EMBEDDED HACKING

Reverse Engineering the Raspberry Pi Pico 2

Week 4: Variables in Embedded Systems: Debugging and Hacking Variables w/ GPIO Output Basics

ARM Cortex-M33 ● GDB ● Ghidra

August 14, 2026

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 what variables are and how they're stored in memory
- Know the difference between initialized, uninitialized, and constant variables
- Use Ghidra to analyze binaries without debug symbols
- Patch binary files to change program behavior permanently
- Control GPIO pins to blink LEDs on the Pico 2
- Convert patched binaries to UF2 format for flashing
- Understand the `.data`, `.bss`, and `.rodata` memory sections

Part 1: Understanding Variables

What is a Variable?

A **variable** is like a labeled box where you can store information. Imagine you have a row of boxes numbered 0 to 9. Each box can hold one item. In programming:

- The **boxes** are memory locations (addresses in SRAM)
- The **items** are the values you store
- The **labels** are the variable names you choose

```
+-------------------------------------------------------------+
|  Memory (SRAM) – Like a row of numbered boxes               |
|                                                             |
|  Box 0    Box 1    Box 2    Box 3    Box 4    ...          |
|  +----+   +----+   +----+   +----+   +----+                 |
|  | 42 |   | 17 |   |  0 |   |255 |   | 99 |                |
|  +----+   +----+   +----+   +----+   +----+                 |
|   age     score    count    max      temp                  |
|                                                             |
+-------------------------------------------------------------+
```

Declaration vs Definition

When working with variables, there are two important concepts:

Concept	What It Does	Example
Declaration	Tells the compiler the name and type	`uint8_t age;`
Definition	Allocates memory for the variable	(happens with declaration)
Initialization	Assigns an initial value	`uint8_t age = 42;`

Important Rule: You must declare a variable BEFORE you use it!

Understanding Data Types

The **data type** tells the compiler how much memory to allocate:

Type	Size	Range	Description
`uint8_t`	1 byte	0 to 255	Unsigned 8-bit integer
`int8_t`	1 byte	-128 to 127	Signed 8-bit integer
`uint16_t`	2 bytes	0 to 65,535	Unsigned 16-bit integer
`int16_t`	2 bytes	-32,768 to 32,767	Signed 16-bit integer
`uint32_t`	4 bytes	0 to 4,294,967,295	Unsigned 32-bit integer
`int32_t`	4 bytes	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer

Anatomy of a Variable Declaration

Let's break down this line of code:

```
uint8_t age = 42;
```

Part	Meaning
`uint8_t`	Data type - unsigned 8-bit integer (1 byte)
`age`	Variable name - how we refer to this storage location
`=`	Assignment operator - puts a value into the variable
`42`	The initial value
`;`	Semicolon - tells compiler the statement is complete

Part 2: Memory Sections - Where Variables Live

The Three Main Sections

When your program is compiled, variables go to different places depending on how they're declared:

```
+-------------------------------------------------------------+
|  .data Section (Flash -> copied to RAM at startup)          |
|  Contains: Initialized global/static variables              |
|  Example: int counter = 42;                                 |
+-------------------------------------------------------------+
|  .bss Section (RAM - zeroed at startup)                     |
|  Contains: Uninitialized global/static variables            |
|  Example: int counter;  (will be 0)                         |
+-------------------------------------------------------------+
|  .rodata Section (Flash - read only)                        |
|  Contains: Constants, string literals                       |
```

```
|  Example: const int MAX = 100;                                  |
|  Example: "hello, world"                                        |
+-----------------------------------------------------------------+
```

What Happens to Uninitialized Variables?

In older C compilers, uninitialized variables could contain "garbage" - random leftover data. But modern compilers (including the Pico SDK) are smarter:

1. Uninitialized global variables go into the `.bss` section
2. The `.bss` section is **NOT stored in the binary** (saves space!)
3. At boot, the startup code uses `memset` to **zero out** all of `.bss`
4. So uninitialized variables are always `0`!

This is why in our code:

```
uint8_t age; // This will be 0, not garbage!
```

Part 3: Understanding GPIO (General Purpose Input/Output)

What is GPIO?

GPIO stands for **General Purpose Input/Output**. These are pins on the microcontroller that you can control with software. Think of them as tiny switches you can turn on and off.

```
+-----------------------------------------------------------------+
|  Raspberry Pi Pico 2                                            |
|                                                                 |
|  GPIO 16 -------> Red LED                                       |
|  GPIO 17 -------> Green LED                                     |
|  GPIO 18 -------> Blue LED                                      |
|  ...                                                            |
|  GPIO 25 -------> Onboard LED                                   |
+-----------------------------------------------------------------+
```

GPIO Functions in the Pico SDK

The Pico SDK provides simple functions to control GPIO pins:

Function	Purpose
`gpio_init(pin)`	Initialize a GPIO pin for use
`gpio_set_dir(pin, direction)`	Set pin as INPUT or OUTPUT
`gpio_put(pin, value)`	Set pin HIGH (1) or LOW (0)
`sleep_ms(ms)`	Wait for specified milliseconds

What Happens Behind the Scenes?

Each high-level function calls lower-level code. Let's trace `gpio_init()`:

```
gpio_init(LED_PIN)
    ↓
gpio_set_dir(LED_PIN, GPIO_IN)           // Initially set as input
    ↓
gpio_put(LED_PIN, 0)                     // Set output value to 0
    ↓
gpio_set_function(LED_PIN, GPIO_FUNC_SIO) // Connect to SIO block
```

The SIO (Single-cycle I/O) block is a special hardware unit in the RP2350 that provides fast GPIO control!

Part 4: Setting Up Your Environment

Prerequisites

Before we start, make sure you have:

1. A Raspberry Pi Pico 2 board
2. Ghidra installed (for static analysis)
3. Python installed (for UF2 conversion)
4. The sample projects:
   - `0x0005_intro-to-variables`
   - `0x0008_uninitialized-variables`
5. A serial monitor (PuTTY, minicom, or screen)

Project Structure

```
Embedded-Hacking/
+-- 0x0005_intro-to-variables/
|   +-- build/
|   |   +-- 0x0005_intro-to-variables.uf2
|   |   +-- 0x0005_intro-to-variables.bin
|   +-- 0x0005_intro-to-variables.c
+-- 0x0008_uninitialized-variables/
|   +-- build/
|   |   +-- 0x0008_uninitialized-variables.uf2
|   |   +-- 0x0008_uninitialized-variables.bin
|   +-- 0x0008_uninitialized-variables.c
+-- uf2conv.py
```

Part 5: Hands-On Tutorial - Analyzing Variables in Ghidra

Step 1: Review the Source Code

First, let's look at the code we'll be analyzing:

File: `0x0005_intro-to-variables.c`

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

int main(void) {
    uint8_t age = 42;

    age = 43;

    stdio_init_all();

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

What this code does:

1. Declares a variable `age` and initializes it to `42`
2. Changes `age` to `43`
3. Initializes the serial output
4. Prints `age` forever in a loop

Step 2: Flash the Binary to Your Pico 2

1. Hold the BOOTSEL button on your Pico 2
2. Plug in the USB cable (while holding BOOTSEL)
3. Release BOOTSEL - a drive called "RPI-RP2" appears
4. Drag and drop `0x0005_intro-to-variables.uf2` onto the drive
5. The Pico will reboot and start running!

Step 3: Verify It's Working

Open your serial monitor (PuTTY, minicom, or screen) and you should see:

```
age: 43
age: 43
age: 43
...
```

The program is printing `43` because that's what we assigned after the initial `42`.

Part 6: Setting Up Ghidra for Binary Analysis

Step 4: Start Ghidra

Open a terminal and type:

```
ghidraRun
```

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

Step 5: Create a New Project

1. Click **File** -> **New Project**
2. Select **Non-Shared Project**
3. Click **Next**
4. Enter Project Name: `0x0005_intro-to-variables`
5. Click **Finish**

Step 6: Import the Binary

1. Open your file explorer
2. Navigate to the `Embedded-Hacking` folder
3. Find `0x0005_intro-to-variables.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 7: 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 8: 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 7: Navigating and Resolving Functions

Step 9: 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 10: 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 11: 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 8: Analyzing the Main Function

Step 12: Examine Main in Ghidra

Click on `main` (or `FUN_10000234`). Look at the **Decompile** window:
You'll see something like:

```
void FUN_10000234(void)

{
  FUN_10002f54();
  do {
    FUN_100030e4(DAT_10000244,0x2b);
  } while( true );
}
```

Step 13: Resolve stdio_init_all

1. Click on `FUN_10002f54`
2. Right-click -> **Edit Function Signature**
3. Change to: `bool stdio_init_all(void)`
4. Click **OK**

Step 14: Resolve printf

1. Click on `FUN_100030e4`
2. Right-click -> **Edit Function Signature**
3. Change the name to `void printf (undefined4 param_1, ...)`
4. Check the **Varargs** checkbox (printf takes variable arguments!)
5. Click **OK**

Step 15: Understand the Optimization

Look at the updated decompiled code. This will look different if you resolved your functions however do you notice something interesting?

```
int main(void)

{
  stdio_init_all();
  do {
    printf(DAT_10000244,0x2b);
  } while( true );
}
```

Where's `uint8_t age = 42`? It's gone!
The compiler **optimized it out**! Here's what happened:

1. Original code: `age = 42`, then `age = 43`
2. Compiler sees: "The `42` is never used, only `43` matters"
3. Compiler removes the unused `42` and just uses `43` directly

What is `0x2b`? Let's check:

- `0x2b` in hexadecimal = `43` in decimal

The compiler replaced our variable with the constant value!

Part 9: Patching the Binary - Changing the Value

Step 16: Find the Value to Patch

Look at the **Listing** window (assembly view). Find the instruction that loads `0x2b`:

```
1000023a    2b 21    movs r1,#0x2b
```

This instruction loads the value `0x2b` (43) into register `r1` before calling `printf`.

Step 17: Patch the Instruction

We're going to change `0x2b` (43) to `0x46` (70)!

1. At address `1000023a`, click the instruction `movs r1,#0x2b`
2. Right-click and select **Patch Instruction**
3. Replace immediate `0x2b` with `0x46`
4. Press Enter and verify the instruction bytes change from `2b 21` to `46 21`

The instruction now reads:

```
1000023a    46 21    movs r1,#0x46
```

Step 18: 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 `0x0005_intro-to-variables-h.bin`
5. Click **OK**

Part 10: Converting and Flashing the Hacked Binary

Step 19: Convert to UF2 Format

The Pico 2 expects UF2 files, not raw BIN files. We need to convert it!

Open a terminal and navigate to your project directory:

```
cd C:\Users\flare-vm\Desktop\Embedded-Hacking-main\0x0005_intro-to-variables
```

Run the conversion command:

```
python ..\uf2conv.py build\0x0005_intro-to-variables-h.bin --base 0x10000000 --family 0xe48bff59 --output
↪  build\hacked.uf2
```

What this command means:

- `uf2conv.py` = the conversion script
- `--base 0x10000000` = the XIP base address
- `--family 0xe48bff59` = the RP2350 family ID
- `--output build\hacked.uf2` = the output filename

Step 20: 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:

```
age: 70
age: 70
age: 70
...
```

BOOM! We hacked it! The value changed from 43 to 70!

Part 11: Uninitialized Variables and GPIO

Now let's work with a more complex example that includes GPIO control.

Step 21: Review the Uninitialized Variables Code

File: `0x0008_uninitialized-variables.c`

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

```c
#define LED_PIN 16

int main(void) {
    uint8_t age; // Uninitialized!

    stdio_init_all();

    gpio_init(LED_PIN);
    gpio_set_dir(LED_PIN, GPIO_OUT);

    while (true) {
        printf("age: %d\r\n", age);

        gpio_put(LED_PIN, 1);
        sleep_ms(500);

        gpio_put(LED_PIN, 0);
        sleep_ms(500);
    }
}
```

What this code does:

1. Declares `age` without initializing it (will be 0 due to BSS zeroing)
2. Initializes GPIO 16 as an output
3. In a loop: prints age, blinks the LED

Step 22: Flash and Verify

1. Flash `0x0008_uninitialized-variables.uf2` to your Pico 2
2. Open your serial monitor

You should see:

```
age: 0
age: 0
age: 0
...
```

And the **red LED on GPIO 16 should be blinking**!
The value is `0` because uninitialized variables in the `.bss` section are zeroed at startup.

Part 12: Analyzing GPIO Code in Ghidra

Step 23: Set Up Ghidra for the New Binary

1. Create a new project: `0x0008_uninitialized-variables`
2. Import `0x0008_uninitialized-variables.bin`
3. Set Language to **ARM Cortex 32 little endian**
4. Set Base Address to `.text` and `10000000`
5. Auto-analyze

Step 24: Resolve the Functions

Find and rename these functions:

Ghidra Name	Actual Name
`FUN_10000234`	`main`
`FUN_100030cc`	`stdio_init_all`

Ghidra Name	Actual Name
FUN_100002b4	gpio_init
FUN_1000325c	printf

For `gpio_init`, set the signature to:

```
void gpio_init(uint gpio)
```

Step 25: Examine the Main Function

The decompiled main should look something like:

```
void FUN_10000234(void)

{
  undefined4 extraout_r1;
  undefined4 extraout_r2;
  undefined4 in_cr0;
  undefined4 in_cr4;

  FUN_100030cc();
  FUN_100002b4(0x10);
  coprocessor_moveto2(0,4,0x10,1,in_cr4);
  do {
    FUN_1000325c(DAT_10000274,0);
    coprocessor_moveto2(0,4,0x10,1,in_cr0);
    FUN_10000d10(500);
    coprocessor_moveto2(0,4,0x10,0,in_cr0);
    FUN_10000d10(500,extraout_r1,extraout_r2,0);
  } while( true );
}
```

Part 13: Hacking GPIO - Changing the LED Pin

Step 26: Find the GPIO Pin Value

Look in the assembly for instructions that use `0x10` (which is 16 in decimal - our LED pin):

```
1000023a    10 20    movs r0,#0x10
```

This is where `gpio_init(LED_PIN)` is called with GPIO 16.

Step 27: Patch GPIO 16 to GPIO 17

We'll change the red LED (GPIO 16) to the green LED (GPIO 17)!

1. At address `1000023a`, select `movs r0,#0x10`
2. Right-click -> **Patch Instruction**
3. Replace immediate `0x10` with `0x11` (17 decimal)
4. Click **OK** and verify bytes change from `10 20` to `11 20`

Step 28: Find All GPIO 16 References

There are more places that use GPIO 16. Look for:

```
10000244    10 23    movs r3,#0x10
```

This is used in `gpio_set_dir`. Patch this to `0x11` as well.

```
10000252    10 24    movs r4,#0x10
```

This is inside the loop for `gpio_put`. Patch this to `0x11` as well. Patch each one with **Patch Instruction**, then verify:

- `10000244: 10 23 -> 11 23`
- `10000252: 10 24 -> 11 24`

Step 29: Bonus - Change the Printed Value

Let's also change the printed value from `0` to `0x42` (66 in decimal):

```
1000024a    00 21    movs r1,#0x0
```

1. Right-click -> **Patch Instruction**
2. Replace immediate `0x0` with `0x42`
3. Click **OK** and verify bytes change from `00 21` to `42 21`

Part 14: Export and Test the Hacked GPIO

Step 30: Export the Patched Binary

1. Click **File** -> **Export Program**
2. Format: **Raw Bytes**
3. Filename: `0x0008_uninitialized-variables-h.bin`
4. Click **OK**

Step 31: Convert to UF2

```
cd C:\Users\flare-vm\Desktop\Embedded-Hacking-main\0x0008_uninitialized-variables
python ..\uf2conv.py build\0x0008_uninitialized-variables-h.bin --base 0x10000000 --family 0xe48bff59 --output
↪  build\hacked.uf2
```

Step 32: Flash and Verify

1. Flash `hacked.uf2` to your Pico 2
2. Check your serial monitor

You should see:

```
age: 66
age: 66
age: 66
...
```

And now the **GREEN LED on GPIO 17** should be blinking instead of the red one!

We successfully:

1. Changed the printed value from 0 to 66
2. Changed which LED blinks from red (GPIO 16) to green (GPIO 17)

Part 15: Deep Dive - GPIO at the Assembly Level

Understanding the GPIO Coprocessor

The RP2350 has a special **GPIO coprocessor** that provides fast, single-cycle GPIO control. This is different from the RP2040!

The coprocessor is accessed using special ARM instructions:

```
mcrr p0, #4, r4, r5, c0    ; GPIO output control
mcrr p0, #4, r4, r5, c4    ; GPIO direction control
```

What this means:

- `mcrr` = Move to Coprocessor from two ARM Registers
- `p0` = Coprocessor 0 (the GPIO coprocessor)
- `r4` = Contains the GPIO pin number
- `r5` = Contains the value (0 or 1)
- `c0` = Output value register
- `c4` = Output enable register

The Full GPIO Initialization Sequence

When you call `gpio_init(16)`, here's what actually happens:

```
Step 1: Configure pad (address 0x40038044)
+-------------------------------------------------------------------+
|  - Clear OD bit (output disable)                                  |
|  - Set IE bit (input enable)                                      |
|  - Clear ISO bit (isolation)                                      |
+-------------------------------------------------------------------+

Step 2: Set function (address 0x40028084)
+-------------------------------------------------------------------+
|  - Set FUNCSEL to 5 (SIO - Software I/O)                          |
+-------------------------------------------------------------------+

Step 3: Enable output (via coprocessor)
+-------------------------------------------------------------------+
|  - mcrr p0, #4, r4, r5, c4  (where r4=16, r5=1)                  |
+-------------------------------------------------------------------+
```

Raw Assembly LED Blink

Here's what a completely hand-written assembly LED blink looks like:

```
; Initialize GPIO 16 as output
movs r4, #0x10          ; GPIO 16
movs r5, #0x01          ; Enable
mcrr p0, #4, r4, r5, c4 ; Set as output

; Configure pad registers
ldr r3, =0x40038044     ; Pad control for GPIO 16
ldr r2, [r3]            ; Load current config
bic r2, r2, #0x80       ; Clear OD (output disable)
orr r2, r2, #0x40       ; Set IE (input enable)
str r2, [r3]            ; Store config

; Set GPIO function to SIO
ldr r3, =0x40028084     ; IO bank control for GPIO 16
movs r2, #5             ; FUNCSEL = SIO
str r2, [r3]            ; Set function

; Main loop
loop:
    ; LED ON
    movs r4, #0x10      ; GPIO 16
    movs r5, #0x01      ; High
    mcrr p0, #4, r4, r5, c0
```

```
    ; Delay
    ldr r2, =0x17D7840  ; ~25 million iterations
delay1:
    subs r2, r2, #1
    bne delay1

    ; LED OFF
    movs r4, #0x10      ; GPIO 16
    movs r5, #0x00      ; Low
    mcrr p0, #4, r4, r5, c0

    ; Delay
    ldr r2, =0x17D7840
delay2:
    subs r2, r2, #1
    bne delay2

    b loop              ; Repeat forever
```

Part 16: Summary and Review

What We Accomplished

1. **Learned about variables** - How they're declared, initialized, and stored
2. **Understood memory sections** - `.data`, `.bss`, and `.rodata`
3. **Analyzed binaries in Ghidra** - Without debug symbols!
4. **Patched binaries** - Changed values directly in the binary
5. **Controlled GPIO** - Made LEDs blink
6. **Changed program behavior** - Different LED, different value

The Binary Patching Workflow

```
+-----------------------------------------------------------------+
|  1. Import .bin file into Ghidra                                |
|     - Set language to ARM Cortex                                |
|     - Set base address to 0x10000000                            |
+-----------------------------------------------------------------+
|  2. Analyze and resolve functions                               |
|     - Rename functions to meaningful names                      |
|     - Fix function signatures                                   |
+-----------------------------------------------------------------+
|  3. Find the values/instructions to patch                       |
|     - Look in the assembly listing                              |
|     - Patch Instruction, then verify old bytes -> new bytes     |
+-----------------------------------------------------------------+
|  4. Export the patched binary                                   |
|     - File -> Export Program                                     |
|     - Format: Raw Bytes                                         |
+-----------------------------------------------------------------+
|  5. Convert to UF2                                              |
|     - python uf2conv.py file.bin --base 0x10000000              |
|       --family 0xe48bff59 --output hacked.uf2                   |
+-----------------------------------------------------------------+
|  6. Flash and verify                                            |
|     - Hold BOOTSEL, plug in, drag UF2                           |
|     - Check serial output and LED behavior                      |
+-----------------------------------------------------------------+
```

Key Memory Sections

Section	Location	Contains	Writable?
`.text`	Flash	Code	No
`.rodata`	Flash	Constants, strings	No
`.data`	RAM	Initialized globals	Yes
`.bss`	RAM	Uninitialized globals (zeroed)	Yes

Important Ghidra Commands

Action	How To Do It
Rename function	Right-click -> Edit Function Signature
Patch instruction	Right-click -> Patch Instruction, then verify old bytes -> new bytes
Export binary	File -> Export Program -> Raw Bytes
Go to address	Press ‘G’ and enter address

Key Takeaways

1. **Variables are just memory locations** - The compiler assigns them addresses in SRAM.
2. **Compilers optimize aggressively** - Unused code and values may be removed entirely.
3. **Uninitialized doesn’t mean random** - Modern compilers zero out the `.bss` section.
4. **Ghidra works without symbols** - You can analyze any binary, even stripped ones.
5. **Binary patching is powerful** - You can change behavior without source code.
6. **UF2 conversion is required** - The Pico 2 needs UF2 format, not raw binaries.
7. **GPIO is just memory-mapped I/O** - Writing to specific addresses controls hardware.

Glossary

Term	Definition
BSS	Block Started by Symbol - section for uninitialized global variables
Declaration	Telling the compiler a variable’s name and type
Definition	Allocating memory for a variable
GPIO	General Purpose Input/Output - controllable pins on a microcontroller
Initialization	Assigning an initial value to a variable
Linker	Tool that combines compiled code and assigns memory addresses
Optimization	Compiler removing or simplifying code for efficiency
Patching	Modifying bytes directly in a binary file
rodata	Read-only data section for constants and string literals
SIO	Single-cycle I/O - fast GPIO control block in RP2350

Term	Definition
UF2	USB Flashing Format - file format for Pico 2 firmware
Variable	A named storage location in memory

Additional Resources

GPIO Coprocessor Reference

The RP2350 GPIO coprocessor instructions:

Instruction	Description
`mcrr p0, #4, Rt, Rt2, c0`	Set/clear GPIO output
`mcrr 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_BANK0 (GPIO control)
`0x40038000`	PADS_BANK0 (pad control)
`0xd0000000`	SIO (single-cycle I/O)

Remember: Every binary you encounter in the real world can be analyzed and understood using these same techniques. Practice makes perfect!

Happy hacking!