

# Embedded Systems Reverse Engineering

---

// WEEK 03

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

---

**George Mason University**

RP2350 // ARM Cortex-M33

# RP2350 Boot Sequence

Power-On to main() – 5 Steps

## STEP 1 Power On

Cortex-M33 wakes, execution at 0x00000000 (Bootrom)



## STEP 2 Bootrom Executes

32KB on-chip ROM – finds IMAGE\_DEF at 0x10000000



## STEP 3 Flash XIP Setup (bootrom-managed)

Bootrom configures flash interface & XIP (no boot2 on RP2350)



## STEP 4 Vector Table & Reset Handler

Reads SP from offset 0x00 -> 0x20082000

Reads Reset Handler from 0x04 -> 0x1000015d



## STEP 5 C Runtime Startup (crt0.S)

Copy .data from flash -> RAM

Zero .bss section

Call runtime\_init() -> main()

## Key Insight

Your main() is the LAST thing to run.

All 5 steps must complete first!

# The Bootrom

32KB Factory-Programmed ROM — Where It All Begins

## Bootrom Properties

Size	32 KB
Location	0x00000000
Modifiable?	NO — mask ROM
Purpose	Boot the chip

Burned into silicon at factory  
Like BIOS in your computer

## What It Does

1. Initialize hardware
2. Check boot sources
3. Validate IMAGE\_DEF
4. Configure flash
5. Jump to your code

## IMAGE\_DEF — Magic Markers

Bootrom looks for these to validate firmware

Start Marker	0xFFFFDED3	"I'm a valid Pico binary!"
End Marker	0xAB123579	"End of header block"

Bootrom reads flash at 0x10000000,  
finds these markers, then boots.

# XIP — Execute In Place

Run Code Directly from Flash — No Copy Needed

## Book Analogy

### Without XIP

Photocopy every page, read copy

### With XIP

Read directly from the book!

## Why Use XIP?

### Saves RAM

Code stays in flash

### Faster Boot

No bulk copy needed

### Simpler

Less memory mgmt

## XIP Flash Region at 0x10000000

### Vector Table

0x10000000

SP at offset 0x00 | Reset Handler at offset 0x04 | Exception handlers...

### Your Code

0x100001xx

\_reset\_handler | main() | other functions

### Read-Only Data

0x10001xxx

Strings like "hello, world" | constant values

CPU fetches instructions directly  
from flash via XIP cache.

# The Vector Table

CPU's Instruction Manual at 0x10000000

## Vector Table Layout

Offset	Address	Value	Meaning
0x00	0x10000000	0x20082000	Initial SP
0x04	0x10000004	0x1000015D	Reset Handler
0x08	0x10000008	0x1000011B	NMI Handler
0x0C	0x1000000C	0x1000011D	HardFault Handler

GDB: x/4x 0x10000000

## On Power-On

- CPU reads SP from 0x00
- Sets SP = 0x20082000
- Reads Reset from 0x04
- Jumps to 0x1000015C

## Default Handlers

NMI, HardFault, SVCall,  
PendSV, SysTick all use:

**bkpt 0x0000**    <- stops debugger

# Thumb Mode Addressing

Why Addresses End in Odd Numbers

## The LSB Rule

ARM Cortex-M uses the Least Significant Bit (LSB) to indicate instruction mode:

**LSB = 1 (odd)**

Thumb mode

**LSB = 0 (even)**

ARM mode

## Reset Handler Example

Vector table stores: **0x1000015D**

Actual code address: **0x1000015C**

The +1 means: **"Use Thumb mode"**

## GDB Shows

**0x1000015D**

with Thumb bit

Vector table raw value

## Ghidra Shows

**0x1000015C**

actual address

Real instruction location

**Both are correct — just displayed differently!**

# Linker Script Memory Map

memmap\_default.ld – Where Everything Lives

## Memory Regions

Flash (XIP)	0x10000000	varies	Your code (read-only)
RAM	0x20000000	512 KB	Main RAM (r/w)
SCRATCH_X	0x20080000	4 KB	Core 0 scratch (HW: SRAM8)
SCRATCH_Y	0x20081000	4 KB	Core 0 stack! (HW: SRAM9)

## Stack Pointer Calculation

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

ORIGIN	0x20081000	+ LENGTH	0x1000	(4 KB)
= __StackTop = 0x20082000				<- matches vector table!

# Reset Handler – 4 Phases

\_reset\_handler at 0x1000015C

## Phase 1: Core Check

0x1000015C – 0x10000168

```
mov r0, #0xD0000000
```

Read CPUID -> Core 0 continues

## Phase 2: Data Copy

0x1000016A – 0x10000176

```
ldmia r4!, {r1,r2,r3}
```

Copy .data from flash -> RAM

## Phase 3: BSS Clear

0x10000178 – 0x10000184

```
stmia r1!, {r0}    r0 = 0
```

Zero all uninitialized globals

## Phase 4: Platform Entry

0x10000186+

```
blx r1    -> main()
```

runtime\_init -> main -> exit

## Execution Flow

### Core Check

CPUID == 0?

->

### Data Copy

flash -> RAM

->

### BSS Clear

zero globals

->

### Platform Entry

-> main()!

## Why check cores?

RP2350 has 2 cores.

Only Core 0 runs startup.

Core 1 returns to bootrom and waits.



# Data Copy & BSS Clear

Initializing RAM Before main() Can Run

## Phase 2: Data Copy

Copy initialized variables flash -> RAM

C code:

```
int counter = 42;
```

Value 42 stored in flash  
but variables live in RAM!

Flash

->

RAM

data\_cpy\_table has entries:

```
src: 0x10001B4C (flash)
dst: 0x20000110 (RAM)
```

## Phase 3: BSS Clear

Zero uninitialized global variables

C code:

```
int my_counter;
```

C standard requires  
this to start at zero.

```
r1 = BSS start
r2 = BSS end
r0 = 0
```

```
Loop: store 0, advance r1
Until r1 == r2 -> done!
```

## Key Assembly Instructions

```
ldmia r4!, {r1,r2,r3}
```

Load source, dest, end from table

```
bl data_cpy
```

Copy word-by-word until done

```
movs r0, #0
```

Load zero into r0

```
stmia r1!, {r0}
```

Store zero, advance pointer

# Platform Entry -> main()

The Final Step – 3 Function Calls at 0x10000186

## platform\_entry Assembly

**0x10000186**      `ldr r1, [DAT]`      -> load runtime\_init addr

**0x10000188**      `blx r1`      -> call runtime\_init()

**0x1000018C**      `blx r1`      -> call main()

**0x10000190**      `blx r1`      -> call exit()

## Call Sequence

**runtime\_init()**

SDK setup

->

**main()**

YOUR CODE

->

**exit()**

cleanup

## runtime\_init()

Initializes SDK systems:

- Clock configuration
- GPIO setup
- C++ constructor calls
- Peripheral initialization

## After main() Returns

exit() handles cleanup.

Then:

**bkpt 0x0000**      <- infinite halt

Should never be reached!

# Secure Boot & Attack Vectors

Why Boot Sequence Knowledge Matters for Security

## Attack Scenarios

### Firmware Replacement

Replace flash with malicious code

### Vector Table Hijack

Modify reset handler address

### Debug Port Attack

SWD/JTAG to dump or inject code

### Startup Code Modification

Change crt0 data copy / BSS init

**Physical access = game over**

## Defense Strategies

1. Secure Boot

2. Debug Port Lock

3. Flash Read Protect

4. MPU Configuration

5. Integrity Checks

**Defense in depth!**

## Secure Boot Chain



Each stage cryptographically verifies the next before handing off control.

Bootrom = trust anchor (can't be changed)