UNIVERSITY OF DHAKA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CSE 3103 : Microprocessor and Assembly Lab

Lab Report - 6
Bare-Metal Programming

Advisor: Dr. Upama Kabir & Md. Mustafizur Rahman

Submitted by: Shahanaz Sharmin (Sk-07)
Anika Tabassum (Rk-61)

Contents

1 Introduction

The primary goal of this assignment is to create a fundamental driver for bare-metal programming on the STM32F446RE microcontroller. The focus is on direct device programming without utilizing higher-level abstractions like HAL (Hardware Abstraction Layer) or CMSIS (Cortex Microcontroller Software Interface Standard). To achieve this, a strong understanding of the linker, loader, and makefiles is necessary. Additionally, familiarity with hardware mapping and address generation is essential to correctly place machine code in various memory areas and sections.

1.1 Problem Description

In this project, we will work with the STM32F446RE microcontroller to directly access its hardware peripherals and registers. This direct access enables interaction with different components and allows us to develop custom functionalities. The absence of HAL or CMSIS grants us full control over low-level programming, leading to a more profound understanding of the microcontroller's architecture and functionality.

To effectively organize our code and manage the build process, makefiles will be utilized. The use of makefiles will automate the build process, making it easier to compile and link our code without relying on complex Integrated Development Environments (IDEs). Additionally, we must carefully configure the linker to place code and data in appropriate memory regions and sections, ensuring proper execution and efficient utilization of resources.

This assignment offers valuable hands-on experience in bare-metal programming and the creation of a minimalist, yet efficient, driver for the STM32F446RE microcontroller. By completing this project, we will demonstrate our ability to work directly with hardware and write optimized code without depending on higher-level software libraries.

1.2 Requirements

In this assignment, we are required to develop a basic driver for bare-metal programming on the STM32F446RE microcontroller with the following general requirements:

1. Our task is to implement code that configures the system clock of the STM32F446RE to run at 180MHz.
2. We need to write code to configure the GPIOA port as an input (to read input from PA1) and as an output (to control the LED using PA4).
3. The program should be capable of reading input from PA1.
4. The program should have the ability to control the LED connected to PA4.
5. We must create a data structure representing the RCC (Reset and Clock Control) registers to configure the system clock.
6. A data structure needs to be designed to represent the GPIOA port's registers to configure its functionality.

7. Our task includes manually creating the NVIC (Nested Vectored Interrupt Controller) vector table, containing addresses of Interrupt Service Routines (ISRs) for handling interrupts.
8. We have to set up the Makefile to automate the build process, configure the linker script to correctly place code and data in memory regions, and design the loader to download the compiled code to the STM32F446RE microcontroller.

After completing the general requirements, we will create a user test program to verify the functionality of our cross-compilation system. The test program should adhere to the following requirements: The main objective of this test program is to perform the following tasks on the STM32F446RE microcontroller:

1. Configure the system clock to run at 180MHz.
2. Enable the GPIOA port for further configuration.
3. Configure PA1 as an input pin to read external input.
4. Configure PA4 as an output pin to control the LED.
5. Ensure that the LED connected to PA4 glows when PA1 is connected to VCC (logic high).
6. Once the test program is completed, compile it, and download the binary file to the STM32F446RE microcontroller board.

By accomplishing this assignment, we will gain valuable hands-on experience in bare-metal programming, demonstrating our ability to work directly with hardware and write optimized code without relying on higher-level software libraries like HAL or CMSIS.

2 Theory

2.1 Memory Mapping

During the boot process of an ARM MCU, it reads a crucial data structure called the "vector table" located at the beginning of flash memory. The vector table is a standardized concept across all ARM MCUs and consists of an array of 32-bit addresses corresponding to interrupt handlers. The first 16 entries in the vector table are reserved by ARM and are common to all ARM MCUs. These entries serve as default handlers for various exceptions and interrupts.

For the STM32F429 MCU, the vector table comprises a total of 91 peripheral interrupt handlers in addition to the standard 16. The peripheral interrupt handlers are specific to the STM32F429 MCU and correspond to various hardware peripherals.

Each entry in the vector table points to a function address that the MCU executes when a corresponding hardware interrupt (IRQ) is triggered. The first two entries in the vector table play a crucial role in the MCU boot process:

1. The first 32-bit value in the vector table represents the initial stack pointer. It sets up the stack for the program's execution.

2. The second 32-bit value in the vector table points to the address of the boot function to be executed. This boot function serves as the firmware's entry point, and when the MCU boots, it jumps to this address to start the execution of the firmware

It is essential to ensure that our firmware is arranged in such a way that the second 32-bit value in the flash memory contains the address of our designated boot function. This ensures that when the MCU boots up, it reads this address from flash and begins executing our boot function.

By properly configuring the vector table and setting up the boot function address correctly, we can control the initial execution flow of the MCU and start our firmware's operation as desired.

2.2 Absolute Addressing

Absolute addressing is a fundamental aspect of low-level, bare-metal programming for STM32 microcontrollers, where direct access to specific hardware registers and memory-mapped peripherals is necessary. By using absolute addressing, developers can precisely and efficiently interact with hardware resources without relying on abstractions or intermediate layers. This level of control is crucial for tasks that demand optimal performance and minimal overhead, such as real-time control applications or low-level device drivers.

2.2.1 GPIOx and RCC Addresses

From the reference manual we get to know the addresses of each of these registers.

```
struct gpio {
    volatile uint32_t MODER, OTYPER, OSPEEDR, PUPDR, IDR, ODR, BSRR, LCKR, AFR[2];
    };
#define GPIOA ((struct gpio *)(0x40020000))
```

The starting address of `GPIO` is at `0x40020000` and each `GPIO` has a size of `1 KB` thus `GPIO(A)` would point to first `GPIO`.

```
struct rcc {
    volatile uint32_t CR, PLLCFGR, CFGR, CIR, AHB1RSTR, AHB2RSTR, AHB3RSTR,
    RESERVED0, APB1RSTR, APB2RSTR, RESERVED1[2], AHB1ENR, AHB2ENR, AHB3ENR,
    RESERVED2, APB1ENR, APB2ENR, RESERVED3[2], AHB1LPENR, AHB2LPENR,
    AHB3LPENR, RESERVED4, APB1LPENR, APB2LPENR, RESERVED5[2], BDCR, CSR,
    RESERVED6[2], SSCGR, PLLI2SCFGR;
};
#define RCC ((struct rcc *)0x40023800)
```

`RCC` starts at `0x40023800`. All of this information can be found in the data sheet for the STM32F446RE.

2.2.2 The weak attribute

The weak attribute is a feature in some C and C++ compilers that allows you to define weak symbols. A weak symbol is a symbol whose definition can be overridden by a strong symbol with the same name during the linking process. If no strong symbol is found, the weak symbol's definition is used. This attribute is particularly useful in certain situations where you want to provide a default implementation for a symbol, but also allow it to be easily replaced or overridden if needed.

```c
void weak_function() __attribute__((weak));

void weak_function() __attribute__((weak)) {
    // Default implementation for weak_function
    // ...
}
```

In the above examples, `weak_function` is declared as a weak symbol. If no other strong symbol with the same name is found during the linking process, the weak symbol's definition will be used. However, if a strong symbol for `weak_function` is found (perhaps in a different object file or library), the strong symbol will override the weak one.

2.3 Compiler and Linker Flags

We have specified a number of flags for the cross compiler. As the host systems options will not work on the target system. The options for the cross compiler are listed below. Only the ones used have been described further commands are linked in the bibliography.

2.3.1 Compiler Flags

Options for Controlling the Type of Output

- `-c`: Compile or assemble the source files, but do not link. The linking stage is not performed, and the ultimate output is in the form of an object file for each source file.
- `-o`: Place the primary output in the file `file`. This option applies to whatever sort of output is being produced, whether it be an executable file, an object file, an assembler file, or preprocessed C code. The `-o` option specifies the name of the output file.

ARM Architecture-Specific Options To cater to ARM architecture-specific requirements, we have options that allow us to specify the target machine and provide machine-specific compiler configurations:

- `-mcpu=cpuname`: This option designates the target ARM architecture's name. GCC utilizes this information to determine the appropriate instructions to generate when producing assembly code. It can be used in conjunction with or as an alternative to the `-march=` option. The `cpuname` is selected from a predefined set of allowed names.

- `-mthumb`: By using this flag, the compiler generates code suitable for execution in the Thumb state. For ARM microcontrollers, like Cortex-M4 CPUs, which only support Thumb instructions, this option is essential.

C Dialect Options For selecting the desired C language standard, we have the following option:

- `-std=standard`: This option determines the C language standard to be used during compilation. The `standard` is chosen from the available C standards, such as `c89`, `c99`, `c11`, etc. By specifying this option, we ensure that the compiler adheres to the specified C language standard.

Warnings To enable compiler warnings and catch potential issues in the code, we have the following option:

- `-Wall`: By using this option, all warnings related to code constructions and some language-specific warnings are enabled. It is a good practice to include this option during compilation to improve code quality and identify potential problems.

Optimization Flags To control the level of code optimization during compilation, we can use the following option:

- `-O0`: This option sets the compilation for minimal optimization, prioritizing shorter compilation times and producing expected debugging results. Most optimizations are disabled entirely at this level, making it suitable for debugging purposes. The default optimization level is `-O0`.

The above compilation options provide control over various aspects, such as target architecture, language standards, warnings, and optimization levels, allowing us to tailor the compilation process to the specific needs of the ARM-based microcontroller programming.

2.3.2 Linker Flags

When you link your compiled object files into an executable or library, you can utilize various linker options to control the process and specify additional settings:

- `-T script`: This option specifies the linker script to be used during the linking process. The linker script provides instructions to the linker on how to organize and place the code and data in memory. This option is widely supported by systems using the GNU linker. In certain scenarios, such as bare-board systems without an operating system, using the `-T` option may be necessary to avoid references to undefined symbols.
- `-nostdlib`: When used during linking, this option instructs the linker not to use the standard system startup files or libraries. It means that no default startup files will be included in the final executable, and only the libraries explicitly specified by the programmer will be linked. Any options specifying linkage of the system libraries will be disregarded.

- `-Wl,option`: This option passes `option` as an argument to the linker. If `option` contains commas, it is split into multiple options at the commas. This syntax allows you to pass specific linker options to control various aspects of the linking process. For example, using `-Wl,-Map=output.map` requests the generation of a memory map file (`output.map`) alongside the linking. The memory map file provides valuable information about memory usage, symbols, and sections in the final executable.

By utilizing these linker options, you gain flexibility and control over the linking process, allowing you to customize how your code and libraries are combined to create the final executable or library. These options are particularly beneficial in embedded systems and low-level programming, where precise control over memory layout and library linkage is essential for optimizing the program's performance and memory utilization.

2.4 Makefile

A Makefile is a script used in software development to automate the build process of a project. Its main purpose is to provide a set of rules and instructions for compiling source code, linking object files, and generating the final executable or library. By using a Makefile, developers can ensure that only the necessary parts of the project are recompiled when changes are made, thus reducing build time.

2.4.1 Makefile Example

A Makefile is composed of rules that define dependencies and actions. Each rule consists of the following components:

```
target: prerequisites
    command 1
    command 2
    ...
```

- **target:** The target is the output file generated by the rule. It can be an executable, a library, or an intermediate object file.
- **prerequisites:** Prerequisites are the files or dependencies required to build the target. These can be source files, header files, or other intermediate files.
- **command:** The commands are shell commands or compiler/linker invocations that execute the build steps for generating the target. Each command must be preceded by a tab character.

When executing 'make', it searches for a Makefile in the current directory and follows the rules defined within it to build the project. By default, 'make' builds the first target mentioned in the Makefile. However, you can specify a different target to build by providing its name as an argument to 'make'.

Makefiles are highly customizable and allow developers to define variables, macros, and conditional statements to control the build process based on various factors like target architecture or compilation mode.

Overall, using a Makefile streamlines the build process and ensures efficient development by automating compilation tasks and maintaining dependencies between files.

```
CFLAGS  ?= -mcpu=cortex-m4 -mthumb
LDFLAGS ?= -T link.ld -nostdlib -Wl,-Map=$@.map
SOURCES = main.c startup.c

build: firmware.bin

firmware.elf: $(SOURCES)
	arm-none-eabi-gcc $(SOURCES) $(CFLAGS) $(LDFLAGS) -o $@

firmware.bin: firmware.elf
	arm-none-eabi-objcopy -O binary $< $@

flash: firmware.bin # write binary
	st-flash --reset write $< 0x8000000
```

2.4.2 Makefile working procedure

The provided Makefile is for building and flashing a firmware binary for an ARM Cortex-M4 microcontroller. Let's break down the Makefile and its rules:

```
CFLAGS  ?= -mcpu=cortex-m4 -mthumb
LDFLAGS ?= -T link.ld -nostdlib -Wl,-Map=$@.map
SOURCES = main.c startup.c
```

In these lines, the Makefile defines two variables, CFLAGS and LDFLAGS, with default values for compiler and linker options. It also sets the SOURCES variable to the list of source files (main.c and startup.c) that are used to build the firmware.

```
build: firmware.bin
```

This rule specifies the default target build, which depends on the target firmware.bin. When you run make, it will build the firmware.bin target.

```
firmware.elf: $(SOURCES)
	arm-none-eabi-gcc $(SOURCES) $(CFLAGS) $(LDFLAGS) -o $@
```

This rule defines how to build the firmware.elf target. It depends on the source files listed in $(SOURCES). The arm-none-eabi-gcc command is used to compile the source files into an ELF executable (firmware.elf) using the specified CFLAGS and LDFLAGS.

```
firmware.bin: firmware.elf
        arm-none-eabi-objcopy -O binary $< $@
```

This rule defines how to build the firmware.bin target. It depends on the firmware.elf target. The arm-none-eabi-objcopy command is used to convert the ELF executable into a binary file (firmware.bin).

```
flash: firmware.bin # write binary
        st-flash --reset write $< 0x8000000
```

2.5 Linker Script

This rule defines the flash target, which depends on the firmware.bin target. When you run make flash, it will use st-flash to write the binary file (firmware.bin) to the microcontroller's flash memory starting at address 0x8000000.

A linker script is a configuration file used by the linker during the build process to define the memory layout of a program or firmware. It provides instructions to the linker on how to organize and allocate the various sections of code and data in the target microcontroller's memory. Linker scripts are essential in bare-metal programming as they enable precise control over memory regions, section placement, and symbol definitions.

The linker script typically contains the following key components:

1. **Memory Regions**: The linker script defines memory regions such as flash memory (for program code), RAM (for data and stack), and any other peripheral memory areas. Each memory region is specified with its starting address and size.

2. **Section Placement**: The linker script specifies how different sections of the compiled code and data should be placed in memory. Common sections include '.text' (code), '.data' (initialized data), '.bss' (uninitialized data), and more. Proper section placement ensures that the program's instructions and data are correctly located in memory.

3. **Symbol Definitions**: The linker script declares symbols for various addresses and sizes that are used by the linker and the program itself. These symbols are often used to define the start and end of specific sections or to access memory-mapped hardware registers.

4. **Stack and Heap Configuration**: The linker script determines the stack and heap size and their initial addresses in memory. The stack is essential for managing function calls and local variables, while the heap is used for dynamic memory allocation (e.g., using 'malloc()' in C).

5. **Exception and Interrupt Handling**: The initial vector table containing addresses of exception and interrupt handlers is defined in the linker script. The vector table must be correctly placed in memory to ensure proper handling of exceptions and interrupts, as managed by the microcontroller's NVIC.

2.5.1 Linker Script Syntax

The syntax of a linker script can vary depending on the specific linker being used (e.g., GNU ld, arm-none-eabi-ld, etc.) and the target architecture. However, there are common elements in the syntax that are shared across different linker scripts. Below is a general outline of the linker script syntax:

Memory Region Definition:

```
MEMORY
{
    region_name (attribute) : ORIGIN = starting_address, LENGTH = size
    ...
}
```

In this section, you define the memory regions of the target microcontroller. Each memory region is given a unique `region_name` and specified with its ORIGIN (starting address) and LENGTH (size). The optional attribute specifies the region's access permissions (e.g., rx for read and execute, rwx for read, write, and execute).

Section Placement:

```
SECTIONS
{
    section_name :
    {
        *(.section_name1)
        *(.section_name2)
        ...
    } > region_name
    ...
}
```

In this section, you define how different sections of code and data are placed in the memory regions. Each section is specified with its `section_name`, and you can include multiple sections within the curly braces. The asterisk * represents all objects of a specific section. The > operator is used to indicate that the sections should be placed in the specified `region_name`.

Symbol Definitions:

```
symbol = expression;
```

In this section, you can define symbols with specific addresses or values that will be used in the linker script or accessed in the program. Symbols are defined using an equal sign = followed by an expression.

Special Sections:

```
__attribute__((section("section_name")))
```

In some cases, you may need to specify custom sections for specific variables or functions in your code. You can use the `_attribute_` syntax to place certain variables or functions into specific sections.
These are the basic elements of a linker script. Depending on your specific requirements and target architecture, you may need to include additional directives, options, or expressions in your linker script to tailor it to your project's needs. Always refer to the documentation of your specific linker and target architecture for precise details on the syntax and available options.

2.5.2 Explanation

```
ENTRY(_reset);
MEMORY {
  flash(rx)  : ORIGIN = 0x08000000, LENGTH = 512k
  sram(rwx) : ORIGIN = 0x20000000, LENGTH = 128k
}
_estack     = ORIGIN(sram) + LENGTH(sram);

SECTIONS {
  .vectors  : { KEEP(*(.vectors)) }   > flash
  .text     : { *(.text*) }           > flash
  .rodata   : { *(.rodata*) }         > flash

  .data : {
    _sdata = .;
    *(.first_data)
    *(.data)
    _edata = .;
  } > sram AT > flash
  _sidata = LOADADDR(.data);

  .bss : {
    _sbss = .;
    *(.bss)
    _ebss = .;
  } > sram

  . = ALIGN(8);
  _end = .;
}
```

This linker script provides instructions to the linker on how to organize and place code and data in the memory regions of the microcontroller. Let's explain each section of the linker script:

1. **'ENTRY(_reset);':** This specifies the entry point of the program, which is the address of the '_reset' function. When the microcontroller starts executing code, it will jump to this address to begin program execution.

2. **'MEMORY ...':** This section defines the memory regions of the microcontroller. There are two memory regions defined here:

   - **'flash(rx)':** This region starts at memory address '0x08000000' and has a length of '512k'. It is marked as read and execute ('rx') to store the program code.
   - **'sram(rwx)':** This region starts at memory address '0x20000000' and has a length of '128k'. It is marked as read, write, and execute ('rwx') to store data and the stack.

3. **'_estack':** This symbol defines the end address of the stack. It is calculated as the sum of the starting address ('ORIGIN(sram)') and the length of the SRAM ('LENGTH(sram)').

4. **'SECTIONS ...':** This section specifies how different sections of the compiled code and data should be placed in the memory regions.

   - **'.vectors':** This section contains the interrupt vector table. The 'KEEP' directive ensures that the vector table is preserved and not optimized out by the linker. It is placed in the 'flash' memory region.
   - **'.text':** This section contains the program code (instructions). It is placed in the 'flash' memory region.
   - **'.rodata':** This section contains read-only data (e.g., constant strings). It is placed in the 'flash' memory region.
   - **'.data':** This section contains initialized data (e.g., global and static variables with initial values). It is placed in the 'sram' memory region, but also mapped to the 'flash' memory region using 'AT $>$ flash'. This means that the data is loaded from flash into SRAM during program startup.
   - **'.bss':** This section contains uninitialized data (e.g., global and static variables without initial values). It is placed in the 'sram' memory region.
   - **'ALIGN(8)':** This directive ensures that the location counter is aligned to an 8-byte boundary. This can be useful for memory alignment optimizations.
   - **'_end':** This symbol defines the end address of the program. It is set to the current location counter, representing the end of all sections.

3 Debugging

Open On-Chip Debugger (OpenOCD) is an open-source tool used for debugging and programming microcontrollers and other embedded devices. It provides a versatile and flexible solution for interacting with the on-chip debugging hardware present in many microcontrollers, such as ARM Cortex-M and RISC-V based devices.Let's explore some key concepts and commands used in debugging with OpenOCD:

1. **Target:** The "target" refers to the microcontroller or embedded device that is being debugged. OpenOCD establishes a connection to the target and allows developers to interact with its on-chip debugging features.

2. **Adapter:** An "adapter" is the hardware interface that connects the development machine (host) to the target microcontroller. It can be a JTAG or SWD adapter, providing the necessary communication channels to access the debugging features.

3. **GDB Server:** OpenOCD acts as a GDB (GNU Debugger) server, allowing developers to use GDB to interact with the target. GDB is a powerful command-line debugger that provides various commands for debugging code. Now, let's go through some essential OpenOCD commands:

    - **Initialization:** Before debugging, OpenOCD needs to initialize the target and establish communication with it. The 'init' command is used to initialize OpenOCD, and it may involve setting up the adapter and other configurations.
    - **Target Configuration Script:** OpenOCD uses configuration scripts to define the target microcontroller and its specific debugging interface. These scripts contain information about the target's memory layout, registers, and debugging features. The 'target' command is used to load and execute the configuration script.
    - **Resetting the Target:** The 'reset' command is used to reset the target microcontroller. This is typically done before starting a debugging session to ensure the target starts in a known state.
    - **Halting and Resuming Execution:** The 'halt' command is used to halt the execution of the target microcontroller, while the 'resume' command is used to continue its execution after it has been halted.
    - **Setting Breakpoints:** The 'bp' command is used to set breakpoints in the target code. Breakpoints are used to stop the execution of the program at a specific location, allowing developers to inspect variables and the program's state.
    - **Reading and Writing Memory:** The 'md' command is used to read memory from the target, while the 'mw' command is used to write data to memory. These commands are useful for inspecting the contents of memory locations and modifying variables during debugging.
    - **Single-Stepping:** The 'step' command is used to perform a single step of the program's execution. This allows developers to move through the code instruction by instruction to examine its behavior.

- **Register Access:** The ‘reg‘ command is used to read and write CPU registers in the target microcontroller. This allows developers to inspect the values of registers, which store important data during program execution.

These are just some of the key concepts and commands used in debugging with OpenOCD. Mastering these concepts and commands enables developers to efficiently debug and troubleshoot their code on microcontrollers and embedded devices.

For the purpose of debugging the STM32F446RE board, we are required to use the "monitor" prefix for all OpenOCD commands, as we will be interacting with OpenOCD through a GDB session on a Telnet client (such as Tabby).

The following steps outline the process:

1. **Start OpenOCD:** Run OpenOCD with the appropriate configuration files. Open a terminal or command prompt and navigate to the directory where your configuration files are located. Use the openocd command followed by the names of the configuration files.

   ```
   openocd -f target_config.cfg -f adapter_config.cfg
   ```

2. **Testing Connection:** After starting OpenOCD, it should initialize and connect to the target microcontroller. Check the console output for any error messages. You can also use OpenOCD commands (e.g., reset, halt) to verify that the connection is working correctly.

3. **Debugging with GDB:** Once OpenOCD is running and connected to the target, you can use GDB to interactively debug your code. Open a separate terminal or command prompt and run the appropriate GDB for your target architecture.

   ```
   arm-none-eabi-gdb
   ```

   In GDB, connect to the OpenOCD server using the target remote command.

   ```
   target remote localhost:3333
   ```

   Now you can use GDB commands to set breakpoints, inspect memory and registers, and step through the code.

4. **Monitor:** The monitor reset is used to perform a reset and initialize the target microcontroller.

   ```
   monitor reset init
   ```

5. **Flash :** The Flash is to flash the binary image into the flash memory of the target microcontroller.

```
monitor flash write_image erase target/out.elf
```

After executing this command, OpenOCD will erase the flash memory on the target microcontroller and then program the out.el binary into the flash memory, making it ready for execution. The specific memory address at which the binary is flashed depends on the memory layout and configuration defined in the linker script and target configuration.

After executing the command in the GDB session connected to OpenOCD, the firmware binary "firmware.elf" will be written to the target's flash memory, replacing the existing contents. Upon power-on or reset, the target device will boot the new firmware.

To ensure a successful flash, it is essential to compile the firmware binary correctly for the target architecture and specify the correct memory address in the OpenOCD configuration file to write the binary to the appropriate location in flash memory.

With the GDB session connected to OpenOCD, you can now issue various debugging commands to interact with the target microcontroller. Some commonly used commands include:

- **'monitor halt':** Halts the target's execution.
- **'monitor reset':** Resets the target.
- **'load':** Loads the compiled program or firmware into the target's memory.
- **'monitor resume':** Resumes the target's execution after being halted.
- **'monitor reset halt':** Combines 'monitor reset' and 'monitor halt', used to reset the target and immediately halt its execution.
- **'break' or 'b':** Sets a breakpoint at a specific address or function in the target's code.
- **'step' or 's':** Single-steps through the target's code, executing one instruction at a time.
- **'continue' or 'c':** Continues the target's execution from a breakpoint or halt state.
- **'info registers':** Displays the target's register values.
- **'quit':** Exits the GDB session and closes the connection to OpenOCD.

These commands allow you to interact with the STM32F4DISCOVERY board, examine its state, and debug the firmware running on the target microcontroller.

Please note that the exact syntax and behavior of GDB and OpenOCD commands may vary depending on your specific setup, target architecture, and OpenOCD configuration file. Always refer to the documentation and tutorials specific to your use case for accurate and detailed information.

4 Implementation

4.1 Reset Handler

```
__attribute__((naked, noreturn)) void _reset(void) {
  extern long _sbss, _ebss, _sdata, _edata, _sidata;
  for (long *src = &_sbss; src < &_ebss; src++) *src = 0;
  for (long *src = &_sdata, *dst = &_sidata; src < &_edata;) *src++ = *dst++;

  extern void main(void);
  main();
  for (;;) (void) 0;
}
```

The code snippet provided appears to be a low-level initialization routine typically found in embedded systems programming for microcontrollers. Let's break down the code:

1. **_attribute_((naked, noreturn)) void _reset(void)‘:** This is a function named _reset, and the attributes naked and noreturn are applied to it.

   - **naked:** The ‘naked‘ attribute tells the compiler not to generate any prologue or epilogue code for the function. It means that the function does not have the usual entry and exit sequences, including stack frame setup and teardown. Instead, the programmer is responsible for managing these aspects manually.
   - **‘noreturn‘:** The ‘noreturn‘ attribute indicates that the function will not return to its caller. In this case, the function ‘_reset‘ is not expected to return, as it typically represents the microcontroller's reset handler that runs only once during startup.

2. **‘extern long _sbss, _ebss, _sdata, _edata, _sidata;‘:** These are external symbols representing various memory locations that are used for initialization purposes. They are expected to be defined and assigned values in other parts of the program or linker script.

3. **‘for (long *src = &_sbss; src < &_ebss; src++) *src = 0;‘:** This loop initializes the BSS section of the memory. The BSS section typically contains uninitialized global and static variables, and by setting them to zero, the loop effectively clears the memory.

4. **‘for (long *src = &_sdata, *dst = &_sidata; src < &_edata;) *src++ = *dst++;‘:** This loop copies the data segment from the read-only (flash) memory to the initialized data section in RAM. This ensures that initialized global and static variables have the correct initial values at runtime.

5. **‘extern void main(void);‘:** This declares the ‘main‘ function, which is the entry point of the user code. The ‘main‘ function is defined elsewhere in the code and is called after the initialization is complete.

6. **‘main();‘:** This calls the ‘main‘ function, which starts the user application. Since the ‘_reset‘ function has the ‘noreturn‘ attribute, the program will not return to this point after executing ‘main()‘.
7. **‘for (;;) (void) 0;‘:** This creates an infinite loop that runs after ‘main()‘ completes. Since the ‘_reset‘ function is not supposed to return, this infinite loop ensures that the microcontroller does not exit or execute random code from memory.

Overall, this code sets up the microcontroller’s environment by clearing the BSS section, initializing the data segment, and then calling the ‘main()‘ function to start the user application. It demonstrates a low-level startup routine typically found in bare-metal programming for microcontrollers, where developers have full control over the hardware and initialization process.

4.2 Interrupt Handlers

```
extern void SysTick_Handler(void);
```

The extern keyword indicates that the SysTick_Handler function is defined elsewhere in the code, and the compiler should treat it as an external symbol. The actual definition of the SysTick_Handler function will be provided in another part of the code or in a separate library file that implements the interrupt service routine (ISR) for the SysTick timer.

By declaring the SysTick_Handler function with extern, the compiler knows that this function will be linked later when the complete code is built and linked together, allowing proper integration of the interrupt handling.

```
extern void _estack(void);
```

_estack refers to the stack pointer, which is a special register that points to the top of the stack in the microcontroller’s memory. The stack is used for managing function calls, local variables, and other data during program execution. By declaring _estack with extern, the compiler knows that the stack pointer will be defined elsewhere in the code, typically in the microcontroller’s hardware-specific startup code or in a linker script. The actual definition of _estack will specify the memory location where the stack should be located in the microcontroller’s memory.

_estack is an important symbol used by the microcontroller’s startup code and the linker to properly set up the stack and manage program execution. Its proper definition is essential for the correct operation of the microcontroller and the management of its memory resources.

4.3 Vector Table

```
__attribute__((section(".vectors"))) void (*const tab[16 + 82])(void) = {
    _estack, _reset, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, SysTick_Handler};
```

Each element of the tab array points to an interrupt service routine (ISR) function, and the array is located in the .vectors section of the microcontroller's memory. The _attribute_((section(".vectors"))) part is used to place this array in the specific memory section named .vectors.

- **_attribute_((section(".vectors"))):** This attribute specifies that the tab array should be placed in the .vectors section of the memory. The .vectors section typically contains the vector table, which is an array of function pointers used to handle interrupts and exceptions.
- **void (*const tab[16 + 82])(void):** This is the declaration of the tab array. It is an array of constant function pointers, and its size is 16 + 82, which means it has 16 entries for the ARM Cortex-M core exceptions (e.g., reset, NMI, hard fault, etc.) and 82 entries for peripheral interrupts specific to the microcontroller. Each element of the tab array points to a function that takes no arguments and returns void.
- **_estack, _reset, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, SysTick_Handler :** This is the initializer list for the tab array. It specifies the addresses of the interrupt service routines to be executed when corresponding interrupts occur.
  - **_estack:** The first entry of the array is the initial stack pointer (_estack), which points to the top of the stack memory.
  - **_reset:** The second entry is the reset handler (_reset), which is the function called at the microcontroller's startup to perform the necessary initialization.
  - **0 (repeated):** The next 14 entries are set to 0, indicating that they are not used. These entries correspond to some reserved exceptions in the ARM Cortex-M core.
  - **SysTick_Handler:** The last entry points to the function SysTick_Handler, which is the interrupt service routine for the SysTick timer interrupt. This routine will be executed when the SysTick timer interrupt occurs.

References

[1] STMICROELECTRONICS. Datasheet - STM32F446xC/E - Arm® Cortex®-M4 32-bit MCU+FPU, 225 DMIPS, up to 512 KB Flash/128+4 KB RAM, USB OTG HS/FS, seventeen TIMs, three ADCs and twenty communication interfaces. [Online; accessed 2023-04-05].

[2] STMicroelectronics. Github - STMicroelectronics/STM32CubeF4: Stm32cube MCU Full Package for the STM32F4 series - (HAL + LL Drivers, CMSIS Core, CMSIS Device, MW libraries plus a set of Projects running on all boards provided by ST (Nucleo, Evaluation and Discovery Kits)). https://github.com/STMicroelectronics/STM32CubeF4. [Online; accessed 2023-04-05].

[3] STMICROELECTRONICS. Stm32f446xx advanced Arm®-based 32-bit MCUs - Reference manual. [Online; accessed 2023-04-05].