
EMBEDDED HACKING

Reverse Engineering the Raspberry Pi Pico 2

Week 11: Structures and Functions in Embedded Systems: Debugging and Hacking w/ IR Remote Control and NEC Protocol 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 C structures (structs) and how they organize related data
 - Know how structs are represented in memory and assembly code
 - Understand the NEC infrared (IR) protocol for remote control communication
 - Create and use functions with parameters and return values
 - Identify struct member access patterns in Ghidra
 - Recognize how compilers “flatten” structs into individual operations
 - Hack GPIO pin assignments to swap LED behavior
 - Understand the security implications of log/behavior desynchronization
 - Analyze .elf files in addition to .bin files in Ghidra
-

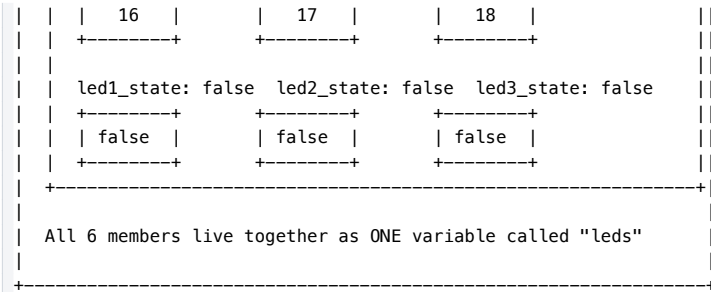
Part 1: Understanding C Structures (Structs)

What is a Struct?

A **structure** (or **struct**) is a user-defined data type that groups related variables together under one name. Think of it like a form with multiple fields - each field can hold different types of data, but they all belong together.

```
// Define a struct type
typedef struct {
    uint8_t led1_pin; // GPIO pin for LED 1
    uint8_t led2_pin; // GPIO pin for LED 2
    uint8_t led3_pin; // GPIO pin for LED 3
    bool led1_state;  // Is LED 1 on?
    bool led2_state;  // Is LED 2 on?
    bool led3_state;  // Is LED 3 on?
} simple_led_ctrl_t;
```

```
+-----+
| Structure as a Container |
| |                         |
| simple_led_ctrl_t leds  |
| +-----+ |
| | led1_pin: 16   led2_pin: 17   led3_pin: 18 | | |
| | +-----+   +-----+   +-----+ |
| | |                         | |
```



Why Use Structs?

Without Structs (Messy!)

```

uint8_t led1_pin = 16;
uint8_t led2_pin = 17;
uint8_t led3_pin = 18;
bool led1_state = false;
bool led2_state = false;
bool led3_state = false;

```

With Structs (Clean!)

```

simple_led_ctrl_t leds;
leds.led1_pin = 16;
leds.led2_pin = 17;
leds.led3_pin = 18;
leds.led1_state = false;
leds.led2_state = false;
leds.led3_state = false;
... (all in one container!)

```

Benefits of Structs:

1. **Organization** - Related data stays together
2. **Readability** - Code is easier to understand
3. **Maintainability** - Changes are easier to make
4. **Scalability** - Easy to add more LEDs or features
5. **Passing to Functions** - Pass one struct instead of many variables

Part 2: Struct Memory Layout

How Structs are Stored in Memory

When you create a struct, the compiler places each member in consecutive memory locations:

Memory Layout of simple_led_ctrl_t			
Address	Member	Size	Value
0x2000000	led1_pin	1 byte	16 (0x10)
0x2000001	led2_pin	1 byte	17 (0x11)
0x2000002	led3_pin	1 byte	18 (0x12)
0x2000003	led1_state	1 byte	0 (false)
0x2000004	led2_state	1 byte	0 (false)
0x2000005	led3_state	1 byte	0 (false)
Total struct size: 6 bytes			

Accessing Struct Members

Use the **dot operator** (.) to access members:

```
simple_led_ctrl_t leds;

// Set values
leds.led1_pin = 16;
leds.led1_state = true;

// Read values
printf("Pin: %d\n", leds.led1_pin);
```

Pointer to Struct (Arrow Operator)

When you have a **pointer** to a struct, use the **arrow operator** ($\rightarrow$):

```
simple_led_ctrl_t leds;
simple_led_ctrl_t *ptr = &leds; // Pointer to the struct

// These are equivalent:
leds.led1_pin = 16;           // Using dot with struct variable
ptr->led1_pin = 16;           // Using arrow with pointer
(*ptr).led1_pin = 16;         // Dereferencing then dot (same thing)
```

Dot vs Arrow Operator
struct_variable.member <-- Use with actual struct
pointer_to_struct->member <-- Use with pointer to struct
The arrow ($\rightarrow$) is shorthand for (*pointer).member

Part 3: Designated Initializers

Clean Struct Initialization

C allows you to initialize struct members by name using **designated initializers**:

```
simple_led_ctrl_t leds = {
    .led1_pin = 16,
    .led2_pin = 17,
    .led3_pin = 18,
    .led1_state = false,
    .led2_state = false,
    .led3_state = false
};
```

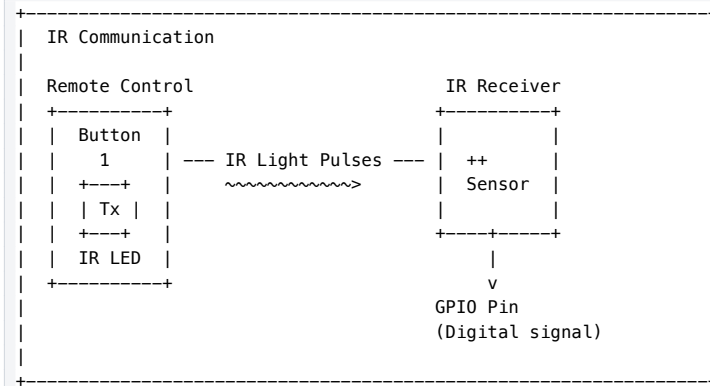
Benefits:

- Clear which value goes to which member
 - Order doesn't matter (can rearrange lines)
 - Self-documenting code
 - Easy to add new members later
-

Part 4: Understanding the NEC IR Protocol

What is Infrared (IR) Communication?

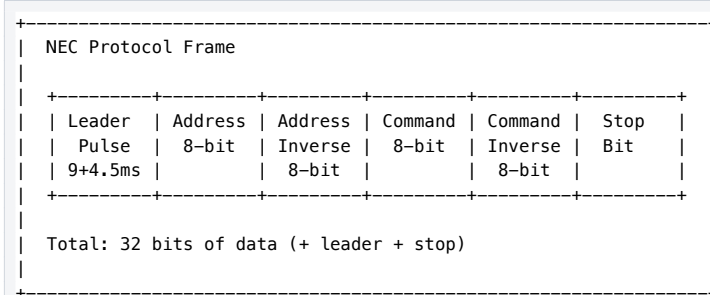
Infrared communication uses invisible light pulses to send data. Your TV remote uses IR to send commands to your TV. The LED in the remote flashes on and off very quickly in specific patterns that represent different buttons.



The NEC Protocol

NEC is one of the most common IR protocols. When you press a button, the remote sends:

1. **Leader pulse** - 9ms HIGH, 4.5ms LOW (says “attention!”)
2. **Address** - 8 bits identifying the device
3. **Address Inverse** - 8 bits (for error checking)
4. **Command** - 8 bits for the button pressed
5. **Command Inverse** - 8 bits (for error checking)



NEC Command Codes for Our Remote

Button	NEC Command Code	Hex Value
1	0x0C	12
2	0x18	24
3	0x5E	94

Note: Different remotes have different codes. These are specific to our example remote.

Part 5: Understanding Functions in C


```

Assembly Level (Flattened):
+-----+
|  movs r0, #16      ; Just the VALUE, no struct reference  |
|  bl  gpio_init    |
|  movs r0, #17      ; Next value directly                 |
|  bl  gpio_init    |
|  movs r0, #18      ; Next value directly                 |
|  bl  gpio_init    |
+-----+

The struct abstraction DISAPPEARS at the assembly level!
We just see individual values being loaded and used.

```

Why This Matters for Reverse Engineering

- In Ghidra, you won't always see "struct" - just individual values
- You must recognize PATTERNS (sequential values like 16, 17, 18)
- Understanding flattening helps you reconstruct the original struct

Part 8: 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. Ghidra installed (for static analysis)
4. Python installed (for UF2 conversion)
5. A serial monitor (PuTTY, minicom, or screen)
6. An IR receiver module (like VS1838B)
7. An IR remote control (any NEC-compatible remote)
8. Three LEDs (red, green, yellow) with resistors
9. The sample projects: `0x0023_structures` and `0x0026_functions`

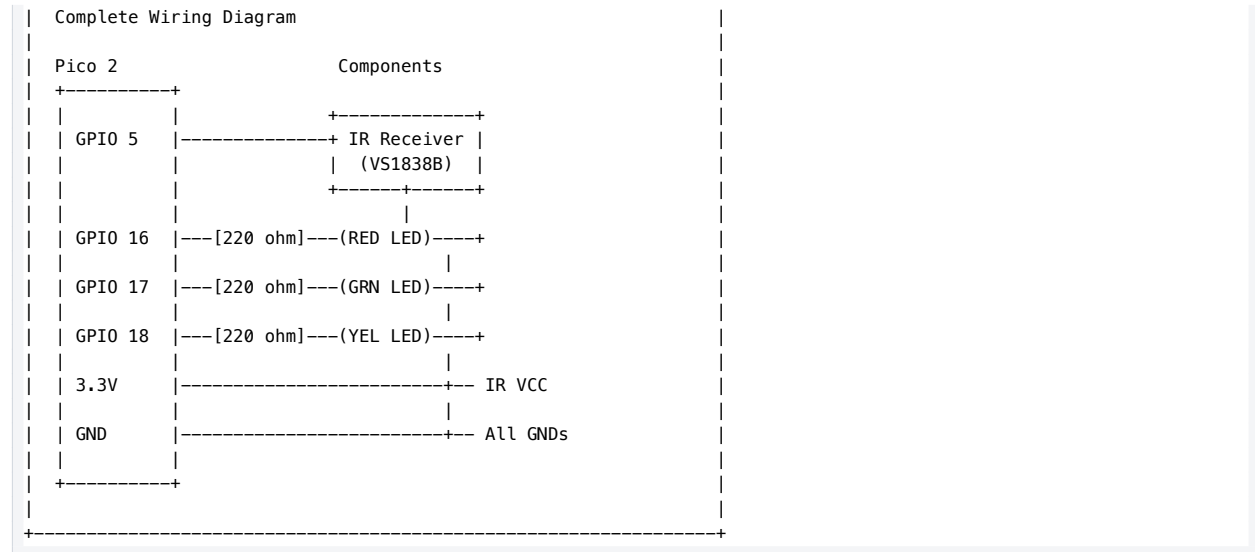
Hardware Setup

IR Receiver Wiring:

IR Receiver Pin	Pico 2 Pin
VCC	3.3V
GND	GND
OUT/DATA	GPIO 5

LED Wiring:

LED	GPIO Pin	Resistor
Red	GPIO 16	220-330 ohm
Green	GPIO 17	220-330 ohm
Yellow	GPIO 18	220-330 ohm



Project Structure

```

Embedded-Hacking/
+-- 0x0023_structures/
|   +-- build/
|   |   +-- 0x0023_structures.uf2
|   |   +-- 0x0023_structures.bin
|   +-- main/
|   |   +-- 0x0023_structures.c
|   +-- ir.h
+-- 0x0026_functions/
|   +-- build/
|   |   +-- 0x0026_functions.uf2
|   |   +-- 0x0026_functions.bin
|   |   +-- 0x0026_functions.elf
|   +-- main/
|   |   +-- 0x0026_functions.c
|   +-- ir.h
+-- uf2conv.py

```

Part 9: Hands-On Tutorial - Structures Code

Step 1: Review the Source Code

Let's examine the structures code:

File: 0x0023_structures.c

```

#include <stdio.h>
#include <stdbool.h>
#include "pico/stdlib.h"
#include "ir.h"

#define IR_PIN 5

typedef struct {
    uint8_t led1_pin;
    uint8_t led2_pin;
    uint8_t led3_pin;
    bool led1_state;
    bool led2_state;
    bool led3_state;
}

```

```

} simple_led_ctrl_t;

int main(void) {
    stdio_init_all();

    simple_led_ctrl_t leds = {
        .led1_pin = 16,
        .led2_pin = 17,
        .led3_pin = 18,
        .led1_state = false,
        .led2_state = false,
        .led3_state = false
    };

    gpio_init(leds.led1_pin); gpio_set_dir(leds.led1_pin, GPIO_OUT);
    gpio_init(leds.led2_pin); gpio_set_dir(leds.led2_pin, GPIO_OUT);
    gpio_init(leds.led3_pin); gpio_set_dir(leds.led3_pin, GPIO_OUT);

    ir_init(IR_PIN);
    printf("IR receiver on GPIO %d ready\n", IR_PIN);

    while (true) {
        int key = ir_getkey();
        if (key >= 0) {
            printf("NEC command: 0x%02X\n", key);

            // Turn all off first
            leds.led1_state = false;
            leds.led2_state = false;
            leds.led3_state = false;

            // Check NEC codes
            if (key == 0x0C) leds.led1_state = true; // GPIO16
            if (key == 0x18) leds.led2_state = true; // GPIO17
            if (key == 0x5E) leds.led3_state = true; // GPIO18

            // Apply states
            gpio_put(leds.led1_pin, leds.led1_state);
            gpio_put(leds.led2_pin, leds.led2_state);
            gpio_put(leds.led3_pin, leds.led3_state);

            sleep_ms(10);
        } else {
            sleep_ms(1);
        }
    }
}

```

Step 2: Understand the Program Flow

Program Flow
1. Initialize UART (stdio_init_all)
2. Create LED struct with pins 16, 17, 18
3. Initialize GPIO pins as outputs
4. Initialize IR receiver on GPIO 5
5. Enter infinite loop:
a. Check for IR key press
b. If key received:
- Print the NEC command code
- Turn all LEDs off
- Check which button: 0x0C, 0x18, or 0x5E
- Turn on the matching LED
- Apply states to GPIO pins
c. Sleep briefly and repeat

Step 3: 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 0x0023_structures.uf2 onto the drive
5. The Pico will reboot and start running!

Step 4: Verify It's Working**Open PuTTY (115200 baud) and test:**

- Press "1" on remote -> Red LED lights, terminal shows NEC command: 0x0C
- Press "2" on remote -> Green LED lights, terminal shows NEC command: 0x18
- Press "3" on remote -> Yellow LED lights, terminal shows NEC command: 0x5E

Part 10: Debugging with GDB (Structures)**Step 5: Start OpenOCD (Terminal 1)**

Open a terminal and 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"
```

You should see output indicating OpenOCD connected successfully to your Pico 2 via the Debug Probe.

Step 6: Start GDB (Terminal 2)

Open a **new terminal** and launch GDB with the binary:

```
arm-none-eabi-gdb build\0x0023_structures.elf
```

Step 7: Connect to the Remote Target

In GDB, connect to OpenOCD:

```
target extended-remote :3333
```

Step 8: Halt the Running Binary

Stop the processor:

```
monitor halt
```

Step 9: Examine Main Function

Disassemble around main to see struct initialization:

```
disassemble 0x10000234,+200
```

Look for the struct member initialization sequence (mov instructions with values 16, 17, 18).

Step 10: Set a Breakpoint at Main

```
break *0x10000234
```

Reset and run to hit the breakpoint:

```
monitor reset halt
continue
```

Step 11: Observe the Flattened Struct

Instead of allocating the struct on the stack, the compiler completely **flattened and optimized** it out! There is no `sub sp` instruction for the struct.

Examine the main disassembly again:

```
disassemble 0x10000234,+200
```

Notice how values 16, 17, and 18 are just loaded directly into registers (`movs r0, #16`, etc.) and passed to functions. The struct abstraction is entirely gone.

Step 12: Watch GPIO Initialization

Set a breakpoint on the first `gpio_init` call and watch the LED pin get initialized:

```
break *0x1000023c
continue
info registers r0
```

You should see `r0 = 16`. If you set breakpoints on the subsequent calls (`0x1000024c`, `0x10000258`) and continue, you will see 17 and 18.

Step 13: Examine IR Key Processing

Because `ir_getkey` is polled in a loop and returns `-1` constantly when no key is pressed, breaking at the immediate return (`0x10000274`) will just flood you with `-1`s!

Instead, set a breakpoint *inside* the `if (key >= 0)` block at `0x10000278`. This is right after the `blt.n` conditional branch that skips over the processing logic when no key is pressed:

```
break *0x10000278
continue
```

Now press a button on the remote. The program will halt. At this point, the key value has been moved into `r4` (from the `subs r4, r0, #0` instruction earlier):

```
info registers r4
```

You'll see the decimal value of the NEC code (e.g., 12 for `0x0C`, 24 for `0x18`, or 94 for `0x5E`).

Step 14: Watch the Conditional Checks

Let's look at the comparisons where the code checks which button was pressed:

```
break *0x10000280
continue
```

(Note: If the program doesn't halt immediately, press a button on the remote to trigger the breakpoint!)

```
x/8i $pc
```

Notice the compiler uses decimal comparisons: `cmp r4, #12` (which is `0x0c`), `cmp r4, #24` (which is `0x18`), and a subtraction trick `sub.w r4, r4, #94` (which is `0x5e`) to determine which button was pressed.

Step 15: Observe Inlined GPIO Operations

The compiler even inlined the `gpio_put` calls! First, let's clear any old breakpoints so we don't accidentally stop somewhere else (like you might have seen if you hit an old breakpoint first!):

```
delete
break *0x10000298
continue
```

(Note: Again, press a button on the remote if it doesn't halt immediately!)

Now check the registers:

```
info registers r1 r2 r3 r4
```

Depending on which button you pressed, one of the state registers will be 1 (ON) and the others will be 0 (OFF):

- `r1` = State for the **Red** LED (GPIO 16)
- `r3` = State for the **Green** LED (GPIO 17)
- `r4` = State for the **Yellow** LED (GPIO 18)
- `r2` = 16 (`0x10`), which is the starting pin number being written to.

(Note: Since you are paused here, the physical LEDs haven't updated yet! They will only change after these `mcr` instructions execute.)

Instead of branching to `gpio_put`, it uses direct hardware instructions (disassembled as `mcr` on the RP2350) to write the states directly to the GPIO hardware.

Step 16: Exit GDB

When done exploring:

```
quit
```

Part 11: Setting Up Ghidra for Structures

Step 17: Start Ghidra

Open a terminal and type:

```
ghidraRun
```

Step 18: Create a New Project

1. Click **File** -> **New Project**
2. Select **Non-Shared Project**
3. Click **Next**
4. Enter Project Name: `0x0023_structures`
5. Click **Finish**

Step 19: Import the Binary

1. Navigate to the `0x0023_structures/build/` folder
2. **Drag and drop** the `.bin` file into Ghidra's project window

Step 20: Configure the Binary Format

Click the three dots (...) next to “Language” and:

1. Search for “Cortex”
2. Select **ARM Cortex 32 little endian default**
3. Click **OK**

Click the “Options...” button and:

1. Change **Block Name** to .text
2. Change **Base Address** to 10000000
3. Click **OK**

Step 21: Analyze the Binary

1. Double-click on the file in the project window
2. A dialog asks “Analyze now?” - Click **Yes**
3. Use default analysis options and click **Analyze**

Wait for analysis to complete.

Part 12: Resolving Functions - Structures Project**Step 22: Navigate to Main**

1. Press G (Go to address) and type 10000234
2. Right-click -> **Edit Function Signature**
3. Change to: int main(void)
4. Click **OK**

Step 23: Resolve stdio_init_all

At address 0x10000236:

1. Double-click on the called function
2. Right-click -> **Edit Function Signature**
3. Change to: bool stdio_init_all(void)
4. Click **OK**

Step 24: Identify gpio_init from Struct Pattern

Look for three consecutive calls with values 16, 17, 18:

```
1000023a 10 20      movs      r0,#0x10
1000023c 00 f0 8c f9 bl      FUN_10000558      ; gpio_init

1000024a 11 20      movs      r0,#0x11
1000024c 00 f0 84 f9 bl      FUN_10000558      ; gpio_init

10000256 12 20      movs      r0,#0x12
10000258 00 f0 7e f9 bl      FUN_10000558      ; gpio_init
```

This pattern reveals the struct members! Update the function signature:

1. Right-click on FUN_10000558 -> **Edit Function Signature**
2. Change to: void gpio_init(uint gpio)
3. Click **OK**

Step 25: Resolve ir_init

Look for a function call with GPIO 5:

```

10000262 05 20      movs      r0,#0x5
10000264 00 f0 38 f8 bl      FUN_100002d8      ; ir_init

```

1. Right-click on FUN_100002d8 -> **Edit Function Signature**
2. Change to: void ir_init(uint pin)
3. Click **OK**

Step 26: Resolve printf

Right after ir_init, look for the “IR receiver on GPIO” string being loaded and passed to a function:

```

1000026a 19 48      ldr      r0=>s_IR_receiver_on_GPIO_%d...
1000026c 03 f0 46 f9 bl      FUN_100034fc      ; printf

```

1. Right-click on FUN_100034fc -> **Edit Function Signature**
2. Change to: int printf(char *format,...)
3. Check the **Varargs** checkbox
4. Click **OK**

Step 27: Resolve ir_getkey

Look for a function that returns a value checked against conditions:

```

10000270 00 f0 46 f8 bl      FUN_10000300      ; Call ir_getkey
10000274 04 1e      subs      r4,r0,#0x0      ; Check if >= 0
10000276 1e db      blt      LAB_100002b6      ; If negative, no key pressed

```

1. Right-click on FUN_10000300 -> **Edit Function Signature**
2. Change to: int ir_getkey(void)
3. Click **OK**

Step 28: Resolve sleep_ms

Look for calls with 10 (0x0A) or 1 (0x01):

```

100002a8 0a 20      movs      r0,#0xa
100002aa 00 f0 81 fe bl      FUN_10000fb0      ; sleep_ms

```

1. Right-click on FUN_10000fb0 -> **Edit Function Signature**
2. Change to: void sleep_ms(uint ms)
3. Click **OK**

Part 13: Recognizing Struct Patterns in Assembly

Step 29: Identify GPIO Set Direction

After each gpio_init, look for direction setting:

```

10000240 4f f0 01 04 mov.w      r4,#0x1      ; direction = output
10000244 10 23      movs      r3,#0x10      ; GPIO 16
10000246 44 ec 44 30 mcrr      p0,0x4,r3,r4,cr4 ; Configure GPIO direction register

```

This is the compiler’s heavily optimized version of gpio_set_dir(pin, GPIO_OUT).

Step 30: Map the Struct Members

Create a mental (or written) map:

Struct Member Mapping			
Assembly Value	->	Struct Member	-> Physical LED
0x10 (16)	->	led1_pin	-> Red LED
0x11 (17)	->	led2_pin	-> Green LED
0x12 (18)	->	led3_pin	-> Yellow LED
NEC Code	->	State Member	-> Action
0x0C	->	led1_state=true	-> Red LED ON
0x18	->	led2_state=true	-> Green LED ON
0x5E	->	led3_state=true	-> Yellow LED ON

Part 14: Hacking Structures

Step 31: Enable Instruction Patching

We will use Ghidra's **Patch Instruction** feature to modify the assembly directly instead of editing raw bytes.

Step 32: Swap LED Pin Assignments

We'll swap the red and green LED pins to reverse their behavior! Because the compiler fully flattened the struct, modifying the gpio_init pins won't actually change the main loop's behavior (since all three pins are initialized anyway). We must patch the hardcoded pins inside the **main loop** itself!

Find and patch the movs calls in the loop:

1. Navigate to 10000296 where the red LED pin is loaded: `movs r2,#0x10`
2. Right-click the instruction -> **Patch Instruction** (or press Ctrl+Shift+G)
3. Change `#0x10` to `#0x11` (swap red to green's pin) and press **Enter**
4. Navigate to 1000029c where the green LED pin is loaded: `movs r2,#0x11`
5. Right-click the instruction -> **Patch Instruction**
6. Change `#0x11` to `#0x10` (swap green to red's pin) and press **Enter**

Before:

```
LED 1 (0x0C) -> GPIO 16 -> Red LED
LED 2 (0x18) -> GPIO 17 -> Green LED
```

After:

```
LED 1 (0x0C) -> GPIO 17 -> Green LED (SWAPPED!)
LED 2 (0x18) -> GPIO 16 -> Red LED (SWAPPED!)
```

Step 33: Export and Flash

1. Click **File** -> **Export Program**
2. Set **Format** to **Raw Bytes**
3. Name: `0x0023_structures-h.bin`
4. Click **OK**

Convert and flash:

```
cd C:\Users\flare-vm\Desktop\Embedded-Hacking-main\0x0023_structures
python ..\uf2conv.py build\0x0023_structures-h.bin --base 0x10000000 --family 0xe48bff59 --output build\hacked.uf2
```

Step 34: Verify the Hack**Open PuTTY and test:**

- Press “1” on remote -> **GREEN** LED lights (was red!)
- Terminal still shows NEC command: 0x0C
- Press “2” on remote -> **RED** LED lights (was green!)
- Terminal still shows NEC command: 0x18

The log says one thing, but the hardware does another!

Part 15: Security Implications - Log Desynchronization**The Danger of Mismatched Logs**

Log vs Reality Desynchronization			
Terminal Log		Physical LEDs	
NEC: 0x0C (expects RED)	+-----	GREEN LED on (not red!)	<-- Mismatch!
NEC: 0x18 (expects GREEN)	+-----	RED LED on (not green!)	<-- Mismatch!
The OPERATOR sees correct logs but WRONG physical behavior!			

Real-World Example: Stuxnet

Stuxnet was a cyberweapon that:

- Attacked Iranian nuclear centrifuges
- Made centrifuges spin at dangerous speeds
- Fed FALSE “everything normal” data to operators
- Operators saw stable readings while equipment was destroyed

Our LED example demonstrates the same principle:

- Logs show expected behavior
 - Hardware performs different actions
 - Attackers can hide malicious activity
-

Part 16: Functions Project - Advanced Code**Step 35: Review the Functions Code**

File: 0x0026_functions.c (key functions shown)

```
// Map IR command to LED number
int ir_to_led_number(int ir_command) {
    if (ir_command == 0x0C) return 1;
    if (ir_command == 0x18) return 2;
    if (ir_command == 0x5E) return 3;
    return 0;
}
```

```

// Get GPIO pin for LED number
uint8_t get_led_pin(simple_led_ctrl_t *leds, int led_num) {
    if (led_num == 1) return leds->led1_pin;
    if (led_num == 2) return leds->led2_pin;
    if (led_num == 3) return leds->led3_pin;
    return 0;
}

// Turn off all LEDs
void leds_all_off(simple_led_ctrl_t *leds) {
    gpio_put(leds->led1_pin, false);
    gpio_put(leds->led2_pin, false);
    gpio_put(leds->led3_pin, false);
}

// Blink an LED
void blink_led(uint8_t pin, uint8_t count, uint32_t delay_ms) {
    for (uint8_t i = 0; i < count; i++) {
        gpio_put(pin, true);
        sleep_ms(delay_ms);
        gpio_put(pin, false);
        sleep_ms(delay_ms);
    }
}

// Main command processor
int process_ir_led_command(int ir_command, simple_led_ctrl_t *leds, uint8_t blink_count) {
    if (!leds || ir_command < 0) return -1;

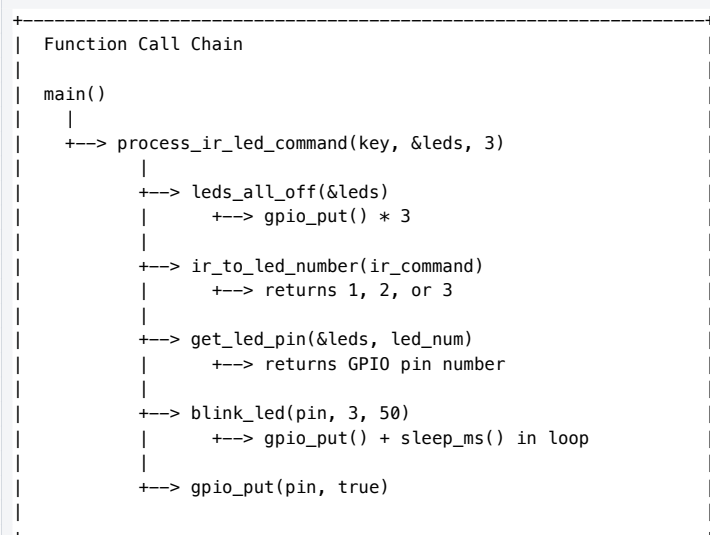
    leds_all_off(leds);
    int led_num = ir_to_led_number(ir_command);
    if (led_num == 0) return 0;

    uint8_t pin = get_led_pin(leds, led_num);
    blink_led(pin, blink_count, 50);
    gpio_put(pin, true);

    return led_num;
}

```

Step 36: Understand the Function Call Chain



Step 37: Flash and Test

1. Flash `0x0026_functions.uf2` to your Pico 2
2. Open PuTTY
3. Press remote buttons:
 - “1” -> Red LED blinks 3 times, then stays on
 - “2” -> Green LED blinks 3 times, then stays on
 - “3” -> Yellow LED blinks 3 times, then stays on

Part 17: Debugging with GDB (Functions)

Step 38: Start OpenOCD (Terminal 1)

Open a terminal and 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"
```

You should see output indicating OpenOCD connected successfully to your Pico 2 via the Debug Probe.

Step 39: Start GDB (Terminal 2)

Open a **new terminal** and launch GDB with the binary:

```
arm-none-eabi-gdb build\0x0026_functions.elf
```

Step 40: Connect to the Remote Target

In GDB, connect to OpenOCD:

```
target extended-remote :3333
```

Step 41: Halt the Running Binary

Stop the processor:

```
monitor halt
```

Step 42: Examine the Function Layout

Disassemble to see the multiple functions:

```
disassemble 0x10000234,+300
```

You'll see multiple function prologues (push) and epilogues (pop) for the helper functions.

Step 43: Discover the Missing Functions

The C code relies heavily on helper functions (`process_ir_led_command`, `leds_all_off`, `ir_to_led_number`, `blink_led`). But if you scroll through the disassembly, you'll notice a distinct lack of function prologues, epilogues, or `bl` calls to anything other than `sleep_ms` or `printf`!

The compiler has aggressively **inlined** every single helper function into the main loop.

Let's trace exactly how the compiler flattened this logic. First, set a breakpoint right when a key press is detected:

```
break *0x10000284
```

```
continue
```

(Press a button on your remote to trigger the breakpoint!)

Step 44: Examine leds_all_off (Inlined)

In the C code, process_ir_led_command starts by calling leds_all_off(&leds).

Let's look at the next few instructions starting from our breakpoint:

```
x/8i $pc
```

You'll see:

```
0x10000284 <main+80>:      mov     r1, r4
0x10000286 <main+82>:      ldr     r0, [pc, #156]
0x10000288 <main+84>:      bl      0x10003554 <__wrap_printf>
0x1000028c <main+88>:      movs   r5, #16
0x1000028e <main+90>:      mcrr   0, 4, r5, r6, cr0
0x10000292 <main+94>:      movs   r3, #17
0x10000294 <main+96>:      mcrr   0, 4, r3, r6, cr0
0x10000298 <main+100>:     movs   r2, #18
```

Because r6 was set to 0 earlier, the compiler is just directly writing 0 to pins 16, 17, and 18 using mcrr. It bypassed the function call entirely!

Step 45: Examine ir_to_led_number (Inlined)

Next, the C code calls ir_to_led_number and get_led_pin. Let's see how the compiler handled that by inspecting further down:

```
x/4i 0x1000029e
```

```
0x1000029e <main+106>:     cmp     r4, #12
0x100002a0 <main+108>:     beq.n   0x100002c6 <main+146>
0x100002a2 <main+110>:     cmp     r4, #24
0x100002a4 <main+112>:     beq.n   0x1000030a <main+214>
```

Instead of a separate function, it simply compares the button code in r4 (12, 24, 94) and branches straight to the correct blinking logic!

Step 46: Watch the blink_led Loop

Let's set a breakpoint where the blinking loop begins for Button 1 (which handles the Red LED on pin 16).

```
break *0x100002c6
continue
```

Once halted, inspect the setup:

```
x/6i $pc
```

```
0x100002c6 <main+146>:     mov.w   r8, #1           @ led_num = 1
0x100002ca <main+150>:     movs   r4, #3           @ blink_count = 3
0x100002cc <main+152>:     mcrr   0, 4, r5, r7, cr0 @ Turn LED ON (r7=1)
0x100002d0 <main+156>:     movs   r0, #50          @ Delay 50ms
0x100002d2 <main+158>:     bl      0x10001008 <sleep_ms>
0x100002d6 <main+162>:     mcrr   0, 4, r5, r6, cr0 @ Turn LED OFF (r6=0)
```

The compiler placed the blink count in r4 and handles the toggling with mcrr instructions surrounding sleep_ms.

Step 47: The Struct Pointer is a Lie

If you were trying to find the `leds` struct pointer to examine the pins using `x/6xb`, you'd be looking forever. Because the compiler realized the struct is never passed to any external non-inlined functions, it **never bothered creating it in memory**. It simply kept track of the pin numbers (16, 17, 18) directly in registers during compilation!

Step 48: Observe the Loop Condition

Continue execution until you hit the end of the blink loop:

```
break *0x100002e0
continue
```

At this point, GDB is halted **right before** it performs the subtraction. If you check `r3` now, it will contain random leftover garbage (like `0x400b0000`) because the instruction hasn't run yet! Check `r4` (the current blink count):

```
info registers r4
```

Now, step forward two instructions to let it actually do the math:

```
stepi 2
info registers r3 r4
```

```
0x100002e0 <main+172>:      subs    r3, r4, #1
0x100002e2 <main+174>:      ands.w  r4, r3, #255
0x100002e6 <main+178>:      bne.n   0x100002cc <main+152>
```

You will now see `r3` become 2, and `r4` become 2! It successfully subtracted 1 from the blink count and stored it back. Since it hasn't hit 0 yet, the `bne.n` instruction will branch back to the start of the blink (`0x100002cc`) to flash the LED again!

Step 49: Exit GDB

When done exploring:

```
quit
```

Part 18: Analyzing .ELF Files in Ghidra

Step 50: Create New Ghidra Project

1. Create project: `0x0026_functions`
2. Import the `.elf` file (NOT the `.bin` this time!)

Why Use .ELF Instead of .BIN?

Feature	.BIN File	.ELF File
Symbols	None	Function/variable names
Sections	Raw bytes only	.text, .data, .rodata, etc.
Debug info	None	May include debug symbols
Size	Smaller	Larger
Use case	Flashing to hardware	Analysis and debugging

Step 51: Import and Analyze the .ELF

1. Drag and drop the `.elf` file into Ghidra
2. Ghidra automatically detects ARM format!
3. Click **Yes** to analyze
4. Wait for analysis to complete

Step 52: Explore the Symbol Tree

With `.ELF` files, you get more information:

1. Look at the **Symbol Tree** panel
2. Expand **Functions** - you may see named functions!
3. Expand **Labels** - data labels may appear

Part 19: Hacking the Functions Project**Step 53: Find LED Pin Values**

Look for the struct initialization pattern:

```
movs r0, #0x10    ; led1_pin = 16
movs r0, #0x11    ; led2_pin = 17
movs r0, #0x12    ; led3_pin = 18
```

Step 54: Swap LED 1 and LED 3

We'll swap the red (GPIO 16) and yellow (GPIO 18) LEDs:

Find and patch in the .bin file:

1. Change `0x10` (16) to `0x12` (18)
2. Change `0x12` (18) to `0x10` (16)

Before:

```
Button 1 -> LED 1 -> GPIO 16 -> Red
Button 3 -> LED 3 -> GPIO 18 -> Yellow
```

After:

```
Button 1 -> LED 1 -> GPIO 18 -> Yellow (SWAPPED!)
Button 3 -> LED 3 -> GPIO 16 -> Red (SWAPPED!)
```

Step 55: Export the Patched .BIN

Important: Even though we analyzed the `.elf`, we patch the `.bin`!

1. Open the original `.bin` file in Ghidra (or a hex editor)
2. Apply the patches
3. Export as `0x0026_functions-h.bin`

Step 56: Convert and Flash

```
cd C:\Users\flare-vm\Desktop\Embedded-Hacking-main\0x0026_functions
python ..\uf2conv.py build\0x0026_functions-h.bin --base 0x10000000 --family 0xe48bff59 --output build\hacked.uf2
```

Step 57: Verify the Hack

Open PuTTY and test:

- Press "1" -> **YELLOW** LED blinks (was red!)

- Terminal shows: LED 1 activated on GPIO 16 (WRONG - it's actually GPIO 18!)
- Press "3" -> **RED** LED blinks (was yellow!)
- Terminal shows: LED 3 activated on GPIO 18 (WRONG - it's actually GPIO 16!)

Again, logs don't match reality!

Part 20: Summary and Review

What We Accomplished

1. **Learned C structures** - Grouping related data together
2. **Understood struct memory layout** - How members are stored consecutively
3. **Mastered dot and arrow operators** - Accessing struct members
4. **Learned the NEC IR protocol** - How remotes communicate
5. **Understood functions with parameters** - Passing data in and out
6. **Saw struct flattening in assembly** - How compilers transform structs
7. **Analyzed .ELF files** - Getting more symbol information
8. **Hacked GPIO assignments** - Swapping LED behavior
9. **Discovered log desynchronization** - Security implications

Struct Operations Summary

```

+-----+
| Struct Operations |
+-----+
| Definition:       |
| typedef struct {  |
|     uint8_t pin;  |
|     bool state;   |
| } led_t;          |
+-----+
| Creation:         |
| led_t led = { .pin = 16, .state = false }; |
+-----+
| Access (variable): led.pin |
| Access (pointer):  ptr->pin or (*ptr).pin |
+-----+
| Passing to function: void func(led_t *led) |
| Calling:             func(&led) |
+-----+

```

Function Types Summary

```

+-----+
| Function Patterns |
+-----+
| No params, no return: |
| void leds_all_off(void) |
+-----+
| With params, no return: |
| void blink_led(uint8_t pin, uint8_t count, uint32_t delay) |
+-----+
| No params, with return: |
| int ir_getkey(void) |
+-----+
| With params, with return: |
| int ir_to_led_number(int ir_command) |
+-----+
| With struct pointer: |
| uint8_t get_led_pin(simple_led_ctrl_t *leds, int led_num) |
+-----+

```

Key Memory Addresses

Memory Address	Description
0x10000234	main() function
0x10 (16)	GPIO 16 - Red LED (led1_pin)
0x11 (17)	GPIO 17 - Green LED (led2_pin)
0x12 (18)	GPIO 18 - Yellow LED (led3_pin)
0x05	GPIO 5 - IR receiver
0x0C	NEC code for button 1
0x18	NEC code for button 2
0x5E	NEC code for button 3

Key Takeaways

1. **Structs group related data** - Better organization than separate variables
2. **Dot operator for variables, arrow for pointers** - `.` vs `->`
3. **Designated initializers are cleaner** - `.member = value` syntax
4. **Compilers flatten structs** - You see values, not struct names, in assembly
5. **NEC protocol uses 8-bit commands** - 0x0C, 0x18, 0x5E for our buttons
6. **Functions separate concerns** - Each function does one job
7. **.ELF files contain more info than .BIN** - Symbols, sections, debug data
8. **Log desynchronization is dangerous** - Logs can lie about real behavior
9. **Pattern recognition is key** - Consecutive values like 16, 17, 18 reveal structs
10. **Always patch the .bin for flashing** - .elf is for analysis only

Glossary

Term	Definition
Arrow Operator (->)	Accesses struct member through a pointer
Designated Initializer	Syntax <code>.member = value</code> for struct initialization
Dot Operator (.)	Accesses struct member from a struct variable
.ELF File	Executable and Linkable Format - contains symbols
Flattening	Compiler converting structs to individual values
IR (Infrared)	Invisible light used for remote control
Log Desynchronization	When logs don't match actual system behavior
Member	A variable inside a struct

Term	Definition
NEC Protocol	Common IR communication standard
Struct	User-defined type grouping related variables
typedef	Creates an alias for a type

Additional Resources

NEC IR Command Reference

Button	Command	Binary
1	0x0C	0000 1100
2	0x18	0001 1000
3	0x5E	0101 1110

GPIO Pin Quick Reference

GPIO	Default Function	Our Usage
5	General I/O	IR Receiver
16	General I/O	Red LED
17	General I/O	Green LED
18	General I/O	Yellow LED

Struct Size Calculation

Type	Size (bytes)
uint8_t	1
bool	1
uint16_t	2
uint32_t	4
int	4
float	4
pointer	4 (on ARM32)

Real-World Implications

What You've Learned in This Course

Over these weeks, you've built skills that few people possess:

1. **Hardware fundamentals** - GPIO, I2C, PWM, IR protocols

2. **Reverse engineering** - Ghidra, disassembly, function identification
3. **Binary patching** - Modifying compiled code
4. **Security awareness** - Understanding vulnerabilities

The Power and Responsibility

The techniques you've learned can be used for:

Good:

- Security research
- Debugging proprietary systems
- Understanding how things work
- Career in cybersecurity

Danger:

- Unauthorized system access
- Sabotage of critical infrastructure
- Fraud and deception

Always use your skills ethically and legally!

Keep Learning

This is just the beginning:

- Explore more complex protocols (SPI, CAN bus)
- Learn dynamic analysis with debuggers
- Study cryptographic implementations
- Practice on CTF challenges

Congratulations on completing this course! You now have the curiosity, persistence, and skills that embedded systems engineers and security researchers thrive on. Keep experimenting, documenting, and sharing your work. The world needs more builders and defenders like you!

Happy hacking! :)