Bare-metal Programming with STM32

Ashok Ashwin

December 14, 2025

Contents

I What is a Bare-metal Project? 2

1 Layers of Abstraction 2
1.1 Layer 1: Microcontroller Hardware - The Physical Foundation . . . 2
1.2 Layer 2: Register Level - The Digital Interface . . . 3
1.2.1 Volatile Semantics: All hardware registers are volatile - compilers cannot cache or reorder accesses . 3
1.2.2 Bitwise Operations: Hardware control through mask-and-set operations on specific bit positions . . 3
1.2.3 Read-Modify-Write Patterns: Essential for preserving unrelated bits in multi-function registers . . . 3
1.2.4 Wait-State Management: Polling status bits for peripheral readiness . . . 4
1.3 Layer 3: CMSIS - The Architectural Standardization . . . 5
1.3.1 Standardization Benefits . . . 5
1.3.2 The CMSIS Compromise . . . 5
1.4 Layer 4: HAL + Application - The Functional Abstraction . . . 6
1.5 Comparison of the layers . . . 6

2 The Anatomy of a Bare-Metal Embedded Project 7
2.1 The smallest possible startup file . . . 7
2.1.1 1. Assembler Directives (Architecture and Syntax) . . . 7
2.2 The smallest possible linker script . . . 9
2.3 Compiling the startup file with the linker . . . 10
2.4 Flashing and debugging with OpenOCD and GDB . . . 11

3 Adding C Sources and Makefiles 14
3.1 Preparing the Linker for C compilation . . . 14
3.2 Adding our C source and modifying the startup file . . . 14
3.2.1 Jumping to main() from the startup . . . 16
3.3 Adding a Makefile . . . 17
3.3.1 Build rules: . . . 18
3.4 Testing the Project . . . 18
3.4.1 Build the project with the makefile . . . 18
3.5 Final Output . . . 20

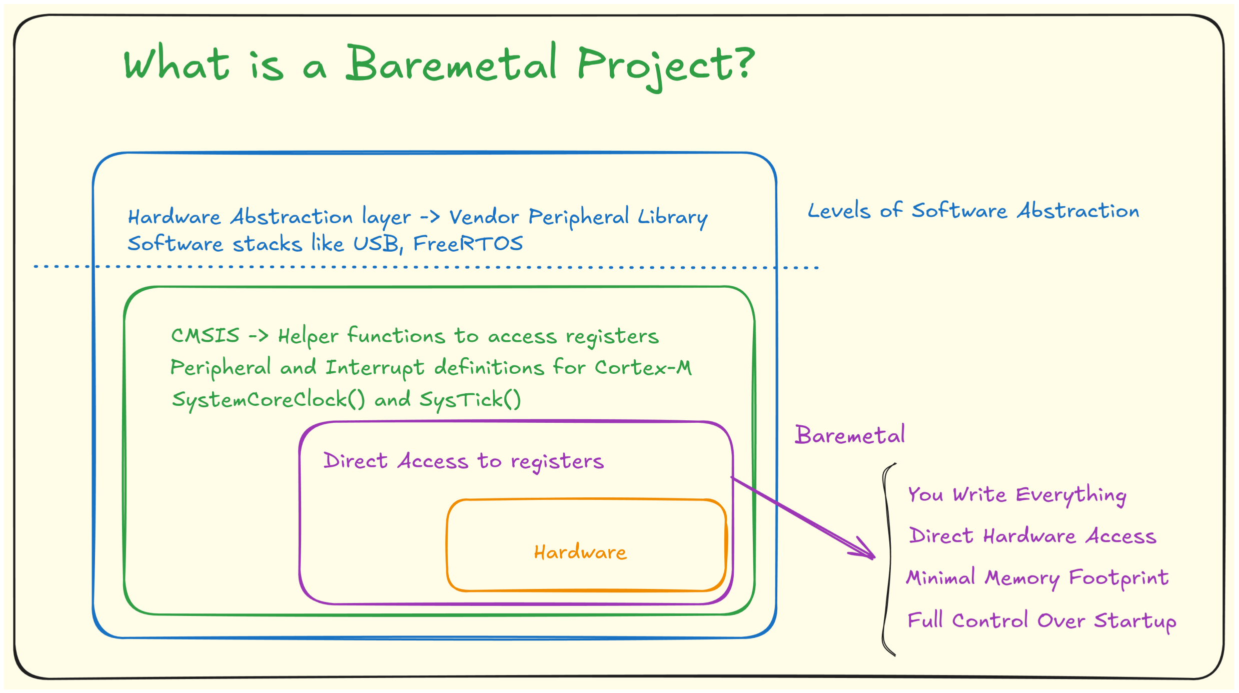

What is a Baremetal Project?
Hardware Abstraction layer -> Vendor Peripheral Library
Software stacks like USB, FreeRTOS
Levels of Software Abstraction
CMSIS -> Helper functions to access registers
Peripheral and Interrupt definitions for Cortex-M
SystemCoreClock() and SysTick()
Baremetal
Direct Access to registers
Hardware
You Write Everything
Direct Hardware Access
Minimal Memory Footprint
Full Control Over Startup

Figure 1: Levels of Embedded Software Abstraction

Part I

What is a Bare-metal Project?

The term "bare-metal" evokes a powerful image: raw, exposed hardware, stripped of any insulating layers, where software operates directly on the silicon substrate. This is programming at its most fundamental level, where every clock cycle matters, every memory byte is precious, and every electrical signal has meaning. In this world, there is no operating system to catch your errors, no memory protection to save you from yourself, no scheduler to manage your tasks. You are both the master and the slave of the machine.

In the realm of embedded systems development, abstraction represents a fundamental engineering compromise between absolute control and development efficiency. Each layer in the embedded software stack represents a deliberate trade-off, sacrificing varying degrees of hardware intimacy for increased productivity, portability, and maintainability.

Learning bare-metal programming is foundational to computer science education. Just as medical students dissect cadavers to understand anatomy, computer scientists should dissect computers to understand their operation. The insights gained from bare-metal programming inform higher-level work:

- Compiler writers understand optimization better
- OS developers understand scheduling and memory management better
- Application programmers understand performance characteristics better
- Security researchers understand attack vectors better

1 Layers of Abstraction

1.1 Layer 1: Microcontroller Hardware - The Physical Foundation

The hardware layer constitutes the physical silicon implementation of the microcontroller architecture. For the STM32F103C8T6 (Blue Pill), this includes:

- ARM Cortex-M3 Core: Harvard architecture with separate instruction and data buses (I-Code, D-Code, System bus)

- Memory Hierarchy: 64KB Flash (0x08000000-0x0800FFFF), 20KB SRAM (0x20000000-0x20004FFF)
- Peripheral Subsystems:
- NVIC (Nested Vectored Interrupt Controller): 60 mask-able interrupts, 16 priority levels
- SysTick Timer: 24-bit down-counter for OS timekeeping
- Memory Protection Unit (MPU): 8 regions for memory access control
- Bus Matrix: Advanced High-performance Bus (AHB) and Advanced Peripheral Bus (APB)
- Clock Tree: HSI (8MHz), HSE (up to 72MHz), PLL multiplication, multiple clock domains

1.2 Layer 2: Register Level - The Digital Interface

The register level provides the first meaningful abstraction: the memory-mapped register interface. Each hardware function exposes control through specific memory addresses.

This layer operates on several key principles:

1.2.1 Volatile Semantics: All hardware registers are volatile - compilers cannot cache or reorder accesses

Hardware registers are not ordinary memory.

In C, any pointer to a hardware register must be declared volatile.

This tells the compiler:

- Do not optimize away reads (the value may change spontaneously due to hardware).
- Do not combine or reorder writes (the hardware may require precise ordering).
- Do not keep values in CPU registers (every access must hit the actual hardware address).

Why is this necessary?

Because most peripheral registers behave in ways that normal memory does not:

- A status register bit may change as hardware events occur.
- Writing to a register may start a transmission or initiate a reset.
- Some registers clear on read.
- Some registers have write-only or read-only fields.

If the compiler cached such values or reordered accesses, the hardware would behave incorrectly.

1.2.2 Bitwise Operations: Hardware control through mask-and-set operations on specific bit positions

Software must manipulate specific bits or bitfields without affecting others.

The fundamental operations are:

- AND with a mask to clear bits
- OR with a mask to set bits
- XOR when toggling
- SHIFT when working with bitfields

1.2.3 Read-Modify-Write Patterns: Essential for preserving unrelated bits in multi-function registers

Necessary because registers pack multiple functions into one word.

Most CPU architectures do not support atomic bitfield updates to memory-mapped registers. Updating one field inside a shared register generally requires:

- READ the current register value
- MODIFY only the desired bits
- WRITE the new composite value back

1.2.4 Wait-State Management: Polling status bits for peripheral readiness

You cannot assume a peripheral is immediately ready after configuration.
Most on-chip peripherals have internal finite-state machines, clock domains, and transfer pipelines.
Thus the CPU must poll specific status bits, waiting for hardware to reach the expected state.
Why polling matters:

- Hardware may need several cycles or microseconds.
- Some peripherals need clock stabilization.
- Some operations trigger internal analog circuitry.
- Advanced designs may use interrupts or DMA to avoid polling, but the underlying mechanism is still a status register bit.

Examples across peripherals:

- ADC ⟶ wait for end-of-conversion
- Timers ⟶ wait for update event
- I2C ⟶ wait for start condition generated
- SPI ⟶ wait for TX FIFO empty
- Flash ⟶ wait for busy bit clear before next write

```
/*
Memory-mapped Peripheral Structure:
0x4002 1000 - 0x4002 13FF: Reset and Clock Control (RCC)
0x4002 1000: RCC_CR      (Clock control)
0x4002 1004: RCC_CFGR    (Clock configuration)
0x4002 1010: RCC_APB2ENR (Peripheral clock enable)
...
*/

Register Bit-level Programming Example:
// Enable GPIOA clock, configure PA5 as output, set pin high
#define RCC_APB2ENR   (*((volatile uint32_t *)0x40021018))
#define GPIOA_CRL     (*((volatile uint32_t *)0x40010800))
#define GPIOA_BSRR    (*((volatile uint32_t *)0x40010810))

void configure_led(void) {
    // Enable clock for GPIOA: set bit 2 in RCC_APB2ENR
    RCC_APB2ENR |= (1 << 2);

    // Configure PA5 as push-pull output, 2MHz: bits 20-23 in CRL
    GPIOA_CRL = (GPIOA_CRL & ~(0xF << 20)) | (0x2 << 20);

    // Set PA5 high: set bit 5 in BSRR
    GPIOA_BSRR = (1 << 5);
}
```

The register level provides maximum transparency - every hardware action corresponds directly to a software operation. This enables:

- Cycle-accurate timing prediction
- Minimal code size (no abstraction overhead)
- Complete understanding of hardware behavior
- Ability to work around errata or undocumented features
- However, this comes at the cost of extreme fragility: a single bit error can crash the system, and code is completely non-portable.

1.3 Layer 3: CMSIS - The Architectural Standardization

The Cortex Microcontroller Software Interface Standard (CMSIS) introduces architectural abstraction while maintaining hardware visibility. It provides:

1. CMSIS-CORE (Core peripheral access)
   (a) NVIC_EnableIRQ(), SysTick_Config()
   (b) Intrinsic functions: __DSB(), __ISB(), __WFE()
2. Device-Specific Headers
   (a) Structure definitions: GPIO_TypeDef, USART_TypeDef
   (b) Register bit definitions: GPIO_ODR_ODR5, USART_SR_TXE
   (c) Peripheral memory mapping: GPIOA = ((GPIO_TypeDef *)0x40010800U)
3. System Files
   (a) SystemInit(): Clock initialization
   (b) SystemCoreClock variable: CPU frequency in Hz
   (c) Startup files: Vector table, reset handler

```
a
// CMSIS-based GPIO control
#include "stm32f1xx.h"  // Device header includes core_cm3.h

void cmsis_gpio_example(void) {
    // 1. Enable peripheral clock (structured access)
    RCC->APB2ENR |= RCC_APB2ENR_IOPAEN;

    // 2. Configure pin using type-safe structures
    GPIOA->CRL &= ~GPIO_CRL_MODE5;  // Clear existing mode
    GPIOA->CRL |= GPIO_CRL_MODE5_0; // Output, max speed 10MHz
    GPIOA->CRL &= ~GPIO_CRL_CNF5;   // Clear configuration
    GPIOA->CRL |= GPIO_CRL_CNF5_0;  // General purpose push-pull

    // 3. Manipulate using named bit definitions
    GPIOA->ODR ^= GPIO_ODR_ODR5;  // Toggle PA5

    // 4. Use core functions for system operations
    NVIC_EnableIRQ(EXTI0_IRQn);  // Enable external interrupt 0
    __DSB();  // Data Synchronization Barrier
    __ISB();  // Instruction Synchronization Barrier
}
}
```

1.3.1 Standardization Benefits

- PortabilityWithin Cortex-M Family: Same NVIC API across M0, M3, M4, M7
- Compiler Independence: Works with GCC, IAR, Arm Compiler, LLVM
- Debugger Integration: Standardized peripheral views in debuggers
- Toolchain Compatibility: Consistent startup sequence across toolchains

1.3.2 The CMSIS Compromise

- CMSIS represents the minimal viable abstraction - it provides structure and naming without hiding functionality. Key characteristics:
- No Runtime Overhead: All CMSIS "functions" are either macros or inline functions
- Full Register Visibility: Structures provide access to every register bit

- Architectural Consistency: Same interrupt controller API across vendors
- Vendor Extensions: Vendors can add device-specific headers while maintaining compliance

1.4 Layer 4: HAL + Application - The Functional Abstraction

The Hardware Abstraction Layer (HAL) introduces functional abstraction, where operations are expressed as intent rather than register manipulation.

HAL embodies several key software engineering principles:

- Information Hiding: Application doesn't need to know register addresses
- Interface Standardization: Same GPIO API across all STM32 families
- Resource Management: Handles enable/disable sequences, clock gating
- Error Containment: Peripheral errors don't crash the system
- Power Management: Integrated low-power mode support

```c
// Inside HAL implementation (stm32f1xx_hal_gpio.c):
HAL_StatusTypeDef HAL_GPIO_Init(GPIO_TypeDef *GPIOx, GPIO_InitTypeDef *GPIO_Init) {
    uint32_t position;
    uint32_t ioposition = 0x00U;
    uint32_t iocurrent = 0x00U;
    uint32_t temp = 0x00U;

    // Parameter validation
    assert_param(IS_GPIO_ALL_INSTANCE(GPIOx));
    assert_param(IS_GPIO_PIN(GPIO_Init->Pin));
    assert_param(IS_GPIO_MODE(GPIO_Init->Mode));

    // Configure each pin
    for(position = 0U; position < GPIO_NUMBER; position++) {
        ioposition = (0x01U << position);
        iocurrent = (GPIO_Init->Pin) & ioposition;

        if(iocurrent == ioposition) {
            // Complex configuration logic (50+ lines)
            // Handles all GPIO modes: input, output, analog, alternate function
            // Manages pull-up/pull-down resistors
            // Sets output speed
            // Validates contradictory settings
        }
    }

    return HAL_OK;
}
```

1.5 Comparison of the layers

True expertise in embedded systems means understanding all four layers and knowing when to operate at each level. The most effective developers inhabit CMSIS for most work, drop to registers for optimization, leverage HAL for complex peripherals, and respect the hardware in all decisions.

The art of embedded development lies not in choosing one layer, but in mastering the transitions between them - knowing when to abstract for productivity and when to descend for control. This fluid movement across abstraction levels distinguishes the competent embedded engineer from the merely proficient programmer.

In the end, abstraction is not about hiding complexity, but about managing it - creating boundaries that contain complexity while providing controlled points of access for those who need to reach through to the layers beneath.

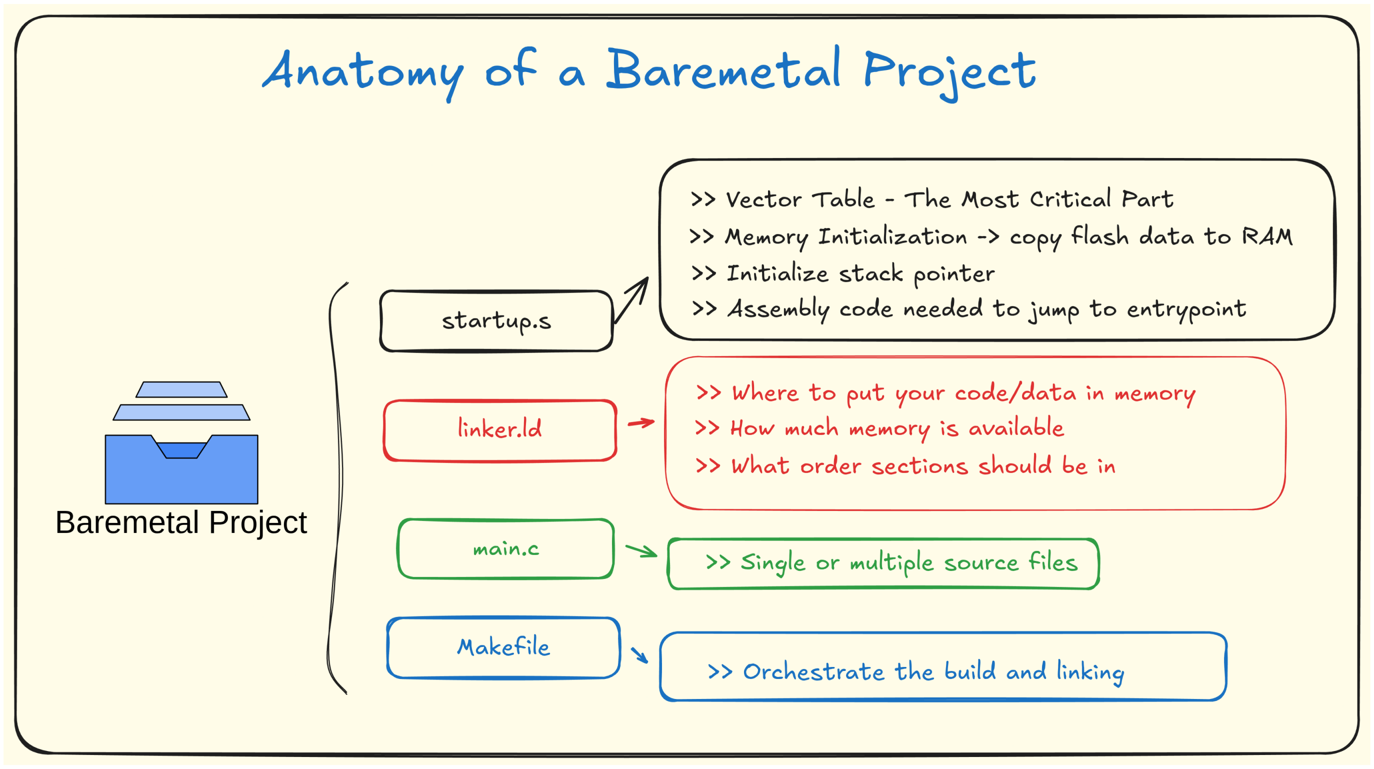

Anatomy of a Baremetal Project
Baremetal Project
startup.s
>> Vector Table - The Most Critical Part
>> Memory Initialization -> copy flash data to RAM
>> Initialize stack pointer
>> Assembly code needed to jump to entrypoint
linker.ld
>> Where to put your code/data in memory
>> How much memory is available
>> What order sections should be in
main.c
>> Single or multiple source files
Makefile
>> Orchestrate the build and linking

Figure 2: The four components of a Bare-Metal Project

2 The Anatomy of a Bare-Metal Embedded Project

A bare-metal embedded system represents the most fundamental approach to microcontroller programming, wherein the developer assumes complete responsibility for system initialization, resource management, and application execution without the intermediary of an operating system. This approach demands a meticulous understanding of four essential components that form the foundation of every bare-metal project. This section examines each component through the lens of both theoretical computer architecture and practical implementation.

2.1 The smallest possible startup file

The startup file is written in assembly, which allows for setting the stack pointer and reset behavior.

2.1.1 1. Assembler Directives (Architecture and Syntax)

```
.syntax unified
```

Enables the Unified Assembly Language (UAL) syntax used by ARM Cortex-M.

- UAL lets you write one consistent syntax for ARM and Thumb instructions.
- Allows movs, lsls, conditional suffixes, etc. required by modern ARM assemblers.

`.cpu cortex-m3`

Tells the assembler:
"Target CPU is ARM Cortex-M3."
This enables:

- Correct instruction set selection
- Correct error checking
- Use of Thumb-2 instructions, available to Cortex-M

`.fpu softvfp`

Indicates:
No hardware floating-point unit is present; use software emulation.

- Cortex-M3 has no FPU.
- This directive prevents the assembler from using floating-point instructions.

`.thumb`

- Force the assembler to emit Thumb / Thumb-2 instructions, not ARM mode.
- Cortex-M only supports Thumb mode, so this is always required.

`.global vtable.global reset_handler`

Makes the symbol vtable and reset_handler visible to the linker and other compilation units.

- The vector table (vtable) can reference it
- The linker knows that it must be kept
- reset_handler becomes the program's entry point

`.section .isr_vector`

The section is a block of memory, that store the vector table
This section is placed at address 0x0800 0000 (Flash start)
The CPU reads the first two words after reset:

- initial stack pointer
- reset handler address

`.align 2`

- Again aligns to 4 bytes or 2 words. All ARM Thumb functions should be 2-byte aligned minimum, but 4 bytes is safer.

`.word _estack`

The initial stack pointer loaded at reset.
When the CPU resets:

- Loads this word into the SP (stack pointer)
- Expects _estack to point to the top of RAM
- This symbol must be defined in the linker script (usually *_ld file).

`ldr r0, =_estack`

Loads the address of _estack into register r0.
This is a literal pool load:

- The assembler places _estack in immediate data near the code
- ldr r0, =value loads a 32-bit constant safely in Thumb mode
- This does not read memory at _estack;
- it loads the value of the symbol, which is the stack top address.

```
.syntax unified
.cpu cortex-m3
.fpu softvfp
.thumb

.global vtable
.global reset_handler

.section .isr_vector
.align 2
vtable:
    .word _estack
    .word reset_handler
.section .text
.align 2
reset_handler:
    ldr r0, =_estack          // Load stack top address
    mov sp, r0                // Initialize Stack Pointer sp
loop:
    mov r3, #0xabcd           // Load immediate test value
    b loop                    // Infinite loop
```

Listing 1: STM32 Startup Code

2.2 The smallest possible linker script

```
MEMORY
{
    FLASH ( rx )      : ORIGIN = 0x08000000, LENGTH = 64K
    RAM ( rxw )       : ORIGIN = 0x20000000, LENGTH = 20K
}
_estack = ORIGIN(RAM) + LENGTH(RAM);
```

The MEMORY { ... } block tells the linker:

- What memory regions exist in the MCU
- Where each region starts
- How big each region is
- What kinds of accesses are allowed (read, write, execute)

This is how the linker knows where to place:

- Code (.text)
- Initialized data (.data)
- Zero-initialized data (.bss)
- The stack
- The heap (if any)

```
_estack = ORIGIN(RAM) + LENGTH(RAM);
```

- This computes the address of the top of RAM, and assigns it to the symbol _estack.

Why the top of RAM?

- ARM Cortex-M uses a descending stack:
- The stack grows downward (toward lower memory addresses)
- The initial stack pointer points to the end of the RAM region

2.3 Compiling the startup file with the linker

```
arm-none-eabi-gcc -x assembler-with-cpp\
    -Og ./startup.s\
    -mthumb\
    -mcpu=cortex-m3 -Wall\
    -T ./linker.inv\
    --specs=nosys.specs\
    -nostdlib\
    -Wl,-Map=program.map\
    -o program.elf
```

This is a typical embedded bare-metal build command in the ARM GCC toolchain.

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

The cross-compiler for ARM bare-metal systems.

- arm → generates ARM/Thumb instructions
- none → no OS (bare-metal)
- eabi → Embedded ABI calling convention
- This compiler produces code suitable for MCUs like STM32, LPC, CH32, etc.

```
-x assembler-with-cpp
```

Treat the input file as: Assembly, but first run it through the C preprocessor (CPP).

```
-Og
```

- Optimize for debugging.
- Keeps code readable in a debugger
- Avoids aggressive optimizations
- Better than -O0 for embedded debugging
- Does not break register-level timing-sensitive code

```
-T ./linker.inv
```

- Use a custom linker script. Without this, GCC tries to use a default linker script, which is wrong for MCUs.

```
--specs=nosys.specs
```

- Do not use any system call stubs.
- No open, read, write, exit
- No semihosting by default
- Avoids linking unwanted POSIX layers
- Useful when writing pure bare-metal firmware.

```
-nostdlib
```

Do not link:

- libc
- libgcc
- crt0 startup files
- Bare-metal applications should not pull in standard libraries unless desired.

```
-Wl,-Map=program.map
```

- Pass the option -Map=a.map to the linker (ld).
- This produces a linker map file, which lists:
  1. memory layout
  2. symbol addresses
  3. section placements
  4. function sizes
- Extremely useful for debugging memory issues in embedded systems.

2.4 Flashing and debugging with OpenOCD and GDB

Once the above binary is built, we need to flash and debug it. Use openOCD to launch a gdb-server

```
openocd -f interface/stlink-v2.cfg -f target/stm32f1x.cfg
```

You may need to manually provide the path to the .cfg files, if you have manually downloaded openOCD. For the stm32f103 blue-pill, we should see the below output, if the wiring is proper and the debugger is connected.

```
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
WARNING: interface/stlink-v2.cfg is deprecated, please switch to interface/stlink.cfg
Info : auto-selecting first available session transport "hla_swd". To override use 'transport select ↩
    ↪<transport>'.
Info : The selected transport took over low-level target control. The results might differ compared ↩
    ↪to plain JTAG/SWD
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : clock speed 1000 kHz
Info : STLINK V2J46S7 (API v2) VID:PID 0483:3748
Info : Target voltage: 3.269519
Info : [stm32f1x.cpu] Cortex-M3 r1p1 processor detected
Info : [stm32f1x.cpu] target has 6 breakpoints, 4 watchpoints
Info : starting gdb server for stm32f1x.cpu on 3333
Info : Listening on port 3333 for gdb connections
[stm32f1x.cpu] halted due to breakpoint, current mode: Thread
xPSR: 0x21000000 pc: 0x08000048 msp: 0x20004ff8
```

This OpenOCD command starts a debug server for your STM32F103 microcontroller:

```
openocd
```

Path to the OpenOCD executable

```
-f interface/stlink-v2.cfg
```

- Loads the configuration file for ST-Link V2 debugger/programmer

- Defines how OpenOCD communicates with your debug probe hardware
- Sets up USB VID/PID, communication protocols, etc.

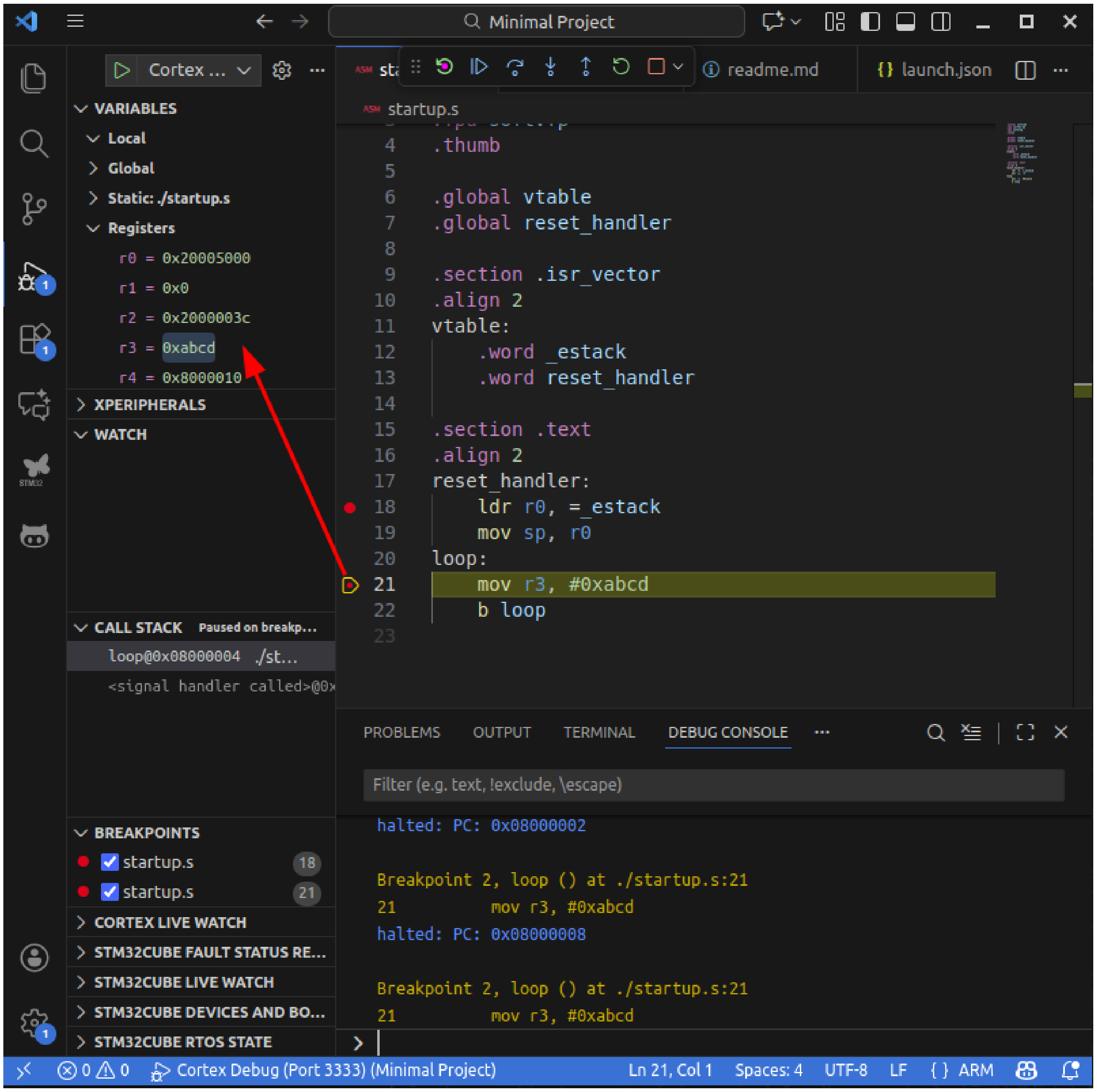
```
-f target/stm32f1x.cfg
```

- Loads the configuration file for STM32F1 family targets
- Defines target-specific settings: flash size, RAM, reset behavior
- Sets up the Cortex-M3 core configuration
- Configures memory regions and flash algorithms

What it does:

- Starts a GDB server on port 3333 (default)
  - Connects to your STM32F103 via ST-Link V2
  - Allows debugging tools (like GDB or VS Code) to connect and control the microcontroller
  - Enables flash programming, breakpoints, memory inspection, and single-stepping
  - Once running, your VS Code debugger connects to localhost:3333 as configured in your launch.json.
  - download the STM32F103xx.svd file for the register view, this should be done manually if not using CudeIDE.

```
# launch.json
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Cortex Debug (Port 3333)",
            "cwd": "${workspaceFolder}",
            "executable": "./program.elf",
            "request": "launch",
            "type": "cortex-debug",
            "servertype": "external",
            "gdbTarget": "localhost:3333",
            "gdbPath": "arm-none-eabi-gdb",
            "runToEntryPoint": "main",
            "device": "STM32F103C8",
            "svdFile": "./STM32F103xx.svd",
            "enableInstructionStepping": true
        }
    ]
}
```



```
r0 = 0x20005000
r1 = 0x0
r2 = 0x2000003c
r3 = 0xabcd
r4 = 0x8000010
```

```
.thumb

.global vtable
.global reset_handler

.section .isr_vector
.align 2
vtable:
    .word _estack
    .word reset_handler

.section .text
.align 2
reset_handler:
    ldr r0, =_estack
    mov sp, r0
loop:
    mov r3, #0xabcd
    b loop
```

```
halted: PC: 0x08000002

Breakpoint 2, loop () at ./startup.s:21
21          mov r3, #0xabcd
halted: PC: 0x08000008

Breakpoint 2, loop () at ./startup.s:21
21          mov r3, #0xabcd
```

Figure 3: Debugging with VSCode cortex-debug extension via launch.json

3 Adding C Sources and Makefiles

3.1 Preparing the Linker for C compilation

When you add main.c to your bare-metal project with startup.s, the linker needs these changes:

```
MEMORY
{
    FLASH ( rx )      : ORIGIN = 0x08000000, LENGTH = 64K
    RAM ( rxw )       : ORIGIN = 0x20000000, LENGTH = 20K
}

_estack = ORIGIN(RAM) + LENGTH(RAM);

SECTIONS
{
    .text :
    {
        *(.isr_vector)
        *(.text)
        *(.text*)
        *(.rodata)
        *(.rodata*)
    } > FLASH

    .data :
    {
        *(.data)
        *(.data*)
    } > RAM AT> FLASH

    .bss :
    {
        *(.bss)
        *(.bss*)
        *(COMMON)
    } > RAM
}
_estack = ORIGIN(RAM) + LENGTH(RAM);
```

1. .text section - For code from main.c
2. .rodata section - For constant data (if any)
3. .data section - For initialized global/static variables
4. .bss section - For uninitialized variablesWhy these are needed:

Without C code: Only .isr_vector needed for the vector table

- With C code: Compiler generates .text (functions), .data (initialized vars), .bss (uninitialized vars), and .rodata (constants)
- The linker must know where to place these sections in FLASH/RAM
- Your current linker.inv already has these sections defined, which is why main.c compiles and links successfully.

3.2 Adding our C source and modifying the startup file

Let us write a simple program to toggle PC13 on the blue-pill.

```c
#define RCC_APB2ENR   (*(volatile unsigned int *)0x40021018)
```

- Defines a macro for the RCC APB2 peripheral clock enable register
- 0x40021018 = memory address of this register

- (volatile unsigned int *) = cast address to pointer to volatile unsigned int
- * = dereference to access the register value
- volatile prevents compiler optimization (ensures actual memory reads/writes)

```c
#define GPIOC_CRH     (*(volatile unsigned int *)0x40011004)
#define GPIOC_ODR     (*(volatile unsigned int *)0x4001100C)
```

- GPIOC Configuration Register High (controls pins PC8-PC15)
  - 0x40011004 = register address
  - Used to configure pin modes (input/output, speed, type)
- GPIOC Output Data Register (controls output pin states)
  - 0x4001100C = register address
  - Writing to this register sets pins HIGH (1) or LOW (0)

```c
int main(void) {
```

- Entry point function for C code
- To be Called by reset_handler in startup.s
- void = takes no parameters

```c
    // Enable GPIOC clock
    RCC_APB2ENR |= (1 << 4);

    // Configure PC13 as output (50MHz, push-pull)
    GPIOC_CRH &= ~(0xF << 20);
    GPIOC_CRH |= (0x3 << 20);)
```

- Enable GPIOC clock in RCC
  - Bit 4 of RCC_APB2ENR controls GPIOC clock
  - `|=` sets bit 4 to 1 without affecting other bits
  - Without this, GPIOC registers won't work
- Clear configuration bits for PC13
  - PC13 uses bits 20-23 in CRH register (4 bits per pin)
  - `0xF << 20 = 0b1111` shifted to bits 20-23
  - ~ inverts to create mask, &= clears those bits
- Set PC13 as output, 50MHz, push-pull
  - 0b0011 << 20 sets bits 20-21
  - MODE[1:0] = 11 = 50MHz output speed
  - CNF[1:0] = 00 = push-pull outp/ut mode

```
1  /* main.c */
2  #define RCC_APB2ENR   (*(volatile unsigned int *)0x40021018)
3  #define GPIOC_CRH     (*(volatile unsigned int *)0x40011004)
4  #define GPIOC_ODR     (*(volatile unsigned int *)0x4001100C)
5
6  int main(void) {
7
8      // Enable GPIOC clock
9      RCC_APB2ENR |= (1 << 4);
10
11     // Configure PC13 as output (50MHz, push-pull)
12     GPIOC_CRH &= ~(0xF << 20);
13     GPIOC_CRH |= (0x3 << 20);
14
15     // Blink loop
16     while(1) {
17         GPIOC_ODR ^= (1 << 13);
18         for(volatile int i = 0; i < 100000; i++);
19     }
20
21     return 0;
22 }
```

In line 16 of the complete listing, Toggle PC13 (flip its state)

- ^= XOR assignment operator
- Bit 13 corresponds to pin PC13
- Changes 0⟶1 or 1⟶0

```
for(volatile int i = 0; i < 100000; i++);
```

This is an inline delay loop to waste CPU cycles, volatile is used to prevent compiler optimization

3.2.1 Jumping to main() from the startup

```
.syntax unified
.cpu cortex-m3
.fpu softvfp
.thumb

.global vtable
.global reset_handler

.section .isr_vector
.align 2
vtable:
    .word _estack
    .word reset_handler

.section .text
.align 2
reset_handler:
    ldr r0, =_estack
    mov sp, r0
    bl main
```

Listing 2: STM32 Startup Code

Branch with Link to main function

```
    bl main
```

Calls main() in C code
Link register (LR) stores return address (though we never return)

3.3 Adding a Makefile

When more sources are added to this project, compilation can become tedious. This is where GNU make program comes in handy.

```
# Toolchain
CC = arm-none-eabi-gcc
OBJCOPY = arm-none-eabi-objcopy
SIZE = arm-none-eabi-size
GDB = arm-none-eabi-gdb

# Project name
TARGET = program

# Source files
ASM_SOURCES = startup.s
C_SOURCES = main.c

# Object files
OBJECTS = $(ASM_SOURCES:.s=.o) $(C_SOURCES:.c=.o)

# Compiler flags
CFLAGS = -Og -g
CFLAGS += -mthumb -mcpu=cortex-m3
CFLAGS += -Wall
CFLAGS += --specs=nosys.specs -nostdlib

# Linker flags
LDFLAGS = -T linker.inv
LDFLAGS += -Wl,-Map=$(TARGET).map

# OpenOCD configuration
OPENOCD = openocd
OPENOCD_INTERFACE = interface/stlink-v2.cfg
OPENOCD_TARGET = target/stm32f1x.cfg

# Build rules
all: $(TARGET).elf $(TARGET).bin

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

%.o: %.s
    $(CC) $(CFLAGS) -c $< -o $@

$(TARGET).elf: $(OBJECTS)
    $(CC) $(CFLAGS) $(OBJECTS) $(LDFLAGS) -o $@
    $(SIZE) $@

$(TARGET).bin: $(TARGET).elf
    $(OBJCOPY) -O binary $< $@

# Flash firmware
flash: $(TARGET).elf
    $(OPENOCD) -f $(OPENOCD_INTERFACE) -f $(OPENOCD_TARGET) \
        -c "program $(TARGET).elf verify reset exit"

# Start OpenOCD server for debugging
openocd:
    $(OPENOCD) -f $(OPENOCD_INTERFACE) -f $(OPENOCD_TARGET)

# Clean build files
clean:
    rm -f $(TARGET).elf $(TARGET).bin $(TARGET).map $(OBJECTS)

# Debug with GDB
debug: $(TARGET).elf
    $(GDB) $(TARGET).elf -ex "target remote localhost:3333"

.PHONY: all flash openocd clean debug
```

To declare environmental variables in the Makefile use the below syntax.
Add the tools path to the makefile.

```
# Toolchain
CC = arm-none-eabi-gcc
OBJCOPY = arm-none-eabi-objcopy
SIZE = arm-none-eabi-size
GDB = arm-none-eabi-gdb
```

3.3.1 Build rules:

Pattern rule for C compilation:
%.o = any .o file
%.c = matching .c file
$< = input file (main.c)
$@ = output file (main.o)
-c = compile only, don't link

```
# Build rules
all: $(TARGET).elf $(TARGET).bin

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

%.o: %.s
    $(CC) $(CFLAGS) -c $< -o $@

$(TARGET).elf: $(OBJECTS)
    $(CC) $(CFLAGS) $(OBJECTS) $(LDFLAGS) -o $@
    $(SIZE) $@

$(TARGET).bin: $(TARGET).elf
    $(OBJCOPY) -O binary $< $@
```

3.4 Testing the Project

3.4.1 Build the project with the makefile

Create the below project stucture

- linker.inv
- main.c
- Makefile
- startup.s

Run"make" build to build the project

```
$ make
arm-none-eabi-gcc -Og -g -mthumb -mcpu=cortex-m3 -Wall --specs=nosys.specs -nostdlib -c startup.s -o ↩
    ↪startup.o
arm-none-eabi-gcc -Og -g -mthumb -mcpu=cortex-m3 -Wall --specs=nosys.specs -nostdlib -c main.c -o main.o
arm-none-eabi-gcc -Og -g -mthumb -mcpu=cortex-m3 -Wall --specs=nosys.specs -nostdlib startup.o main.o ↩
    ↪-T linker.inv -Wl,-Map=program.map -o program.elf
arm-none-eabi-size program.elf
   text    data     bss     dec     hex filename
     92       0       0      92      5c program.elf
arm-none-eabi-objcopy -O binary program.elf program.bin
```

Run "make flash" to flash the bluepill over OpenOCD

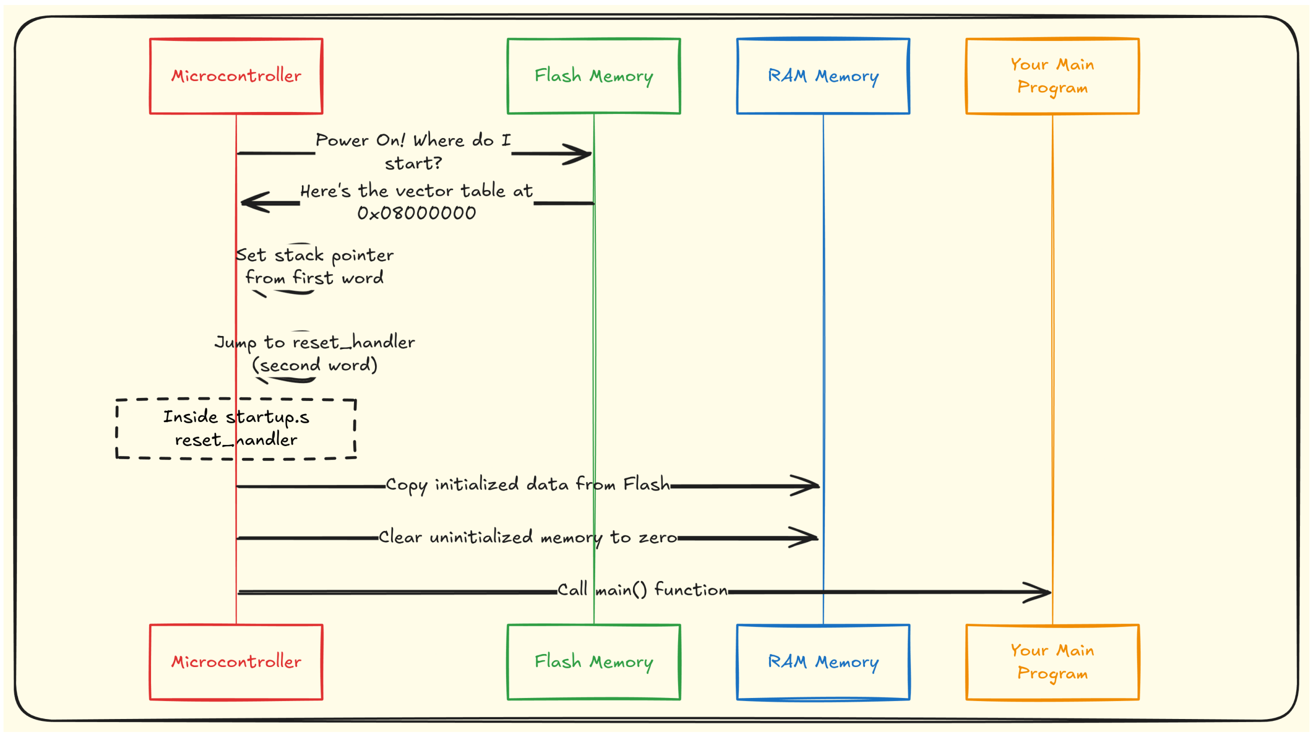
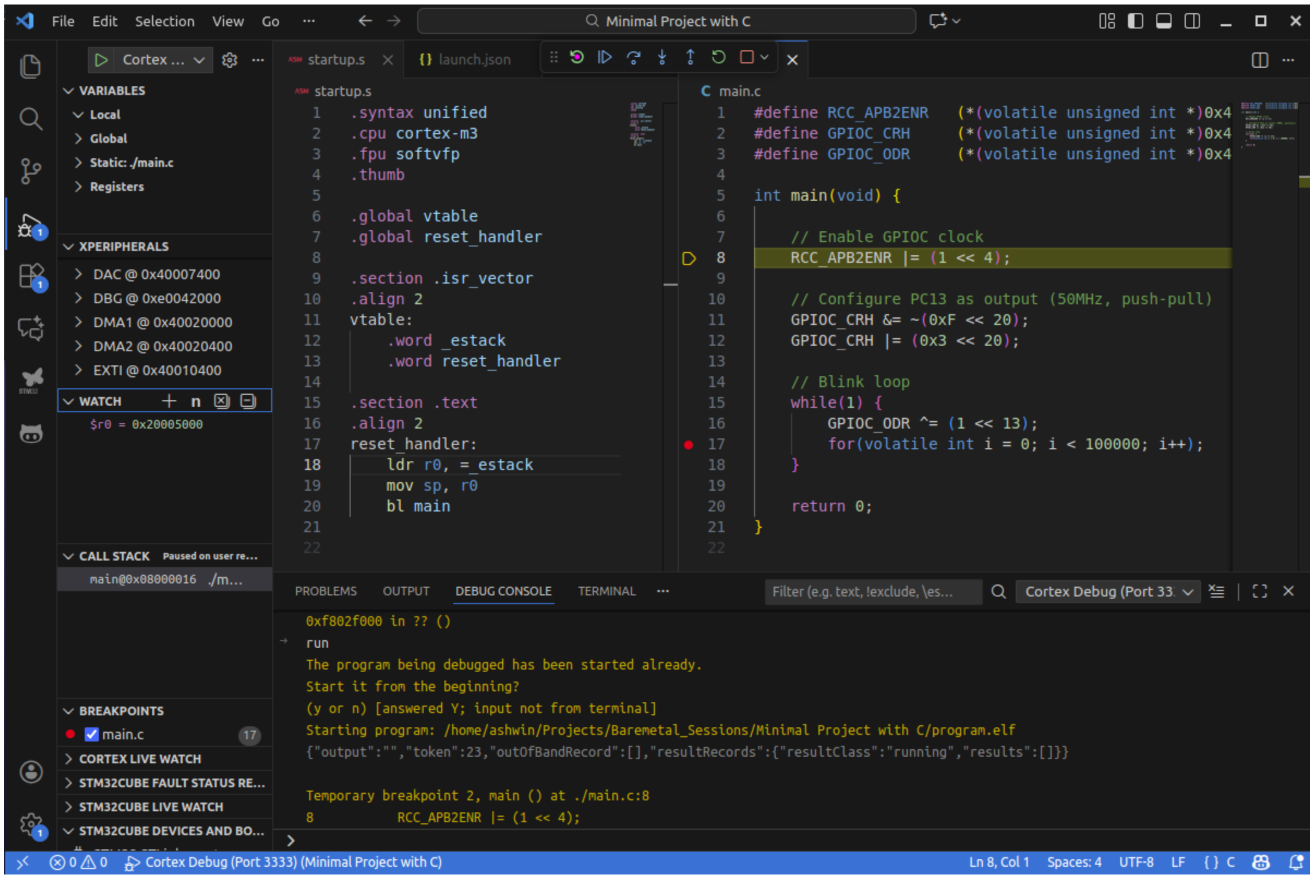

Microcontroller
Flash Memory
RAM Memory
Your Main Program
Power On! Where do I start?
Here's the vector table at 0x08000000
Set stack pointer from first word
Jump to reset_handler (second word)
Inside startup.s reset_handler
Copy initialized data from Flash
Clear uninitialized memory to zero
Call main() function
Microcontroller
Flash Memory
RAM Memory
Your Main Program

Figure 4: Baremetal Workflow

```
$ make flash
openocd -f interface/stlink-v2.cfg -f target/stm32f1x.cfg \
        -c "program program.elf verify reset exit"
Open On-Chip Debugger 0.12.0
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
WARNING: interface/stlink-v2.cfg is deprecated, please switch to interface/stlink.cfg
Info : auto-selecting first available session transport "hla_swd". To override use 'transport select ↩
    ↪<transport>'.
Info : The selected transport took over low-level target control. The results might differ compared ↩
    ↪to plain JTAG/SWD
Info : clock speed 1000 kHz
Info : STLINK V2J46S7 (API v2) VID:PID 0483:3748
Info : Target voltage: 3.269519
Info : [stm32f1x.cpu] Cortex-M3 r1p1 processor detected
Info : [stm32f1x.cpu] target has 6 breakpoints, 4 watchpoints
Info : starting gdb server for stm32f1x.cpu on 3333
Info : Listening on port 3333 for gdb connections
[stm32f1x.cpu] halted due to debug-request, current mode: Thread
xPSR: 00000000 pc: 0x08000008 msp: 0x20005000
** Programming Started **
Info : device id = 0x20036410
Info : flash size = 64 KiB
Warn : Adding extra erase range, 0x0800005c .. 0x080003ff
** Programming Finished **
** Verify Started **
** Verified OK **
** Resetting Target **
shutdown command invoked
```



```
.syntax unified
.cpu cortex-m3
.fpu softvfp
.thumb

.global vtable
.global reset_handler

.section .isr_vector
.align 2
vtable:
    .word _estack
    .word reset_handler

.section .text
.align 2
reset_handler:
    ldr r0, =_estack
    mov sp, r0
    bl main
```

```
#define RCC_APB2ENR   (*(volatile unsigned int *)0x4
#define GPIOC_CRH     (*(volatile unsigned int *)0x4
#define GPIOC_ODR     (*(volatile unsigned int *)0x4

int main(void) {

    // Enable GPIOC clock
    RCC_APB2ENR |= (1 << 4);

    // Configure PC13 as output (50MHz, push-pull)
    GPIOC_CRH &= ~(0xF << 20);
    GPIOC_CRH |= (0x3 << 20);

    // Blink loop
    while(1) {
        GPIOC_ODR ^= (1 << 13);
        for(volatile int i = 0; i < 100000; i++);
    }

    return 0;
}
```

```
$r0 = 0x20005000
main@0x08000016 ./m...
```

```
0xf802f000 in ?? ()
run
The program being debugged has been started already.
Start it from the beginning?
(y or n) [answered Y; input not from terminal]
Starting program: /home/ashwin/Projects/Baremetal_Sessions/Minimal Project with C/program.elf
{"output":"","token":23,"outOfBandRecord":[],"resultRecords":{"resultClass":"running","results":[]}}

Temporary breakpoint 2, main () at ./main.c:8
8           RCC_APB2ENR |= (1 << 4);
```

Figure 5: Bare-metal debugging with OpenOCD

3.5 Final Output