Bare-Metal Embedded Software Development

In the world of embedded systems, IDEs like STM32CubeIDE offer a convenient starting point. But as we evolve, it's good to understand how to code on bare metal, without an IDE. It empowers you with more control and flexibility. Here, we'll focus on how to do that using toolchains.

1. Toolchains

Purpose of a Toolchain

A toolchain is a set of software tools used to create a complete workflow for embedded software development. This includes a compiler, assembler, and linker, among other utilities. These tools work together to convert your source code into an executable program that can run on an embedded system.

Popular Toolchains

There are several toolchains you can choose from, depending on your needs. Below are some popular ones.

GNU Tools (GCC) for ARM Embedded Processors

We're going to use this toolchain for our examples, mainly because it's robust, open-source, and widely supported. If you've been using STM32CubeIDE, you should already have this toolchain installed on your system.

Keil MDK-ARM

Another option is the Keil MDK-ARM toolchain. This one's a commercial product but offers some additional features that could be useful in a professional setting, like advanced debugging and real-time OS support.

2. Getting Started

Installing the Toolchain

This should already be done if you've been using STM32CubeIDE. If not, you can download the toolchain from here. Make sure to select the correct version for your operating system.

Toolchain Naming

The name `arm-none-eabi-gcc` is actually a breakdown of target specifications and tells you a lot about what the tool is designed for. Let's break it down:

- **ARM**: This indicates that the toolchain is targeting ARM processors.
- **None**: This refers to the vendor. "None" means it's not specific to any vendor, making it more versatile.
- **EABI**: Stands for "Embedded Application Binary Interface". This specifies the standard for software interfaces in the embedded system.
- **GCC**: GNU Compiler Collection. This is the actual compiler, and in this case, it's set up to work with the above parameters.

So, in short, `arm-none-eabi-gcc` is the GNU Compiler Collection configured for targeting ARM processors with no specific vendor in mind, adhering to the Embedded Application Binary Interface.

Confirming the Installation

To check if GNU GCC for ARM is installed on your Mac, open Terminal and run the following command:

```
arm-none-eabi-gcc --version
```

If the toolchain is installed, this command will output version information for the ARM GCC compiler. If it's not installed, you'll likely see a message like "command not found."

If you've been using STM32CubeIDE, the toolchain might be installed but not added to your system's PATH. In that case, you can locate it within the STM32CubeIDE installation folder and run it with the full path.

Open the `~/.zshrc` file in TextEdit (macOS only):

```
open -a TextEdit ~/.zshrc
```

Add the following line to the end of the file:

```
# GNU Tools (GCC) for ARM Embedded Processors
export
PATH=$PATH:/Applications/STM32CubeIDE.app/Contents/Eclipse/plugins/com.st.stm32cube.ide.mcu.externaltools.gnu-tools-for-stm32.10.3-2021.10.macos64_1.0.0.202111181127/tools/bin/
```

Adjust the path to match the location of the toolchain on your system. Then save and close the file. Finally, run the following command to apply the changes:

```
source ~/.zshrc
```

Now you should be able to run the `arm-none-eabi-gcc --version` command from anywhere in the terminal.

Creating a Project

We will reuse the Task Scheduler project from Section 14. Create a new folder and copy the `Inc` and `Src` folders from the Task Scheduler project into it.

---

3. Build Process

Preprocessor

Input: .c files
Output: .i files
What happens:
The preprocessor takes your source `.c` files and applies transformations like macro expansion, file inclusion, and conditional compilation. The output is an intermediate `.i` file, which is essentially your source code with all preprocessing directives resolved.

Compiler

Input: .i files
Output: .s files
What happens:
The compiler reads the `.i` files and translates them into assembly code,

outputting `.s` files. This is the stage where most of the syntax checking and error reporting happen. The compiler optimizes the code but doesn't yet create machine code.

Assembler

Input: .s files
Output: .o files
What happens:
The assembler takes the `.s` assembly files and translates them into processor-specific machine code. These `.o` files are still "relocatable", meaning they don't contain information about where they will reside in memory.

Linker

Input: .o files
Output: .elf files
What happens:
While the compiler can go directly to `.o` files, the linker's role is to take these `.o` files and combine them into a single `.elf` ("Executable and Linkable Format") file. The linker also resolves all symbolic references to absolute addresses.

(Optional) Object Copy

Input: .elf files
Output: .bin (binary file) or .hex (hexadecimal file)
What happens:
This step is optional and is used to transform the `.elf` file into a more easily flashable or distributable format, such as `.bin` or `.hex`. It doesn't change the actual machine code but merely rearranges it into a different format.

(Optional) Object Dump

Input: .elf files or .o files **Output**: .lst (list file)
What happens:
The Object Dump utility (`objdump`) disassembles the `.elf` file, giving you a detailed view of its contents. It provides insights into the assembly instructions, addresses, and other information that make up your program.

The output is usually a `.lst` file that contains the disassembled code along with the original source code, allowing you to see how your high-level code translates into machine instructions.

This can be incredibly useful for debugging and understanding the low-level operations of your program. If you've ever wondered how your C functions look in assembly or how variables are stored in memory, the `.lst` file will be a valuable resource. It's not a necessary step for generating a runnable binary, but it's often used for deep debugging and analysis.

To generate a `.lst` file, you might use a command like:

```
arm-none-eabi-objdump -S ./Obj/main.o > ./Lst/main.lst
```

This will generate a `.lst` file named `main.lst` containing both source and disassembly information if available.

---

4. Makefile

```
# Toolchain
CC := arm-none-eabi-gcc
LD := arm-none-eabi-ld

# Flags
CFLAGS := -mcpu=cortex-m4 -mthumb -Wall -g -O0

# Directories
SRC_DIR := Src
INC_DIR := Inc
OBJ_DIR := Obj

# Files
SRC_FILES := $(wildcard $(SRC_DIR)/*.c)
OBJ_FILES := $(patsubst $(SRC_DIR)/%.c,$(OBJ_DIR)/%.o,$(SRC_FILES))

# Targets
all: main.elf

main.elf: $(OBJ_FILES)
    $(CC) $(CFLAGS) -o $@ $^

$(OBJ_DIR)/%.o: $(SRC_DIR)/%.c
    $(CC) $(CFLAGS) -I$(INC_DIR) -c -o $@ $<

clean:
    rm -f $(OBJ_DIR)/*.o main.elf

.PHONY: all clean
```

1. **LD**: `LD` specifies the linker, which in this case is set to `arm-none-eabi-ld`. The linker is the tool that takes multiple object files and libraries, then combines them to produce a single executable file. In this Makefile, `LD` isn't actually used because `CC` is taking care of the linking step as well. It's just there for completeness or future use.

2. **Compiler Flags**:
    - `-mcpu=cortex-m4`: Specifies the target CPU architecture, in this case, the Cortex-M4.
    - `-mthumb`: Specifies that Thumb instructions should be generated, a more compact instruction set for ARM.
    - `-Wall`: Enables all common warning messages.
    - `-g`: Produces debugging information.
    - `-O0`: Optimization level zero; this means no optimization, making debugging easier.
3. **wildcard and patsubst**:
    - `$(wildcard $(SRC_DIR)/*.c)`: This finds all `.c` files in the `$(SRC_DIR)` directory.
    - `$(patsubst $(SRC_DIR)/%.c,$(OBJ_DIR)/%.o,$(SRC_FILES))`: This transforms the list of source files into a list of object files, changing their directory and extension.
4. **all: main.elf**: This is a make target. When you run `make` with no arguments, it will build the first target in the Makefile, which in this case is `all`. This target depends on `main.elf`, meaning `main.elf` will be built when you run `make`.
5. **Special Variables**:
    - `$@`: Represents the target file name.
    - `$^`: Represents all the prerequisites (dependencies) for the target.
6. **clean**: This is a utility target that removes all generated files like object files and the `main.elf` executable. This keeps your directory clean, allowing you to start a fresh build.
7. **.PHONY**: This tells `make` that `all` and `clean` are not files. This is useful in case you have or will have files named `all` or `clean` in the directory; `make` won't get confused and will still execute the corresponding recipes.

---

5. Code and Data in a Program

Programs can be broken down into several parts, often represented by specific sections in the compiled binary.

Part	Storage	Extension	Description
Text Segment	Code Memory	`.text`	Contains the executable instructions of a program.
Constants	Read-Only Memory (ROM) or Code Memory	`.rodata`	Immutable values that are often optimized to be stored in read-only memory.
Literals	Code Memory or RAM	Various	Directly embedded in the code or sometimes stored in read-only data sections.
Data Segment	RAM	`.data`	Initialized global and static variables.
BSS Segment	RAM	`.bss`	Unitialized global and static variables. Initialized to zero.

> Note: All ROM can serve as code memory, but not all code memory has to be ROM, especially in systems where the code can be updated or loaded dynamically.

---

6. Linker and Locator

Responsibilities

Linker

- **Address Conflicts**: When you inspect the `.lst` files generated by the object dump, it becomes apparent that some sections may have overlapping or relative addresses. The linker resolves these conflicts by assigning absolute addresses, based on a set of rules and a linker script, if provided.
- **Undefined Symbols**: During the compilation phase, certain symbols (like function or variable names) might remain undefined. These are usually part of other object files or libraries. The linker resolves these undefined symbols by matching them with their definitions in other files.

Locator

Linker Script: Sometimes called the "Locator," this component of the linker takes a linker script you write to understand how you want different sections (.text, .data, .bss, etc.) to be laid out in memory. It assigns absolute addresses to each section based on your specifications in the linker script.

Code Memory Layout (FLASH)

- **Unused Code Memory**: Reserved space not currently being used.
- **.data**: Initialized data segment; copied into RAM at startup.
- **.rodata**: Read-only data like constants; stored in flash.
- **.text**: The actual machine code that the CPU executes, includes the startup code.
- **Startup Code**: Included within the `.text` segment; initializes variables and calls `main()`.
- **.vector table**: The interrupt vector table, vital for handling interrupts.

RAM Memory Layout

- **Stack**: Memory area where function call data is managed.
- **Heap**: Dynamic memory, if used.
- **.data**: Initialized data, loaded from FLASH at startup.
- **.bss**: Uninitialized data, zeroed at startup.

Transferring of `.data` section to RAM

In embedded systems, the `.data` segment is often stored in non-volatile memory like Flash but must be copied to RAM when the program starts. This is done by a piece of startup code before the main application runs. It ensures that variables with initial values start with those values each time the system boots up.

Initialization of `.bss` section

Unlike the `.data` section, the `.bss` section doesn't have its initial values stored in non-volatile memory like Flash. Instead, the startup code directly initializes this section in RAM to zero upon program startup.

Summary

Variable	Load Time Memory	Run Time Memory	Section	Note
Global, Initialized	FLASH	RAM	`.data`	Copied from FLASH to RAM by startup code.
Global, Uninitialized	N/A	RAM	`.bss`	Initialized to zero by startup code.
Global, Initialized, `const`	FLASH	N/A	`.rodata`	Read-only. Stays in FLASH, not moved to RAM.
Local, Initialized (within function)	N/A	Stack (RAM)	N/A	Allocated on the stack at runtime.
Local, Uninitialized (within function)	N/A	Stack (RAM)	N/A	Allocated on the stack. Initial value is garbage.
Local, Initialized, `static`	FLASH	RAM	`.data`	Copied from FLASH to RAM by startup code. Retains value between function calls.
Local, Uninitialized, `static`	N/A	RAM	`.bss`	Initialized to zero by startup code. Retains value between function calls.
Local, Initialized, `const`	N/A	Stack (RAM)	N/A	Allocated on the stack. Read-only.
Local, Initialized, `static`, `const`	FLASH	N/A	`.rodata`	Read-only. Stays in FLASH, not moved to RAM. Retains value between function calls.
Global, Initialized, `static`	FLASH	RAM	`.data`	Copied from FLASH to RAM by startup code. File scope.
Global, Uninitialized, `static`	N/A	RAM	`.bss`	Initialized to zero by startup code. File scope.
Global, Initialized, `const`, `static`	FLASH	N/A	`.rodata`	Read-only. Stays in FLASH, not moved to RAM. File scope.

- **Load Time Memory**: This indicates where the variable is stored at the time the program is loaded into memory. For example, initialized global variables might be stored in FLASH memory initially. If N/A is specified, it can mean the following depending on the variable type:
    - **Local Variables**: These are stored on the stack when their enclosing function is called. The stack is part of RAM, and space is allocated and deallocated dynamically as functions are called and return.
    - **Global Uninitialized Variables**: These are usually placed in the `.bss` section and are initialized to zero before the `main()` function is called. The startup code takes care of this.
    - **Static Local Variables**: These are similar to global uninitialized variables but have local scope. They are also typically placed in the `.bss` section if uninitialized, or `.data` if initialized.
    - **Heap Variables**: If you're using dynamic memory allocation (like `malloc` in C), the variables will be placed in the heap, which is also a part of RAM.
- **Run Time Memory**: This specifies where the variable resides when the program is running. Variables could be in RAM or on the stack, which is also a part of RAM.
- **Section**: This tells you which memory section the variable belongs to, like `.data` for initialized data, `.bss` for uninitialized data, and `.rodata` for read-only data. These sections are relevant for the linker and can be seen in the compiled output.

Example

Go through this code, and ask yourself what is the load time memory, run time memory, and section for each variable:

```c
#include <stdio.h>

// Global, Initialized
int global_init_var = 5;

// Global, Uninitialized
int global_uninit_var;

// Global, Initialized, const
const int global_const_init_var = 10;

// Global, Initialized, static
static int global_static_init_var = 15;

// Global, Uninitialized, static
static int global_static_uninit_var;

void my_function() {
    // Local, Initialized
    int local_init_var = 20;

    // Local, Uninitialized
    int local_uninit_var;

    // Local, Initialized, static
    static int local_static_init_var = 25;

    // Local, Uninitialized, static
    static int local_static_uninit_var;

    // Local, Initialized, const
    const int local_const_init_var = 30;

    printf("Dummy function to use variables.\n");
}

int main() {
    my_function();
    return 0;
}
```

Here's the table indicating the answers:

Variable	Load Time Memory	Run Time Memory	Section
`global_init_var`	FLASH	RAM	`.data`

Variable	Load Time Memory	Run Time Memory	Section
`global_uninit_var`	N/A	RAM	`.bss`
`global_const_init_var`	FLASH	N/A	`.rodata`
`global_static_init_var`	FLASH	RAM	`.data`
`global_static_uninit_var`	N/A	RAM	`.bss`
`local_init_var`	N/A	Stack (RAM)	N/A
`local_uninit_var`	N/A	Stack (RAM)	N/A
`local_static_init_var`	FLASH	RAM	`.data`
`local_static_uninit_var`	N/A	RAM	`.bss`
`local_const_init_var`	N/A	Stack (RAM)	N/A

7. Introduction to Startup File

Responsibilities

The startup file serves as the initialization script for an embedded C program. Its main responsibilities are:

1. **Initialization of Stack and Heap**: It sets the initial stack pointer and, if necessary, configures the heap memory.
2. **Data Segment Initialization**: Copies initialized variables from Flash to RAM (`.data` section).
3. **Zero Initialization**: Initializes global and static variables in RAM that haven't been explicitly initialized (`bss` section).
4. **Call Global Constructors**: If the language being used has features that require global construction (like C++), the startup file takes care of this.
5. **Call `main()`**: Finally, it jumps to the `main()` function, effectively transferring control to the user's code.

Where is the Startup File?

The startup file is usually part of the software development kit (SDK) or standard library provided for the microcontroller you're working with. When you build your code, the compiler and linker ensure that the startup code is positioned correctly within the final executable, so it runs before any application-specific code.

From the IDE to the microcontroller

When you press "Build" or "Run" in your IDE, the following happens:

1. Your code is compiled, along with the startup file.
2. The linker combines these into an executable, usually an `.elf` or `.hex` file.
3. The executable is flashed onto the microcontroller, where it's stored in non-volatile memory (like Flash).
4. On reset or power-up, the microcontroller's CPU starts executing from the startup file.

Components of the Startup File

1. **Reset Handler**: This is the function that gets called upon a reset. This is usually the entry point in the startup file.
2. **Vector Table**: Contains the addresses of the functions that should be called in response to the corresponding interrupt or exception.
3. **Initialization Routines**: These are functions or blocks of code that perform the responsibilities mentioned above (like initializing `.data` and `.bss`).
4. **Libc Initialization**: If your program uses standard C libraries, the startup file might include initialization code for this.
5. **Startup Script or Linker Script**: Often, the startup file comes with a linker script that specifies the memory layout of the program. This includes where to place `.text`, `.data`, `.bss`, etc.

The exact components can vary depending on the architecture, the compiler, and the standard library in use. But generally, these are the key parts you'll find in a startup file.

---

8. Writing a Startup File

The start up file can be either `.c` or `.s` file. The `.c` file is easier to write and understand, but the `.s` file is more efficient and can be used to write more complex startup code. We will use the `.c` file for this example.

Create a `stm32_startup.c` file in the project directory.

Vector Table

```
#include <stdint.h>

#define SRAM_START 0x20000000U
#define SRAM_SIZE (128U * 1024U) // 128 KB
#define SRAM_END ((SRAM_START) + (SRAM_SIZE))

#define STACK_START SRAM_END

extern int main(void);

/* function prototypes of STM32F407x system exception and IRQ
handlers */

void Reset_Handler(void);

void NMI_Handler(void) __attribute__((weak,
alias("Default_Handler")));
void HardFault_Handler(void) __attribute__((weak,
alias("Default_Handler")));
void MemManage_Handler(void) __attribute__((weak,
alias("Default_Handler")));
void BusFault_Handler(void) __attribute__((weak,
alias("Default_Handler")));
void UsageFault_Handler(void) __attribute__((weak,
alias("Default_Handler")));
// ... rest  of the handlers

uint32_t vectors[] __attribute__((section(".isr_vector"))) = {
    STACK_START,
    (uint32_t)&Reset_Handler,
    (uint32_t)&NMI_Handler,
    (uint32_t)&HardFault_Handler,
    (uint32_t)&MemManage_Handler,
    (uint32_t)&BusFault_Handler,
    (uint32_t)&UsageFault_Handler,
    0,
    0,
    0,
    0,
    (uint32_t)&SVC_Handler,
    (uint32_t)&DebugMon_Handler,
    0,
    (uint32_t)&PendSV_Handler,
    (uint32_t)&SysTick_Handler,
    (uint32_t)&WWDG_IRQHandler,
```

```
    // ... rest of the handlers
};
```

Default and Reset Handlers

```
void Default_Handler(void)
{
    while (1)
    {
    }
}

void Reset_Handler(void)
{
    // 1. Copy .data section to SRAM
    // TODO

    // 2. Initialize .bss section to zero in SRAM
    // TODO

    // 3. Initialize C Standard Library
    // TODO

    // 4. Call main()
    main();
}
```

9. Linker Script

Introduction

- **Section Merging**: It's a text file that outlines how various sections from the object files should be combined to create a final output file, such as an ELF binary.
- **Address Assignment**: The combination of the linker and locator refers to the linker script to assign unique, absolute addresses to different sections of the output file.
- **Memory Layout**: The script often contains information on the starting addresses and sizes of different memory regions, such as code and data memory.

- **Scripting Language**: Linker scripts are usually written using the GNU linker command language.
- **File Extension**: The standard file extension for a GNU linker script is `.ld`.
- **Compilation Flag**: During the linking phase, the linker script is typically supplied to the linker using the `-T` option.

Linker Script Commands

Here are some common GNU linker script commands along with brief code snippets:

1. **`ENTRY`**: Specifies the entry point of the program.

```
ENTRY(start)
```

1. **`MEMORY`**: Defines the memory layout.

Following the syntax:

```
MEMORY {
    memory_region_name (access_type) : ORIGIN = origin_address,
LENGTH = length
}
```

where the `access_type` can include the followings (case-insensitive):

Access Type	Description
`r`	Read-only
`w`	Write-only
`x`	Executable
`rw`	Read-write
`rx`	Read and executable
`wx`	Write and executable
`rwx`	Read, write, and executable
`!`	Inversion (make the region non-accessible)
`L`	Loadable, specifying that the data should be loaded into memory

Access Type	Description
`I`	Initialized, meaning the region should be initialized

These can also be combined in various ways to suit your needs. For example:

```
MEMORY {
    FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 1M
    RAM (rw)   : ORIGIN = 0x20000000, LENGTH = 256K
}
```

1. **`SECTIONS`**: Defines the sections of the output file and how they should be combined.

The syntax typically follows this general form:

```
SECTIONS
{
    output_section :
    {
        input_section(s)
    } >memory_region [AT> load_memory_region]
}
```

Location Counter

The location counter (denoted as `.`) represents the current address in the output section. It is used to set the starting and ending addresses of a section.

Example Breakdown:

`.text` Section

```
.text :
{
    KEEP(*(.isr_vector))
    *(.text)
    *(.text*)
    *(.rodata)
    _etext = .;
} >FLASH
```

- `KEEP(*(.isr_vector))`: Ensures that the interrupt service routine vectors are kept in the final executable.

- `*(.text)`: Includes the `.text` section of all input files.
- `*(.text*)`: Includes all sections whose names start with `.text`.
- `*(.rodata)`: Includes the `.rodata` section, which contains read-only data.
- `_etext = .;`: Records the end address of the `.text` section.
- `>FLASH`: Indicates that the `.text` section should be placed in the FLASH memory region.

`.data` Section

```
.data :
{
    _sdata = .;
    *(.data)
    _edata = .;
} >SRAM AT> FLASH
```

- `_sdata = .;`: Records the start address of the `.data` section in RAM.
- `*(.data)`: Includes the `.data` section, which contains initialized variables.
- `_edata = .;`: Records the end address of the `.data` section in RAM.
- `>SRAM AT> FLASH`: Indicates that the `.data` section should be placed in the SRAM but initialized from FLASH.

`.bss` Section

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

- `_sbss = .;`: Records the start address of the `.bss` section in RAM.
- `*(.bss)`: Includes the `.bss` section, which contains uninitialized variables.
- `_ebss = .;`: Records the end address of the `.bss` section in RAM.
- `>SRAM`: Indicates that the `.bss` section should be placed in the SRAM.

- **`OUTPUT_FORMAT`**: Specifies the format of the output file (e.g., ELF).

```
OUTPUT_FORMAT("elf32-littlearm")
```

1. **`EXTERN`**: Tells the linker that a certain symbol will be provided by another file.

```
EXTERN(main)
```

1. **`PROVIDE`**: Specifies a default value for a symbol if it's not defined elsewhere.

```
PROVIDE(_stack_top = 0x20020000);
```

1. **`. = ALIGN(n)`**: Aligns the location counter (represented by `.`) to the `n`-byte boundary. Consider an example SECTION with `.data` and `.bss` subsections:

```
SECTIONS
{
    .data :
    {
        _sdata = .;
        *(.data)
        . = ALIGN(4);
        _edata = .;
    } >SRAM

    .bss :
    {
        _sbss = .;
        *(.bss)
        . = ALIGN(4);
        _ebss = .;
    } >SRAM
}
```

In this example, we have `.data` and `.bss` sections, and both are located in SRAM.

1. `.data` **Section**:
    - `_sdata = .;` sets `_sdata` to the current value of the location counter. This is the start address of the `.data` section.
    - `*(.data)` places all `.data` segments from the input files.
    - `. = ALIGN(4);` aligns the location counter to the next 4-byte boundary. This ensures that whatever comes next starts at an address divisible by 4.
    - `_edata = .;` sets `_edata` to the aligned location counter, representing the end of `.data`.
2. `.bss` **Section**:
    - `_sbss = .;` sets `_sbss` to the current (possibly aligned from the previous section) value of the location counter.
    - `*(.bss)` places all `.bss` segments.
    - `. = ALIGN(4);` aligns the location counter to the next 4-byte boundary.
    - `_ebss = .;` sets `_ebss` to this newly aligned value, marking the end of `.bss`.

The `ALIGN` function adjusts the location counter (`.`), aligning it to a boundary that is a multiple of the specified number (4 in this case, since ARM Cortex-M4 has a 32-bit architecture). This ensures that subsequent data is placed at addresses that are easy for the CPU to access, potentially optimizing performance.

So in summary, the `ALIGN` function will modify the position where the next piece of code or data will be placed, by adjusting the location counter to an address that aligns with the specified byte boundary.

---

10. Implementing Reset Handler

We can actually access the symbols in the linker script from our `stm32_startup.c`. For example, we can access the start and end addresses of the `.data` section using the symbols `_sdata` and `_edata`, respectively:

```
extern uint32_t _etext;
extern uint32_t _sdata;
extern uint32_t _edata;
extern uint32_t _sbss;
extern uint32_t _ebss;
```

To print out symbols, we can use the command in the terminal:

```
arm-none-eabi-nm main.elf
```

The output will be something like:

```
(base) username BareMetalTaskScheduler % arm-none-eabi-nm main.elf
08000798 W ADC_IRQHandler
08000790 T BusFault_Handler
08000798 W CAN1_RX0_IRQHandler
08000798 W CAN1_RX1_IRQHandler
...
20000058 B _ebss
20000004 D _edata
080007b4 T _etext
20000004 B _sbss
20000000 D _sdata
...
```

In our `stm32_startup.c`, we can use the following code implement the `Reset_Handler`:

```
extern int main(void);

void Reset_Handler(void)
{
    // 1. Copy .data section to SRAM
    uint32_t size = &_edata - &_sdata;

    uint8_t *pDst = (uint8_t *)&_sdata; // SRAM
    uint8_t *pSrc = (uint8_t *)&_etext; // Flash

    for (uint32_t i = 0; i < size; i++)
    {
        *pDst++ = *pSrc++;
    }

    // 2. Initialize .bss section to zero in SRAM
    size = &_ebss - &_sbss;
    pDst = (uint8_t *)&_sbss;
    for (uint32_t i = 0; i < size; i++)
    {
        *pDst++ = 0;
    }

    // 3. Call main()
    main();
}
```

11. OpenOCD

Introduction

Open On-Chip Debugger (OpenOCD) is a free, open-source tool that provides debugging, in-system programming, and boundary-scan testing for embedded devices. It is commonly used for development work on various platforms and supports a wide range of debugging probes and protocols. OpenOCD is a vital tool for embedded developers and offers a bridge between your development machine and the target system's on-chip debug module.

How it works?

OpenOCD works by communicating directly with the debug access port of the target system. It establishes this link via a hardware debugging probe, such as a JTAG or SWD adapter. Once connected, it allows the host system to perform various debugging and programming operations like setting breakpoints, stepping through code, inspecting memory, and flashing firmware. Essentially, OpenOCD creates a server on your development machine, which can be interfaced with via GDB (GNU Debugger) or other compatible clients to control and monitor the target system.

The architecture of OpenOCD is quite modular, allowing it to be easily extended to support new debug protocols and chips. Configuration scripts are usually employed to set up the tool for specific hardware, making it versatile and adaptable to different development needs.

Installation

Installation steps can vary depending on the operating system you are using. Below are basic guidelines:

For macOS:

1. Open Terminal.
2. If you have Homebrew installed, run `brew install openocd`.
3. If you don't have Homebrew, you can download the source and compile it manually.

After installation, you can check if OpenOCD is successfully installed by running `openocd -v` in the command line. It should return the installed version of OpenOCD.

How to download the firmware to the microcontroller on macOS

After installing OpenOCD via Homebrew, you'll want to download your compiled firmware onto your microcontroller. For this example, let's assume you're working with an STM32-based board and you've compiled your code into a binary or ELF file (e.g., `main.elf`).

1. **Plug in your hardware debugger and connect it to your STM32 board**
   Make sure your JTAG or SWD debugger is correctly connected to your STM32 board's debugging pins.

2. **Locate the OpenOCD Configuration File**
   Depending on the debugger and board you're using, you'll need an OpenOCD configuration file that matches your setup. These are usually available in OpenOCD's `scripts` directory.

3. **Open Terminal**
   You'll be running the OpenOCD commands from here.

4. **Start OpenOCD**
   You can start OpenOCD with the proper configurations for your board and debugger. If using an STM32 and an ST-Link debugger, your command might look like this:

```
openocd -f interface/stlink.cfg -f target/stm32f1x.cfg
```

Make sure you are in the directory where your `.cfg` files are or provide the full path to them.

This will start OpenOCD and it should connect to your debugger and microcontroller. You'll see a lot of output ending with something like "Info: Listening on port 3333 for gdb connections".

1. **Open a new Terminal Tab**
   Keep the OpenOCD running in the original terminal and open a new one for the next steps.

2. **Connect to OpenOCD with GDB**
   Open GDB with your `.elf` file as the target.

```
arm-none-eabi-gdb main.elf
```

After GDB starts, connect to the OpenOCD server:

```
(gdb) target remote localhost:3333
```

1. **Load Firmware and Run**
   Now, you can load your firmware onto the microcontroller and run it:

```
(gdb) load
(gdb) continue
```

And that's it! Your firmware should now be running on the STM32 board. You can now use GDB commands to debug as needed.

(Aside) GDB Commands

Here's a quick guide to some of the basics, assuming you're using it for an embedded ARM application.

Starting GDB

After you've built your code and have an ELF file, you can open GDB with the following command:

```
arm-none-eabi-gdb main.elf
```

This will launch GDB and preload your `main.elf` file. Once GDB is running, you'll see a `(gdb)` prompt.

Connecting to OpenOCD

You already started OpenOCD in another terminal. In the GDB terminal, you can connect to OpenOCD using:

```
(gdb) target remote localhost:3333
```

This tells GDB to connect to OpenOCD, which is serving as the GDB server on port 3333.

Basic Commands

Here are some basic commands you'd likely use:

Load Firmware:

```
(gdb) load
```

This uploads your compiled code onto your microcontroller.

Run Code:

```
(gdb) continue
```

This will run the program.

Pause Code:

```
(gdb) interrupt
```

This halts the microcontroller and allows you to inspect the current state.

Step Into:

```
(gdb) step
```

This command allows you to step into the function calls.

Step Over:

```
(gdb) next
```

This steps over the function calls, effectively running them without entering into them.

Inspect Variables:

```
(gdb) print variable_name
```

Replace `variable_name` with the name of the variable you want to inspect.

Set Breakpoints:

```
(gdb) break function_name
```

This will pause execution when `function_name` is called.

Exit GDB:

To exit GDB, you can type:

```
(gdb) quit
```

---

12. C Standard Library Integration

Newlib

Newlib is a lightweight C standard library aimed at embedded systems. It is designed to be extensible and fully configurable to fit into various operating systems, including bare-metal environments. It's widely used because of its portability and modest resource requirements.

Newlib Nano

Newlib Nano is a smaller footprint version of Newlib, optimized to save space, making it more suitable for microcontrollers with limited flash and RAM. While it offers fewer features and capabilities than full-fledged Newlib, it's often sufficient for basic tasks such as string manipulation and basic I/O functions like `printf()`.

How to Use Newlib Nano

To use Newlib Nano, you need to specify it in your linker options. In your makefile, you can include `--specs=nano.specs` as part of the linker flags:

```
LDFLAGS = -mcpu=cortex-m4 -mthumb -mfloat-abi=soft --
specs=nano.specs -Tstm32_ls.ld -Wl,-Map=main.map
```

This will link your project against Newlib Nano instead of the standard Newlib.

System Calls

Newlib itself doesn't implement low-level system calls like `_write` or `_read`. These must be provided by the developer to interface with the underlying hardware or operating system.

Example with `printf`

To use functions like `printf`, you'll need to provide an implementation of `_write`. Typically, this is done in a file called `syscalls.c`.

Here's a simple example that redirects `_write` to a UART send function:

```
#include <errno.h>
#include <sys/unistd.h> // STDOUT_FILENO, STDERR_FILENO

int _write(int file, char *data, int len) {
    if ((file != STDOUT_FILENO) && (file != STDERR_FILENO)) {
        errno = EBADF;
        return -1;
    }

    // Your UART send function here, for example:
    // UART_send(data, len);

    return len;
}
```

You'll need to include this `syscalls.c` file in your project to resolve the low-level system calls used by Newlib or Newlib Nano.

Makefile Changes

Here's how you can include `syscalls.c` in your makefile:

```
# Define toolchain and flags
CC = arm-none-eabi-gcc
AS = arm-none-eabi-as
LD = arm-none-eabi-gcc
CFLAGS = -IInc -mcpu=cortex-m4 -mthumb -Wall -O0 -g -nostdlib
LDFLAGS = -mcpu=cortex-m4 -mthumb -mfloat-abi=soft --
specs=nano.specs -Tstm32_ls.ld -Wl,-Map=main.map

# Add syscalls.c to your source files
SRC = src/main.c src/syscalls.c # add your other source files here

# Compilation rule
all:
    $(CC) $(CFLAGS) $(SRC) $(LDFLAGS) -o your_project.elf
```

Section Merging of Standard Library

Add the following to `stm32_startup.c`:

```
void Reset_Handler(void)
{
    // 1. Copy .data section to SRAM
    uint32_t size = &_edata - &_sdata;

    uint8_t *pDst = (uint8_t *)&_sdata; // SRAM
    uint8_t *pSrc = (uint8_t *)&_la_data; // Flash

    for (uint32_t i = 0; i < size; i++)
    {
        *pDst++ = *pSrc++;
    }

    // 2. Initialize .bss section to zero in SRAM
    size = &__bss_end__ - &__bss_start__;
    pDst = (uint8_t *)&__bss_start__;
    for (uint32_t i = 0; i < size; i++)
    {
        *pDst++ = 0;
    }

    // 3. Initialize C Standard Library
    __libc_init_array();

    // 4. Call main()
    main();
}
```

Also adjust the linker script to prevent overlapping of `.data` and `.bss` sections:

```
ENTRY(Reset_Handler)

MEMORY
{
    FLASH (rx)  : ORIGIN = 0x08000000, LENGTH = 1024K
    SRAM  (rwx) : ORIGIN = 0x20000000, LENGTH = 128K
}

SECTIONS
{
    .text :
    {
        KEEP(*(.isr_vector))
        . = ALIGN(4);
        *(.text)
        *(.text*)
        *(.init)
        *(.fini)
        . = ALIGN(4);
        *(.rodata)
        *(.rodata*)
        _etext = .;
    } >FLASH

    _la_data = LOADADDR(.data);

    .data : AT (_etext)
    {
        . = ALIGN(4);
        _sdata = .;
        *(.data)
        *(.data*)
        . = ALIGN(4);
        _edata = .;
    } >SRAM

    .bss :
    {
        . = ALIGN(4);
        __bss_start__ = .;
        *(.bss)
        *(.bss*)
        . = ALIGN(4);
        __bss_end__ = .;
    } >SRAM
```

```
    _end = .;
    end = _end;
}
```