
EMBEDDED HACKING

Reverse Engineering the Raspberry Pi Pico 2

Week 10: Conditionals in Embedded Systems: Debugging and Hacking Static & Dynamic Conditionals w/ SG90 Servo Motor PWM 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 static and dynamic conditionals in C
 - Know how if/else statements and switch/case blocks work at the assembly level
 - Understand Pulse Width Modulation (PWM) and how it controls servo motors
 - Calculate PWM timing from system clock to servo pulse width
 - Identify conditional branches in Ghidra (beq, bne instructions)
 - Hack string literals and timing delays in binary files
 - Modify branch targets to change program flow
 - Create “stealth” functionality by NOP-ing out print statements
 - Understand IEEE-754 floating-point for angle calculations
-

Part 1: Understanding Conditionals in C

What Are Conditionals?

Conditionals are programming structures that make decisions. They let your program choose different paths based on whether a condition is true or false. Think of them like a fork in the road - the program checks a condition and decides which way to go.

Two Types of Conditionals

Type	Description	Example
Static	Condition value is known/fixed at compile time	<code>if (choice == 1)</code> where choice never changes
Dynamic	Condition value changes based on runtime input	<code>if (choice == getchar())</code> where user types input

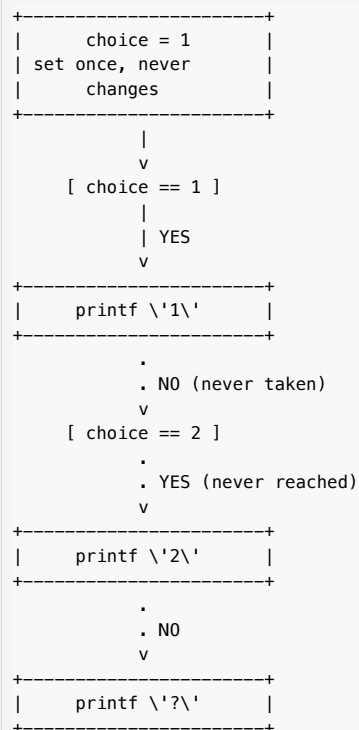
Part 2: Static Conditionals

What Makes a Conditional “Static”?

A **static conditional** is one where the outcome is predetermined because the condition variable never changes during program execution:

```
int choice = 1; // This NEVER changes!

while (true) {
    if (choice == 1) {
        printf("1\r\n"); // This ALWAYS runs
    } else if (choice == 2) {
        printf("2\r\n"); // This NEVER runs
    } else {
        printf("?r\n"); // This NEVER runs
    }
}
```



The if/else Statement

The if/else structure checks conditions in order:

```
if (choice == 1) {
    // Do something if choice is 1
} else if (choice == 2) {
    // Do something if choice is 2
} else {
    // Do something for all other values
}
```

The switch Statement

The switch statement is another way to handle multiple conditions:

```

switch (choice) {
    case 1:
        printf("one\r\n");
        break;
    case 2:
        printf("two\r\n");
        break;
    default:
        printf("??\r\n");
}

```

Key Differences:

Feature	if/else	switch
Condition	Any boolean expression	Single variable comparison
Values	Ranges, complex logic	Discrete values only
Fall-through	No	Yes (without break)
Readability	Good for 2-3 conditions	Better for many conditions

Part 3: Dynamic Conditionals

What Makes a Conditional “Dynamic”?

A **dynamic conditional** is one where the condition variable changes based on runtime input:

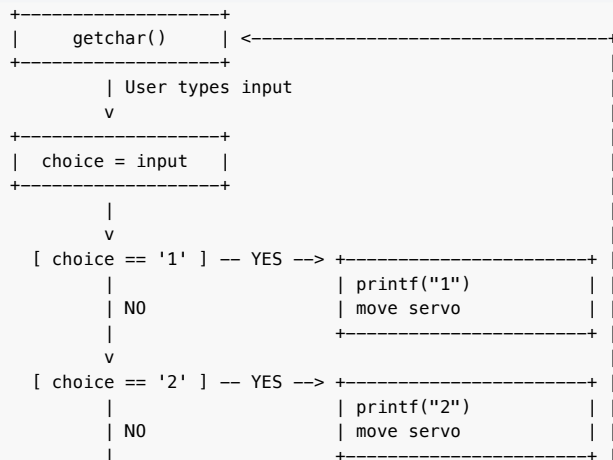
```

uint8_t choice = 0;

while (true) {
    choice = getchar(); // User types a key - VALUE CHANGES!

    if (choice == '1') {
        printf("1\r\n");
    } else if (choice == '2') {
        printf("2\r\n");
    } else {
        printf("??\r\n");
    }
}

```





The getchar() Function

getchar() reads a single character from the serial terminal:

```
uint8_t choice = getchar(); // Waits for user to type something
```

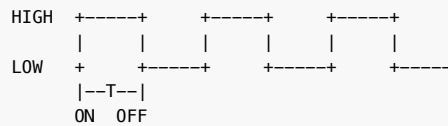
- Returns the ASCII value of the key pressed
- '1' = 0x31, '2' = 0x32, 'x' = 0x78, 'y' = 0x79
- Blocks (waits) until a key is pressed

Part 4: Understanding PWM (Pulse Width Modulation)

What is PWM?

PWM (Pulse Width Modulation) is a technique for controlling power by rapidly switching a signal on and off. The ratio of “on time” to “off time” determines the average power delivered.

PWM Signal - 50% Duty Cycle



Duty Cycle = ON time / Total period = 50%

PWM for Servo Control

Servo motors use PWM differently - they care about the **pulse width**, not the duty cycle percentage:

Servo PWM Signal (50 Hz = 20ms period)

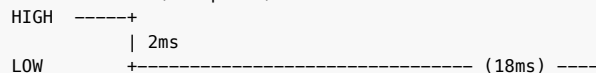
0° Position (1ms pulse):



90° Position (1.5ms pulse):



180° Position (2ms pulse):



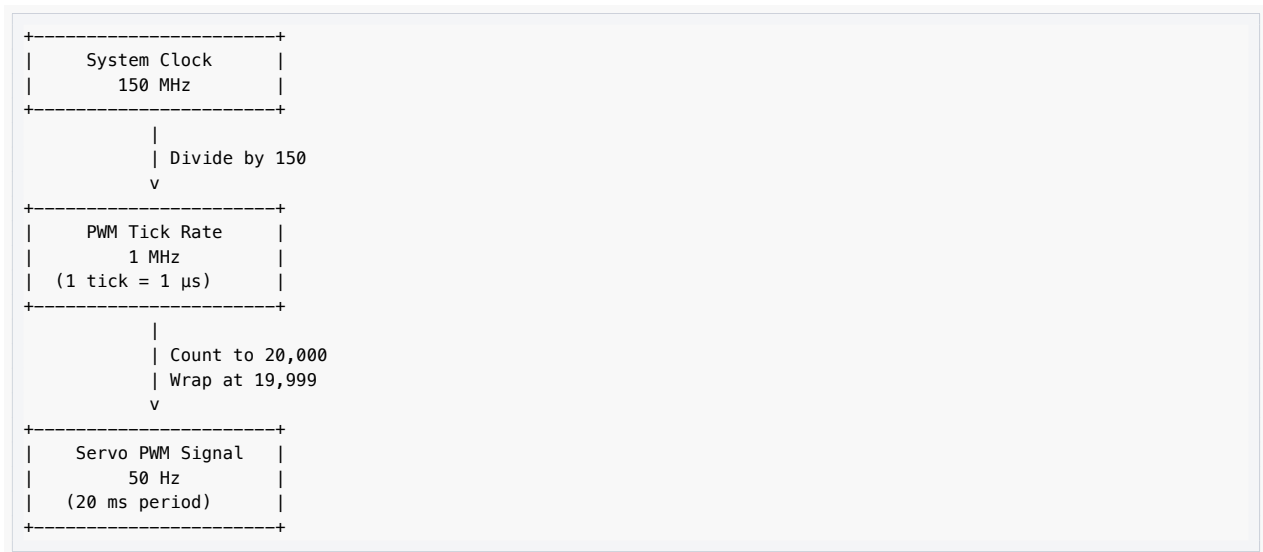
The Magic Numbers

Angle	Pulse Width	PWM Ticks (at 1MHz)
0°	1000 μ s	1000
90°	1500 μ s	1500
180°	2000 μ s	2000

Part 5: PWM Timing Calculations

From 150 MHz to 50 Hz

The RP2350's system clock runs at **150 MHz** (150 million cycles per second). A servo needs a **50 Hz** signal (one pulse every 20 ms). How do we bridge this gap?



The Math

Step 1: Clock Division

PWM Tick Rate = System Clock / Divider
 $1,000,000 \text{ Hz} = 150,000,000 \text{ Hz} / 150$

Step 2: Frame Period

Period = (Wrap Value + 1) * Tick Duration
 $20 \text{ ms} = 20,000 \text{ ticks} * 1 \mu\text{s/tick}$

Step 3: Pulse Width to Ticks

Ticks = Pulse Width (μ s) * 1 tick/ μ s
 $1500 \text{ ticks} = 1500 \mu\text{s} * 1$

Worked Example: 90° Angle

Let's calculate what happens when we command 90°:

1. Angle to Pulse Width:

Pulse = MIN + (angle/180) * (MAX - MIN)
Pulse = $1000 + (90/180) * (2000 - 1000)$
Pulse = $1000 + 0.5 * 1000$

```
Pulse = 1500 μs
```

2. Pulse to PWM Ticks:

```
Level = 1500 μs * 1 tick/μs = 1500 ticks
```

3. Hardware Timing:

- Signal HIGH for 1500 ticks (1.5 ms)
- Signal LOW for 18,500 ticks (18.5 ms)
- Total period: 20,000 ticks (20 ms)

Part 6: Understanding the SG90 Servo Motor

What is the SG90°

The **SG90** is a small, inexpensive hobby servo motor commonly used in robotics projects:

```
[ SG90 Servo Motor ]
+-----+
| Motor ---> Gearbox ---> Arm |
|                               |
|                               |
|                               |
+-----+
|                               |
|                               |
|                               |
|                               |
+-----+
| Wires |
|       |
|       |
|       |
|       |
+-----+
[ Wires ]
+-----+
| Orange: Signal / PWM |
| Red: VCC / 5V        |
| Brown: GND / Ground  |
+-----+
```

SG90 Specifications

Parameter	Value
Voltage	4.8V - 6V (typically 5V)
Rotation	0° to 180°
Pulse Width	1000 us - 2000 us
Frequency	50 Hz (20ms period)
Stall Current	~650mA (can spike to 1A+)

Wire Colors

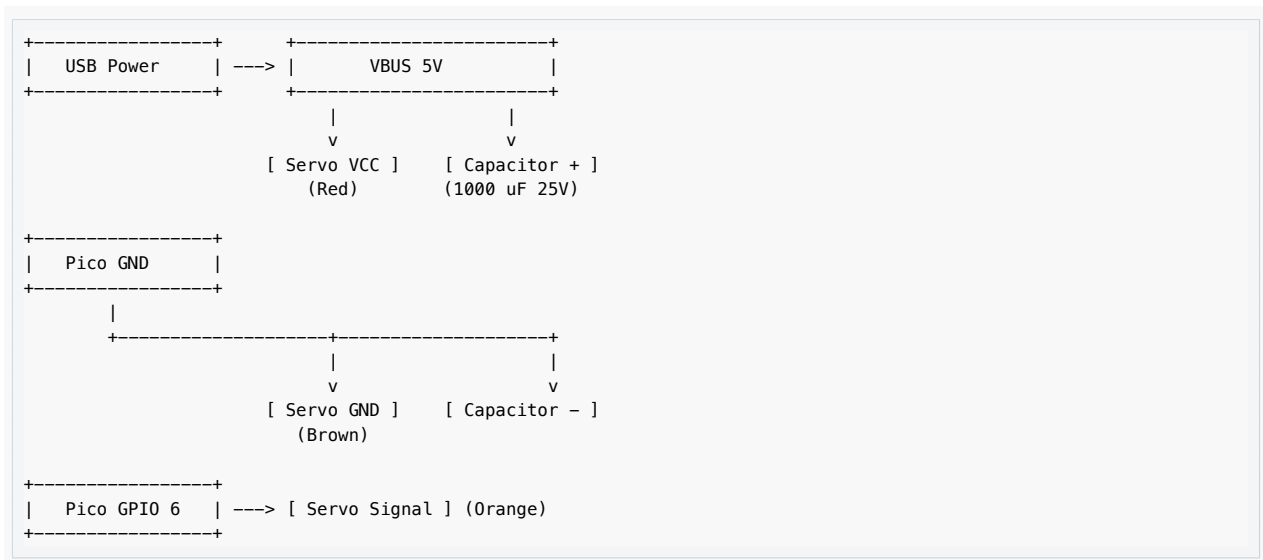
Wire Color	Function	Connect To
Brown	GND	Ground
Red	VCC	5V Power (VBUS)
Orange	Signal	GPIO Pin (PWM)

Part 7: Power Supply Safety

CRITICAL WARNING**NEVER power the servo directly from the Pico's 3.3V pin!**

Servos can draw over 1000mA during movement spikes. The Pico's 3.3V regulator cannot handle this and you will:

- Cause brownouts (Pico resets)
- Damage the Pico's voltage regulator
- Potentially damage your USB port

Correct Power Setup**Why the Capacitor?**

The **1000 uF capacitor** acts as a tiny battery:

- Absorbs sudden current demands when servo moves
- Prevents voltage drops that could reset the Pico
- Smooths out electrical noise

Part 8: Setting Up Your Environment**Prerequisites**

Before we start, make sure you have:

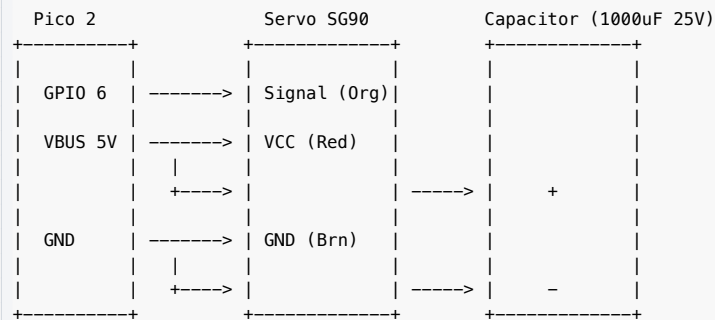
1. A Raspberry Pi Pico 2 board
2. A Raspberry Pi Pico Debug Probe
3. Ghidra installed (for static analysis)
4. Python installed (for UF2 conversion)
5. A serial monitor (PuTTY, minicom, or screen)
6. An SG90 servo motor
7. A 1000 uF 25V capacitor
8. The sample projects: `0x001d_static-conditionals` and `0x0020_dynamic-conditionals`

Hardware Setup

Connect your servo like this:

Servo Wire	Pico 2 Pin
Brown (GND)	GND
Red (VCC)	VBUS (5V)
Orange (Signal)	GPIO 6

Hardware Setup:



Project Structure

```

Embedded-Hacking/
+-- 0x001d_static-conditionals/
|   +-- build/
|   |   +-- 0x001d_static-conditionals.uf2
|   |   +-- 0x001d_static-conditionals.bin
|   +-- main/
|   |   +-- 0x001d_static-conditionals.c
|   +-- servo.h
+-- 0x0020_dynamic-conditionals/
|   +-- build/
|   |   +-- 0x0020_dynamic-conditionals.uf2
|   |   +-- 0x0020_dynamic-conditionals.bin
|   +-- main/
|   |   +-- 0x0020_dynamic-conditionals.c
|   +-- servo.h
+-- uf2conv.py
  
```

Part 9: Hands-On Tutorial - Static Conditionals Code

Step 1: Review the Source Code

Let's examine the static conditionals code:

File: `0x001d_static-conditionals.c`

```

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

#define SERVO_GPIO 6

int main(void) {
    stdio_init_all();

    int choice = 1; // STATIC - never changes!

    servo_init(SERVO_GPIO);
  
```

```

while (true) {
    // if/else conditional
    if (choice == 1) {
        printf("1\r\n");
    } else if (choice == 2) {
        printf("2\r\n");
    } else {
        printf("? \r\n");
    }

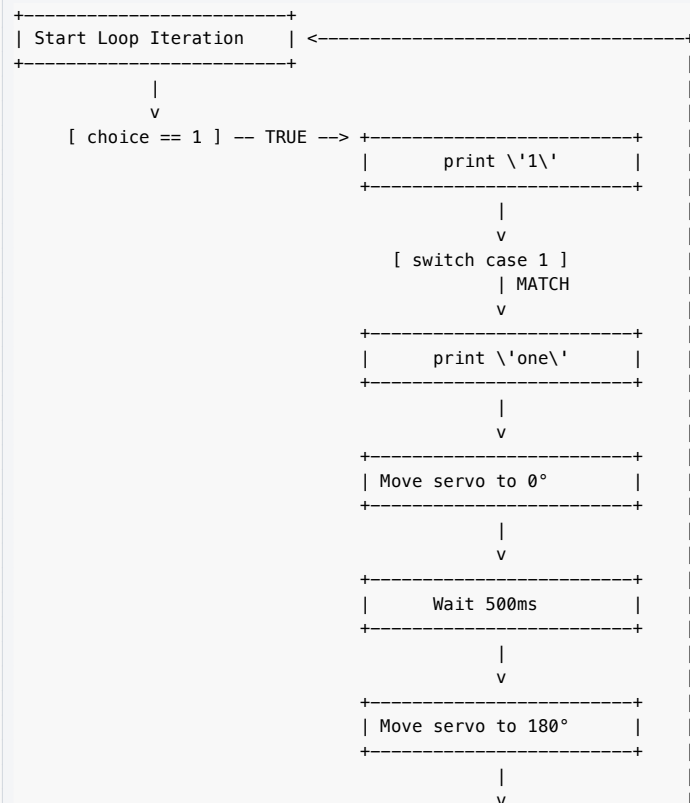
    // switch/case conditional
    switch (choice) {
        case 1:
            printf("one\r\n");
            break;
        case 2:
            printf("two\r\n");
            break;
        default:
            printf("??\r\n");
    }

    // Servo movement
    servo_set_angle(0.0f);
    sleep_ms(500);
    servo_set_angle(180.0f);
    sleep_ms(500);
}

```

Step 2: Understand the Program Flow

Since choice = 1 and NEVER changes:



```

+-----+ |
|      Wait 500ms      | ----+
+-----+

```

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 0x001d_static-conditionals.uf2 onto the drive
5. The Pico will reboot and start running!

Step 4: Verify It's Working

Check the serial monitor (PuTTY at 115200 baud):

```

1
one
1
one
1
one
...

```

Watch the servo:

- It should sweep from 0° to 180° every second
- The movement is continuous and repetitive

Part 10: Debugging with GDB (Static Conditionals)

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\0x001d_static-conditionals.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 the conditionals:

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

Look for comparison and branch instructions that implement the if/else and switch logic.

Step 10: Set a Breakpoint at Main

```
break *0x10000234
```

Reset and run to hit the breakpoint:

```
monitor reset halt
continue
```

Step 11: Find the Comparison Instructions... Or Not!

To see the assembly instructions as you execute them, tell GDB to display the current instruction automatically:

```
display/i $pc
nexti
```

Keep pressing **Enter** to repeat the nexti command and watch the instructions.

Wait, where is the cmp instruction?! You might notice that there is NO cmp instruction comparing our choice variable anywhere! Why? Because we hardcoded `int choice = 1;` at the start of our C code and never changed it.

The C compiler is smart (even at basic optimization levels). It realized the `if (choice == 1)` condition would *always* be true, and the *else* conditions would *never* happen. Instead of wasting CPU cycles checking a condition that never changes, the compiler **optimized out the check entirely!** It just compiled the code inside the `choice == 1` block unconditionally.

This is the defining characteristic of a “Static Conditional”—the condition is resolved at compile-time, not run-time!

Step 12: Examine the Printf/Puts Arguments

If you look closely at your disassembly, you’ll notice there are no printf calls! The compiler optimized our simple printf statements into puts (specifically `__wrap_puts` in the Pico SDK) because they didn’t contain any complex formatting variables.

When you reach a `bl <__wrap_puts>` instruction, check `r0` for the string address:

```
x/s $r0
```

You should see strings like `"1\r"` or `"one\r"`.

(Wait, what happened to the `\n`? Because the puts function automatically adds a newline to the end of whatever it prints, the compiler cleverly trimmed the `\n` out of the string literal in memory to save space!)

Step 13: Watch the Servo Commands

Let’s set a breakpoint on `servo_set_angle`. How do you know the address? If you look at your main disassembly, you can find the `bl` instruction calling it (for example, `0x10000310`).

But hardcoded addresses change every time you recompile! Luckily, GDB is smart enough to resolve function names directly from the ELF file:

```
break servo_set_angle
continue
```

Check the argument passed to the function. You might assume the float is in `s0`, but check out the very first instruction of `servo_set_angle`:

```
vmov s14, r0
```

Because of how the ARM compiler handles arguments, the floating-point angle is actually passed into the function via the standard integer register `r0`. The `vmov` instruction immediately moves it from `r0` into the floating-point register `s14` so the math unit can use it!

To see the angle, you can ask GDB to interpret the raw hex value in `r0` as a float. You can also step forward (`nexti` or `n`) to let the `vmov` execute, and then check `s14` directly!

Here is exactly what that process looks like in your GDB terminal (assuming you hit `continue` to catch the second call where the angle is 180):

```
=> 0x10000310 <servo_set_angle>:      vmov    s14, r0
(gdb) print /f $r0
$2 = 180
(gdb) nexti
=> 0x10000314 <servo_set_angle+4>:    push    {r4, r5, lr}
(gdb) info registers s14
s14          180                (raw 0x43340000)
```

(Note: On your very first breakpoint hit, `print /f $r0` will just show 0 because the first call in the C code is `servo_set_angle(0)`!)

Step 14: Examine the Timing Delay

Set a breakpoint on `sleep_ms` (using the function name, not a hardcoded address!) and check the delay value:

```
break sleep_ms
continue
info registers r0
```

You should see `0x1f4` (500 decimal) for the 500ms delay.

Step 15: Watch the Loop Iterate

Because this loop contains `sleep_ms(500)` calls, trying to use `nexti 100` to step forward will cause GDB to “hang” (either because it’s waiting for multiple seconds of sleep to finish, or because GDB stepping interferes with the Pico’s hardware timer interrupts!).

Instead, since we already have breakpoints set on `servo_set_angle` and `sleep_ms`, just type `continue`!

```
continue
```

Every time you type `continue` (or press **Enter** to repeat it), you will see the CPU safely jump to the next function call. Because this is a *Static Conditional*, it will never hit a `cmp` instruction or take a different branch—it just bounces between `servo_set_angle` and `sleep_ms` forever!

Step 16: Exit GDB

When done exploring:

```
quit
```

Part 11: Setting Up Ghidra for Static Conditionals

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: 0x001d_static-conditionals
5. Click **Finish**

Step 19: Import the Binary

1. Open your file explorer
2. Navigate to the 0x001d_static-conditionals/build/ folder
3. **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 in Ghidra (Static)

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: Resolve servo_init

Look for a function call where `r0` is loaded with `0x6` (GPIO pin 6):

```
movs r0, #0x6      ; GPIO pin 6
bl  FUN_1000027c ; servo_init
```

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

Step 25: Resolve puts

Look for function calls that load string addresses into `r0`:

```
ldr      r0=>DAT_10001c54 , [DAT_10000274 ] = 00000D31h
bl      FUN_10001884      undefined FUN_10001884()
```

How do we know it's puts?

- It takes a single string argument
- The hex `0x31` is ASCII "1"
- The hex `0x0d` is carriage return "\r"
- We saw "1" echoed in PuTTY

1. Right-click -> **Edit Function Signature**
2. Change to: `int puts(char *s)`
3. Click **OK**

Step 26: Resolve servo_set_angle

Look for a function call (like `bl FUN_10000310`) that occurs right after the float arguments are loaded into `r0`.

```
mov      r0, r5
bl      FUN_10000310      undefined FUN_10000310()
```

If you double-click the `FUN_10000310` label to look inside the function, you'll find a section of code checking the pulse limits:

```
cmp.w    r3, #0x7d0
it       cs
mov.cs.w r3, #0x7d0
cmp.w    r3, #0x3e8
it       cc
mov.cc.w r3, #0x3e8
```

These values are:

- `0x7D0` (2000 decimal) - maximum pulse width
- `0x3E8` (1000 decimal) - minimum pulse width

These are the servo pulse limits!

1. Right-click -> **Edit Function Signature**
2. Change to: `void servo_set_angle(float degrees)`
3. Click **OK**

Step 27: Resolve sleep_ms

Look for a function where `r0` is loaded with `0x1f4` (500 decimal):

```
mov.w    r0, #0x1f4
```

```
bl      FUN_10000e20      undefined FUN_10000e20()
```

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

Part 13: Hacking Static Conditionals

Step 28: Open the Bytes Editor

1. Click **Window** -> **Bytes**
2. A new panel appears showing raw hex bytes
3. Click the pencil icon to enable editing

Step 29: Hack #1 - Change “1” to “2”

First, we need to find the string “1” in the binary.

1. Go back to your main assembly code and look for the first `puts` call.
2. In the `ldr` instruction right before it, look for the reference next to the arrow: `r0=>DAT_10001c54`. **Double-click exactly on DAT_10001c54.** (Warning: Do NOT click the reference inside the brackets like `[DAT_10000274]`, or you’ll end up in the pointer table instead of the string!)
3. The Listing and Bytes windows will automatically jump to the string’s exact address!
4. Look at your **Bytes** window. You should see the byte 31 (which is ASCII for “1”).
5. Click on the 31 and type 32 to overwrite it (which is ASCII for “2”).

Step 30: Hack #2 - Change “one” to “fun”

Next, let’s change the word “one” to “fun”:

1. Go back to main and look for the second `puts` call.
2. In the `ldr` instruction, double-click the `DAT_10001c5c` reference next to the arrow (again, ignore the one in the brackets!).
3. Look at the **Bytes** window again. You’ll find the bytes `6f 6e 65` (which are ASCII for “o-n-e”).
4. Click on the `6f` byte and type `66 75 6e` on your keyboard to overwrite those three bytes with “f-u-n”.

ASCII Reference:

Character	Hex
o	0x6f
n	0x6e
e	0x65
f	0x66
u	0x75
n	0x6e

Step 31: Hack #3 - Speed Up the Servo

Let’s change the 500ms delay to a 100ms delay. Since the compiler packed the 500 directly into a `mov.w` instruction, we can’t just find it in the data section. Instead, let’s use Ghidra’s built-in assembler to rewrite the instruction!

1. In your main assembly code, look for the two `mov.w r0,#0x1f4` instructions (which happen right before calling `sleep_ms`).

2. Right-click the first `mov.w r0, #0x1f4` instruction and select **Patch Instruction** (or press Ctrl+Shift+G).
3. Delete the `#0x1f4` and type `#0x64` (which is 100 in hex). Press **Enter**!
4. Repeat this for the second `mov.w r0, #0x1f4` instruction.

Before: 500ms delay (servo moves slowly) **After:** 100ms delay (servo moves FAST!)

Step 32: Export and Flash

When exporting from Ghidra, you must make sure to save the file inside your `build/` directory so the python script can find it!

1. Click **File -> Export Program**
2. Set **Format** to **Raw Bytes**
3. **IMPORTANT:** Click the `...` next to the Output File field and navigate into your `0x001d_static-conditionals/build/` folder!
4. Save the file as `0x001d_static-conditionals-h.bin`.
5. Click **OK**

Convert and flash (make sure your terminal is inside the `0x001d_static-conditionals/` folder, NOT the `build/` folder!):

```
python ../uf2conv.py build\0x001d_static-conditionals-h.bin --base 0x10000000 --family 0xe48bff59 --output
↳ build\hacked.uf2
```

(Troubleshooting: If you get `FileNotFoundError` for the `.bin` file, it means you didn't save the exported file into the `build/` folder! Go back to Ghidra and export it again. If you get an error that Python can't open `../uf2conv.py`, it means you accidentally `cd'd` into the `build/` folder. Type `cd ..` to go back up one directory and run the command again!)

Step 33: Verify the Hacks

Serial output now shows:

```
2
fun
2
fun
...
```

The servo now moves 5x faster! It's spinning back and forth like crazy!

Part 14: Dynamic Conditionals - The Source Code

Step 34: Review the Dynamic Code

File: `0x0020_dynamic-conditionals.c`

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

#define SERVO_GPIO 6

int main(void) {
    stdio_init_all();

    uint8_t choice = 0; // DYNAMIC - changes with user input!

    servo_init(SERVO_GPIO);

    while (true) {
        choice = getchar(); // Wait for keyboard input
```

```

if (choice == 0x31) {          // '1'
    printf("1\r\n");
} else if (choice == 0x32) { // '2'
    printf("2\r\n");
} else {
    printf("??\r\n");
}

switch (choice) {
    case '1':
        printf("one\r\n");
        servo_set_angle(0.0f);
        sleep_ms(500);
        servo_set_angle(180.0f);
        sleep_ms(500);
        break;
    case '2':
        printf("two\r\n");
        servo_set_angle(180.0f);
        sleep_ms(500);
        servo_set_angle(0.0f);
        sleep_ms(500);
        break;
    default:
        printf("??\r\n");
}
}
}

```

Step 35: Understand the Dynamic Behavior

User Types	Output	Servo Action
'1' (0x31)	"1" + "one"	0° -> 180°
'2' (0x32)	"2" + "two"	180° -> 0°
Anything else	"??" + "??"	No movement

Step 36: Flash and Test

1. Flash 0x0020_dynamic-conditionals.uf2
2. Open PuTTY
3. Press '1' - servo sweeps one direction
4. Press '2' - servo sweeps the other direction
5. Press 'x' - prints "??" and no movement

Part 15: Debugging with GDB (Dynamic Conditionals)

Step 37: 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 38: Start GDB (Terminal 2)

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

```
arm-none-eabi-gdb build\0x0020_dynamic-conditionals.elf
```

Step 39: Connect to the Remote Target

In GDB, connect to OpenOCD:

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

Step 40: Halt the Running Binary

Stop the processor:

```
monitor halt
```

Step 41: Examine Main Function

Disassemble around main to see the dynamic conditionals:

```
disassemble 0x10000234,+250
```

Look for the `bl <__wrap_getchar>` call. Because choice is now dynamic (based on user input), the compiler *had* to generate the comparison logic! It should look something like this:

```
bl      0x10001860 <__wrap_getchar>
uxtb    r4, r0
cmp     r4, #49 @ 0x31
beq.n   0x10000270 <main+60>
cmp     r4, #50 @ 0x32
```

There they are! The `cmp` instructions checking for `0x31` ('1') and `0x32` ('2').

Step 42: Set a Breakpoint After getchar

We want to pause execution right after we type a character so we can inspect the comparison. Find the address of the first `cmp` instruction in your disassembly (in the example above, it's `0x1000024a`) and set a breakpoint:

```
break *0x1000024a
```

(Make sure you use the exact address of the cmp instruction from YOUR disassembly!)

Reset and continue:

```
monitor reset halt
continue
```

Step 43: Watch the Input Value

When you press a key in PuTTY, the breakpoint hits. Check the return value:

```
info registers r0
```

If you pressed '1', you should see `0x31`. If you pressed '2', you should see `0x32`.

Step 44: Execute the Comparison

Right now, GDB is paused *before* the `cmp` instruction runs (the `=>` arrow points to the instruction that will execute *next*).

To see the result of the comparison, we must execute the `cmp` instruction by stepping forward exactly once:

```
display/i $pc  
stepi
```

Notice that the => arrow has now moved to the `beq.n` (Branch if Equal) instruction. The `cmp` instruction has just executed.

Step 45: Examine the Branch Decisions

Now that the comparison has run, we can look at the CPU's condition flags to see what happened:

```
info registers xpsr
```

(Note: Older ARM chips call this *cpsr*, but Cortex-M chips like the Pico's RP2350 call it *xpsr*!)

The zero flag (Z) determines if the branch is taken. The flags are stored in the highest nibble (the first hex digit) of the `xpsr` register in the order `N Z C V`.

If you typed '1', the comparison resulted in zero difference, setting the Z flag to 1. You should see `xpsr` start with a 6 (like `0x69000000`). 6 in binary is `0110`, which means the Z flag (the second bit) is 1! The `beq.n` instruction will see this flag and take the branch.

Step 46: Watch Different Input Paths

Continue and press different keys to see how the program takes different branches:

```
continue
```

Press '2' in PuTTY, then examine registers again.

Step 47: Examine Servo Control

Set a breakpoint on `servo_set_angle`:

```
break *0x10000280  
continue
```

Check the angle value:

```
info registers s0
```

Step 48: Exit GDB

When done exploring:

```
quit
```

Part 16: Setting Up Ghidra for Dynamic Conditionals

Step 49: Create New Project

1. Create project: `0x0020_dynamic-conditionals`
2. Import the `.bin` file
3. Configure as ARM Cortex, base address `10000000`
4. Analyze

Step 50: Navigate to Main

Press G and go to `10000234`.

Step 51: Resolve Functions

Follow the same process:

1. **main** at 0x10000234 -> int main(void)
2. **stdio_init_all** -> bool stdio_init_all(void)
3. **servo_init** -> void servo_init(uint pin)
4. **puts** -> int puts(char *s)
5. **servo_set_angle** -> void servo_set_angle(float degrees)
6. **sleep_ms** -> void sleep_ms(uint ms)

Step 52: Identify getchar

Look for a function that:

- Returns a value in r0
- That value is then compared against 0x31 ("1")

```
bl      FUN_10001860      undefined FUN_10001860()
uxtb    r4, r0
cmp     r4, #0x31
beq     LAB_10000270
```

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

Step 53: Identify Hardware Addresses

Double-click into stdio_init_all and look for hardware addresses:

```
ldr r0, =0x40070000 ; UART0 base address
```

Check the RP2350 datasheet Section 2.2 (Address Map):

- 0x40070000 = UART0

This confirms it's a UART initialization function!

Part 17: Understanding Branch Instructions

ARM Branch Instructions

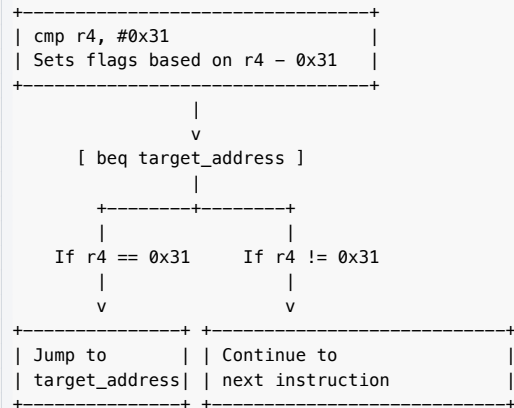
Instruction	Meaning	Condition
b	Branch (always)	Unconditional jump
beq	Branch if Equal	Zero flag set
bne	Branch if Not Equal	Zero flag clear
bgt	Branch if Greater Than	Signed greater
blt	Branch if Less Than	Signed less

How Conditionals Become Branches

```
if (choice == 0x31) {
    printf("1");
}
```

Becomes:

```
cmp    r4, #0x31      ; Compare choice to '1'
bne    skip_printf    ; If NOT equal, skip the printf
; ... printf code here ...
skip_printf:
```



Part 18: Advanced Hacking - Creating Stealth Commands

The Goal

We want to create **secret commands** that:

1. Respond to 'x' and 'y' instead of '1' and '2'
2. Move the servo **WITHOUT** printing anything
3. Leave **NO** trace in the terminal

Step 54: Plan the Patches

Original behavior:

- '1' (0x31) -> prints "1" and "one", moves servo
- '2' (0x32) -> prints "2" and "two", moves servo

Hacked behavior:

- 'x' (0x78) -> moves servo SILENTLY (replacing '1')
- 'v' (0x79) -> moves servo SILENTLY (replacing '2')

Step 55: Change Comparison Values

Navigate to the `main` function and find the two `cmp` instructions right after `getchar`:

1. At address `1000024a`, you will see `cmp r4,#0x31`.
 - Right-click it, select **Patch Instruction**, and change it to `cmp r4,#0x78` (which is ASCII 'x').
2. At address `1000024e`, you will see `cmp r4,#0x32`.
 - Right-click it, select **Patch Instruction**, and change it to `cmp r4,#0x79` (which is ASCII 'y').

Step 56: Redirect Branches to Skip Prints

For the stealth keys, we need to jump PAST the printf calls directly to the servo code.

Original flow:

```
compare -> branch -> printf("1") -> printf("one") -> servo code
```

Hacked flow:

```
compare 'x' -> branch -> [skip prints] -> servo code
```

Use **Patch Instruction** to rewrite the beq target addresses:

- At address 1000024c, you will see beq LAB_10000270.
 - 10000270 is the block that prints “1”. We want to skip it!
 - Right-click, select **Patch Instruction**, and change it to beq 0x1000027c (this jumps straight to the mov r0, r6 servo code!).
- At address 10000250, you will see beq LAB_1000029a.
 - 1000029a is the block that prints “2”.
 - Right-click, select **Patch Instruction**, and change it to beq 0x100002a6 (this jumps straight to the mov r0, r5 servo code!).

Step 57: NOP Out Print Calls (Alternative Method)

NOP (No Operation) is an instruction that does absolutely nothing. Hackers use it to “erase” code without changing the size of the binary! Since we already redirected the branches to skip the prints, this is technically redundant, but let’s do it anyway just to learn the technique.

The bl instruction is 32-bits (4 bytes) long. Standard Thumb nop instructions are only 16-bits (2 bytes) long. If you try to patch a 4-byte instruction with a 2-byte instruction, Ghidra’s assembler gets very confused and corrupts the code.

To fix this, we use the special 32-bit version of NOP: nop.w (Wide NOP).

- In the Listing view, right-click the bl FUN_10001954 instruction at 10000272.
- Select **Patch Instruction**.
- Type nop.w (don’t forget the .w!) and hit Enter.
- You will see the entire 4-byte instruction neatly get replaced by a single 32-bit NOP.
- Repeat this for the second bl instruction at 10000278.

Step 58: Summary of Control Flow Patches

Here is a quick summary of the stealth command patches we just applied to the control flow:

Location	Original Action	Patched Action	Purpose
1000024a	cmp r4, #0x31	cmp r4, #0x78	Check for ‘x’ instead of ‘1’
1000024e	cmp r4, #0x32	cmp r4, #0x79	Check for ‘y’ instead of ‘2’
1000024c	beq LAB_10000270	beq LAB_1000027c	Skip printf for ‘x’
10000250	beq LAB_1000029a	beq LAB_100002a6	Skip printf for ‘y’

Step 59: One Final Hack - Modify the Angle

Let’s also change the servo’s movement angle from 180° to 30° for fun!

If you look back near the top of the main function (around address 10000242), you’ll see this instruction: ldr r5, [DAT_100002c4] = 43340000h

The compiler stored the 180.0 float value (0x43340000) in a literal pool at address 100002c4. To change the angle, we just need to overwrite that raw data!

Original: 0x43340000 (180.0f) **New:** 0x41f00000 (30.0f)

Here is how to apply the patch:

- Press G and jump to address 100002c4.

2. In your **Bytes** window (make sure the Pencil icon is still clicked!), you will see the raw little-endian bytes:
00 00 34 43.
3. Click on the first 00 and type 00 00 f0 41.
4. The Listing view will instantly update to show the new 41f00000 value!

(For the math nerds, here is how we calculated 0x41f00000 manually using the IEEE-754 standard):

Calculation for 30.0f:

```
30.0 = 1.875 * 2^4
Sign = 0
Exponent = 127 + 4 = 131 = 0x83
Mantissa = 0.875 = 0x700000

Binary: 0 10000011 111000000000000000000000
Hex: 0x41f00000
Little-endian: 00 00 f0 41
```

Step 60: Export and Test

When exporting, make sure to save it in your build/ folder!

1. Export as 0x0020_dynamic-conditionals-h.bin inside build/.
2. Convert and flash (run from the 0x0020_dynamic-conditionals directory!):

```
python ../uf2conv.py build\0x0020_dynamic-conditionals-h.bin --base 0x10000000 --family 0xe48bff59 --output
↳ build\hacked.uf2
```

3. Flash and test:
 - Press 'x' -> NO OUTPUT, but servo moves silently!
 - Press 'y' -> NO OUTPUT, but servo moves silently!
 - (The original '1' and '2' keys no longer work!)

Part 19: Summary and Review

What We Accomplished

1. **Learned static vs dynamic conditionals** - Fixed vs runtime-determined values
2. **Understood if/else and switch/case** - Two ways to branch in C
3. **Mastered PWM calculations** - 150MHz to 50Hz servo signal
4. **Identified conditional branches in assembly** - beq, bne, cmp instructions
5. **Hacked string literals** - Changed "one" to "fun"
6. **Modified timing values** - Sped up servo from 500ms to 100ms
7. **Created stealth commands** - Hidden 'x' and 'y' keys
8. **NOPed out print statements** - Removed logging for stealth
9. **Redirected branch targets** - Changed program flow

Static vs Dynamic Summary

Static Conditionals

- Variable set once, never changes
- Same path taken every iteration
- Compiler may optimize out dead branches
- Example: int choice = 1; if (choice == 1)

Dynamic Conditionals

- Variable changes based on input/sensors

- Different paths taken based on runtime state
- All branches must remain in binary
- Example: `choice = getchar(); if (choice == '1')`

PWM Calculation Summary

Angle degrees	Formula	Pulse Width μ s	1us= 1 tick	PWM Ticks	Servo Motion	Servo Motion
0°		1000 μ s		1000 ticks		Fully CCW
90°		1500 μ s		1500 ticks		Center
180°		2000 μ s		2000 ticks		Fully CW

Key Memory Addresses

Memory Address	Description
0x10000234	main() function
0x40070000	UART0 hardware registers
0x1f4	500 (sleep_ms delay)
0x7D0	2000 (max pulse width)
0x3E8	1000 (min pulse width)
0x43340000	180.0f (max angle)

Key Takeaways

1. **Static conditionals have fixed outcomes** - The same path always executes
2. **Dynamic conditionals respond to input** - Different paths based on runtime state
3. **PWM frequency = 50Hz for servos** - One pulse every 20ms
4. **Pulse width encodes position** - 1ms=0°, 1.5ms=90°, 2ms=180°
5. **beq = branch if equal** - Jumps when comparison matches
6. **bne = branch if not equal** - Jumps when comparison doesn't match
7. **NOP erases code without changing size** - 00 bf in ARM Thumb
8. **Branch targets can be redirected** - Change where code jumps to
9. **IEEE-754 is needed for angles** - Floats have specific bit patterns
10. **Stealth requires removing ALL output** - NOP out printf AND puts

Glossary

Term	Definition
beq	Branch if Equal - ARM conditional jump
bne	Branch if Not Equal - ARM conditional jump
Dynamic Conditional	Condition that changes based on runtime input
Duty Cycle	Percentage of time signal is HIGH
getchar()	C function that reads one character from input
NOP	No Operation - instruction that does nothing
PWM	Pulse Width Modulation - variable duty cycle signal
SG90	Common hobby servo motor model
Static Conditional	Condition with fixed/predetermined outcome
switch/case	C structure for multiple discrete value comparisons
Wrap Value	PWM counter maximum before reset

Additional Resources

ASCII Reference Table

Character	Hex	Decimal
'0'	0x30	48
'1'	0x31	49
'2'	0x32	50
'x'	0x78	120
'y'	0x79	121
'\r'	0x0d	13
'\n'	0x0a	10

IEEE-754 Common Angles

Angle	IEEE-754 Hex	Little-Endian Bytes
0.0	0x00000000	00 00 00 00
30.0	0x41f00000	00 00 f0 41
45.0	0x42340000	00 00 34 42
90.0	0x42b40000	00 00 b4 42
135.0	0x43070000	00 00 07 43
180.0	0x43340000	00 00 34 43

ARM Thumb NOP Encodings

Instruction	Encoding	Size
nop	00 bf	2 bytes
nop.w	00 f0 00 80	4 bytes

RP2350 Key Addresses

Address	Peripheral
0x40070000	UART0
0x40078000	UART1
0x40050000	PWM

Real-World Implications

Why Stealth Commands Matter

The ability to create hidden commands has serious implications:

Legitimate Uses:

- Factory test modes
- Debugging interfaces
- Emergency recovery features

Malicious Uses:

- Backdoors in firmware
- Hidden surveillance features
- Unauthorized control of systems

Real-World Example

Imagine a drone with hacked firmware:

- Normal keys ('1', '2') control it visibly with logging
- Hidden keys ('x', 'y') control it with NO log entries
- An attacker could operate the drone while security monitors show nothing

The Nuclear Fuel Rod Analogy

A fast-moving servo is like a nuclear fuel rod:

- Both are small components with immense power
- Both require precise control to prevent damage
- Both can “go critical” if pushed beyond limits
- Both teach the importance of safety margins

Remember: The techniques you learned today demonstrate how conditional logic can be manipulated at the binary level. Understanding these attacks helps us build more secure embedded systems. Always use your skills ethically and responsibly!

Happy hacking!