# BARE METAL STM32

## SETTING UP STARTUP, LINKERSCRIPT AND MAKEFILE

| Sections | Description |
| --- | --- |
| Introduction | Introduction to bare-metal register-level programming of S |
| Bitwise Operators | Basics of bitwise operators and compound operators |
| ARM-GNU-Toolchain | Explanation of GNU-Toolchain and on-chip debugging and |
| OpenOCD | Explanation debugging and flashing using OpenOCD |
| Linker Script | Step-by-step walk through of a basic linker script for a STM board |
| Startup | Step-by-step walk through of a basic startup file for a STM |
| Makefile | Step-by-step walk through of a basic Makefile for a STM32 |
| Blinky-Project | This section describes the register-level programming for a blink project |
| Demonstration of Compiling and Debugging | This section demonstrates how to compile and flash the STM32F401RE |
| Code and Documentation | Check out my code and documentation! |

Suppose you've ever wondered how to set up a bare metal register level C program for developing e systems without an IDE or libraries such as HAL or any other abstractions. In that case, this blog pos you! First, let's define what register-level programming is and why you would want to use this appro

Register-level programming is low-level programming where operations are carried out directly on th hardware registers of a processor. This is in contrast to the higher-level programming approach, whi by applying modifications using abstractions such as functions and objects. There are numerous rea you would want to program at the register level:

1. First, it can be very powerful, as it provides more precise control and flexibility over the hardware
2. Second, it can be used to create very efficient code, since you're not dealing with the overhead of level abstractions
3. Third, it's necessary for circumstances where operations must be performed that otherwise would difficult or even impossible at a higher level
4. And finally, IT CAN BE MORE FUN! :)

Still it's important to keep some disadvantages in mind using this approach:

1. First, it can be very difficult to debug register level code, as it is hard to read and follow the logic code,
2. Second, it can be dangerous, as it's easy to make mistakes and even damage the hardware

Though register level programming can be difficult as it requires proficient knowledge about the har can be very rewarding as you can get maximum out of your embedded system.

To get started some requirements are:
1. A good development board, ie. from ST, Texas Instruments or Arduino, etc. (The board used in this is STM32F401RE)
2. A compiler and an assembler (The compiler used in this example is arm-none-eabi-gcc source: Clic

These tools will allow us to write and compile code. Furthermore, register-level programming also re knowledge of how to use bitwise operations. Bitwise operators perform bit-level modifications to int values. Those operators are AND, OR, and XOR, which can be used to set, clear, toggle or invert indiv These operators can be very powerful, but also somewhat confusing. The following section provides overview of the most important applications of these operators.

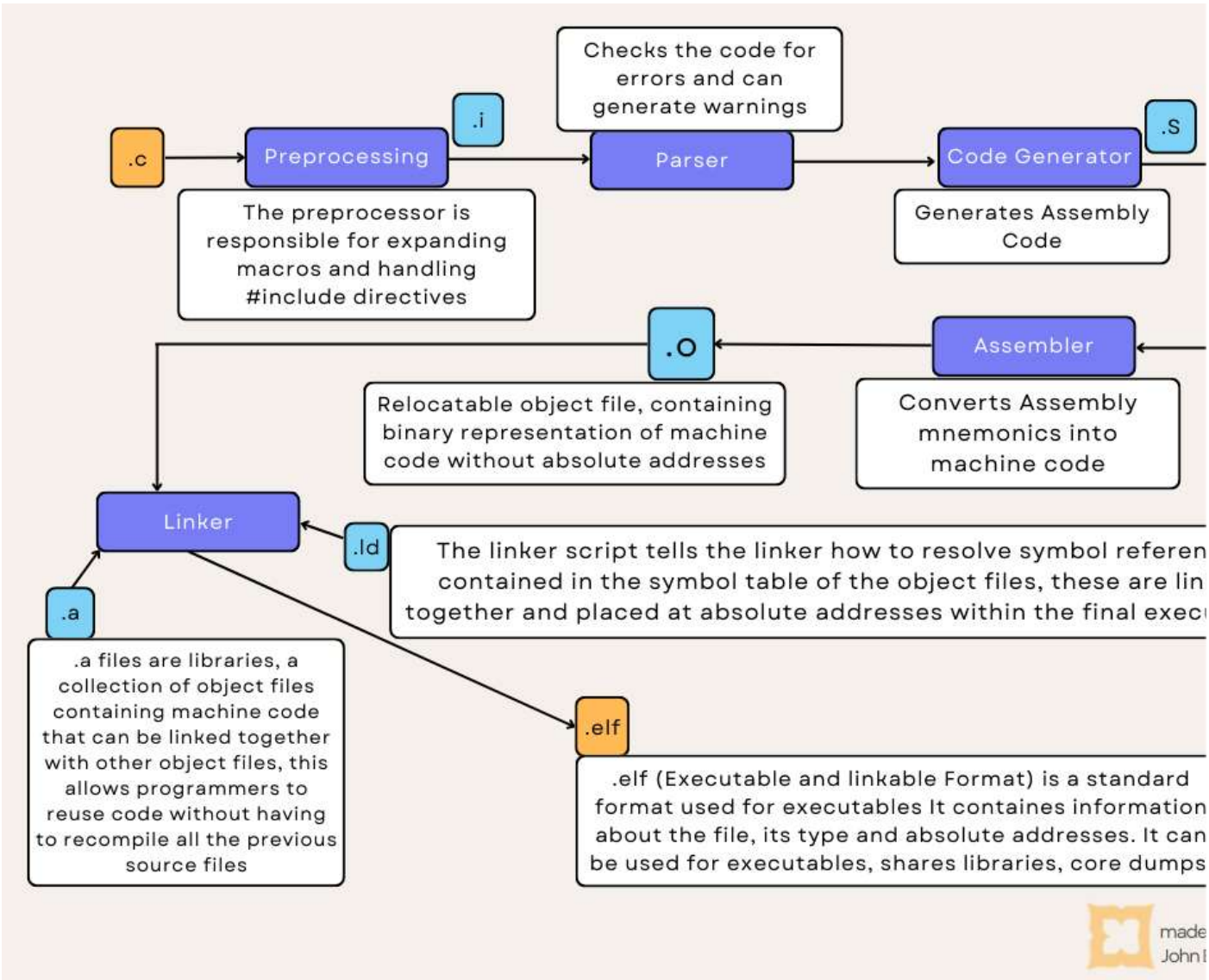
## BITWISE OPERATORS

| Operator | Examples |
|---|---|
| ANDing | 0x2F & 0xF0 = 0x20 // is used to clear individual bits |
| ORing | 0x50 \| 0x11 = 0x51 // is used to set individual bits |

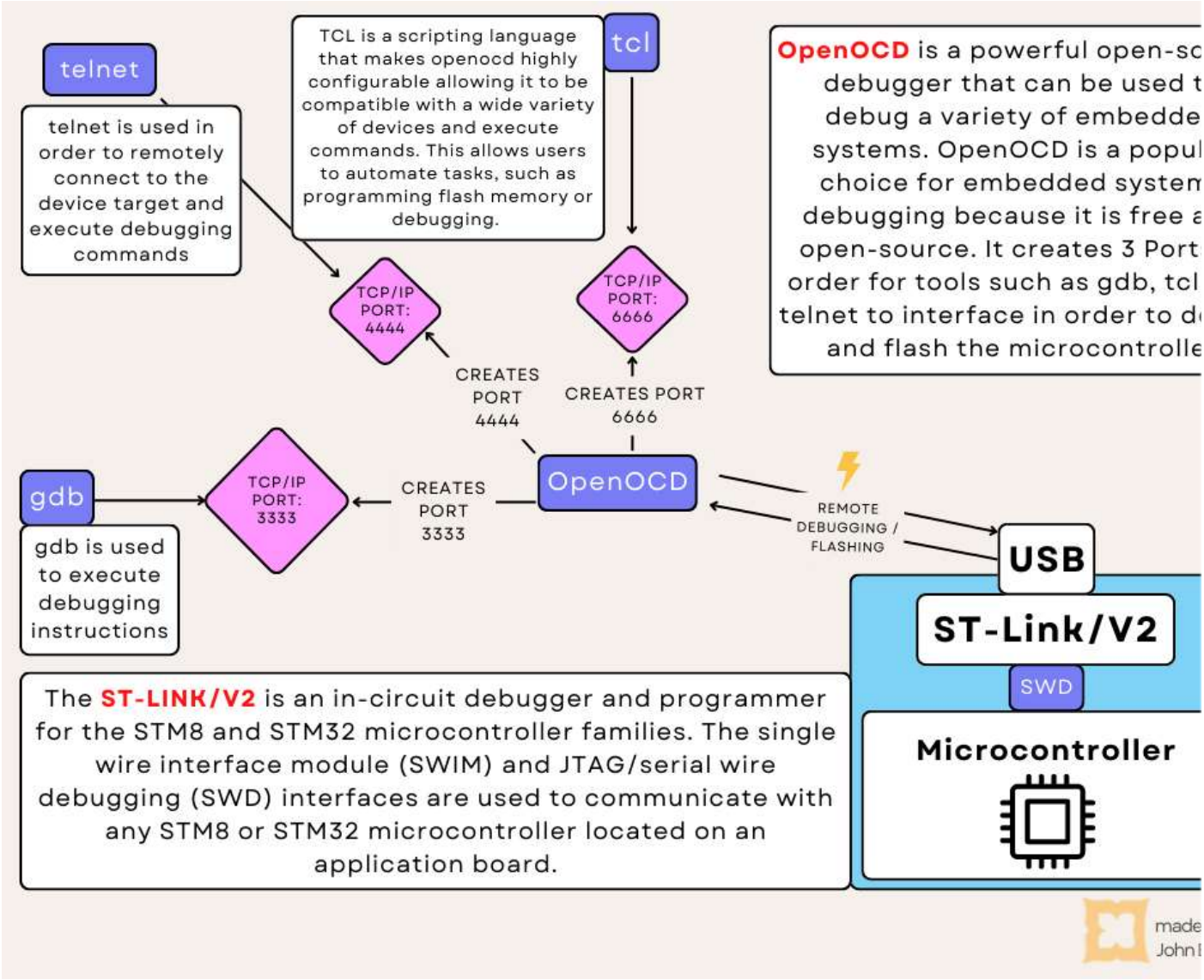
| | |
|---|---|
| XORing | 0x01 ^ 0x11 = 0x10 // is used to toggle individual bits |
| Inverting / NOT | ~0x55 = 0xAA // is used to invert individual bits |
| Shift Left | 0b0001 << 3 = 0b1000 // 1 has shifted 3 positions to the left |
| Shift Right | 0b1000 >> 3 = 0b0001 // 1 has shifted 3 positions to the write |

## ARM-GNU-TOOLCHAIN

The GNU Toolchain is a collection of programming tools produced by the GNU Project. It's used to tu code into executables.It is a powerful tool that is completely free and highly portable. It includes the Compiler Collection (GCC), the GNU Debugger (GDB), and further binaries and tools that are well-ma The ARM GNU Toolchain is a collection of these tools specialized for ARM - Chips. The flow diagram b illustrates how the ARM-GNU Toolchain compiles source files into a finished executable.



There are multiple ways for debugging and flashing the finished executable onto the chip, however, example, the free open-source on-chip debugger OpenOCD is used. The following flow diagram illust OpenOCD flashes and debugs the executable on the chip.




OpenOCD needs scripts for the target(ie. microcontroller) and interface(ie. debugger) because it use embedded debugger to communicate with the target device. The debugger needs to know how to int the target device in order to properly communicate with it. The scripts provide this information to th debugger. The following code will later be used in order to debug and flash the program onto the microcontroller.

```
$ openocd -f /usr/share/openocd/scripts/interface/stlink-v2.cfg -f /usr/share/openocd/scripts/target/stm32f4
```

## LINKER - SCRIPT

order to write a linker script the GNU linker script language is used, which is actually a subset of the programming language.

The linker itself is a program integrated in the GNU Compiler Collection (GCC), which utilizes the link determine how to map the contents of an executable file into memory. It controls the memory layou executable, including where in memory the executable's code and data are placed. Additionally, it co linking process itself. A Linker Script contains multiple sections, the memory section the define varic memories in the systems and their attributes. The attribute of a memory includes its size, its startin its type and its reading permissions. The actual documentation for those sections can be looked up i official GNU Documentation. A snippet of the documentation describing the memory section is provi following image.

## Memory Layout

The linker's default configuration permits allocation of all available memory. You can ove this configuration by using the MEMORY command. The MEMORY command describes the locati size of blocks of memory in the target. By using it carefully, you can describe which mem regions may be used by the linker, and which memory regions it must avoid. The linker d not shuffle sections to fit into the available regions, but does move the requested section the correct regions and issue errors when the regions become too full.

A command file may contain at most one use of the MEMORY command; however, you can de many blocks of memory within it as you wish. The syntax is:

```
MEMORY
  {
    name (attr) : ORIGIN = origin, LENGTH = len
    ...
  }
```

GNU Linker Documentation of Memory Section

The regions sections defines the various regions in the memories. A region is a contiguous block of r with a specific purpose. For example, a region may be created for the code, data or stack. This sectic linker how to link the various pieces of code and data together. A snippet of the documentation desc regions section is provided in the following image.

The SECTIONS command controls exactly where input sections are placed into output secti
their order in the output file, and to which output sections they are allocated.

You may use at most one SECTIONS command in a script file, but you can have as many
statements within it as you wish. Statements within the SECTIONS command can do one of
things:

- define the entry point;
- assign a value to a symbol;
- describe the placement of a named output section, and which input sections go int

GNU Linker Documentation of Region Section

**THE FOLLOWING CODE CONTAINS THE ACTUAL CODE OF THE LINKER SCRIPT USED IN THIS EXAMP**

```
1 ENTRY(Reset_handler)
2
3 /** Top of Stack **/
4 _estack = ORIGIN(SRAM) + LENGTH(SRAM);
5
6 /** Define Memory **/
7 MEMORY
8 {
9     SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 96
10    FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 51
11 }
```

| | The code above defines |
|---|---|
| [Line:1] | the first entry point of the microcontroller, the reset_handler which handles the main ha and software initializations |
| [Line:4] | the top address of the stack, this is required by the manufacturer |
| [Line:7] | as well as the actual memory section. This section defines that RAM memory has read \| v execute rights, with a starting address at 0x20000000 with a total size of 96KB. Additiona |

is defined to have read | execute rights, with a starting address at 0x08000000 and a tot
512KB. All these information is specific to the STM32F401RE microcontroller.

```
1  /** Define OUTPUT Sections **/
2  SECTIONS
3  {
4      .isr_vector :
5      {
6          KEEP(*(.isr_vector))
7      }> FLASH
8
9      .text :
10     {
11         . = ALIGN(4);
12         *(.text)
13         *(.text.*)
14         *(.rodata)
15         *(.rodata.*)
16         . = ALIGN(4);
17     }> FLASH
18
19     _sidata = LOADADDR(.data);
20
21     .data :
```

```
23              . = ALIGN(4);
24              _sdata = .;
25              *(.data)
26              *(.data.*)
27              . = ALIGN(4);
28              _edata = .;
29      }> SRAM AT> FLASH
30
31      .bss :
32      {
33              . = ALIGN(4);
34              _sbss = .;
35              *(.bss)
36              *(.bss.*)
37              *(COMMON)
38              . = ALIGN(4);
39              _ebss = .;
40      }> SRAM
41 }
```

| | |
|---|---|
| [Line:2] | This is the Region Section, which tells the linker which parts of the code should be linke together |
| [Line:4] | The Code Sections which are associated with the .isr_vector symbol will be linked togeth placed in the FLASH memory |
| [Line:9] | The Code Sections associated with .text and .rodata are linked together, meaning the ac and read only memory such as constants will be placed in the .text memory section in F |
| [Line:19] | This Line writes the starting address of the .data section into the symbol _sidata, this is used to copy the data section from FLASH to RAM |
| [Line:21] | All .data sections, meaning all initialized global and local static variables will be placed united .data section. Line 29 tells the linker that during loading .data will be placed in t and later in the SRAM memory during runtime. Additionally _sdata and edata are initiali denoting the staring and ending address of .data, which are are lated used in the reset to copy data from FLASH to RAM. |
| [Line:31] | All .bss sections, meaning all uninitialized global and local variables will be linked toget .bss section in SRAM. Additionally _sbss and ebss are initialized denoting the staring an address of .bss, which are are lated used in the reset handler to initialized previously uninitialized values with zero. |

## STARTUP FILE

The Startup File is the first piece of code that runs when an embedded system is turned on. It is resp for initializing the hardware and software of a system. The startup file is typically written in C or ass language. In this example, C language was used.

The two main parts of a startup file is

- the vector table and
- the startup code, defined in the Reset Handler

### THE VECTOR TABLE

The vector table is a table of pointers to the interrupt handler functions. In embedded systems, a ve is a data structure that contains a list of pointers to functions that the processor can execute. The v is typically located at the beginning of memory and is used by the processor to determine which fun execute when an interrupt occurs. When an interrupt occurs, the processor looks up the address of function to execute in the vector table, and then branches to that address. This allows the processor execute the correct function for the type of interrupt that occurred. The vector table is a critical par embedded system and must be carefully designed to ensure that the processor can correctly handle of interrupts that may occur.

The startup code is the code that initializes the hardware and software of the system.

One of the most common ways to reset an embedded system today is through the use of a reset han reset handler is a piece of code that is executed when the system is reset. This code can perform an necessary actions to ensure that the system is brought back to a known state. One common use for handler is to initialize all of the hardware and software components of the system. This ensures that everything is in a known state when the system starts up again. Reset handlers can also be used to p other tasks such as logging the reset event or sending a notification to a remote monitoring system. tasks of a Reset Handler implemented in this example are:

| Task | Description |
| --- | --- |
| Copy Data from flash to ram | All data sections meaning all initialized global and local static data will usually flashed onto the flash memory first. As data is part of the systems working mer has to be copied to the RAM to be able to apply read and write instructions to t memory. |
| Initialize the bss section with zeroes | The BSS section contains all uninitialized data. Therefore all its containing valu assigned with zeroes |
| Call main function | The actual main entry point of the program will be called |

**THE FOLLOWING CODE CONTAINS THE ACTUAL CODE OF THE LINKER SCRIPT USED IN THIS EXAMP**

```
1  /** global variables **/
2  extern uint32_t _estack;
3  extern uint32_t _sidata;
4  extern uint32_t _sdata;
5  extern uint32_t _edata;
6  extern uint32_t _sbss;
7  extern uint32_t _ebss;
8
9  /** Prototypes **/
10 extern int main(void);
11 void Reset_handler          (void);
12 void NMI_handler            (void)__attribute__((weak, alias("Default_handler
13 void HardFault_handler      (void)__attribute__((weak, alias("Default_handler
14 void MemManage_handler      (void)__attribute__((weak, alias("Default_handler
15 void BusFault_handler       (void)__attribute__((weak, alias("Default_handler
16 void UsuageFault_handler    (void)__attribute__((weak, alias("Default_handler
17 void SVCall_handler         (void)__attribute__((weak, alias("Default_handler
18 void DebugMonitor_handler   (void)__attribute__((weak, alias("Default_handler
19 void PendSV_handler         (void)__attribute__((weak, alias("Default_handler
20 void Systick_handler        (void)__attribute__((weak, alias("Default_handler
21 ...
22 ...
23 ...
```

| | Description of the code above: |
|---|---|
| [Line:2-7] | Here the symbol names taken from the linker script are initialized. These will later be u the reset handler. |
| [Line:10-20] | The prototypes used to call main as well as the interrupt handlers of the microcontroll initialized. These are documented in Reference Manual of the STM32F401RE. If the Inte Handlers are not initialized with a customized Handler, a default handler will be called |

```
1 /** Initialize Interrupt Vector **/
2 __attribute__ ((section(".isr_vector"))
3 void (* const fpn_vector[])(void) = {
4     (void (*)(void))(&_estack),
5     Reset_handler,
6     NMI_handler,
7     HardFault_handler,
8     MemManage_handler,
9     BusFault_handler,
10    UsuageFault_handler,
11    0,
12    0,
13    0,
14    0,
15    SVCall_handler,
16    DebugMonitor_handler,
17    0,
18    PendSV_handler,
19    Systick_handler,
20    ...
21    ...
22    ...
23 };
```

**Description of the code above:**

| | |
|---|---|
| [Line:2] | Here the array of function pointers are initialized using the section attribute. This will te compiler to place the array in the .isr_vector memory section in FLASH. |

```
1  void Reset_handler(void){
2
3      /** Copy Data from FLASH to SRAM **/
4      uint32_t * pSRC = (uint32_t *)&_sidata;
5      uint32_t * pDST = (uint32_t *)&_sdata;
6
7      for(uint32_t *dataptr = (uint32_t *)pDST; dataptr < &_edata;
8
9          *dataptr++ = *pSRC++;
10     }
11     /** Initialize BSS with ZEROES **/
12     for(uint32_t *bss_ptr = (uint32_t *)&_sbss; bss_ptr < &_ebss
13         *bss_ptr++ = 0;
14     }
15
16     /** CALL main() **/
17     main();
18 }
19
20 void Default_handler(void){
21
22     for(;;);
23 }
```

| | Description of the code above: |
|---|---|
| [Line:4-5] | Here pSRC is assigned the starting address of data and pDST is assigned the ending ad data. Notice that an ampersand symbol is used for the symbols. This is due to the fact symbols _sidata and _sdata are not usual variables but symbols created in the linker sc order access their value an Ampersand is needed. |
| [Line:7-10] | Each Value of .data in the RAM memory is overwritten with the values from .data in FL/ thereby copying .data from FLASH to RAM. |
| [Line:12-14] | All values of the bss section are assigned with zeroes. |
| [Line:10-23] | An endless loop, that will be entered ones an interrupt handler is called which has not initialized. This is a common method to set up a basic program without having to defin handler. |

## MAKEFILE

contains all of the code and data necessary to run a piece of hardware. In embedded systems, this b
often stored in flash memory. When you create a new project, you will typically start with a blank ma

**THE FOLLOWING CODE CONTAINS THE ACTUAL CODE OF THE MAKEFILE USED IN THIS EXAMPLE:**

**Upcoming!** A new blog post explaining Makefiles will be published soon, stay tuned!

```
1   # Binaries
2   CC = arm-none-eabi-gcc
3
4   # Directories
5   SRC_DIR = src
6   OBJ_DIR = obj
7   INC_DIR = inc
8   SUP_DIR = startup
9   DEB_DIR = debug
10
11  # Files
12  SRC := $(wildcard $(SRC_DIR)/*.c)
13  SRC += $(wildcard $(SUP_DIR)/*.c)
14  OBJ := $(patsubst $(SRC_DIR)/%.c, $(SRC_DIR)/$(OBJ_DIR)/%.o, $(SRC))
15  OBJ := $(patsubst $(SUP_DIR)/%.c, $(SRC_DIR)/$(OBJ_DIR)/%.o, $(OBJ))
16  LD := $(wildcard $(SUP_DIR)/*.ld)
```

| | Description of the code above: |
|---|---|
| [Line:2] | Here we assign the CC macro with the name of the arm cross compiler |
| [Line:5-9] | This macros are used to assign names to the directories used in this project. This is sp this example and can be configured to suit the project's needs. |
| [Line:12-13] | Using the wildcard feature of "make" we assign all the c file names to one single macr we can later use during compilation |
| [Line:14-15] | Here the values assigned to the SRC macros are substituted with a .o exten using the p feature. Furthermore, its file paths have been changed with that of the directory make object files. This also is specific to this project. |
| [Line:16] | Lastly the macro LD is used to hold the name of our linker script |

```
19  MARCH = cortex-m4
20  CFLAGS = -g -Wall -mcpu=$(MARCH) -mthumb -mfloat-abi=soft -I$(INC_DIR)
21  LFLAGS = -nostdlib -T $(LD) -Wl,-Map=$(DEB_DIR)/main.map
22
23  #PATHS
24  OPENOCD_INTERFACE = /usr/share/openocd/scripts/interface/stlink-v2.cfg
25  OPENOCD_TARGET = /usr/share/openocd/scripts/target/stm32f4x.cfg
```

| | Description of the code above: |
|---|---|
| [Line:19] | The MARCH macro is the processor used for this project. This will later be used during compilation |
| [Line:20] | These are the flags used during compilation. -g is used to generate debugging informat is an option used by the compiler to generate warnings. -mcpu and -mthumb tell the cc which cpu is used and that thumb instructions are used. -mfloat-abi=soft tells the comp floating functionalities will be enabled by a software implementation. -I tells the compi to look for include files, in this project this is used, as the Peripherals have been imple using structs which are placed in the include directory. |
| [Line:21] | These are the linker flags. -nostdlib means no standard library functions are used that be linked. -T tells the linker where to search for the linker script. -Wl, tells the linker th options are being passed. -Map is used to generate a map file of the executable which helpful for debugging purposes. |
| [Line:24-25] | These are the paths to the openOCD scripts used to tell openOCD how to connect and communicate with the on chip ST-Link V2 debugger, as well as the debugger unit on the STM32F401RE. |

```
28  TARGET = $(DEB_DIR)/main.elf
29
30  all: $(OBJ) $(TARGET)
31
32  $(SRC_DIR)/$(OBJ_DIR)/%.o : $(SRC_DIR)/%.c | mkobj
33          $(CC) $(CFLAGS) -c -o $@ $^
34
35  $(SRC_DIR)/$(OBJ_DIR)/%.o : $(SUP_DIR)/%.c | mkobj
36          $(CC) $(CFLAGS) -c -o $@ $^
37
38  $(TARGET) : $(OBJ) | mkdeb
39          $(CC) $(CFLAGS) $(LFLAGS) -o $@ $^
40
41  mkobj:
42          mkdir -p $(SRC_DIR)/$(OBJ_DIR)
43
44  mkdeb:
45          mkdir -p $(DEB_DIR)
46
```

| | Description of the code above: |
|---|---|
| [Line:28] | The TARGET is the macro used for our ELF executable |
| [Line:30] | In case "make" is executed by the default all files and targets specified after this keyw be generated |
| [Line:32-36] | These are makefile rules, telling make to generate object files using the c source files. Object file directory is not present, the "mkobj" target is called which will create a dire needed for compilation |
| [Line:38-39] | This makefile rule, tells make how to generate the target using object file, similarly to will call mkdeb if DEB_DIR is not available |

```
48          openocd -f $(OPENOCD_INTERFACE) -f $(OPENOCD_TARGET
49          gdb-multiarch $(TARGET) -x $(SUP_DIR)/flash.gdb
50
51  debug: FORCE
52          openocd -f $(OPENOCD_INTERFACE) -f $(OPENOCD_TARGET
53          gdb-multiarch $(TARGET) -x $(SUP_DIR)/debug.gdb
54
55  edit: FORCE
56          vim -S Session.vim
57
58  doxy: FORCE
59          cd ./docs && doxygen Doxyfile
60
61  clean: FORCE
62          rm -rf $(SRC_DIR)/$(OBJ_DIR) $(DEB_DIR)
63
64  FORCE:
65
66  .PHONY = mkobj mkdeb clean FORCE flash debug edit doxy
```

| | Description of the code above: |
|---|---|
| [Line:47-49] | this target will call openocd and subsequently gdb-multiarch to flash the elf file onto |
| [Line:51-53] | this target will also call openocd and gdb, however without flashing the elf |

## BLINKY - PROJECT

If you're working on register level programming for an embedded system, one thing you might want consider is using struct pointers that are assigned with fixed addresses. This can be helpful in a num ways. For one, it can make your code more readable. When you're working with a lot of registers, it difficult to keep track of what each one is used for. But if you give each register a meaningful name access it through a pointer, it can be much easier to understand what your code is doing. Another ac that it can make your code more efficient. If you know the address of a register, you can directly acc without having to go through any intermediate steps. This can be a significant speed boost, especial working with time-critical code.

A peripheral can be defined using a struct with each of its registers initialized as the struct's membe following code is the struct data structure of the STM32F401RE GPIO Peripheral.

```
typedef struct GPIOx_t{
    volatile uint32_t GPIOx_MODER;
    volatile uint32_t GPIOx_OTYPER;
    volatile uint32_t GPIOx_OSPEEDER;
    volatile uint32_t GPIOx_PUPDR;
    volatile uint32_t GPIOx_IDR;
    volatile uint32_t GPIOx_ODR;
    volatile uint32_t GPIOx_BSRR;
    volatile uint32_t GPIOx_LCKR;
    volatile uint32_t GPIOx_AFRL;
    volatile uint32_t GPIOx_AFRH;
}GPIOx_t;
```

Now that the registers have been initialized, an instance of that struct can be assigned a fixed addre the system's memory, defined in the reference manual. A possible method is to instantiate a struct p which is then assigned an address, as illustrated by the following code:

```
STRUCT_NAME * const defined_instance = (STRUCT_NAME *) ADDRESS;
```

To instantiate the GPIO Peripheral from the previously shown GPIOx_t using this method, we can no

```
GPIOx_t * const GPIOA = (GPIOx_t *)0x40020000; // This address is specific to STM32F401RE
```

is what we want for our purposes.

**THE FOLLOWING CODE CONTAINS THE ACTUAL CODE OF THE STARTUP FILE USED IN THIS EXAMPL**

```c
#define pin5  5
#define MODER 2

// Struct Pointers assigned with fixed addresses -> Loop up memory map of the reference m
RCC_t   * const RCC     = (RCC_t    *)  0x40023800;
GPIOx_t * const GPIOA   = (GPIOx_t  *)  0x40020000;

// Simple way to implement a sleep(ms) function
void wait_ms(int time){
    for(int i = 0; i < time; i++){
        // Loop for 1600 CLK Cycles, around 1ms
        //Number of Cycles is uC Specific, the more accurant way is to use timers
        for(int j = 0; j < 1600; j++);
    }
}

int main(void){

    //Enable CLOCK to GPIOA Peripheral
    //Common Clockr-Gating-Technique used by STM32 for power saving
    RCC->RCC_AHB1ENR |= 1;

    // Reset MODER Bitfield
    GPIOA->GPIOx_MODER &= ~(3 << (pin5 * MODER));
    // Write Value 1 to MODER Bitfield = OUTPUT
    GPIOA->GPIOx_MODER |=  (1 << (pin5 * MODER));

    for(;;){
        // Toggle 5th Bit or PIN5 of GPIOA
        GPIOA->GPIOx_ODR ^= (1 << pin5);
        wait_ms(100);
    }
}
```

| | Description of the code above: |
|---|---|
| [Line:1] | The macro pin5 is later used to configure and manipulate the Pin 5 of GPIOA |
| [Line:2] | The MODER Register is part of the GPIO Port which is later used to configure a pin eithe output or even assign an alternate function. As the width of MODER is a 2 Bit Bitfield, t is used for a more readable code |
| [Line:5-6] | Here struct pointers of the RCC Peripheral (used to handle Clock Enabling in this examp GPIOA Port is initialized and assigned with their associated fixed addresses in memory be looked up in the reference manual under the "memory map" section) |

| | |
|---|---|
| [Line:9-13] | This is a simple delay function implementation generating an approximately 1ms delay 1600 Clock Cycles with a 16MHz internal clock, which can generating the desired ms del looping the inner-for-loop by the amount of times certain ms are passed to the functio |
| [Line:21] | One of the great features of the STM32 is its clock gating feature, which allows periphe save power when they are not in use. In a nutshell, clock gating is a power-saving techr stops the clock signal to a particular circuit when that circuit is not in use. This prevent circuit from wasting power by doing unnecessary work. In order to enable that periphe need to enable it using the Reset Clock Control Peripheral of the microcontroller. Using AHB1ENR Register GPIOA is enabled by writing a 1 to that specific bit associated with th peripheral. |

## DEMONSTRATION OF COMPILING AND DEBUGGING

The following video demonstrates how to compile and debug using ARM GNU tools, OpenOCD and gc multiarch:


The following video demonstrates how to compile and debug using Make to automate all the previou tools


## CODE AND DOCUMENTATION

If you're interested in seeing how this project was made, please check out the source code and docu All the code is available for you to view, and the documentation goes into detail about how everythir Thanks for your interest!

If you have any questions about the project or any particular area that I haven't elaborated on prope free to message me on any of the social media platforms that I've shared on my website. I'll be happ answer any questions you may have.