

EMBEDDED SYSTEMS REVERSE ENGINEERING

Week 1

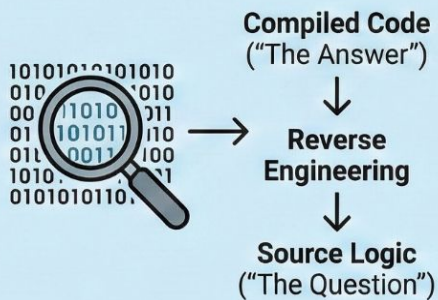
Introduction to Embedded Systems Reverse Engineering

Ethics, Scoping, and Core Concepts



Course Goals & The Mindset

What Are We Doing?



Working backward from compiled, running code to understand the original source logic.

Why do we do it?



Understanding: Learning complex hardware/software interactions without documentation.



Security Auditing: Finding vulnerabilities before bad actors do.



Interoperability: Figuring out how to make new software talk to old hardware.

The Ethical Hacker's Mindset



Stay within scope: Only analyze targets you have permission to touch.



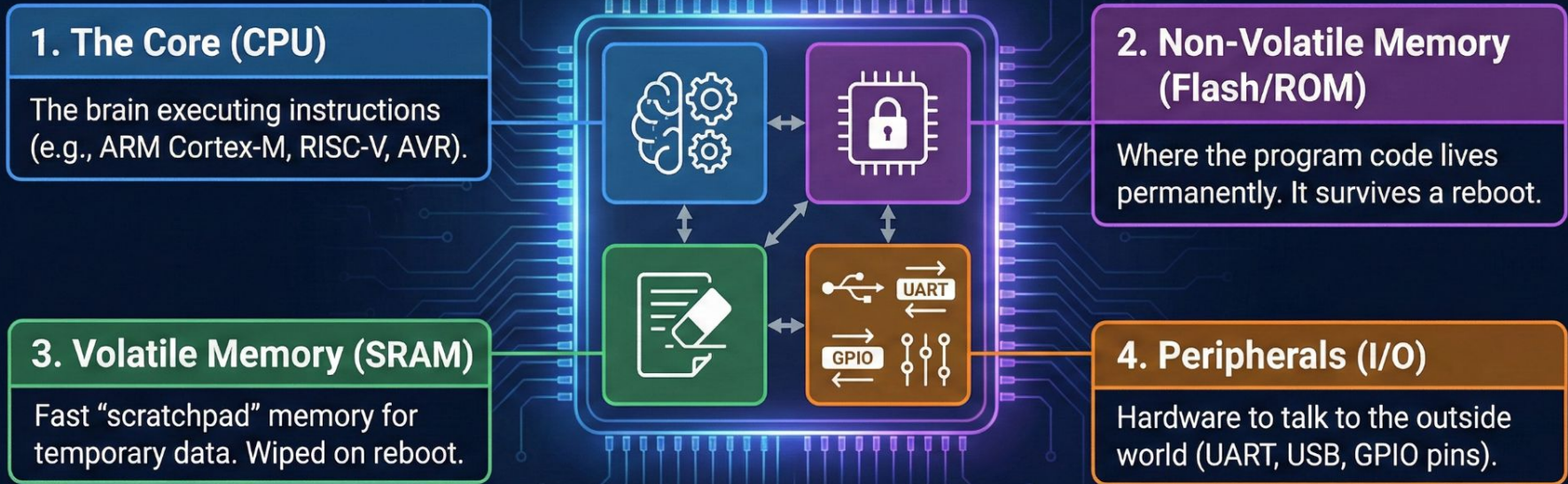
Do no harm: Avoid bricking hardware or disrupting live systems.



Curiosity driven: It is a detective game of digital clues.

The Target: What is a Microcontroller (MCU)?

A Computer on a Single Chip



Unlike a desktop, an MCU integrates everything.

i Note: In this course, our example hardware is the RP2350, which uses ARM Cortex-M33 cores.

The Two Paths of Analysis

RE is rarely done using just one method. We combine two approaches:

1. Static Analysis ("Dead" Code)



Analogy: Reading a blueprint of a building.



Action: Examining the binary file without running it.



Tools: Ghidra, IDA Pro, Binary Ninja.



Goal: To see the "big picture," understand program structure, find strings, and identify key functions.

2. Dynamic Analysis ("Live" Code)



Analogy: Walking through the building, opening doors, and testing light switches.



Action: Running the code on actual hardware while attached to a debugger.



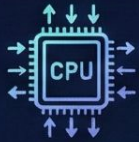
Tools: GDB (GNU Debugger), JTAG/SWD hardware probes (like OpenOCD).



Goal: To watch register values change, see exactly which path execution takes, and examine memory at a specific moment in time.

The CPU's Perspective: Registers

Registers are the fastest storage locations in a computer, located directly **inside the CPU core**. The CPU cannot do math directly on memory; it must load data into registers first.



General Purpose Registers

- Used for holding temporary data during calculations, passing arguments to functions, and holding return values.

- ARM Cortex-M example:** *r0 through r12.*

R0	Data
R1	Data
R2	0
R3	Data
R4	Data
R5	Data
R6	Data
R7	Data
R8	Data
R9	Data
R9	Data
R10	Data
R11	Data
R12	Data



The Special Registers (The "State" of the Machine)



Program Counter (PC):

The most critical register. It holds the memory address of the *_next_* instruction to be executed.



Stack Pointer (SP):

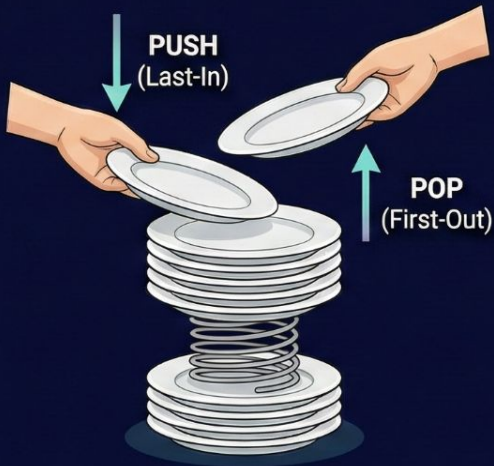
Points to the current "top" location of the temporary stack memory.



The Stack Concept

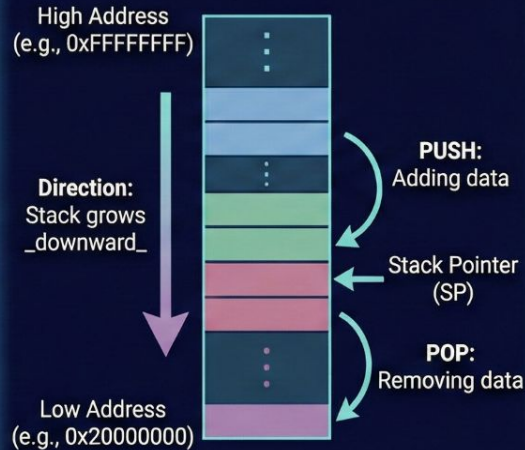
The Stack is a fundamental concept in almost all computer architectures used for managing function calls and temporary data.

LIFO (Last-In, First-Out)



Think of a stack of plates in a cafeteria. The last plate you put on top is the first one you take off.

How it works in Embedded Memory



PUSH: SP decreases, data saved.
POP: Data read, SP increases.

What is it used for?

```
int x,  
char y  
}
```

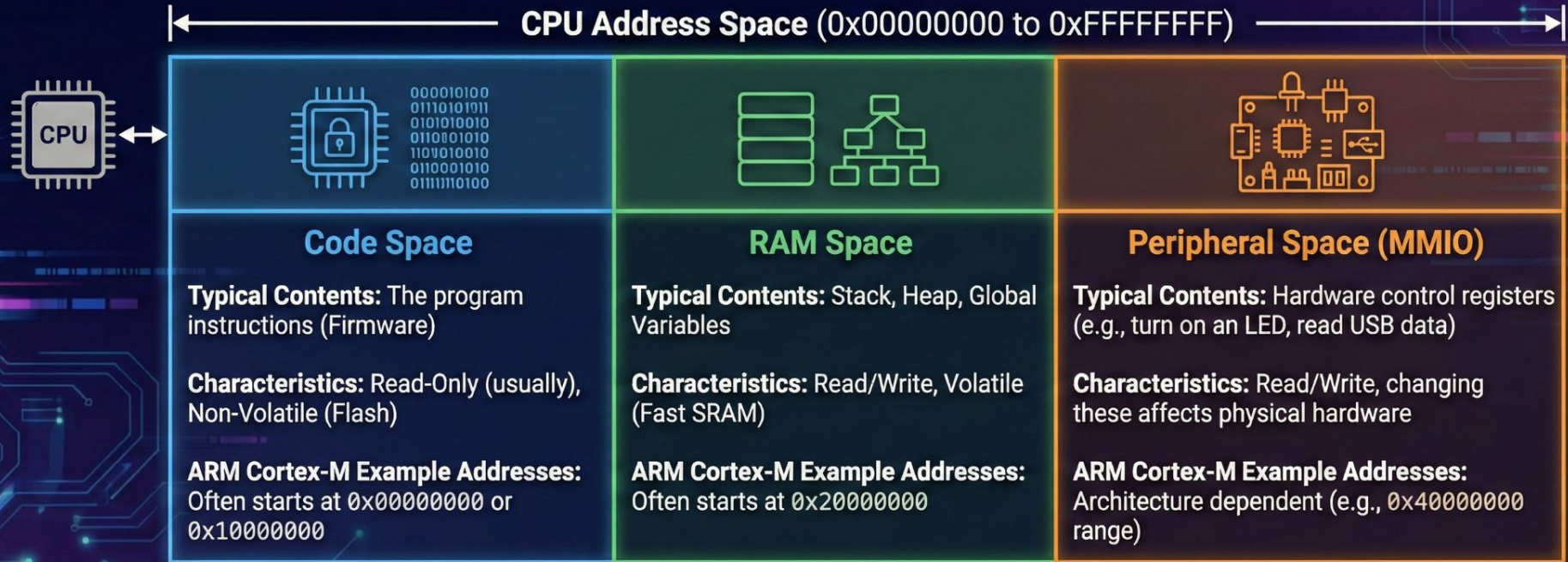
- Saving local variables inside a function.



- Remembering where to return to after a function call finishes.

The Memory Map Landscape

In embedded systems, “Memory” doesn’t just mean RAM. Everything the CPU can talk to has a memory address. This is called a “Memory Map”.



Tip: In RE, knowing *_where_* data is tells you a lot about *_what_* it is.



From C to Machine Code (The Reverse Path)

As reverse engineers, we are walking backward through the compilation process.

Forward Process (Development):

1. C Source Code:

High-level logic
(`printf("hello");`)

2. Compiler:



3. Assembler:



Machine Code (Hex/ Binary):

0x1000023c: 01 f0 de f9...
0x1000023c: 01 f0 de f9...
...



Reverse Process (Our Job):

3. Decompiler (Ghidra):



Attempts to turn
Assembly back into
C-like pseudocode.

2. Disassembler (Ghidra/GDB):



Turns hex back into
readable Assembly
(`b1 __wrap_puts`)

1. Machine Code (Binary):

We start with the raw
`0x1000023c: 01 f0 de f9``

Tool Theory: Static Analysis (Ghidra)

We use tools like Ghidra to perform static analysis on binary files (like .elf or .bin).

Key Views:



The Disassembly View (Listing):

```
Disassembly x
Clearword_Local1
Row  CLASSIC
0x00100000 35 48 89 25  push rbp
0x00100000 36 48 89 25  mov rbp, rbp
0x00100000 30 54 45 43 96  mov rcx=sscanf(stdin)
0x00100000 80 85  push rcx
0x00100000 55 05 03  mov rcx, rsp
0x00100000 70 23 23 84  push rcx=rcx, rbp
0x00100000 33 33 33 28  mov rcx=rcx[16], rcx[16]
0x00100000 40 88  mov rbp, rbp
0x00100000 89 72  push rcx
0x00100000 58 48 03 99 65  mov ecx=rcx[16], rcx[16]
0x00100000 32 40 03  call rcx=rcx
0x00100000 33 13 08  lli rcx=rcx[16], rcx[16]
0x00100000 30 39  wui rcx
0x00100000 32 40 03  mov rbp, rbp
0x00100000 37 48 03  mov rbp, rbp
0x00100000 41 48 03  mov rcx, rcx
```

Shows the exact assembly instructions exactly as they exist in the binary. This is the “truth,” but it’s hard to read quickly.



The Decompiler View:

```
Decompiler x
1
2 void main() {
3     int i;
4     for(i=0; i<10; i++) {
5         int j;
6         int i = 0x 10;
7         int i = 0x 10;
8         ... 0;
9     }
10 }
11 }
```

A “best guess” reconstruction of C code. It is incredibly helpful for understanding high-level logic rapidly, but it can sometimes be wrong or misleading.

Symbols matter:



ELF Files:

Usually contain “symbols” (human readable names for functions like `main` or `stdio_init`). These are training wheels for RE.



Binary Files (.bin):

“Stripped” binaries with no symbols. Real-world embedded RE often looks like this, requiring you to recognize functions by their behavior, not their names.

Tool Theory: Dynamic Analysis (Debuggers/GDB)

We use debuggers like GDB to interact with the hardware while it runs. This is surgery on a living patient.

The Essential Debugger Verbs:



Breakpoint (b): “Stop execution immediately when the PC reaches this address.”



Continue (c): “Run at full speed until you hit a breakpoint.”



SI/NI

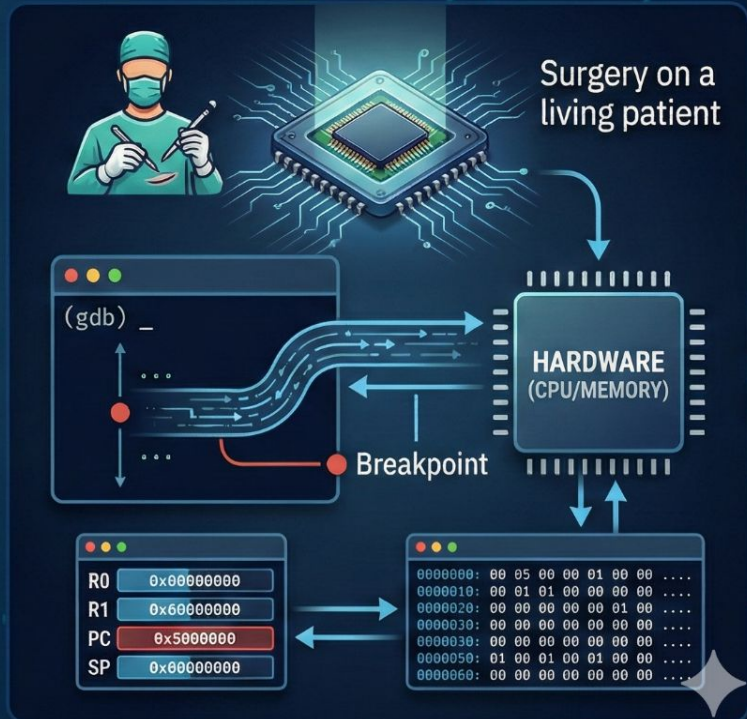
Step (si / ni): “Execute exactly one instruction and then pause again.”



Examine Registers (i r): “Show me the current state of the CPU brain right now.”

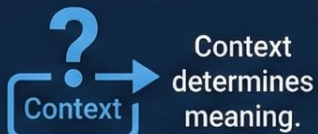


Examine Memory (x/...): “Show me what values are currently stored at this specific RAM or Flash address.”



Key Takeaways for Week 1

It's all just numbers:

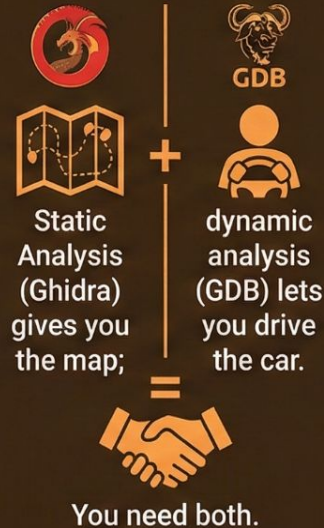


The PC is King:



Knowing where the Program Counter is pointing tells you exactly what the machine is about to do.

Combine your tools:



Assembly isn't magic:

```
MOV R0, #1  
ADD R1, R0, R2  
STR R1, [R3]  
STR R1, R3  
...
```

It's just a series of very small, very simple steps. Don't get overwhelmed by the whole program; focus on one instruction at a time.