
EMBEDDED HACKING

Reverse Engineering the Raspberry Pi Pico 2

Week 7: Constants in Embedded Systems: Debugging and Hacking Constants w/ 1602 LCD I2C 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 `#define` macros and `const` variables
 - Know how constants are stored differently in memory (compile-time vs runtime)
 - Understand the I2C (Inter-Integrated Circuit) communication protocol
 - Configure I2C peripherals and communicate with LCD displays
 - Understand C structs and how the Pico SDK uses them for hardware abstraction
 - Use GDB to examine constants, structs, and string literals in memory
 - Hack constant values and string literals using a hex editor
 - Patch LCD display text without access to source code
-

Part 1: Understanding Constants in C

Two Types of Constants

In C, there are two ways to create values that shouldn't change:

Type	Syntax	Where It Lives	When Resolved
#define	<code>#define FAV_NUM 42</code>	Nowhere (replaced)	Compile time
const	<code>const int NUM = 1337;</code>	Flash (.rodata)	Runtime

Preprocessor Macros (`#define`)

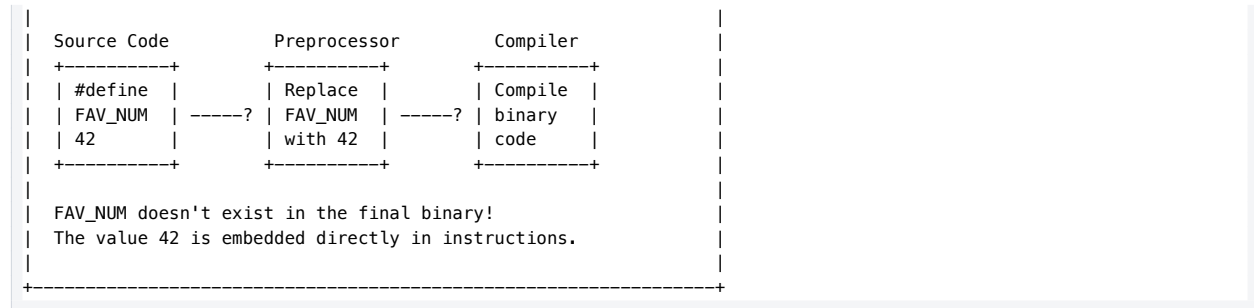
A **preprocessor macro** is a text replacement that happens BEFORE your code is compiled:

```
#define FAV_NUM 42

printf("Value: %d", FAV_NUM);
// Becomes: printf("Value: %d", 42);
```

Think of it like a “find and replace” in a text editor. The compiler never sees `FAV_NUM` - it only sees 42!

```
+-----+
| Preprocessor Macro Flow |
```

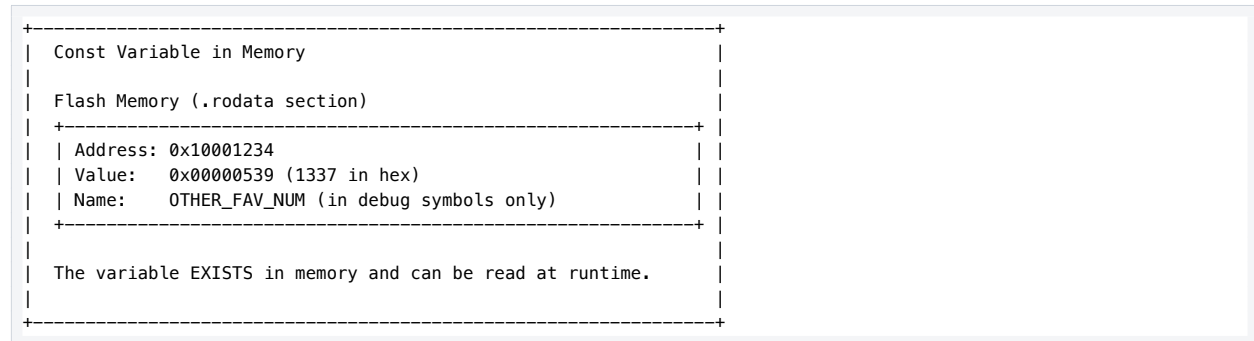


Const Variables

A **const variable** is an actual variable stored in memory, but marked as read-only:

```
const int OTHER_FAV_NUM = 1337;
```

Unlike `#define`, this creates a real memory location in the `.rodata` (read-only data) section of flash:



Comparison: #define vs const

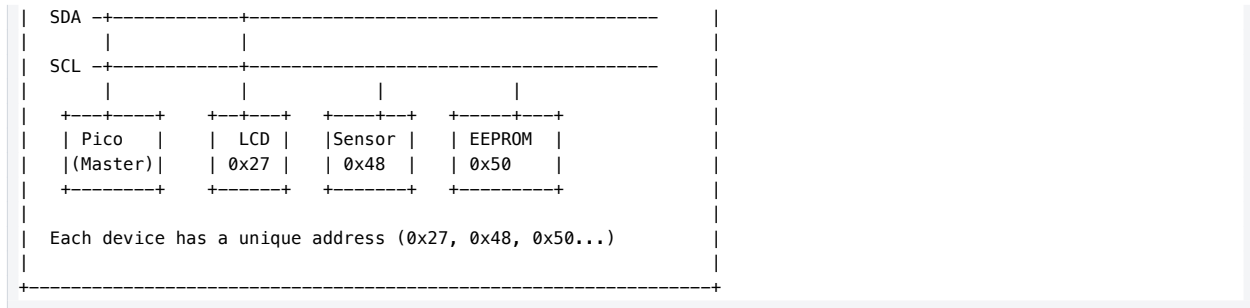
Feature	#define	const
Type checking	None (just text)	Yes (compiler enforced)
Memory usage	None (inlined)	Uses flash space
Debugger visible	No	Yes (with symbols)
Can take address	No (&FAV_NUM fails)	Yes (&OTHER_FAV_NUM works)
Scope	Global (from definition)	Normal C scoping rules

Part 2: Understanding I2C Communication

What is I2C?

I2C (pronounced “I-squared-C” or “I-two-C”) stands for **Inter-Integrated Circuit**. It’s a way for chips to talk to each other using just TWO wires!





The Two I2C Wires

Wire	Name	Purpose
SDA	Serial Data	Carries the actual data bits
SCL	Serial Clock	Timing signal that synchronizes data

Why Pull-Up Resistors?

I2C uses **open-drain** signals, meaning devices can only pull the line LOW. They can't drive it HIGH! Pull-up resistors are needed to bring the lines back to HIGH when no device is pulling them down.

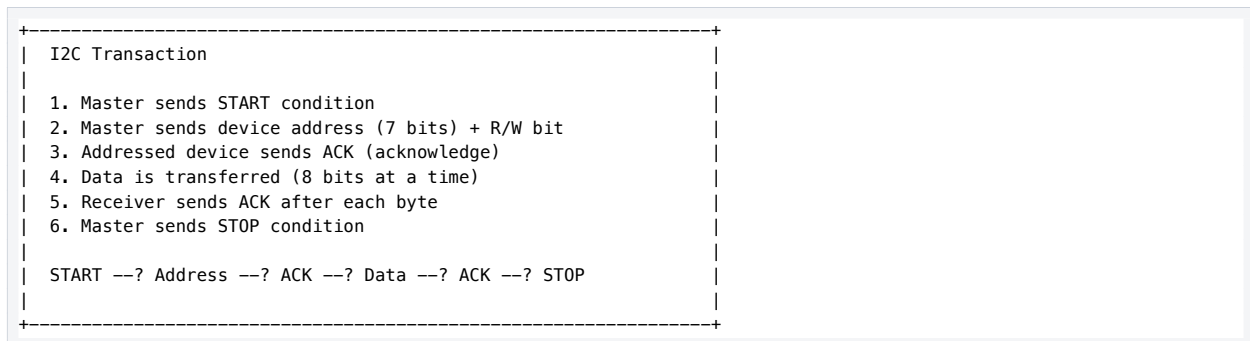
The Pico 2 has internal pull-ups that we can enable with `gpio_pull_up()`.

I2C Addresses

Every I2C device has a unique **7-bit address**. Common addresses:

Device Type	Typical Address
1602 LCD with PCF8574	0x27 or 0x3F
Temperature sensor	0x48
EEPROM	0x50
Real-time clock	0x68

I2C Communication Flow



Part 3: Understanding C Structs

What is a Struct?

A **struct** (short for “structure”) is a way to group related variables together under one name. Think of it like a form with multiple fields:

```
// A struct definition - like a template
struct student {
    char name[50];
    int age;
    float gpa;
};

// Creating a variable of this struct type
struct student alice = {"Alice", 16, 3.8};
```

Why Use Structs?

Instead of passing many separate variables:

```
void print_student(char *name, int age, float gpa); // Messy!
```

You pass one struct:

```
void print_student(struct student s); // Clean!
```

The typedef Keyword

Writing `struct student` everywhere is tedious. The `typedef` keyword creates an alias:

```
typedef struct student student_t;

// Now you can write:
student_t alice; // Instead of: struct student alice;
```

Forward Declaration

Sometimes you need to tell the compiler “this struct exists” before defining it:

```
typedef struct i2c_inst i2c_inst_t; // Forward declaration + alias

// Later, the full definition:
struct i2c_inst {
    i2c_hw_t *hw;
    bool restart_on_next;
};
```

Part 4: Understanding the Pico SDK's I2C Structs

The i2c_inst_t Struct

The Pico SDK uses a struct to represent each I2C controller:

```
struct i2c_inst {
    i2c_hw_t *hw;           // Pointer to hardware registers
    bool restart_on_next;   // SDK internal flag
};
```

What each member means:

Member	Type	Purpose
hw	i2c_hw_t *	Pointer to the actual hardware registers
restart_on_next	bool	Tracks if next transfer needs a restart

The Macro Chain

When you write I2C_PORT in your code, here's what happens:

```

+-----+
| Macro Expansion Chain                                     |
+-----+
| In your code:      #define I2C_PORT i2c1                 |
|                    |                                     |
|                    ?                                     |
| In i2c.h:          #define i2c1 (&i2c1_inst)             |
|                    |                                     |
|                    ?                                     |
| In i2c.c:          i2c_inst_t i2c1_inst = {i2c1_hw, false}; |
|                    |                                     |
|                    ?                                     |
| In i2c.h:          #define i2c1_hw ((i2c_hw_t *)I2C1_BASE) |
|                    |                                     |
|                    ?                                     |
| In addressmap.h:   #define I2C1_BASE 0x40098000           |
+-----+

```

So I2C_PORT eventually becomes a pointer to a struct that contains a pointer to hardware registers at address 0x40098000!

The Hardware Register Pointer

The i2c_hw_t *hw member points to the actual silicon:

```

+-----+
| Memory Map                                               |
+-----+
| Address 0x40098000: I2C1 Hardware Registers              |
| +-----+                                               |
| | Offset 0x00: IC_CON (Control register)                 |
| | Offset 0x04: IC_TAR (Target address register)          |
| | Offset 0x10: IC_DATA_CMD (Data command register)       |
| | ...                                                    |
| +-----+                                               |
| The i2c_hw_t struct maps directly to these registers!   |
+-----+

```

Part 5: The ARM Calling Convention (AAPCS)

How Arguments Are Passed

On ARM Cortex-M, the **ARM Architecture Procedure Call Standard (AAPCS)** defines how functions receive arguments:

Register	Purpose
r0	First argument
r1	Second argument

Register	Purpose
r2	Third argument
r3	Fourth argument
Stack	Fifth+ arguments
r0	Return value

Example: i2c_init(i2c1, 100000)

```
i2c_init(I2C_PORT, 100000);
```

In assembly:

```
ldr r0, [address of i2c1_inst] ; r0 = pointer to struct (first arg)
ldr r1, =0x186A0               ; r1 = 100000 (second arg)
bl i2c_init                    ; Call the function
```

Part 6: Setting Up Your Environment

Prerequisites

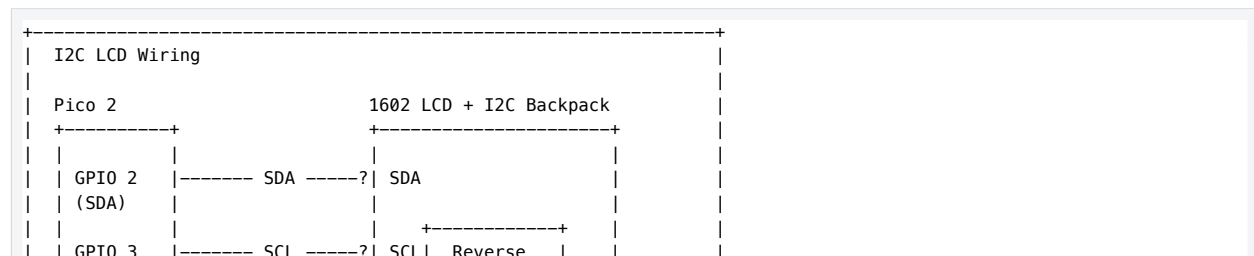
Before we start, make sure you have:

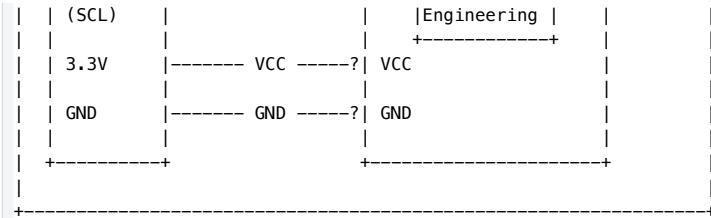
1. A Raspberry Pi Pico 2 board
2. A Raspberry Pi Pico Debug Probe
3. OpenOCD installed and configured
4. GDB (arm-none-eabi-gdb) installed
5. Python installed (for UF2 conversion)
6. A serial monitor (PuTTY, minicom, or screen)
7. A 1602 LCD display with I2C backpack (PCF8574)
8. A hex editor (HxD, ImHex, or similar)
9. The sample project: 0x0017_constants

Hardware Setup

Connect your LCD like this:

LCD Pin	Pico 2 Pin
VCC	3.3V or 5V
GND	GND
SDA	GPIO 2
SCL	GPIO 3





Project Structure

```
Embedded-Hacking/
+-- 0x0017_constants/
|   +-- build/
|   |   +-- 0x0017_constants.uf2
|   |   +-- 0x0017_constants.bin
|   +-- 0x0017_constants.c
|   +-- lcd_1602.h
+-- uf2conv.py
```

Part 7: Hands-On Tutorial - Constants and I2C LCD

Step 1: Review the Source Code

Let's examine the constants code:

File: 0x0017_constants.c

```
#include <stdio.h>
#include <string.h>
#include "pico/stdlib.h"
#include "hardware/i2c.h"
#include "lcd_1602.h"

#define FAV_NUM 42
#define I2C_PORT i2c1
#define I2C_SDA_PIN 2
#define I2C_SCL_PIN 3

const int OTHER_FAV_NUM = 1337;

int main(void) {
    stdio_init_all();

    i2c_init(I2C_PORT, 100000);
    gpio_set_function(I2C_SDA_PIN, GPIO_FUNC_I2C);
    gpio_set_function(I2C_SCL_PIN, GPIO_FUNC_I2C);
    gpio_pull_up(I2C_SDA_PIN);
    gpio_pull_up(I2C_SCL_PIN);

    lcd_i2c_init(I2C_PORT, 0x27, 4, 0x08);
    lcd_set_cursor(0, 0);
    lcd_puts("Reverse");
    lcd_set_cursor(1, 0);
    lcd_puts("Engineering");

    while (true) {
        printf("FAV_NUM: %d\r\n", FAV_NUM);
        printf("OTHER_FAV_NUM: %d\r\n", OTHER_FAV_NUM);
    }
}
```

What this code does:

1. **Lines 7-10:** Define preprocessor macros for constants and I2C configuration
2. **Line 12:** Define a const variable stored in flash
3. **Line 15:** Initialize UART for serial output
4. **Lines 17-21:** Initialize I2C1 at 100kHz, configure GPIO pins, enable pull-ups
5. **Lines 23-27:** Initialize LCD and display “Reverse” on line 0, “Engineering” on line 1
6. **Lines 29-32:** Infinite loop printing both constant values to serial terminal

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 0x0017_constants.uf2 onto the drive
5. The Pico will reboot and start running!

Step 3: Verify It's Working

Check the LCD:

- Line 1 should show: Reverse
- Line 2 should show: Engineering

Check the serial monitor (PuTTY/screen):

```
FAV_NUM: 42
OTHER_FAV_NUM: 1337
FAV_NUM: 42
OTHER_FAV_NUM: 1337
...
```

Part 8: 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\0x0017_constants.elf
```

Connect to target:

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

Step 4: Examine Main Function

Let's examine the main function. Disassemble from the entry point:

```
x/54i 0x10000234
```

You should see output like:

```
(gdb) x/54i 0x1000234
0x1000234 <main>:      push    {r3, lr}
0x1000236 <main+2>:      bl      0x100037fc <stdio_init_all>
0x100023a <main+6>:      ldr     r1, [pc, #104] @ (0x10002a4 <main+112>)
0x100023c <main+8>:      ldr     r0, [pc, #104] @ (0x10002a8 <main+116>)
0x100023e <main+10>:     bl      0x10003cdc <i2c_init>
0x1000242 <main+14>:     movs    r1, #3
0x1000244 <main+16>:     movs    r0, #2
0x1000246 <main+18>:     bl      0x100008f0 <gpio_set_function>
0x100024a <main+22>:     movs    r1, #3
0x100024c <main+24>:     mov     r0, r1
0x100024e <main+26>:     bl      0x100008f0 <gpio_set_function>
0x1000252 <main+30>:     movs    r2, #0
0x1000254 <main+32>:     movs    r1, #1
0x1000256 <main+34>:     movs    r0, #2
0x1000258 <main+36>:     bl      0x1000092c <gpio_set_pulls>
0x100025c <main+40>:     movs    r2, #0
0x100025e <main+42>:     movs    r1, #1
0x1000260 <main+44>:     movs    r0, #3
0x1000262 <main+46>:     bl      0x1000092c <gpio_set_pulls>
0x1000266 <main+50>:     movs    r3, #8
0x1000268 <main+52>:     movs    r2, #4
0x100026a <main+54>:     movs    r1, #39 @ 0x27
0x100026c <main+56>:     ldr     r0, [pc, #56] @ (0x10002a8 <main+116>)
0x100026e <main+58>:     bl      0x100002bc <lcd_i2c_init>
0x1000272 <main+62>:     movs    r1, #0
0x1000274 <main+64>:     mov     r0, r1
0x1000276 <main+66>:     bl      0x100006f4 <lcd_set_cursor>
0x100027a <main+70>:     ldr     r0, [pc, #48] @ (0x10002ac <main+120>)
0x100027c <main+72>:     bl      0x100007f0 <lcd_puts>
0x1000280 <main+76>:     movs    r0, #1
0x1000282 <main+78>:     movs    r1, #0
0x1000284 <main+80>:     bl      0x100006f4 <lcd_set_cursor>
0x1000288 <main+84>:     ldr     r0, [pc, #36] @ (0x10002b0 <main+124>)
0x100028a <main+86>:     bl      0x100007f0 <lcd_puts>
0x100028e <main+90>:     movs    r1, #42 @ 0x2a
0x1000290 <main+92>:     ldr     r0, [pc, #32] @ (0x10002b4 <main+128>)
0x1000292 <main+94>:     bl      0x1000398c <__wrap_printf>
0x1000296 <main+98>:     movw    r1, #1337 @ 0x539
0x100029a <main+102>:     ldr     r0, [pc, #28] @ (0x10002b8 <main+132>)
0x100029c <main+104>:     bl      0x1000398c <__wrap_printf>
0x10002a0 <main+108>:     b.n     0x1000028e <main+90>
0x10002a2 <main+110>:     nop
0x10002a4 <main+112>:     strh    r0, [r4, #52] @ 0x34
0x10002a6 <main+114>:     movs    r1, r0
0x10002a8 <main+116>:     lsls    r4, r5, #24
0x10002aa <main+118>:     movs    r0, #0
0x10002ac <main+120>:     subs    r6, #232 @ 0xe8
0x10002ae <main+122>:     asrs    r0, r0, #32
0x10002b0 <main+124>:     subs    r6, #240 @ 0xf0
0x10002b2 <main+126>:     asrs    r0, r0, #32
0x10002b4 <main+128>:     subs    r6, #252 @ 0xfc
0x10002b6 <main+130>:     asrs    r0, r0, #32
0x10002b8 <main+132>:     subs    r7, #12
0x10002ba <main+134>:     asrs    r0, r0, #32
```

Step 5: Set a Breakpoint at Main

```
b *0x1000234
c
```

Step 6: Find the #define Constant (FAV_NUM)

Step through to the printf call and examine the registers:

```
x/20i 0x100028e
```

Look for:

```
...
0x1000028e <main+90>:      movs    r1, #42 @ 0x2a
...
```

The #define constant is embedded directly as an immediate value in the instruction!

Step 7: Find the const Variable (OTHER_FAV_NUM)

Continue examining the loop body:

```
(gdb) x/5i 0x10000296
```

Look for this instruction:

```
...
0x10000296 <main+98>:      movw    r1, #1337      @ 0x539
...
```

Surprise! The const variable is ALSO embedded as an immediate value - not loaded from memory! The compiler saw that OTHER_FAV_NUM is never address-taken (&OTHER_FAV_NUM is never used), so it optimized the const the same way as #define - as a constant embedded directly in the instruction.

The difference is the instruction encoding:

- FAV_NUM (42): movs r1, #0x2a - 16-bit Thumb instruction (values 0-255)
- OTHER_FAV_NUM (1337): movw r1, #0x539 - 32-bit Thumb-2 instruction (values 0-65535)

Tip: Why movw instead of movs? The value 1337 doesn't fit in 8 bits (max 255), so the compiler uses movw (Move Wide) which can encode any 16-bit immediate (0-65535) in a 32-bit instruction.

Step 8: Examine the Literal Pool

The literal pool after the loop contains addresses and constants that are too large for regular instruction immediates. Let's examine it:

```
(gdb) x/6wx 0x100002a4
0x100002a4 <main+112>:  0x000186a0    0x2000062c    0x10003ee8    0x10003ef0
0x100002b4 <main+128>:  0x10003efc    0x10003f0c
```

These are the values that ldr rN, [pc, #offset] instructions load:

Literal Pool Addr	Value	Used By
0x100002a4	0x000186A0	I2C baudrate (100000)
0x100002a8	0x2000062C	&i2c1_inst (I2C struct in RAM)
0x100002ac	0x10003EE8	"Reverse" string address
0x100002b0	0x10003EF0	"Engineering" string address
0x100002b4	0x10003EFC	"FAV_NUM: %d\r\n" format str
0x100002b8	0x10003F0C	"OTHER_FAV_NUM: %d\r\n" fmt

Tip: Why does the disassembly at 0x100002a4 show strh r0, [r4, #52] instead of data? Same reason as Week 6 - GDB's x/i tries to decode raw data as instructions. Use x/wx to see the actual word values or we can also use x/x.

```
(gdb) x/x 0x100002a4
0x100002a4 <main+112>:  0x000186a0
```

```
(gdb) x/x 0x100002a8
0x100002a8 <main+116>: 0x2000062c
(gdb) x/x 0x100002ac
0x100002ac <main+120>: 0x10003ee8
(gdb) x/x 0x100002b0
0x100002b0 <main+124>: 0x10003ef0
(gdb) x/x 0x100002b4
0x100002b4 <main+128>: 0x10003efc
(gdb) x/x 0x100002b8
0x100002b8 <main+132>: 0x10003f0c
```

Step 9: Examine the I2C Struct

Find the i2c1_inst struct address loaded into r0 before i2c_init:

```
x/2wx 0x2000062c
```

You should see:

```
0x2000062c <i2c1_inst>: 0x40098000      0x00000000
```

Step 10: Examine the LCD String Literals

Find the strings passed to lcd_puts:

```
x/s 0x10003ee8
```

Output:

```
0x10003ee8:      "Reverse"
```

```
x/s 0x10003ef0
```

Output:

```
0x10003ef0:      "Engineering"
```

Step 11: Step Through I2C Initialization

Step through instructions and watch the I2C setup:

```
(gdb) b *0x1000023e
(gdb) c
(gdb) i r r0 r1
r0      0x2000062c      536872492
r1      0x186a0         100000
```

Part 9: Understanding the Assembly

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

Step 12: Analyze #define vs const in Assembly

From GDB, we discovered something interesting - **both constants ended up as instruction immediates!**

For FAV_NUM (42) - a #define macro:

```
0x1000028e <+90>:      movs      r1, #42 @ 0x2a
```

The value 42 is embedded directly in a 16-bit Thumb instruction. This is expected - #define is text replacement, so the compiler never sees FAV_NUM, only 42.

For OTHER_FAV_NUM (1337) - a const variable:

```
0x10000296 <+98>:    movw    r1, #1337    @ 0x539
```

The value 1337 is ALSO embedded directly in an instruction - but this time a 32-bit Thumb-2 movw because the value doesn't fit in 8 bits.

Why wasn't const stored in memory? In theory, `const int OTHER_FAV_NUM = 1337` creates a variable in the `.rodata` section. But the compiler optimized it away because:

1. We never take the address of `OTHER_FAV_NUM` (no `&OTHER_FAV_NUM`)
2. The value fits in a 16-bit movw immediate
3. Loading from an immediate is faster than loading from memory

Tip: Key takeaway for reverse engineering: Don't assume const variables will appear as memory loads. Modern compilers aggressively inline constant values. The C keyword `const` is a **source-level** concept - the compiler may or may not honor it in the final binary.

Step 13: Analyze the I2C Struct Layout

In GDB, we examined the `i2c1_inst` struct at `0x2000062c`:

```
(gdb) x/2wx 0x2000062c
0x2000062c <i2c1_inst>: 0x40098000    0x00000000
```

This maps to the `i2c_inst_t` struct:

```
-----+-----
| i2c_inst_t at 0x2000062c |
|-----+-----|
| Offset  Type      Name      Value |
|-----+-----+-----|
| 0x00    i2c_hw_t *  hw       0x40098000 |
| 0x04    bool       restart_on_next 0x00 (false) |
|-----+-----+-----|
|-----+-----|
```

The first member (`hw`) points to `0x40098000` - the I2C1 hardware register base. This is the end of the macro chain: `I2C_PORT -> i2c1 -> &i2c1_inst -> hw -> 0x40098000`.

Step 14: Locate the String Literals

We found the LCD strings in flash memory:

```
(gdb) x/s 0x10003ee8
0x10003ee8:    "Reverse"

(gdb) x/s 0x10003ef0
0x10003ef0:    "Engineering"
```

These are stored consecutively in the `.rodata` section. Note the addresses - we'll need them for patching.

Part 10: 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 and read-only data live. Trying `set *(char *)0x1000028e = 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 15: 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\0x0017_constants\build\
4. Open 0x0017_constants.bin

Step 16: 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 0x1000028e -> file offset 0x28E (654 in decimal)
- Address 0x10003ee8 -> file offset 0x3EE8 (16104 in decimal)

Step 17: Understand FAV_NUM Encoding (movs - 16-bit Thumb)

From our GDB analysis, we know the instruction at 0x1000028e is:

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

In HxD, use **Ctrl+G** to navigate to file offset 28E and verify you see the byte 2A followed by 21.

?? 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` (42). If you wanted to change this to 43, you'd change 2A to 2B.

Step 18: Understand OTHER_FAV_NUM Encoding (movw - 32-bit Thumb-2)

From GDB, we found the `movw r1, #1337` instruction at 0x10000296. Examine the exact bytes:

```
(gdb) x/4bx 0x10000296
0x10000296 <main+98>:  0x40    0xf2    0x39    0x51
```

This is the 32-bit Thumb-2 encoding of `movw r1, #0x539` (1337). The bytes break down as:

```
+-----+
| movw r1, #0x539 -> bytes: 40 F2 39 51 |
+-----+
| Byte 0: 0x40 -?? |
| Byte 1: 0xF2 -+ First halfword (opcode + upper imm bits) |
| Byte 2: 0x39 ---- Lower 8 bits of immediate (imm8) ?? CHANGE |
| Byte 3: 0x51 ---- Destination register (r1) + upper imm bits |
+-----+
| imm16 = 0x0539 = 1337 decimal |
| imm8 field = 0x39 (lower 8 bits of the value) |
+-----+
```

The file offset is $0x10000296 - 0x10000000 = 0x296$. The imm8 byte is the 3rd byte of the instruction: $0x296 + 2 = 0x298$.

To change `movw r1, #1337` to `movw r1, #1344`:

1. In HxD, press **Ctrl+G** (Go to offset)
2. Enter offset: 298 (the third byte of the 4-byte instruction)
3. You should see the byte 39 at this position
4. Change 39 to 40

?? **Why offset 0x298 and not 0x296?** The lower 8 bits of the immediate (`imm8`) are in the **third byte** of the 4-byte `movw` instruction. The instruction starts at file offset 0x296, so `imm8` is at `0x296 + 2 = 0x298`. Changing 0x39 to 0x40 changes the value from 0x539 (1337) to 0x540 (1344).

Step 19: Hack - Change LCD Text from “Reverse” to “Exploit”

IMPORTANT: The new string must be the **same length** as the original! “Reverse” and “Exploit” are both 7 characters - perfect!

From our GDB analysis in Step 10, we found the string at 0x10003ee8. File offset = `0x10003ee8 - 0x10000000 = 0x3EE8`.

1. In HxD, press **Ctrl+G** and enter offset: 3EE8
2. You should see the bytes for “Reverse”: 52 65 76 65 72 73 65 00
3. Change the bytes to spell “Exploit”: 45 78 70 6c 6f 69 74 00

ASCII Reference:

Character	Hex
E	0x45
x	0x78
p	0x70
l	0x6c
o	0x6f
i	0x69
t	0x74

Step 20: Save the Patched Binary

1. Click **File** -> **Save As**
2. Save as `0x0017_constants-h.bin` in the build directory
3. Close the hex editor

Part 11: Converting and Flashing the Hacked Binary

Step 21: Convert to UF2 Format

Open a terminal and navigate to your project directory:

```
cd C:\Users\flare-vm\Desktop\Embedded-Hacking-main\0x0017_constants
```

Run the conversion command:

```
python ..\uf2conv.py build\0x0017_constants-h.bin --base 0x10000000 --family 0xe48bff59 --output build\hacked.uf2
```

Step 22: 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. Check your LCD and serial monitor

Step 23: Verify the Hack

Check the LCD:

- Line 1 should now show: Exploit (instead of “Reverse”)
- Line 2 should still show: Engineering

Check the serial monitor:

```
FAV_NUM: 42
OTHER_FAV_NUM: 1337
FAV_NUM: 42
OTHER_FAV_NUM: 1337
...
```

The numbers are unchanged - we only patched the LCD string!

BOOM! We successfully changed the LCD text from “Reverse” to “Exploit” without access to the source code!

Part 12: Summary and Review

What We Accomplished

1. **Learned about constants** - #define macros vs const variables
2. **Understood I2C communication** - Two-wire protocol for peripheral communication
3. **Explored C structs** - How the Pico SDK abstracts hardware
4. **Mastered the macro chain** - From I2C_PORT to 0x40098000
5. **Examined structs in GDB** - Inspected memory layout of i2c_inst_t
6. **Analyzed instruction encodings** - Both movs (8-bit) and movw (16-bit) immediates in the hex editor
7. **Patched a string literal** - Changed LCD display text from “Reverse” to “Exploit”

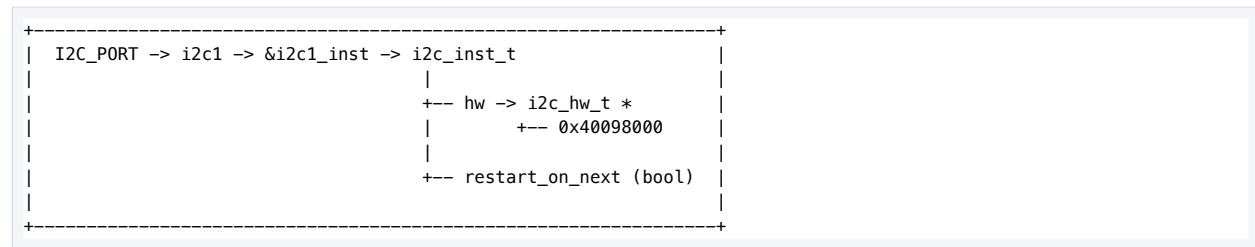
#define vs const Summary

```
+-----+
| #define FAV_NUM 42                                     |
| -----|
| - Text replacement at compile time                    |
| - No memory allocated                                |
| - Cannot take address (&FAV_NUM is invalid)          |
| - In binary: value appears as immediate (movs r1, #0x2a)|
| - To hack: patch the instruction operand              |
|-----+
| const int OTHER_FAV_NUM = 1337                         |
| -----|
| - Theoretically in .rodata, but compiler optimized it away|
| - Value embedded as immediate: movw r1, #0x539 (32-bit instr)|
| - Optimization: compiler saw &OTHER_FAV_NUM is never used|
| - In binary: immediate in instruction, same as #define!|
| - To hack: patch instruction operand (imm8 byte at offset +2)|
|-----+
```

I2C Configuration Summary

```
+-----+
| I2C Setup Steps                                       |
|-----+
| 1. i2c_init(i2c1, 100000)      - Initialize at 100kHz  |
| 2. gpio_set_function(pin, I2C) - Assign pins to I2C   |
| 3. gpio_pull_up(sda_pin)       - Enable SDA pull-up   |
| 4. gpio_pull_up(scl_pin)       - Enable SCL pull-up   |
| 5. lcd_i2c_init(...)           - Initialize the device |
|-----+
```

The Struct Chain



Key Memory Addresses

Address	Description
0x10000234	main() entry point
0x1000028e	FAV_NUM value in instruction
0x10000296	OTHER_FAV_NUM value in instruction
0x10003ee8	“Reverse” string literal (example)
0x40098000	I2C1 hardware registers base
0x2000062C	i2c1_inst struct in SRAM

Key Takeaways

1. **#define is text replacement** - It happens before compilation, no memory used.
2. **const creates real variables** - Stored in .rodata, takes memory, has an address.
3. **I2C uses two wires** - SDA for data, SCL for clock, pull-ups required.
4. **Structs group related data** - The SDK uses them to abstract hardware.
5. **Macros can chain** - I2C_PORT -> i2c1 -> &i2c1_inst -> hardware pointer.
6. **ARM passes args in registers** - r0-r3 for first four arguments.
7. **GDB reveals struct layouts** - Examine memory to understand data organization.
8. **String hacking requires same length** - Or you'll corrupt adjacent data!
9. **Constants aren't constant** - With binary patching, everything can change!
10. **Compiler optimization changes code** - gpio_pull_up becomes gpio_set_pulls.

Glossary

Term	Definition
#define	Preprocessor directive for text replacement
AAPCS	ARM Architecture Procedure Call Standard
const	Keyword marking a variable as read-only
Forward Declaration	Telling compiler a type exists before defining it

Term	Definition
I2C	Inter-Integrated Circuit - two-wire serial protocol
Immediate Value	A constant embedded directly in an instruction
Open-Drain	Output that can only pull low, not drive high
PCF8574	Common I2C I/O expander chip used in LCD backpacks
Preprocessor	Tool that processes code before compilation
Pull-Up Resistor	Resistor that holds a line HIGH by default
SCL	Serial Clock - I2C timing signal
SDA	Serial Data - I2C data line
Struct	User-defined type grouping related variables
typedef	Creates an alias for a type

Additional Resources

I2C Timing Reference

Speed Mode	Maximum Frequency
Standard	100 kHz
Fast	400 kHz
Fast Plus	1 MHz

Common I2C Addresses

Device	Address
PCF8574 LCD (default)	0x27
PCF8574A LCD	0x3F
DS3231 RTC	0x68
BMP280 Sensor	0x76/0x77
SSD1306 OLED	0x3C/0x3D

Key ARM Instructions for Constants

Instruction	Description
<code>movs rN, #imm</code>	Load small immediate (0-255) directly
<code>ldr rN, [pc, #off]</code>	Load larger value from literal pool
<code>ldr rN, =value</code>	Pseudo-instruction for loading any constant

RP2350 I2C Memory Map

Address	Description
0x40090000	I2Co hardware registers
0x40098000	I2C1 hardware registers

Remember: When you see complex nested structures in a binary, take your time to understand the hierarchy. Use GDB to examine struct layouts in memory and trace pointer chains. And always remember - even “constants” can be hacked!

Happy hacking! ?