What is Mlibs?

MLibs is a collection of modular C-based libraries designed for embedded systems.

It provides implementations for various data structures and algorithms that are optimized for ARM Cortex-M microcontrollers, particularly the STM32F4 series.

Project Objectives:

- To enable high-performance, efficient data structures and algorithms for embedded environments.
- To provide a robust test automation framework for validating embedded code.
- To integrate standard C++ libraries like std::vector with custom allocators tailored for embedded constraints.

Key Features:

- Support for fixed-size arrays, linked lists, and open-addressed hash tables.
- Use of CMSIS and HAL libraries for STM32-specific operations.
- Integration of custom memory allocators for efficient memory management.

MLibs Architecture and Testing Framework

Architecture Overview:

The project is modular, allowing individual data structures and algorithms to be used independently.

Utilizes a generic function pointer-based approach to handle various types and operations.

Test Automation Framework:

Automated testing setup using a bfr.py script to build, flash, and run tests.

Supports semi-hosted testing with st-link and openocd for real-time feedback from the microcontroller.

Toolchain and Debug Support:

The toolchain includes gcc-arm-none-eabi, openocd, and multiarch-gdb for ARM debugging.

Integration with doxygen for generating detailed documentation and visual summaries.

MLibs Build, Flash, Using bfr.py

Automating the Build and Flash Process:

Use the bfr.py script to automate building, flashing, and running the entire test suite.

Command:

> source ./env.sh

```
> python3./bfr tests clean build flashrun
Breakdown:
     Clean: cleans all the binary and object files
     build: Compiles all test directories.
     flashrun: Flashes the compiled .bin file and runs it on the microcontroller.
Process Summary:
     creates a ‘bin” file from the elf
     Erases the microcontroller.
     Flashes the .bin file using st-flash.
     Starts openocd and runs the tests using gdb in an automated fashion.
```

MLibs: Debugging a Specific Test with GDB

Go to the desired test directory (e.g., tests/random) and run make:

openocd -f board/stm32f4discovery.cfg in a separate shell

gdb-multiarch -tui test.elf

target remote :3333

load

break main

next, step, etc.

MLibs: How the ELF and BIN Files Work

1. Generating the .bin File:

   objcopy converts the compiled ELF file into a binary format (BIN).

   The .bin file is what gets flashed onto the microcontroller.

   This process extracts only the raw data and instructions from the ELF file, without debug symbols or additional metadata.

2. Flashing the Microcontroller:

   The .bin file is flashed to the STM32 microcontroller using st-flash or openocd.

   This step loads the binary code directly into the microcontroller’s memory.

MLibs: How the ELF and BIN Files Work (cont.)

3. Using OpenOCD as a Proxy:

   openocd acts as a bridge between the microcontroller and gdb.

   It communicates with the microcontroller’s onboard debugger registers.

   OpenOCD handles the low-level interaction and provides a GDB server endpoint (usually localhost:3333).

4. Debugging with GDB:

   GDB connects to the running openocd session.

   GDB uses the local ELF file (which contains debug symbols and additional metadata) to provide a comprehensive debugging experience.

   This allows breakpoints, variable inspection, and source-level debugging.

MLibs: Library Categories

Data Structures:

- Dynamic Arrays, Linked Lists, Open-Addressed Hash Tables
- Priority Queues, Graphs
- ARM Assembly: Sort, Copy, Compare, Transform, Spinlock

Algorithms:

- Sorting and Searching
- Transformation and Modification
- Graph Algorithms

Functors and Utilities:

- Comparison, Swapping, and Printing Functions, Generic Operations for Flexibility

MLibs: Key Features

Memory Awareness:

- Libraries are optimized for limited-memory environments.
- Integration with custom allocators for efficient memory management.

Generics and Flexibility:

- Generic function pointers allow for type-agnostic algorithms.

Provided functors ensure safety and consistency.

- Patterns and Modularity:
- Modular functions and utilities that can be combined to build complex structures.
- Reusable patterns to ensure code maintainability.