
EMBEDDED HACKING

Reverse Engineering the Raspberry Pi Pico 2

Week 9: Operators in Embedded Systems: Debugging and Hacking Operators w/ DHT11 Temperature & Humidity Sensor Single-Wire 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 all six types of C operators (arithmetic, increment, relational, logical, bitwise, assignment)
 - Know how the DHT11 temperature and humidity sensor communicates with the Pico 2
 - Understand how post-increment operators affect variable values
 - Navigate to the Reset_Handler and main function in stripped binaries
 - Identify function arguments by analyzing register values in Ghidra
 - Understand IEEE-754 floating-point representation
 - Hack floating-point constants to manipulate sensor readings
-

Part 1: Understanding C Operators

What Are Operators?

Operators are symbols that tell the compiler to perform specific mathematical, logical, or data manipulation operations. Think of them as the “verbs” of programming - they describe actions to perform on data.

The Six Types of Operators

Type	Example	What It Does
Arithmetic	<code>x * y</code>	Math operations (+, -, *, /, %)
Increment	<code>++x</code> or <code>x++</code>	Increase/decrease by 1
Relational	<code>x > y</code>	Compare values (returns true/false)
Logical	<code>(x > y) && (y > x)</code>	Combine conditions (AND, OR, NOT)
Bitwise	<code>x << 1</code>	Manipulate individual bits
Assignment	<code>x += 5</code>	Assign and modify values

Part 2: Arithmetic Operators

Basic Math in C

Arithmetic operators perform mathematical calculations:

```
int x = 5;
int y = 10;
int result = x * y; // result = 50
```

Arithmetic Operators			
Operator	Example	Result	Description
+	5 + 10	15	Addition
-	10 - 5	5	Subtraction
*	5 * 10	50	Multiplication
/	10 / 5	2	Division
%	10 % 3	1	Modulus (remainder)

Part 3: Increment and Decrement Operators

Pre-Increment vs Post-Increment

This is where many beginners get confused! There are TWO ways to increment:

```
int x = 5;
int a = x++; // Post-increment: a = 5, then x becomes 6
int b = ++x; // Pre-increment: x becomes 7, then b = 7
```

The Key Difference:

Type	Syntax	When Value Changes	Example Result
Post-increment	x++	AFTER the expression	a = x++ -> a=5, x=6
Pre-increment	++x	BEFORE the expression	b = ++x -> x=7, b=7

Post-Increment (x++)	
int x = 5;	
int result = x++;	
Step 1: result = x (result gets 5)	
Step 2: x = x + 1 (x becomes 6)	
Final: result = 5, x = 6	
Think of it as: "Use first, THEN increment"	

Part 4: Relational Operators

Comparing Values

Relational operators compare two values and return true (1) or false (0):

```
int x = 6;
int y = 10;
bool result = (x > y); // false, because 6 is NOT greater than 10
```

Relational Operators			
Operator	Example	Result	Description
>	6 > 10	false	Greater than
<	6 < 10	true	Less than
>=	6 >= 6	true	Greater than or equal
<=	6 <= 10	true	Less than or equal
==	6 == 10	false	Equal to
!=	6 != 10	true	Not equal to

Part 5: Logical Operators

Combining Conditions

Logical operators combine multiple conditions:

```
int x = 6;
int y = 10;
bool result = (x > y) && (y > x); // false AND true = false
```

Logical Operators			
Operator	Name	Example	Result
&&	AND	true && true	true
&&	AND	true && false	false
	OR	true false	true
	OR	false false	false
!	NOT	!true	false

Truth Table for AND (&&):

A	B	A && B
false	false	false
false	true	false
true	false	false
true	true	true

Part 6: Bitwise Operators

Manipulating Individual Bits

Bitwise operators work on the binary representation of numbers:

```
int x = 6; // Binary: 0b00000110
int result = x << 1; // Shift left by 1: 0b00001100 = 12
```

```

Bitwise Left Shift (<<)

x = 6 = 0b00000110

x << 1 means "shift all bits LEFT by 1 position"

Before: 0 0 0 0 0 1 1 0 (6)
        ? ? ? ? ? ? ? ?
After:  0 0 0 0 1 1 0 0 (12)

Each left shift DOUBLES the value!
6 << 1 = 12 (same as 6 * 2)

```

Common Bitwise Operators:

Operator	Name	Example	Result
<<	Left shift	6 << 1	12 (multiply by 2)
>>	Right shift	6 >> 1	3 (divide by 2)
&	AND	6 & 3	2 (bits in common)
\	OR	6 \ 3	7 (all set bits)
^	XOR	6 ^ 3	5 (different bits)
~	NOT	~6	Inverts all bits

Part 7: Assignment Operators

Shorthand for Math + Assign

Assignment operators combine math with assignment:

```

int x = 6;
x += 5;    // Same as: x = x + 5; Result: x = 11

```

Compound Assignment Operators			
Operator	Example	Equivalent To	Result (if x=6)
+=	x += 5	x = x + 5	x = 11
-=	x -= 2	x = x - 2	x = 4
*=	x *= 3	x = x * 3	x = 18
/=	x /= 2	x = x / 2	x = 3
%=	x %= 4	x = x % 4	x = 2

Part 8: Understanding the DHT11 Sensor

What is the DHT11?

The **DHT11** is a low-cost digital temperature and humidity sensor. It uses a single wire for communication (plus power and ground).

```

DHT11 Pinout

```



DHT11 Specifications

Parameter	Range	Accuracy
Humidity	20% - 90% RH	?5% RH
Temperature	0 deg C - 50 deg C	?2 deg C

How DHT11 Communication Works

The DHT11 uses a custom one-wire protocol:

1. **Host sends start signal** - Pull data line LOW for 18ms
2. **DHT11 responds** - Pulls line LOW for 80 us, then HIGH for 80 us
3. **Data transmission** - 40 bits sent (8 humidity int, 8 humidity decimal, 8 temp int, 8 temp decimal, 8 checksum)

Part 9: Understanding Pointers (Quick Review)

The & Operator (Address-Of)

When you see `&variable`, it means “the memory address of variable”:

```
float hum, temp;

// Pass ADDRESSES to the function so it can modify our variables
if (dht11_read(&hum, &temp)) {
    printf("Humidity: %.1f%%, Temperature: %.1f?C\n", hum, temp);
}
```

```
Passing by Reference

Stack Memory
+-----+
| Address 0x20000008: hum |?--- &hum (passed to function)|
| Value: 51.0           |                               |
+-----+
| Address 0x2000000C: temp|?--- &temp (passed to function)|
| Value: 23.8           |                               |
+-----+

dht11_read() receives the ADDRESSES, so it can write
```

```
| new values directly into hum and temp!
|
+-----+
|
```

Part 10: Setting Up Your Environment

Prerequisites

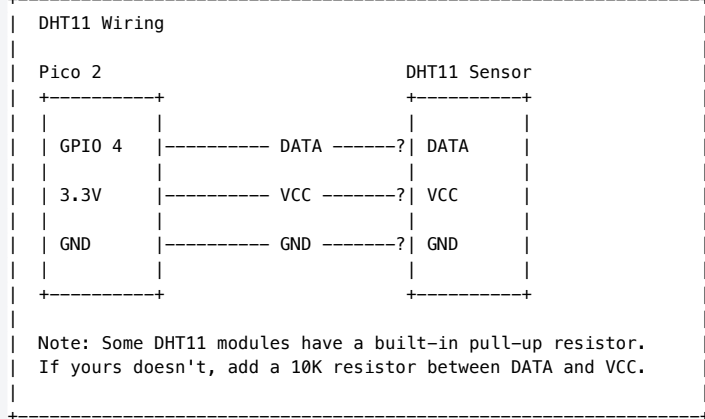
Before we start, make sure you have:

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. A DHT11 temperature and humidity sensor
7. The sample project: `0x001a_operators`

Hardware Setup

Connect your DHT11 like this:

DHT11 Pin	Pico 2 Pin
VCC	3.3V
DATA	GPIO 4
GND	GND



Project Structure

```

Embedded-Hacking/
+-- 0x001a_operators/
|   +-- build/
|   |   +-- 0x001a_operators.uf2
|   |   +-- 0x001a_operators.bin
|   +-- main/
|   |   +-- 0x001a_operators.c
|   +-- dht11.h
+-- uf2conv.py

```

Part 11: Hands-On Tutorial - The Operators Code

Step 1: Review the Source Code

Let's examine the operators code:

File: 0x001a_operators.c

```
#include <stdio.h>
#include "pico/stdlib.h"
#include "dht11.h"

int main(void) {
    stdio_init_all();

    dht11_init(4);

    int x = 5;
    int y = 10;
    int arithmetic_operator = (x * y);
    int increment_operator = x++;
    bool relational_operator = (x > y);
    bool logical_operator = (x > y) && (y > x);
    int bitwise_operator = (x << 1); // x is now 6 because of x++ or 0b00000110 and (x << 1) is 0b00001100 or 12
    int assignment_operator = (x += 5);

    while (true) {
        printf("arithmetic_operator: %d\r\n", arithmetic_operator);
        printf("increment_operator: %d\r\n", increment_operator);
        printf("relational_operator: %d\r\n", relational_operator);
        printf("logical_operator: %d\r\n", logical_operator);
        printf("bitwise_operator: %d\r\n", bitwise_operator);
        printf("assignment_operator: %d\r\n", assignment_operator);

        float hum, temp;
        if (dht11_read(&hum, &temp)) {
            printf("Humidity: %.1f%%, Temperature: %.1f?C\r\n", hum, temp);
        } else {
            printf("DHT11 read failed\r\n");
        }

        sleep_ms(2000);
    }
}
```

Step 2: Understand the Variable Flow

Let's trace through what happens to x:

Variable x Through the Program		
Line	x value	Result
int x = 5;	5	x initialized to 5
x * y	5	arithmetic = 5 * 10 = 50
x++	5→6	increment = 5 (then x becomes 6)
x > y	6	relational = (6 > 10) = false
(x>y) && (y>x)	6	logical = false && true = false
x << 1	6	bitwise = 6 << 1 = 12
x += 5	6→11	assignment = 6 + 5 = 11

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 0x001a_operators.uf2 onto the drive
5. The Pico will reboot and start running!

Step 4: Verify It's Working

Open your serial monitor (PuTTY at 115200 baud) and you should see:

```
arithmetic_operator: 50
increment_operator: 5
relational_operator: 0
logical_operator: 0
bitwise_operator: 12
assignment_operator: 11
Humidity: 51.0%, Temperature: 23.8 deg C
```

Understanding the Output:

Variable	Value	Explanation
arithmetic_operator	50	$5 * 10 = 50$
increment_operator	5	Post-increment returns value BEFORE increment
relational_operator	0	$6 > 10$ is false (0)
logical_operator	0	false AND true = false (0)
bitwise_operator	12	$6 (0b0110) \ll 1 = 12 (0b1100)$
assignment_operator	11	$6 + 5 = 11$
Humidity	51.0%	Real reading from DHT11
Temperature	23.8 deg C	Real reading from DHT11

Part 12: Debugging with GDB

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\0x001a_operators.elf
```

Step 7: Connect to the Remote Target

Inside GDB, type:

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

This connects GDB to OpenOCD.

Step 8: Halt the Running Binary

```
monitor reset halt
```

This stops the Pico 2 so we can examine its state.

Step 9: Examine Main Function

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

```
x/60i 0x10000234
```

You should see the operator calculations and function calls:

```
0x10000234 <main>:  push    {r4, r5, r6, r7, lr}
0x10000236 <main+2>:  sub     sp, #20
0x10000238 <main+4>:  bl      0x10003384 <stdio_init_all>
0x1000023c <main+8>:  movs    r0, #4
0x1000023e <main+10>: bl      0x100002d4 <dht11_init>
...
```

Step 10: Set a Breakpoint at Main

```
b *0x10000234
c
```

Step 11: Find the Operator Calculations

The compiler likely optimized many of these calculations at compile time. Look for immediate values:

```
x/32i 0x10000240
```

You may see values like:

- #0x32 (50) for arithmetic_operator
- #0x5 (5) for increment_operator
- #0x0 (0) for relational and logical operators
- #0xc (12) for bitwise_operator
- #0xb (11) for assignment_operator

Step 12: Examine Printf Arguments

Set a breakpoint before the first printf and examine registers:

```
b *0x10000262
c
i r r0 r1
```

You should see:

- r0 = address of format string
- r1 = value to print

Step 13: Examine the Format Strings

```
x/s 0x10003978
```

Find the format strings and value for print:

```
(gdb) x/s 0x10003978
0x10003978:  "Humidity: %.1f%%, Temperature: %.1fA°C\r\n"
(gdb) x/x 0x4037cccc
0x4037cccc:  0x00
```

```
(gdb) x/x $r1
0x4037cccc:    0x00
...
```

Step 14: Examine DHT11 Function Call

Find where dht11_read is called:

```
(gdb) x/3i 0x1000029f
```

You'll see stack addresses being passed as arguments:

```
0x1000029f <main+106>:    add    r1, sp, #12
0x100002a1 <main+108>:    add    r0, sp, #8
0x100002a3 <main+110>:    bl     0x100002f4 <dht11_read>
```

Step 15: Watch the Float Values

After dht11_read returns, examine the float values on the stack:

```
(gdb) x/2fw $sp+8
0x20081fe0:    62      23.7999992
```

This shows the humidity and temperature as floats.

Step 16: Step Through the Loop

Continue execution and watch the values:

```
c
```

The program will loop, printing values to serial.

Part 13: Setting Up Ghidra for Analysis

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: 0x001a_operators
5. Click **Finish**

Step 19: Import the Binary

1. Open your file explorer
2. Navigate to the 0x001a_operators/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 14: Finding the Reset_Handler

Step 22: Understand the Vector Table

In ARM Cortex-M, the **vector table** is at the base of flash (`0x10000000`). The second entry (offset 4) contains the `Reset_Handler` address.

ARM Vector Table at 0x10000000		
Offset	Contents	Description
0x00	Initial SP value	Stack pointer at reset
0x04	Reset_Handler addr	First code to execute
0x08	NMI_Handler addr	Non-maskable interrupt
0x0C	HardFault_Handler	Hard fault handler
...		

Step 23: Read the Reset_Handler Address

1. Press G (Go to address) and type `10000004`
2. You'll see bytes like `5d 01 00 10` (your exact bytes may vary)

Important: This is **little-endian**, so we need to reverse the byte order!

Little-Endian Byte Order	
In memory:	<code>5d 01 00 10</code>
Reversed:	<code>10 00 01 5d</code>
As hex:	<code>0x1000015d</code>
But wait! ARM uses the THUMB bit!	
The lowest bit indicates Thumb mode (always set for Cortex-M)	
Real address: <code>0x1000015d - 1 = 0x1000015c</code>	

Step 24: Navigate to Reset_Handler

1. Press G and type `1000015c` (or your calculated address)
2. You might see undefined data - that's OK!

Step 25: Create the Reset_Handler Function

If Ghidra didn't automatically recognize this as a function:

1. Click on the address `0x1000015c`
2. Right-click and press F to create a function
3. Right-click -> **Edit Function Signature**
4. Change the name to `Reset_Handler`
5. Click **OK**

Step 26: Find Main from Reset_Handler

The `Reset_Handler` typically calls three functions:

```

+-----+
| Reset_Handler Flow (crt0.S) |
+-----+
| Reset_Handler:              |
|   1. Call some_init()       |-- Initialize hardware |
|   2. Call main()            |-- THIS IS WHAT WE WANT! |
|   3. Call exit()            |-- Never returns         |
|                               |
| The MIDDLE function is main! |
+-----+

```

Look at the end of `Reset_Handler` for three function calls. The middle one is `main`!

Step 27: Navigate to Main

1. Double-click on the middle function call (should be around `0x10000234`)
2. Right-click -> **Edit Function Signature**
3. Change to: `int main(void)`
4. Click **OK**

Part 15: Resolving Functions in Ghidra**Step 28: Resolve stdio_init_all**

The first function call in `main` is `stdio_init_all`:

1. Find the call at approximately `0x10000238`
2. Double-click to navigate to the function
3. Right-click -> **Edit Function Signature**
4. Change to: `bool stdio_init_all(void)`
5. Click **OK**

Step 29: Resolve dht11_init

Look for a function call where `r0` is loaded with `0x4`:

```

movs r0, #0x4      ; GPIO pin 4
bl  FUN_0000xxxx    ; dht11_init

```

How do we know it's dht11_init?

- The argument 4 is the GPIO pin number
- We physically connected the DHT11 to GPIO 4!

1. Right-click -> **Edit Function Signature**
2. Change to: `void dht11_init(uint pin)`
3. Click **OK**

Step 30: Resolve printf

Look for repeated function calls with string addresses:

1. Find a call like the one at 0x10000262
2. Right-click -> **Edit Function Signature**
3. Change to: `int printf(char *format,...)`
4. Check the **Varargs** checkbox
5. Click **OK**

Step 31: Resolve sleep_ms

Look for a function call where `r0` is loaded with 0x7d0 (2000 in decimal):

```
ldr r0, =0x7d0 ; 2000 milliseconds
bl  FUN_XXXXX ; sleep_ms
```

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

Step 32: Resolve dht11_read

This is trickier! Look for a function call with TWO address arguments:

```
add r1, sp, #0xc ; Address of temp on stack
add r0, sp, #0x8 ; Address of hum on stack
bl  FUN_XXXXX ; dht11_read
```

Understanding the stack offsets:

- `sp + 0x8` = address of hum variable
- `sp + 0xc` = address of temp variable
- These are float pointers passed to the function

1. Right-click -> **Edit Function Signature**
2. Change to: `bool dht11_read(float *humidity, float *temperature)`
3. Click **OK**

Step 33: Resolve puts

Look for a function call after the if statement that takes a single string argument:

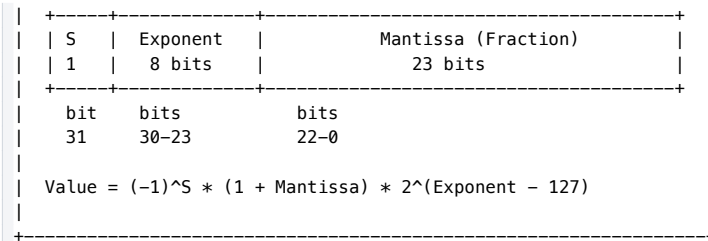
```
ldr r0, ="DHT11 read failed"
bl  FUN_XXXXX ; puts
```

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

Part 16: Understanding IEEE-754 Floating-Point**What is IEEE-754?**

IEEE-754 is the standard for representing decimal numbers in binary. A 32-bit float is divided into three parts:

```
+-----+
| IEEE-754 Single Precision (32-bit) Float |
+-----+
```



Example: Decoding 0x3dcccccc (0.1f)

Let's decode the bytes cc cc cc 3d:

- Reverse for little-endian:** 0x3dcccccc
- Convert to binary:** 00111101 11001100 11001100 11001100
- Extract fields:**
 - Sign (bit 31): 0 (positive)
 - Exponent (bits 30-23): 01111011 = 123
 - Mantissa (bits 22-0): 10011001100110011001100
- Calculate value:**
 - Actual exponent: $123 - 127 = -4$
 - Mantissa value: 1.6 (approximately)
 - Final value: $1.6 * 2^{-4} \approx 0.1$

Example: Encoding -1.0f as 0xbf800000

For the number -1.0:

- Sign:** 1 (negative)
- Exponent:** 127 (for $2^0 = 1$)
- Mantissa:** 0 (because value is exactly 1.0)

Binary: 1 01111111 000000000000000000000000 Hex: 0xbf800000 Little-endian: 00 00 80 bf

Python for Float Conversion

```
import struct

# Decode bytes to float
bytes_data = bytes.fromhex('cdcccc3d')
value = struct.unpack('<f', bytes_data)[0]
print(f"Value: {value}") # 0.1

# Encode float to bytes
float_value = -1.0
encoded = struct.pack('<f', float_value)
print(f"Bytes: {encoded.hex()}") # 0000 80bf
```

Part 17: Finding the Temperature Hack Point

Step 34: Locate the dht11_read Function

Navigate to the dht11_read function you identified earlier.

Step 35: Find the Scaling Constant

At the end of the `dht11_read` function, and around `0x10000410`, look for floating-point instructions. You'll find instructions like:

```
vfma.f32 s14, s12, s11 ; Fused multiply-add for humidity
vfma.f32 s15, s13, s11 ; Fused multiply-add for temperature
```

The constant `0.1` (at address `0x1000042c`) is loaded into register `s11` and used to scale the raw sensor readings.

```
+-----+
| DHT11 Scaling Calculation |
|                             |
| Raw sensor data: integer + decimal parts |
| Example: 238 (integer=23, decimal=8) |
|                             |
| Formula: result = integer + (decimal * 0.1) |
|           23.8 = 23 + (8 * 0.1) |
|                             |
| The vfma.f32 instruction does: s15 = s13 + (s11 * something) |
| Where s11 = 0.1f (our target!) |
|                             |
+-----+
```

Step 36: Identify Key Offsets

Make note of these offsets in the binary file:

Offset	Current Bytes	Current Instruction/Value
0x410	a6 ee 25 7a	vfma.f32 s14, s12, s11 (humidity)
0x414	e6 ee a5 7a	vfma.f32 s15, s13, s11 (temp)
0x42C	cc cc cc 3d	0.1f (scaling constant)

Part 18: Manual Hacking in Ghidra

Step 37: Open the Bytes Editor

1. Click **Window** -> **Bytes**
2. A new panel appears showing raw hex bytes

Step 38: Enable Editing

Look for the pencil icon in the Bytes window toolbar and click it to enable editing mode.

Step 39: Hack the Scaling Constant

Let's change the temperature scaling to add 25% more!

1. Press **G** and go to address `1000042c`
2. Current bytes: `cd cc cc 3d` (0.1f in little-endian)
3. Change to: `00 00 a0 40` (5.0f in little-endian)

This changes the multiplier from 0.1 to 5.0, which will dramatically increase the temperature reading!

Step 40: Verify the Change

Use Python to verify what we changed:

```
import struct

# Original value
original = struct.unpack('<f', bytes.fromhex('cdcccc3d'))[0]
print(f"Original: {original}") # 0.1

# New value
new = struct.unpack('<f', bytes.fromhex('0000a040'))[0]
print(f"New: {new}") # 5.0
```

```
>>> import struct
>>>
>>> # Original value
>>> original = struct.unpack('<f', bytes.fromhex('cdcccc3d'))[0]
>>> print(f"Original: {original}") # 0.1
Original: 0.10000000149011612
>>>
>>> # New value
>>> new = struct.unpack('<f', bytes.fromhex('0000a040'))[0]
>>> print(f"New: {new}") # 5.0
New: 5.0
```

Part 19: Exporting and Testing

Step 41: Export the Patched Binary

1. Click **File -> Export Program**
2. Set **Format to Raw Bytes**
3. Navigate to your build directory
4. Name the file `0x001a_operators-h.bin`
5. Click **OK**

Step 42: Convert to UF2 Format

Open a terminal and run:

```
cd C:\Users\flare-vm\Desktop\Embedded-Hacking-main\0x001a_operators
python ..\uf2conv.py build\0x001a_operators-h.bin --base 0x10000000 --family 0xe48bff59 --output build\hacked.uf2
```

Step 43: Flash and Test

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

You should see dramatically increased temperature readings!

```
Humidity: 60.0%, Temperature: 63.0°C
arithmetic_operator: 50
increment_operator: 5
relational_operator: 0
logical_operator: 0
bitwise_operator: 12
assignment_operator: 11
```

Part 20: Summary and Review

What We Accomplished

1. **Learned all six C operator types** - Arithmetic, increment, relational, logical, bitwise, assignment
2. **Understood post-increment behavior** - `x++` returns value BEFORE incrementing
3. **Learned about the DHT11 sensor** - One-wire protocol for temperature/humidity
4. **Found Reset_Handler from vector table** - Offset 4 contains the address
5. **Identified functions by their arguments** - GPIO pin 4, sleep 2000ms, etc.
6. **Understood IEEE-754 floating-point** - How computers represent decimals
7. **Hacked floating-point constants** - Changed 0.1f to other values

The Hacking Workflow

```

+-----+
| Binary Hacking Workflow |
+-----+
| 1. Analyze the binary in Ghidra |
| 2. Identify target values/instructions |
| 3. Calculate file offsets from memory addresses |
| 4. Determine replacement bytes |
| 5. Patch the binary (manual in hex editor) |
| 6. Export and convert to UF2 |
| 7. Flash and test |
+-----+

```

Key Memory Addresses

Memory Address	File Offset	Description
0x10000000	0x000	Vector table start
0x10000004	0x004	Reset_Handler address
0x10000234	0x234	main() function (approximately)
0x10000410	0x410	Humidity vfma instruction
0x10000414	0x414	Temperature vfma instruction
0x1000042C	0x42C	0.1f scaling constant

Key Takeaways

1. **Post-increment returns the OLD value** - `x++` gives you `x`, THEN adds 1
2. **Bitwise left shift multiplies by 2** - `x << 1` is the same as `x * 2`
3. **Vector table points to Reset_Handler** - Offset 4 from flash base
4. **Arguments go in ro-r3** - Follow them to identify functions
5. **IEEE-754 is how floats are stored** - Sign, exponent, mantissa
6. **File offset = Memory address - Base** - 0x10000410 -> offset 0x410
7. **Little-endian reverses byte order** - 0x3decccc stored as cc cc cc 3d
8. **Incremental testing is essential** - Test each change before the next
9. **Binary patching has real consequences** - Sensor spoofing can be dangerous!

Glossary

Term	Definition
Arithmetic Op	Operators for math (+, -, *, /, %)
Assignment Op	Operators that assign and modify (+=, -=, etc.)
Bitwise Op	Operators on individual bits («, », &, , ^)
DHT11	Digital humidity and temperature sensor
Exponent	Power of 2 in IEEE-754 float representation
IEEE-754	Standard for floating-point number representation
Increment Op	Operators that add/subtract 1 (++ , -)
Little-Endian	Byte order where least significant byte comes first
Logical Op	Operators combining conditions (&&, , !)
Mantissa	Fractional part of IEEE-754 float
Post-Increment	x++ - returns value, then increments
Pre-Increment	++x - increments, then returns value
Relational Op	Operators comparing values (<, >, ==, !=)
Reset_Handler	First function executed after CPU reset
Thumb Bit	Lowest bit of ARM address indicating Thumb mode
Vector Table	Table of exception/interrupt handler addresses
vfma.f32	ARM floating-point fused multiply-add instruction
vadd.f32	ARM floating-point add instruction

Additional Resources

IEEE-754 Float Quick Reference

Value	Hex Encoding	Bytes (LE)
0.1	0x3dcccccd	cd cc cc 3d
1.0	0x3f800000	00 00 80 3f
-1.0	0xbf800000	00 00 80 bf
2.0	0x40000000	00 00 00 40
-2.0	0xc0000000	00 00 00 c0
5.0	0x40a00000	00 00 a0 40
10.0	0x41200000	00 00 20 41

ARM Floating-Point Instructions

Instruction	Description
vfma.f32 Sd, Sn, Sm	Sd = Sd + (Sn * Sm) (fused multiply-add)

Instruction	Description
vadd.f32 Sd, Sn, Sm	$Sd = Sn + Sm$
vsub.f32 Sd, Sn, Sm	$Sd = Sn - Sm$
vmul.f32 Sd, Sn, Sm	$Sd = Sn * Sm$
vldr.f32 Sd, [addr]	Load float from memory
vstr.f32 Sd, [addr]	Store float to memory

Real-World Implications

Why This Matters

Imagine a scenario where temperature sensors control critical systems:

- **Industrial processes** - Chemical reactions that must stay within temperature ranges
- **Medical equipment** - Refrigerators storing vaccines or organs
- **Nuclear facilities** - Cooling systems for reactors
- **HVAC systems** - Climate control in sensitive environments

By manipulating sensor readings, an attacker could:

- Cause equipment to overheat while displaying normal temperatures
- Trigger false alarms
- Bypass safety interlocks
- Cause physical damage or safety hazards

Defensive Measures

1. **Redundant sensors** - Multiple sensors with consistency checks
2. **Physical security** - Prevent access to programming interfaces
3. **Anomaly detection** - Alert on sudden reading changes

Remember: The techniques you learned today can be used for good (security research, debugging) or bad (sabotage, fraud). Always use your skills ethically and legally. Understanding how attacks work helps us build more secure systems!

Happy hacking!