
EMBEDDED HACKING

Reverse Engineering the Raspberry Pi Pico 2

Week 6: Static Variables in Embedded Systems: Debugging and Hacking Static Variables w/ GPIO Input Basics

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 tutorial, you will be able to:

- Understand the difference between regular (automatic) variables and static variables
 - Know where different types of variables are stored (stack vs static storage)
 - Configure GPIO pins as inputs and use internal pull-up resistors
 - Read button states using `gpio_get()` and control LEDs based on input
 - Use GDB to examine how the compiler handles static vs automatic variables
 - Identify compiler optimizations by stepping through assembly
 - Hack variable values and invert GPIO input/output logic using a hex editor
 - Convert patched binaries to UF2 format for flashing
-

Part 1: Understanding Static Variables

What is a Static Variable?

A **static variable** is a special kind of variable that “remembers” its value between function calls or loop iterations. Unlike regular variables that get created and destroyed each time, static variables **persist** for the entire lifetime of your program.

Think of it like this:

- **Regular variable:** Like writing on a whiteboard that gets erased after each class
- **Static variable:** Like writing in a notebook that you keep forever

+-----+ Regular vs Static Variables +-----+	
REGULAR (automatic):	
+-----+	
Loop 1: Create -> Set to 42 -> Increment to 43 -> Destroy	
Loop 2: Create -> Set to 42 -> Increment to 43 -> Destroy	
Loop 3: Create -> Set to 42 -> Increment to 43 -> Destroy	
Result: Always appears as 42!	
+-----+	
STATIC:	
+-----+	
Loop 1: Already exists -> Read 42 -> Increment -> Store 43	

```
| | Loop 2: Already exists -> Read 43 -> Increment -> Store 44
| | Loop 3: Already exists -> Read 44 -> Increment -> Store 45
| | Result: Keeps incrementing!
| +-----+
|
| +-----+
```

The static Keyword

In C, you declare a static variable by adding the `static` keyword:

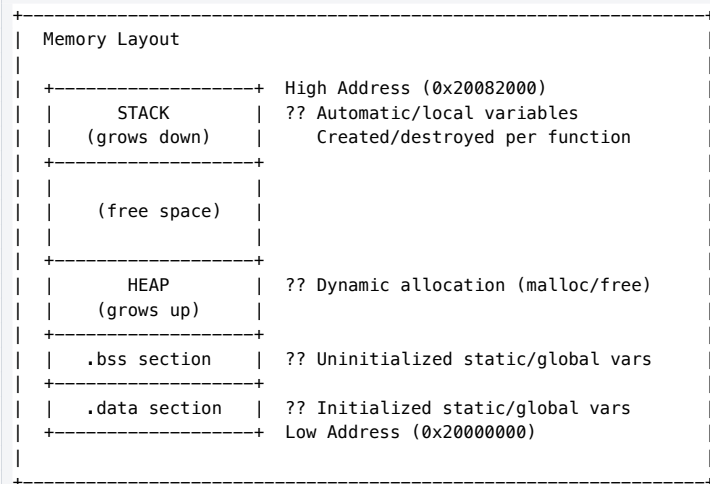
```
uint8_t regular_fav_num = 42;      // Regular - recreated each time
static uint8_t static_fav_num = 42; // Static - persists forever
```

Where Do Variables Live in Memory?

Different types of variables are stored in different memory locations:

Variable Type	Storage Location	Lifetime	Example
Automatic (local)	Stack	Until function/block ends	<code>uint8_t x = 5;</code>
Static	Static Storage	Entire program lifetime	<code>static uint8_t x = 5;</code>
Global	Static Storage	Entire program lifetime	<code>uint8_t x = 5;</code> (outside functions)
Dynamic (heap)	Heap	Until <code>free()</code> is called	<code>malloc(sizeof(int))</code>

Stack vs Static Storage vs Heap



Key Point: Static variables are NOT on the heap! They live in a fixed location in the `.data` section (if initialized) or `.bss` section (if uninitialized). This is different from heap memory which is dynamically allocated at runtime.

What Happens with Overflow?

Since `static_fav_num` is a `uint8_t` (unsigned 8-bit), it can only hold values 0-255. What happens when it reaches 255 and we add 1?

255 + 1 = 256... but that doesn't fit in 8 bits!
Binary: 11111111 + 1 = 100000000 (9 bits)
The 9th bit is lost, so we get: 00000000 = 0

This is called **overflow** or **wrap-around**. The value “wraps” back to 0 and starts counting again!

Part 2: Understanding GPIO Inputs

Input vs Output

So far, we’ve used GPIO pins as **outputs** to control LEDs. Now we’ll learn to use them as **inputs** to read button states!

```

+-----+
| GPIO Direction                                     |
|+-----+                                           |
| OUTPUT (what we've done before):                 |
|+-----+                                           |
| | Pico | | -----? LED                          |
| | GPIO 16 | | (We control the LED)                |
|+-----+                                           |
| INPUT (new this week):                           |
|+-----+                                           |
| | Pico | | ?----- Button                       |
| | GPIO 15 | | (We read the button state)          |
|+-----+                                           |
+-----+

```

The Floating Input Problem

When a GPIO pin is set as an input but nothing is connected, it’s called a **floating input**. The voltage on the pin is undefined and can randomly read as HIGH (1) or LOW (0) due to electrical noise.

```

+-----+
| Floating Input = Random Values!                   |
|+-----+                                           |
| GPIO Pin (no connection):                         |
| Reading 1: HIGH                                   |
| Reading 2: LOW                                    |
| Reading 3: HIGH                                   |
| Reading 4: HIGH                                   |
| Reading 5: LOW                                    |
| (Completely unpredictable!)                       |
|+-----+                                           |
+-----+

```

Pull-Up and Pull-Down Resistors

To solve the floating input problem, we use **pull resistors**:

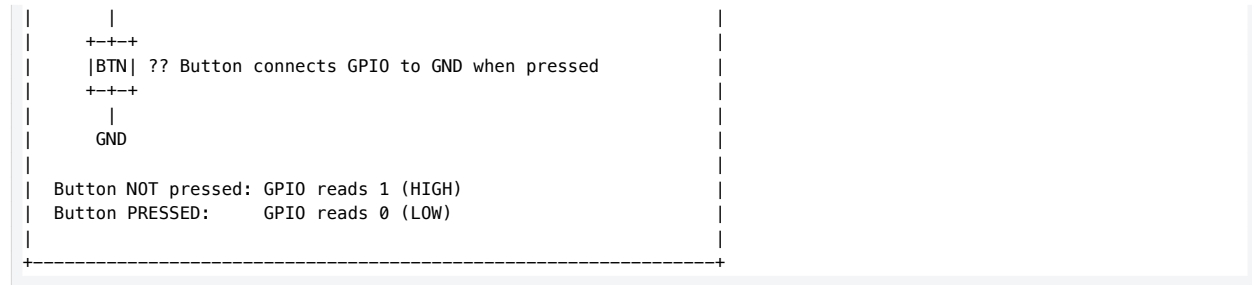
Resistor Type	Default State	When Button Pressed
Pull-Up	HIGH (1)	LOW (0)
Pull-Down	LOW (0)	HIGH (1)

The Pico 2 has **internal** pull resistors that you can enable with software - no external components needed!

```

+-----+
| Pull-Up Resistor (what we're using)               |
|+-----+                                           |
| 3.3V                                              |
| |                                                 |
| | + (internal pull-up resistor)                  |
| |                                                 |
| | +-----? GPIO 15 (reads HIGH normally)        |
|+-----+                                           |
+-----+

```



GPIO Input Functions

Function	Purpose
<code>gpio_init(pin)</code>	Initialize a GPIO pin for use
<code>gpio_set_dir(pin, GPIO_IN)</code>	Set pin as INPUT
<code>gpio_pull_up(pin)</code>	Enable internal pull-up resistor
<code>gpio_pull_down(pin)</code>	Enable internal pull-down resistor
<code>gpio_get(pin)</code>	Read the current state (returns 0 or 1)

The Ternary Operator

The code uses a **ternary operator** to control the LED based on button state:

```
gpio_put(LED_GPIO, pressed ? 0 : 1);
```

This is a compact if-else statement:

- If pressed is **true (1)**: output 0 (LED OFF... wait, that seems backwards!)
- If pressed is **false (0)**: output 1 (LED ON)

Why is it inverted? Because of the pull-up resistor!

- Button **released** -> GPIO reads 1 -> pressed = 1 -> output 0 -> LED OFF
- Button **pressed** -> GPIO reads 0 -> pressed = 0 -> output 1 -> LED ON

A clearer way to write this:

```
gpio_put(LED_GPIO, !gpio_get(BUTTON_GPIO));
```

Part 3: Understanding Compiler Optimizations

Why Does Code Disappear?

When you compile code, the compiler tries to make it faster and smaller. This is called **optimization**. Sometimes the compiler removes code that it thinks has no effect!

Example from our code:

```
while (true) {
    uint8_t regular_fav_num = 42; // Created
    regular_fav_num++;           // Incremented to 43
    // But then it's destroyed and recreated as 42 next loop!
}
```

The compiler sees that incrementing `regular_fav_num` has no lasting effect (because it's recreated as 42 each

loop), so it may **optimize away** the increment operation entirely!

Function Inlining

Sometimes the compiler **inlines** functions, meaning it replaces a function call with the function's code directly.

Original code:

```
gpio_pull_up(BUTTON_GPIO);
```

What the compiler might do:

```
// Instead of calling gpio_pull_up, it calls the underlying function:
gpio_set_pulls(BUTTON_GPIO, true, false);
```

This is why when you look for `gpio_pull_up` in the binary, you might find `gpio_set_pulls` instead!

Part 4: Setting Up Your Environment

Prerequisites

Before we start, make sure you have:

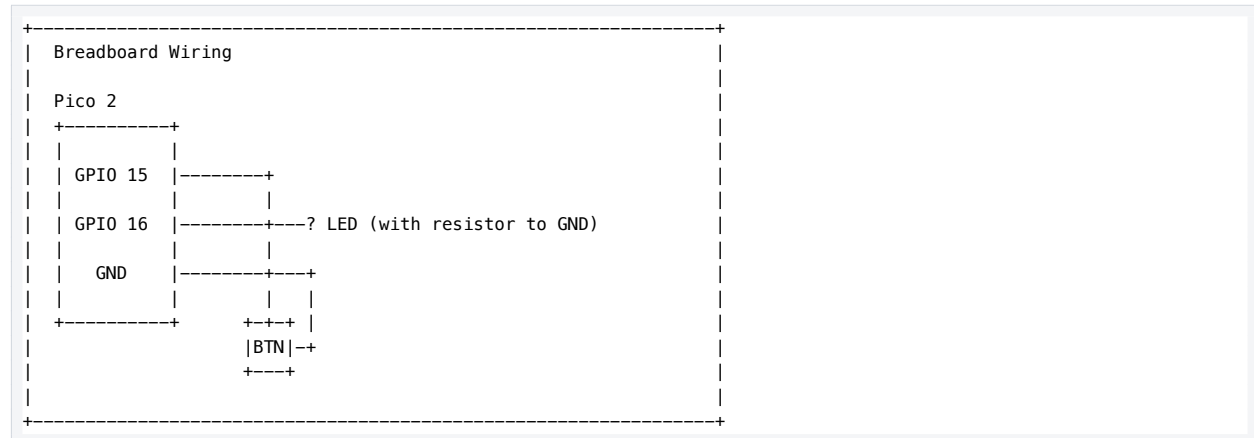
1. A Raspberry Pi Pico 2 board
2. A Raspberry Pi Pico Debug Probe
3. OpenOCD installed and configured
4. GDB (arm-none-eabi-gdb) installed
5. Python installed (for UF2 conversion)
6. A serial monitor (PuTTY, minicom, or screen)
7. A push button connected to GPIO 15
8. An LED connected to GPIO 16 (or use the breadboard LED)
9. A hex editor (HxD, ImHex, or similar)
10. The sample project: `0x0014_static-variables`

Hardware Setup

Connect your button like this:

- One side of button -> GPIO 15
- Other side of button -> GND

The internal pull-up resistor provides the 3.3V connection, so you only need to connect to GND!



Project Structure

```
Embedded-Hacking/
+-- 0x0014_static-variables/
|   +-- build/
|       +-- 0x0014_static-variables.uf2
|       +-- 0x0014_static-variables.elf
|   +-- 0x0014_static-variables.c
+-- uf2conv.py
```

Part 5: Hands-On Tutorial - Static Variables and GPIO Input

Step 1: Review the Source Code

Let's examine the static variables code:

File: 0x0014_static-variables.c

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

int main(void) {
    stdio_init_all();

    const uint BUTTON_GPIO = 15;
    const uint LED_GPIO = 16;
    bool pressed = 0;

    gpio_init(BUTTON_GPIO);
    gpio_set_dir(BUTTON_GPIO, GPIO_IN);
    gpio_pull_up(BUTTON_GPIO);

    gpio_init(LED_GPIO);
    gpio_set_dir(LED_GPIO, GPIO_OUT);

    while (true) {
        uint8_t regular_fav_num = 42;
        static uint8_t static_fav_num = 42;

        printf("regular_fav_num: %d\r\n", regular_fav_num);
        printf("static_fav_num: %d\r\n", static_fav_num);

        regular_fav_num++;
        static_fav_num++;

        pressed = gpio_get(BUTTON_GPIO);
        gpio_put(LED_GPIO, pressed ? 0 : 1);
    }
}
```

What this code does:

- Line 6-8:** Defines constants for button (GPIO 15) and LED (GPIO 16) pins
- Line 10-12:** Sets up GPIO 15 as input with internal pull-up resistor
- Line 14-15:** Sets up GPIO 16 as output for the LED
- Line 18-19:** Creates two variables:
 - regular_fav_num - a normal local variable (recreated each loop)
 - static_fav_num - a static variable (persists across loops)
- Line 21-22:** Prints both values to the serial terminal
- Line 24-25:** Increments both values
- Line 27-28:** Reads button and controls LED accordingly

Step 2: Flash the Binary to Your Pico 2

1. Hold the BOOTSEL button on your Pico 2
2. Plug in the USB cable (while holding BOOTSEL)
3. Release BOOTSEL - a drive called “RPI-RP2” appears
4. Drag and drop 0x0014_static-variables.uf2 onto the drive
5. The Pico will reboot and start running!

Step 3: Open Your Serial Monitor

Open PuTTY, minicom, or screen and connect to your Pico’s serial port.

You should see output like this:

```
...
regular_fav_num: 42
static_fav_num: 42
regular_fav_num: 42
static_fav_num: 43
regular_fav_num: 42
static_fav_num: 44
regular_fav_num: 42
static_fav_num: 45
...
```

Notice the difference:

- `regular_fav_num` stays at 42 every time (it’s recreated each loop)
- `static_fav_num` increases each time (it persists and remembers its value)

Step 4: Test the Button

Now test the button behavior:

- **Button NOT pressed:** LED should be OFF
- **Button PRESSED:** LED should turn ON

That may feel backwards at first glance, but it matches the pull-up + ternary logic in the code. We’ll hack this later to make the behavior more intuitive.

Step 5: Watch for Overflow

Keep the program running and watch `static_fav_num`. After 255, you’ll see:

```
static_fav_num: 254
static_fav_num: 255
static_fav_num: 0      ?? Wrapped around!
static_fav_num: 1
static_fav_num: 2
...
```

This demonstrates unsigned integer overflow!

Part 6: Debugging with GDB (Dynamic Analysis)

? **REVIEW:** This setup is identical to previous weeks. If you need a refresher on OpenOCD and GDB connection, refer back to Week 3 Part 6.

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\0x0014_static-variables.elf
```

Connect to target:

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

Step 6: Examine Main Function

Let's examine the main function at its entry point. First, disassemble from the start:

```
x/38i 0x10000234
```

You should see output like:

```
(gdb) x/38i 0x10000234
0x10000234 <main>:      push    {r4, lr}
0x10000236 <main+2>:      bl      0x10003014 <stdio_init_all>
0x1000023a <main+6>:      movs    r0, #15
0x1000023c <main+8>:      bl      0x10000300 <gpio_init>
0x10000240 <main+12>:     movs    r0, #15
0x10000242 <main+14>:     mov.w   r3, #0
0x10000246 <main+18>:     mcrr    0, 4, r0, r3, cr4
0x1000024a <main+22>:     movs    r2, #0
0x1000024c <main+24>:     movs    r1, #1
0x1000024e <main+26>:     bl      0x100002d8 <gpio_set_pulls>
0x10000252 <main+30>:     movs    r0, #16
0x10000254 <main+32>:     bl      0x10000300 <gpio_init>
0x10000258 <main+36>:     movs    r3, #16
0x1000025a <main+38>:     mov.w   r2, #1
0x1000025e <main+42>:     mcrr    0, 4, r3, r2, cr4
0x10000262 <main+46>:     ldr     r4, [pc, #44] @ (0x10000290 <main+92>)
0x10000264 <main+48>:     movs    r1, #42 @ 0x2a
0x10000266 <main+50>:     ldr     r0, [pc, #44] @ (0x10000294 <main+96>)
0x10000268 <main+52>:     bl      0x100031a4 <__wrap_printf>
0x1000026c <main+56>:     ldrb    r1, [r4, #0]
0x1000026e <main+58>:     ldr     r0, [pc, #40] @ (0x10000298 <main+100>)
0x10000270 <main+60>:     bl      0x100031a4 <__wrap_printf>
0x10000274 <main+64>:     mov.w   r1, #3489660928 @ 0xd0000000
0x10000278 <main+68>:     ldrb    r3, [r4, #0]
0x1000027a <main+70>:     movs    r2, #16
0x1000027c <main+72>:     adds    r3, #1
0x1000027e <main+74>:     strb    r3, [r4, #0]
0x10000280 <main+76>:     ldr     r3, [r1, #4]
0x10000282 <main+78>:     ubfx    r3, r3, #15, #1
0x10000286 <main+82>:     eor.w   r3, r3, #1
0x1000028a <main+86>:     mcrr    0, 4, r2, r3, cr0
0x1000028e <main+90>:     b.n     0x10000264 <main+48>
0x10000290 <main+92>:     lsls    r0, r5, #22
0x10000292 <main+94>:     movs    r0, #0
0x10000294 <main+96>:     adds    r5, #96 @ 0x60
0x10000296 <main+98>:     asrs    r0, r0, #32
0x10000298 <main+100>:    adds    r5, #120 @ 0x78
0x1000029a <main+102>:    asrs    r0, r0, #32
```

Step 7: Set a Breakpoint at Main

```
b *0x10000234
c
```

Step 8: Examine the Static Variable Location

Static variables live at fixed RAM addresses. But how do we find that address? Look at 0x10000262 in the disassembly from Step 6:

```
(gdb) x/16i 0x10000234
=> 0x10000234 <main>:      push    {r4, lr}
0x10000236 <main+2>:      bl      0x10003014 <stdio_init_all>
0x1000023a <main+6>:      movs    r0, #15
0x1000023c <main+8>:      bl      0x10000300 <gpio_init>
0x10000240 <main+12>:     movs    r0, #15
0x10000242 <main+14>:     mov.w   r3, #0
0x10000246 <main+18>:     mcrr    0, 4, r0, r3, cr4
0x1000024a <main+22>:     movs    r2, #0
0x1000024c <main+24>:     movs    r1, #1
0x1000024e <main+26>:     bl      0x100002d8 <gpio_set_pulls>
0x10000252 <main+30>:     movs    r0, #16
0x10000254 <main+32>:     bl      0x10000300 <gpio_init>
0x10000258 <main+36>:     movs    r3, #16
0x1000025a <main+38>:     mov.w   r2, #1
0x1000025e <main+42>:     mcrr    0, 4, r3, r2, cr4
0x10000262 <main+46>:     ldr     r4, [pc, #44] @ (0x10000290 <main+92>)
```

This loads r4 from the **literal pool** at address 0x10000290. The literal pool stores constants that are too large for immediate encoding - in this case, a 32-bit RAM address. Let's examine what's stored there:

```
(gdb) x/1wx 0x10000290
0x10000290 <main+92>: 0x200005a8
```

That's 0x200005a8 - the RAM address of static_fav_num! The compiler placed this address in the literal pool because it can't encode a full 32-bit address in a single Thumb instruction.

Common confusion three things that trip people up here:

1. GDB shows <striped_spin_lock_num> instead of <static_fav_num>. static_fav_num is a function-local static, so it does not appear in GDB's global symbol table. GDB labels the address with the nearest *named* global symbol it can find in this case an SDK-internal spin-lock variable that happens to be nearby. The address 0x200005a8 is still correct; the label is just GDB's best guess at a name.

```
(gdb) x/x 0x200005a8
0x200005a8 <striped_spin_lock_num>: 0x0000122a
```

2. x/x 0x200005a8 shows 0x0000122a, not 0x0000002a (42). x/x reads a **4-byte word** (32 bits). static_fav_num is a uint8_t only **1 byte**. The low byte of 0x0000122a is 0x2a = 42. That is the variable. The upper bytes belong to whatever happens to be in adjacent RAM. To read exactly one byte and see 42, use:

```
(gdb) x/1ub 0x200005a8
0x200005a8 <striped_spin_lock_num>: 42
```

3. x/x *0x200005a8 shows junk at address 0x122a. The * is a pointer dereference. GDB first reads the 4-byte word at 0x200005a8 (= 0x0000122a), then treats that *value* as a new address and reads from 0x0000122a. You are asking GDB to follow a pointer that was never meant to be a pointer the result is meaningless. Drop the *: use x/1ub 0x200005a8 to read the variable's value directly.

Tip: Why did the disassembly at 0x10000290 show lsls r0, r5, #22 instead? Because x/i (disassemble) interprets raw data as instructions. The bytes A8 05 00 20 at that address are the little-endian encoding of 0x200005A8, but GDB's disassembler doesn't know it's data - it tries to decode it as

a Thumb instruction. Using x/wx (examine as word) shows the actual value.

Step 9: Step Through the Loop

The loop body starts at 0x10000264. Set a breakpoint there and continue:

```
(gdb) b *0x10000264
(gdb) c
(gdb) x/i $pc
=> 0x10000264 <main+48>:      movs    r1, #42 @ 0x2a
```

GDB will stop at `movs r1, #42` that is the compiler's constant for `regular_fav_num`. Let's set a breakpoint and continue at 0x10000278.

```
(gdb) b *0x10000278
(gdb) c
(gdb) x/i $pc
=> 0x10000278 <main+68>:      ldrb    r3, [r4, #0]
```

This is where the static variable is actually manipulated:

```
(gdb) x/4i $pc
=> 0x10000278 <main+68>:      ldrb    r3, [r4, #0] <- load static_fav_num from RAM into r3
0x1000027a <main+70>:      movs    r2, #16    <- load LED pin number (16) into r2 for later
0x1000027c <main+72>:      adds    r3, #1      <- increment r3 by 1
0x1000027e <main+74>:      strb    r3, [r4, #0] <- store incremented value back to RAM
```

Confirm what r4 actually contains right now:

```
(gdb) x/x $r4
0x200005a8 <striped_spin_lock_num>:      0x2a
```

r4 is 0x200005a8, a RAM address. The value at that address is 0x2a (42 in decimal), which is the current value of `static_fav_num`. After `adds r3, #1` and `strb r3, [r4, #0]` execute, examining r4 again will show 0x0000002b (43).

What is the literal pool?

A Thumb `ldr` instruction is only 16 or 32 bits wide. There is no room inside those bits to encode a full 32-bit constant like 0x200005a8. To solve this, the compiler collects all such large constants and writes them into a small block of **raw data** embedded directly in the flash image this block is called a **literal pool** (also called a constant pool).

There is not one literal pool per flash, there is (at least) one per function. Every function that references large constants gets its own pool appended to its code. A large function can have several pools scattered through it wherever the compiler decides to emit them. Each pool is purely data; the CPU must never try to execute it.

Where exactly is a literal pool placed? Always **after an unconditional branch**, never before or in the middle of a straight-line code path. This is by design: the CPU falls through code sequentially, so the pool must be unreachable by fall-through. In our case the pool sits at 0x10000290, directly after the `b.n 0x10000264` branch at 0x1000028e that closes the `while (true)` loop:

```
0x1000028e: b.n 0x10000264 <- unconditional branch back to loop top
0x10000290: [literal pool] <- CPU never reaches here by fall-through
```

Why does the pool have to be nearby? The `ldr rN, [pc, #offset]` instruction encodes the offset in a limited number of bits. In standard Thumb the maximum reach is **1020 bytes** from the instruction. If the function is long enough that the end is out of range, the assembler emits an intermediate pool mid-function after the next unconditional branch it can find. This is in the ARM v8-M Architecture Reference Manual p. 672.

How the load works in our function: At 0x10000262 the compiler emits:

```
ldr r4, [pc, #44] @ (0x10000290)
```

On ARM, PC reads as the address of the instruction **plus 4** during execution. So the effective address is $0x10000262 + 4 + 44 = 0x10000290$. That word in the pool contains $0x200005a8$, the RAM address of `static_fav_num`. The pool is also where the `printf` format-string pointers live (the other `ldr` instructions you saw in the disassembly).

Flash (read-only)		RAM
0x10000262: ldr r4, [pc, #44]		0x200005a8: 42 <- static_fav_num
...loop body...		
0x1000028e: b.n 0x10000264 (end of loop)		
+----- literal pool (data, not code) -----+		
0x10000290: 0x200005a8	+-----+ ^	
0x10000294: ptr to "regular_fav_num: %d"		r4 holds this address
0x10000298: ptr to "static_fav_num: %d"		
+----- next function -----+		
0x1000029c: push {r4}		

You can see this directly. Run `x/10i 0x1000028e` in GDB and you will get exactly this:

```
(gdb) x/10i 0x1000028e
(gdb) x/10i 0x1000028e
0x1000028e <main+90>:      b.n      0x10000264 <main+48> <- unconditional branch, loop top
0x10000290 <main+92>:      lsls     r0, r5, #22      <-+
0x10000292 <main+94>:      movs     r0, #0           <-+--- word 1: 0x200005a8 (fixed RAM addr
-> static_fav_num)
0x10000294 <main+96>:      adds     r5, #96 @ 0x60     <-+
0x10000296 <main+98>:      asrs     r0, r0, #32       <-+--- word 2: 0x10004b60 (flash addr
-> regular_fav_num text)
0x10000298 <main+100>:     adds     r5, #120          <-+
0x1000029a <main+102>:     asrs     r0, r0, #32       <-+--- word 3: 0x10004b78 (flash addr
-> static_fav_num text)
```

`0x10000290 - 0x1000029b` is the literal pool 12 bytes, 3 32-bit words. GDB's `x/i` (disassemble) has no idea those bytes are data, so it blindly decodes them as Thumb instructions and produces nonsense (`lsls r0, r5, #22`, etc.). The giveaway that you have entered pool territory is always the same: **garbled, implausible instructions immediately after an unconditional branch**. Use `x/wx` to read those addresses as data and you get the real values:

```
(gdb) x/3wx 0x10000290
0x10000290 <main+92>:  0x200005a8      0x10003560      0x10003578
```

```
(gdb) x/1ub 0x200005a8
0x200005a8 <striped_spin_lock_num>:  42
```

- `0x200005a8` RAM address of `static_fav_num`, lives in the **.data section** (initialized static RAM, `0x20000000`0x200xxxxx`). It is **.data** and not **.bss** because the variable has an explicit initializer (= 42). **.bss** is for variables that are zero-initialized or have no initializer at all.

```
(gdb) x/s 0x10003560
0x10003560:  "regular_fav_num: %d\r\n"
```

- `0x10003560` flash address of the `"regular_fav_num: %d\r\n"` string literal, lives in the **.rodata section** (read-only data) inside flash (`0x10000000+`). String literals are constants baked into the binary at compile time they never change and never move, so they stay in flash.

```
(gdb) x/s 0x10003578
0x10003578:    "static_fav_num: %d\r\n"
```

- 0x10003578 flash address of the "static_fav_num: %d\r\n" string literal, also in **.rodata** in flash for the same reason.

So the pool contains addresses into three different regions: RAM **.data** (the static variable), and flash **.rodata** (both format strings). Only the RAM address needed to be in the pool, the flash addresses could in principle be reached other ways, but they are also too large to encode as 16-bit immediates, so they go in the pool too.

Why .data and not .bss? The distinction matters:

Section	What goes here	Example declarations	Initial value stored in flash?
.data	Any static-duration variable with a non-zero initializer	<code>static uint8_t x = 42;</code> (inside a function)	Yes the initial value lives in flash; startup copies it to RAM
.data	Same rule applies to global variables with non-zero initializers	<code>uint8_t g = 10;</code> (outside all functions)	Yes same startup copy
.bss	Any static-duration variable with no initializer or zero initializer	<code>static uint8_t x;</code> or <code>static uint8_t x = 0;</code>	No startup just zeroes the RAM range; no flash copy needed
.bss	Same rule for global variables that are zero/uninitialized	<code>uint8_t g;</code> or <code>uint8_t g = 0;</code> (outside functions)	No same zero-fill at startup

The rule is not about `static` specifically it is about **lifetime and initial value**. Any variable whose lifetime spans the whole program (static locals, globals) goes into **.data** or **.bss**. The `static` keyword inside a function just forces that lifetime. A global variable without `static` has the same lifetime and follows the same rule.

`static_fav_num = 42` has an explicit non-zero initializer, so the linker puts it in **.data**. The startup code copies the initial value 42 from flash into RAM address 0x200005a8 before `main()` runs. If it had been `static uint8_t static_fav_num;` (no initializer), the linker would put it in **.bss** instead and the startup code would zero that RAM region no flash copy needed.

Why is there no pool entry for the address of `regular_fav_num`?

Because `regular_fav_num` is an **automatic (stack) variable** it has no fixed address. Automatic variables are allocated on the call stack at runtime, relative to the current stack pointer (`sp`). Their address changes every time the function runs and is different on every call. There is no constant 32-bit address to put in a pool.

In this specific case the compiler went even further: it **never gave `regular_fav_num` a stack slot at all**. The compiler saw that the value is always 42 when printed, so it baked the constant 42 directly into the `movs r1, #42` instruction. The variable lives only in a register at the moment it is needed — no stack space reserved, no pool entry, no load-store cycle.

You can prove this with three GDB checks:

Proof 1 info locals shows the value came from a register, not a stack slot:

```
(gdb) b *0x10000264
(gdb) c
(gdb) info locals
regular_fav_num = 42 '*'
static_fav_num = 42 '*'
BUTTON_GPIO = 15
LED_GPIO = 16
pressed = <optimized out>
```

The '*' suffix after 42 means GDB retrieved the value from a **register** (specifically r1, which holds 42 at this point), not from a stack slot. The variable has no fixed memory address — GDB can report its current value only because the right register happens to hold it right now. Notice that pressed is the one showing <optimized out> — the compiler removed it from tracking entirely because its value is never needed after the ternary expression.

Proof 2 The function prologue allocates no stack space for locals:

```
(gdb) x/2i 0x10000234
0x10000234 <main>:  push    {r4, lr}
0x10000236 <main+2>:  bl      0x10003014 <stdio_init_all>
```

The very first instruction is push {r4, lr}, that saves r4 (a callee-saved register) and lr (return address). There is **no** sub sp, #N instruction after it. On ARM, allocating stack space for local variables requires subtracting from sp to reserve room. If regular_fav_num lived on the stack, there would be a sub sp, #4 (or similar) here. There isn't, so no stack slot was ever created.

Proof 3 The loop body contains no store to the stack for regular_fav_num:

```
(gdb) x/6i 0x10000264
=> 0x10000264 <main+48>:      movs    r1, #42 @ 0x2a
                                // constant 42 baked directly into instruction
0x10000266 <main+50>:      ldr r0, [pc, #44] @ (0x10000294 <main+96>)
                                // load format string address from pool
0x10000268 <main+52>:      bl      0x100031a4 <__wrap_printf>
                                // call printf
0x1000026c <main+56>:      ldrb    r1, [r4, #0]
                                // load static_fav_num from RAM
0x1000026e <main+58>:      ldr r0, [pc, #40] @ (0x10000298 <main+100>)
                                // load format string address from pool
0x10000270 <main+60>:      bl      0x100031a4 <__wrap_printf>
                                call printf
```

For regular_fav_num, the value 42 goes straight into r1 via movs, it is never written to memory and never read back from memory. There is no strb r1, [sp, #N] storing it to the stack, and no ldrb r1, [sp, #N] loading it back. The variable exists only in the source code. In the binary it is just a number in an instruction encoding.

static_fav_num is the exact opposite. It lives in the .data section of RAM, a fixed, known address (0x200005a8) that is set at link time and never changes for the life of the program. That fixed address is exactly the kind of value that cannot fit in a 16-bit Thumb instruction, so the linker writes it into the literal pool and the CPU fetches it with ldr r4, [pc, #44].

IMPORTANT: Static variables are **not** on the heap. The heap is for dynamic memory (malloc/free), memory whose lifetime is controlled explicitly at runtime. Static and global variables live in the .data section (if initialized) or .bss section (if zero/uninitialized), a region of RAM with a fixed layout determined at link time, not at runtime.

Memory regions for our two variables:

```
+-----+
| FLASH (read-only, 0x10000000+) |
| - Code (instructions)         |
```

```

| - Literal pool (our 3 words) |
| - String literals ("regular_fav_num...") |
+-----+
| RAM .data section (0x20000000+) |
| - static_fav_num @ 0x200005a8 <- pool |
+-----+
| STACK (grows down from 0x20082000) |
| - regular_fav_num (if not optimized) |
|   frame-relative, no fixed address |
+-----+
| HEAP (grows up, between stack and .bss) |
| - malloc/free only, nothing here today |
+-----+

```

The pool ends at 0x1000029b. At 0x1000029d a new function begins, specifically `gpio_set_function`. The first two instructions, `push {r4}` (save caller's `r4` to the stack) and `mov.w r4, #256` (load a working constant), are a standard SDK function prologue, nothing to do with `main`. The `ldr r3, [pc, #48]` at 0x100002a3 (`gpio_set_function+6`) is that next function loading from **its own** literal pool further ahead in flash — proof that every function manages its own pool.

Step 10: Examine Register Values

After breaking at 0x10000264, check the registers:

```
(gdb) i r
```

What you will actually see:

- `r4 = 0x200005a8` — the static variable's fixed RAM address. This was loaded from the literal pool in the function prologue and never changes across loop iterations.
- `r1 = leftover value from the previous printf call (e.g. 0x40039044)`. The breakpoint is stopped **at** `movs r1, #42` — that instruction has not executed yet, so `r1` does not yet hold 42.
- `r3 = leftover from prior operations (e.g. 0x10)`. It will be used later in the loop body to load, increment, and store `static_fav_num`, but not yet.
- `pc = 0x10000264` — the loop top, pointing at the `movs r1, #42` instruction.

You can confirm `r4` holds the right address and value:

```
(gdb) x/1db $r4
0x200005a8 <striped_spin_lock_num>: 42
```

The `<striped_spin_lock_num>` label is just the nearest named global symbol in the SDK that GDB finds at that address — the actual variable stored there is our `static_fav_num`.

Step 11: Watch the Static Variable Change

Now that we know the static variable lives at 0x200005a8, examine it directly:

```
(gdb) x/1db 0x200005a8
0x200005a8 <striped_spin_lock_num>: 42
```

Step through a full loop iteration (back to 0x10000264) and re-examine:

```
(gdb) c
(gdb) x/1db 0x200005a8
0x200005a8 <striped_spin_lock_num>: 43
```

The value incremented from 42 to 43! Each loop iteration, the `adds r3, #1` at 0x1000027c bumps it by 1, and `strb r3, [r4, #0]` at 0x1000027e writes it back to RAM.

Step 12: Examine GPIO State

Read the GPIO input register to see the button state:

```
(gdb) x/1wx 0xd0000004
0xd0000004:    0x00008003
```

The SIO GPIO input register at `0xd0000004` shows the current state of all GPIO pins. Bit 15 corresponds to our button on GPIO 15. To extract just bit 15:

```
(gdb) p/x (*(unsigned int *)0xd0000004 >> 15) & 1
$1 = 0x1
```

- Returns 1 when button is **not pressed** (pull-up holds it HIGH)
- Returns 0 when button is **pressed** (connected to GND)

TRY IT!

Part 7: Understanding the Assembly

Now that we've explored the binary in GDB, let's make sense of the key patterns.

Step 13: Analyze the Regular Variable

In GDB, examine the code at the start of the loop:

```
(gdb) x/5i 0x10000262
0x10000262 <main+46>:    ldr r4, [pc, #44]    @ (0x10000290 <main+92>)
=> 0x10000264 <main+48>:    movs    r1, #42 @ 0x2a
0x10000266 <main+50>:    ldr r0, [pc, #44]    @ (0x10000294 <main+96>)
0x10000268 <main+52>:    bl      0x100031a4 <__wrap_printf>
0x1000026c <main+56>:    ldrb    r1, [r4, #0]
```

Look for this instruction:

```
0x10000264 <main+48>:    movs    r1, #42 @ 0x2a
```

This loads the value `0x2a` (42 in decimal) directly into register `r1` for the first `printf` call.

Key insight: The compiler **never allocated stack space** for `regular_fav_num`. Since it is always 42 when printed, the compiler bakes the constant directly into the `movs r1, #42` instruction. The `regular_fav_num++` after the print is also removed because it has no observable effect — the variable is recreated as 42 on the next loop iteration anyway.

Step 14: Analyze the Static Variable

Examine the static variable operations in the second half of the loop body:

```
(gdb) x/10i 0x10000274
0x10000274 <main+64>:    mov.w    r1, #3489660928 @ 0xd0000000
0x10000278 <main+68>:    ldrb    r3, [r4, #0]
0x1000027a <main+70>:    movs    r2, #16
0x1000027c <main+72>:    adds    r3, #1
0x1000027e <main+74>:    strb    r3, [r4, #0]
0x10000280 <main+76>:    ldr     r3, [r1, #4]
0x10000282 <main+78>:    ubfx    r3, r3, #15, #1
0x10000286 <main+82>:    eor.w    r3, r3, #1
0x1000028a <main+86>:    mcrr    0, 4, r2, r3, cr0
0x1000028e <main+90>:    b.n     0x10000264 <main+48>
```

Look for the load-increment-store pattern using `r4` (which holds the static variable's RAM address):

...

```

0x10000278 <main+68>:    ldrb    r3, [r4, #0]
0x1000027a <main+70>:    movs    r2, #16
0x1000027c <main+72>:    adds    r3, #1
0x1000027e <main+74>:    strb    r3, [r4, #0]
...

```

Note that r4 was loaded earlier at 0x10000262 via `ldr r4, [pc, #44]` - this pulled the static variable's RAM address (0x200005a8) from the literal pool at 0x10000290.

Key insight: The static variable lives at a **fixed RAM address** (0x200005a8). It's loaded, incremented, and stored back. The regular variable, by contrast, never gets a stack slot — the compiler holds it in a register and bakes the constant 42 directly into the instruction, so there is nothing to load or store.

Verify the static variable value which should be 43:

```

(gdb) x/ldb 0x200005a8
0x200005a8 <striped_spin_lock_num>:    43

```

Step 15: Analyze the GPIO Logic

Examine the GPIO input/output code:

```

(gdb) x/10i 0x10000274
0x10000274 <main+64>:    mov.w    r1, #3489660928 @ 0xd0000000
0x10000278 <main+68>:    ldrb    r3, [r4, #0]
0x1000027a <main+70>:    movs    r2, #16
0x1000027c <main+72>:    adds    r3, #1
0x1000027e <main+74>:    strb    r3, [r4, #0]
0x10000280 <main+76>:    ldr     r3, [r1, #4]
0x10000282 <main+78>:    ubfx    r3, r3, #15, #1
0x10000286 <main+82>:    eor.w    r3, r3, #1
0x1000028a <main+86>:    mcrr    0, 4, r2, r3, cr0
0x1000028e <main+90>:    b.n     0x10000264 <main+48>

```

Breaking this down:

Address	Instruction	Purpose
0x10000274	<code>mov.w r1, #0xd0000000</code>	Load SIO (Single-cycle I/O) base address into r1
0x10000278	<code>ldrb r3, [r4, #0]</code>	Load <code>static_fav_num</code> from RAM into r3
0x1000027a	<code>movs r2, #16</code>	Load LED pin number (16) into r2 for later
0x1000027c	<code>adds r3, #1</code>	Increment <code>static_fav_num</code> by 1
0x1000027e	<code>strb r3, [r4, #0]</code>	Store incremented value back to RAM
0x10000280	<code>ldr r3, [r1, #4]</code>	Read GPIO input state (SIO_GPIO_IN at offset 0x04)
0x10000282	<code>ubfx r3, r3, #15, #1</code>	Extract bit 15 (GPIO 15 = button)
0x10000286	<code>eor.w r3, r3, #1</code>	XOR with 1 to invert (implements ? 0 : 1)
0x1000028a	<code>mcrr 0, 4, r2, r3, cr0</code>	Write r3 (button) and r2 (pin 16) to GPIO output
0x1000028e	<code>b.n 0x10000264</code>	Loop back to start (while (true))

Tip: Notice how the compiler interleaves the static variable increment with the GPIO logic. It loads the SIO base address (r1) *before* doing the increment, and sets up r2 = 16 (LED pin) in between. This is called **instruction scheduling** - the compiler reorders instructions to avoid pipeline stalls while waiting for memory reads.

Step 16: Find the Infinite Loop

The last instruction at 0x1000028e is already covered in the table above:

```
0x1000028e <main+90>:      b.n      0x10000264 <main+48>
```

This is an **unconditional branch** back to 0x10000264 (the `movs r1, #42` at the top of the loop) - this is the `while (true)` in our code! There is no `pop` or `bx lr` to return from `main` because the loop never exits.

Part 8: Hacking the Binary with a Hex Editor

Now for the fun part - we'll patch the `.bin` file directly using a hex editor!

Tip: Why a hex editor? GDB cannot write to flash memory - the 0x10000000+ address range where program instructions live. Trying `set *(char *)0x10000264 = 0x2b` in GDB gives `Writing to flash memory forbidden in this context`. To make **permanent** patches that survive a power cycle, we edit the `.bin` file directly with a hex editor and re-flash it.

Step 17: Open the Binary in a Hex Editor

1. Open **HxD** (or your preferred hex editor: ImHex, 010 Editor, etc.)
2. Click **File** -> **Open**
3. Navigate to `C:\Users\flare-vm\Desktop\Embedded-Hacking-main\0x0014_static-variables\build\`
4. Open `0x0014_static-variables.bin`

Step 18: Calculate the File Offset

The binary is loaded at base address 0x10000000. To find the file offset of any address:

```
file_offset = address - 0x10000000
```

For example:

- Address 0x10000264 -> file offset 0x264 (612 in decimal)
- Address 0x10000286 -> file offset 0x286 (646 in decimal)

Step 19: Hack #1 - Change regular_fav_num from 42 to 43

From our GDB analysis, we know the instruction at 0x10000264 is:

```
movs r1, #0x2a    ->    bytes: 2a 21
```

To change the value from 42 (0x2a) to 43 (0x2b):

1. In HxD, open `C:\Users\flare-vm\Desktop\Embedded-Hacking-main\0x0014_static-variables\build\0x0014_static-variables.bin`
2. Press **Ctrl+G** (Go to offset)
3. Enter offset: 264
4. You should see the byte 2A at this position
5. Change 2A to 2B
6. The instruction is now `movs r1, #0x2b` (43 in decimal)

?? How Thumb encoding works: In `movs r1, #imm8`, the immediate value is the first byte, and the opcode 21 is the second byte. So the bytes 2a 21 encode `movs r1, #0x2a`.

Step 20: Hack #2 - Invert the Button Logic

Understand the Encoding

From GDB, we found the `eor.w r3, r3, #1` instruction at 0x10000286 that inverts the button value. Examine the exact bytes:

```
(gdb) x/4bx 0x10000286
0x10000286 <main+82>: 0x83 0xf0 0x01 0x03
```

This is the 32-bit Thumb-2 encoding of `eor.w r3, r3, #1`. The bytes break down as:

```
+-----+
| eor.w r3, r3, #1 -> bytes: 83 F0 01 03 |
|                                         |
| Byte 0: 0x83  -??                      |
| Byte 1: 0xF0  -+ First halfword (opcode + source register) |
| Byte 2: 0x01  ---- Immediate value (#1) ?? CHANGE THIS    |
| Byte 3: 0x03  ---- Destination register (r3)                |
|                                         |
+-----+
```

To change `eor.w r3, r3, #1` to `eor.w r3, r3, #0` (making XOR do nothing):

The file offset is $0x10000286 - 0x10000000 = 0x286$. The immediate byte is the 3rd byte of the instruction, so: $0x286 + 2 = 0x288$.

To change `eor.w r3, r3, #1` to `eor.w r3, r3, #0`:

1. In HxD, press **Ctrl+G** (Go to offset)
2. Enter offset: 288 (the third byte of the 4-byte instruction)
3. You should see the byte 01 at this position
4. Change 01 to 00

?? **Why offset 0x288 and not 0x286?** The immediate value #1 is in the **third byte** of the 4-byte instruction. The instruction starts at file offset 0x286, so the immediate byte is at $0x286 + 2 = 0x288$.

Now the logic is permanently changed:

- Button released (input = 1): $1 \text{ XOR } 0 = 1 \rightarrow$ **LED ON**
- Button pressed (input = 0): $0 \text{ XOR } 0 = 0 \rightarrow$ **LED OFF**

This is the **opposite** of the original behavior!

Step 21: Save the Patched Binary

1. Click **File** -> **Save As**
2. Save as `0x0014_static-variables-h.bin` in the build directory
3. Close the hex editor

Part 9: Converting and Flashing the Hacked Binary

Step 22: Convert to UF2 Format

Open a terminal and navigate to your project directory:

```
cd C:\Users\flare-vm\Desktop\Embedded-Hacking-main\0x0014_static-variables
```

Run the conversion command:

```
python ..\uf2conv.py build\0x0014_static-variables-h.bin --base 0x10000000 --family 0xe48bff59 --output
↳ build\hacked.uf2
```

What this command means:

- `uf2conv.py` = the conversion script (in the parent Embedded-Hacking directory)
- `--base 0x10000000` = the XIP base address where code runs from
- `--family 0xe48bff59` = the RP2350 family ID

- `--output build\hacked.uf2` = the output filename

Step 23: Flash the Hacked Binary

1. Hold BOOTSEL and plug in your Pico 2
2. Drag and drop `hacked.uf2` onto the RPI-RP2 drive
3. Open your serial monitor

Step 24: Verify the Hacks

Check the serial output:

```
regular_fav_num: 43    ?? Changed from 42!
static_fav_num: 42
regular_fav_num: 43
static_fav_num: 43
...
```

Check the LED behavior:

- LED should now be **ON by default** (when button is NOT pressed)
- LED should turn **OFF** when you press the button

BOOM! We successfully:

1. Changed the printed value from 42 to 43
2. Inverted the LED/button logic

Part 10: Summary and Review

What We Accomplished

1. **Learned about static variables** - How they persist across function calls and loop iterations
2. **Understood memory layout** - Stack vs static storage vs heap
3. **Configured GPIO inputs** - Using pull-up resistors and reading button states
4. **Analyzed compiled code in GDB** - Saw how the compiler optimizes code
5. **Discovered function inlining** - `gpio_pull_up` became `gpio_set_pulls`
6. **Hacked variable values** - Changed 42 to 43 using a hex editor
7. **Inverted GPIO logic** - Made LED behavior opposite

Static vs Automatic Variables

Aspect	Automatic (Regular)	Static
Storage	Stack	Static storage (.data/.bss)
Lifetime	Block/function scope	Entire program
Initialization	Every time block entered	Once at program start
Persistence	Lost when scope exits	Retained between calls
Compiler view	May be optimized away	Always has memory location

GPIO Input Configuration

```
+-----+
| GPIO Input Setup Steps |
| 1. gpio_init(pin)      | - Initialize the pin |
| 2. gpio_set_dir(pin, GPIO_IN) | - Set as input |
| 3. gpio_pull_up(pin)   | - Enable pull-up |
+-----+
```

OR gpio_pull_down(pin)	- OR enable pull-down
4. gpio_get(pin)	- Read the state

The Binary Hacking Workflow

1. Analyze the binary with GDB	
- Disassemble functions with x/Ni	
- Identify key instructions and addresses	
2. Understand compiler optimizations	
- Some functions get inlined (gpio_pull_up -> gpio_set_pulls)	
- Some variables are optimized away	
3. Calculate file offsets	
- file_offset = address - 0x10000000	
4. Patch the .bin file with a hex editor	
- Open the .bin file in HxD / ImHex	
- Go to the calculated offset	
- Change the target byte(s)	
5. Convert to UF2	
python uf2conv.py file.bin --base 0x10000000	
--family 0xe48bff59 --output hacked.uf2	
6. Flash and verify	
- Hold BOOTSEL, plug in, drag UF2	
- Check serial output and button/LED behavior	

Key Memory Addresses

Address	Description
0x10000234	Typical main() entry point
0x10003014	stdio_init_all() function
0x200005a8	Static variable storage (example)
0xd0000000	SIO (Single-cycle I/O) base address

Key Takeaways

1. **Static variables persist** - They keep their value between function calls and loop iterations.
2. **Static storage ? heap** - Static variables are in a fixed location, not dynamically allocated.
3. **Compilers optimize aggressively** - Regular variables may be optimized away if the compiler sees no effect.
4. **Function inlining is common** - gpio_pull_up becomes gpio_set_pulls in the binary.
5. **Pull-up resistors invert logic** - Button pressed = LOW, button released = HIGH.
6. **XOR is useful for inverting** - eor r3, r3, #0x1 flips a bit between 0 and 1.
7. **Static variables have fixed addresses** - You can find them in the .data section at known RAM addresses.

8. **Overflow wraps around** - A `uint8_t` at 255 becomes 0 when incremented.
 9. **UBFX extracts bits** - Used to read a single GPIO pin from a register.
 10. **Binary patching is powerful** - Change values and logic without source code!
-

Glossary

Term	Definition
Automatic	Variable that's created and destroyed automatically (local vars)
eor/XOR	Exclusive OR - flips bits where operands differ
Floating Input	GPIO input with undefined voltage (reads random values)
Function Inlining	Compiler replaces function call with the function's code
gpio_get	Function to read the current state of a GPIO pin
Heap	Memory area for dynamic allocation (malloc/free)
Overflow	When a value exceeds its type's maximum and wraps around
Pull-Down	Resistor that holds a pin LOW when nothing drives it
Pull-Up	Resistor that holds a pin HIGH when nothing drives it
SIO	Single-cycle I/O - fast GPIO access on RP2350
Stack	Memory area for local variables and function call frames
Static Storage	Fixed memory area for static and global variables
Static Variable	Variable declared with <code>static</code> that persists across calls
Ternary Operator	<code>condition ? value_if_true : value_if_false</code>
UBFX	Unsigned Bit Field Extract - extracts bits from a register
Varargs	Variable arguments - functions that take unlimited parameters

Additional Resources

GPIO Input Reference

Function	Purpose
<code>gpio_init(pin)</code>	Initialize GPIO pin
<code>gpio_set_dir(pin, GPIO_IN)</code>	Set pin as input
<code>gpio_set_dir(pin, GPIO_OUT)</code>	Set pin as output
<code>gpio_pull_up(pin)</code>	Enable internal pull-up
<code>gpio_pull_down(pin)</code>	Enable internal pull-down
<code>gpio_disable_pulls(pin)</code>	Disable all pull resistors
<code>gpio_get(pin)</code>	Read pin state (0 or 1)
<code>gpio_put(pin, value)</code>	Set pin output (0 or 1)

Key Assembly Instructions

Instruction	Description
<code>movs rN, #imm</code>	Move immediate value to register
<code>ldrb rN, [rM, #off]</code>	Load byte from memory
<code>strb rN, [rM, #off]</code>	Store byte to memory
<code>adds rN, #imm</code>	Add immediate value to register
<code>eor rN, rM, #imm</code>	Exclusive OR (XOR) with immediate
<code>ubfx rN, rM, #lsb, #w</code>	Extract unsigned bit field
<code>mcrr p0, ...</code>	Move to coprocessor (GPIO control on RP2350)
<code>b LABEL</code>	Unconditional branch (jump)

Memory Map Quick Reference

Address Range	Description
<code>0x10000000</code>	XIP Flash (code execution)
<code>0x20000000-200005xx</code>	SRAM (.data section)
<code>0x20082000</code>	Stack top (initial SP)
<code>0x40038000</code>	PADS_BANK0 (pad configuration)
<code>0xd0000000</code>	SIO (single-cycle I/O)

Remember: Static variables are your friends when you need to remember values across function calls. But they also make your program's behavior more complex to analyze - which is exactly why we practice reverse engineering!

Happy hacking! ?