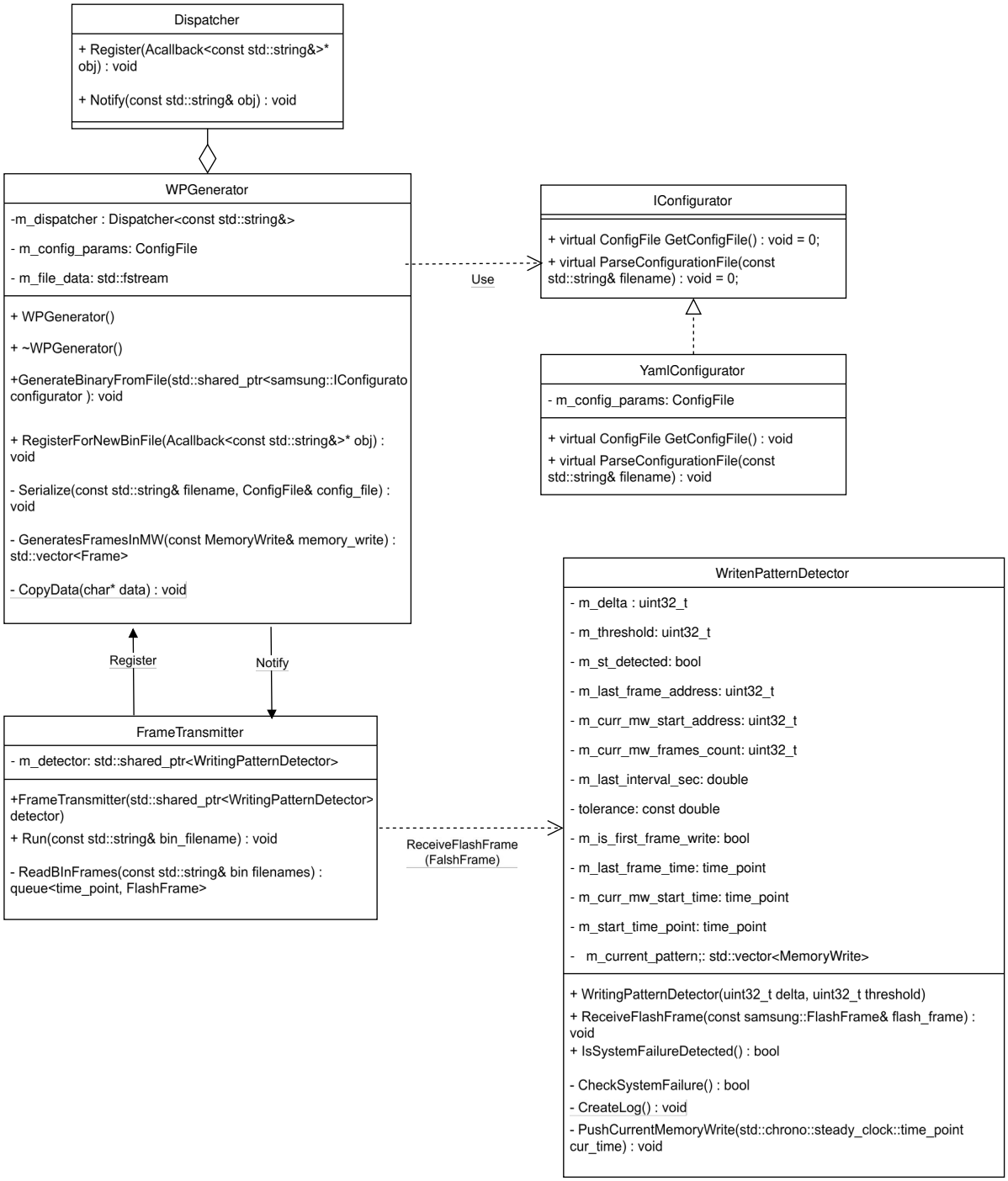
Home assigment

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 and contiguous memory writes happen within a defined time frame. Upon detecting such a condition, the system rites 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 receives an IConfigurator instance to retrieve the concrete ConfigFile object. It then creates a new binary file, prepares all the required frames based on the configuration, and serializes both the metadata and the frame data into the file.
The binary file begins with a dedicated metadata section, which ensures that the content can be correctly parsed and interpreted during subsequent transmission or failure 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 random 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 take advantage of parallelism by using multiple threads to efficiently handle CPU-bound tasks. For instance, the process of generating the FRAME content—each consisting of 4KB blocks—can be distributed across several threads to accelerate memory preparation.
- 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.
- 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 receives configuration parameters: THRESHOLD (the minimum number of memory writes required to consider a system failure) and DELTA (the maximum time interval allowed for those writes in seconds).
- The detector maintains an internal vector of MemoryWrite structures representing the current writing pattern being observed.
- The core method is ReceiveFlashFrame, which is called every time a new flash frame arrives from the transmitter. Its behavior is as follows:

    1. System Failure Already Detected:the method returns immediately.

    2. First Frame Initialization:the detector initializes relevant internal variables such as last address, last timestamp, and memory write counters.

    3. Time and Address Analysis:For subsequent frames, the method calculates the time interval since the last frame and checks whether the current frame address is contiguous with the last.

    4. New Writing Pattern Detection:If the addresses are not contiguous, this signals the start of a new writing pattern. The internal memory write vector is cleared, all counters reset, and detection logic starts over.

    5. New Memory Write Within Pattern:If the addresses are contiguous, but the time interval differs from the last one (beyond a small tolerance), it indicates a new memory write within the same pattern. The previous memory write is pushed into the vector, and a new one is started with the current frame.

6. Continuation of Current Memory Write: If both the addresses are contiguous and the time interval is within tolerance, the current memory write continues. The frame count (N) for the current memory write is incremented.

7. System Failure Check:the method checks if a system failure condition is met:

- The total number of contiguous frames exceeds the THRESHOLD.
- The time interval from the first memory write to now is less than or equal to DELTA.

- System Failure Logging: If the failure condition is met:
  - A log file is created, detailing the system failure: start address, threshold, delta, and all memory writes.
  - A flag is set to prevent further processing of incoming frames.
- No Failure:
  If the condition is not met, the flash write is considered safe, and the detector continues processing future frames.

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 design enables frame scheduling to start as soon as enough of the binary file has been parsed — without waiting for the entire file to load. As a result, the system achieves higher throughput and lower startup latency.
- Asynchronous Log Writing: Logging operations can be offloaded to a separate thread, This prevents the detector's critical path from being blocked by disk I/O operations. 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.