
EMBEDDED HACKING

Reverse Engineering the Raspberry Pi Pico 2

Week 5: Integers and Floats in Embedded Systems: Debugging and Hacking Integers and Floats w/ Intermediate GPIO Output Assembler Analysis

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 how integers and floating-point numbers are stored in memory
- Know the difference between signed and unsigned integers (`uint8_t` vs `int8_t`)
- Understand how floats and doubles are represented using IEEE 754 encoding
- Use inline assembly to control GPIO pins directly at the hardware level
- Debug numeric data types using GDB and the OpenOCD debugger
- Hack integer values by modifying registers at runtime
- Hack floating-point values by understanding and manipulating their binary representation
-

Reconstruct 64-bit doubles from two 32-bit registers

Part 1: Understanding Integer Data Types

What is an Integer?

An **integer** is a whole number without any decimal point. Think of it like counting apples: you can have 0 apples, 1 apple, 42 apples, but you can't have 3.5 apples (that would be a fraction!).

In C programming for embedded systems, we have special integer types that tell the compiler exactly how much memory to use:

```
+-----+
| Integer Types - Different Sizes for Different Needs |
+-----+
| uint8_t:  1 byte (0 to 255)      - like a small box |
| int8_t:   1 byte (-128 to 127)   - can hold negatives! |
| uint16_t: 2 bytes (0 to 65,535)  - medium box |
| uint32_t: 4 bytes (0 to 4 billion) - big box |
+-----+
```

Signed vs Unsigned Integers

The difference between `uint8_t` and `int8_t` is whether the number can be **negative**:

Type	Prefix	Range	Use Case
uint8_t	u	0 to 255	Ages, counts, always positive
int8_t	none	-128 to 127	Temperature, can be negative

The Integer Variables

Let's say a program declares two integer variables that demonstrate the difference between **signed** and **unsigned** types:

```
uint8_t age = 43;
int8_t range = -42;
```

The variable `age` is a `uint8_t` - an **unsigned** 8-bit integer that can only hold values from 0 to 255. Since `age` is always a positive number, unsigned is the right choice. The variable `range` is an `int8_t` - a **signed** 8-bit integer that can hold values from -128 to 127. The signed type allows it to represent negative numbers like -42. Under the hood, negative values are stored using **two's complement** encoding: the CPU flips all the bits of 42 (0x2A) and adds 1, producing 0xD6, which is how -42 lives in a single byte of memory.

Part 2: Understanding Floating-Point Data Types

What is a Float?

A **float** is a number that can have a decimal point. Unlike integers which can only hold whole numbers like 42, a float can hold values like 42.5, 3.14, or -0.001. In C, the float type uses **32 bits (4 bytes)** to store a number using the **IEEE 754** standard.

```
+-----+
| IEEE 754 Single-Precision (32-bit float) |
+-----+
| Sign | Exponent | Mantissa (Fraction) |
| 1bit | 8 bits  | 23 bits              |
+-----+
| Value = (-1)^sign * 2^(exponent-127) * 1.mantissa |
| Example: 42.5 |
| Sign: 0 (positive) |
| Exponent: 10000100 (132 - 127 = 5) |
| Mantissa: 01010100000000000000000 |
| Full: 0 10000100 01010100000000000000000 |
| Hex: 0x422A0000 |
+-----+
```

How to Compute This by Hand (42.5 -> IEEE 754)

Use this exact process any time you need to encode a decimal float manually.

- Determine the sign bit.
 - 42.5 is positive, so `sign = 0`.
- Convert the number to binary.
 - Integer part: 42 = 101010 (base 2)
 - Fractional part: use repeated multiply-by-2 on the fraction.
 - Start with 0.5
 - 0.5 * 2 = 1.0 -> integer part is 1 (this is the first binary fractional bit)

- Remaining fractional part is now 0.0 , so we stop.
 - Therefore $0.5 = 0.1$ (base 2).
 - Combined: $42.5 = 101010.1$ (base 2)
3. Normalize to the form $1.xxxxx \times 2^n$.
 - 101010.1 (base 2) = 1.010101 (base 2) $\times 2^5$
 - So the true exponent is $n = 5$.
 4. Compute the stored exponent (bias 127 for float).
 - stored exponent = $n + 127 = 5 + 127 = 132$
 - 132 in binary is 10000100 (8 bits).

Tip: Why 127? The exponent field is 8 bits wide, giving $2^8 = 256$ total values. Half of that range should represent negative exponents and half positive. The midpoint is $(2^8/2) - 1 = 127$. So a stored exponent of 127 means a real exponent of 0, values below 127 are negative exponents, and values above 127 are positive exponents. Doubles use an 11-bit exponent field so their midpoint (bias) is $(2^{11}/2) - 1 = 1023$ instead.

5. Build the mantissa (fraction bits).
 - Take bits after the leading 1. from $1.010101 \rightarrow 010101$
 - Pad with zeros to 23 bits:
 - 01010100000000000000000
6. Assemble all fields.
 - sign | exponent | mantissa
 - $0 \mid 10000100 \mid 01010100000000000000000$
 - Full 32-bit pattern:
 - $01000010001010100000000000000000$
7. Convert the 32-bit binary to hex (group by 4 bits).
 - $0100 \ 0010 \ 0010 \ 1010 \ 0000 \ 0000 \ 0000 \ 0000$
 - $4 \quad 2 \quad 2 \quad A \quad 0 \quad 0 \quad 0 \quad 0$
 - Final result: $0x422A0000$

Quick decode check (reverse direction, fully expanded):

Given the 32-bit pattern:

• $0 \mid 10000100 \mid 01010100000000000000000$

Decode it field by field:

1. Sign bit
 - Sign bit is 0 $\rightarrow$ number is positive.
 - So the sign multiplier is (+1).
2. Exponent field
 - Exponent bits are 10000100 .
 - Convert to decimal: 10000100 (base 2) = 132.
 - Float bias is 127, so true exponent is:
 - $132 - 127 = 5$.
3. Mantissa field
 - Stored mantissa bits are 01010100000000000000000 .
 - IEEE 754 normal numbers use an implicit leading 1, so significand becomes:
 - 1.010101 (base 2).

4. Rebuild the value

- Formula: $\text{value} = (+1) * 1.010101 \text{ (base 2)} * 2^5$.
- Shift binary point right by 5:
- $1.010101 * 2^5 = 101010.1 \text{ (base 2)}$.

5. Convert 101010.1 (base 2) to decimal

- Integer part: $101010 = 32 + 8 + 2 = 42$
- Fraction part: $.1 = 1/2 = 0.5$
- Total: $42 + 0.5 = 42.5$

So the decoded value is exactly 42.5.

Float vs Integer - Key Differences

Property	Integer (uint8_t)	Float (float)
Size	1 byte	4 bytes
Precision	Exact	~7 decimal digits
Range	0 to 255	$3.4 * 10^{38}$
Encoding	Direct binary	IEEE 754 (sign/exp/mantissa)
printf	%d	%f

Our Floating-Point Program

Let's look at a simple program that uses a float variable:

File: 0x000e_floating-point-data-type.c

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

int main(void) {
    float fav_num = 42.5;

    stdio_init_all();

    while (true)
        printf("fav_num: %f\r\n", fav_num);
}
```

Note: fav_num is declared inside main, so by C rules it is an automatic (stack) variable. In optimized embedded builds, the compiler may avoid creating a real stack slot and instead materialize the value from read-only constant storage (typically .rodata and/or an ARM literal pool).

An ARM **literal pool** is a small table of constants that the assembler places near code in memory. Instead of encoding a large immediate value directly in an instruction, the CPU executes a load instruction (such as ldr) that reads the constant from that nearby table. That is why Ghidra can show constant loads rather than a classic stack local.

What this code does:

1. Declares a float variable fav_num and initializes it to 42.5
2. Initializes the serial output
3. Prints fav_num forever in a loop using the %f format specifier

Tip: Why %f instead of %d? The %d format specifier tells printf to expect an integer. The %f specifier tells it to expect a floating-point number. Using the wrong one would print garbage!

Step 1: 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 `0x000e_floating-point-data-type.uf2` onto the drive
5. The Pico will reboot and start running!

Step 2: Verify It's Working

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

You should see:

```
fav_num: 42.500000
fav_num: 42.500000
fav_num: 42.500000
...
```

The program is printing `42.500000` because `printf` with `%f` defaults to 6 decimal places.

Part 2.5: Setting Up Ghidra for Float Analysis

Step 3: Start Ghidra

Open a terminal and type:

```
ghidraRun
```

Ghidra will open. Now we need to create a new project.

Step 4: Create a New Project

1. Click **File** -> **New Project**
2. Select **Non-Shared Project**
3. Click **Next**
4. Enter Project Name: `0x000e_floating-point-data-type`
5. Click **Finish**

Step 5: Import the Binary

1. Open your file explorer
2. Navigate to the Embedded-Hacking folder
3. Find `0x000e_floating-point-data-type.bin`
4. Select Cortex M Little Endian 32
5. Select Options and set up the `.text` and offset `10000000`
6. **Drag and drop** the `.bin` file into Ghidra's project window

Step 6: Configure the Binary Format

A dialog appears. The file is identified as a “BIN” (raw binary without debug symbols).

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` (the XIP address!)

3. Click **OK**

Step 7: Open and Analyze

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 (watch the progress bar in the bottom right).

Part 2.6: Navigating and Resolving Functions

Step 8: Find the Functions

Look at the **Symbol Tree** panel on the left. Expand **Functions**.

You’ll see function names like:

- FUN_1000019a
- FUN_10000210
- FUN_10000234

These are auto-generated names because we imported a raw binary without symbols!

Step 9: Resolve Known Functions

From our previous chapters, we know what some of these functions are:

Ghidra Name	Actual Name	How We Know
FUN_1000019a	data_cpy	From Week 3 boot analysis
FUN_10000210	frame_dummy	From Week 3 boot analysis
FUN_10000234	main	This is where our code is!

Step 10: Update Main’s Signature

For main, let’s also fix the return type:

1. Right-click on `main` in the Decompile window
 2. Select **Edit Function Signature**
 3. Change to: `int main(void)`
 4. Click **OK**
-

Part 2.7: Analyzing the Main Function

Step 11: Examine Main in Ghidra

Click on `main` (or `FUN_10000234`). Look at the **Decompile** window:

You’ll see something like:

```
int main(void)
{
    undefined4 uVar1;
    undefined4 extraout_r1;
    undefined4 uVar2;
    undefined4 extraout_r1_00;
```

```

FUN_10002f5c();
uVar1 = DAT_1000024c;
uVar2 = extraout_r1;
do {
    FUN_100030ec(DAT_10000250,uVar2,0,uVar1);
    uVar2 = extraout_r1_00;
} while( true );
}

```

Step 12: Resolve stdio_init_all

1. Click on FUN_10002f5c
2. Right-click -> **Edit Function Signature**
3. Change to: `bool stdio_init_all(void)`
4. Click **OK**

Step 13: Resolve printf

1. Click on FUN_100030ec
2. Right-click -> **Edit Function Signature**
3. Change to: `int printf(char *format,...)`
4. Check the **Varargs** checkbox (printf takes variable arguments!)
5. Click **OK**

Step 14: Understand the Float Encoding

Look at the decompiled code after resolving functions:

```

int main(void)
{
    undefined4 uVar1;
    undefined4 extraout_r1;
    undefined4 uVar2;
    undefined4 extraout_r1_00;

    stdio_init_all();
    uVar1 = DAT_1000024c;
    uVar2 = extraout_r1;
    do {
        printf(DAT_10000250,uVar2,0,uVar1);
        uVar2 = extraout_r1_00;
    } while( true );
}

```

Where's float fav_num = 42.5? It's been optimized into an immediate value!

The compiler replaced our float variable with constants passed directly to printf. But wait - we see **two** values: `0x0`, in `r2` and `DAT_1000024c` or `0x40454000`, in `r3`. That's because printf with `%f` always receives a **double** (64-bit), not a float (32-bit). The C standard requires that float arguments to variadic functions like printf are **promoted to double**.

A 64-bit double is passed in two 32-bit registers:

Register	Value	Role
r2	0x00000000	Low 32 bits
r3	0x40454000	High 32 bits

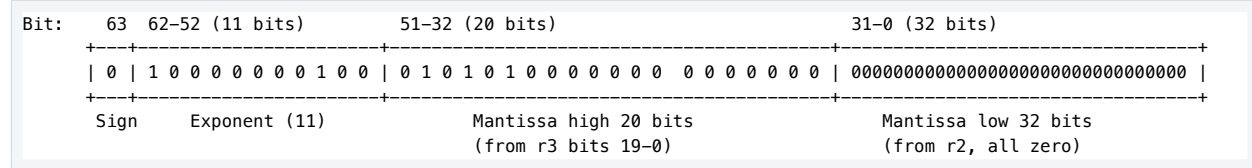
Together they form `0x40454000_00000000` - the IEEE 754 **double-precision** encoding of 42.5.

Step 15: Verify the Double Encoding

We need to decode `0x4045400000000000` field by field. The two registers give us the full 64-bit value:

```
r3 (high 32 bits): 0x40454000 = 0100 0000 0100 0101 0100 0000 0000 0000
r2 (low 32 bits): 0x00000000 = 0000 0000 0000 0000 0000 0000 0000 0000
```

Laid out as a single 64-bit value with every bit numbered:



Step-by-step field extraction:

1. Sign bit

In IEEE 754, the **sign bit** is the very first (leftmost) bit of the 64-bit double. In the full 64-bit layout we call it **bit 63**:

```
64-bit double: [bit 63] [bit 62 ... bit 0]
                ^
                sign bit
```

But we don't have a single 64-bit register - we have **two** 32-bit registers. The high register `r3` holds bits 63-32 of the double. So bit 63 of the double is the same physical bit as **bit 31 of `r3`** (the topmost bit of `r3`):

```
r3 holds bits 63-32 of the double
r2 holds bits 31-0 of the double
```

Now let's check it. IEEE 754 uses a simple rule for the sign bit:

Sign bit	Meaning
0	Positive
1	Negative

```
r3 = 0x40454000 = 0100 0000 0100 0101 0100 0000 0000 0000
                ^
                r3 bit 31 = 0 -> sign = 0 -> Positive number
```

The topmost bit of `r3` is 0, so the number is **positive**. If that bit were 1 instead (e.g. `0xC0454000`), the number would be negative (-42.5).

2. Exponent - bits 62-52 of the 64-bit value = bits 30-20 of `r3`

Extract bits 30-20 from `0x40454000`:

```
0x40454000 in binary: 0      10000000100      01010100000000000000
                    sign  exponent      mantissa (top 20 bits)
```

Exponent bits: `10000000100`

Convert to decimal: $2^{10} + 2^2 = 1024 + 4 = 1028$

But 1028 is **not** the actual power of 2 yet. IEEE 754 stores exponents with a **bias** - a fixed number that gets added during encoding so that the stored value is always positive (no sign bit needed for the exponent). For doubles, the bias is **1023**.

Tip: Why 1023? The exponent field is 11 bits wide, giving $2^{11} = 2048$ total values. Half of that range should represent negative exponents and half positive. The midpoint is $(2^{11}/2) - 1 = 1023$. So a stored exponent of 1023 means a real exponent of 0, values below 1023 are negative exponents, and values above 1023 are positive exponents.

To recover the real exponent, we subtract the bias:

$$\text{real exponent} = \text{stored exponent} - \text{bias}$$

$$\text{real exponent} = 1028 - 1023 = 5$$

This means the number is scaled by $2^5 = 32$. In other words, the mantissa gets shifted left by 5 binary places.

3. Mantissa - bits 51-0 of the 64-bit value

- **High 20 bits of mantissa** (bits 51-32) = bits 19-0 of r3:

```
r3 bits 19-0: 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

- **Low 32 bits of mantissa** (bits 31-0) = all of r2:

```
r2 = 0x00000000 -> 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Full 52-bit mantissa:

```
0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
<- top 20 bits from r3 ->                <- bottom 32 bits from r2 (all zero) ->
```

IEEE 754 always prepends an **implied leading 1**, so the actual value represented is:

```
1.010101 00000... (the 1. is implicit, not stored)
```

4. Reconstruct the value

$$1.010101 (\text{base } 2) \times 2^5$$

Shift the binary point 5 places right:

$$101010.1 (\text{base } 2)$$

Now convert each bit position to decimal:

Bit position	Power of 2	Value
1 (bit 5)	2^5	32
0 (bit 4)	2^4	0
1 (bit 3)	2^3	8
0 (bit 2)	2^2	0
1 (bit 1)	2^1	2
0 (bit 0)	2^0	0
1 (bit -1)	2^{-1}	0.5

$$32 + 8 + 2 + 0.5 = \mathbf{42.5}$$

Step 16: Examine the Assembly

Look at the **Listing** window (assembly view). Find the main function:

```
*****
*                                     FUNCTION
*****
int __stdcall main (void )
```

int	r0:4	<RETURN>	
	main+1		XREF[1,1]: 1000018c (c) , 1000018a (*)
	main		
10000234 38 b5	push	{r3,r4,r5,lr}	
10000236 02 f0 91 fe	bl	stdio_init_all	bool stdio_init_all(void)
1000023a 00 24	movs	r4,#0x0	
1000023c 03 4d	ldr	r5,[DAT_1000024c]	= 40454000h
	LAB_1000023e		XREF[1]: 10000248 (j)
1000023e 22 46	mov	r2,r4	
10000240 2b 46	mov	r3,r5	
10000242 03 48	ldr	r0=>s_fav_num:_%f_100034a8 ,[DAT_10000250]	= "fav_num: %f\r\n"
			= 100034A8h
10000244 02 f0 52 ff	bl	printf	int printf(char * format, ...)
10000248 f9 e7	b	LAB_1000023e	
1000024a 00	??	00h	
1000024b bf	??	BFh	
	DAT_1000024c		XREF[1]: main:1000023c (R)
1000024c 00 40 45 40	undefine	40454000h	
	DAT_10000250		XREF[1]: main:10000242 (R)
10000250 a8 34 00 10	undefine	100034A8h	* -> 100034a8

Key Insight: The `mov.w r2, #0x0` loads the low 32 bits (all zeros) and `ldr r3, [DAT_...]` loads the high 32 bits (0x40454000) of the double. Together, `r2:r3 = 0x40454000_00000000 = 42.5` as a double.

Step 17: Find the Format String

In the Listing view, click on the data reference to find the format string:

	s_fav_num:_%f_100034a8	XREF[1]:	main:10000242 (*)
100034a8 66 61 76	ds	"fav_num: %f\r\n"	
5f 6e 75			
6d 3a 20			

This confirms `printf` is called with the format string `"fav_num: %f\r\n"` and the double-precision value of 42.5.

Part 2.8: Patching the Float - Changing 42.5 to 99.0

Step 18: Calculate the New IEEE 754 Encoding

We want to change 42.5 to 99.0. First, we need to figure out the double-precision encoding of 99.0:

Step A - Convert the integer part (99) to binary:

Division	Quotient	Remainder
99 2	49	1
49 2	24	1
24 2	12	0
12 2	6	0
6 2	3	0
3 2	1	1
1 2	0	1

Read remainders bottom-to-top: 99 (base 10) = 1100011 (base 2)

Step B - Convert the fractional part (.0) to binary:

There is no fractional part - .0 is exactly zero, so the fractional binary is just 0.

Step C - Combine:

$$99.0 \text{ (base 10)} = 1100011.0 \text{ (base 2)}$$

Step D - Normalize to IEEE 754 form (move the binary point so there's exactly one 1 before it):

$$1100011.0 \text{ (base 2)} = 1.100011 \text{ (base 2)} \times 2^6$$

We shifted the binary point 6 places left, so the exponent is **6**.

Step E - Extract the IEEE 754 fields:

1. **Sign:** 0 (positive)
2. **Exponent:** $6 + 1023 = 1029 = 10000000101 \text{ (base 2)}$
3. **Mantissa:** $1000110000000000 \dots$ (everything after the 1., padded with zeros to 52 bits)
4. **Full double:** $0x4058C00000000000$

Register	Old Value	New Value
r2	0x00000000	0x00000000
r3	0x40454000	0x4058C000

Since r2 stays $0x00000000$, we only need to patch the high word loaded into r3.

Step 19: Find the Value to Patch

Look in the Listing view for the data that loads the high word of the double:

```
1000024c 00 40 45 40  undefined4  40454000h
```

This is the 32-bit constant that gets loaded into r3 - the high word of our double 42.5.

Step 20: Patch the Constant

1. Click on Window -> Bytes
2. Click on the Pencil icon to enable byte editing
3. At address $1000024c$, overwrite $00\ 40\ 45\ 40$ with $00\ C0\ 58\ 40$ (little-endian for $0x40454000 \rightarrow 0x4058C000$)
4. Press Enter

This changes the high word from $0x40454000$ (42.5 as double) to $0x4058C000$ (99.0 as double).

Part 2.9: Export and Test the Hacked Binary**Step 21: 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 `0x000e_floating-point-data-type-h.bin`
5. Click **OK**

Step 22: Convert to UF2 Format

Open a terminal and navigate to your project directory:

```
cd C:\Users\flare-vm\Desktop\Embedded-Hacking-main\0x000e_floating-point-data-type
```

Run the conversion command:

```
python ../uf2conv.py build\0x000e_floating-point-data-type-h.bin --base 0x10000000 --family 0xe48bff59 --output
↳ build\hacked.uf2
```

Step 23: Flash the Hacked Binary

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

You should see:

```
fav_num: 99.000000
fav_num: 99.000000
fav_num: 99.000000
...
```

BOOM! We hacked the float! The value changed from 42.5 to 99.0!

Part 3: Understanding Double-Precision Floating-Point Data Types

What is a Double?

A **double** (short for “double-precision floating-point”) is like a `float` but with **twice the precision**. While a `float` uses 32 bits, a double uses **64 bits (8 bytes)**, giving it roughly **15-16 significant decimal digits** of precision compared to a `float`’s ~7.

```
+-----+
| IEEE 754 Double-Precision (64-bit double) |
+-----+
| Sign | Exponent | Mantissa (Fraction) |
| 1bit | 11 bits  | 52 bits              |
+-----+
|
| Value = (-1)^sign * 2^(exponent-1023) * 1.mantissa
|
| Example: 42.52525
| Sign: 0 (positive)
| Exponent: 10000000100 (1028 - 1023 = 5)
| Mantissa: 0101010000110011101101100100010110100001110010101100
| Hex: 0x4045433B645A1CAC
|
+-----+
```

Float vs Double - Key Differences

Property	Float (<code>float</code>)	Double (<code>double</code>)
Size	4 bytes (32 bits)	8 bytes (64 bits)
Precision	~7 decimal digits	~15 decimal digits
Exponent	8 bits (bias 127)	11 bits (bias 1023)
Mantissa	23 bits	52 bits
Range	3.4×10^{38}	1.8×10^{308}
printf	%f	%lf
ARM passing	Promoted to double	Native in r2:r3

Tip: Why does precision matter? With a float, the value 42.52525 might be stored as 42.525249 due to rounding. A double can represent it as 42.525250 with much higher fidelity. For scientific or financial applications, that extra precision is critical!

Our Double-Precision Program

Let's look at a program that uses a double variable:

File: 0x0011_double-floating-point-data-type.c

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

int main(void) {
    double fav_num = 42.52525;

    stdio_init_all();

    while (true)
        printf("fav_num: %lf\r\n", fav_num);
}
```

What this code does:

1. Declares a double variable `fav_num` and initializes it to 42.52525
2. Initializes the serial output
3. Prints `fav_num` forever in a loop using the `%lf` format specifier

Tip: %lf vs %f: While `printf` actually treats `%f` and `%lf` identically (both expect a double), using `%lf` makes your intent clear - you're explicitly working with a double, not a float. It's good practice to match the format specifier to your variable type.

Step 1: 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 0x000A_intro-to-doubles.uf2 onto the drive
5. The Pico will reboot and start running!

Step 2: Verify It's Working

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

You should see:

```
fav_num: 42.525250
fav_num: 42.525250
fav_num: 42.525250
...
```

The program is printing 42.525250 because `printf` with `%lf` defaults to 6 decimal places.

Part 3.5: Setting Up Ghidra for Double Analysis

Step 3: Start Ghidra

Open a terminal and type:

```
ghidraRun
```

Ghidra will open. Now we need to create a new project.

Step 4: Create a New Project

1. Click **File -> New Project**
2. Select **Non-Shared Project**
3. Click **Next**
4. Enter Project Name: 0x000A_intro-to-doubles
5. Click **Finish**

Step 5: Import the Binary

1. Open your file explorer
2. Navigate to the Embedded-Hacking folder
3. Find 0x0011_double-floating-point-data-type.bin
4. Select Cortex M Little Endian 32
5. Select Options and set up the .text and offset 10000000
6. **Drag and drop** the .bin file into Ghidra's project window

Step 6: Configure the Binary Format

A dialog appears. The file is identified as a "BIN" (raw binary without debug symbols).

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 (the XIP address!)
3. Click **OK**

Step 7: Open and Analyze

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 (watch the progress bar in the bottom right).

Part 3.6: Navigating and Resolving Functions**Step 8: Find the Functions**

Look at the **Symbol Tree** panel on the left. Expand **Functions**.

You'll see function names like:

- FUN_1000019a
- FUN_10000210
- FUN_10000238

These are auto-generated names because we imported a raw binary without symbols!

Step 9: Resolve Known Functions

From our previous chapters, we know what some of these functions are:

Ghidra Name	Actual Name	How We Know
FUN_1000019a	data_cpy	From Week 3 boot analysis
FUN_10000210	frame_dummy	From Week 3 boot analysis
FUN_10000238	main	This is where our code is!

Step 10: Update Main's Signature

For main, let's also fix the return type:

1. Right-click on main in the Decompile window
 2. Select **Edit Function Signature**
 3. Change to: `int main(void)`
 4. Click **OK**
-

Part 3.7: Analyzing the Main Function

Step 11: Examine Main in Ghidra

Click on main (or FUN_10000234). Look at the **Decompile** window:

You'll see something like:

```
int main(void)
{
    undefined4 uVar1;
    undefined4 uVar2;
    undefined4 extraout_r1;
    undefined4 uVar3;
    undefined4 extraout_r1_00;

    uVar2 = DAT_10000258;
    uVar1 = DAT_10000254;
    FUN_10002f64();
    uVar3 = extraout_r1;
    do {
        FUN_100030f4(DAT_10000250,uVar3,uVar1,uVar2);
        uVar3 = extraout_r1_00;
    } while( true );
}
```

Step 12: Resolve stdio_init_all

1. Click on FUN_10002f64
2. Right-click -> **Edit Function Signature**
3. Change to: `bool stdio_init_all(void)`
4. Click **OK**

Step 13: Resolve printf

1. Click on FUN_100030f4
2. Right-click -> **Edit Function Signature**
3. Change to: `int printf(char *format,...)`
4. Check the **Varargs** checkbox (printf takes variable arguments!)
5. Click **OK**

Step 14: Understand the Double Encoding

Look at the decompiled code after resolving functions:

```

int main(void)
{
    undefined4 uVar1;
    undefined4 uVar2;
    undefined4 extraout_r1;
    undefined4 uVar3;
    undefined4 extraout_r1_00;

    uVar2 = DAT_10000258;
    uVar1 = DAT_10000254;
    stdio_init_all();
    uVar3 = extraout_r1;
    do {
        printf(DAT_10000250,uVar3,uVar1,uVar2);
        uVar3 = extraout_r1_00;
    } while( true );
}

```

Where's double fav_num = 42.52525? It's been optimized into immediate values!

This time we see **two** non-zero values: 0x645a1cac and 0x4045433b. Unlike the float example where the low word was 0x0, a double with a fractional part like 42.52525 needs **all 52 mantissa bits** - so both halves carry data.

A 64-bit double is passed in two 32-bit registers:

Register	Value	Role
r2	0x645A1CAC	Low 32 bits
r3	0x4045433B	High 32 bits

Together they form 0x4045433B645A1CAC - the IEEE 754 **double-precision** encoding of 42.52525.

Key Difference from Float: In the float example, r2 was 0x00000000 because 42.5 has a clean fractional part. But 42.52525 has a repeating binary fraction, so the low 32 bits are non-zero (0x645A1CAC). This means **both** registers matter when patching doubles with complex fractional values!

Step 15: Verify the Double Encoding

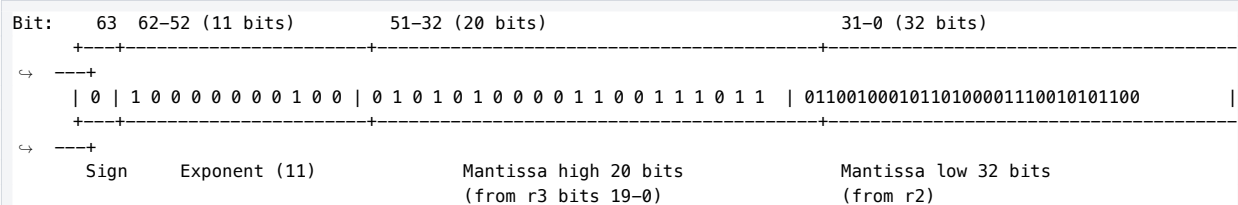
We need to decode 0x4045433B645A1CAC field by field. The two registers give us the full 64-bit value:

```

r3 (high 32 bits): 0x4045433B = 0100 0000 0100 0101 0100 0011 0011 1011
r2 (low 32 bits): 0x645A1CAC = 0110 0100 0101 1010 0001 1100 1010 1100

```

Laid out as a single 64-bit value with every bit numbered:



Step-by-step field extraction:

1. Sign bit

The sign bit is bit 63 of the 64-bit double, which is bit 31 of r3 (the high register holds bits 63-32):

```

r3 = 0x4045433B = 0100 0000 0100 0101 0100 0011 0011 1011
                    ^
                    r3 bit 31 = 0 -> sign = 0 -> Positive number ✓

```

2. Exponent - bits 62-52 = bits 30-20 of r3

Extract bits 30-20 from 0x4045433B:

```
0x4045433B in binary: 0      10000000100      01010100001100111011
                      sign   exponent      mantissa (top 20 bits)
```

Exponent bits: 10000000100

Convert to decimal: $2^{10} + 2^2 = 1024 + 4 = 1028$

Subtract the bias (same formula as Part 2 - the bias is 1023 for all doubles):

$$\text{real exponent} = 1028 - 1023 = 5$$

This means the mantissa gets shifted left by 5 binary places (i.e. multiplied by $2^5 = 32$).

3. Mantissa - bits 51-0

Unlike the 42.5 example where r2 was all zeros, **both registers contribute non-zero bits** here:

- **High 20 bits of mantissa** (bits 51-32) = bits 19-0 of r3:

```
r3 bits 19-0: 0 1 0 1 0 1 0 0 0 0 1 1 0 0 1 1 1 0 1 1
```

- **Low 32 bits of mantissa** (bits 31-0) = all of r2:

```
r2 = 0x645A1CAC -> 0 1 1 0 0 1 0 0 0 1 0 1 1 0 1 0 0 0 0 1 1 1 0 0 1 0 1 0 1 1 0 0
```

Full 52-bit mantissa:

```
0 1 0 1 0 1 0 0 0 0 1 1 0 0 1 1 1 0 1 1 | 0 1 1 0 0 1 0 0 0 1 0 1 1 0 1 0 0 0 0 1 1 1 0 0 1 0 1 0 1 1 0 0
      <- top 20 bits from r3 ->                <- bottom 32 bits from r2 ->
```

IEEE 754 always prepends an **implied leading 1**, so the actual value represented is:

```
1.0101010000110011101100100010110100001110010101100 (the 1. is implicit, not stored)
```

4. Reconstruct the value

$$1.0101010000110011101101100100... (\text{base } 2) \times 2^5$$

Shift the binary point 5 places right:

$$101010.10000110011101101100100010110100001110010101100 (\text{base } 2)$$

Integer part (101010):

Bit position	Power of 2	Value
1 (bit 5)	2^5	32
0 (bit 4)	2^4	0
1 (bit 3)	2^3	8
0 (bit 2)	2^2	0
1 (bit 1)	2^1	2
0 (bit 0)	2^0	0

$$32 + 8 + 2 = 42$$

Fractional part (.10000110011101101...):

Bit position	Power of 2	Decimal value
1 (bit -1)	2^{-1}	0.5
0 (bit -2)	2^{-2}	0
0 (bit -3)	2^{-3}	0
0 (bit -4)	2^{-4}	0
0 (bit -5)	2^{-5}	0
1 (bit -6)	2^{-6}	0.015625
1 (bit -7)	2^{-7}	0.0078125
0 (bit -8)	2^{-8}	0
0 (bit -9)	2^{-9}	0
1 (bit -10)	2^{-10}	0.0009765625
1 (bit -11)	2^{-11}	0.00048828125
1 (bit -12)	2^{-12}	0.000244140625
...	...	(remaining 35 bits add smaller and smaller fractions)

First 12 fractional bits sum: $0.5 + 0.015625 + 0.0078125 + 0.0009765625 + 0.00048828125 + 0.000244140625 \approx 0.5251$

The remaining 35 fractional bits refine this to ≈ 0.52525 . This is because 0.52525 is a **repeating fraction** in binary - it can never be represented with a finite number of bits, so double precision stores the closest possible 52-bit approximation.

$$42 + 0.52525 = 42.52525\checkmark$$

Step 16: Examine the Assembly

Look at the **Listing** window (assembly view). Find the main function:

		***** * FUNCTION *****	
	int	int __stdcall main (void) r0:4 <RETURN>	
	main+1		XREF[1,1]: 1000018c (c) , 1000018a (*)
	main		
10000238 38 b5	push	{r3,r4,r5,lr}	
1000023a 06 a5	adr	r5,[0x10000254]	
1000023c d5 e9 00 45	ldrd	r4,r5,[r5,#0x0]=>DAT_10000254	= 645A1CACH = 4045433Bh
10000240 02 f0 90 fe	bl	stdio_init_all	bool stdio_init_all(void)
	LAB_10000244		XREF[1]: 1000024e (j)
10000244 22 46	mov	r2,r4	
10000246 2b 46	mov	r3,r5	
10000248 01 48	ldr	r0=>s_fav_num:_%lf_100034b0 ,[DAT_10000250]	= "fav_num: %lf\r\n" = 100034B0h
1000024a 02 f0 53 ff	bl	printf	int printf(char * format, ...)
1000024e f9 e7	b	LAB_10000244	
	DAT_10000250		XREF[1]: main:10000248 (R)
10000250 b0 34 00 10	undefine	100034B0h	? -> 100034b0
	DAT_10000254		XREF[1]: main:1000023c (R)
10000254 ac 1c 5a 64	undefine	645A1CACH	
	DAT_10000258		XREF[1]: main:1000023c (R)
10000258 3b 43 45 40	undefine	4045433Bh	

Key Insight: Notice that **both** r2 and r3 are loaded from data constants using `ldr`. Compare this to the float example where r2 was loaded with `mov.w r2, #0x0`. Because 42.52525 requires all 52 mantissa bits, neither word can be zero - the compiler must store both halves as separate data constants.

Step 17: Find the Format String

In the Listing view, click on the data reference to find the format string:

```

100034b0 66 61 76      s_fav_num: %lf_100034b0      XREF[1]:      main:10000248 (*)
          5f 6e 75      ds      "fav_num: %lf\r\n"
          6d 3a 20

```

This confirms `printf` is called with the format string `"fav_num: %lf\r\n"` and the double-precision value of 42.52525.

Part 3.8: Patching the Double - Changing 42.52525 to 99.99

Step 18: Calculate the New IEEE 754 Encoding

We want to change 42.52525 to 99.99. First, we need to figure out the double-precision encoding of 99.99:

1. $99.99 = 1.5623... \times 2^6 = 1.100011111111... \text{ (base 2)} \times 2^6$
2. **Sign:** 0 (positive)
3. **Exponent:** $6 + 1023 = 1029 = 10000000101 \text{ (base 2)}$
4. **Mantissa:** `100011111010111000010100011110101110000101000111...` (base 2)
5. **Full double:** `0x4058FF5C28F5C28F`

Register	Old Value	New Value
r2	0x645A1CAC	0x28F5C28F
r3	0x4045433B	0x4058FF5C

Unlike the float example, **both** registers change! The value 99.99 has a repeating binary fraction, so both the high and low words are different.

Step 19: Find the Values to Patch

Look in the Listing view for the two data constants:

Low word (loaded into r2):

```
10000254 ac 1c 5a 64      undefined4      645A1CACH
```

High word (loaded into r3):

```
10000258 3b 43 45 40      undefined4      4045433Bh
```

Step 20: Patch Both Constants

Patch the low word:

1. Click on the data at address 10000254 containing 645A1CAC
2. Open the Bytes window and enable byte editing (Pencil icon)
3. Overwrite bytes `ac 1c 5a 64` with `8f c2 f5 28` (little-endian for 0x645A1CAC -> 0x28F5C28F)
4. Press Enter

Patch the high word:

1. Click on the data at address 10000258 containing 4045433B

2. Keep byte editing enabled in the Bytes window
3. Overwrite bytes 3b 43 45 40 with 5c ff 58 40 (little-endian for 0x4045433B -> 0x4058FF5C)
4. Press Enter

This changes the full 64-bit double from 0x4045433B645A1CAC (42.52525) to 0x4058FF5C28F5C28F (99.99).

Key Difference from Float Patching: When we patched the float 42.5, we only needed to change one word (the high word in r3) because the low word was all zeros. With 42.52525 -> 99.99, **both** words change. Always check whether the low word is non-zero before patching!

Part 3.9: Export and Test the Hacked Binary

Step 21: 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 0x0011_double-floating-point-data-type-h.bin
5. Click **OK**

Step 22: Convert to UF2 Format

Open a terminal and navigate to your project directory:

```
cd C:\Users\flare-vm\Desktop\Embedded-Hacking-main\0x0011_double-floating-point-data-type
```

Run the conversion command:

```
python ../uf2conv.py build\0x0011_double-floating-point-data-type-h.bin --base 0x10000000 --family 0xe48bff59
↵ --output build\hacked.uf2
```

Step 23: Flash the Hacked Binary

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

You should see:

```
fav_num: 99.990000
fav_num: 99.990000
fav_num: 99.990000
...
```

BOOM! We hacked the double! The value changed from 42.52525 to 99.99!

Part 3.95: Summary - Float and Double Analysis

What We Accomplished

1. **Learned about IEEE 754** - How floating-point numbers are encoded in 32-bit (float) and 64-bit (double) formats
2. **Discovered float-to-double promotion** - printf with %f always receives a double, even when you pass a float
3. **Decoded register pairs** - 64-bit doubles are split across r2 (low) and r3 (high)
4. **Patched a float value** - Changed 42.5 to 99.0 by modifying only the high word
5. **Patched a double value** - Changed 42.52525 to 99.99 by modifying **both** words

6. **Understood the key difference** - Clean fractions (like 42.5) have a zero low word; complex fractions (like 42.52525) require patching both words

IEEE 754 Quick Reference for Common Values

Value	Double Hex	High Word (r3)	Low Word (r2)
42.0	0x4045000000000000	0x40450000	0x00000000
42.5	0x4045400000000000	0x40454000	0x00000000
42.52525	0x4045433B645A1CAC	0x4045433B	0x645A1CAC
43.0	0x4045800000000000	0x40458000	0x00000000
99.0	0x4058C00000000000	0x4058C000	0x00000000
99.99	0x4058FF5C28F5C28F	0x4058FF5C	0x28F5C28F
100.0	0x4059000000000000	0x40590000	0x00000000
3.14	0x40091EB851EB851F	0x40091EB8	0x51EB851F

The Float/Double Patching Workflow

1. Identify the float/double value in the decompiled view
 - Look for hex constants like 0x40454000 or 0x4045433B
2. Determine if it's float (32-bit) or double (64-bit)
 - printf promotes floats to doubles!
 - Check if value spans r2:r3 (double) or just r0 (float)
3. Check if the low word (r2) is zero or non-zero
 - Zero low word = only patch the high word
 - Non-zero low word = patch BOTH words
4. Calculate the new IEEE 754 encoding
 - Convert your desired value to IEEE 754
 - Split into high/low words
5. Patch the constant(s) in Ghidra
 - Edit bytes in the Bytes window (Pencil mode)
 - Replace the old encoding with the new one
6. Export -> Convert to UF2 -> Flash -> Verify
 - Same workflow as integer patching

Tip: Key takeaway: Hacking doubles is the same process as hacking floats - find the IEEE 754 constant, calculate the new encoding, patch it. The only extra step is checking whether the **low word** (r2) is also non-zero. Clean values like 42.5 only need one patch; messy fractions like 42.52525 need two!

Key Takeaways

- Integers have fixed sizes** - uint8_t is 1 byte (0-255), int8_t is 1 byte (-128 to 127). The u prefix means unsigned.
- IEEE 754 encodes floats in binary** - Sign bit, exponent (with bias), and mantissa form the encoding for both 32-bit floats and 64-bit doubles.

3. **printf promotes floats to doubles** - Even when you pass a float, printf receives a 64-bit double due to C's variadic function rules.
 4. **64-bit values span two registers** - On ARM Cortex-M33, doubles use r2 (low 32 bits) and r3 (high 32 bits).
 5. **Clean fractions have zero low words** - Values like 42.5 have 0x00000000 in the low word; complex fractions like 42.52525 have non-zero low words.
 6. **Inline assembly controls hardware directly** - The mcrp coprocessor instruction talks to the GPIO block without any SDK overhead.
 7. **Binary patching works on any data type** - Integers, floats, and doubles can all be patched in Ghidra using the same workflow.
-

Glossary

Term	Definition
Bias	Constant added to the exponent in IEEE 754 (127 for float, 1023 for double)
Double	64-bit floating-point type following IEEE 754 double-precision format
Exponent	Part of IEEE 754 encoding that determines the magnitude of the number
Float	32-bit floating-point type following IEEE 754 single-precision format
FUNCSEL	Function Select - register field that assigns a GPIO pin's function (e.g., SIO)
GPIO	General Purpose Input/Output - controllable pins on a microcontroller
IEEE 754	International standard for floating-point arithmetic and binary encoding
Inline Assembly	Assembly code embedded directly within C source using <code>__asm volatile</code>
int8_t	Signed 8-bit integer type (-128 to 127)
IO_BANK0	Register block at 0x40028000 that controls GPIO pin function selection
Mantissa	Fractional part of IEEE 754 encoding (23 bits for float, 52 bits for double)
mcrp	ARM coprocessor register transfer instruction used for GPIO control
PADS_BANK0	Register block at 0x40038000 that controls GPIO pad electrical properties
Promotion	Automatic conversion of a smaller type to a larger type (float -> double)
Register Pair	Two 32-bit registers (r2:r3) used together to hold a 64-bit value
UF2	USB Flashing Format - file format for Pico 2 firmware
uint8_t	Unsigned 8-bit integer type (0 to 255)

Additional Resources

GPIO Coprocessor Reference

The RP2350 GPIO coprocessor instructions:

Instruction	Description
mcrp p0, #4, Rt, Rt2, c0	Set/clear GPIO output
mcrp p0, #4, Rt, Rt2, c4	Set/clear GPIO output enable

RP2350 Memory Map Quick Reference

Address	Description
0x10000000	XIP Flash (code)
0x20000000	SRAM (data)
0x40028000	IO_BANKo (GPIO control)
0x40038000	PADS_BANKo (pad control)
0xd0000000	SIO (single-cycle I/O)

IEEE 754 Encoding Formula

```
+-----+
| Float (32-bit):  [1 sign] [8 exponent] [23 mantissa] |
| Double (64-bit): [1 sign] [11 exponent] [52 mantissa] |
| Value = (-1)^sign * 2^(exponent - bias) * (1 + mantissa) |
| Float bias: 127 |
| Double bias: 1023 |
+-----+
```

Remember: Every binary you encounter in the real world can be analyzed and understood using these same techniques. Whether it's an integer, a float, or a double - it's all just bits waiting to be decoded. Practice makes perfect!

Happy hacking!