EMBEDDED HACKING

Reverse Engineering the Raspberry Pi Pico 2

Week 3: Embedded System Analysis: Understanding the RP2350 Architecture w/ Comprehensive Firmware Analysis

ARM Cortex-M33 • GDB • Ghidra

September 7, 2026

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 tutorial, you will be able to:

- Understand how the RP2350 boots from the on-chip bootrom
- Know what the vector table is and why it's important
- Trace the complete boot sequence from power-on to `main()`
- Understand XIP (Execute In Place) and how code runs from flash
- Read and analyze the startup assembly code (`crt0.S`)
- Use GDB to examine the boot process step by step
- Use Ghidra to statically analyze the boot sequence
- Understand the difference between Thumb mode addressing and actual addresses

---

Review from Weeks 1-2

This week builds on your GDB and Ghidra skills from previous weeks:

- **GDB Commands** (`x`, `b`, `c`, `si`, `disas`, `i r`) - We'll use all of these to trace the boot process
- **Memory Layout** (Flash at `0x10000000`, RAM at `0x20000000`) - Understanding where code and data live
- **Registers** (`r0` - `r12`, SP, LR, PC) - We'll watch how they're initialized during boot
- **Ghidra Analysis** - Decompiling and understanding assembly in a visual tool
- **Thumb Mode** - Remember addresses with LSB=1 indicate Thumb code

---

The Code We're Analyzing

Throughout this week, we'll continue working with 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");
}
```

But this week, we're going **deeper** - we'll understand everything that happens BEFORE `main()` even runs! How does the chip know where `main()` is? How does the stack get initialized? Let's find out!

Part 1: Understanding the Boot Process

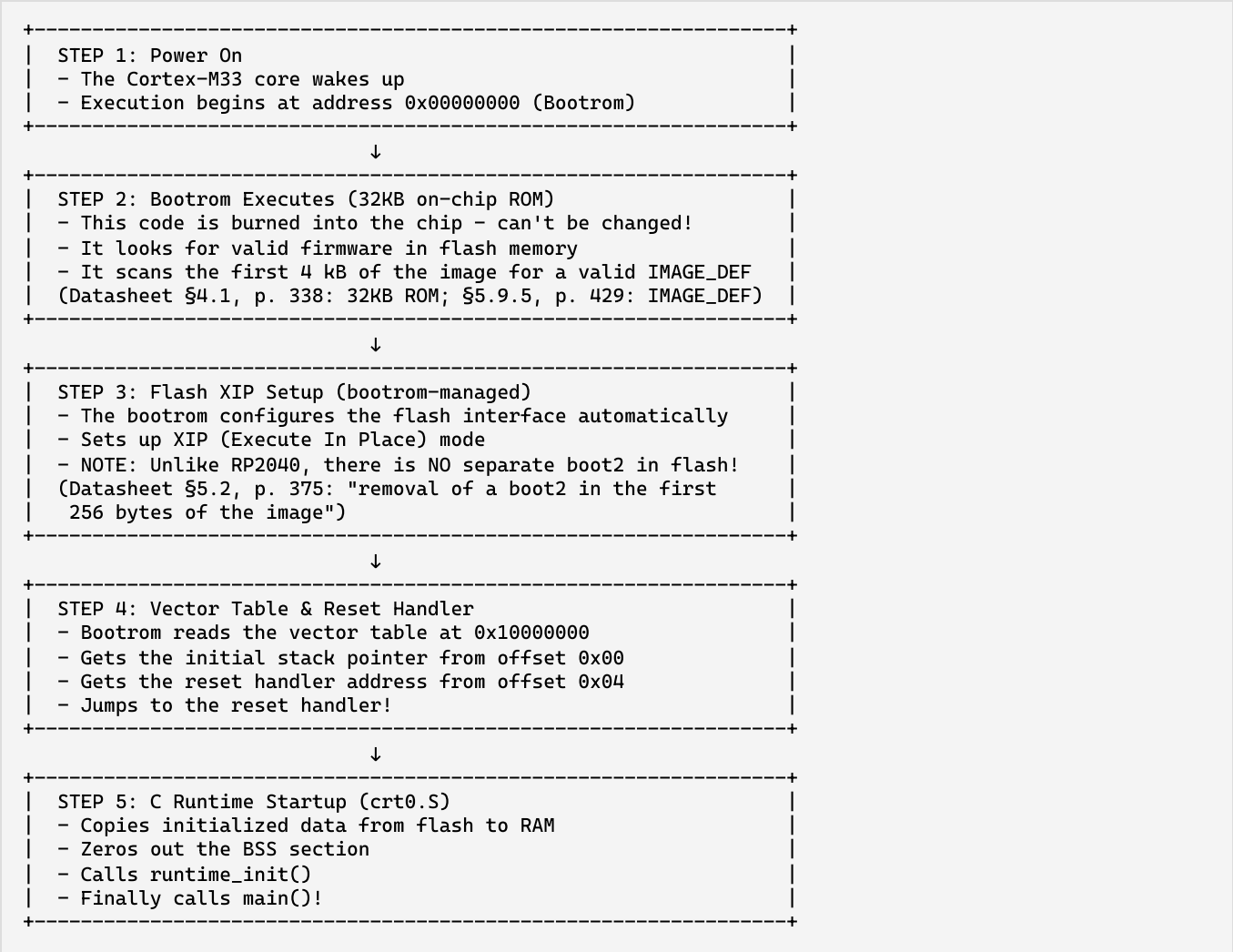
What Happens When You Power On?

When you plug in your Raspberry Pi Pico 2, a lot happens before your `main()` function runs! Think of it like waking up in the morning:

1. **First, your alarm goes off** (Power is applied to the chip)
2. **You open your eyes** (The bootrom starts running)
3. **You check your phone** (The bootrom looks for valid code in flash)
4. **You get out of bed** (The bootrom jumps to your program)
5. **You brush your teeth, get dressed** (Startup code initializes everything)
6. **Finally, you start your day** (Your `main()` function runs!)

Each of these steps has a corresponding piece of code. Let's explore them all!

The RP2350 Boot Sequence Overview



Part 2: The Bootrom - Where It All Begins

What is the Bootrom?

The **bootrom** is a 32KB piece of code that is permanently burned into the RP2350 chip at the factory. You cannot change it - it's "mask ROM" (Read Only Memory).

Think of the bootrom like the BIOS in your computer - it's the first thing that runs and is responsible for finding and loading your actual program.

Key Bootrom Facts

Property	Value	Description
Size	32 KB	Small but powerful
Location	`0x00000000`	The very first address in memory
Modifiable?	**NO**	Burned into silicon at the factory
Purpose	Boot the chip	Find and load your firmware

What Does the Bootrom Do?

1. **Initialize Hardware**: Sets up clocks, resets peripherals
2. **Check Boot Sources (Discovery)**: Scans configured boot sources (for this course: flash) to find a candidate firmware image region.
3. **Validate Firmware (Validation)**: Verifies that candidate by finding IMAGE_DEF start/end markers and parsing the block.
4. **Configure Flash**: Sets up the XIP interface
5. **Jump to Your Code**: Reads the vector table and jumps to your reset handler

The IMAGE_DEF Structure

The bootrom looks for a special marker in your firmware called **IMAGE_DEF**. This tells the bootrom "Hey, there's valid code here!"

Here's what it looks like in the Pico SDK:

```
.section .picobin_block, "a" // placed in flash
.word 0xffffded3             // PICOBIN_BLOCK_MARKER_START ← ROM looks for this!
.byte 0x42                   // PICOBIN_BLOCK_ITEM_1BS_IMAGE_TYPE
.byte 0x1                    // item is 1 word in size
.hword 0b0001000000100001    // SECURE mode (0x1021)
.byte 0xff                   // PICOBIN_BLOCK_ITEM_2BS_LAST
.hword 0x0001                // item is 1 word in size
.byte 0x0                    // pad
.word 0x0                    // relative pointer to next block (0 = loop to self)
.word 0xab123579             // PICOBIN_BLOCK_MARKER_END
```

The magic numbers:

- `0xffffded3` = Start marker ("I'm a valid Pico binary!")
- `0xab123579` = End marker ("End of the header block")

See This Exact Block in Your ELF (Commands + Real Output)

Use these commands to view the IMAGE_DEF bytes directly in the ELF:

```
arm-none-eabi-objdump -s --start-address=0x1000013c --stop-address=0x10000150 build/0x0001_hello-
world.elf
arm-none-eabi-objdump -s --start-address=0x10000130 --stop-address=0x10000154 build/0x0001_hello-
world.elf
arm-none-eabi-gdb build/0x0001_hello-world.elf -ex "x/20bx 0x1000013c" -ex quit
```

Actual output from this lesson build:

```
build/0x0001_hello-world.elf:     file format elf32-littlearm

Contents of section .text:
 1000013c 42012110 ff010000 b01b0000 793512ab  B.!.........y5..
 1000014c 4ff00000                             O...

build/0x0001_hello-world.elf:     file format elf32-littlearm

Contents of section .text:
 10000130 a0010010 90a31ae7 d3deffff 42012110  ............B.!.
 10000140 ff010000 b01b0000 793512ab 4ff00000  ........y5..O...
 10000150 1e490860                             .I.`
```

Command 1 explained (`--start-address=0x1000013c --stop-address=0x10000150`):

- Starts at `0x1000013c` , so it does **not** include the start marker at `0x10000138` (`d3deffff`).
- Shows IMAGE_DEF body fields and the end marker:
  - `42012110` = `42 01 21 10` (item type/size + secure mode field)
  - `ff010000` = last-item marker + size + pad
  - `b01b0000` = next word in the block payload for this build
  - `793512ab` = `PICOBIN_BLOCK_MARKER_END` (`0xab123579` in little-endian)
- `4ff00000` at `0x1000014c` is already the next instruction word after IMAGE_DEF.

Command 2 explained (`--start-address=0x10000130 --stop-address=0x10000154`):

- Starts earlier, so it captures context **and** both IMAGE_DEF markers.
- `a0010010 90a31ae7` = binary-info context before IMAGE_DEF.
- `d3deffff` at `0x10000138` = `PICOBIN_BLOCK_MARKER_START` .
- `793512ab` at `0x10000148` = `PICOBIN_BLOCK_MARKER_END` .
- `4ff00000 1e490860` = code words after the IMAGE_DEF block.
- This command proves the full block location for this build: `0x10000138` to `0x1000014b` .

Important: IMAGE_DEF offset can vary by build. In this build, the start marker `d3deffff` is at `0x10000138` (not `0x1000013c`), so always search for the marker bytes instead of assuming a fixed address.

Part 3: Understanding XIP (Execute In Place)

> **REVIEW:** In Week 1, we learned that our code lives at `0x10000000` in flash memory. We used `x/1000i 0x10000000` to find our `main` function. Now we'll understand WHY code is at this address!

What is XIP?

XIP (Execute In Place) means the processor can run code directly from flash memory without copying it to RAM first.

Think of it like reading a book:

- **Without XIP**: You photocopy every page into a notebook, then read from the notebook
- **With XIP**: You just read directly from the book!

Why Use XIP?

Advantage	Explanation
Saves RAM	Code stays in flash, RAM is free for data
Faster Boot	No need to copy entire program to RAM first
Simpler	Less memory management needed

XIP Memory Address

The XIP flash region starts at address `0x10000000`. This is where your compiled code lives!

```
+-----------------------------------------------------+
|  Address: 0x10000000 (XIP Base)                     |
|  +-------------------------------------------------+|
|  |  Vector Table (first thing here!)               ||
|  |  - Stack Pointer at offset 0x00                 ||
|  |  - Reset Handler at offset 0x04                 ||
|  |  - Other exception handlers...                  ||
|  +-------------------------------------------------+|
|  |  Your Code                                      ||
|  |  - Reset handler                                ||
|  |  - main() function                              ||
|  |  - Other functions                              ||
|  +-------------------------------------------------+|
|  |  Read-Only Data                                 ||
|  |  - Strings like "hello, world"                  ||
|  |  - Constant values                              ||
|  +-------------------------------------------------+|
+-----------------------------------------------------+
```

Part 4: The Vector Table - The CPU's Instruction Manual

What is the Vector Table?

The **vector table** is a list of addresses at the very beginning of your program. It tells the CPU:

1. Where to set the stack pointer
2. Where to start executing code (reset handler)
3. Where to go when errors or interrupts happen

Think of it like the table of contents in a book - it tells you where to find everything!

Vector Table Layout

The vector table lives at `0x10000000` and looks like this:

Offset	Address	Content	Description
`0x00`	`0x10000000`	`0x20082000`	Initial Stack Pointer (SP)
`0x04`	`0x10000004`	`0x1000015d`	Reset Handler (entry point)
`0x08`	`0x10000008`	`0x1000011b`	NMI Handler
`0x0C`	`0x1000000C`	`0x1000011d`	HardFault Handler

Understanding Thumb Mode Addressing

Important Concept Alert!

Look at the reset handler address: `0x1000015d`. Notice it ends in `d` (an odd number)?

On ARM Cortex-M processors, all code runs in **Thumb mode**. The processor uses the **least significant bit (LSB)** of an address to indicate this:

LSB	Mode	Meaning
`1` (odd)	Thumb	"This is Thumb code"
`0` (even)	ARM	"This is ARM code" (not used on Cortex-M)

So `0x1000015d` means:

- The actual code is at `0x1000015c` (even address)
- The `+1` tells the processor "use Thumb mode"

GDB vs Ghidra:

- GDB shows `0x1000015d` (with Thumb bit)
- Ghidra shows `0x1000015c` (actual instruction address)
- Both are correct! They're just displaying it differently.

Part 5: The Linker Script - Memory Mapping

What is a Linker Script?

The **linker script** tells the compiler where to put different parts of your program in memory. It's like an architect's blueprint for memory!

Finding the Linker Script

On Windows with the Pico SDK 2.2.0, you'll find it at:

```
C:\Users\<username>\.pico-sdk\sdk\2.2.0\src\rp2_common\pico_crt0\rp2350\memmap_default.ld
```

Key Parts of the Linker Script

```
MEMORY
{
    INCLUDE "pico_flash_region.ld"
    RAM(rwx) : ORIGIN = 0x20000000, LENGTH = 512k
    SCRATCH_X(rwx) : ORIGIN = 0x20080000, LENGTH = 4k
    SCRATCH_Y(rwx) : ORIGIN = 0x20081000, LENGTH = 4k
}
```

What this means:

Region	Start Address	Size	Purpose
Flash	`0x10000000`	(varies)	Your code (XIP)
RAM	`0x20000000`	512 KB	Main striped SRAM
SCRATCH_X	`0x20080000`	4 KB	SRAM8: shared, non-striped SRAM
SCRATCH_Y	`0x20081000`	4 KB	SRAM9: shared, non-striped SRAM

`SCRATCH_X` and `SCRATCH_Y` are linker-script names for SRAM8 and SRAM9. They are **not hardware-assigned to Core 0 or Core 1**: both cores can access both banks. Software may reserve either bank for per-core data to reduce bank contention. This particular linker script selects `SCRATCH_Y` for the Core 0 stack; that is a software allocation choice, not a property of SRAM9.

Where Does This Build's Core 0 Stack Come From?

The linker script calculates the initial stack pointer:

```
__StackTop = ORIGIN(SCRATCH_Y) + LENGTH(SCRATCH_Y);
```

Let's do the math:

- `ORIGIN(SCRATCH_Y)` = `0x20081000`
- `LENGTH(SCRATCH_Y)` = `0x1000` (4 KB)
- `__StackTop` = `0x20081000` + `0x1000` = `0x20082000`

This value (`0x20082000`) is what we see at offset `0x00` in the vector table. It is the initial Core 0 stack pointer for this build because its linker script chose `SCRATCH_Y` ; it does not reserve `SCRATCH_Y` for Core 0 in hardware.

Part 6: Setting Up Your Environment (GDB - Dynamic Analysis)

> **REVIEW:** This setup is identical to Weeks 1-2. If you need a refresher on OpenOCD and GDB connection, refer back to Week 1 Part 4 or Week 2 Part 5.

Prerequisites

Before we start, make sure you have:

1. A Raspberry Pi Pico 2 board with debug probe connected

2. OpenOCD installed and configured
3. GDB (`arm-none-eabi-gdb`) installed
4. The "hello-world" binary loaded on your Pico 2
5. Access to the Pico SDK source files (for reference)

Starting the Debug Session

Terminal 1 - 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"
```

Terminal 2 - Start GDB:

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

Connect to target:

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

Part 7: Hands-On GDB Tutorial - Examining the Vector Table

> **REVIEW:** We're using the same `x` (examine) command from Week 1. Remember: `x/Nx` shows N hex values, `x/Ni` shows N instructions, `x/s` shows strings.

Step 1: Examine the Vector Table

Let's look at the first 4 entries of the vector table at `0x10000000`:

Type this command:

```
(gdb) x/4x 0x10000000
```

What this command means:

- `x` = examine memory (Week 1 review!)
- `/4x` = show 4 values in hexadecimal
- `0x10000000` = the address of the vector table

You should see:

```
0x10000000 <__vectors>: 0x20082000  0x1000015d  0x1000011b  0x1000011d
```

Step 2: Understanding What We See

> **REVIEW:** In Weeks 1-2, we saw both `sp = 0x20082000` at the clean breakpoint at `main` and lower values like `0x20081fc8` or `0x20081ff8` after additional stack activity. Here we are looking at the *initial* stack pointer from the vector table before any code runs.

Let's decode each value:

Address	Value	Meaning
`0x10000000`	`0x20082000`	Initial Stack Pointer - top of SCRATCH_Y
`0x10000004`	`0x1000015d`	Reset Handler + 1 (Thumb bit)
`0x10000008`	`0x1000011b`	NMI Handler + 1 (Thumb bit)
`0x1000000C`	`0x1000011d`	HardFault Handler + 1 (Thumb bit)

Key Insight: The stack pointer (`0x20082000`) is exactly what the linker script calculated! And all the handler addresses have their LSB set to `1` for Thumb mode.

Step 3: Verify the Stack Pointer Calculation

Let's confirm our math by examining what's at `0x10000000` :

Type this command:

```
(gdb) x/x 0x10000000
```

You should see:

```
0x10000000 <__vectors>: 0x20082000
```

This matches:

- `SCRATCH_Y` starts at `0x20081000`
- `SCRATCH_Y` is 4 KB (`0x1000` bytes)
- `0x20081000` + `0x1000` = `0x20082000`

Part 8: Examining the Reset Handler

> **REVIEW:** We used `x/5i` extensively in Weeks 1-2 to examine our `main` function. Now we'll use the same technique to examine the code that runs BEFORE `main` !

Step 4: Disassemble the Reset Handler

The reset handler is where execution begins after the bootrom hands off control. Let's look at it:

Type this command:

```
(gdb) x/3i 0x1000015c
```

Note: We use `0x1000015c` (even) not `0x1000015d` (odd) because we want to see the actual instructions!

You should see:

```
0x1000015c <_reset_handler>: mov.w   r0, #3489660928 @ 0xd0000000
0x10000160 <_reset_handler+4>:       ldr     r0, [r0, #0]
0x10000162 <_reset_handler+6>:
 cbz r0, 0x1000016a <hold_non_core0_in_bootrom+6>
```

Step 5: Understanding the Reset Handler

Let's break down what these first three instructions do:

Instruction 1: `mov.w r0, #0xd0000000`

This loads the address `0xd0000000` into register `r0`. But what's at that address?

That's the **SIO (Single-cycle I/O) base address**! The SIO block contains a special register called **CPUID** that tells us which core we're running on.

Instruction 2: `ldr r0, [r0, #0]`

This reads the value at address `0xd0000000` (the CPUID register) into `r0`.

Core	CPUID Value
Core 0	`0`
Core 1	`1`

Instruction 3: `cbz r0, 0x1000016a`

This is "Compare and Branch if Zero". If `r0` is `0` (meaning we're on Core 0), branch to `0x1000016a` to continue with startup. Otherwise, we're on Core 1 and need to handle that differently.

Why Check Which Core We're On?

The RP2350 has **two cores**, but only **Core 0** should run the startup code! If both cores tried to initialize the same memory and peripherals, chaos would ensue.

So the reset handler checks:

- **Core 0?** -> Continue with startup
- **Core 1?** -> Go back to the bootrom and wait

Part 9: The Complete Reset Handler Flow

Step 6: Examine More of the Reset Handler

Let's look at more instructions to see the full picture:

Type this command:

```
(gdb) x/20i 0x1000015c
```

You should see:

```
  0x1000015c <_reset_handler>: mov.w   r0, #3489660928 @ 0xd0000000
  0x10000160 <_reset_handler+4>:       ldr     r0, [r0, #0]
  0x10000162 <_reset_handler+6>:
   cbz r0, 0x1000016a <hold_non_core0_in_bootrom+6>
  0x10000164 <hold_non_core0_in_bootrom>:      mov.w   r0, #0
  0x10000168 <hold_non_core0_in_bootrom+4>:
   b.n 0x10000150 <_enter_vtable_in_r0>
  0x1000016a <hold_non_core0_in_bootrom+6>:
   add r4, pc, #52     @ (adr r4, 0x100001a0 <data_cpy_table>)
  0x1000016c <hold_non_core0_in_bootrom+8>:    ldmia   r4!, {r1, r2, r3}
  0x1000016e <hold_non_core0_in_bootrom+10>:   cmp     r1, #0
  0x10000170 <hold_non_core0_in_bootrom+12>:
   beq.n       0x10000178 <hold_non_core0_in_bootrom+20>
  0x10000172 <hold_non_core0_in_bootrom+14>:
   bl  0x1000019a <data_cpy>
  0x10000176 <hold_non_core0_in_bootrom+18>:
   b.n 0x1000016c <hold_non_core0_in_bootrom+8>
  0x10000178 <hold_non_core0_in_bootrom+20>:
   ldr r1, [pc, #84]   @ (0x100001d0 <data_cpy_table+48>)
  0x1000017a <hold_non_core0_in_bootrom+22>:
   ldr r2, [pc, #88]   @ (0x100001d4 <data_cpy_table+52>)
  0x1000017c <hold_non_core0_in_bootrom+24>:   movs    r0, #0
  0x1000017e <hold_non_core0_in_bootrom+26>:
   b.n 0x10000182 <bss_fill_test>
  0x10000180 <bss_fill_loop>:  stmia   r1!, {r0}
  0x10000182 <bss_fill_test>:  cmp     r1, r2
  0x10000184 <bss_fill_test+2>:        bne.n   0x10000180 <bss_fill_loop>
  0x10000186 <platform_entry>:
   ldr r1, [pc, #80]   @ (0x100001d8 <data_cpy_table+56>)
  0x10000188 <platform_entry+2>:       blx     r1
```

Step 7: Understanding the Startup Phases

The reset handler performs several phases:

```
+-------------------------------------------------------------------+
|  PHASE 1: Core Check (0x1000015c - 0x10000168)                    |
|  - Check CPUID to see which core we're on                         |
|  - If not Core 0, go back to bootrom                              |
+-------------------------------------------------------------------+
                              ↓
+-------------------------------------------------------------------+
|  PHASE 2: Data Copy Setup & Loop (0x1000016a - 0x10000176)        |
|  - Set up the data_cpy_table pointer and load each copy triplet   |
|  - Copy initialized variables from flash to RAM                   |
+-------------------------------------------------------------------+
                              ↓
+-------------------------------------------------------------------+
|  PHASE 3: BSS Setup & Clear (0x10000178 - 0x10000184)             |
|  - Load the BSS start/end addresses into r1 and r2                |
|  - GDB labels those literals as `data_cpy_table+48/+52`           |
|  - Zero out all uninitialized global variables                    |
+-------------------------------------------------------------------+
                              ↓
+-------------------------------------------------------------------+
|  PHASE 4: Platform Entry Begins (0x10000186 - 0x10000188 shown)   |
|  - Load the runtime_init() pointer from the table                 |
|  - Branch to runtime_init() with `blx r1`                         |
|  - `main()` and `exit()` appear a few instructions later          |
+-------------------------------------------------------------------+
```

Part 10: Understanding the Data Copy Phase

What is the Data Copy Phase?

> **REVIEW:** In Week 2, we learned that flash is read-only and SRAM is read-write. That's why the startup code must COPY initialized variables from flash to RAM - they can't be modified in flash!

When you write C code like this:

```
int my_counter = 42; // Initialized global variable
```

The value `42` is stored in flash memory (because flash is non-volatile). But variables need to live in RAM to be modified! So the startup code **copies** these initial values from flash to RAM.

Step 8: Find the Data Copy Table

The data copy table contains entries that describe what to copy where. Let's examine it:

Type this command:

```
(gdb) x/12x 0x100001a0
```

You should see something like:

```
0x100001a0 <data_cpy_table>:    0x10001b4c      0x20000110      0x200002ac 0x10001ce8
0x100001b0 <data_cpy_table+16>: 0x20080000      0x20080000      0x10001ce8 0x20081000
0x100001c0 <data_cpy_table+32>: 0x20081000      0x00000000      0x00004770 0xe000ed08
```

The data_cpy_table contains multiple entries. Each entry has three values:

1. **Source address** (in flash)
2. **Destination address** (in RAM)
3. **End address** (where to stop copying)

In the output above, we see:

- **First entry**: `0x10001b4c` (source), `0x20000110` (dest), `0x200002ac` (end)
- **Second entry starts**: `0x10001ce8` (source of next entry), ...

The table ends with an entry where the source address is `0x00000000` (which signals "no more entries").

Step 9: Watch the Data Copy Loop

The data copy loop works like this:

```
+-------------------------------------------+
|  1. Load source, dest, end from table     |
|  2. If source == 0, we're done            |
|  3. Otherwise, copy word by word          |
|  4. Go back to step 1 for next entry      |
+-------------------------------------------+
```

The actual code (starting at `0x1000016c` in the reset handler):

```
0x1000016c <hold_non_core0_in_bootrom+8>:    ldmia   r4!, {r1, r2, r3}
0x1000016e <hold_non_core0_in_bootrom+10>:   cmp     r1, #0
0x10000170 <hold_non_core0_in_bootrom+12>:
beq.n       0x10000178 <hold_non_core0_in_bootrom+20>
0x10000172 <hold_non_core0_in_bootrom+14>:
bl  0x1000019a <data_cpy>
0x10000176 <hold_non_core0_in_bootrom+18>:
b.n 0x1000016c <hold_non_core0_in_bootrom+8>
```

> Tip: **Note:** You can see this code in **Step 6** earlier where we examined the reset handler with `x/20i 0x1000015c`.

Part 11: Understanding the BSS Clear Phase

What is BSS?

BSS stands for "Block Started by Symbol" (historical name). It's the section of memory for **uninitialized global variables**.

When you write:

```
int my_counter; // Uninitialized - will be in BSS
```

The C standard says this variable **must start at zero**. The BSS clear phase zeros out this entire region.

Step 10: Examine the BSS Clear Loop

Type this command:

```
(gdb) x/5i 0x10000178
```

You should see:

```
0x10000178 <hold_non_core0_in_bootrom+20>:
ldr r1, [pc, #84]   @ (0x100001d0 <data_cpy_table+48>)
0x1000017a <hold_non_core0_in_bootrom+22>:
ldr r2, [pc, #88]   @ (0x100001d4 <data_cpy_table+52>)
0x1000017c <hold_non_core0_in_bootrom+24>:   movs    r0, #0
0x1000017e <hold_non_core0_in_bootrom+26>:
b.n 0x10000182 <bss_fill_test>
0x10000180 <bss_fill_loop>:  stmia   r1!, {r0}
```

The first two `ldr` instructions are still part of the **BSS clear setup**, even though GDB shows the source words as `data_cpy_table+48` and `data_cpy_table+52`. That label means the two literal words live in the same nearby constant block as the copy-table entries; it does **not** mean the code is still performing `.data` copies. At this point, `r1` becomes the BSS start address, `r2` becomes the BSS end address, and the loop beginning at `0x10000180` zeros that range.

Understanding the Loop

```
+--------------------------------------------+
|  r1 = start of BSS section                 |
|  r2 = end of BSS section                   |
|  r0 = 0                                    |
|                                            |
|  LOOP:                                     |
|    Store 0 at address r1                   |
|    Increment r1 by 4 bytes                 |
|    If r1 != r2, repeat                     |
+--------------------------------------------+
```

Part 12: Examining Exception Handlers

Step 11: Look at the Default Exception Handlers

What happens if an exception occurs (like a HardFault)? Let's look:

Type this command:

```
(gdb) x/10i 0x10000110
```

You should see:

```
0x10000110 <isr_usagefault>: mrs     r0, IPSR
0x10000114 <isr_usagefault+4>:       subs    r0, #16
0x10000116 <unhandled_user_irq_num_in_r0>:   bkpt    0x0000
0x10000118 <isr_invalid>:    bkpt    0x0000
0x1000011a <isr_nmi>:        bkpt    0x0000
0x1000011c <isr_hardfault>:  bkpt    0x0000
0x1000011e <isr_svcall>:     bkpt    0x0000
0x10000120 <isr_pendsv>:     bkpt    0x0000
0x10000122 <isr_systick>:    bkpt    0x0000
0x10000124 <__default_isrs_end>:
            @ <UNDEFINED> instruction: 0xebf27188
```

What is `bkpt`?

The `bkpt` instruction is a **breakpoint**. When executed, it stops the processor and triggers the debugger!

These are the **default** exception handlers - they just stop the program so you can debug. In your own code, you can override these with real handlers.

Why So Many Handlers?

Each type of exception has its own handler:

Handler	Purpose
`isr_nmi`	Non-Maskable Interrupt (can't be disabled)
`isr_hardfault`	Serious error (bad memory access, etc.)
`isr_svcall`	Supervisor Call (used by RTOSes)
`isr_pendsv`	Pendable Supervisor (also for RTOSes)
`isr_systick`	System Timer tick interrupt

Part 13: Finding Where Main is Called

Step 12: Trace Reset Handler to `main`

Start at the Cortex-M vector table. Word `0x00000000` is the initial stack pointer; word `0x00000004` is the **reset handler address** loaded into `pc` when the processor resets:

```
(gdb) x/2wx 0x00000000
```

Disassemble the reset-handler address shown at `0x00000004`, then continue through startup until `platform_entry`:

```
(gdb) x/10i RESET_HANDLER_ADDRESS
(gdb) b platform_entry
(gdb) c
```

At `platform_entry`, the three `ldr r1` / `blx r1` pairs are indirect calls:

```
0x10000186 <platform_entry+0>:  ldr     r1, [pc, #80]
0x10000188 <platform_entry+2>:  blx     r1       (first call)
0x1000018a <platform_entry+4>:  ldr     r1, [pc, #80]
0x1000018c <platform_entry+6>:  blx     r1       (second call)
0x1000018e <platform_entry+8>:  ldr     r1, [pc, #80]
0x10000190 <platform_entry+10>: blx     r1       (third call)
```

Prove That the Second Call Is `main`

Stop at the second `blx r1`. `r1` holds the target; inspect it and then disassemble that address:

```
(gdb) b *0x1000018c
(gdb) c

Thread 1 "rp2350.dap.core0" hit Breakpoint 1, platform_entry ()
   at C:/Users/assem.KEVINTHOMAS/.pico-sdk/sdk/2.2.0/src/rp2_common/pico_crt0/crt0.S:515
515         blx r1
(gdb) x/x 0x1000018c
0x1000018c <platform_entry+6>:  0x49144788
(gdb) disas
Dump of assembler code for function platform_entry:
   0x10000186 <+0>:     ldr     r1, [pc, #80]   @ (0x100001d8 <data_cpy_table+56>)
   0x10000188 <+2>:     blx     r1
   0x1000018a <+4>:     ldr     r1, [pc, #80]   @ (0x100001dc <data_cpy_table+60>)
=> 0x1000018c <+6>:     blx     r1
   0x1000018e <+8>:     ldr     r1, [pc, #80]   @ (0x100001e0 <data_cpy_table+64>)
   0x10000190 <+10>:    blx     r1
   0x10000192 <+12>:    bkpt    0x0000
   0x10000194 <+14>:    b.n     0x10000192 <platform_entry+12>
End of assembler dump.
(gdb) x/x 0x1000018c
0x1000018c <platform_entry+6>:  0x49144788
(gdb) x/x $r1
0x10000235 <main>:      0x99f001b5
(gdb) disas $r1
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.
(gdb) x/x 0x100001dc
0x100001dc <data_cpy_table+60>: 0x10000235
```

`x/x $r1` reports `0x10000235 <main>` because Thumb function pointers have bit 0 set. The actual instruction starts at `0x10000234`, as `disas $r1` shows. The preceding `ldr r1, [pc, #80]` reads the literal-pool word at `0x100001dc`; `x/x 0x100001dc` confirms that word is `0x10000235`. This proves that the **second** indirect call enters `main()`.

The first call performs runtime initialization. When `main()` returns, the third call enters the SDK exit path; the following `bkpt` catches the unexpected case where that exit path returns.

Step 13: Set a Breakpoint at Main

> **REVIEW:** We've used `b main` and `b *ADDRESS` many times in Weeks 1-2. This is the same technique!

Let's verify we understand the boot process by setting a breakpoint at main:

Type this command:

```
(gdb) b main
```

You should see:

```
Breakpoint 1 at 0x10000234: file C:/Users/assem.KEVINTHOMAS/OneDrive/Documents/Embedded-
Hacking/0x0001_hello-world/0x0001_hello-world.c, line 5.
Note: automatically using hardware breakpoints for read-only addresses.
```

Now continue:

```
(gdb) c
```

You should see:

```
Continuing.

Thread 1 "rp2350.cm0" hit Breakpoint 1, main ()
    at C:/Users/assem.KEVINTHOMAS/OneDrive/Documents/Embedded-Hacking/0x0001_hello-world/0x0001_hello-
world.c:5
5           stdio_init_all();
(gdb)
```

We've traced the entire boot process from power-on to `main()`!

Part 14: Understanding the Binary Info Header

Step 14: Examine the Binary Info Header

Between the default ISRs and the reset handler, there's a special data structure called the **binary info header**. Let's look at it:

Type this command:

```
(gdb) x/5x 0x10000138
```

You should see:

```
0x10000138 <__binary_info_header_end>:  0xffffded3      0x10210142      0x000001ff  0x00001bb0
0x10000148 <__binary_info_header_end+16>:       0xab123579
```

Decoding the Binary Info Header

Address	Value	Meaning
`0x10000138`	`0xffffded3`	Start marker (PICOBIN_BLOCK_MARKER_START)
`0x1000013c`	`0x10212142`	Image type descriptor
`0x10000140`	`0x000001ff`	Item header/size field
`0x10000144`	`0x00001bb0`	Link to next block or data
`0x10000148`	`0xab123579`	End marker (PICOBIN_BLOCK_MARKER_END)

Why does GDB show this as instructions?

GDB doesn't know this is data, not code! It tries to disassemble it as Thumb instructions, which results in nonsense. This is why you'll see things like:

```
(gdb) x/i 0x10000138
```

```
   0x10000138 <__binary_info_header_end>:       udf     #211    @ 0xd3
```

That's not real code - it's the magic number `0xffffded3` being misinterpreted!

Part 15: Static Analysis with Ghidra - Examining the Boot Sequence

> **REVIEW:** In Week 1, we set up a Ghidra project and analyzed our hello-world binary. Now we'll use Ghidra to understand the boot sequence from a static analysis perspective!

Why Use Ghidra for Boot Analysis?

While GDB is excellent for dynamic analysis (watching code execute), Ghidra excels at:

- **Seeing the big picture** - Understanding code flow without running it
- **Cross-references** - Finding all places that call a function
- **Decompilation** - Seeing C-like code even for assembly routines
- **Annotation** - Adding notes and renaming functions for clarity

Step 15: Open Your Project in Ghidra

> **REVIEW:** If you haven't created the project yet, refer back to Week 1 Part 5 for setup instructions.

1. Launch Ghidra and open your `0x0001_hello-world` project
2. Double-click on the `.elf` file to open it in the CodeBrowser
3. If prompted to auto-analyze, click **Yes**

Step 16: Navigate to the Vector Table

1. In the **Navigation** menu, select **Go To...**
2. Type `0x10000000` and press Enter
3. You should see the vector table data

What you'll see in the Listing view:

```
                            //
                            // .text
                            // SHT_PROGBITS  [0x10000000 - 0x100019cb]
                            // ram:10000000-ram:100019cb
                            //
             assume spsr = 0x0  (Default)
                            __vectors                                       XREF[4]:     Entry Point
(*) ,
                            __flash_binary_start
runtime_init_install_ram_vector_
                            __VECTOR_TABLE
_elfProgramHeaders::00000028 (*) ,
                            __logical_binary_start
_elfSectionHeaders::00000034 (*)
        10000000 00             undefine   00h
        10000001 20             ??         20h
        10000002 08             ??         08h
        10000003 20             ??         20h
        10000004 5d             ??         5Dh    ]                                  ?  ->
1000015d
        10000005 01             ??         01h
        10000006 00             ??         00h
        10000007 10             ??         10h
        10000008 1b             ??         1Bh                                       ?  ->
1000011b
        10000009 01             ??         01h
        1000000a 00             ??         00h
        1000000b 10             ??         10h
        1000000c 1d             ??         1Dh                                       ?  ->
1000011d
        1000000d 01             ??         01h
        1000000e 00             ??         00h
        1000000f 10             ??         10h
...
```

> Tip: **Notice:** Ghidra shows the vector table data as individual bytes by default. You can see it has labeled the start as `__vectors`, `__flash_binary_start`, `__VECTOR_TABLE`, and `__logical_binary_start`. The arrows (like `? -> 1000015d`) show that Ghidra recognizes these bytes as pointers to code addresses! To see the data formatted as 32-bit addresses instead of bytes, you can right-click and retype the data.

Step 17: Navigate to the Reset Handler

1. In the Symbol Tree panel (left side), expand **Functions**
2. Find and click on `_reset_handler` (or search for it)
3. Alternatively, double-click on `_reset_handler` in the vector table listing

What you'll see in the Decompile view (right panel):

Ghidra will show you a decompiled version of the reset handler. While it won't be perfect C code (since this is hand-written assembly), it helps visualize the flow:

```
void _reset_handler(void)

{
  bool bVar1;
  undefined4 uVar2;
  int iVar3;
  undefined4 *puVar4;
  int *piVar5;
  int *piVar6;
  int *piVar7;

  if (_DAT_d0000000 != 0) {
    _DAT_e000ed08 = 0;
    bVar1 = (bool)isCurrentModePrivileged();
    if (bVar1) {
      setMainStackPointer(_gpio_set_function_masked64);
    }
                    /* WARNING: Could not recover jumptable at 0x1000015a. Too many branches */
                    /* WARNING: Treating indirect jump as call */
    (*pcRam00000004)(8,_gpio_set_function_masked64);
    return;
  }
  piVar5 = &data_cpy_table;
  uVar2 = 0;
  while( true ) {
    iVar3 = *piVar5;
    piVar6 = piVar5 + 1;
    piVar7 = piVar5 + 2;
    piVar5 = piVar5 + 3;
    if (iVar3 == 0) break;
    uVar2 = data_cpy(uVar2,iVar3,*piVar6,*piVar7);
  }
  for (puVar4 = (undefined4 *)&__TMC_END__; puVar4 != (undefined4 *)&end; puVar4 = puVar4 + 1) {
    *puVar4 = 0;
  }
  runtime_init();
  iVar3 = main();
                    /* WARNING: Subroutine does not return */
  exit(iVar3);
}
```

Step 18: Trace the Path to Main

Use the same evidence chain as GDB, but statically in the Listing view:

1. In the Symbol Tree, find the `main` function at `0x10000234`.
2. Right-click `main` and select **References -> Show References to main**.
3. Double-click the reference at `0x1000018c` to jump to the second `blx r1`.
4. Select the instruction immediately above it: `ldr r1,[DAT_100001dc]` at `0x1000018a`.
5. Double-click `DAT_100001dc`, or press **G** and enter `0x100001dc`.

You should see:

Location	Type	Label
`1000018c`	CALL	`blx r1` (to main)

At `0x100001dc`, Ghidra shows the literal-pool value `0x10000235`. That is the Thumb function pointer loaded into `r1` immediately before the call. Clear bit 0 to obtain the actual first instruction address:

```
0x10000235  (Thumb function pointer; bit 0 is set)
0x10000234  (main's first instruction; bit 0 cleared)
```

This proves the second indirect call at `0x1000018c` reaches `main`.

Step 19: Examine Platform Entry

In Ghidra, look at `platform_entry`:

Listing View:

```
                        platform_entry
                        crt0.S:512 (2)
        10000186 14  49         ldr        r1,[DAT_100001d8 ]                              =
1000137Dh
                        crt0.S:513 (2)
        10000188 88  47         blx        r1=>runtime_init                                void
runtime_init(void)
                        crt0.S:514 (2)
        1000018a 14  49         ldr        r1,[DAT_100001dc ]                              =
10000235h
                        crt0.S:515 (2)
        1000018c 88  47         blx        r1=>main                                        int
main(void)
                        crt0.S:516 (2)
        1000018e 14  49         ldr        r1,[DAT_100001e0 ]                              =
10001375h
                        crt0.S:517 (2)
        10000190 88  47         blx        r1=>exit                                        void
exit(int status)
                        LAB_10000192                                  XREF[1]:     10000194 (j)
                        crt0.S:521 (2)
        10000192 00  be         bkpt       0x0
                        crt0.S:522 (2)
        10000194 fd  e7         b          LAB_10000192
```

The `DAT_100001dc = 10000235h` annotation is the static proof. The preceding `ldr` loads that Thumb function pointer into `r1`; the following `blx r1` at `0x1000018c` calls it. Ghidra clears the Thumb bit and labels the target `main` at `0x10000234`.

> **Key Insight:** The reset vector identifies the reset handler, the reset handler reaches `platform_entry`, and this literal-pool entry proves that `platform_entry`'s second indirect call reaches `main`.

Step 20: Create a Boot Sequence Graph

Ghidra can visualize the call flow:

1. With `_reset_handler` selected, go to **Window -> Function Call Graph**
2. This shows a visual graph of all function calls from the reset handler
3. You will see `_reset_handler` at the top with arrows going down to its four direct callees: `data_cpy`, `runtime_init`, `main`, and `exit`

Comparing GDB and Ghidra for Boot Analysis

Aspect	GDB (Dynamic)	Ghidra (Static)
Sees runtime values	Yes - register contents, memory	No - must infer from code
Needs hardware	Yes - Pico 2 must be connected	No - works offline
Shows code flow	Step-by-step execution	Full graph visualization
Best for	Watching what happens	Understanding structure
Thumb bit handling	Shows with +1 (0x1000015d)	Shows actual addr (0x1000015c)

Ghidra Tips for Boot Analysis

1. **Rename functions** - Right-click and rename unclear labels for future reference
2. **Add comments** - Press `;` to add inline comments explaining code
3. **Set data types** - Help Ghidra understand structures like the vector table
4. **Use bookmarks** - Mark important locations with **Ctrl+D**

Part 16: Summary and Review

The Complete Boot Sequence

```
+-----------------------------------------------------------------+
|  1. POWER ON                                                    |
|     Cortex-M33 begins at 0x00000000 (bootrom)                   |
+-----------------------------------------------------------------+
|  2. BOOTROM                                                     |
|     - Initializes hardware                                      |
|     - Configures flash XIP (no separate boot2 on RP2350)        |
|     - Finds IMAGE_DEF within first 4 kB of flash image          |
+-----------------------------------------------------------------+
|  3. VECTOR TABLE (0x10000000)                                   |
|     - Reads SP from offset 0x00 -> 0x20082000                    |
|     - Reads Reset Handler from offset 0x04 -> 0x1000015d         |
+-----------------------------------------------------------------+
|  4. RESET HANDLER (0x1000015c)                                  |
|     - Checks CPUID (Core 0 continues, Core 1 waits)             |
|     - Copies .data from flash to RAM                            |
|     - Zeros .bss section                                        |
+-----------------------------------------------------------------+
|  5. PLATFORM ENTRY (0x10000186)                                 |
|     - Calls runtime_init()                                      |
|     - Calls main()                                              |
|     - Calls exit() when main returns                            |
+-----------------------------------------------------------------+
|  6. YOUR CODE RUNS!                                             |
|     main() at 0x10000234                                        |
+-----------------------------------------------------------------+
```

Key Addresses to Remember

Address	What's There
`0x00000000`	Bootrom (32KB, read-only)
`0x10000000`	Vector table / XIP flash start
`0x1000015c`	Reset handler (`_reset_handler`)
`0x10000234`	Your `main()` function
`0x20000000`	Start of RAM
`0x20082000`	Initial stack pointer (top of SCRATCH_Y)
`0xd0000000`	SIO base (CPUID register)

Weeks 1-2 Concepts We Applied

Previous Concept	How We Used It This Week
Memory Layout (Flash/RAM)	Understood why data must be copied from flash to RAM
GDB `x` command	Examined vector table, reset handler, and boot code
Breakpoints (`b`)	Set breakpoints to trace the boot sequence
Thumb Mode Addresses	Recognized LSB=1 means Thumb code in vector table
Stack Pointer	Saw how SP is initialized from the vector table
Ghidra Analysis	Used decompiler to understand boot flow

GDB Commands Reference

Command	What It Does	New/Review
`x/Nx ADDRESS`	Examine N hex values at ADDRESS	Review
`x/Ni ADDRESS`	Examine N instructions at ADDRESS	Review
`b main`	Set breakpoint at main function	Review
`b *ADDRESS`	Set breakpoint at exact address	Review
`si`	Step one instruction	Review
`c`	Continue execution	Review
`info registers`	Show all register values	Review
`monitor reset halt`	Reset and halt the target	Review

Key Concepts

Concept	Definition
Bootrom	32KB factory-programmed ROM that initializes the chip
Vector Table	List of addresses for SP and exception handlers
XIP	Execute In Place - running code directly from flash
Thumb Mode	ARM's compact instruction set (LSB=1 in addresses)
BSS	Section for uninitialized globals (must be zeroed)
crt0.S	C Runtime startup assembly file
Reset Handler	First function called after power-on/reset
CPUID	Register identifying which CPU core is executing

Ghidra Actions We Used

Action	How to Access	Purpose
Go To Address	Navigation -> Go To...	Jump to specific memory address
Show References	Right-click -> References -> Show References to	Find all callers of a function
Function Call Graph	Window -> Function Call Graph	Visualize call flow
Add Comment	Press `;`	Document your analysis
Rename Symbol	Right-click -> Rename	Give meaningful names to functions

---

Key Takeaways

Building on Weeks 1-2

1. **GDB skills compound** - The `x`, `b`, `si`, and `disas` commands you learned in Weeks 1-2 are essential for understanding the boot process. Each week adds new applications for the same core skills.
2. **Memory layout is fundamental** - Understanding flash vs RAM from Week 2 explains why startup code must copy data and zero BSS.
3. **Ghidra complements GDB** - Dynamic analysis (GDB) shows what happens at runtime; static analysis (Ghidra) reveals the overall structure. Use both together!

New Concepts This Week

4. **The boot process is deterministic** - Every RP2350 boots the same way, and understanding this helps you debug startup problems.

5. **The bootrom can't be changed** - It's burned into silicon. Security features depend on this immutability.
6. **The vector table is critical** - It tells the CPU where to start and how to handle errors.
7. **Thumb mode uses the LSB** - Address `0x1000015d` means "run Thumb code at `0x1000015c`".
8. **Startup code does essential work** - Copying data, zeroing BSS, and initializing the runtime all happen before `main()`.
9. **Only Core 0 runs startup** - Core 1 waits in the bootrom until explicitly started.

Security Implications

How Boot Sequence Knowledge Applies to Security

Understanding the boot process is critical for both attackers and defenders. Knowledge of how the RP2350 boots reveals potential attack vectors and defense strategies.

Attack Scenarios

Scenario	Attack	Boot Process Knowledge Required
Firmware Replacement	Replace the entire flash image with malicious firmware	Understanding IMAGE_DEF structure and how bootrom validates firmware
Vector Table Hijacking	Modify the reset handler address to point to malicious code	Knowing the vector table location at `0x10000000`
Bootrom Exploitation	Find bugs in the immutable bootrom to bypass security	Understanding bootrom behavior and sequence
Debug Port Attack	Use SWD/JTAG to dump firmware or inject code	Knowledge of how to halt and examine the boot process
Startup Code Modification	Change how data is copied or BSS is cleared	Understanding crt0 and runtime_init sequences

Real-World Applications

Industrial Control Systems:

- An attacker with physical access could replace firmware to hide malicious behavior
- Understanding the boot sequence helps identify the earliest point where security checks can be added

IoT Devices:

- Compromised boot code could establish backdoors before the main application runs
- Secure boot implementations verify the vector table and reset handler integrity

Medical Devices:

- Boot-time attacks could modify critical safety parameters before device operation
- Understanding initialization helps implement tamper detection

Defense Strategies

1. Secure Boot Implementation

```
+-----------------------------------------------------+
|  SECURE BOOT FLOW                                   |
+-----------------------------------------------------+
|  Bootrom (immutable)                                |
|    ↓                                                |
|  Verify IMAGE_DEF signature                         |
|    ↓                                                |
|  Verify application image signature                 |
|    ↓                                                |
|  If all valid: Jump to reset handler                |
|  If any invalid: Refuse to boot                     |
+-----------------------------------------------------+
```

Implementation: Use cryptographic signatures to verify each boot stage before execution.

2. Debug Port Protection

- **Production devices:** Permanently disable SWD/JTAG in final products
- **Debug authentication:** Require cryptographic challenge-response before allowing debug access
- **Fuses:** Blow hardware fuses to disable debug ports permanently

3. Flash Protection

- **Read protection:** Enable flash read protection to prevent dumping firmware
- **Write protection:** Make critical boot sectors write-protected after initial programming
- **Encrypted storage:** Store firmware encrypted in flash

4. Memory Protection Unit (MPU)

Configure the Cortex-M33's MPU to:

- Mark code regions as execute-only (no reading code as data)
- Separate privileged and unprivileged memory regions
- Prevent execution from RAM regions (defend against code injection)

5. Boot-Time Integrity Checks

```
// Early in reset handler or runtime_init
void verify_boot_integrity(void) {
    // Check vector table hasn't been modified
    uint32_t vector_table_checksum = calculate_checksum(0x10000000, VECTOR_TABLE_SIZE);
    if (vector_table_checksum != EXPECTED_CHECKSUM) {
        // Vector table tampered - refuse to boot
        secure_halt();
    }

    // Check critical data structures
    // Verify stack pointer is in valid range
    // etc.
}
```

6. Anti-Tampering Hardware

- **Tamper detection:** Sensors that detect case opening or voltage glitching
- **Response actions:** Erase sensitive keys, refuse to boot, or alert monitoring systems

- **Secure elements:** Store cryptographic keys in separate tamper-resistant chips

Lessons for Defenders

1. **The bootrom is your trust anchor** - Its immutability makes it the foundation of security. RP2350's secure boot features leverage this.
2. **Early is critical** - Security checks in the reset handler or runtime_init run before any application code, making them harder to bypass.
3. **Defense in depth** - Multiple layers (hardware fuses, encrypted storage, secure boot, MPU) make attacks much harder.
4. **Physical access = game over** - If an attacker can connect a debug probe, they can potentially compromise the device. Physical security matters!
5. **Know your boot sequence** - Understanding exactly what runs when helps you identify where to add security checks and what assets need protection.

Security Research Value

For security researchers and penetration testers, boot sequence analysis helps:

- **Find vulnerabilities:** Many security bugs exist in startup code that runs before normal security checks
- **Develop exploits:** Understanding memory layout and initialization is essential for exploit development
- **Assess attack surface:** Knowing what's accessible at boot time reveals potential attack vectors
- **Build better defenses:** You can't defend what you don't understand

> **"To know your enemy, you must become your enemy."** - Sun Tzu

Understanding how an attacker would analyze and exploit the boot sequence is essential for building robust defenses.

Glossary

New Terms This Week

Term	Definition
Bootrom	Factory-programmed ROM containing first-stage bootloader
BSS	Block Started by Symbol - section for uninitialized global variables
CPUID	Register that identifies which CPU core is executing
crt0	C Runtime Zero - the startup code that runs before main
IMAGE_DEF	Structure that marks valid firmware for the bootrom
Linker Script	File that defines memory layout for the compiled program
Reset Handler	First function called after reset/power-on
Thumb Mode	Compact instruction encoding used by Cortex-M
Vector Table	Array of addresses for stack pointer and exception handlers
VTOR	Vector Table Offset Register - tells CPU where to find the vector table
XIP	Execute In Place - running code directly from flash memory

Review Terms from Weeks 1-2

Term	Definition	How We Used It
Breakpoint	Marker that pauses program execution	Set at reset handler and main
Register	Fast storage inside the processor	Watched SP, LR, PC during boot
Stack Pointer	Register pointing to top of stack	Saw initial value in vector table
Flash Memory	Read-only storage for code	Contains vector table and boot code
SRAM	Read-write memory for data	Where stack and variables live

Additional Resources

RP2350 Datasheet

For more details on the boot process, see Chapter 5 of the RP2350 Datasheet:
https://datasheets.raspberrypi.com/rp2350/rp2350-datasheet.pdf

Pico SDK Source Code

The startup code lives in:

- `crt0.S` - Main startup assembly (vector table at `.section .vectors`, reset handler, data copy, BSS clear, platform_entry)
- `memmap_default.ld` - Default linker script (section ordering: `.vectors` -> `.binary_info_header` -> `.embedded_block` -> `.reset`)
- `embedded_start_block.inc.S` - IMAGE_DEF block (replaces RP2040's `boot2_generic_03h.S`)

> **Note:** The RP2040 used a `boot2_generic_03h.S` second-stage bootloader occupying the first 256 bytes of flash. The RP2350 eliminated this; the bootrom handles flash XIP setup directly. The SDK still includes a `boot2` mechanism for compatibility, but it is **not** placed at flash address 0 - it is embedded in the data copy table and executed from the stack during startup.

Bootrom Source

The bootrom source is available at: https://github.com/raspberrypi/pico-bootrom-rp2350

Part 17: Proving the Boot Sequence with objdump

Everything we have learned about the boot sequence can be proven directly from the compiled ELF binary using `arm-none-eabi-objdump`. The bootrom is not in your ELF (it is mask ROM burned into the chip at `0x00000000`), but everything your firmware provides - the vector table, the IMAGE_DEF, and the reset handler - lives in your ELF starting at `0x10000000`.

Step 1: List All Sections

```
arm-none-eabi-objdump -h build/0x0001_hello-world.elf
```

Expected output (key sections):

```
Idx Name          Size      VMA       LMA
  0 .text         000019cc  10000000  10000000
  3 .binary_info  0000002c  10001b20  10001b20
  4 .ram_vector_table 00000110  20000000  20000000
  6 .data         0000019c  20000110  10001b4c
```

> Tip: The Pico SDK merges `.vectors`, `.embedded_block`, and `.reset` all into `.text` at `0x10000000`. They are not separate named ELF sections - they are sub-regions inside `.text`.

Step 2: Dump the First 0x150 Bytes of Flash - One Command, Zero Skips

```
arm-none-eabi-objdump -s --start-address=0x10000000 --stop-address=0x10000150 build/0x0001_hello-
world.elf
```

Raw output from that command:

```
10000000 00200820 5d010010 1b010010 1d010010  . . ]...........
10000010 11010010 11010010 11010010 11010010  ................
10000020 11010010 11010010 11010010 11010010  ................
10000030 11010010 11010010 11010010 11010010  ................
10000040 11010010 11010010 11010010 11010010  ................
10000050 11010010 11010010 11010010 11010010  ................
10000060 11010010 11010010 11010010 11010010  ................
10000070 11010010 11010010 11010010 11010010  ................
10000080 11010010 11010010 11010010 11010010  ................
10000090 11010010 11010010 11010010 11010010  ................
100000a0 11010010 11010010 11010010 11010010  ................
100000b0 11010010 11010010 11010010 11010010  ................
100000c0 11010010 11010010 11010010 11010010  ................
100000d0 11010010 11010010 11010010 11010010  ................
100000e0 11010010 11010010 11010010 11010010  ................
100000f0 11010010 11010010 11010010 11010010  ................
10000100 11010010 11010010 11010010 11010010  ................
10000110 eff30580 103800be 00be00be 00be00be  .....8..........
10000120 00be00be f2eb8871 201b0010 4c1b0010  .......q ...L...
10000130 a0010010 90a31ae7 d3deffff 42012110  ............B.!.
10000140 ff010000 b01b0000 793512ab           ........y5..
```

Every address annotated, no skips:

0x10000000 - Vector Table, Mandatory Entries

Address	Raw Bytes (LE)	Decoded	What it is
`0x10000000`	`00 20 08 20`	`0x20082000`	**Initial SP** - top of SCRATCH_Y RAM. Bootrom loads MSP from here before doing anything else.
`0x10000004`	`5d 01 00 10`	`0x1000015d`	**Reset_Handler** address with Thumb bit set. Strip bit 0 -> real address `0x1000015c` . Bootrom jumps here.
`0x10000008`	`1b 01 00 10`	`0x1000011b`	**NMI** handler address (Thumb, -> `0x1000011a`).
`0x1000000c`	`1d 01 00 10`	`0x1000011d`	**HardFault** handler address (Thumb, -> `0x1000011c`).

0x10000010-0x1000010f - Vector Table, IRQ Slots (all 52 external IRQs)

```
10000010 11010010 11010010 ...(repeats through 0x1000010f)...
```

Every 4-byte word here is `11 01 00 10` = pointer `0x10000111` . That is the **default IRQ handler** address with Thumb bit set (-> `0x10000110`). The RP2350 Cortex-M33 has 16 system vectors (offsets 0x00-0x3f) plus up to 52 external IRQ vectors (offsets 0x40-0xff = addresses `0x10000040` - `0x1000010f`). Every IRQ the application does not register gets this default handler pointer. This block is 240 bytes (`0x10000010` to `0x1000010f`) of nothing but that one repeated pointer.

0x10000110-0x10000127 - Default IRQ Handler Code

```
10000110 eff30580 103800be 00be00be 00be00be
10000120 00be00be f2eb8871
```

Address	Bytes	ARM Thumb-2 Instruction	What it does
`0x10000110`	`ef f3 05 80`	`MRS r0, IPSR`	Read the Interrupt Program Status Register into r0. The low 9 bits = the active vector number.
`0x10000114`	`10 38`	`SUBS r0, #16`	Vector 16 = IRQ0, so subtract 16 to convert vector number -> IRQ index.
`0x10000116`	`00 be`	`BKPT #0`	Software breakpoint. If a debugger is attached, it stops here and you can inspect r0 to see which IRQ fired. If no debugger is attached, the CPU enters a fault loop and the chip hangs.
`0x10000118` - `0x10000127`	`00 be` *12	`BKPT #0` repeating	Alignment padding to the next 4-byte boundary.

This is the **entire** default IRQ handler. It is intentionally minimal: if your code triggers an IRQ you did not register, it crashes visibly instead of silently.

0x10000128-0x1000013b - Binary Info Pointer Table

```
10000120              201b0010 4c1b0010
10000130 a0010010 90a31ae7
```

Address	Bytes (LE)	Decoded Value	What it is
`0x10000128`	`20 1b 00 10`	`0x10001b20`	Pointer to **start** of `.binary_info` data section in flash.
`0x1000012c`	`4c 1b 00 10`	`0x10001b4c`	Pointer to **end** of `.binary_info` data section in flash.
`0x10000130`	`a0 01 00 10`	`0x100001a0`	Pointer to `binary_info_callback` function.
`0x10000134`	`90 a3 1a e7`	(magic marker)	`BINARY_INFO_MARKER_END` - marks the end of this pointer table.

`picotool` reads this table to extract the program name, version string, URL, and GPIO pin map from any compiled binary without running it.

0x10000138-0x1000014c - IMAGE_DEF Block (this build)

```
10000130 a0010010 90a31ae7 d3deffff 42012110
10000140 ff010000 b01b0000 793512ab 4ff00000
```

Address	Bytes	What it is
`0x10000138`	`d3 de ff ff`	`PICOBIN_BLOCK_MARKER_START` - the bootrom scans flash for this exact 4-byte sequence to locate the IMAGE_DEF.
`0x1000013c`	`42 01 21 10`	IMAGE_DEF content (image type, flags, version).
`0x10000140`	`ff 01 00 00`	IMAGE_DEF content (continuation).
`0x10000144`	`b0 1b 00 00`	IMAGE_DEF content (continuation).
`0x10000148`	`79 35 12 ab`	`PICOBIN_BLOCK_MARKER_END` - bootrom stops scanning here.

The IMAGE_DEF sits at `0x10000138` - `0x1000014b` in this build, well within the 4 KB scan window the bootrom uses (Datasheet 5.9.5, p. 429).

Full Flash Map: 0x10000000-0x1000015c

```
0x10000000–0x1000000f  Vector Table: mandatory entries (SP, Reset, NMI, HardFault)
0x10000010–0x1000010f  Vector Table: 52 external IRQ slots -> all point to default handler
0x10000110–0x10000127  Default IRQ handler code (MRS / SUBS / BKPT)
0x10000128–0x10000137  Binary info pointer table (start / end / callback / magic end)
0x10000138–0x1000014b  IMAGE_DEF block (d3 de ff ff ... 79 35 12 ab)
0x10000150–0x1000015b  (padding / alignment)
0x1000015c             Reset_Handler (_reset_handler in crt0.S) ← bootrom jumps here
```

Step 4: Confirmed Boot Sequence (proven from ELF)

```
+-----------------------------------------------------------------+
|  PROVEN BOOT SEQUENCE (0x0001_hello-world)                      |
+-----------------------------------------------------------------+
|  1. Bootrom reads 0x10000000                                    |
|     -> SP  = 0x20082000  (offset +0x00 of vector table)          |
|     -> RST = 0x1000015d  (offset +0x04, Thumb -> 0x1000015c)     |
+-----------------------------------------------------------------+
|  2. Bootrom scans first 4 kB for IMAGE_DEF                      |
|     -> Found at 0x10000138 (this build)                          |
|     -> Start marker: d3 de ff ff                                 |
|     -> End marker:   79 35 12 ab                                 |
+-----------------------------------------------------------------+
|  3. Bootrom jumps to reset handler at 0x1000015c                |
|     -> _reset_handler (crt0.S) runs                              |
|     -> Checks CPUID - Core 1 sent back to bootrom                |
|     -> Core 0: .data copied, .bss zeroed, platform_entry called  |
+-----------------------------------------------------------------+
|  4. platform_entry calls runtime_init -> main -> exit             |
+-----------------------------------------------------------------+
```

> **Datasheet References:**
>
> - 5.1.5.1 (p. 357): Block markers `0xffffded3` (start) and `0xab123579` (end)
> - 5.9.5 (p. 429): IMAGE_DEF must appear within first 4 kB of flash image
> - 5.9.5.1 (p. 429): Bootrom enters via reset handler at vector table offset +4

Remember: Understanding the boot process is fundamental to embedded systems work. Whether you're debugging a system that won't start, reverse engineering firmware, or building secure boot chains, this knowledge is essential!

Happy exploring!