
EMBEDDED HACKING

Reverse Engineering the Raspberry Pi Pico 2

Week 1: Introduction and Overview of Embedded Reverse Engineering: Ethics, Scoping, and Basic Concepts

ARM Cortex-M33 • GDB • Ghidra

August 14, 2026

For educational and defensive security research only

LEGAL DISCLAIMER: The information, tools, and code provided in this repository and course are strictly for educational, research, and defensive purposes only.

You are explicitly prohibited from using any materials contained herein to access, test, modify, or exploit any device, network, or system that you do not own 100% or for which you do not have explicit, documented, and legally binding authorization to interact with.

By using this repository and course, you acknowledge and agree that:

1. Any illegal, unauthorized, or malicious use of this information is solely your responsibility.
2. The author(s) and contributor(s) of this repository and course shall not be held liable for any damages, legal repercussions, criminal charges, or unauthorized actions resulting from the use, misuse, or abuse of the contents herein.
3. You will comply with all applicable local, state, national, and international laws regarding cybersecurity and computer fraud.

IF YOU DO NOT AGREE WITH THESE TERMS, DO NOT USE THIS REPOSITORY AND COURSE.

What You'll Learn This Week

By the end of this week, you will be able to:

- Understand what a microcontroller is and how it works
 - Know the basic registers of the ARM Cortex-M33 processor
 - Understand memory layout (Flash vs RAM) and why it matters
 - Understand how the stack works in embedded systems
 - Set up and connect GDB to your Pico 2 for debugging
 - Use Ghidra for static analysis of your binary
 - Read basic ARM assembly instructions and understand what they do
-

Part 1: Understanding the Basics

What is a Microcontroller?

Think of a microcontroller as a tiny computer on a single chip. Just like your laptop has a processor, memory, and storage, a microcontroller has all of these packed into one small chip. The **RP2350** is the microcontroller chip that powers the **Raspberry Pi Pico 2**.

What is the ARM Cortex-M33?

The RP2350 has two “brains” inside it - we call these **cores**. One brain uses ARM Cortex-M33 instructions, and the other can use RISC-V instructions. In this course, we'll focus on the **ARM Cortex-M33** core because it's more commonly used in the industry.

What is Reverse Engineering?

Reverse engineering is like being a detective for code. Instead of writing code and compiling it, we take compiled code (the 1s and 0s that the computer actually runs) and figure out what it does. This is useful for:

- Understanding how things work
 - Finding bugs or security issues
 - Learning how software interacts with hardware
-

Part 2: Understanding Processor Registers

What is a Register?

A **register** is like a tiny, super-fast storage box inside the processor. The processor uses registers to hold numbers while it's doing calculations. Think of them like the short-term memory your brain uses when doing math in your head.

The ARM Cortex-M33 Registers

The ARM Cortex-M33 has several important registers:

Register	Also Called	Purpose
r0 - r12	General Purpose	Store numbers, pass data between functions
r13	SP (Stack Pointer)	Keeps track of where we are in the stack
r14	LR (Link Register)	Remembers where to go back after a function
r15	PC (Program Counter)	Points to the next instruction to run

General Purpose Registers (r0 - r12)

These 13 registers are your “scratch paper.” When the processor needs to add two numbers, subtract, or do any calculation, it uses these registers to hold the values.

Example: If you want to add 5 + 3:

1. Put 5 in r0
2. Put 3 in r1
3. Add them and store the result (8) in r2

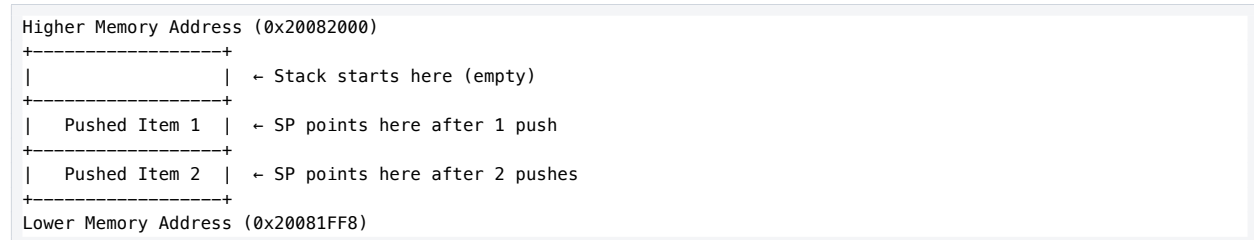
The Stack Pointer (r13 / SP)

The **stack** is a special area of memory that works like a stack of plates:

- When you add something, you put it on top (called a **PUSH**)
- When you remove something, you take it from the top (called a **POP**)

The Stack Pointer always points to the top of this stack. On ARM systems, the stack **grows downward** in memory. This means when you push something onto the stack, the address number gets smaller!

The two Arm ABI documents we verified give the formal proof for these rules. In AAPCS32, page 17 defines the core register roles used by the base procedure call standard: r13 is SP, r14 is LR, r15 is PC, r0-r3 are argument and scratch registers, and r4-r11 are the longer-lived variable registers. In Advisory Note 132, page 7 states that SP must be aligned to a multiple of 8 at every conforming call site and must already be 8-byte aligned when control first enters conforming code. That is why compiler-generated prologues often push an even number of registers, such as `push {r3, lr}`, to preserve both saved state and the required ABI stack alignment.



The Link Register (r14 / LR)

When you call a function, the processor needs to remember where to come back to. The Link Register stores this “return address.”

Example:

```

main() calls print_hello()
  ↓
LR = address right after the call in main()
  ↓
print_hello() runs
  ↓
print_hello() finishes, looks at LR
  ↓
Jumps back to main() at the address stored in LR

```

The Program Counter (r15 / PC)

The Program Counter always points to the **next instruction** the processor will execute. It's like your finger following along as you read a book - it always points to where you are.

Part 3: Understanding Memory Layout

XIP - Execute In Place

The RP2350 uses something called **XIP (Execute In Place)**. This means the processor can run code directly from the flash memory (where your program is stored) without copying it to RAM first.

Key Memory Address: 0x10000000

This is where your program code starts in flash memory. Remember this address - we'll use it a lot!

Memory Map Overview

```

+-----+
| Flash Memory (XIP) |
| Starts at: 0x10000000 |
| Contains: Your program code |
+-----+
| RAM |
| Starts at: 0x20000000 |
| Contains: Stack, Heap, Variables |
+-----+

```

Stack vs Heap

Stack	Heap
Automatic memory management	Manual memory management
Fast	Slower
Limited size	More flexible size
Used for function calls, local variables	Used for dynamic memory allocation
Grows downward	Grows upward

Part 3.5: Reviewing Our Hello World Code

Before we start debugging, let's understand the code we'll be working with. Here's our 0x0001_hello-world.c program:

```

#include <stdio.h>
#include "pico/stdlib.h"

```

```
int main(void) {
    stdio_init_all();

    while (true)
        printf("hello, world\r\n");
}
```

Breaking Down the Code

The Includes

```
#include <stdio.h>
#include "pico/stdlib.h"
```

- **<stdio.h>** - This is the standard input/output library. It gives us access to the `printf()` function that lets us print text.
- **"pico/stdlib.h"** - This is the Pico SDK's standard library. It provides essential functions for working with the Raspberry Pi Pico hardware.

The Main Function

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

Every C program starts running from the `main()` function. The `void` means it takes no arguments, and `int` means it returns an integer (though our program never actually returns).

Initializing Standard I/O

```
stdio_init_all();
```

This function initializes all the standard I/O (input/output) for the Pico. It sets up:

- **USB CDC** (so you can see output when connected to a computer via USB)
- **UART** (serial communication pins)

Without this line, `printf()` wouldn't have anywhere to send its output!

The Infinite Loop

```
while (true)
    printf("hello, world\r\n");
```

- **while (true)** - This creates an infinite loop. The program will keep running forever (or until you reset/power off the Pico).
- **printf("hello, world\r\n")** - This prints the text "hello, world" followed by a carriage return (`\r`) and newline (`\n`).

Tip: Why `\r\n` instead of just `\n`?

In embedded systems, we often use both carriage return (`\r`) and newline (`\n`) together. The `\r` moves the cursor back to the beginning of the line, and `\n` moves to the next line. This ensures proper display across different terminal programs.

What Happens When This Runs?

1. **Power on** - The Pico boots up and starts executing code from flash memory
2. **stdio_init_all()** - Sets up USB and/or UART for communication
3. **Infinite loop begins** - The program enters the `while(true)` loop

4. **Print forever** - “hello, world” is sent over and over as fast as possible

Why This Code is Perfect for Learning

This simple program is ideal for reverse engineering practice because:

- It has a clear, recognizable function call (`printf`)
- It has an infinite loop we can observe
- It’s small enough to understand completely
- It demonstrates real hardware interaction (USB/UART output)

When we debug this code, we’ll be able to see how the C code translates to ARM assembly instructions!

Compiling and Flashing to the Pico 2

Now that we understand the code, let’s get it running on our hardware:

Step 1: Compile the Code

In VS Code, look for the **Compile** button in the status bar at the bottom of the window. This is provided by the Raspberry Pi Pico extension. Click it to compile your project.

The extension will run CMake and build your code, creating a `.uf2` file that can be loaded onto the Pico 2.

Step 2: Put the Pico 2 in Flash Loading Mode

To flash new code to your Pico 2, you need to put it into **BOOTSEL mode**:

1. **Press and hold** the right-most button on your breadboard (the BOOTSEL button)
2. **While holding BOOTSEL**, press the white **Reset** button
3. **Release the Reset button** first
4. **Then release the BOOTSEL button**

When done correctly, your Pico 2 will appear as a USB mass storage device (like a flash drive) on your computer. This means it’s ready to receive new firmware!

Tip: You’ll see a drive called “RP2350” appear in your file explorer when the Pico 2 is in flash loading mode.

Step 3: Flash and Run

Back in VS Code, click the **Run** button in the status bar. The extension will:

1. Copy the compiled `.uf2` file to the Pico 2
2. The Pico 2 will automatically reboot and start running your code

Once flashed, your Pico 2 will immediately start executing the hello-world program, printing “hello, world” continuously when we open PuTTY!

Part 4: Dynamic Analysis with GDB

Prerequisites

Before we start, make sure you have:

1. A Raspberry Pi Pico 2 board
2. GDB (GNU Debugger) installed
3. OpenOCD or another debug probe connection
4. The sample “hello-world” binary loaded on your Pico 2

Connecting to Your Pico 2 with OpenOCD

Open a terminal and start OpenOCD:

```
openocd -s "$env:USERPROFILE\.pico-sdk\openocd\0.12.0+dev\scripts" -f interface/cmsis-dap.cfg -f target/rp2350.cfg -c
↳ "adapter speed 5000"
```

Connecting to Your Pico 2 with GDB

Open another terminal and start GDB with your binary:

```
arm-none-eabi-gdb build\0x0001_hello-world.elf
```

Connect to your target:

```
(gdb) target extended-remote :3333
(gdb) monitor reset halt
```

Basic GDB Commands: Your First Steps

Now that we're connected, let's learn three essential GDB commands that you'll use constantly in embedded reverse engineering.

Setting a Breakpoint with `b main`

A **breakpoint** tells the debugger to pause execution when it reaches a specific point. Let's set one at our main function:

```
(gdb) b main
Breakpoint 1 at 0x10000234: file ../0x0001_hello-world.c, line 5.
```

What this tells us:

- GDB found our main function
- It's located at address `0x10000234` in flash memory
- The source file and line number are shown (because we have debug symbols)

Now let's run to that breakpoint:

```
(gdb) c
Continuing.

Breakpoint 1, main () at ../0x0001_hello-world.c:5
5     stdio_init_all();
```

The program has stopped right at the beginning of main!

Disassembling with `disas`

The `disas` (disassemble) command shows us the assembly instructions for the current function:

```
(gdb) disas
Dump of assembler code for function main:
=> 0x10000234 <+0>:    push    {r3, lr}
0x10000236 <+2>:    bl      0x1000156c <stdio_init_all>
0x1000023a <+6>:    ldr     r0, [pc, #8]    @ (0x10000244 <main+16>)
0x1000023c <+8>:    bl      0x100015fc <__wrap_puts>
0x10000240 <+12>:   b.n     0x1000023a <main+6>
0x10000242 <+14>:   nop
0x10000244 <+16>:   adds   r4, r1, r7
0x10000246 <+18>:   asrs   r0, r0, #32
End of assembler dump.
```

Understanding the output:

- The `=>` arrow shows where we're currently stopped
- Each line shows: address <offset>: instruction operands

- We can see the calls to `stdio_init_all` and `__wrap_puts` (`printf` was optimized to `puts`)
- The `b.n 0x1000023a` at the end is our infinite loop - it jumps back to reload the string!

Viewing ELF Sections with `info files` and `maintenance info sections`

To see how the ELF is laid out in memory, use:

```
(gdb) info files
Symbols from "C:\Users\flare-vm\Desktop\Embedded-Hacking-main\0x0001_hello-world\build\0x0001_hello-world.elf".
Extended remote target using gdb-specific protocol:
`C:\Users\flare-vm\Desktop\Embedded-Hacking-main\0x0001_hello-world\build\0x0001_hello-world.elf', file type
↳ elf32-littlearm.
Entry point: 0x1000014c
0x10000000 - 0x100019cc is .text
0x100019cc - 0x10001b18 is .rodata
0x10001b18 - 0x10001b20 is .ARM.exidx
0x10001b20 - 0x10001b4c is .binary_info
0x20000000 - 0x20000110 is .ram_vector_table
0x20000110 - 0x200002ac is .data
0x200002ac - 0x200002ac is .tdata
0x200002ac - 0x200002ac is .tbss
0x200002b0 - 0x200004d8 is .bss
0x200004d8 - 0x20000cd8 is .heap
0x20081000 - 0x20081800 is .stack_dummy
0x10001ce8 - 0x10001cfc is .flash_end
While running this, GDB does not access memory from...
Local exec file:
`C:\Users\flare-vm\Desktop\Embedded-Hacking-main\0x0001_hello-world\build\0x0001_hello-world.elf', file type
↳ elf32-littlearm.
Entry point: 0x1000014c
0x10000000 - 0x100019cc is .text
0x100019cc - 0x10001b18 is .rodata
0x10001b18 - 0x10001b20 is .ARM.exidx
0x10001b20 - 0x10001b4c is .binary_info
0x20000000 - 0x20000110 is .ram_vector_table
0x20000110 - 0x200002ac is .data
0x200002ac - 0x200002ac is .tdata
0x200002ac - 0x200002ac is .tbss
0x200002b0 - 0x200004d8 is .bss
0x200004d8 - 0x20000cd8 is .heap
0x20081000 - 0x20081800 is .stack_dummy
0x10001ce8 - 0x10001cfc is .flash_end
(gdb) maintenance info sections
Exec file: `C:\Users\flare-vm\Desktop\Embedded-Hacking-main\0x0001_hello-world\build\0x0001_hello-world.elf', file
↳ type elf32-littlearm.
[0] 0x10000000->0x100019cc at 0x00001000: .text ALLOC LOAD READONLY CODE HAS_CONTENTS
[1] 0x100019cc->0x10001b18 at 0x000029cc: .rodata ALLOC LOAD READONLY DATA HAS_CONTENTS
[2] 0x10001b18->0x10001b20 at 0x00002b18: .ARM.exidx ALLOC LOAD READONLY DATA HAS_CONTENTS
[3] 0x10001b20->0x10001b4c at 0x00002b20: .binary_info ALLOC LOAD READONLY DATA HAS_CONTENTS
[4] 0x20000000->0x20000110 at 0x00004000: .ram_vector_table ALLOC
[5] 0x20000110->0x20000110 at 0x00003cfc: .uninitialized_data HAS_CONTENTS
[6] 0x20000110->0x200002ac at 0x00003110: .data ALLOC LOAD READONLY CODE HAS_CONTENTS
[7] 0x200002ac->0x200002ac at 0x00003cfc: .tdata ALLOC LOAD DATA HAS_CONTENTS
[8] 0x200002ac->0x200002ac at 0x00000000: .tbss ALLOC
[9] 0x200002b0->0x200004d8 at 0x000042b0: .bss ALLOC
[10] 0x200004d8->0x20000cd8 at 0x000044d8: .heap ALLOC READONLY
[11] 0x20080000->0x20080000 at 0x00003cfc: .scratch_x HAS_CONTENTS
[12] 0x20081000->0x20081000 at 0x00003cfc: .scratch_y HAS_CONTENTS
[13] 0x20081000->0x20081800 at 0x00004000: .stack_dummy ALLOC READONLY
[14] 0x10001ce8->0x10001cfc at 0x00003ce8: .flash_end ALLOC LOAD READONLY DATA HAS_CONTENTS
[15] 0x0000->0x0034 at 0x00003cfc: .ARM.attributes READONLY HAS_CONTENTS
[16] 0x0000->0x0045 at 0x00003d30: .comment READONLY HAS_CONTENTS
[17] 0x0000->0x2069a at 0x00003d75: .debug_info READONLY HAS_CONTENTS
[18] 0x0000->0x54ff at 0x0002440f: .debug_abbrev READONLY HAS_CONTENTS
[19] 0x0000->0x0af0 at 0x00029910: .debug_aranges READONLY HAS_CONTENTS
[20] 0x0000->0x2f86 at 0x0002a400: .debug_rnglists READONLY HAS_CONTENTS
[21] 0x0000->0x15526 at 0x0002d386: .debug_line READONLY HAS_CONTENTS
[22] 0x0000->0x56a7 at 0x000428ac: .debug_str READONLY HAS_CONTENTS
[23] 0x0000->0x1ed4 at 0x00047f54: .debug_frame READONLY HAS_CONTENTS
```

```
[24] 0x0000->0xffd1 at 0x00049e28: .debug_loclists READONLY HAS_CONTENTS
[25] 0x0000->0x0178 at 0x00059df9: .debug_line_str READONLY HAS_CONTENTS
```

What each section means:

Section	Purpose
.text	Executable machine code (your functions/instructions).
.rodata	Read-only constants (strings like "hello, world", lookup tables, const data).
.ARM.exidx	ARM exception unwind index used for stack unwinding/backtraces.
.binary_info	Pico metadata used by tools (program identity/build information).
.ram_vector_table	Interrupt vector table copied/placed in RAM for runtime use.
.uninitialized_data	Deliberately non-zeroed RAM region that can survive certain reset paths.
.data	Initialized global/static variables in RAM (initial values come from flash).
.tdata	Initialized thread-local storage data (often empty in simple bare-metal apps).
.tbss	Zero-initialized thread-local storage (often empty).
.bss	Zero-initialized global/static variables in RAM.
.heap	Heap allocation region (malloc/new) reserved in RAM.
.scratch_x	RP2 scratch RAM bank X section (core-local/low-contention placement).
.scratch_y	RP2 scratch RAM bank Y section (core-local/low-contention placement).
.stack_dummy	Linker-reserved stack range marker used to size/place the stack.
.flash_end	Marker/metadata near the logical end of flash image region.
.ARM.attributes	ARM build attributes (ABI/architecture metadata for tools/linkers).
.comment	Compiler/build comment strings (toolchain identification).
.debug_info	Main DWARF debug database (types, variables, symbols, scopes).
.debug_abbrev	Abbreviation table referenced by .debug_info.
.debug_aranges	Address-to-compilation-unit lookup acceleration data.
.debug_rnglists	DWARF range lists for non-contiguous code/data ranges.
.debug_line	Address-to-source-line mapping used for stepping/breakpoints.

Section	Purpose
<code>.debug_str</code>	Shared string pool used by DWARF debug sections.
<code>.debug_frame</code>	Call frame information used for unwinding stack frames.
<code>.debug_loclists</code>	Variable location lists (where variables live over PC ranges).
<code>.debug_line_str</code>	Extra string pool used by <code>.debug_line</code> data.

Tip: Practical rule: For reverse engineering runtime behavior, focus first on `.text`, `.rodata`, `.data`, `.bss`, heap/stack regions, and the vector table. Debug sections are for source-level mapping and symbol intelligence.

Fast interpretation checklist (use this every time):

1. **Find where code executes:** Verify `.text` starts at `0x10000000` (XIP flash) and note its end.
2. **Find immutable constants:** Use `.rodata` for strings/tables; cross-reference these addresses in disassembly.
3. **Find initialized RAM state:** `.data` lives in RAM at runtime, but its initial bytes come from flash.
4. **Find zeroed runtime state:** `.bss` is RAM that startup code clears to zero before main.
5. **Find interrupt control point:** Confirm `.ram_vector_table` location for exception/IRQ handler mapping.
6. **Bound dynamic memory:** Note `.heap` range so you can classify allocator activity vs static data.
7. **Bound call-stack activity:** Use `.stack_dummy` as linker stack reservation, then track live stack with `$sp`.
8. **Separate runtime vs debug-only sections:** `.debug_*`, `.comment`, and `.ARM.attributes` help tooling, not execution.
9. **Correlate any suspicious address quickly:** Flash/XIP (`0x100...`) usually code/const; SRAM (`0x200...`) usually data/stack/heap.
10. **Validate in memory:** After identifying a section, inspect it with `x` in GDB to confirm actual bytes/instructions.

Viewing Registers with `i r`

The `i r` (info registers) command shows the current state of all CPU registers:

```
(gdb) i r
r0          0x0          0
r1          0x10000235    268436021
r2          0x80808080    -2139062144
r3          0xe000ed08    -536810232
r4          0x100001d0    268435920
r5          0x88526891    -2007865199
r6          0x4f54710     83183376
r7          0x400e0014    1074659348
r8          0x43280035    1126694965
r9          0x0          0
r10         0x10000000    268435456
r11         0x62707361    1651536737
r12         0xed07f600    -318245376
sp          0x20082000    0x20082000
lr          0x1000018f    268435855
pc          0x10000234    0x10000234 <main>
xpsr       0x69000000    1761607680
```

Key registers to watch:

Register	Value	Meaning
pc	0x10000234	Program Counter - we're at the start of main
sp	0x20081fc8	Stack Pointer - top of our stack in RAM
lr	0x100002d5	Link Register - where we return after main
r0-r3	Various	Will hold function arguments and return values

Tip: You can also use `i r pc sp lr` to show only specific registers you care about.

Quick Reference: Essential GDB Commands

Command	Short Form	What It Does
break main	b main	Set a breakpoint at main
continue	c	Continue execution until breakpoint
disassemble	disas	Show assembly for current function
info registers	i r	Show all register values
stepi	si	Execute one assembly instruction
nexti	ni	Execute one instruction (skip calls)
x/10i \$pc		Examine 10 instructions at PC
monitor reset halt		Reset the target and halt

Watching the Stack Change After push {r3, lr}

The first instruction in main is `push {r3, lr}`. Before we step it, SP is `0x20082000`. After a single `si`, SP becomes `0x20081ff8`, which tells us the processor reserved 8 bytes on the stack for two 32-bit values. The first word at the new top of stack is `0xe000ed08`, which is the old value of r3, and the second word is `0x1000018f`, which is the saved lr return address. This matches the ABI rule we discussed earlier: the compiler pushes an even number of registers so the stack stays 8-byte aligned at the next call site before `stdio_init_all()` runs.

Notice the difference between inspecting memory at `$sp` and inspecting `$lr`. `x/4x $sp` is enough here to show the relevant stack words in RAM, while `x/x $lr` shows the instruction word stored at the flash address held in the link register. In other words, `$sp` points to saved data on the stack, but `$lr` points to code that execution will return to later.

```
(gdb) x/x $sp
0x20082000:  0x00000000
(gdb) si
0x10000236    5      stdio_init_all();

(gdb) x/x $sp
0x20081ff8:  0xe000ed08
(gdb) x/4x $sp
0x20081ff8:  0xe000ed08    0x1000018f    0x00000000    0x00000000
(gdb) x/x $lr
0x1000018f <platform_entry+8>: 0x00478849
```

Tip: What's Next? In Week 2, we'll put these GDB commands to work with hands-on debugging exercises! We'll step through code, examine the stack, watch registers change, and ultimately use these skills to modify a running program. The commands you learned here are the foundation for everything that follows.

Part 5: Static Analysis with Ghidra

Setting Up Your First Ghidra Project

Before we dive into GDB debugging, let's set up Ghidra to analyze our hello-world binary. Ghidra is a powerful reverse engineering tool that will help us visualize the disassembly and decompiled code.

Step 1: Create a New Project

1. Launch Ghidra
2. A window will appear - select **File -> New Project**
3. Choose **Non-Shared Project** and click **Next**
4. Enter the Project Name: `0x0001_hello-world`
5. Click **Finish**

Step 2: Import the Binary

1. Open your file explorer and navigate to the Embedded-Hacking folder
2. **Drag and drop** the `0x0001_hello-world.elf` file into the folder panel within the Ghidra application

Step 3: Understand the Import Dialog

In the small window that appears, you will see the file identified as an **ELF** (Executable and Linkable Format).

Tip: What is an ELF file?

ELF stands for **Executable and Linkable Format**. This format includes **symbols** - human-readable names for functions and variables. These symbols make reverse engineering much easier because you can see function names like `main` and `printf` instead of just memory addresses.

In future weeks, we will work with **stripped binaries** (`.bin` files) that do not contain these symbols. This is more realistic for real-world reverse engineering scenarios where symbols have been removed to make analysis harder.

3. Click **Ok** to import the file
4. **Double-click** on the file within the project window to open it in the CodeBrowser

Step 4: Auto-Analyze the Binary

When prompted, click **Yes** to auto-analyze the binary. Accept the default analysis options and click **Analyze**.

Ghidra will now process the binary, identifying functions, strings, and cross-references. This may take a moment.

Reviewing the Main Function in Ghidra

Once analysis is complete, let's find our `main` function:

1. In the **Symbol Tree** panel on the left, expand **Functions**
2. Look for `main` in the list (you can also use **Search -> For Address or Label** and type "main")
3. Click on `main` to navigate to it

What You'll See

Ghidra shows you two views of the code:

Listing View (Center Panel) - The disassembled ARM assembly:

```

*****
*                               FUNCTION
*****
int main (void )
    assume LRset = 0x0
    assume TMode = 0x1
    r0:4          <RETURN>
main
XREF[3]:      Entry Point (*) ,
              _reset_handler:1000018c (c) ,

```

		0x0001_hello-world.c:4 (2)		.debug_frame:00000018 (*)
		0x0001_hello-world.c:5 (2)		
10000234	08 b5	push {r3,lr}		
		0x0001_hello-world.c:5 (4)		
10000236	01 f0 99 f9	bl stdio_init_all		_Bool stdio_init_all(void)
		LAB_1000023a	XREF[1]:	10000240 (j)
		0x0001_hello-world.c:7 (6)		
		0x0001_hello-world.c:8 (6)		
1000023a	02 48	ldr r0=>__EH_FRAME_BEGIN__ ,[DAT_10000244]		= "hello, world\r"
				= 100019CC
1000023c	01 f0 de f9	bl __wrap_puts		int __wrap_puts(char * s)
		0x0001_hello-world.c:7 (8)		
10000240	fb e7	b LAB_1000023a		
10000242	00	?? 00h		
10000243	bf	?? BFh		
		DAT_10000244	XREF[1]:	main:1000023a (R)
10000244	cc 19 00 10	undefine 100019CC		? -> 100019cc

Decompile View (Right Panel) - The reconstructed C code:

```
int main(void)
{
    stdio_init_all();
    do {
        __wrap_puts("hello, world\r");
    } while( true );
}
```

Notice how Ghidra reconstructed our original C code! The decompiler recognized the infinite loop and the puts call (the compiler optimized printf to puts since we're just printing a simple string).

Why We Start with .elf Files

We're using the `.elf` file because it contains symbols that help us learn:

- Function names are visible (`main`, `stdio_init_all`, `puts`)
- Variable names may be preserved
- The structure of the code is easier to understand

In future weeks, we'll work with .bin files that have been stripped of symbols. This will teach you how to identify functions and understand code when you don't have these helpful hints!

Part 6: Summary and Review

What We Learned

1. **Registers:** The ARM Cortex-M33 has 13 general-purpose registers ($r0$ - $r12$), plus special registers for the stack pointer ($r13$ /SP), link register ($r14$ /LR), and program counter ($r15$ /PC).
2. **The Stack:**
 - Grows downward in memory
 - PUSH adds items (SP decreases)
 - POP removes items (SP increases)
 - Used to save return addresses and register values
3. **Memory Layout:**
 - Code lives in flash memory starting at 0×10000000
 - Stack lives in RAM around 0×20080000

4. **GDB Basics:** We learned the essential commands for connecting to hardware and examining code:

Command	What It Does
target extended-remote :3333	Connect to OpenOCD debug server
monitor reset halt	Reset and halt the processor
b main	Set breakpoint at main function
c	Continue running until breakpoint
disas	Disassemble current function
i r	Show all register values

5. **Ghidra Static Analysis:** We set up a Ghidra project and analyzed our binary:

- Imported the ELF file with symbols
- Found the main function
- Saw the decompiled C code
- Understood how assembly maps to C

6. **Little-Endian:** The RP2350 stores multi-byte values with the least significant byte at the lowest address, making them appear “backwards” when viewed as a single value.

The Program Flow

1. push {r3, lr}	
Save registers to stack	
2. bl stdio_init_all	
Initialize standard I/O	
3. ldr r0, [pc, #8]	
Load address of "hello, world" into r0	
4. bl __wrap_puts	
Print the string	
5. b.n (back to step 3)	
Infinite loop!	

Note: The detailed hands-on GDB debugging (stepping through code, watching the stack, examining memory) will be covered in Week 2!

Key Takeaways

1. **Reverse engineering combines static and dynamic analysis** - we look at the code (static with Ghidra) and run it to see what happens (dynamic with GDB).
2. **The stack is fundamental** - understanding how push/pop work is essential for following function calls.
3. **GDB and Ghidra work together** - Ghidra helps you understand the big picture, GDB lets you watch it happen live.
4. **Assembly isn't scary** - each instruction does one simple thing. Put them together and you understand the whole program!

5. **Everything is just numbers** - whether it's code, data, or addresses, it's all stored as numbers in memory.
-

Glossary

Term	Definition
Assembly	Human-readable representation of machine code
Breakpoint	A marker that tells the debugger to pause execution
GDB	GNU Debugger - a tool for examining running programs
Hex/Hexadecimal	Base-16 number system (0-9, A-F)
Little-Endian	Storing the least significant byte at the lowest address
Microcontroller	A small computer on a single chip
Program Counter	Register that points to the next instruction
Register	Fast storage inside the processor
Stack	Memory region for temporary storage during function calls
Stack Pointer	Register that points to the top of the stack
XIP	Execute In Place - running code directly from flash