FIRST EDITION 1.0001 IN-DEVELOPMENT-ALPHA

Copyright © 2025 Kevin Thomas

Forward

I remember when I started learning programming to which my first language was 6502 Assembler to allow me to program a Commodore 64 and right from the beginning of my journey, I learned the lowest level development possible.

Literally every piece of the Commodore 64 was understood as it was a simple machine. There was absolutely no abstraction layer of any kind.

We had an absolute mastery of everything however it was a very simple architecture.

Microcontrollers are small systems without an operating system and are also very simple in their design. They are literally everywhere from your toaster to your fridge to your TV and billions of other electronics that you never think about.

Most microcontrollers are developed in the C programming language which has its roots to the 1970's however dominates the landscape.

We will take our time and learn the basics of C utilizing a Pico 2 microcontroller.

Below are items you will need for this course.

Raspberry Pi Pico 2
https://www.amazon.com/Raspberry-Pre-Soldered-Cortex-M33-Microcontroller-Development/dp/B0DHN5D45D

Micro USB Data Cable
https://www.amazon.com/Amazon-Basics-Charging-Transfer-Gold-Plated/dp/B07QD4WVLN

Raspberry Pi Pico Debug Probe
https://www.amazon.com/Debug-Probe-Raspberry-Microcontroller-XYGStudy/dp/B0CQJB5FC5

Upgraded Electronics Fun Kit
https://www.amazon.com/ELEGOO-Electronics-Potentiometer-tie-Points-Breadboard/dp/B09YRJQRFF

Soldering Iron Kit
https://www.amazon.com/Liouhoum-Auto-Sleep-Adjustable-Temperature-Thermostatic/dp/B08PZBPXLZ

RYLR998 UART 915 MHz Lora Module w/ Antenna
https://www.amazon.com/REYAX-RYLR998-Interface-Antenna-Transceiver/dp/B099RM1XMG

NOTE: The item links may NOT be available, but the descriptions allow you to shop on any online or physical store of your choosing.

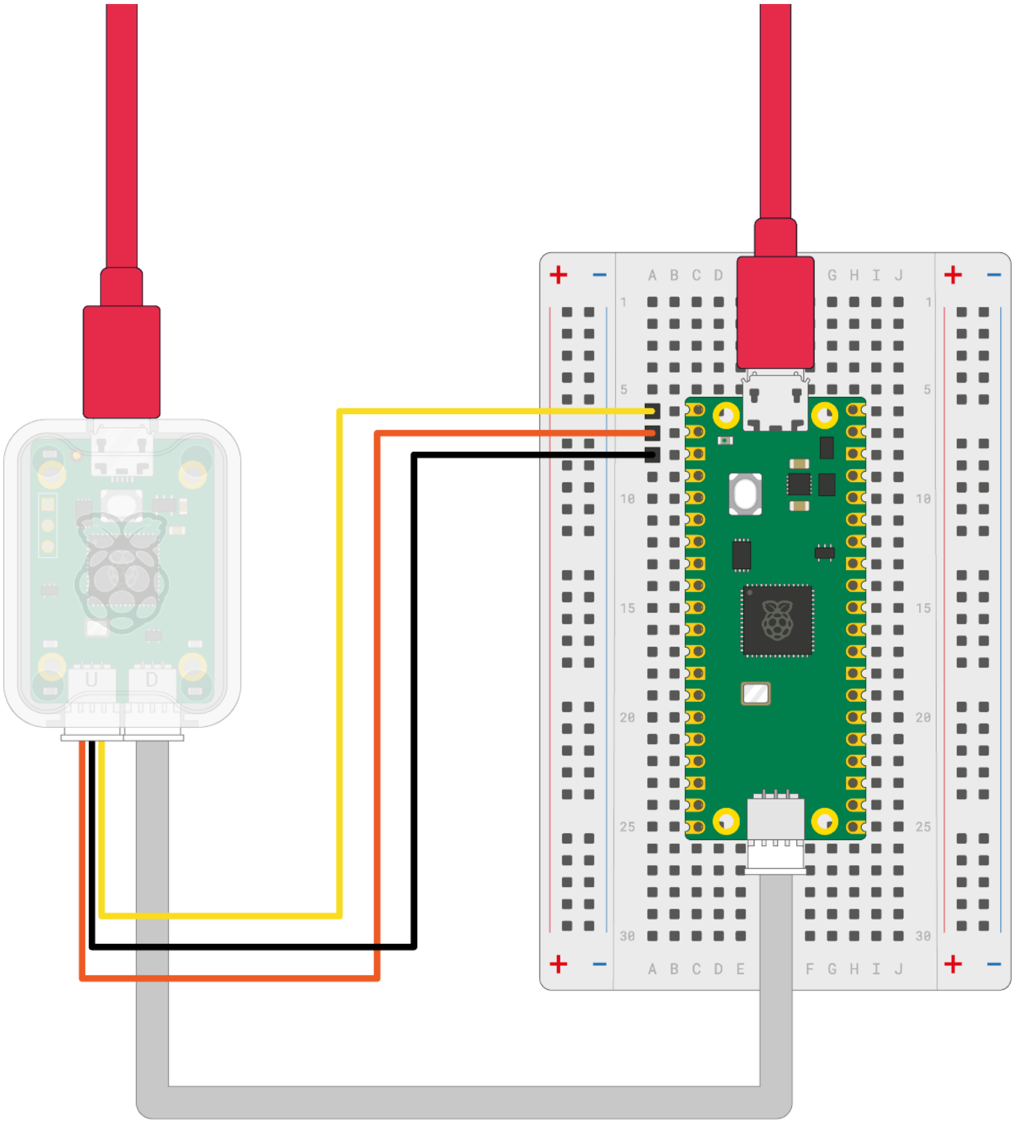
Let’s begin...

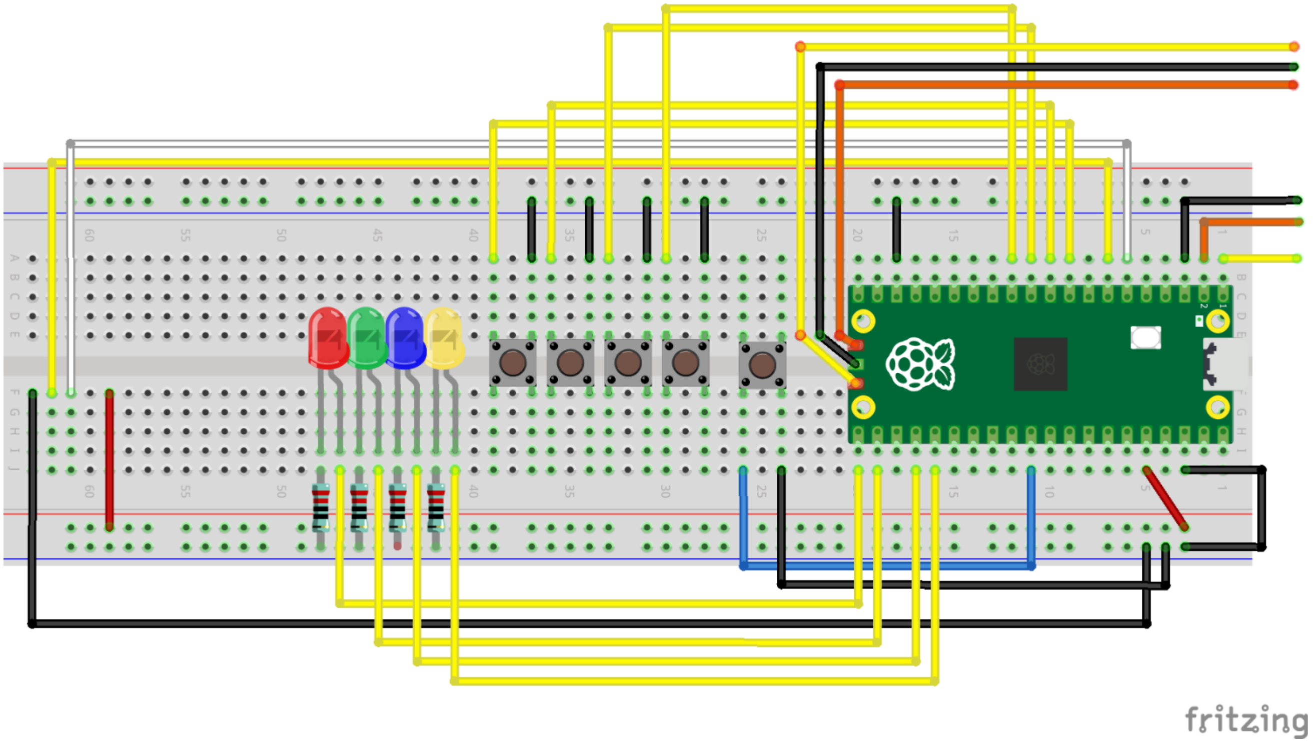
Table Of Contents

Chapter 1: hello, world

We begin our journey building the traditional *hello, world* example in Embedded C.

Below we see our diagrams for the Pico Debug Probe and our breadboard schematic which includes our Pico 2 microcontroller.





To setup our development environment, we will download VS Code.
https://code.visualstudio.com/download

Once VS Code is installed, we will install the Raspberry Pi Pico VS Code extension.
https://marketplace.visualstudio.com/items?itemName=raspberry-pi.raspberry-pi-pico

We will setup the Raspberry Pi Pico Debug Probe as there are detailed instructions below as well to get started.
https://www.raspberrypi.com/documentation/microcontrollers/debug-probe.html

A pinout of the Pico 2 board is linked below as well.
https://www.raspberrypi.com/documentation/microcontrollers/images/pico-2-r4-pinout.svg

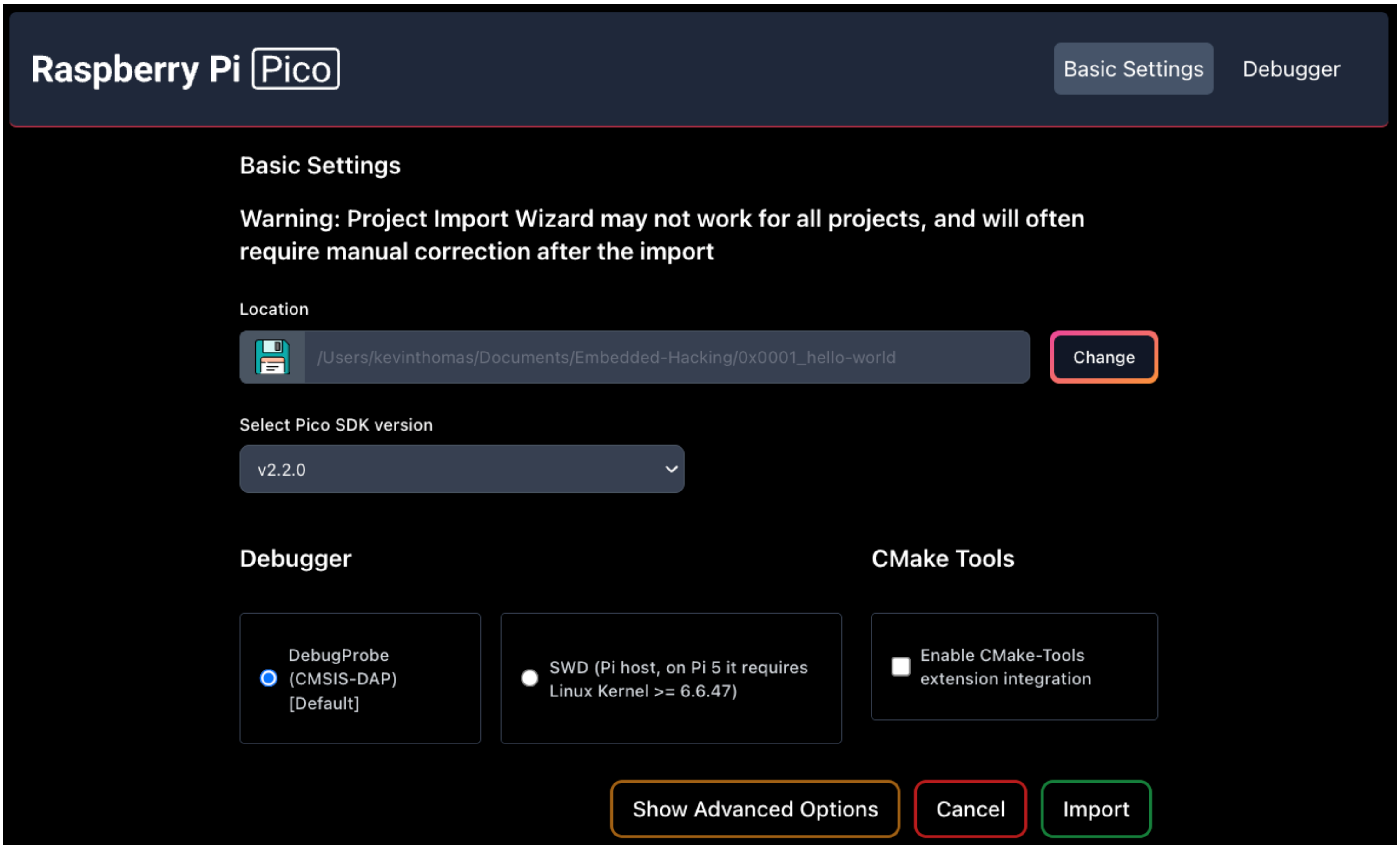
If you do not have Git installed, here is a link to install git on Windows, MAC and Linux.
https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

We need to clone our course repo to whatever folder you prefer.

```
git clone https://github.com/mytechnotalent/Embedded-Hacking.git
```

Open VS Code and click **File** then **Open Folder** then click on the **Embedded-Hacking** folder and then select **0x0001_hello-world**.

This may pop up a screen asking to import the project. Once visible, click **Import**, otherwise just continue.



Basic Settings

Warning: Project Import Wizard may not work for all projects, and will often require manual correction after the import

Location: /Users/kevinthomas/Documents/Embedded-Hacking/0x0001_hello-world

Select Pico SDK version: v2.2.0

Debugger

● DebugProbe (CMSIS-DAP) [Default]

○ SWD (Pi host, on Pi 5 it requires Linux Kernel >= 6.6.47)

CMake Tools

□ Enable CMake-Tools extension integration

Now we are ready to compile and flash our code onto the Pico.

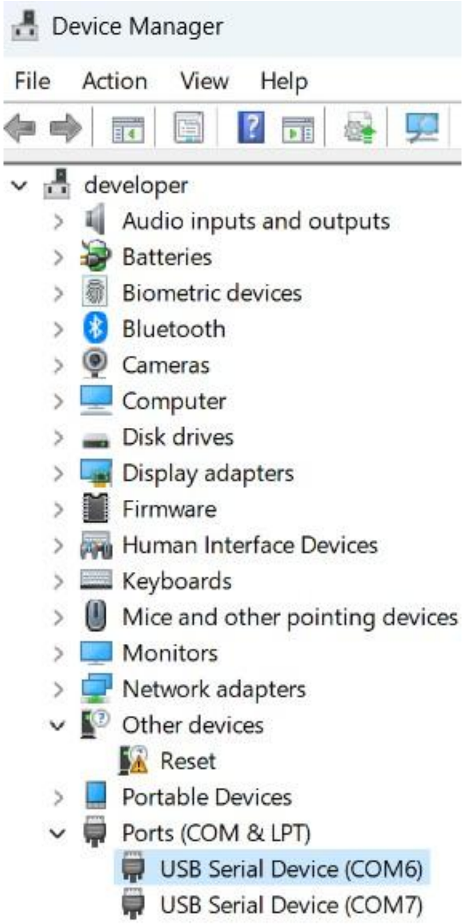
You can click on **Compile** and then **Run** in the bottom right-hand side of the VS Code editor assuming you have your Pico 2 plugged in.

Press and hold the push button we attached to the breadboard while pressing the white BOOSEL button on the Pico 2; then release the white BOOTSEL button on the Pico 2 and then release the push button we attached to the breadboard.

If the **Compile** and **Run** buttons within VS Code does not work, you can also open a file explorer window to copy our **0x0001_hello-world.uf2** firmware into the **RPI-RP2** drive.

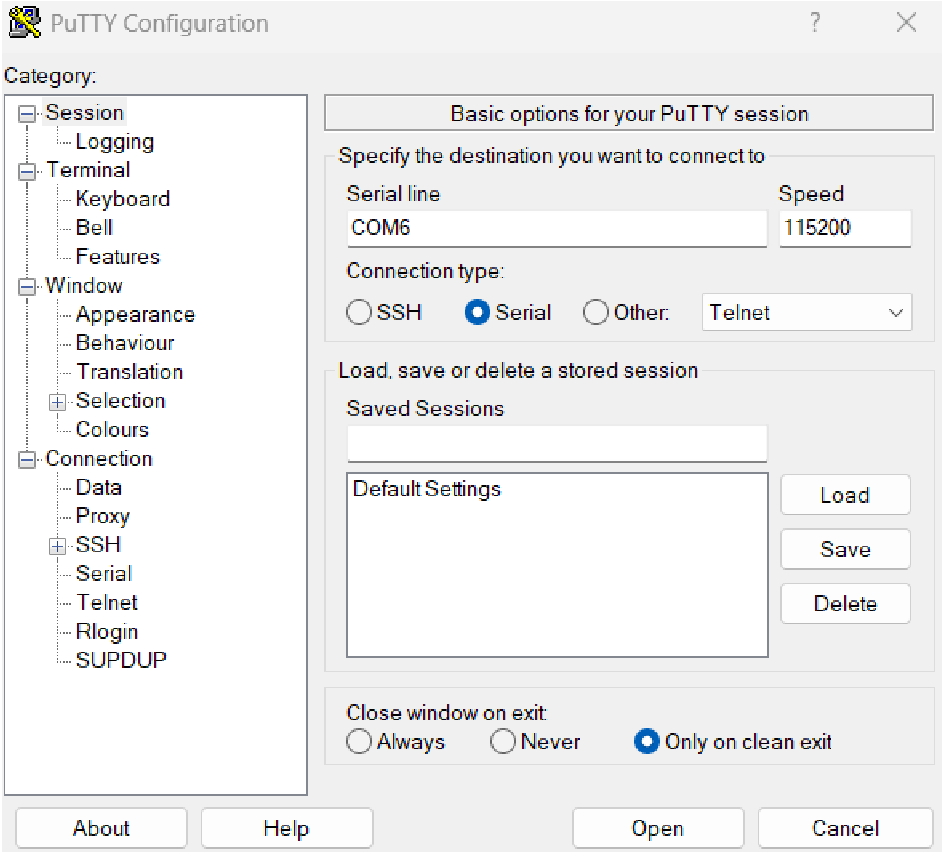
We need to download a serial monitor to interact with our Pico. If you are on Windows download PuTTY as the link is below.
https://www.putty.org

If you are on Windows, you can open the Device Manager and look for the COM port that will be used to connect PuTTY to. There are at minimum two ports one for the Pico 2 UART and the other for the Pico Debug Probe. Try both and one of them will be UART that we are looking for.



```
developer
    Audio inputs and outputs
    Batteries
    Biometric devices
    Bluetooth
    Cameras
    Computer
    Disk drives
    Display adapters
    Firmware
    Human Interface Devices
    Keyboards
    Mice and other pointing devices
    Monitors
    Network adapters
    Other devices
        Reset
    Portable Devices
    Ports (COM & LPT)
        USB Serial Device (COM6)
        USB Serial Device (COM7)
```

The next step is to run PuTTY.



Specify the destination you want to connect to

Serial line: COM6

Speed: 115200

Connection type: ○ SSH ● Serial ○ Other: Telnet

Saved Sessions: Default Settings

Close window on exit: ○ Always ○ Never ● Only on clean exit

You want to type in your COM port, in my case COM6, and click the *Open* button.

If you are on MAC or Linux, you can use the screen program.

```
ls /dev/tty.
screen /dev/tty.XXX 115200
```

Now let's review our **0x0001_hello-world.c** file as this is located within the main folder.

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

int main(void) {
    stdio_init_all();

    while (true)
      printf("hello, world\r\n");
}
```

Let's break down this code.

```
#include <stdio.h>
```

This line includes the `stdio.h` header file, which contains declarations for standard input and output functions.

```
#include "pico/stdlib.h"
```

This line includes the `pico/stdlib.h` header file, which contains declarations for various Raspberry Pi Pico standard library functions.

```
int main(void)
```

The above line declares the main function, which is the entry point for all C and Python programs.

```
stdio_init_all();
```

This line initializes the standard input and output system.

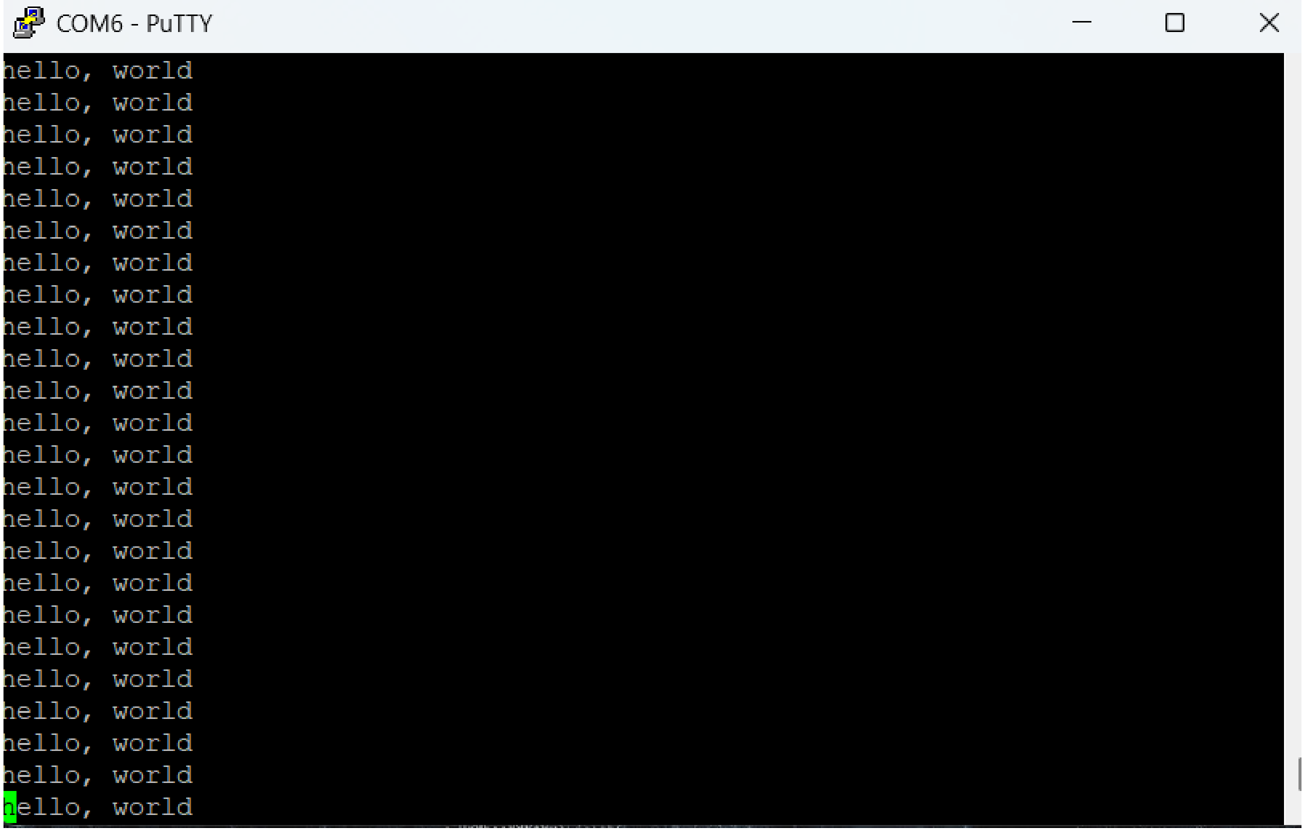
```
while (true)
```

This line starts a while loop that will run forever.

```
printf("hello, world\r\n");
```

This line prints the message, *hello, world*, to the console.

Open the terminal to see, *hello, world*, as expected being printed over and over again.



```
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
```

In our next lesson we will debug *hello, world* using Ghidra and the ARM embedded GDB with OpenOCD to which we will connect LIVE to our running Pico 2!

Chapter 2: Debugging hello, world

Today we debug!

There are two main types of reverse engineering: static and dynamic. Static reverse engineering involves examining the binary without executing it. Tools like Ghidra allow you to inspect raw assembly instructions, control flow, and code structure. Dynamic reverse engineering, on the other hand, involves running the binary and observing its behavior in real time. With tools like GDB, you can monitor memory changes, register values, and execution paths as the program runs.

We will download Ghidra, a free static disassembler from the NSA at the link below.
https://github.com/NationalSecurityAgency/ghidra/releases

If you are using Windows, we will move the Ghidra folder to the **C:\** drive and make sure to update the path accordingly. If you are on MAC or Linux, move to the root of your drive as well and update version in path.

```
C:\ghidra_11.4.2_PUBLIC
```

Please download and install the proper Java version based on your system.
https://adoptium.net/temurin/releases

Once complete, a file called **ghidraRun** will be created. To launch Ghidra, execute this file. If you're on Windows, be sure to run the batch file version.

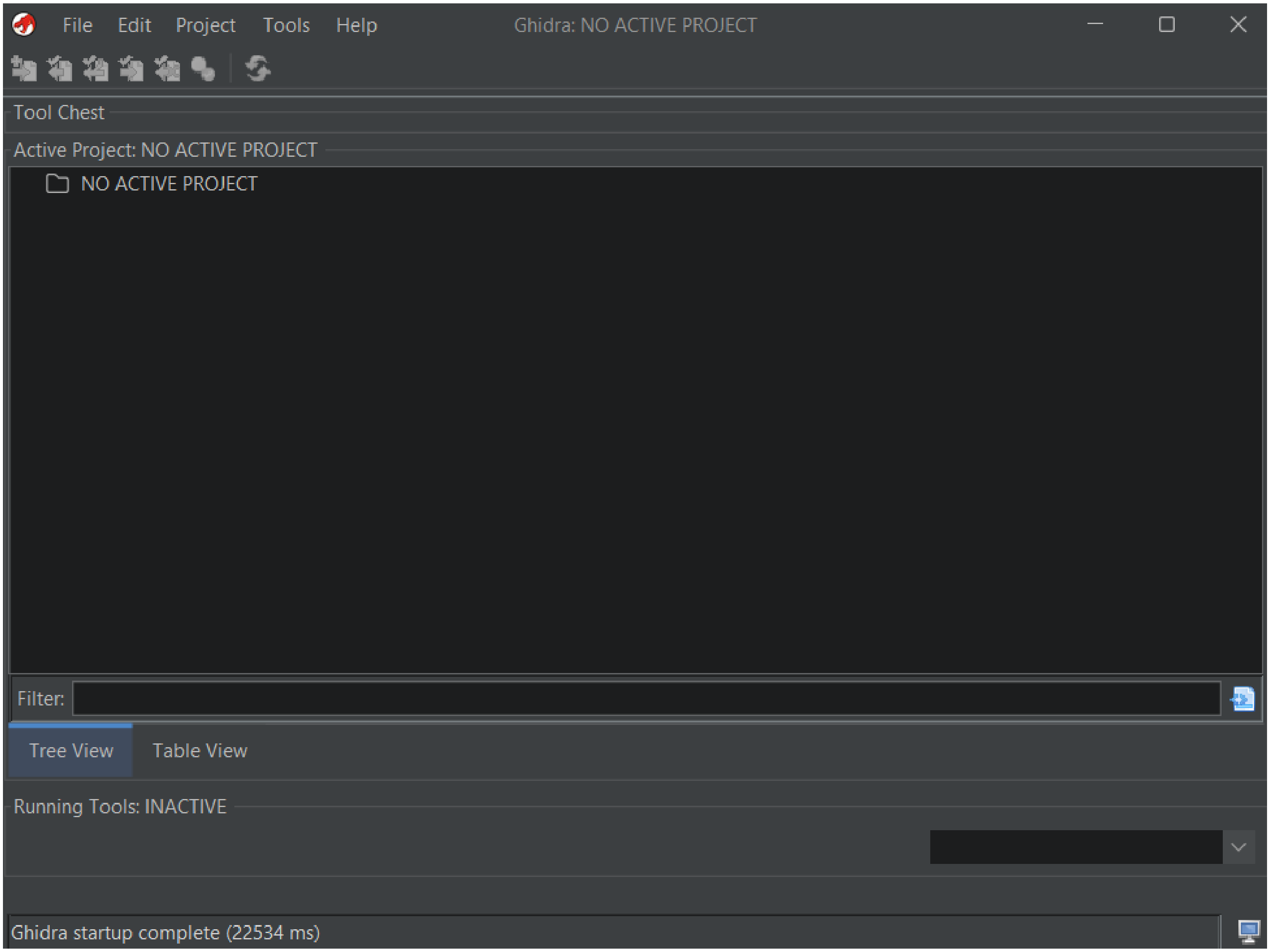
A window will appear where we will select **File**, **New Project**, **Non-Shared Project**, **Next**, and create a **Project Name**. Here we will call it **0x0001_hello-world** and press **Finish**.

Open the file explorer and navigate to the **Embedded-Hacking** folder and drag-and-drop the **0x0001_hello-world.elf** file into the folder within the Ghidra application panel.

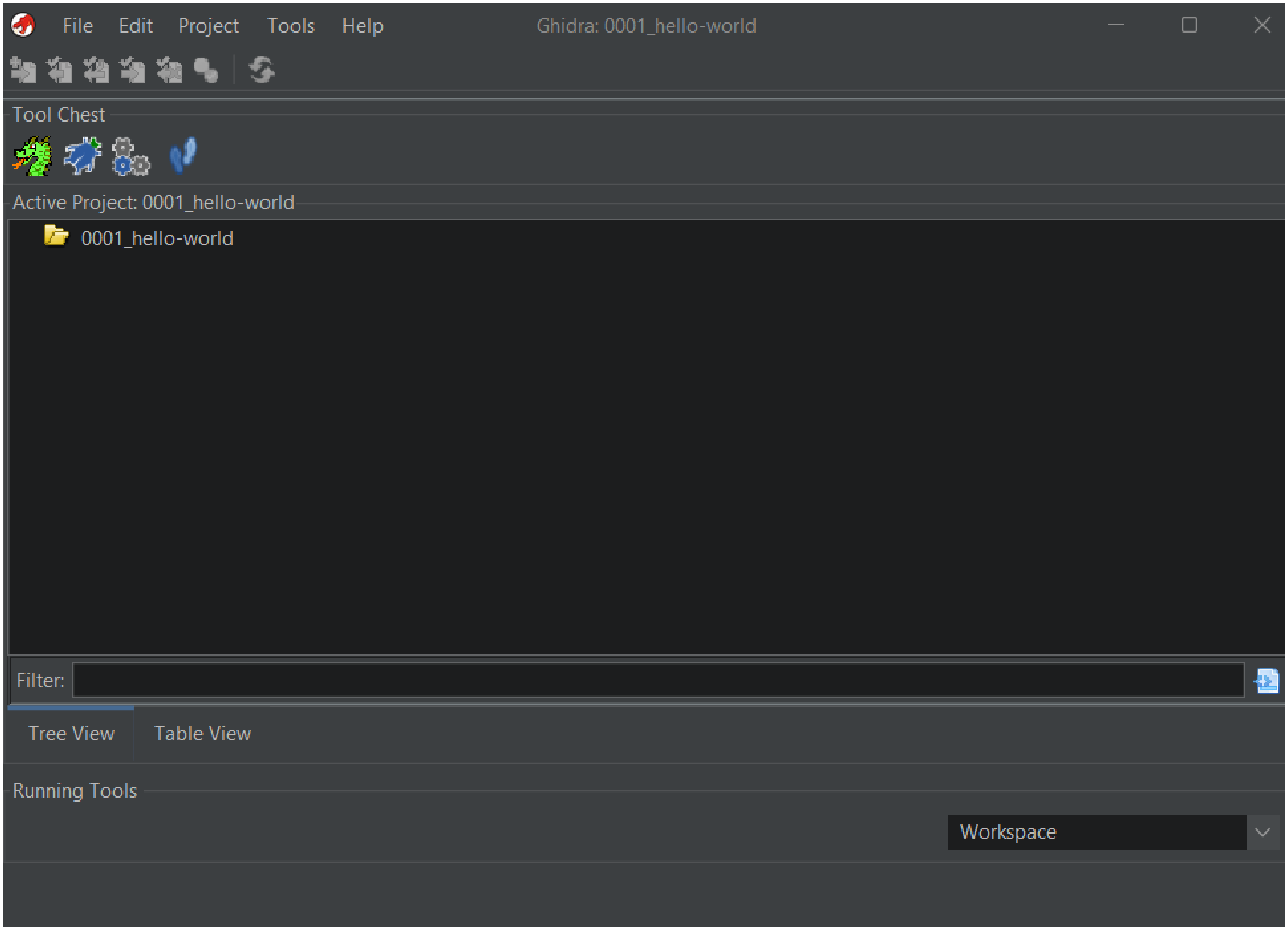
In the small window that appears, you will see the file identified as an ELF, which stands for Executable and Linkable Format. This format includes symbols that make reverse engineering easier. In future chapters, we will work with stripped binaries that do not contain these symbols.

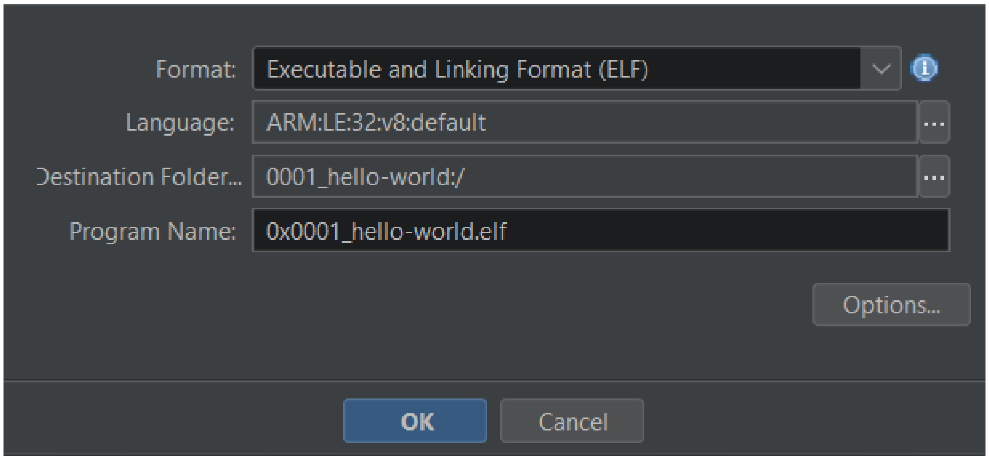
At this point, click **Ok** and then double-click on the file within the window.

Finally click the auto-analyze and let’s begin reviewing the binary.



Ghidra startup complete (22534 ms)

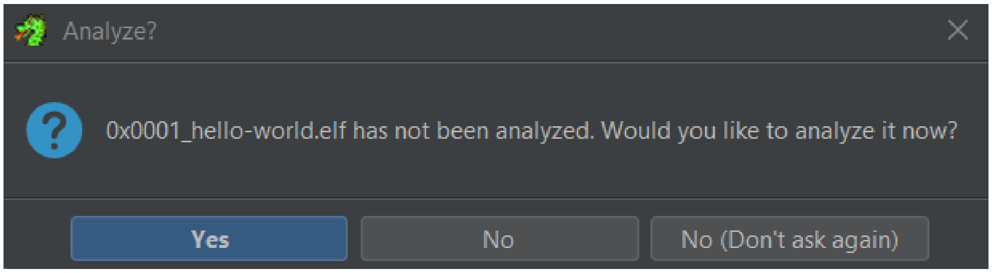



Format: Executable and Linking Format (ELF)

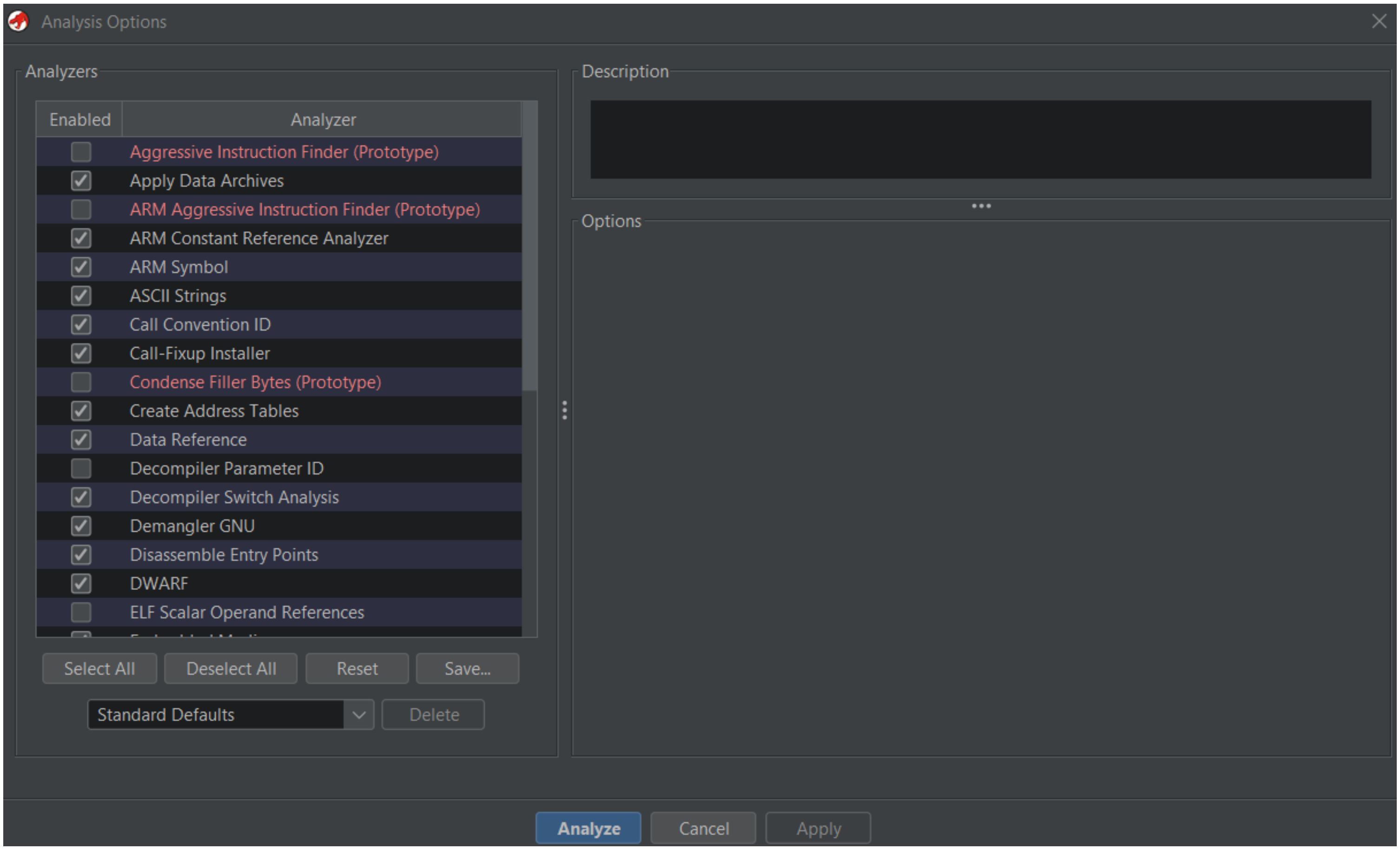
Language: ARM:LE:32:v8:default

Destination Folder... 0001_hello-world:/

Program Name: 0x0001_hello-world.elf

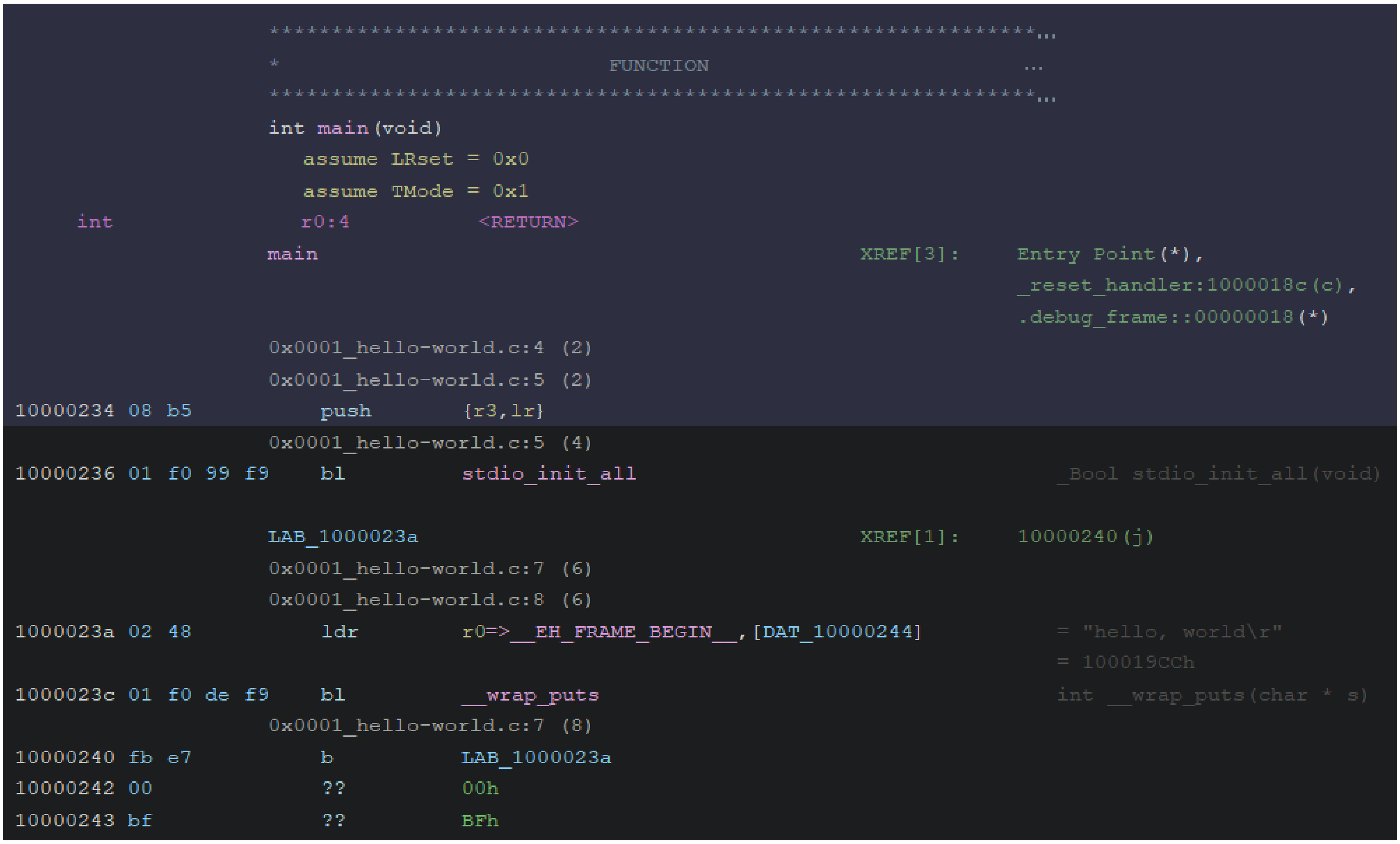


0x0001_hello-world.elf has not been analyzed. Would you like to analyze it now?

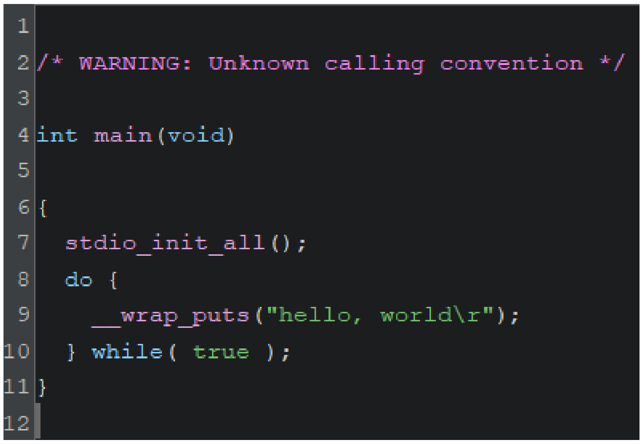


Enabled	Analyzer
□	Aggressive Instruction Finder (Prototype)
☑	Apply Data Archives
□	ARM Aggressive Instruction Finder (Prototype)
☑	ARM Constant Reference Analyzer
☑	ARM Symbol
☑	ASCII Strings
☑	Call Convention ID
☑	Call-Fixup Installer
□	Condense Filler Bytes (Prototype)
☑	Create Address Tables
☑	Data Reference
□	Decompiler Parameter ID
☑	Decompiler Switch Analysis
☑	Demangler GNU
☑	Disassemble Entry Points
☑	DWARF
□	ELF Scalar Operand References

Standard Defaults



```
                     **************************************************************...
                     *                          FUNCTION                          ...
                     **************************************************************...
                     int main(void)
                        assume LRset = 0x0
                        assume TMode = 0x1
     int                r0:4           <RETURN>
                     main                                          XREF[3]:     Entry Point(*),
                                                                                _reset_handler:1000018c(c),
                                                                                .debug_frame::00000018(*)
                     0x0001_hello-world.c:4 (2)
                     0x0001_hello-world.c:5 (2)
10000234 08 b5           push       {r3,lr}
                     0x0001_hello-world.c:5 (4)
10000236 01 f0 99 f9     bl         stdio_init_all                                 _Bool stdio_init_all(void)

                     LAB_1000023a                                  XREF[1]:     10000240(j)
                     0x0001_hello-world.c:7 (6)
                     0x0001_hello-world.c:8 (6)
1000023a 02 48           ldr        r0=>__EH_FRAME_BEGIN__,[DAT_10000244]          = "hello, world\r"
                                                                                   = 100019CCh
1000023c 01 f0 de f9     bl         __wrap_puts                                    int __wrap_puts(char * s)
                     0x0001_hello-world.c:7 (8)
10000240 fb e7           b          LAB_1000023a
10000242 00              ??         00h
10000243 bf              ??         BFh
```



```

/* WARNING: Unknown calling convention */

int main(void)

{
  stdio_init_all();
  do {
    __wrap_puts("hello, world\r");
  } while( true );
}

```

I have held off on exploring the deeper meaning behind all of this because our first goal is to establish a solid static reverse engineering workflow.

Now we can see our main function displayed in raw assembly, a decompiled view, and a pseudo source code window.

One of the first differences we notice is that our original source used a while true loop, but the decompiled output shows a do while loop. This is not a major issue, as the logic is still clear and we can see the code echoing *hello, world* to the terminal.

In our original source, we used the printf function. After compilation, the compiler optimized this and replaced it with the puts function, which is a common substitution for simple output.

At this point, I am going to pause on reviewing the assembly and shift focus to setting up GDB. This will allow us to begin dynamic reverse engineering, along with a basic introduction to the ARM architecture we are working with.

To enable dynamic reverse engineering capabilities, we will download the GNU ARM toolchain tailored to our embedded architecture. Be sure to select the version appropriate for your system.
https://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads

The next step is to download OpenOCD. If you are on Windows, there are pre-build binaries at the location below.
https://gnutoolchains.com/arm-eabi/openocd

If you are on Windows, the next step is to extract the folder to your **C:\** drive and update your path to include the following directories and keep in mind the version you downloaded as you may need to adjust the path.

```
C:\OpenOCD-20250710-0.12.0\bin
C:\OpenOCD-20250710-0.12.0\share\openocd\scripts\interface
C:\OpenOCD-20250710-0.12.0\share\openocd\scripts\target
```

For MAC, we first install Homebrew and the various dependencies and OpenOCD.

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
brew install git libtool automake pkg-config libusb
brew install openocd
```

For Linux, we install the various dependencies and OpenOCD.

```
sudo apt update
sudo apt install git build-essential libtool autoconf pkg-config libusb-1.0-0-dev
libftdi1-dev
sudo apt install openocd
```

Run OpenOCD with the below config.

```
openocd -f interface/cmsis-dap.cfg -f target/rp2350.cfg -c "adapter speed 5000"
```

Open a new terminal and then run the following to launch our dynamic debugger called GDB.

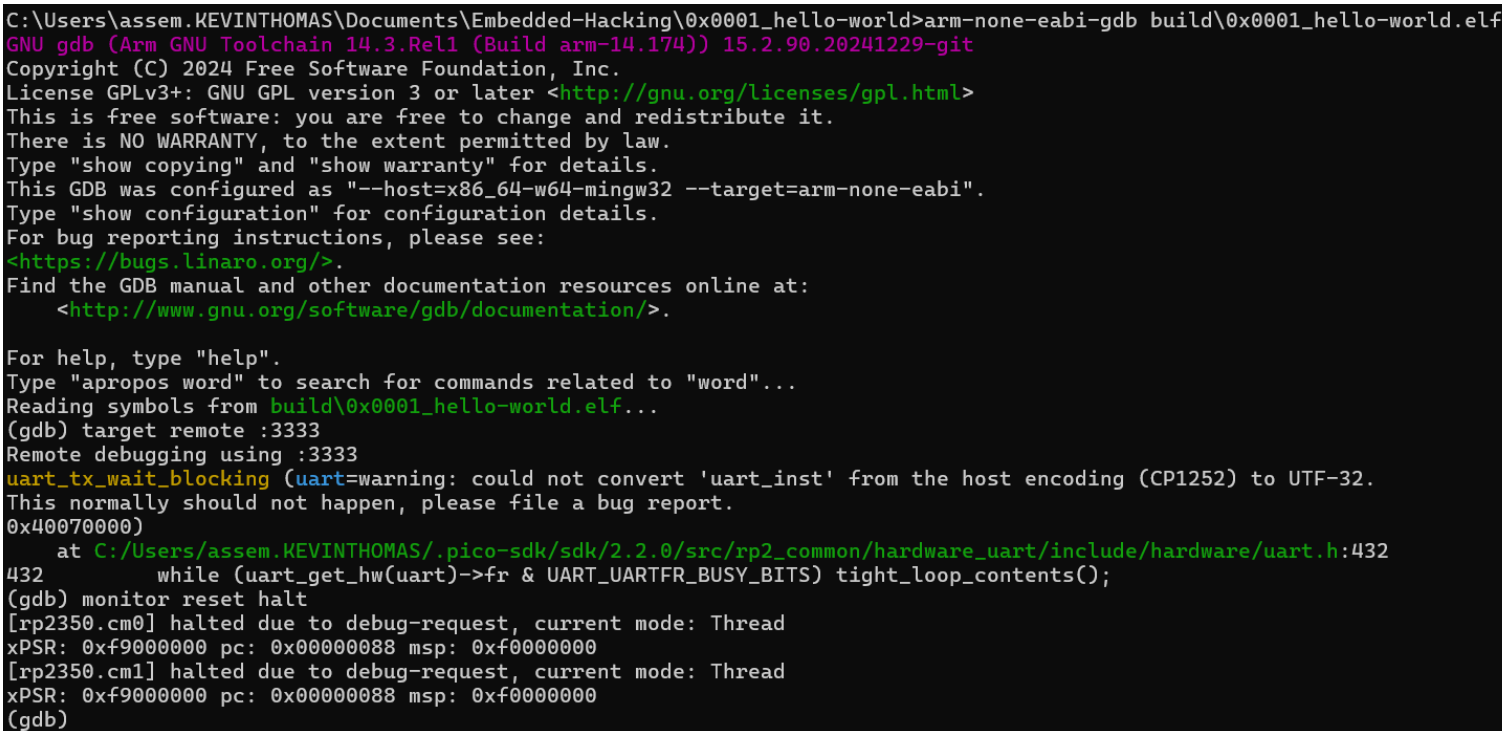
```
arm-none-eabi-gdb build/0x0001_hello-world.elf
```

Once it loads, we need to target our remote server.

```
target remote :3333
```

We need to halt the currently running binary.

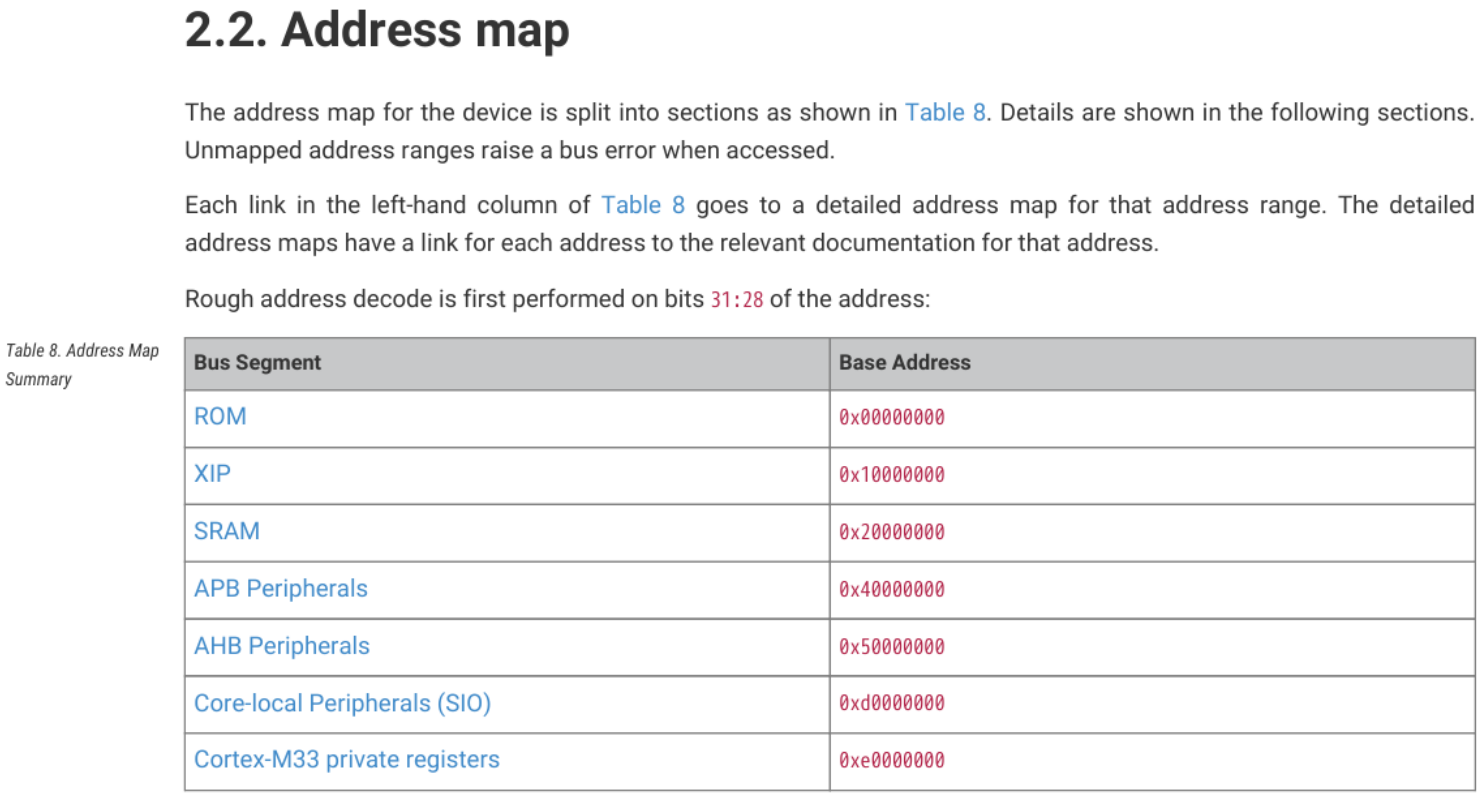
```
monitor reset halt
```



```
C:\Users\assem.KEVINTHOMAS\Documents\Embedded-Hacking\0x0001_hello-world>arm-none-eabi-gdb build\0x0001_hello-world.elf
GNU gdb (Arm GNU Toolchain 14.3.Rel1 (Build arm-14.174)) 15.2.90.20241229-git
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-w64-mingw32 --target=arm-none-eabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://bugs.linaro.org/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from build\0x0001_hello-world.elf...
(gdb) target remote :3333
Remote debugging using :3333
uart_tx_wait_blocking (uart=warning: could not convert 'uart_inst' from the host encoding (CP1252) to UTF-32.
This normally should not happen, please file a bug report.
0x40070000)
    at C:/Users/assem.KEVINTHOMAS/.pico-sdk/sdk/2.2.0/src/rp2_common/hardware_uart/include/hardware/uart.h:432
432         while (uart_get_hw(uart)->fr & UART_UARTFR_BUSY_BITS) tight_loop_contents();
(gdb) monitor reset halt
[rp2350.cm0] halted due to debug-request, current mode: Thread
xPSR: 0xf9000000 pc: 0x00000088 msp: 0xf0000000
[rp2350.cm1] halted due to debug-request, current mode: Thread
xPSR: 0xf9000000 pc: 0x00000088 msp: 0xf0000000
(gdb)
```

Before we go any further, we need to turn to the RP2350 datasheet.

https://datasheets.raspberrypi.com/rp2350/rp2350-datasheet.pdf



2.2. Address map

The address map for the device is split into sections as shown in Table 8. Details are shown in the following sections. Unmapped address ranges raise a bus error when accessed.

Each link in the left-hand column of Table 8 goes to a detailed address map for that address range. The detailed address maps have a link for each address to the relevant documentation for that address.

Rough address decode is first performed on bits `31:28` of the address:

Table 8. Address Map Summary

Bus Segment	Base Address
ROM	`0x00000000`
XIP	`0x10000000`
SRAM	`0x20000000`
APB Peripherals	`0x40000000`
AHB Peripherals	`0x50000000`
Core-local Peripherals (SIO)	`0xd0000000`
Cortex-M33 private registers	`0xe0000000`

Above is page 30 where we see our address map.

XIP, a technique where firmware instructions are executed directly from non-volatile memory rather than being copied into RAM.

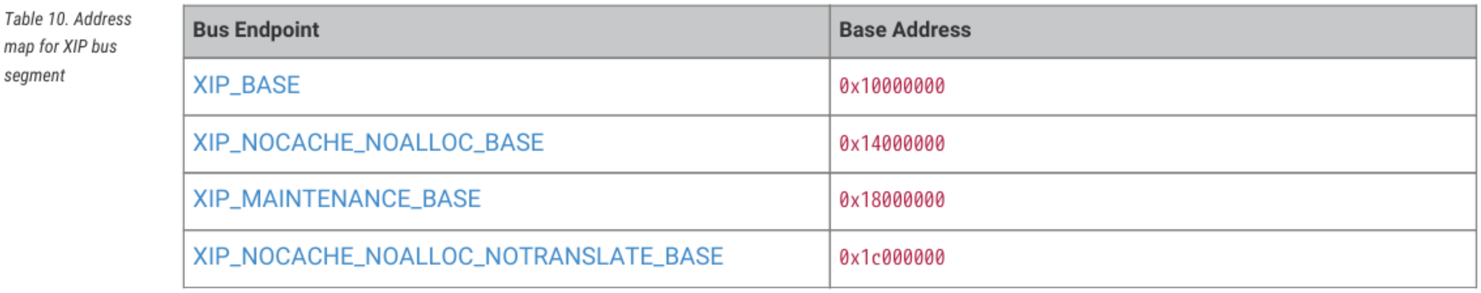


Table 10. Address map for XIP bus segment

Bus Endpoint	Base Address
XIP_BASE	`0x10000000`
XIP_NOCACHE_NOALLOC_BASE	`0x14000000`
XIP_MAINTENANCE_BASE	`0x18000000`
XIP_NOCACHE_NOALLOC_NOTRANSLATE_BASE	`0x1c000000`

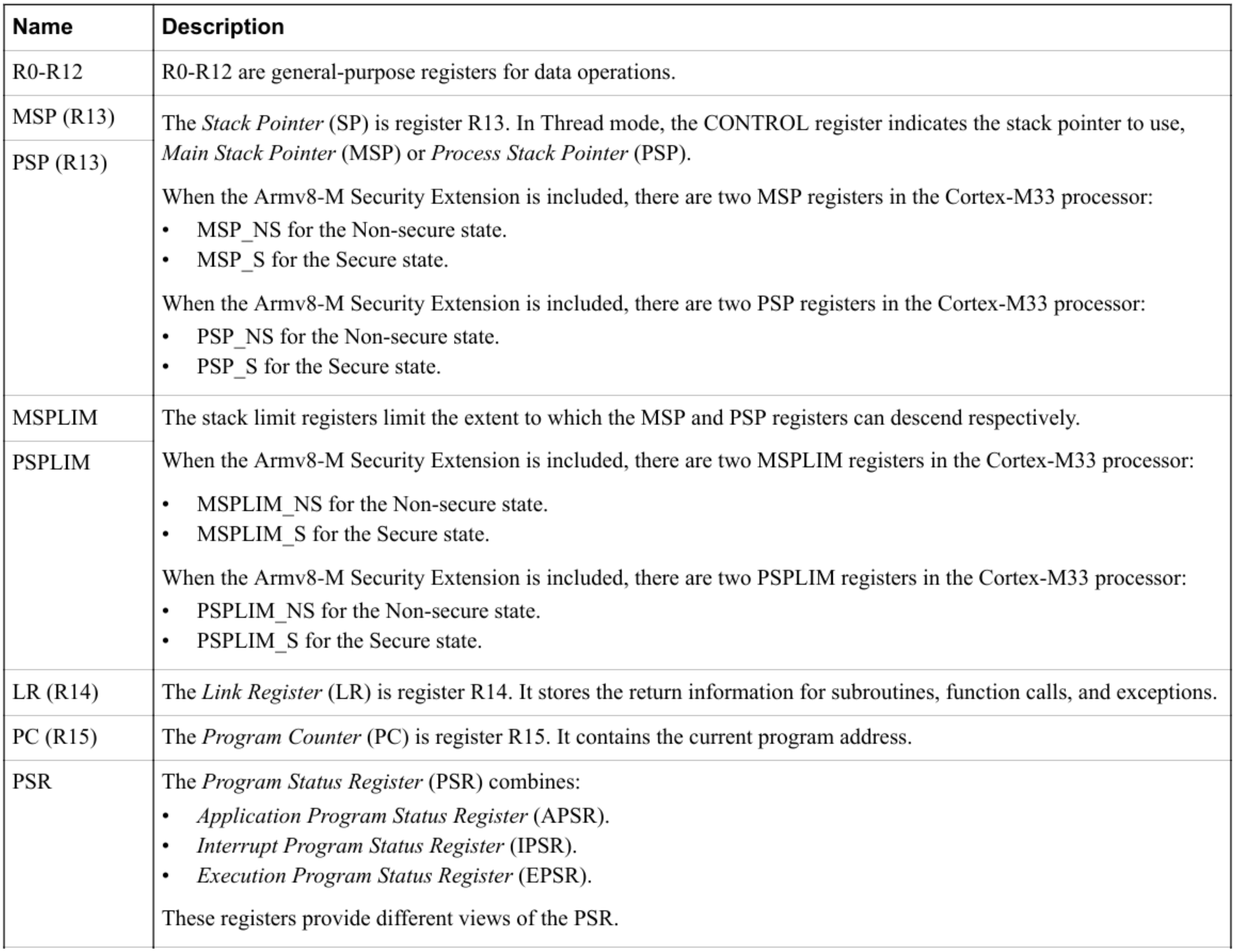
At address `0x10000000`, is where we will focus within GDB.

Before we dive into the assembler, we need to understand we are working with an RP2350 microcontroller that has a dual-core architecture.

This course will not focus on the RISC-V core however will focus on the ARM Cortex-M33 core as this is more prevalent in the industry today however a future course may cover the RISC-V core.

The ARM Cortex-M33 core is part of what we refer to as the Armv8-M Mainline family.

We will review the Arm Cortex-M33 Processor Technical Reference Manual that is included in the course Github repo.



Name	Description
R0-R12	R0-R12 are general-purpose registers for data operations.
MSP (R13)	The *Stack Pointer* (SP) is register R13. In Thread mode, the CONTROL register indicates the stack pointer to use, *Main Stack Pointer* (MSP) or *Process Stack Pointer* (PSP). When the Armv8-M Security Extension is included, there are two MSP registers in the Cortex-M33 processor: • MSP_NS for the Non-secure state. • MSP_S for the Secure state. When the Armv8-M Security Extension is included, there are two PSP registers in the Cortex-M33 processor: • PSP_NS for the Non-secure state. • PSP_S for the Secure state.
PSP (R13)	
MSPLIM	The stack limit registers limit the extent to which the MSP and PSP registers can descend respectively. When the Armv8-M Security Extension is included, there are two MSPLIM registers in the Cortex-M33 processor: • MSPLIM_NS for the Non-secure state. • MSPLIM_S for the Secure state. When the Armv8-M Security Extension is included, there are two PSPLIM registers in the Cortex-M33 processor: • PSPLIM_NS for the Non-secure state. • PSPLIM_S for the Secure state.
PSPLIM	
LR (R14)	The *Link Register* (LR) is register R14. It stores the return information for subroutines, function calls, and exceptions.
PC (R15)	The *Program Counter* (PC) is register R15. It contains the current program address.
PSR	The *Program Status Register* (PSR) combines: • *Application Program Status Register* (APSR). • *Interrupt Program Status Register* (IPSR). • *Execution Program Status Register* (EPSR). These registers provide different views of the PSR.

On page B1-40, we see the above processor core register summary.

Our microcontroller has 13 general-purpose 32-bit wide registers called `r0-r12`. These registers will be used for storing intermediate values, passing function arguments, and performing arithmetic or logical operations during program execution. They form the core working set for most instructions and are essential for efficient data manipulation and control flow within the processor.

The r13 register is called the stack pointer. The stack pointer holds the address of the top of the stack—a region of memory used for temporary storage during function calls. When a function is called, local variables, return addresses, and saved register states are pushed onto the stack. As the function exits, these values are popped off. The stack grows downward in memory on ARM Cortex-M systems, and the sp ensures that data is stored and retrieved in the correct order. It’s critical for managing nested function calls and interrupt handling.

The r14 register is called the link register. The link register stores the return address when a function or subroutine is called. In ARM assembly, instructions like bl (Branch with Link) automatically place the address of the next instruction into lr so the processor knows where to return after the function finishes. If lr is overwritten or mishandled, the program may jump to an unintended location, leading to crashes or undefined behavior. In exception handling, lr also plays a role in determining the return path after servicing an interrupt.

The r15 register is called the program counter. The program counter holds the address of the next instruction to be executed. It’s automatically updated as the processor steps through instructions, and can be manually modified during jumps, branches, or exceptions. The pc is central to control flow—whether you're executing sequential code, branching conditionally, or handling interrupts. In debugging or reverse engineering, tracking the pc helps you understand exactly where the processor is in its execution lifecycle.

We need to touch base on what XIP is within the RP2350 MCU microcontroller. This is the actual chip that powers the Pico 2.

As mentioned earlier, XIP is called, execute in place, and is capable of directly executing code from non-volatile storage (such as flash memory) without the need to copy the code to random-access memory (RAM) first. Instead of loading the entire program into RAM, XIP systems fetch instructions directly from their storage location and execute them on the fly.

Our goal is to find the main function within our binary to reverse engineer it. Before our main function there will be a large amount of setup code to include the vector table which will handle hardware interrupts and exceptions within our firmware which will be at the address close to the beginning of 0x10000000.

Our XIP address starts at 0x10000000 so let’s examine 1000 instructions and look for a push {r3, lr} followed by a call to stdio_init_all which would indicate our main stack frame being called.

```
(gdb) x/1000i 0x10000000
...
0x10000234 <main>:   push    {r3, lr}
...
```

This is our main program. If you are new to assembler, do not be discouraged as we will take this step-by-step!

To begin working effectively with the RP2350, it is important to understand how memory is organized within the microcontroller. The RP2350 features a dual-core ARM Cortex-M33 processor, which introduces more advanced memory management capabilities compared to earlier architectures. We start by examining the stack and heap, as these are essential concepts in embedded systems.

The stack is a region of memory used to manage function calls and local variables. It automatically grows and shrinks as functions are called and return. Each time a function is invoked, a stack frame is created to store its local variables and the return address. The stack pointer register keeps track of the current position in the stack and is updated automatically during function calls and returns.

Because the RP2350 has two cores, each core maintains its own dedicated stack. The size of each stack is typically defined in the linker script or project configuration and is constrained by the available RAM. When data is added to the stack, such as function parameters, it is referred to as a push operation. When data is removed, such as return values or saved registers, it is called a pop operation.

If the stack grows beyond its allocated space, it can result in a stack overflow. This may cause unpredictable behavior or system crashes. In contrast, the heap is a region of memory used for dynamic allocation. It is managed manually by the programmer, who must explicitly allocate and free memory as needed.

Dynamic memory allocation is performed using functions such as malloc in C or new in C++. This approach is useful for handling data structures whose size may vary during runtime. The heap in the RP2350 is typically located in the RAM region. Its size is flexible and can be adjusted based on the needs of the application.

Memory on the heap can be allocated to obtain a block of space and deallocated to return it for reuse. Over time, repeated allocation and deallocation can lead to fragmentation, which makes it harder to find large contiguous blocks of memory. The RP2350 uses standard C library functions such as malloc and free to manage heap memory. The size and location of the heap are usually defined in the linker script or project settings.

In this course, we will not necessarily focus on dynamic memory allocation. Instead, we will use safer and more predictable strategies for managing memory. The RP2350 has a limited amount of RAM, so careful planning is essential. Code is stored in Flash memory and is executed directly from that location. Understanding this memory layout is key to building reliable and efficient embedded applications.

Now let's examine our main function.

```
(gdb) x/5i 0x10000234
   0x10000234 <main>:   push    {r3, lr}
   0x10000236 <main+2>: bl      0x1000156c <stdio_init_all>
   0x1000023a <main+6>: ldr     r0, [pc, #8]    @ (0x10000244 <main+16>)
   0x1000023c <main+8>: bl      0x100015fc <__wrap_puts>
   0x10000240 <main+12>:        b.n     0x1000023a <main+6>
```

Let's set a breakpoint to our main function and continue.

```
(gdb) b *0x10000234
Breakpoint 1 at 0x10000234: file C:/Users/assem.KEVINTHOMAS/Documents/Embedded-Hacking/0x0001_hello-world/0x0001_hello-world.c, line 5.
Note: automatically using hardware breakpoints for read-only addresses.
(gdb) c
Continuing.

Thread 1 "rp2350.cm0" hit Breakpoint 1, main ()
    at C:/Users/assem.KEVINTHOMAS/Documents/Embedded-Hacking/0x0001_hello-world/0x0001_hello-world.c:5
warning: Source file is more recent than executable.
5           stdio_init_all();
```

Let's re-examine our main function and we will see an arrow pointing to the instruction we are about to execute. Keep in mind, we have NOT executed it yet.

```
(gdb) x/5i 0x10000234
=> 0x10000234 <main>:   push    {r3, lr}
   0x10000236 <main+2>: bl      0x1000156c <stdio_init_all>
   0x1000023a <main+6>: ldr     r0, [pc, #8]    @ (0x10000244 <main+16>)
   0x1000023c <main+8>: bl      0x100015fc <__wrap_puts>
   0x10000240 <main+12>:        b.n     0x1000023a <main+6>
```

We push the `r3` register and the `lr` register to the stack.

Keep in mind, the base pointer is not a register in the RP2350's ARM Cortex-M33 architecture. Unlike some other architectures such as x86, which use a dedicated base pointer for stack frame management, the Cortex-M33 relies on the stack pointer and the link register to handle function calls and returns.

In this architecture, the stack pointer, also known as `sp` or `r13`, points to the top of the stack and is automatically adjusted as functions are called and return. The link register, referred to as `lr` or `r14`, holds the return address when a function is invoked. These two registers work together to manage the stack and control program flow during subroutine execution.

The concept of a base pointer, as seen in x86/64 systems with the RBP register, is not part of the standard conventions used in the RP2350. Instead, stack frames are managed directly through sp and lr without a separate frame pointer.

It is important to note that in microcontroller environments like the RP2350, the main function typically runs in an infinite loop and does not return. As a result, the value stored in the link register after main begins execution is never used, but it remains part of the standard calling convention.

We have not executed our first main assembler function yet so let’s first examine what our stack contains.

```
(gdb) x/10x $sp
0x20082000:     0x00000000      0x00000000      0x00000000      0x00000000
0x20082010:     0x00000000      0x00000000      0x00000000      0x00000000
0x20082020:     0x00000000      0x00000000
```

Now let’s step-into which means take a single step in assembler.

```
(gdb) si
0x10000236      5           stdio_init_all();
(gdb) x/5i 0x10000234
   0x10000234 <main>:   push    {r3, lr}
=> 0x10000236 <main+2>: bl      0x1000156c <stdio_init_all>
   0x1000023a <main+6>: ldr     r0, [pc, #8]    @ (0x10000244 <main+16>)
   0x1000023c <main+8>: bl      0x100015fc <__wrap_puts>
   0x10000240 <main+12>:        b.n     0x1000023a <main+6>
```

Let’s review our stack.

```
(gdb) x/10x $sp
0x20081ff8:     0xe000ed08      0x1000018f      0x00000000      0x00000000
0x20082008:     0x00000000      0x00000000      0x00000000      0x00000000
0x20082018:     0x00000000      0x00000000
```

We can see that we have two new addresses that were pushed onto our stack.

To prove this, let’s look at the values of `r3` and `lr`.

```
(gdb) x/x $r3
0xe000ed08:     Cannot access memory at address 0xe000ed08
(gdb) x/x $lr
0x1000018f <platform_entry+8>:  0x00478849
(gdb) x/x $sp
0x20081ff8:     0xe000ed08
```

The stack pointer is currently at address `0x20081ff8`, and the value at that location is `0xe000ed08`, which matches the value in `r3`. This suggests that `r3` was pushed onto the stack first.

```
(gdb) x/x $sp+4
0x20081ffc:     0x1000018f
```

We find the value `0x1000018f`, which matches the value in the link register. This confirms that the link register was pushed onto the stack after `r3`.

Because the stack grows downward in memory, each push operation moves the stack pointer to a lower address. The original stack pointer was at `0x20082000`, and after pushing two values, it moved down to `0x20081ff8`.

This behavior aligns with ARM calling conventions. During a function prologue, registers such as `lr` and any callee-saved registers are pushed onto the stack to preserve their values. The stack pointer is adjusted accordingly, and the return address stored in `lr` ensures that control can return to the correct location once the function completes.

I hope this helps you understand how the stack works. We will continue to examine the stack throughout this course.

Let’s step-over the next instruction as it is a call to our below C- SDK function which is not of interest to as it simply sets up the MCU peripherals to communicate.

Our next step is to step-over the call to standard IO initialize all.

```
(gdb) x/5i 0x10000234
   0x10000234 <main>:   push    {r3, lr}
=> 0x10000236 <main+2>: bl      0x1000156c <stdio_init_all>
   0x1000023a <main+6>: ldr     r0, [pc, #8]    @ (0x10000244 <main+16>)
   0x1000023c <main+8>: bl      0x100015fc <__wrap_puts>
   0x10000240 <main+12>:        b.n     0x1000023a <main+6>
(gdb) n
8               printf("hello, world\r\n");
(gdb) x/5i 0x10000234
   0x10000234 <main>:   push    {r3, lr}
   0x10000236 <main+2>: bl      0x1000156c <stdio_init_all>
=> 0x1000023a <main+6>: ldr     r0, [pc, #8]    @ (0x10000244 <main+16>)
   0x1000023c <main+8>: bl      0x100015fc <__wrap_puts>
   0x10000240 <main+12>:        b.n     0x1000023a <main+6>
```

Now we are about to load the value INSIDE of a memory address at 0x10000244 into r0. The r0, [pc, #8] means take the value at the current program counter and add 8 to it and take that address's value and store it into r0. This is a pointer which means we are pointing to the value inside that address.

Let's si one step and examine what is inside r0 at this point.

```
(gdb) si
0x1000023c      8               printf("hello, world\r\n");
(gdb) x/x $r0
0x100019cc:     0x6c6c6568
```

Hmm… This does not look like an address however it does look like ascii chars to me. Let's look at an ascii table.

https://www.asciitable.com

We see 0x6c is l and we see it again so another l and 0x65 is e and 0x68 is h.

This is our *hello, world* string however it is backward! The reason is memory is stored in reverse byte order or little-endian order from memory to registers within the MCU.

We can see the full pointer to this char array or string by doing the below.

```
(gdb) x/s $r0
0x100019cc:     "hello, world\r"
```

In this chapter, we established a foundational reverse engineering workflow using both static and dynamic techniques. Through Ghidra, we examined the binary statically, observing the raw assembly and decompiled views to understand control flow and compiler optimizations. We noted subtle differences between our original source code and the decompiled output, such as the transformation of a while (true) loop into a do-while construct and the substitution of printf with puts for efficiency.

Using GDB, we transitioned into dynamic analysis, inspecting live register values and stack behavior during execution. We confirmed how the stack grows downward, how the link register is pushed to preserve return addresses, and how memory inspection reveals the inner workings of function calls. These observations aligned with ARM Cortex-M33 calling conventions and gave us a practical view of how the RP2350 handles execution at the instruction level.

Although the example was simple, it demonstrated the power of combining static and dynamic reverse engineering to gain insight into compiled binaries. With this workflow in place, we are now prepared to tackle more complex binaries, explore deeper architectural features of the RP2350, and refine our debugging strategies for embedded development.

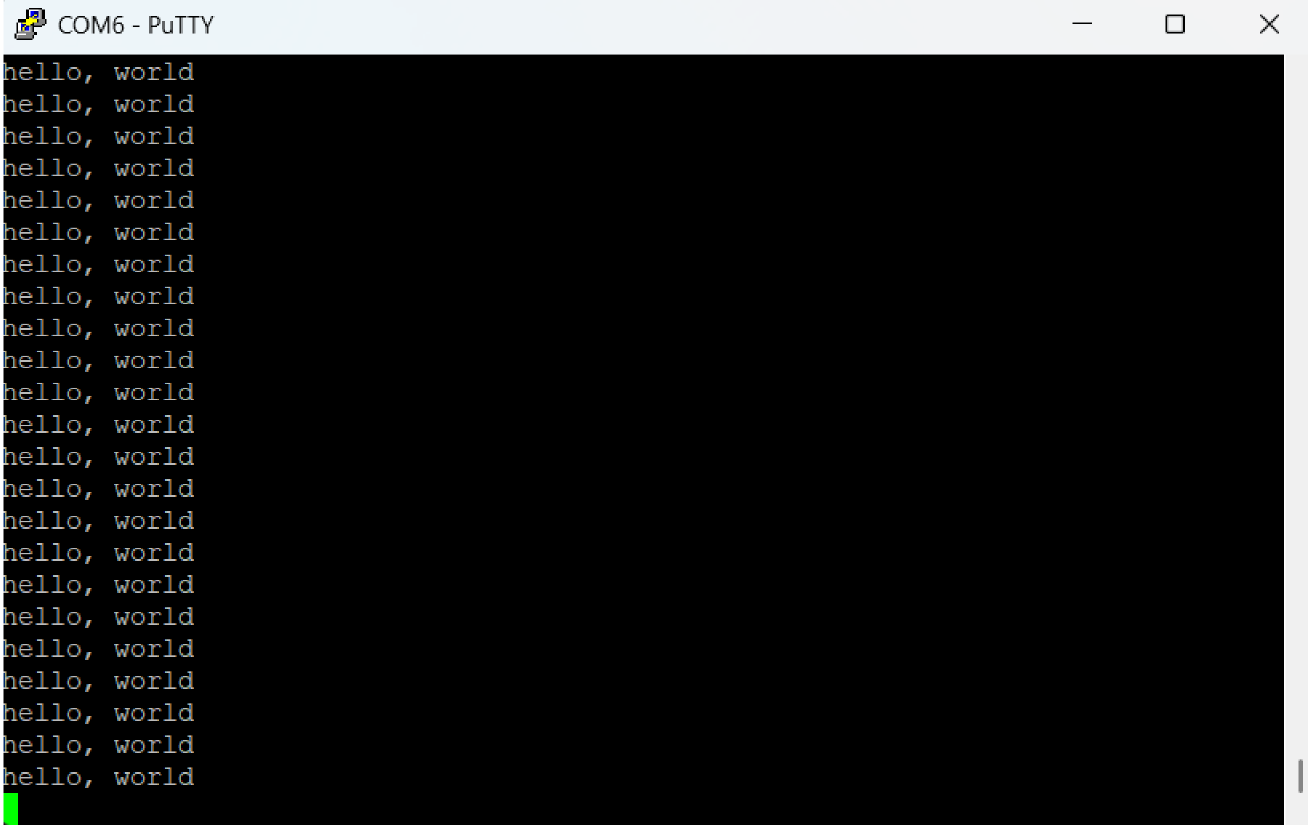
In our next chapter we will hack this simple binary.

Chapter 3: Hacking hello, world

Today we hack!

Let’s run OpenOCD to get our remote debug server going.

Let’s run our serial monitor and observe *hello, world* in the infinite loop.



```
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
```

Run OpenOCD with the below config.

```
openocd -f interface/cmsis-dap.cfg -f target/rp2350.cfg -c "adapter speed 5000"
```

Open a new terminal and then run the following to launch our dynamic debugger called GDB.

```
arm-none-eabi-gdb build/0x0001_hello-world.elf
```

Once it loads, we need to target our remote server.

```
target remote :3333
```

We need to halt the currently running binary.

```
monitor reset halt
```

We notice our *hello, world* within the serial monitor is halted as expected.

Let’s re-examine main.

```
(gdb) x/5i 0x10000234
    0x10000234 <main>:   push    {r3, lr}
    0x10000236 <main+2>: bl      0x1000156c <stdio_init_all>
    0x1000023a <main+6>: ldr     r0, [pc, #8]    @ (0x10000244 <main+16>)
    0x1000023c <main+8>: bl      0x100015fc <__wrap_puts>
    0x10000240 <main+12>:        b.n     0x1000023a <main+6>
```

The first thing we need to do to hack our system LIVE is to set a breakpoint to the address right before the call to puts and then continue.

```
(gdb) b *0x1000023c
Breakpoint 1 at 0x1000023c: file C:/Users/assem.KEVINTHOMAS/Documents/Embedded-
Hacking/0x0001_hello-world/0x0001_hello-world.c, line 8.
Note: automatically using hardware breakpoints for read-only addresses.
(gdb) c
Continuing.

Thread 1 "rp2350.cm0" hit Breakpoint 1, 0x1000023c in main ()
    at C:/Users/assem.KEVINTHOMAS/Documents/Embedded-Hacking/0x0001_hello-world/0x0001_hello-
world.c:8
warning: Source file is more recent than executable.
8               printf("hello, world\r\n");
(gdb) disas
Dump of assembler code for function main:
   0x10000234 <+0>:     push    {r3, lr}
   0x10000236 <+2>:     bl      0x1000156c <stdio_init_all>
   0x1000023a <+6>:     ldr     r0, [pc, #8]    @ (0x10000244 <main+16>)
=> 0x1000023c <+8>:     bl      0x100015fc <__wrap_puts>
   0x10000240 <+12>:    b.n     0x1000023a <main+6>
   0x10000242 <+14>:    nop
   0x10000244 <+16>:    adds    r4, r1, r7
   0x10000246 <+18>:    asrs    r0, r0, #32
End of assembler dump.
(gdb)
```

The next thing we need to do is hijack the value of *hello, world* which is pointed to in r0 and create our own data and fill it with a hacked malicious string.

```
(gdb) x/s $r0
0x100019cc:     "hello, world\r"
(gdb) set $r0 = "hacky, world\r"
evaluation of this expression requires the program to have a function "malloc".
```

```
(gdb) x/s $r0
0x100019cc:     "hello, world\r"
```

Oh no it did not work! Now what!

GDB interprets "hacky, world\r" as a string literal, and it tries to evaluate it as a pointer to a valid memory address where that string resides. But GDB itself does not allocate memory for that string unless the program being debugged has already loaded it somewhere, typically via the C runtime or a statically defined string in the binary.

The error isn't because GDB is trying to call malloc. It's because GDB is trying to resolve the string literal to a memory address, and it fails because that string doesn't exist in the program's memory space. If you're debugging a bare-metal binary or early startup code on the RP2350, there's no runtime environment to provide that string, and no global symbols or .data section initialized with it.

Therefore, we need to create our string in SRAM!

If we remember from the last chapter, the RP2350 datasheet states that the SRAM starts at 0x20000000. With that we can create a new string in SRAM directly.

```
(gdb) set {char[14]} 0x20000000 = {'h','a','c','k','y',',',' ','w','o','r','l','d','\r','\0'}
(gdb) x/s 0x20000000
0x20000000 <ram_vector_table>:  "hacky, world\r"
```

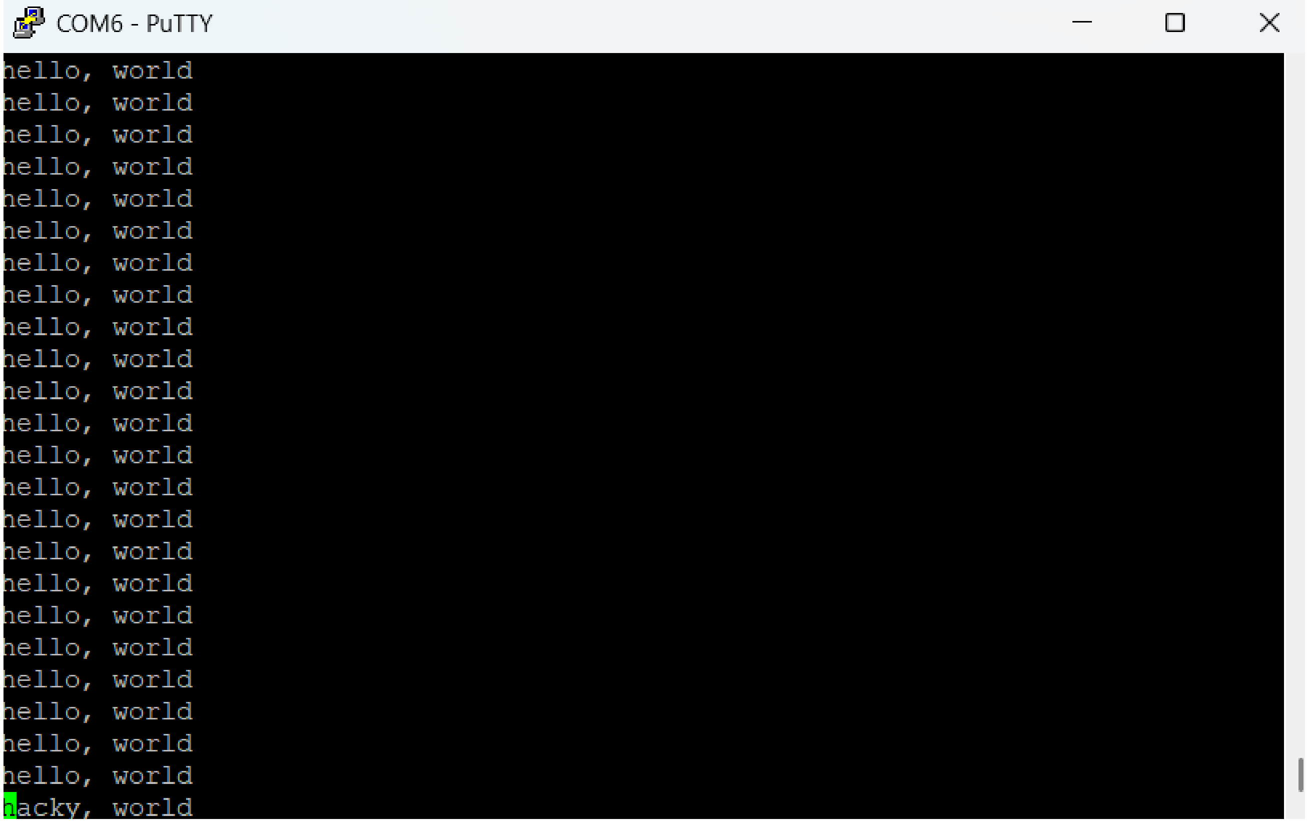
Now to need to hijack the address inside r0 and change it to our hacked address in SRAM and verify our hack.

```
(gdb) set $r0 = 0x20000000
(gdb) x/x $r0
0x20000000 <ram_vector_table>:  0x68
(gdb) x/s $r0
0x20000000 <ram_vector_table>:  "hacky, world\r"
```

Now let's continue and execute our hack!

```
(gdb) c
Continuing.

Thread 1 "rp2350.cm0" hit Breakpoint 1, 0x1000023c in main ()
    at C:/Users/assem.KEVINTHOMAS/Documents/Embedded-Hacking/0x0001_hello-world/0x0001_hello-
world.c:8
8               printf("hello, world\r\n");
```

Let’s verify our hack!



```
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hacky, world
```

BOOM! We did it! We successfully hacked our LIVE binary! You can see *hacky, world* now being printed to our serial monitor!

“With great power comes great responsibility!”

Imagine we were in an enemy ICS industrial control facility, say a nuclear power enrichment facility, and we had to hack the value of one of their centrifuges.

After we hacked the centrifuges, we need to make sure the value that the engineers are seeing on their monitor shows normal.

THIS IS EXACTLY HOW WE WOULD DO THIS!

In our next lesson we will discuss Embedded System Analysis.

Chapter 4: Embedded System Analysis

We are working with a microcontroller so there is no operating system in use. This is what we refer to as bare-metal programming.

We must start with what happens when the RP2350 gets power.

The RP2350 has an on-chip bootloader (bootrom) that executes immediately when the chip gets power.

```
// ROOT ADDRESSES
#define BOOTROM_MAGIC_OFFSET 0x10
#define BOOTROM_FUNC_TABLE_OFFSET 0x14
#if PICO_RP2040
#define BOOTROM_DATA_TABLE_OFFSET 0x16
#endif

#if PICO_RP2040
#define BOOTROM_VTABLE_OFFSET           0x00
#define BOOTROM_TABLE_LOOKUP_OFFSET     0x18
#else
#define BOOTROM_WELL_KNOWN_PTR_SIZE 2
#if defined(__riscv)
#define BOOTROM_ENTRY_OFFSET            0x7dfc
#define BOOTROM_TABLE_LOOKUP_ENTRY_OFFSET (BOOTROM_ENTRY_OFFSET - 
BOOTROM_WELL_KNOWN_PTR_SIZE)
#define BOOTROM_TABLE_LOOKUP_OFFSET     (BOOTROM_ENTRY_OFFSET - 
BOOTROM_WELL_KNOWN_PTR_SIZE*2)
#else
#define BOOTROM_VTABLE_OFFSET 0x00
```

src/rp2_common/boot_bootrom_headers/include/boot/bootrom_constants.h

The RP2350 bootrom is a mask ROM that contains the first-stage bootloader code. This bootrom provides various functions including flash initialization, boot path selection, and hardware setup.

```
static inline void rom_connect_internal_flash(void) {
    rom_connect_internal_flash_fn func = (rom_connect_internal_flash_fn) 
rom_func_lookup_inline(ROM_FUNC_CONNECT_INTERNAL_FLASH);
    func();
}
```

src/rp2_common/pico_bootrom/include/pico/bootrom.h

On RP2350, boot stage 2 is called as a regular function and must return normally, unlike RP2040, **boot2_generic_03h.S**. The second stage bootloaders are responsible for setting up external flash to enable XIP operation.

The default boot2_generic_03h implementation configures the QMI for basic serial flash operation.

```
// The QMI is automatically configured for 03h XIP straight out of reset,
// but this code can't assume it's still in that state. Set up memory
// window 0 for 03h serial reads.

// Setup timing parameters: short sequential-access cooldown, configured
// CLKDIV and RXDELAY, and no constraints on CS max assertion, CS min
// deassertion, or page boundary burst breaks.

#define INIT_M0_TIMING (\
    1                      << QMI_M0_TIMING_COOLDOWN_LSB |\
    PICO_FLASH_SPI_RXDELAY << QMI_M0_TIMING_RXDELAY_LSB |\
    PICO_FLASH_SPI_CLKDIV  << QMI_M0_TIMING_CLKDIV_LSB |\
0)

// Set command constants
#define INIT_M0_RCMD (\
    CMD_READ             << QMI_M0_RCMD_PREFIX_LSB |\
0)

// Set read format to all-serial with a command prefix
#define INIT_M0_RFMT (\
    QMI_M0_RFMT_PREFIX_WIDTH_VALUE_S << QMI_M0_RFMT_PREFIX_WIDTH_LSB |\
    QMI_M0_RFMT_ADDR_WIDTH_VALUE_S   << QMI_M0_RFMT_ADDR_WIDTH_LSB |\
    QMI_M0_RFMT_SUFFIX_WIDTH_VALUE_S << QMI_M0_RFMT_SUFFIX_WIDTH_LSB |\
    QMI_M0_RFMT_DUMMY_WIDTH_VALUE_S  << QMI_M0_RFMT_DUMMY_WIDTH_LSB |\
    QMI_M0_RFMT_DATA_WIDTH_VALUE_S   << QMI_M0_RFMT_DATA_WIDTH_LSB |\
    QMI_M0_RFMT_PREFIX_LEN_VALUE_8   << QMI_M0_RFMT_PREFIX_LEN_LSB |\
0)
```

src/rp2350/boot_stage2/boot2_generic_03h.S

The configuration sets up timing parameters with a short cooldown, configurable clock divider and RX delay, and configures the QMI for 03h serial read commands with all-serial format.

After QMI configuration, boot stage 2 performs a dummy transfer to initialize the flash device and then configures continuous read mode.

```
    // Dummy transfer
    mov r1, #XIP_NOCACHE_NOALLOC_BASE
    ldrb r1, [r1]

    // Set prefix length to 0, as flash no longer expects to see commands
    bic r0, #QMI_M0_RFMT_PREFIX_LEN_BITS
    str r0, [r3, #QMI_M0_RFMT_OFFSET]
```

src/rp2350/boot_stage2/boot2_w25q080.S

The dummy transfer activates XIP mode, and the prefix length is set to 0 since the flash no longer expects command prefixes for subsequent reads.

Boot stage 2 returns control to the bootrom, which then jumps to the reset_vector as that value is the second entry in the vector table at 0x10000004 which in our case is 0x1000015d.

```
// Pull in standard exit routine
#include "boot2_helpers/exit_from_boot2.S"
```

We will focus on execute in place, or XIP, a technique where firmware instructions are executed directly from non-volatile memory rather than being copied into RAM. In the context of the RP2350, this typically means that code is mapped from external or internal Flash memory into the processor's address space, allowing instructions to be fetched and executed without relocation.

This approach conserves RAM and simplifies startup, since the processor can begin executing code immediately after reset. The Flash region is memory-mapped, so the CPU treats it as part of its normal instruction space. While XIP is efficient for read-only code execution, it's important to note that Flash access times are generally slower than RAM, and write operations require special handling.

Understanding XIP is essential for debugging and reverse engineering, as it affects how code is laid out, how breakpoints behave, and how memory regions are protected or cached. Let me know if you'd like to walk through the RP2350's memory map or trace instruction fetches from Flash during startup.

When we examine the first few values at 0x10000000, we begin with the vector table.

```
(gdb) x/4x 0x10000000
0x10000000 <__vectors>: 0x20082000      0x1000015d      0x1000011b      0x1000011d
```

Address	Value	Meaning
0x10000000	0x20082000	Initial Stack Pointer (SP)
0x10000004	0x1000015d	Reset Handler (entry point after boot)
0x10000008	0x1000011b	NMI Handler
0x1000000C	0x1000011d	HardFault Handler

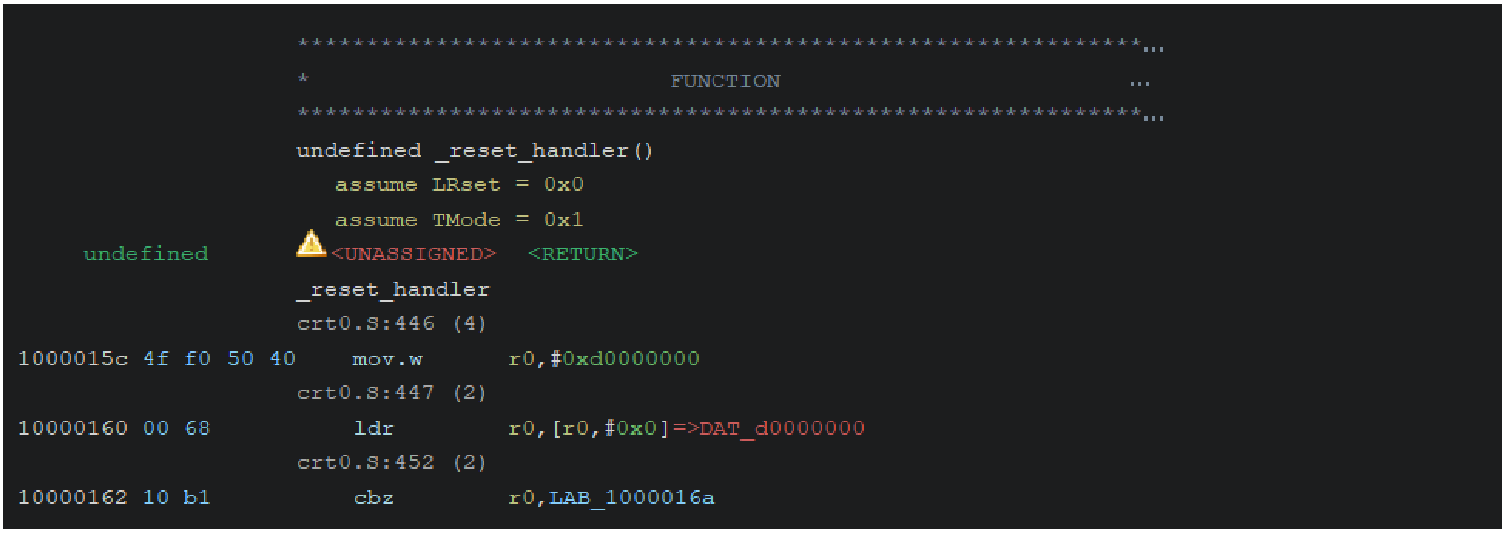
The reset handler is at 0x1000015d, so disassembling from there will show the actual startup logic.

```
(gdb) x/3i 0x1000015d
   0x1000015d <_reset_handler>: mov.w   r0, #3489660928 @ 0xd0000000
   0x10000161 <_reset_handler+4>:       ldr     r0, [r0, #0]
   0x10000163 <_reset_handler+6>:       cbz     r0, 0x1000016a <hold_non_core0_in_bootrom+6>
```

On ARM Cortex-M chips, all code runs in Thumb mode, and the processor uses the least significant bit of an address to mark this: if bit 0 is set, it means "Thumb," if clear, it means "ARM." The actual instructions still live at even addresses, but debuggers and disassemblers handle this flag differently as GDB shows the address exactly as it appears in the vector table (with the Thumb bit set), while Ghidra strips that bit off and shows the true instruction address. So, both are correct, they're just presenting the same location in two slightly different ways.

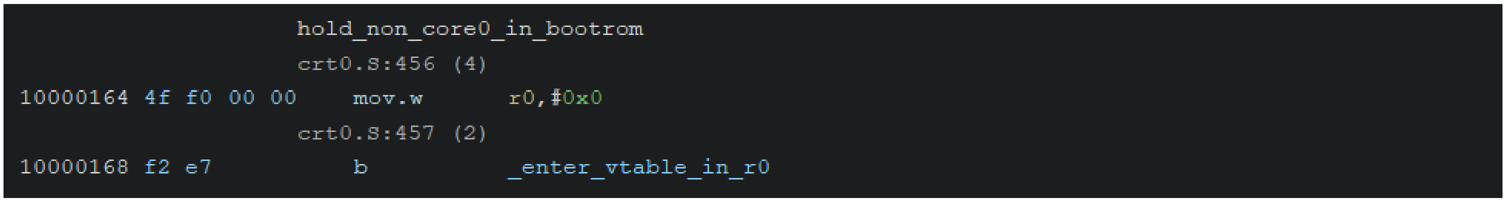
Lets start from the reset handler and work our way to main.

At 0x1000015d: mov.w r0, #0xd0000000 - Load SIO base address.
At 0x10000161: ldr r0, [r0, #0x0] - Read the CPUID register.
At 0x10000163: cbz r0, LAB_1000016a - Branch if core 0 (r0 == 0).



```
                    **************************************************************...
                    *                          FUNCTION                             ...
                    **************************************************************...
                    undefined _reset_handler()
                       assume LRset = 0x0
                       assume TMode = 0x1
     undefined        ⚠<UNASSIGNED>   <RETURN>
                    _reset_handler
                    crt0.S:446 (4)
1000015c 4f f0 50 40    mov.w      r0,#0xd0000000
                    crt0.S:447 (2)
10000160 00 68          ldr        r0,[r0,#0x0]=>DAT_d0000000
                    crt0.S:452 (2)
10000162 10 b1          cbz        r0,LAB_1000016a
```

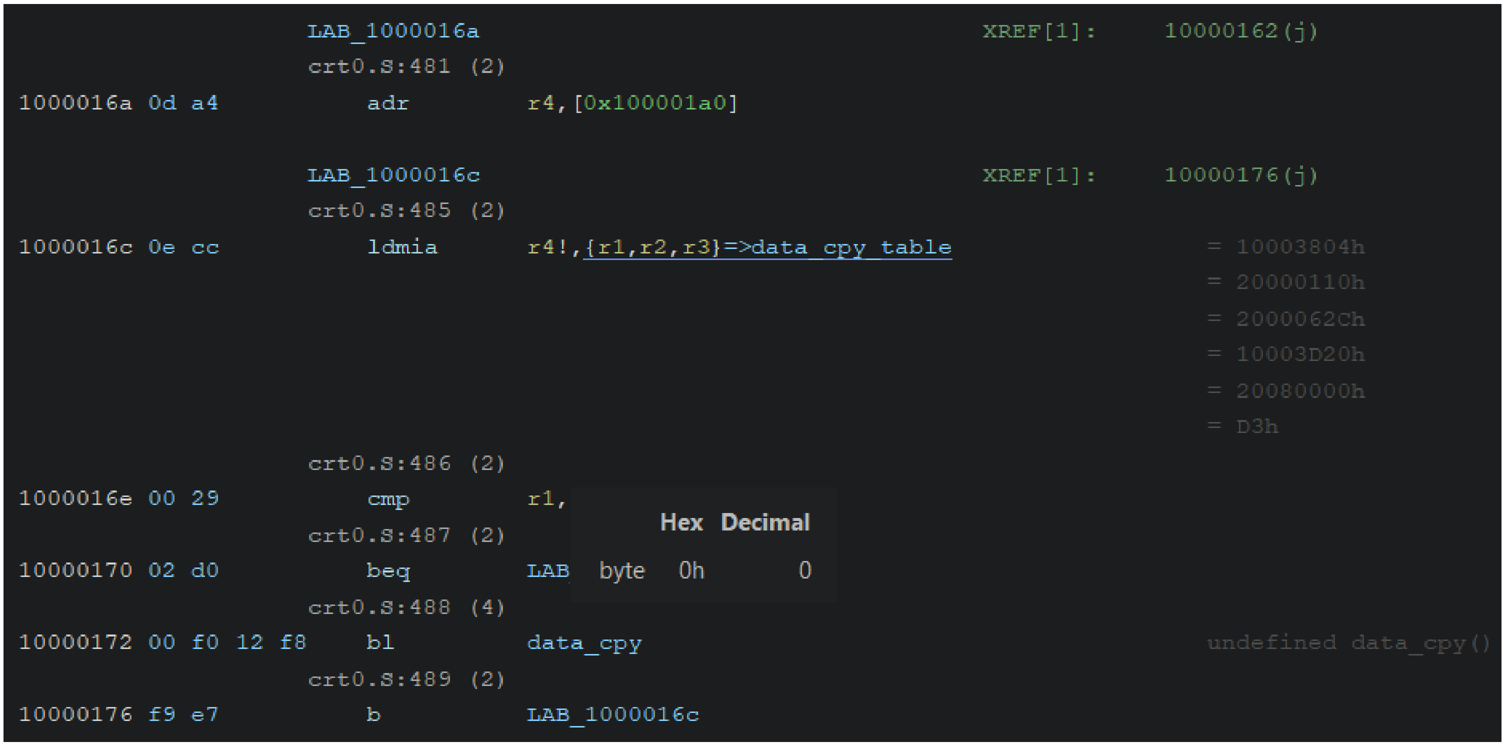
At 0x10000164-0x10000168: If not core 0, send back to bootrom.



```
                    hold_non_core0_in_bootrom
                    crt0.S:456 (4)
10000164 4f f0 00 00    mov.w      r0,#0x0
                    crt0.S:457 (2)
10000168 f2 e7          b          _enter_vtable_in_r0
```

Data Copy Phase (0x1000016a-0x10000176)

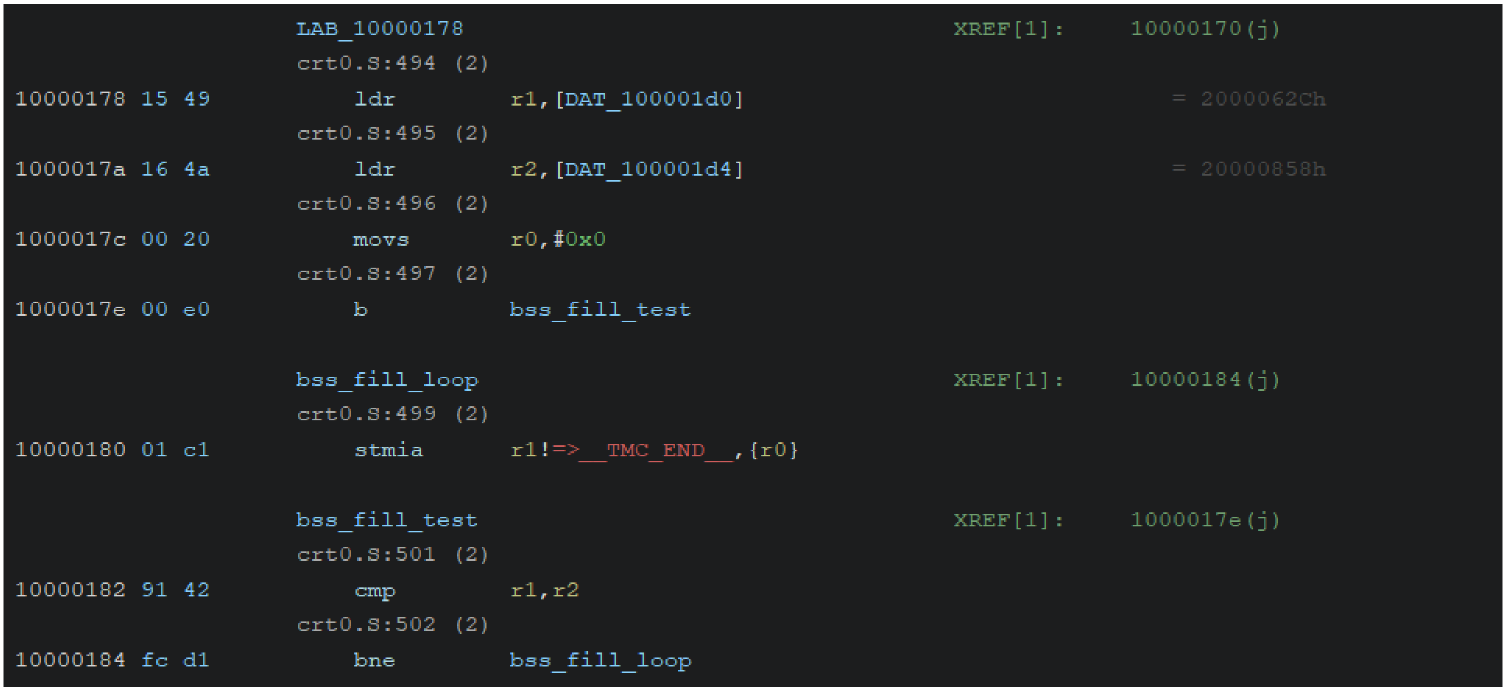
- Copies initialized data from flash to RAM using the data_cpy_table.
- The loop at LAB_1000016c processes each entry in the copy table.



```
                     LAB_1000016a                                    XREF[1]:     10000162(j)
                     crt0.S:481 (2)
1000016a 0d a4           adr         r4,[0x100001a0]

                     LAB_1000016c                                    XREF[1]:     10000176(j)
                     crt0.S:485 (2)
1000016c 0e cc           ldmia       r4!,{r1,r2,r3}=>data_cpy_table                  = 10003804h
                                                                                      = 20000110h
                                                                                      = 2000062Ch
                                                                                      = 10003D20h
                                                                                      = 20080000h
                                                                                      = D3h
                     crt0.S:486 (2)
1000016e 00 29           cmp         r1,
                     crt0.S:487 (2)
10000170 02 d0           beq         LAB
                     crt0.S:488 (4)
10000172 00 f0 12 f8     bl          data_cpy                                         undefined data_cpy()
                     crt0.S:489 (2)
10000176 f9 e7           b           LAB_1000016c
```

BSS Clear Phase (0x10000178-0x10000184)

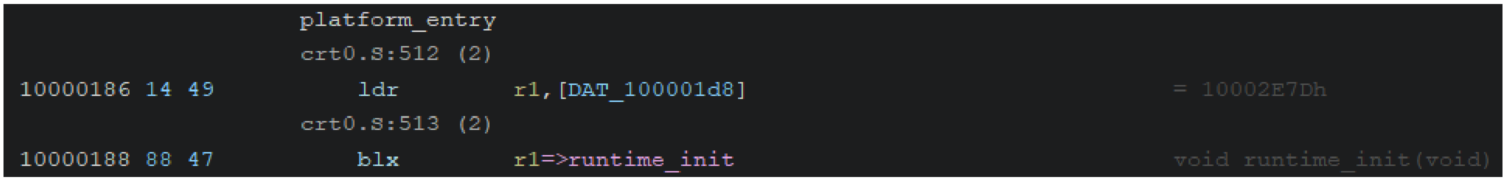
- Zeros out the BSS section in RAM.
- The loop clears memory from 0x2000062c to 0x20000858.



```
                    LAB_10000178                                 XREF[1]:     10000170(j)
                    crt0.S:494 (2)
10000178 15 49          ldr         r1,[DAT_100001d0]                                  = 2000062Ch
                    crt0.S:495 (2)
1000017a 16 4a          ldr         r2,[DAT_100001d4]                                  = 20000858h
                    crt0.S:496 (2)
1000017c 00 20          movs        r0,#0x0
                    crt0.S:497 (2)
1000017e 00 e0          b           bss_fill_test

                    bss_fill_loop                                XREF[1]:     10000184(j)
                    crt0.S:499 (2)
10000180 01 c1          stmia       r1!=>__TMC_END__,{r0}

                    bss_fill_test                                XREF[1]:     1000017e(j)
                    crt0.S:501 (2)
10000182 91 42          cmp         r1,r2
                    crt0.S:502 (2)
10000184 fc d1          bne         bss_fill_loop
```

Runtime Initialization (0x10000186-0x10000188)

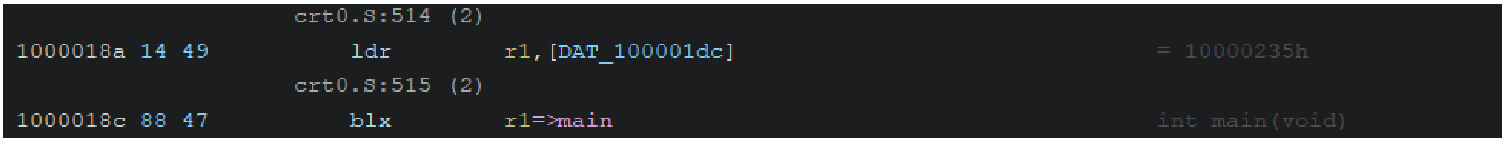
- Calls runtime_init.
- This sets up the C runtime environment.



```
                    platform_entry
                    crt0.S:512 (2)
10000186 14 49          ldr         r1,[DAT_100001d8]                                  = 10002E7Dh
                    crt0.S:513 (2)
10000188 88 47          blx         r1=>runtime_init                                   void runtime_init(void)
```

Main Function Call (0x1000018a-0x1000018c)

- Finally calls main at 0x10000234.



```
                    crt0.S:514 (2)
1000018a 14 49          ldr         r1,[DAT_100001dc]                                  = 10000235h
                    crt0.S:515 (2)
1000018c 88 47          blx         r1=>main                                           int main(void)
```

But where does this all come from?

We setup VSCode with the Pico extension. In Windows you will see something like the following.
C:\Users\assem.KEVINTHOMAS\.pico-sdk\sdk\2.2.0\src\rp2_common\pico_crt0

There is a file called **crt0.S** to which this all begins!

Below is a snippet from the file.

```
.section .vectors, "ax"
.align 2

.global __vectors, __VECTOR_TABLE, __vectors_end
__VECTOR_TABLE:
__vectors:
.word __StackTop
.word _reset_handler
```

These entries correspond to 0x20082000 which is the stack pointer and 0x1000015d which is the reset handler.

The RP2350 vector table is a critical structure that defines how the microcontroller responds to exceptions and interrupts, but it's not the first thing the ARM Cortex-M33 core looks at when it powers up as the on-chip bootrom executes first, followed by boot stage 2 configuration of the flash interface, and only then does the bootrom read the vector table and jump to the application's reset handler.

The vector table lives at 0x10000000 in the RP2350's XIP Flash region with the stack pointer at offset 0x00 and the reset vector at offset 0x04, but this location is determined by the application's linker script rather than being a fixed hardware requirement as the bootrom uses the Vector Table Offset Register (VTOR) to locate the table dynamically.

We will find the linker scripts specifically for our 2.2.0 sdk in a folder similar to this.
C:\Users\assem.KEVINTHOMAS\.pico-sdk\sdk\2.2.0\src\rp2_common\pico_crt0\rp2350

There you will see **memmap_default.ld** which is the standard XIP configuration where code executes directly from Flash at 0x10000000.

In our linker script we see the following.

```
MEMORY
{
    INCLUDE "pico_flash_region.ld"
    RAM(rwx) : ORIGIN =  0x20000000, LENGTH = 512k
    SCRATCH_X(rwx) : ORIGIN = 0x20080000, LENGTH = 4k
    SCRATCH_Y(rwx) : ORIGIN = 0x20081000, LENGTH = 4k
}
```

Then as we look deeper, we see the following.

```
__StackTop = ORIGIN(SCRATCH_Y) + LENGTH(SCRATCH_Y);
```

We see above that the ORIGIN(SCRATCH_Y) is 0x20081000 and the length is 4k therefore we get the following which we can verify in GDB.

```
__StackTop = 0x20081000 + 0x1000 = 0x20082000

(gdb) x/x 0x10000000
0x10000000 <__vectors>: 0x20082000
```

This value is emitted into the vector table at address 0x10000000 via the **crt0.S** file.

```
.section .vectors, "ax"
.align 2

.global __vectors, __VECTOR_TABLE, __vectors_end
__VECTOR_TABLE:
__vectors:
.word __StackTop
.word _reset_handler
```

The Cortex-M33 core loads this into the stack pointer register and places the stack at the top of the SCRATCH_Y region, which is a small, dedicated RAM block reserved for the core 0 stack.

At the end of the vector table, we see the following.

```
(gdb) x/36i 0x10000110
   0x10000110 <isr_usagefault>: mrs     r0, IPSR
   0x10000114 <isr_usagefault+4>:       subs    r0, #16
   0x10000116 <unhandled_user_irq_num_in_r0>:   bkpt    0x0000
   0x10000118 <isr_invalid>:    bkpt    0x0000
   0x1000011a <isr_nmi>:        bkpt    0x0000
   0x1000011c <isr_hardfault>:  bkpt    0x0000
   0x1000011e <isr_svcall>:     bkpt    0x0000
   0x10000120 <isr_pendsv>:     bkpt    0x0000
   0x10000122 <isr_systick>:    bkpt    0x0000
   0x10000124 <__default_isrs_end>:                     @ <UNDEFINED> instruction: 0xebf27188
   0x10000128 <__default_isrs_end+4>:   subs    r0, r4, r4
   0x1000012a <__default_isrs_end+6>:   asrs    r0, r0, #32
   0x1000012c <__default_isrs_end+8>:   subs    r4, r1, r5
   0x1000012e <__default_isrs_end+10>:  asrs    r0, r0, #32
   0x10000130 <__default_isrs_end+12>:  lsls    r0, r4, #6
   0x10000132 <__default_isrs_end+14>:  asrs    r0, r0, #32
   0x10000134 <__default_isrs_end+16>:  add     r3, pc, #576    @ (adr r3, 0x10000378
<runtime_init_per_core_irq_priorities+44>)
   0x10000136 <__default_isrs_end+18>:  b.n     0xfffff6e
   0x10000138 <__binary_info_header_end>:       udf     #211    @ 0xd3
   0x1000013a <__binary_info_header_end+2>:                     @ <UNDEFINED> instruction:
0xffff0142
   0x1000013e <__binary_info_header_end+6>:     asrs    r1, r4, #32
   0x10000140 <__binary_info_header_end+8>:     lsls    r7, r7, #7
   0x10000142 <__binary_info_header_end+10>:    movs    r0, r0
   0x10000144 <__binary_info_header_end+12>:    subs    r0, r6, r6
   0x10000146 <__binary_info_header_end+14>:    movs    r0, r0
   0x10000148 <__binary_info_header_end+16>:    adds    r5, #121        @ 0x79
   0x1000014a <__binary_info_header_end+18>:    add     r3, sp, #72     @ 0x48
   0x1000014c <_entry_point>:   mov.w   r0, #0
   0x10000150 <_enter_vtable_in_r0>:    ldr     r1, [pc, #120]  @ (0x100001cc
<data_cpy_table+44>)
   0x10000152 <_enter_vtable_in_r0+2>:  str     r0, [r1, #0]
   0x10000154 <_enter_vtable_in_r0+4>:  ldmia   r0!, {r1, r2}
   0x10000156 <_enter_vtable_in_r0+6>:  msr     MSP, r1
   0x1000015a <_enter_vtable_in_r0+10>: bx      r2
   0x1000015c <_reset_handler>: mov.w   r0, #3489660928 @ 0xd0000000
   0x10000160 <_reset_handler+4>:       ldr     r0, [r0, #0]
   0x10000162 <_reset_handler+6>:       cbz     r0, 0x1000016a <hold_non_core0_in_bootrom+6>
```

The first section is the `isr_usagefault` to which we will do a little digging.

```
arm-none-eabi-nm -C build\0x0001_hello-world.elf | findstr isr_usagefault
10000110 W isr_usagefault
```

This means this is weakly defined as **crt0.S** has only the stub but the code we see below is elsewhere.

```
0x10000110 <isr_usagefault>: mrs     r0, IPSR
0x10000114 <isr_usagefault+4>:       subs    r0, #16
```

In **crt0.S** we see the following.

```
// Declare a weak symbol for each ISR.
// By default, they will fall through to the undefined IRQ handler below (breakpoint),
// but can be overridden by C functions with correct name.

.macro decl_isr_bkpt name
.weak \name
.type \name,%function
.thumb_func
\name:
    bkpt #0
.endm
```

We can try searching with PowerShell.

```
PS C:\Users\assem.KEVINTHOMAS> Get-ChildItem -Recurse -Include *.S -Path
"C:\Users\assem.KEVINTHOMAS\.pico-sdk" | Select-String "mrs r0, IPSR"
```

Sadly, this returns no result. Let’s look within our running GDB instance.

```
(gdb) list isr_usagefault
315     .global __unhandled_user_irq
316     .thumb_func
317     __unhandled_user_irq:
318     // if we include the implementation if there could be a valid IRQ hanler in the
vtable that uses it
```

```
#if !(PICO_NO_RAM_VECTOR_TABLE && PICO_MINIMAL_STORED_VECTOR_TABLE)
    mrs  r0, ipsr
    subs r0, #16
.global unhandled_user_irq_num_in_r0
unhandled_user_irq_num_in_r0:
#endif
```

Now when we look in **crt0.S**, we can see the following.

```
// All unhandled USER IRQs fall through to here.
// Additionally, if the Armv9-M MemManage/BusFault/UsageFault/SecureFault/DebugMonitor
exceptions
// are enabled, but the handlers are not defined, then unhandled_user_irq_num_in_r0 will
// also be reached, but with a negative exception number (e.g. MemManage == -12)
.global __unhandled_user_irq
.thumb_func
__unhandled_user_irq:
// if we include the implementation if there could be a valid IRQ hanler in the vtable
that uses it
#if !(PICO_NO_RAM_VECTOR_TABLE && PICO_MINIMAL_STORED_VECTOR_TABLE)
    mrs  r0, ipsr
    subs r0, #16
#endif
```

Let’s now examine the next few lines of GDB.

```
(gdb) x/36i 0x10000110
   ..
   0x10000116 <unhandled_user_irq_num_in_r0>:   bkpt    0x0000
   0x10000118 <isr_invalid>:    bkpt    0x0000
   0x1000011a <isr_nmi>:        bkpt    0x0000
   0x1000011c <isr_hardfault>:  bkpt    0x0000
   0x1000011e <isr_svcall>:     bkpt    0x0000
   0x10000120 <isr_pendsv>:     bkpt    0x0000
   0x10000122 <isr_systick>:    bkpt    0x0000
   ..
```

We can see in **crt0.S**, directly below our other code, we see the following.

```
.global unhandled_user_irq_num_in_r0
unhandled_user_irq_num_in_r0:
#endif
    // note the next instruction is a breakpoint too, however we have a 2 byte alignment
hole
    // and it is preferrable to have distinct labels, to inform the user what has happened
in the debugger.
    bkpt #0

decl_isr_bkpt isr_invalid
#if !PICO_MINIMAL_STORED_VECTOR_TABLE
// these are separated out into individual BKPT instructions with label for clarity
decl_isr_bkpt isr_nmi
decl_isr_bkpt isr_hardfault
decl_isr_bkpt isr_svcall
decl_isr_bkpt isr_pendsv
decl_isr_bkpt isr_systick
#endif
```

Let’s continue our analysis with the next few lines.

```
(gdb) x/36i 0x10000110
   ..
   0x10000124 <__default_isrs_end>:                      @ <UNDEFINED> instruction: 0xebf27188
   0x10000128 <__default_isrs_end+4>:   subs    r0, r4, r4
   0x1000012a <__default_isrs_end+6>:   asrs    r0, r0, #32
   0x1000012c <__default_isrs_end+8>:   subs    r4, r1, r5
   0x1000012e <__default_isrs_end+10>:  asrs    r0, r0, #32
   0x10000130 <__default_isrs_end+12>:  lsls    r0, r4, #6
   0x10000132 <__default_isrs_end+14>:  asrs    r0, r0, #32
   0x10000134 <__default_isrs_end+16>:  add     r3, pc, #576    @ (adr r3, 0x10000378
<runtime_init_per_core_irq_priorities+44>)
   0x10000136 <__default_isrs_end+18>:  b.n     0xfffff6e
   ..
```

In our **crt0.S**, we see only the following.

```
.global __default_isrs_end
__default_isrs_end:
```

Where does this actual code come from?

It’s not code it is a binary-info header emitted by the startup assembly, sitting immediately after the default ISR marker.

In PowerShell, let’s do the following.

```
arm-none-eabi-objdump -d --source build\0x0001_hello-world.elf |
  Select-String '^\s*1000012[4-9]|^\s*1000013[0-6]' -Context 1,2 |
  ForEach-Object { $_.Context.PreContext + $_.Line + $_.Context.PostContext } |
  ForEach-Object { $_.Trim() } |
  Where-Object { $_ -ne "" } |
  Select-Object -Unique
10000124 <__default_isrs_end>:
10000124:       7188ebf2        .word   0x7188ebf2
10000128:       10001b20        .word   0x10001b20
1000012c:       10001b4c        .word   0x10001b4c
10000130:       100001a0        .word   0x100001a0
10000134:       e71aa390        .word   0xe71aa390
10000138 <__binary_info_header_end>:
```

Address	Value	Field / Symbol	Description
0x10000124	0x7188EBF2	Magic signature	A fixed identifier marking the start of the binary-info header. Used by tools/boot ROM to recognize this structure.
0x10000128	0x10001B20	Binary info start pointer	Address of the first entry in the .binary_info section. In this build, that’s __bi_ptr84.
0x1000012C	0x10001B4C	Binary info end pointer	Address just past the last .binary_info entry. Here it’s start + 0x2C bytes.
0x10000130	0x100001A0	Data copy table pointer	Address of data_cpy_table, used by the reset handler to copy initialised .data from flash to RAM.
0x10000134	0xE71AA390	Reserved / trailer constant	A fixed value defined in the SDK’s startup assembly; may serve as a checksum, version marker, or reserved field.
0x10000138	*(label)*	__binary_info_header_end	Symbol marking the end of the binary-info header block.

```
PS C:\Users\assem.KEVINTHOMAS\Documents\Embedded-Hacking\0x0001_hello-world> arm-none-eabi-
objdump -s -j .text build\0x0001_hello-world.elf | Select-String "10000120" -Context 0,6

10000120 00be00be f2eb8871 201b0010 4c1b0010  .......q ...L...
10000130 a0010010 90a31ae7 d3deffff 42012110  ............B.!.
10000140 ff010000 b01b0000 793512ab 4ff00000  ........y5..O...
10000150 1e490860 06c881f3 08881047 4ff05040  .I.`.......GO.P@
10000160 006810b1 4ff00000 f2e70da4 0ecc0029  .h..O..........)
10000170 02d000f0 12f8f9e7 1549164a 002000e0  .........I.J. ..
10000180 01c19142 fcd11449 88471449 88471449  ...B...I.G.I.G.I
```

Let’s continue with our GDB analysis.

```
(gdb) x/36i 0x10000110
   ..
   0x10000138 <__binary_info_header_end>:       udf     #211    @ 0xd3
   0x1000013a <__binary_info_header_end+2>:                     @ <UNDEFINED> instruction:
0xffff0142
   0x1000013e <__binary_info_header_end+6>:     asrs    r1, r4, #32
   0x10000140 <__binary_info_header_end+8>:     lsls    r7, r7, #7
   0x10000142 <__binary_info_header_end+10>:    movs    r0, r0
   0x10000144 <__binary_info_header_end+12>:    subs    r0, r6, r6
   0x10000146 <__binary_info_header_end+14>:    movs    r0, r0
   0x10000148 <__binary_info_header_end+16>:    adds    r5, #121        @ 0x79
   0x1000014a <__binary_info_header_end+18>:    add     r3, sp, #72     @ 0x48
   ..
```

In **crt0.S** we see the following.

```
.section .binary_info_header, "a"

// Header must be in first 256 bytes of main image (i.e. excluding flash boot2).
// For flash builds we put it immediately after vector table; for NO_FLASH the
// vectors are at a +0x100 offset because the bootrom enters RAM images directly
// at their lowest address, so we put the header in the VTOR alignment hole.

#if !PICO_NO_BINARY_INFO
binary_info_header:
.word BINARY_INFO_MARKER_START
.word __binary_info_start
.word __binary_info_end
.word data_cpy_table // we may need to decode pointers that are in RAM at runtime.
.word BINARY_INFO_MARKER_END
#endif

#include "embedded_start_block.inc.S"
```

Let’s dig in and see what we can find.

```
arm-none-eabi-objdump -d --source build\0x0001_hello-world.elf |
  Select-String '^\s*1000013[8-9]|^\s*1000014[0-9a-f]' -Context 1,2 |
  ForEach-Object { $_.Context.PreContext + $_.Line + $_.Context.PostContext } |
  ForEach-Object { $_.Trim() } |
  Where-Object { $_ -ne "" } |
  Select-Object -Unique
10000138 <__binary_info_header_end>:
10000138:       ffffded3        .word   0xffffded3
1000013c:       10210142        .word   0x10210142
10000140:       000001ff        .word   0x000001ff
10000144:       00001bb0        .word   0x00001bb0
10000148:       ab123579        .word   0xab123579
```

```
PS C:\Users\assem.KEVINTHOMAS\Documents\Embedded-Hacking\0x0001_hello-world> arm-none-eabi-
objdump -s --start-address=0x10000138 --stop-address=0x1000014c build\0x0001_hello-world.elf

build\0x0001_hello-world.elf:     file format elf32-littlearm

Contents of section .text:
 10000138 d3deffff 42012110 ff010000 b01b0000  ....B.!.........
 10000148 793512ab                             y5..
PS C:\Users\assem.KEVINTHOMAS\Documents\Embedded-Hacking\0x0001_hello-world> # Dump all
symbols and grep for our addresses
PS C:\Users\assem.KEVINTHOMAS\Documents\Embedded-Hacking\0x0001_hello-world> arm-none-eabi-nm
--numeric-sort build\0x0001_hello-world.elf |
>>     Select-String "10000138|1000013c|10000140|10000144|10000148"

10000138 T __binary_info_header_end
10000138 t embedded_block
```

Raw words and interpretations.

0x10000138: 0xFFFFDED3

- Marker start: Picobin block start marker (BlockMarkerStart).

0x1000013C: 0x10212142

- Four item-header bytes: This is not a pointer; it's the first item header packed into 4 bytes (1B head/type + 1B size + 2B typedata). In default RP2350 builds this is the IMAGE_TYPE item emitted by embedded_start_block.inc.S3.

0x10000140: 0x000001FF

- Next item header bytes or size field: Another 4 bytes belonging to the item sequence (depends on which items are compiled in; see decode steps below). For minimum metadata images, you'll see the LAST item header here1.

0x10000144: 0x00001BB0

- Link to next block (relative bytes) or continuation of item data: Picobin blocks store a 32-bit link
- "offset to next block from this header." If END_BLOCK is enabled in your build, this will be a positive offset to the end block; otherwise, it is 0 to loop to self2.

0x10000148: 0xAB123579

- Marker end: Picobin block end marker (BlockMarkerEnd).

As we continue our analysis.

```
(gdb) x/36i 0x10000110
   ..
   0x1000014c <_entry_point>:   mov.w   r0, #0
   ..
```

In **crt0.S** we see the following.

```
#if !PICO_CRT0_NO_RESET_SECTION
.section .reset, "ax"

// On flash builds, the vector table comes first in the image (conventional).
// On NO_FLASH builds, the reset handler section comes first, as the entry
// point is at offset 0 (fixed due to bootrom), and VTOR is highly-aligned.
// Image is entered in various ways:
//
// - NO_FLASH builds are entered from beginning by UF2 bootloader
//
// - Flash builds vector through the table into _reset_handler from boot2
//
// - Either type can be entered via _entry_point by the debugger, and flash builds
//   must then be sent back round the boot sequence to properly initialise flash

// ELF entry point:
.type _entry_point,%function
.thumb_func
.global _entry_point
_entry_point:
```

```
#if PICO_NO_FLASH
    // on the NO_FLASH case, we do not do a rest thru bootrom below, so the RCP may or may
not have been initialized:
    //
    // in the normal (e.g. UF2 download etc. case) we will have passed thru bootrom
initialization, but if
    // a NO_FLASH binary is loaded by the debugger, and run directly after a reset, then
we won't have.
    //
    // we must therefore initialize the RCP if it hasn't already been

#if HAS_REDUNDANCY_COPROCESSOR
    // just enable the RCP which is fine if it already was (we assume no other co-
processors are enabled at this point to save space)
    ldr r0, = PPB_BASE + M33_CPACR_OFFSET
    movs r1, #ARM_CPU_PREFIXED(CPACR_CP7_BITS)
    str r1, [r0]
    // only initialize canary seeds if they haven't been (as to do so twice is a fault)
    mrc p7, #1, apsr_nzcv, c0, c0, #0
    bmi 1f
    // i dont think it much matters what we initialized to, as to have gotten here we must
have not
    // gone thru the bootrom (which a secure boot would have)
    mcrr p7, #8, r0, r0, c0
    mcrr p7, #8, r0, r0, c1
    sev
1:
#endif
#if !__ARM_ARCH_6M__
    // Make sure stack limit is 0 if we came in thru the debugger; we do not know what it
should be
    movs r0, #0
    msr msplim, r0
#endif

    ldr r0, =__vectors
    // Vector through our own table (SP, VTOR will not have been set up at
    // this point). Same path for debugger entry and bootloader entry.
#else
    // Debugger tried to run code after loading, so SSI is in 03h-only mode.
    // Go back through bootrom + boot2 to properly initialise flash.
    ldr r0, =BOOTROM_VTABLE_OFFSET
#endif
```

What we see is ldr r0, =BOOTROM_VTABLE_OFFSET and this optimizes down to mov.w r0, #0.

The rest of the code up to main is here and directly translates in **crt0.S** nicely. Let's break this down piece by piece.

```
_enter_vtable_in_r0:
    ldr r1, =(PPB_BASE + ARM_CPU_PREFIXED(VTOR_OFFSET))
    str r0, [r1]
    ldmia r0!, {r1, r2}
    msr msp, r1
    bx r2
```

```
(gdb) x/36i 0x10000110
   ..
   0x10000150 <_enter_vtable_in_r0>:    ldr     r1, [pc, #120]  @ (0x100001cc
<data_cpy_table+44>)
   0x10000152 <_enter_vtable_in_r0+2>:  str     r0, [r1, #0]
   0x10000154 <_enter_vtable_in_r0+4>:  ldmia   r0!, {r1, r2}
   0x10000156 <_enter_vtable_in_r0+6>:  msr     MSP, r1
   0x1000015a <_enter_vtable_in_r0+10>: bx      r2
   ..
```

On the RP2350's Cortex-M33 core, _enter_vtable_in_r0 is a tiny hand-off routine that takes a pointer to a new vector table in r0, writes it into the Vector Table Offset Register (VTOR) so all future exceptions and interrupts use it, then reads the first two words from that table—the initial Main Stack Pointer value and the Reset_Handler address—loads the MSP accordingly, and finally branches to the Reset_Handler, effectively transferring execution as if the CPU had just reset into the new firmware.

```
.type _reset_handler,%function
.thumb_func
_reset_handler:
    // Note if we entered thru here on core 0, then we should have gone thru bootrom, so
SP (and MSPLIM) on Armv8-M
    // should already be set

    // Only core 0 should run the C runtime startup code; core 1 is normally
    // sleeping in the bootrom at this point but check to be sure (e.g. if
    // debugger put core 1 at the ELF entry point for some reason)
    ldr r0, =(SIO_BASE + SIO_CPUID_OFFSET)
    ldr r0, [r0]
#if __ARM_ARCH_6M__
    cmp r0, #0
    beq 1f
#else
    cbz r0, 1f
#endif
```

```
(gdb) x/36i 0x10000110
   ..
    0x1000015c <_reset_handler>: mov.w   r0, #3489660928 @ 0xd0000000
    0x10000160 <_reset_handler+4>:       ldr     r0, [r0, #0]
    0x10000162 <_reset_handler+6>:       cbz     r0, 0x1000016a
    ..
```

This `_reset_handler` snippet is the very first C-runtime entry point after reset on the RP2350, and its opening instructions are checking which CPU core is running. The `mov.w r0, #0xd0000000 / ldr r0, [r0]` sequence reads the SIO_CPUID register in the RP2350's SIO block, which returns 0 for core 0 and 1 for core 1. The `cbz r0`, 1f means "if this is core 0, branch to label 1," allowing only core 0 to proceed into the full C runtime startup (stack already set by the boot ROM). Core 1 normally sits idle in the boot ROM until explicitly started, so this guard prevents both cores from running the same initialization code—avoiding double-init of data sections, clocks, and peripherals if, for example, a debugger dropped core 1 directly at the ELF entry point.

```
hold_non_core0_in_bootrom:
    // Send back to the ROM to wait for core 0 to launch it.
    ldr r0, =BOOTROM_VTABLE_OFFSET
    b _enter_vtable_in_r0
1:

#if !PICO_RP2040 && PICO_EMBED_XIP_SETUP && !PICO_NO_FLASH
    // Execute boot2 on the core 0 stack (it also gets copied into BOOTRAM due
    // to inclusion in the data copy table below). Note the reference
    // to __boot2_entry_point here is what prevents the .boot2 section from
    // being garbage-collected.
_copy_xip_setup:
    ldr r1, =__boot2_entry_point
    mov r3, sp
    add sp, #-256
    mov r2, sp
    bl data_cpy
_call_xip_setup:
    mov r0, sp
    adds r0, #1
    blx r0
    add sp, #256
#endif

    // In a NO_FLASH binary, don't perform .data etc copy, since it's loaded
    // in-place by the SRAM load. Still need to clear .bss
#if !PICO_NO_FLASH
    adr r4, data_cpy_table

    // assume there is at least one entry
1:
    ldmia r4!, {r1-r3}
    cmp r1, #0
    beq 2f
    bl data_cpy
    b 1b
2:
#endif
```

```
(gdb) x/36i 0x10000110
   ..
   0x10000164 <hold_non_core0_in_bootrom>:      mov.w   r0, #0
   0x10000168 <hold_non_core0_in_bootrom+4>:    b.n     0x10000150 <_enter_vtable_in_r0>
   0x1000016a <hold_non_core0_in_bootrom+6>:    add     r4, pc, #52     @ (adr r4, 0x100001a0
 <data_cpy_table>)
   0x1000016c <hold_non_core0_in_bootrom+8>:    ldmia   r4!, {r1, r2, r3}
   0x1000016e <hold_non_core0_in_bootrom+10>:   cmp     r1, #0
   0x10000170 <hold_non_core0_in_bootrom+12>:   beq.n   0x10000178
 <hold_non_core0_in_bootrom+20>
   0x10000172 <hold_non_core0_in_bootrom+14>:   bl      0x1000019a <data_cpy>
   0x10000176 <hold_non_core0_in_bootrom+18>:   b.n     0x1000016c
 <hold_non_core0_in_bootrom+8>
   0x10000178 <hold_non_core0_in_bootrom+20>:   ldr     r1, [pc, #84]   @ (0x100001d0
 <data_cpy_table+48>)
   0x1000017a <hold_non_core0_in_bootrom+22>:   ldr     r2, [pc, #88]   @ (0x100001d4
 <data_cpy_table+52>)
   0x1000017c <hold_non_core0_in_bootrom+24>:   movs    r0, #0
   0x1000017e <hold_non_core0_in_bootrom+26>:   b.n     0x10000182 <bss_fill_test>
   ..
```

This block funnels non-core0 straight back into the Boot ROM and then performs core0’s C-runtime staging: the label loads r0 with BOOTROM_VTABLE_OFFSET (in the build you’re disassembling it assembles to 0) and immediately branches to _enter_vtable_in_r0, which installs the Boot ROM’s vector table and jumps into its reset handler so secondary cores wait there until launched by core0; if we’re on core0, the code optionally stages and runs the boot2 XIP setup stub on core0’s stack (copy via data_cpy, then blx into it) to bring external flash online, then iterates the data_cpy_table with ldmia r4!, {r1-r3} until a zero sentinel in r1, copying each region described by the triples, and finally loads the .bss start/end from the literal pool, sets r0=0, and falls through to the bss zeroing routine.

```
    // Zero out the BSS
    ldr r1, =__bss_start__
    ldr r2, =__bss_end__
    movs r0, #0
    b bss_fill_test
bss_fill_loop:
    stm r1!, {r0}
bss_fill_test:
    cmp r1, r2
    bne bss_fill_loop
```

```
(gdb) x/36i 0x10000110
   ..
   0x10000180 <bss_fill_loop>:  stmia   r1!, {r0}
   0x10000182 <bss_fill_test>:  cmp     r1, r2
   0x10000184 <bss_fill_test+2>:        bne.n   0x10000180 <bss_fill_loop>
   ..
```

This is the RP2350's standard .bss zero-fill loop that runs during C runtime startup to ensure all uninitialized global/static variables start at zero, as required by the C standard. It loads __bss_start__ into r1 and __bss_end__ into r2, sets r0 to zero, then repeatedly executes stmia r1!, {r0} to store that zero word into memory and post-increment r1 to the next word. After each store, it compares r1 to r2; if they're not equal, it branches back to bss_fill_loop and continues until the entire .bss region is cleared. Once r1 reaches __bss_end__, the loop exits and the system can safely enter main() with all zero-initialized data in place.

```
platform_entry: // symbol for stack traces
#if PICO_CRT0_NEAR_CALLS && !PICO_COPY_TO_RAM
    bl runtime_init
```

```
(gdb) x/36i 0x10000110
   ..
   0x10000186 <platform_entry>: ldr     r1, [pc, #80]   @ (0x100001d8 <data_cpy_table+56>)
   0x10000188 <platform_entry+2>:       blx     r1
   0x1000018a <platform_entry+4>:       ldr     r1, [pc, #80]   @ (0x100001dc
<data_cpy_table+60>)
   0x1000018c <platform_entry+6>:       blx     r1
   0x1000018e <platform_entry+8>:       ldr     r1, [pc, #80]   @ (0x100001e0
<data_cpy_table+64>)
   0x10000190 <platform_entry+10>:      blx     r1
   0x10000192 <platform_entry+12>:      bkpt    0x0000
   0x10000194 <platform_entry+14>:      b.n     0x10000192 <platform_entry+12>
   0x10000196 <data_cpy_loop>:  ldmia   r1!, {r0}
   0x10000198 <data_cpy_loop+2>:        stmia   r2!, {r0}
   0x1000019a <data_cpy>:       cmp     r2, r3
   0x1000019c <data_cpy+2>:     bcc.n   0x10000196 <data_cpy_loop>
   0x1000019e <data_cpy+4>:     bx      lr
   0x100001a0 <data_cpy_table>: subs    r4, r1, r5
   0x100001a2 <data_cpy_table+2>:       asrs    r0, r0, #32
   0x100001a4 <data_cpy_table+4>:       lsls    r0, r2, #4
   0x100001a6 <data_cpy_table+6>:       movs    r0, #0
   0x100001a8 <data_cpy_table+8>:       lsls    r4, r5, #10
   0x100001aa <data_cpy_table+10>:      movs    r0, #0
   0x100001ac <data_cpy_table+12>:      adds    r0, r5, #3
   0x100001ae <data_cpy_table+14>:      asrs    r0, r0, #32
   0x100001b0 <data_cpy_table+16>:      movs    r0, r0
   0x100001b2 <data_cpy_table+18>:      movs    r0, #8
--Type <RET> for more, q to quit, c to continue without paging--
   0x100001b4 <data_cpy_table+20>:      movs    r0, r0
   0x100001b6 <data_cpy_table+22>:      movs    r0, #8
   0x100001b8 <data_cpy_table+24>:      adds    r0, r5, #3
   0x100001ba <data_cpy_table+26>:      asrs    r0, r0, #32
   0x100001bc <data_cpy_table+28>:      asrs    r0, r0, #32
   0x100001be <data_cpy_table+30>:      movs    r0, #8
   0x100001c0 <data_cpy_table+32>:      asrs    r0, r0, #32
   0x100001c2 <data_cpy_table+34>:      movs    r0, #8
   0x100001c4 <data_cpy_table+36>:      movs    r0, r0
   0x100001c6 <data_cpy_table+38>:      movs    r0, r0
   0x100001c8 <data_cpy_table+40>:      bx      lr
   0x100001ca <data_cpy_table+42>:      movs    r0, r0
   0x100001cc <data_cpy_table+44>:                      @ <UNDEFINED> instruction:
```

```
 0xed08e000
    0x100001d0 <data_cpy_table+48>:      lsls    r4, r5, #10
   0x100001d2 <data_cpy_table+50>:      movs    r0, #0
   0x100001d4 <data_cpy_table+52>:      lsls    r0, r3, #19
   0x100001d6 <data_cpy_table+54>:      movs    r0, #0
   0x100001d8 <data_cpy_table+56>:      asrs    r5, r7, #13
   0x100001da <data_cpy_table+58>:      asrs    r0, r0, #32
   0x100001dc <data_cpy_table+60>:      lsls    r5, r6, #8
   0x100001de <data_cpy_table+62>:      asrs    r0, r0, #32
   0x100001e0 <data_cpy_table+64>:      asrs    r5, r6, #13
   0x100001e2 <data_cpy_table+66>:      asrs    r0, r0, #32
   0x100001e4 <_init>:  push    {r3, r4, r5, r6, r7, lr}
   0x100001e6 <_init+2>:        nop
   0x100001e8 <register_tm_clones>:     ldr     r3, [pc, #24]   @ (0x10000204
<register_tm_clones+28>)
   0x100001ea <register_tm_clones+2>:   ldr     r1, [pc, #28]   @ (0x10000208
<register_tm_clones+32>)
   0x100001ec <register_tm_clones+4>:   subs    r1, r1, r3
   0x100001ee <register_tm_clones+6>:   asrs    r1, r1, #2
   0x100001f0 <register_tm_clones+8>:   it      mi
   0x100001f2 <register_tm_clones+10>:  addmi   r1, #1
   0x100001f4 <register_tm_clones+12>:  asrs    r1, r1, #1
   0x100001f6 <register_tm_clones+14>:  beq.n   0x10000200 <register_tm_clones+24>
   0x100001f8 <register_tm_clones+16>:  ldr     r3, [pc, #16]   @ (0x1000020c
<register_tm_clones+36>)
   0x100001fa <register_tm_clones+18>:  cbz     r3, 0x10000200 <register_tm_clones+24>
   0x100001fc <register_tm_clones+20>:  ldr     r0, [pc, #4]    @ (0x10000204
<register_tm_clones+28>)
   0x100001fe <register_tm_clones+22>:  bx      r3
--Type <RET> for more, q to quit, c to continue without paging--
   0x10000200 <register_tm_clones+24>:  bx      lr
   0x10000202 <register_tm_clones+26>:  nop
   0x10000204 <register_tm_clones+28>:  lsls    r4, r5, #10
   0x10000206 <register_tm_clones+30>:  movs    r0, #0
   0x10000208 <register_tm_clones+32>:  lsls    r4, r5, #10
   0x1000020a <register_tm_clones+34>:  movs    r0, #0
   0x1000020c <register_tm_clones+36>:  movs    r0, r0
   0x1000020e <register_tm_clones+38>:  movs    r0, r0
   0x10000210 <frame_dummy>:    push    {r3, lr}
   0x10000212 <frame_dummy+2>:  ldr     r3, [pc, #20]   @ (0x10000228 <frame_dummy+24>)
   0x10000214 <frame_dummy+4>:  cbz     r3, 0x1000021e <frame_dummy+14>
   0x10000216 <frame_dummy+6>:  ldr     r1, [pc, #20]   @ (0x1000022c <frame_dummy+28>)
   0x10000218 <frame_dummy+8>:  ldr     r0, [pc, #20]   @ (0x10000230 <frame_dummy+32>)
   0x1000021a <frame_dummy+10>: nop.w
   0x1000021e <frame_dummy+14>: ldmia.w sp!, {r3, lr}
   0x10000222 <frame_dummy+18>: b.w     0x100001e8 <register_tm_clones>
   0x10000226 <frame_dummy+22>: nop
   0x10000228 <frame_dummy+24>: movs    r0, r0
   0x1000022a <frame_dummy+26>: movs    r0, r0
   0x1000022c <frame_dummy+28>: lsls    r0, r2, #18
   0x1000022e <frame_dummy+30>: movs    r0, #0
   0x10000230 <frame_dummy+32>: adds    r4, r1, r7
   0x10000232 <frame_dummy+34>: asrs    r0, r0, #32
   ..
```

In the final linked binary, platform_entry has been expanded far beyond the single bl runtime_init you see in **crt0.S** as the compiler and linker have transformed that into a small call sequence that loads three function pointers from a nearby literal pool and calls them in turn. Those pointers, stored at data_cpy_table+56, +60, and +64, are filled in at link time with whatever initialization routines the Pico SDK and GCC's C runtime require. In a typical build, they correspond to the SDK's runtime_init, the standard __libc_init_array for running C++ constructors, and finally your application's main (or a wrapper). Using ldr/blx through a literal pool instead of a direct bl allows the linker to insert any combination of functions, handle long call distances, and keep the assembly source minimal.

Immediately after platform_entry is the data_cpy routine, a generic word-copy loop used earlier in startup to populate RAM sections from flash or other sources. It works by loading a word from the source pointer in r1, storing it to the destination in r2, and looping until r2 reaches the end address in r3. The label data_cpy_table that follows is not actually executable code, it's a block of constants the startup code uses. The first part holds triples of (source, destination, end) addresses for each region that needs copying. Later entries include other constants such as the VTOR register address (0xE000ED08 in little-endian form) and the .bss bounds, as well as the three function pointers used by platform_entry. GDB's disassembler shows these raw words as nonsensical Thumb instructions because it doesn't know they're data.

After this data region come a few standard GCC/EABI stubs: _init, register_tm_clones, and frame_dummy. These are pulled in automatically by the toolchain. _init is a hook for pre-main setup, often empty in embedded builds. The register_tm_clones and frame_dummy, are part of GCC's support for transactional memory and exception frame registration; on bare-metal targets they usually do nothing but are still linked in. Together, this sequence shows how a minimal assembly entry point in **crt0.S** grows into a fully linked startup chain, with the linker and runtime glue inserting the necessary initialization calls, memory setup routines, and housekeeping code before your program ever reaches main.

Chapter 5: Intro To Variables

In this chapter we are going to introduce the concept of a variable. If we have a series of boxes all laid out in a row and we numbered them from 0 to 9 (we start with 0 in Engineering) and then placed item 0 in box 0 and then item 1 in box 1 all the way to item 9 in box 9.

The boxes in this analogy represents our SRAM. The items are nothing more than variables of different types, which we will discuss later, that are stored in each of these addresses.

For the Developer, you simply provide a type and a name and the compiler will assign to the value to an actual address.

One of the most important considerations is that you have to declare variables before you use them in a program.

The process of declaration provides the compiler the size and name of the variable you are creating.

The process of definition allocates memory to a variable.

These two processes are usually done at the same time.

Let’s look at some code.

```
uint8_t age;
```

Here we have a data type which is *uint8_t* and the name of the variable which is age.

The data type determines how much space a variable is going to occupy in memory. This will signal the compiler to allocate space for it.

A semicolon signals to the compiler that a statement is complete. In our case the statement was the uint8_t age.

The uint8_t type takes up 1 byte of memory it is an unsigned integer type that can store a value between 0 and 255.

If you declare a value during declaration it is referred to as initialization.

Let’s open up our folder **0x0005_intro-to-variables**.

Now let’s review our **0x0005_intro-to-variables.c** file as this is located in the main folder.

```
#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);
}
```

Let’s flash the uf2 file onto the Pico 2. If you are unsure about this step, please take a look at Chapter 1 to get re-familiar with this process.

The first lines you should be familiar with and if not again refer to Chapter 1 to get re-familiar with those lines.

Let’s break down this code.

```
uint8_t age = 42;
```

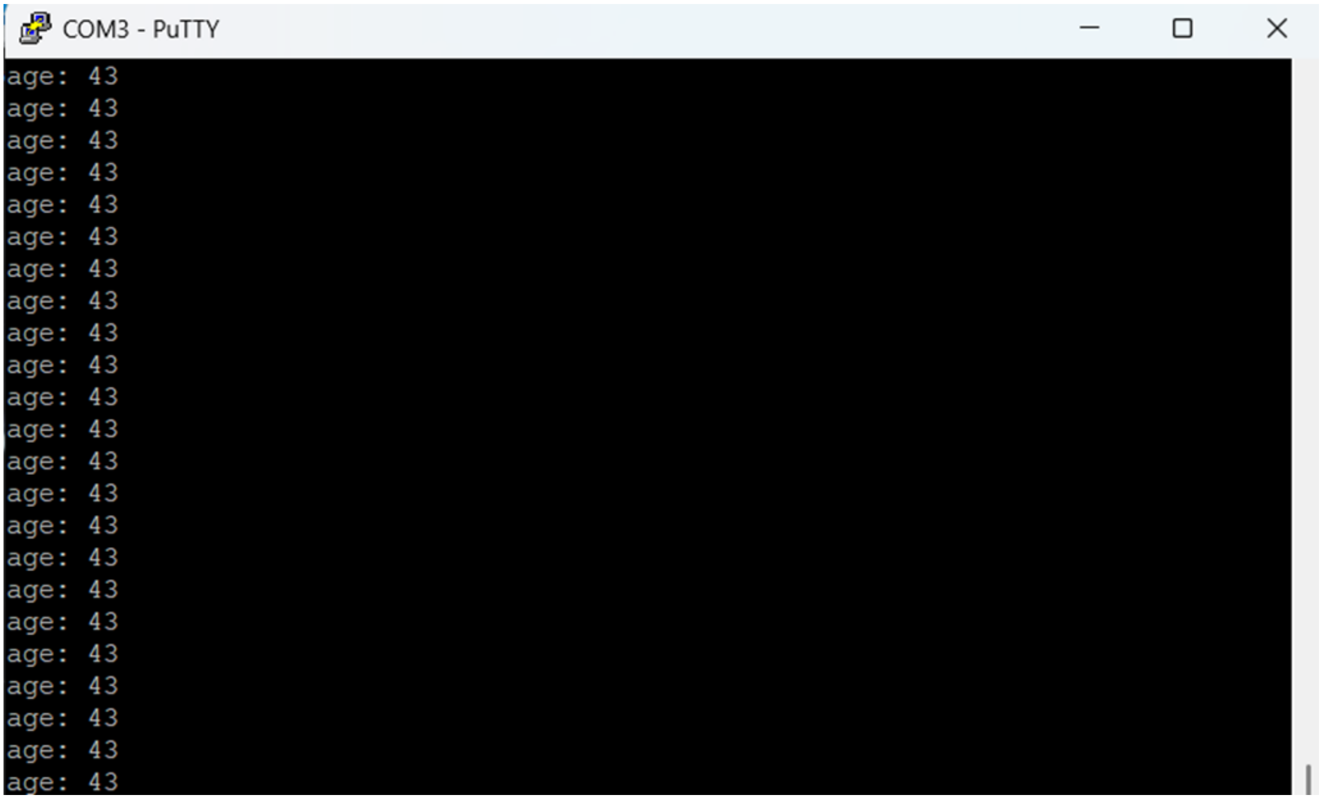
We start by declaring and initializing the variable to hold a 1-byte unsigned integer and assign the value of 42 to it.

```
age = 43;
```

We then change the value stored in age to 43.

Then inside the while loop we have a printf where we print text to indicate that we are going to print the age and then use what we refer to as a format specifier which is %d to indicate we are using a decimal value and then our new line chars \r\n and then we have the value that will populate %d which is 43.

Let’s open up PuTTY or your terminal editor of choice and we will see our values being printed in an infinite loop.



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

In our next chapter we will debug this.

Chapter 6: Debugging Intro To Variables

Today we debug!

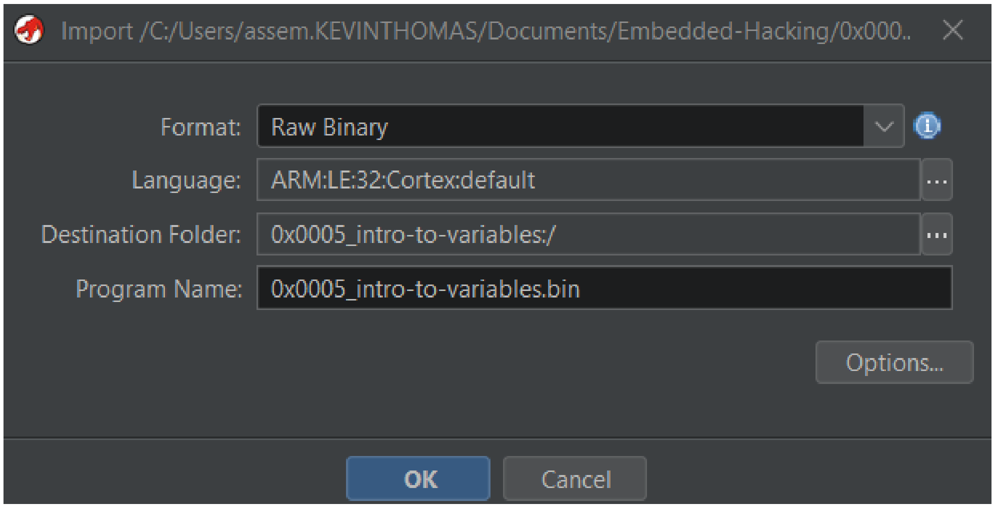
We will start with Ghidra.

Open up a terminal and run **ghidraRun** and when the window appears, we will select **File, New Project, Non-Shared Project, Next**, and create a **Project Name**. Here we will all it **0x0005_intro-to-variables** and press **Finish.**

Open the file explorer and navigate to the **Embedded-Hacking** folder and drag-and-drop the **0x0005_intro-to-variables.bin** file into the folder within the Ghidra application panel.

In the small window that appears, you will see the file identified as a BIN, which is a binary format without symbols. We will be using the BIN format going forward as this is what we would normally see in the wild so there will be additional setup required based on what we have learned so far.

The window will show a Raw Binary format. Here we click on the three dots to the right of Language and search for Cortex. We want to select Cortex little endian default and click **Ok.**



Import /C:/Users/assem.KEVINTHOMAS/Documents/Embedded-Hacking/0x000..

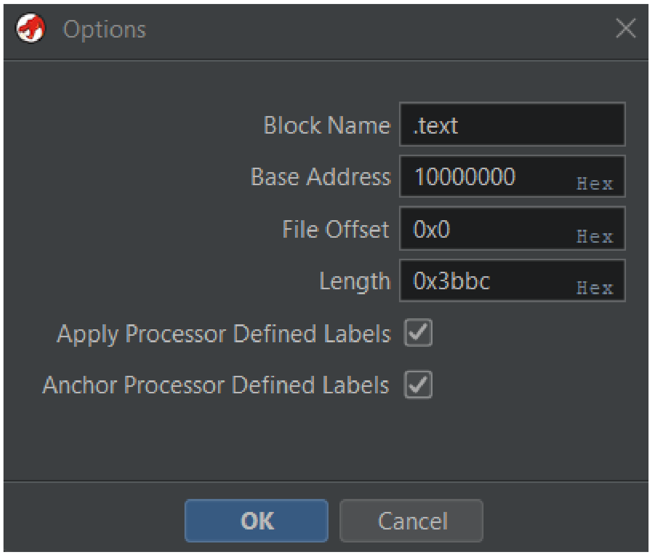
Format: Raw Binary

Language: ARM:LE:32:Cortex:default

Destination Folder: 0x0005_intro-to-variables:/

Program Name: 0x0005_intro-to-variables.bin

Click on the **Options…** button. Change the Block Name to .text and the base address to XIP which is 10000000 hex and click **Ok.**



Options

Block Name: .text

Base Address: 10000000 Hex

File Offset: 0x0 Hex

Length: 0x3bbc Hex

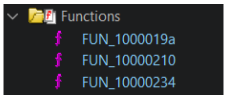
Apply Processor Defined Labels ☑

Anchor Processor Defined Labels ☑

Let’s double-click on the file within the window.

Finally click the auto-analyze and let’s begin reviewing the binary.

Let’s look at the Functions in the Symbol Tree.



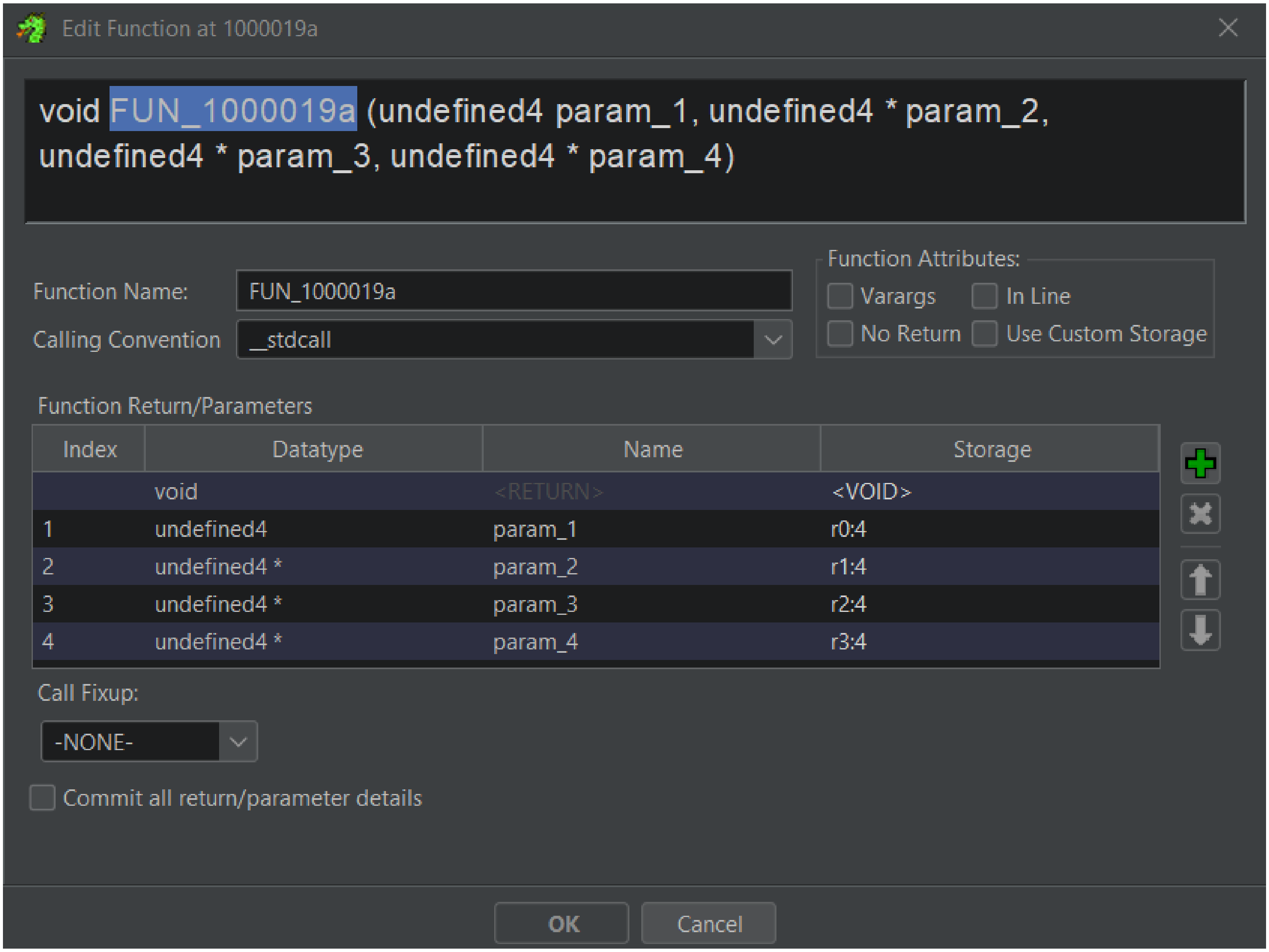
Functions

- FUN_1000019a
- FUN_10000210
- FUN_10000234

Remember back to Chapter 4, what function existed at 0x1000019a?

The answer is data_cpy, so now we can resolve this symbol in Ghidra.

Click on FUN_1000019a, in the Decompile view, click on the function name and right-click and select **Edit Function Signature**.



Edit Function at 1000019a

```
void FUN_1000019a (undefined4 param_1, undefined4 * param_2, undefined4 * param_3, undefined4 * param_4)
```

Function Name: FUN_1000019a

Calling Convention: __stdcall

Function Attributes:

□ Varargs □ In Line

□ No Return □ Use Custom Storage

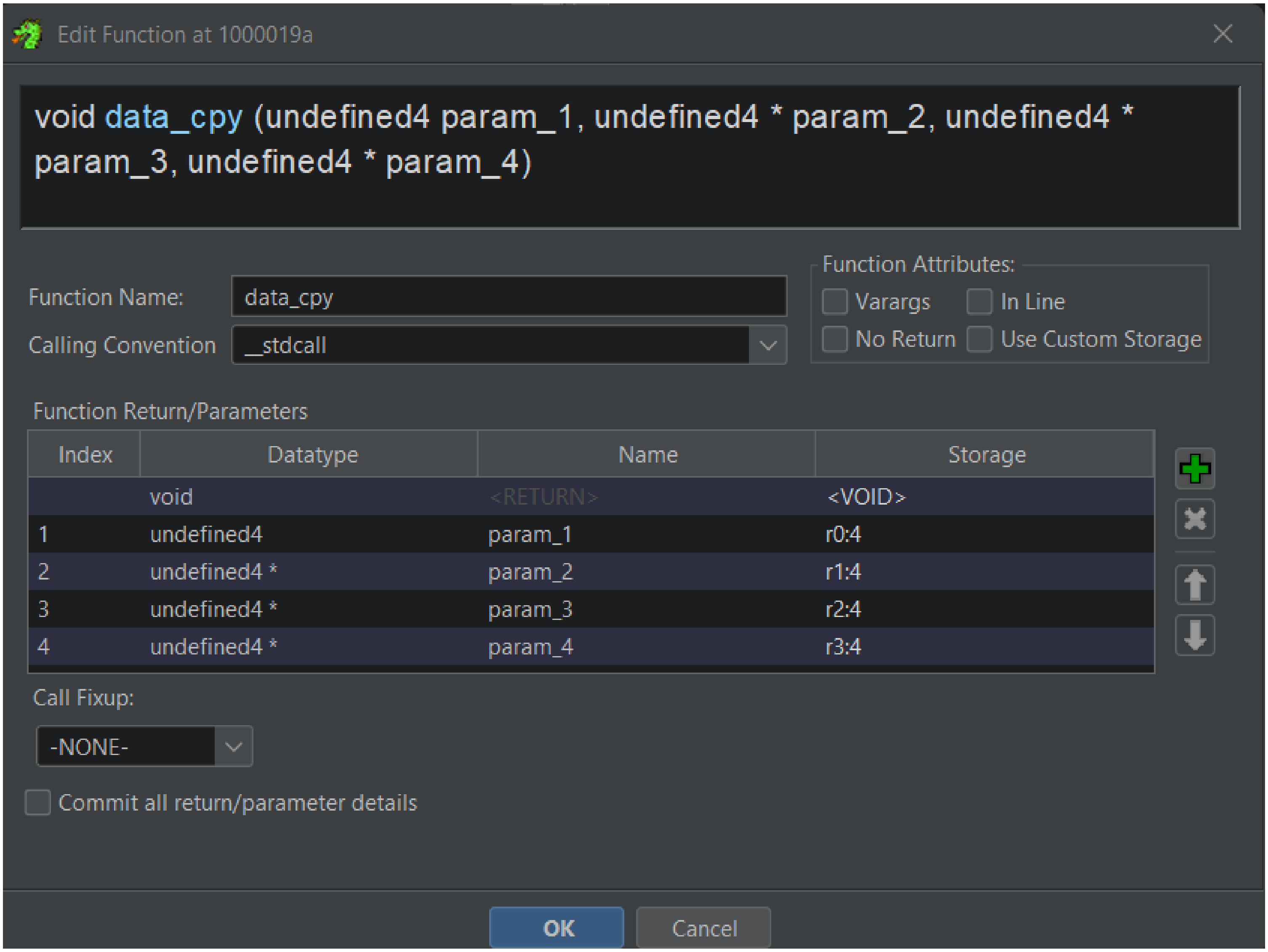
Function Return/Parameters

Index	Datatype	Name	Storage
	void	<RETURN>	<VOID>
1	undefined4	param_1	r0:4
2	undefined4 *	param_2	r1:4
3	undefined4 *	param_3	r2:4
4	undefined4 *	param_4	r3:4

Call Fixup: -NONE-

□ Commit all return/parameter details

Update this to `data_cpy` then click **Ok.**



void data_cpy (undefined4 param_1, undefined4 * param_2, undefined4 * param_3, undefined4 * param_4)

Function Name: data_cpy

Calling Convention: __stdcall

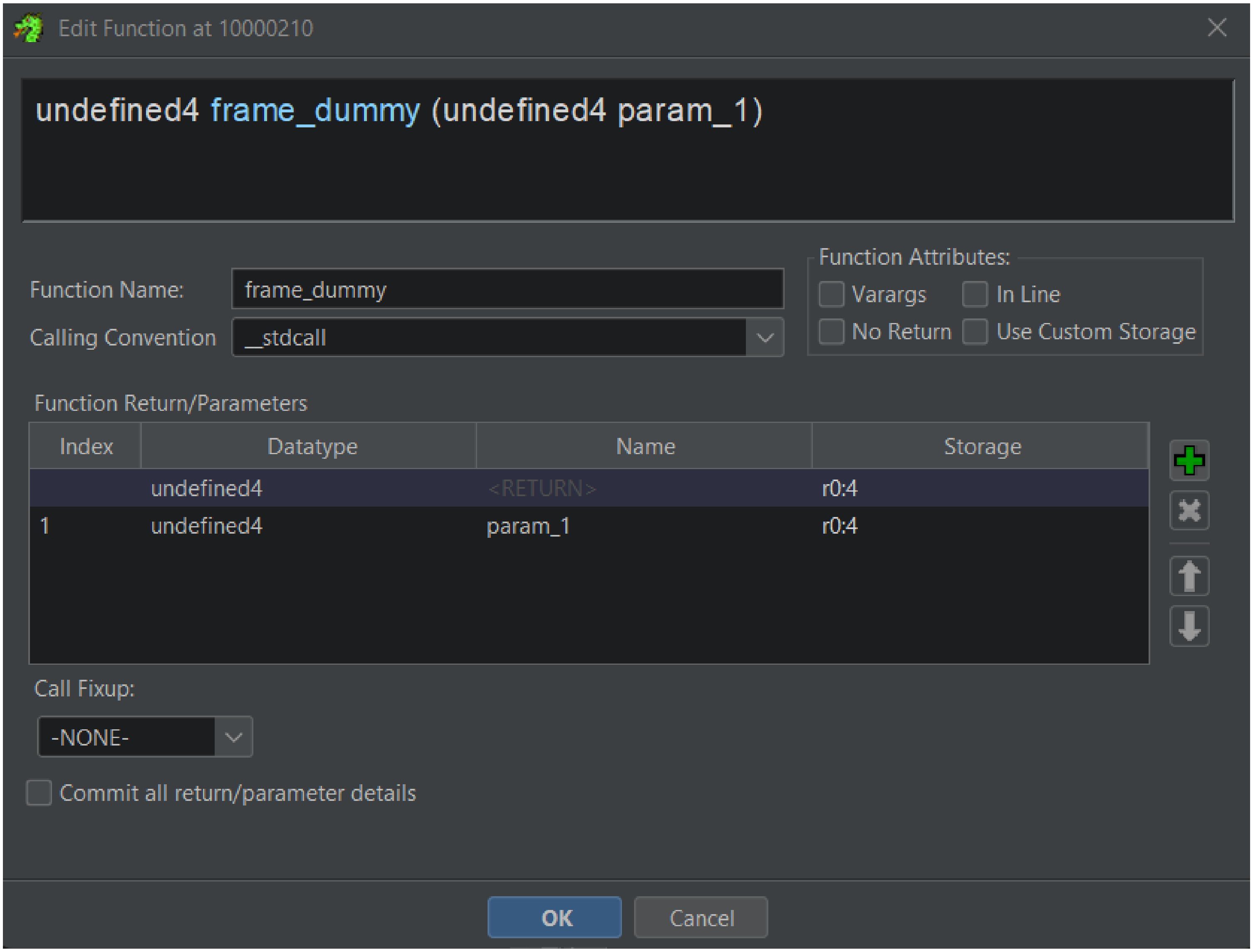
Function Return/Parameters

Index	Datatype	Name	Storage
	void	<RETURN>	<VOID>
1	undefined4	param_1	r0:4
2	undefined4 *	param_2	r1:4
3	undefined4 *	param_3	r2:4
4	undefined4 *	param_4	r3:4

Call Fixup: -NONE-

In Chapter 4, what was the function at `FUN_10000210`

The answer is `frame_dummy` so let's update that function as well then click **Ok.**



undefined4 frame_dummy (undefined4 param_1)

Function Name: frame_dummy

Calling Convention: __stdcall

Function Attributes:

□ Varargs □ In Line

□ No Return □ Use Custom Storage

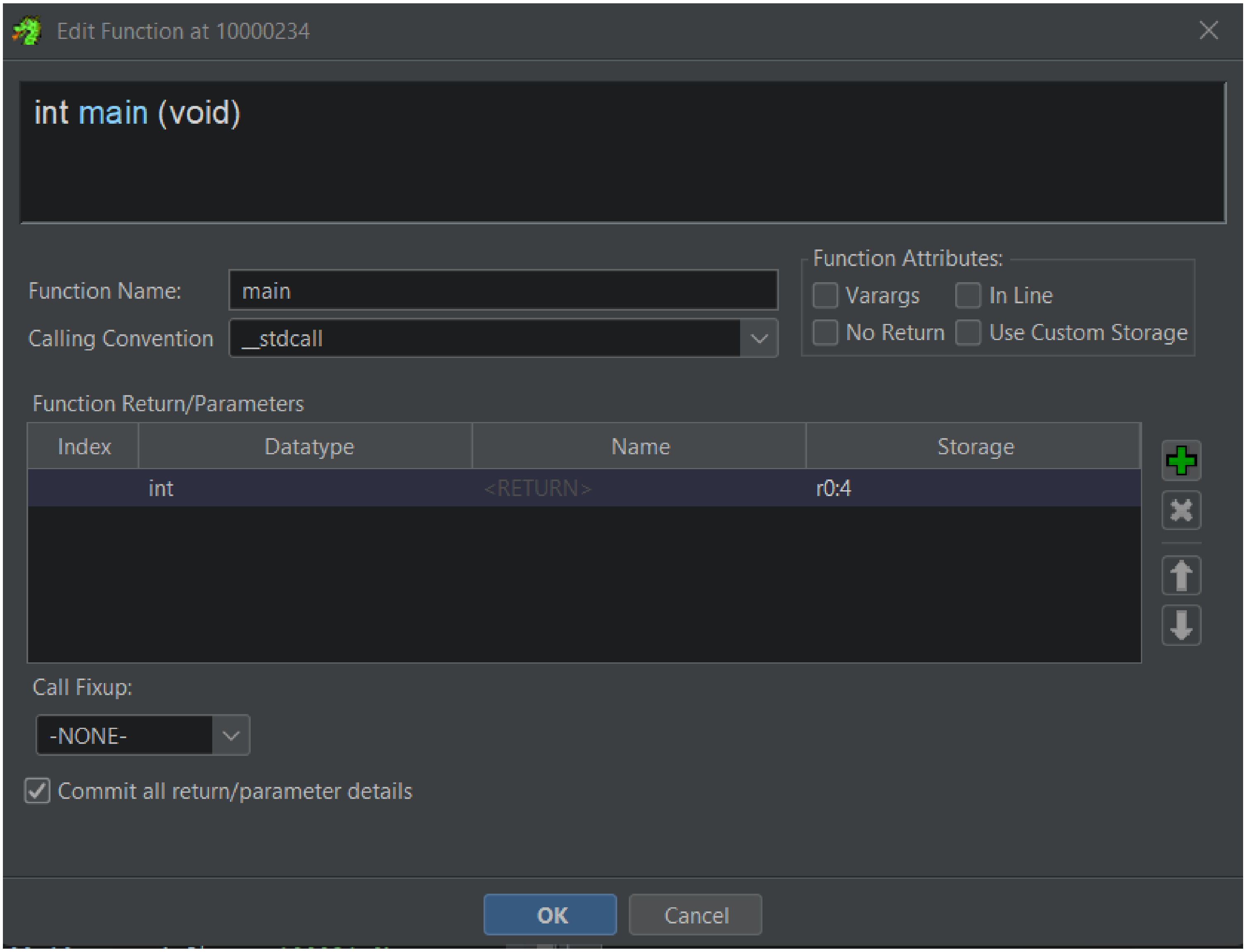
Function Return/Parameters

Index	Datatype	Name	Storage
	undefined4	<RETURN>	r0:4
1	undefined4	param_1	r0:4

Call Fixup: -NONE-

□ Commit all return/parameter details

The final function we will resolve is main then click **Ok**.



Edit Function at 10000234

int main (void)

Function Name: main

Calling Convention: __stdcall

Function Attributes:

□ Varargs □ In Line

□ No Return □ Use Custom Storage

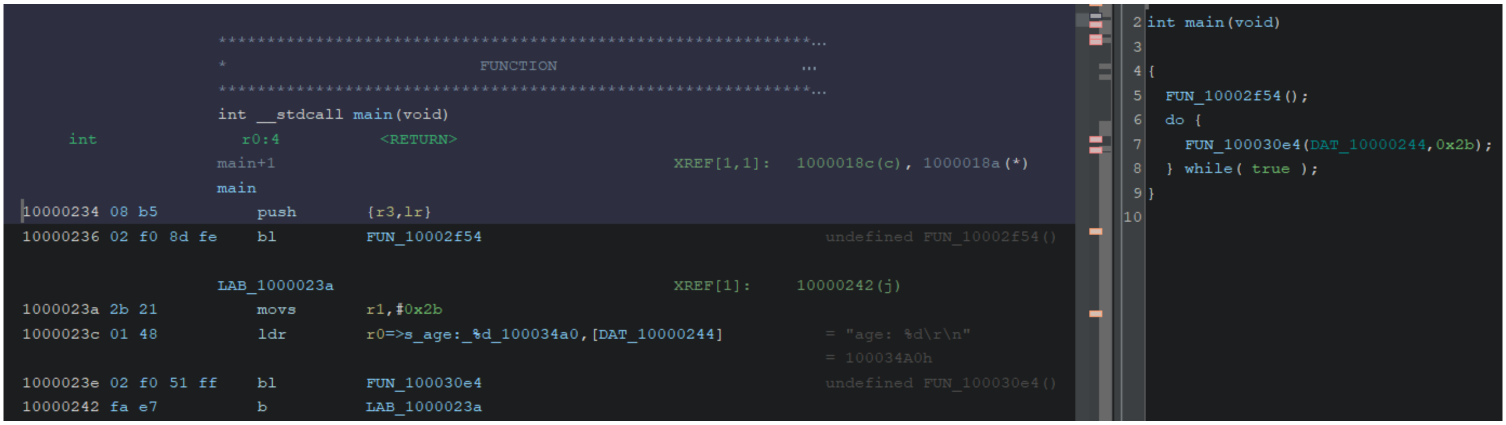
Function Return/Parameters

Index	Datatype	Name	Storage
	int	<RETURN>	r0:4

Call Fixup: -NONE-

☑ Commit all return/parameter details

Let’s review the assembler and decompile views.

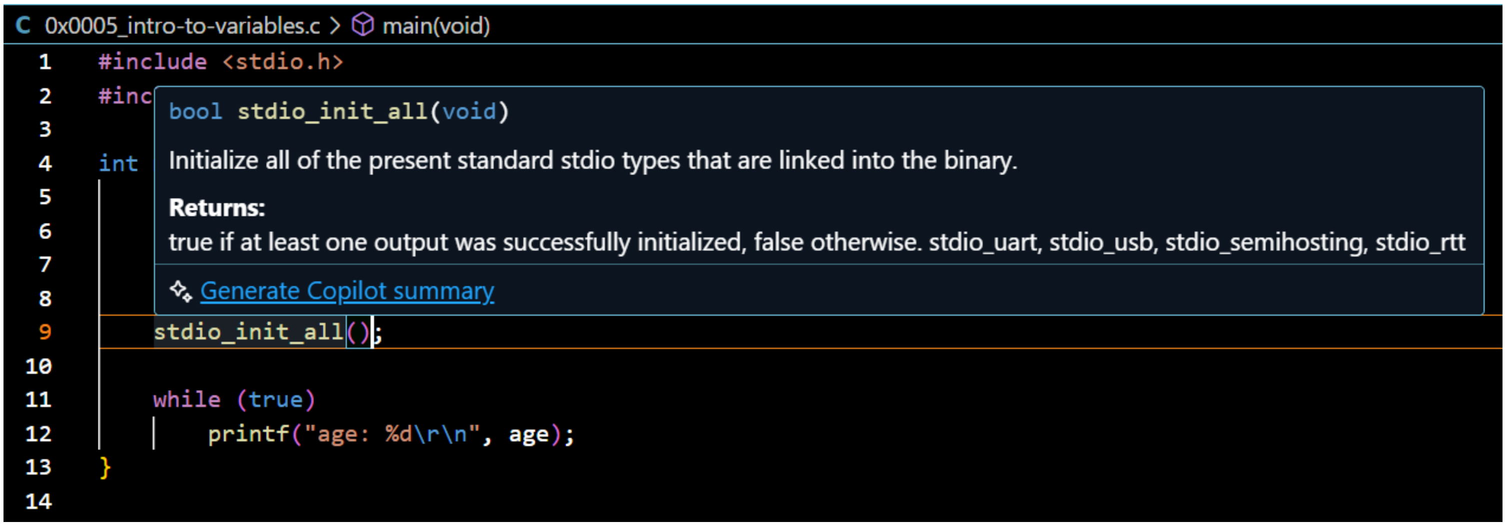


```
                    *******************************************************************...
                    *                          FUNCTION                              ...
                    *******************************************************************...
                    int __stdcall main(void)
     int              r0:4           <RETURN>
                    main+1                                      XREF[1,1]:   1000018c(c), 1000018a(*)
                    main
10000234 08 b5          push       {r3,lr}
10000236 02 f0 8d fe    bl         FUN_10002f54                              undefined FUN_10002f54()

                    LAB_1000023a                                XREF[1]:     10000242(j)
1000023a 2b 21          movs       r1,#0x2b
1000023c 01 48          ldr        r0=>s_age:_%d_100034a0,[DAT_10000244]     = "age: %d\r\n"
                                                                             = 100034A0h
1000023e 02 f0 51 ff    bl         FUN_100030e4                              undefined FUN_100030e4()
10000242 fa e7          b          LAB_1000023a
```

```
int main(void)

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

We see two more functions that needs to be resolved. The first one is the Pico C SDK stdio_init_all.

If we review our source code, we see that the function returns a bool.



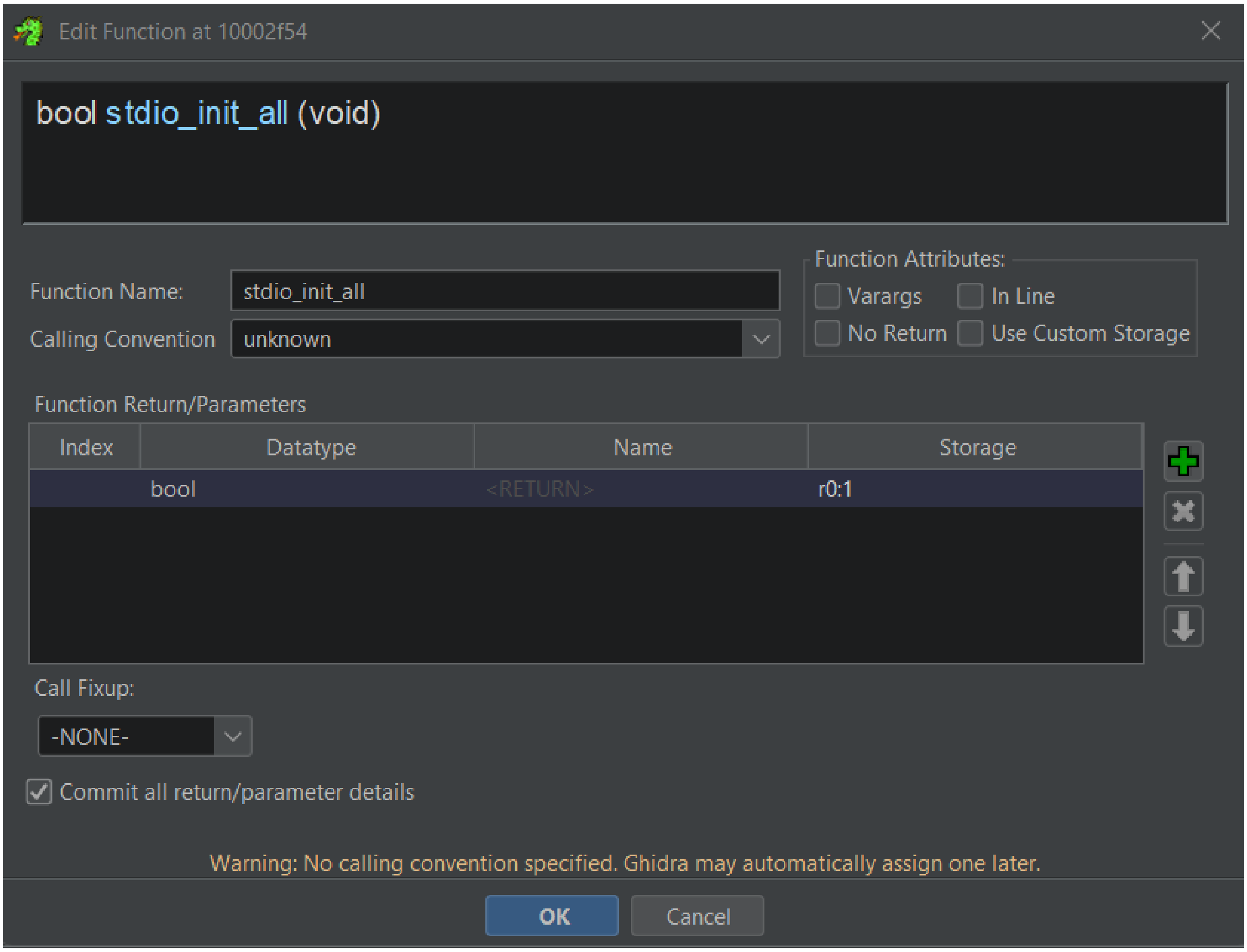
```
#include <stdio.h>
#inc

int

    stdio_init_all();

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

bool stdio_init_all(void)

Initialize all of the present standard stdio types that are linked into the binary.

Returns:
true if at least one output was successfully initialized, false otherwise. stdio_uart, stdio_usb, stdio_semihosting, stdio_rtt

Therefore, we need to update accordingly and click **Ok**.



Edit Function at 10002f54

bool stdio_init_all (void)

Function Name: stdio_init_all

Calling Convention: unknown

Function Attributes:
□ Varargs □ In Line
□ No Return □ Use Custom Storage

Function Return/Parameters

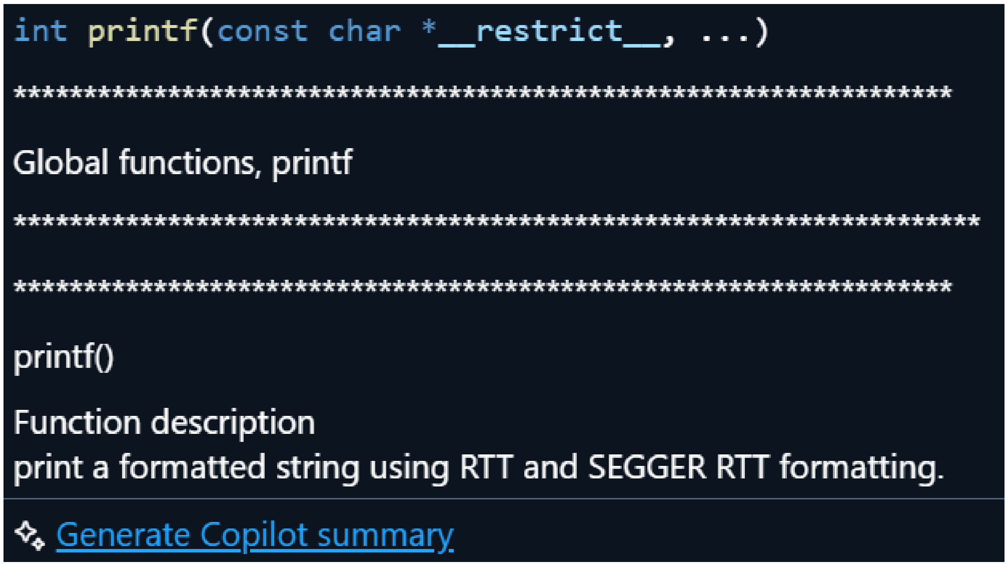
Index	Datatype	Name	Storage
	bool	<RETURN>	r0:1

Call Fixup: -NONE-

☑ Commit all return/parameter details

Warning: No calling convention specified. Ghidra may automatically assign one later.

The other function we have to resolve is `printf`.



```
int printf(const char *__restrict__, ...)
```

****

Global functions, printf

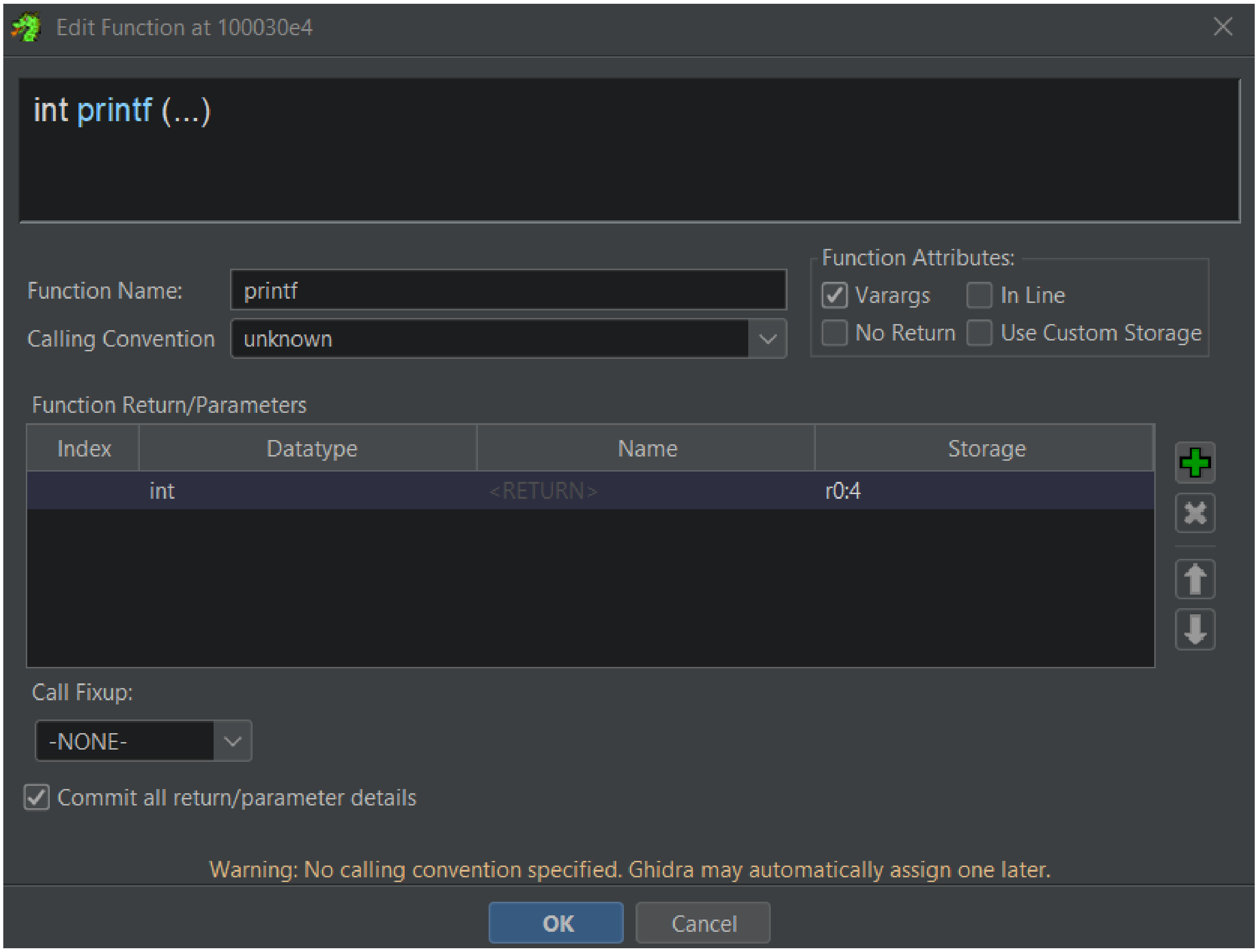
****

****

printf()

Function description
print a formatted string using RTT and SEGGER RTT formatting.

Here you want to select Varargs for variadic args as `printf` can take any number of args and click **Ok.**



Edit Function at 100030e4

```
int printf (...)
```

Function Name: printf

Calling Convention: unknown

Function Attributes:

☑ Varargs □ In Line

□ No Return □ Use Custom Storage

Function Return/Parameters

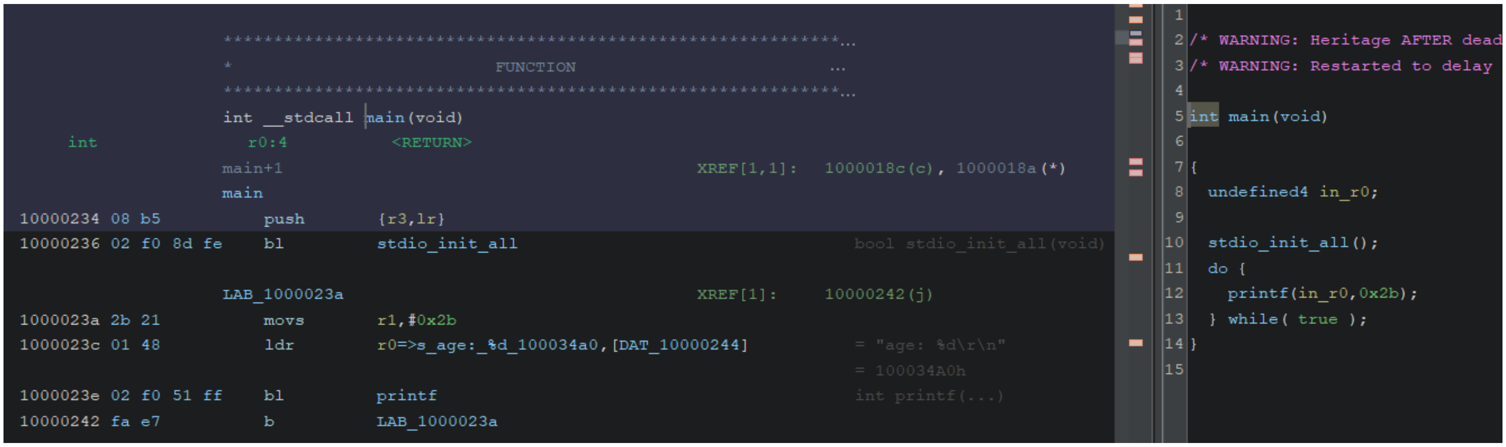
Index	Datatype	Name	Storage
	int	<RETURN>	r0:4

Call Fixup: -NONE-

☑ Commit all return/parameter details

Warning: No calling convention specified. Ghidra may automatically assign one later.

Let’s reexamine our assembler and de-compilation.



```
                         ****************************************************************...
                         *                          FUNCTION                            ...
                         ****************************************************************...
                         int __stdcall main(void)
     int                   r0:4            <RETURN>
                         main+1                                  XREF[1,1]:   1000018c(c), 1000018a(*)
                         main
10000234 08 b5               push       {r3,lr}
10000236 02 f0 8d fe         bl         stdio_init_all                              bool stdio_init_all(void)

                         LAB_1000023a                            XREF[1]:     10000242(j)
1000023a 2b 21               movs       r1,#0x2b
1000023c 01 48               ldr        r0=>s_age:_%d_100034a0,[DAT_10000244]       = "age: %d\r\n"
                                                                                     = 100034A0h
1000023e 02 f0 51 ff         bl         printf                                      int printf(...)
10000242 fa e7               b          LAB_1000023a
```

```

/* WARNING: Heritage AFTER dead
/* WARNING: Restarted to delay

int main(void)

{
  undefined4 in_r0;

  stdio_init_all();
  do {
    printf(in_r0,0x2b);
  } while( true );
}

```

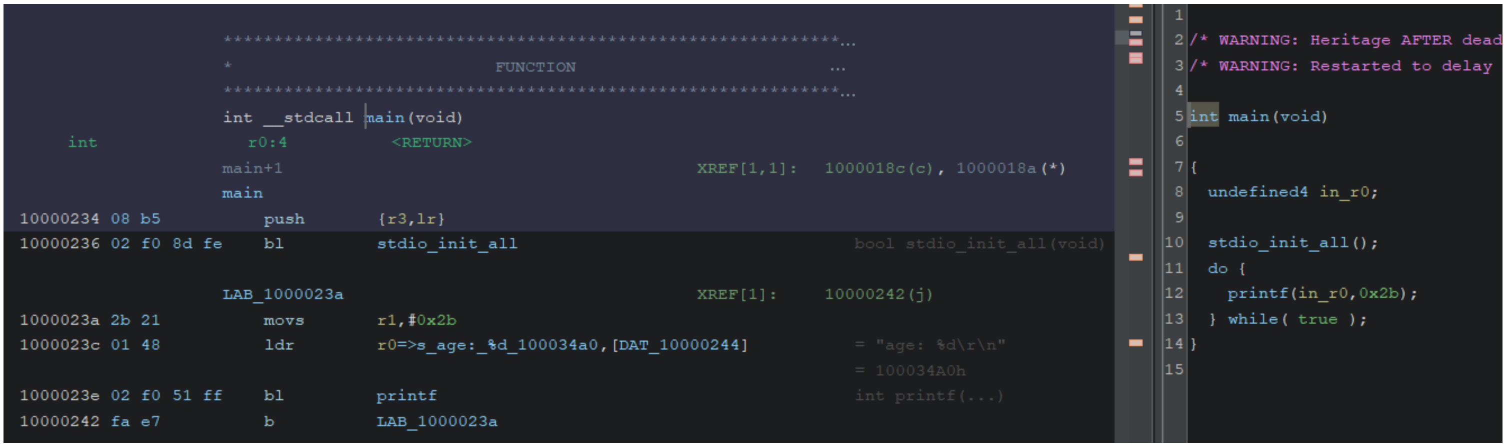
We know that 0x2b in hex is 43. We can always double-check with the ascii table that we have worked with previously.

Take note that the initialization of `uint8_t age = 42` was optimized out by the compiler so we are only seeing 43 which the original code was `age = 43`.

In our next lesson we will hack this!

Chapter 7: Hacking Intro To Variables

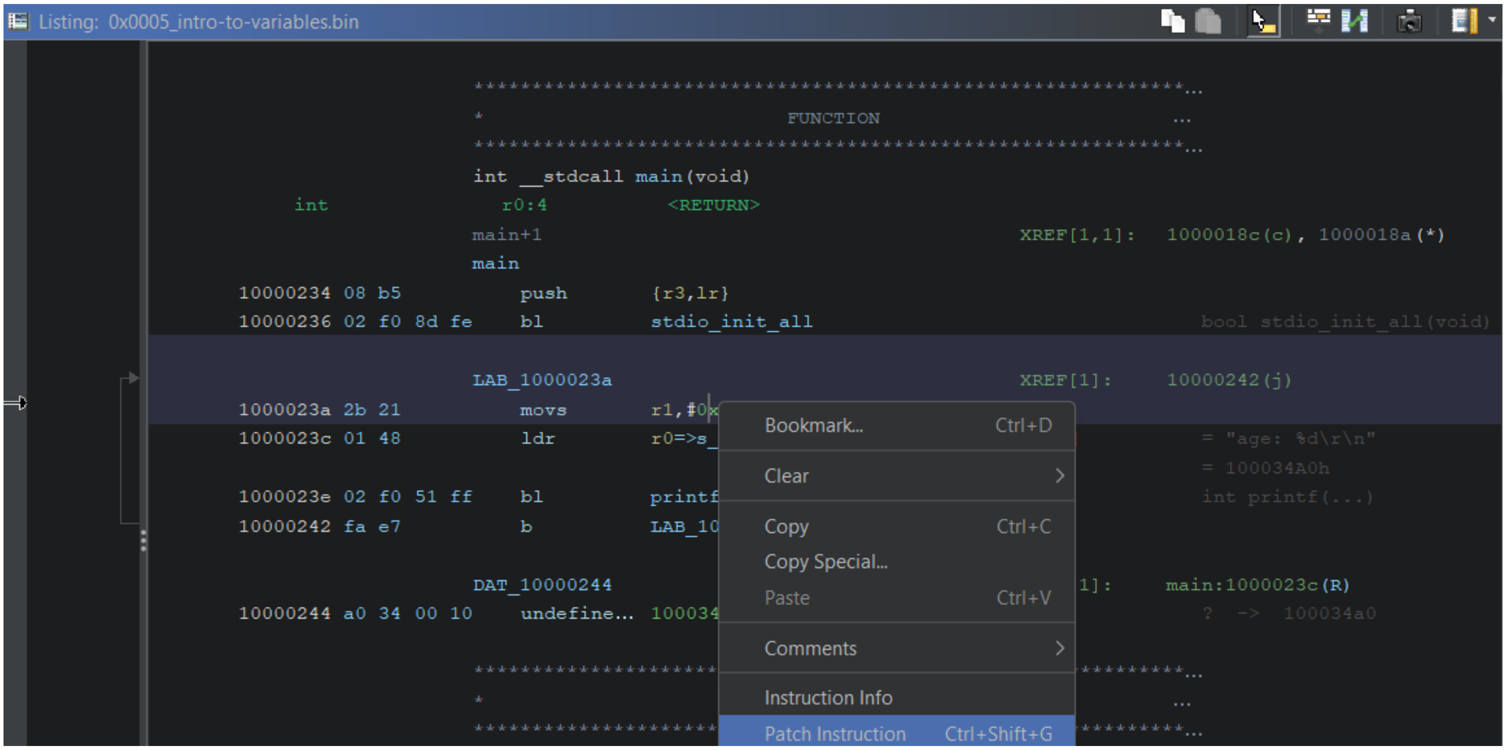
Let’s continue where we were in Ghidra from our last chapter.



```
                    ****************************************************************...
                    *                            FUNCTION                              ...
                    ****************************************************************...
                    int __stdcall main(void)
       int             r0:4          <RETURN>
                    main+1                                   XREF[1,1]:   1000018c(c), 1000018a(*)
                    main
10000234 08 b5           push       {r3,lr}
10000236 02 f0 8d fe     bl         stdio_init_all                         bool stdio_init_all(void)

                    LAB_1000023a                             XREF[1]:     10000242(j)
1000023a 2b 21           movs       r1,#0x2b
1000023c 01 48           ldr        r0=>s_age:_%d_100034a0,[DAT_10000244]       = "age: %d\r\n"
                                                                                = 100034A0h
1000023e 02 f0 51 ff     bl         printf                                 int printf(...)
10000242 fa e7           b          LAB_1000023a
```

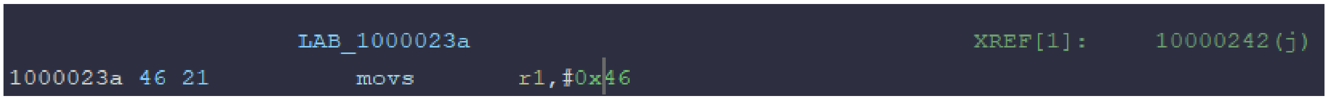
```

/* WARNING: Heritage AFTER dead
/* WARNING: Restarted to delay

int main(void)

{
  undefined4 in_r0;

  stdio_init_all();
  do {
    printf(in_r0,0x2b);
  } while( true );
}

```

Let’s hack 0x2b to 0x46! Highlight 0x2b and right-click and select **Patch Instruction.**



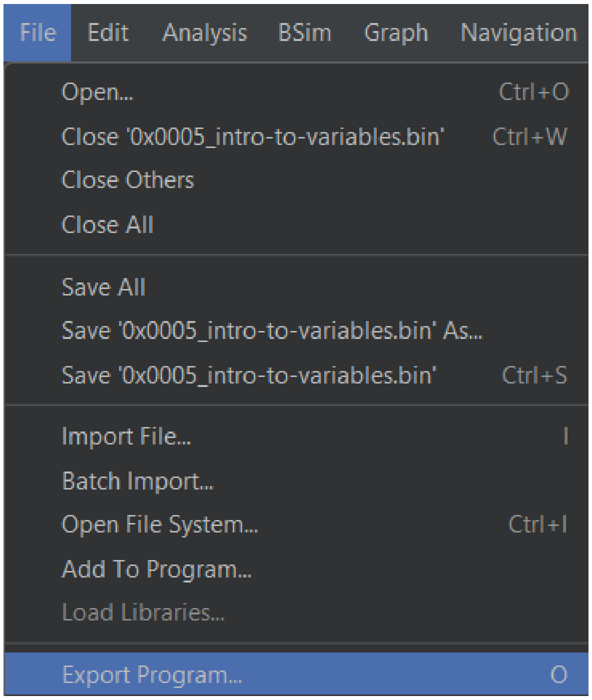
Listing: 0x0005_intro-to-variables.bin

```
                    ****************************************************************...
                    *                            FUNCTION                              ...
                    ****************************************************************...
                    int __stdcall main(void)
        int            r0:4           <RETURN>
                    main+1                                   XREF[1,1]:   1000018c(c), 1000018a(*)
                    main
   10000234 08 b5           push       {r3,lr}
   10000236 02 f0 8d fe     bl         stdio_init_all                      bool stdio_init_all(void)

                    LAB_1000023a                             XREF[1]:     10000242(j)
   1000023a 2b 21           movs       r1,#0x
   1000023c 01 48           ldr        r0=>s_                                  = "age: %d\r\n"
                                                                               = 100034A0h
   1000023e 02 f0 51 ff     bl         printf                                  int printf(...)
   10000242 fa e7           b          LAB_10

                    DAT_10000244                             1]:    main:1000023c(R)
   10000244 a0 34 00 10     undefine...  100034                        ?  ->  100034a0

                    *********************                    *********...
                    *                                                  ...
                    *********************                    *********...
```

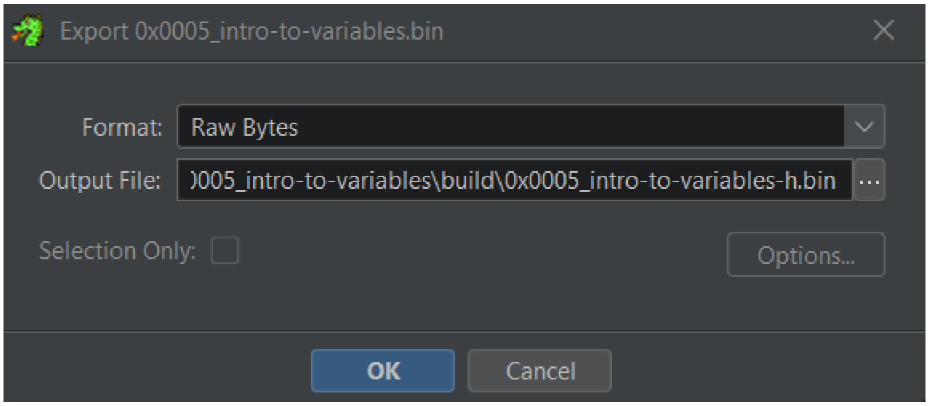


```
                    LAB_1000023a                                XREF[1]:     10000242(j)
1000023a 46 21          movs        r1,#0x46
```

Let’s export the hacked bin.



Select **Raw Bytes** as a **Format** and put the file in the **0x0005_intro-to-variables** bin directory and name the new bin **0x0005_intro-to-variables-h.bin** and click **Ok.**



Export 0x0005_intro-to-variables.bin

Format: Raw Bytes

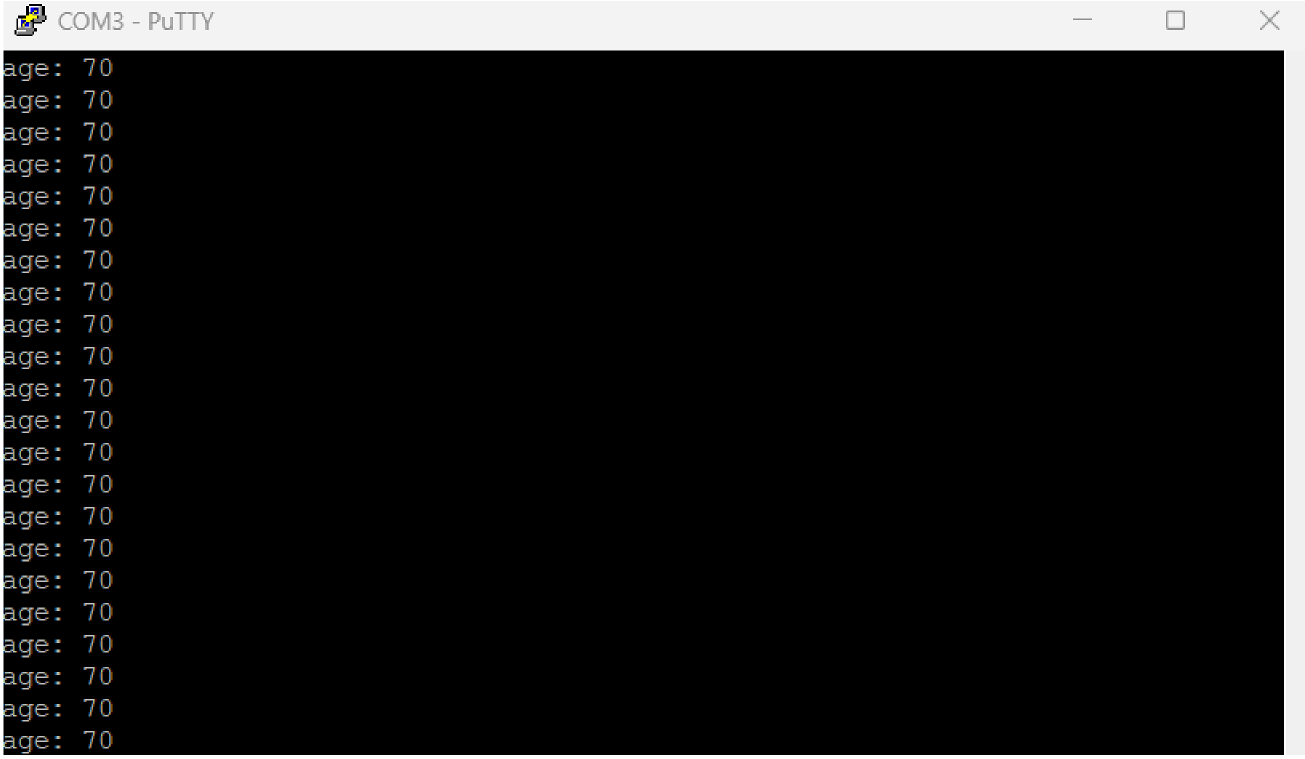
Output File:)005_intro-to-variables\build\0x0005_intro-to-variables-h.bin

Selection Only: □

We need to use a tool to convert this hacked binary into the UF2 format.

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

After flashing the hacked.uf2 to the Pico 2, we see the following in the serial terminal.



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

Boom! We hacked it!

In the coming chapters we will see a great deal of repetition so to some of you this may be a bit boring however to the majority I hope this helps to reinforce techniques that will help you beyond this course as an embedded reverse engineer.

Chapter 8: Uninitialized Variables

In this chapter we are going to examine what happens in memory when we create variables that are not initialized.

We will also introduce the RP2350 GPIO or general-purpose input/output by toggling our red LED on GPIO16.

Let’s open up our folder **0x0008_unitialized-variables**.

Now let’s review our **0x0008_unitialized-variables.c** file as this is located in the main folder.

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

#define LED_PIN 16

int main(void) {
    uint8_t age;

    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);
    }
}
```

Let’s flash the uf2 file onto the Pico. If you are unsure about this step, please take a look at Chapter 1 to get re-familiar with this process.

The only difference is that we have no idea what the value will be inside of `age` or do we?

In other versions of C you would see garbage data if a value is uninitialized however what we see in the C Pico SDK is that like other modern compilers, if you have a value that is not initialized, it will get assigned to the `.bss` section of memory.

The entire .bss section is assigned an address in RAM via the linker and does not reside in the binary or flash.

When the Pico boots, behind the scenes memset which is a C standard lib function is zeroing out the entire .bss so this is why these values are in fact 0.

When you initialize a variable, it will go into the .data section.

When you initialize a constant it will go into the .rodata section.

Let's look at what stdio_init_all is behind the scenes.

```
bool stdio_init_all(void) {
    // todo add explicit custom, or registered although you can call stdio_enable_driver explicitly anyway
    // These are well known ones

    bool rc = false;
#if LIB_PICO_STDIO_UART
    stdio_uart_init();
    rc = true;
#endif

#if LIB_PICO_STDIO_SEMIHOSTING
    stdio_semihosting_init();
    rc = true;
#endif

#if LIB_PICO_STDIO_RTT
    stdio_rtt_init();
    rc = true;
#endif

#if LIB_PICO_STDIO_USB
    rc |= stdio_usb_init();
#endif
    return rc;
}
```

The gpio_init function prepares the chosen pin for use, and gpio_set_dir configures it as an output so it can drive the LED. Inside the main loop, gpio_put is called with a value of 1 to switch the LED on and with 0 to switch it off. A call to sleep_ms is added between these operations to create a visible delay, producing the familiar blink effect at a human-perceivable rate.

Let’s review our other 4 functions within the Pico C SDK.

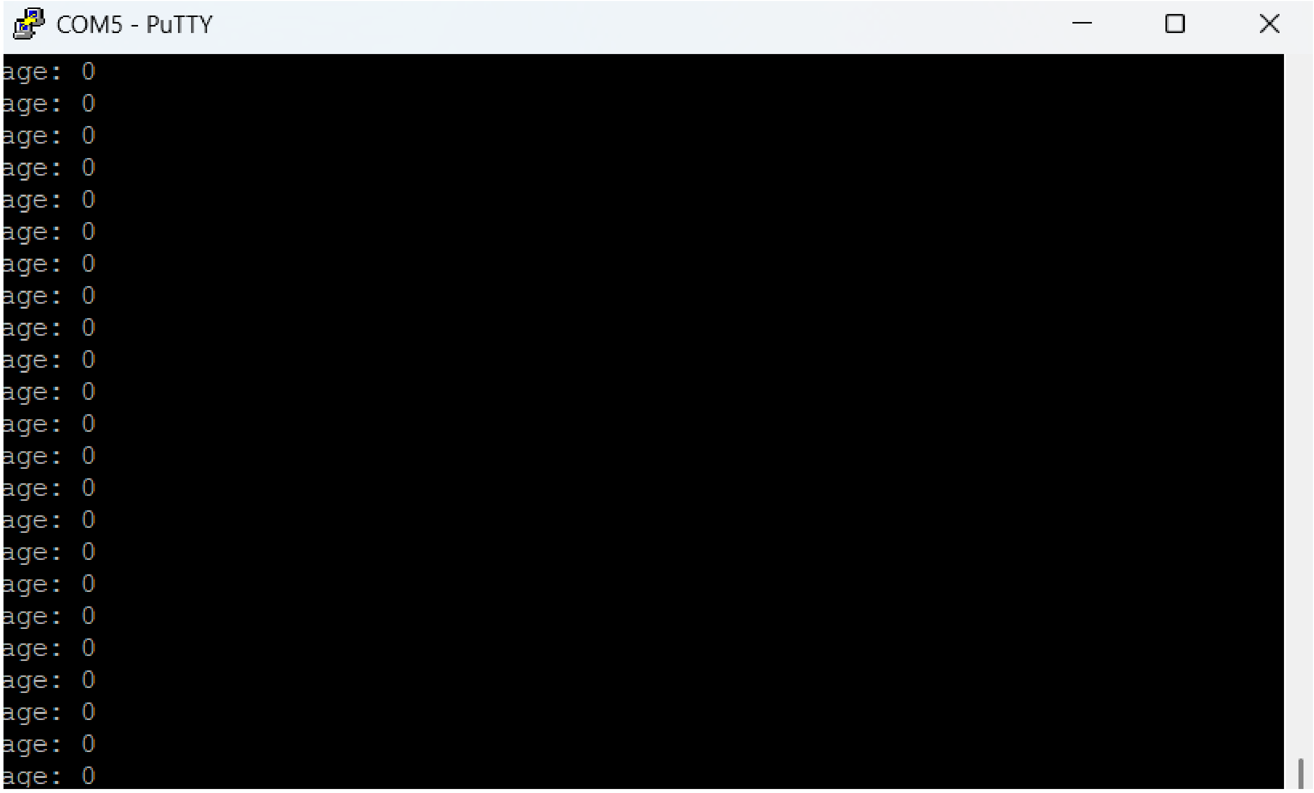
```
void gpio_init(uint gpio) {
    gpio_set_dir(gpio, GPIO_IN);
    gpio_put(gpio, 0);
    gpio_set_function(gpio, GPIO_FUNC_SIO);
}
```

```
static inline void gpio_set_dir(uint gpio, bool out) {
#if PICO_USE_GPIO_COPROCESSOR
    gpioc_bit_oe_put(gpio, out);
#elif PICO_RP2040 || NUM_BANK0_GPIOS <= 32
    uint32_t mask = 1ul << gpio;
    if (out)
        gpio_set_dir_out_masked(mask);
    else
        gpio_set_dir_in_masked(mask);
#else
    uint32_t mask = 1u << (gpio & 0x1fu);
    if (gpio < 32) {
        if (out) {
            sio_hw->gpio_oe_set = mask;
        } else {
            sio_hw->gpio_oe_clr = mask;
        }
    } else {
        if (out) {
            sio_hw->gpio_hi_oe_set = mask;
        } else {
            sio_hw->gpio_hi_oe_clr = mask;
        }
    }
#endif
}
```

```
static inline void gpio_put(uint gpio, bool value) {
#if PICO_USE_GPIO_COPROCESSOR
    gpioc_bit_out_put(gpio, value);
#elif NUM_BANK0_GPIOS <= 32
    uint32_t mask = 1ul << gpio;
    if (value)
        gpio_set_mask(mask);
    else
        gpio_clr_mask(mask);
#else
    uint32_t mask = 1ul << (gpio & 0x1fu);
    if (gpio < 32) {
        if (value) {
            sio_hw->gpio_set = mask;
        } else {
            sio_hw->gpio_clr = mask;
        }
    } else {
        if (value) {
            sio_hw->gpio_hi_set = mask;
        } else {
            sio_hw->gpio_hi_clr = mask;
        }
    }
#endif
}
```

```
void sleep_ms(uint32_t ms) {
    sleep_us(ms * 1000ull);
}
```

Let’s flash and examine the binary. We also see the red LED blinking.



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

In our next lesson we will debug this.

Chapter 9: Debugging Uninitialized Variables

Today we debug!

We will start with Ghidra.

Open up a terminal and run **ghidraRun** and when the window appears, we will select **File, New Project, Non-Shared Project, Next**, and create a **Project Name**. Here we will all it **0x0008_unitialized-variables** and press **Finish**.

Open the file explorer and navigate to the **Embedded-Hacking** folder and drag-and-drop the **0x0008_unitialized-variables.bin** file into the folder within the Ghidra application panel.

In the small window that appears, you will see the file identified as a BIN, which is a binary format without symbols. We will be using the BIN format going forward as this is what we would normally see in the wild so there will be additional setup required based on what we have learned so far.

The window will show a Raw Binary format. Here we click on the three dots to the right of Language and search for Cortex. We want to select Cortex little endian default and click **Ok**.

We will skip all of the Ghira setup as these are detailed in Chapter 6.

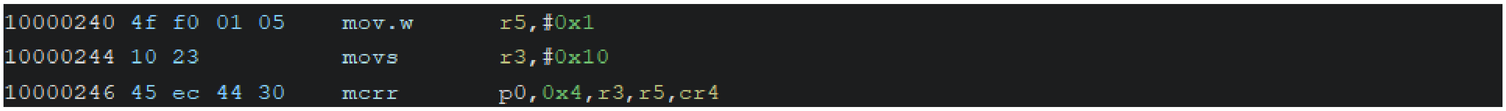
First, we need to set up our Cortex little-endian and options to the .text section to 0x10000000.

We then auto-analyze the binary and set up the memory map as well.

We then update our function signature of int main(void) at FUN_10000234.

We then update our function signature of bool stdio_init_all(void) at FUN_100030cc.

We then update our function signature of void gpio_init(uint gpio) at FUN_100002b4.



```
10000240 4f f0 01 05    mov.w      r5,#0x1
10000244 10 23          movs       r3,#0x10
10000246 45 ec 44 30    mcrr       p0,0x4,r3,r5,cr4
```

The `gpioc_bit_out_put` is a tiny, always-inlined helper that atomically sets or clears a single GPIO by emitting a coprocessor instruction: it calls `pico_default_asm_volatile("mcrr p0, #4, %0, %1, c0" : : "r"(pin), "r"(val))`, passing the pin number and the boolean value to the RP2 GPIO coprocessor; the effect is equivalent to "`if (val) gpioc_hilo_out_set(1ull << pin); else gpioc_hilo_out_clr(1ull << pin)`", so a true value sets the pin, false clears it, and the operation happens in one atomic coprocessor-backed cycle.

```
static inline void gpio_set_dir(uint gpio, bool out) {
#if PICO_USE_GPIO_COPROCESSOR
    gpioc_bit_oe_put(gpio, out);
#elif PICO_RP2040 || NUM_BANK0_GPIOS <= 32
    uint32_t mask = 1ul << gpio;
    if (out)
        gpio_set_dir_out_masked(mask);
    else
        gpio_set_dir_in_masked(mask);
#else
    uint32_t mask = 1u << (gpio & 0x1fu);
    if (gpio < 32) {
        if (out) {
            sio_hw->gpio_oe_set = mask;
        } else {
            sio_hw->gpio_oe_clr = mask;
        }
    } else {
        if (out) {
            sio_hw->gpio_hi_oe_set = mask;
        } else {
            sio_hw->gpio_hi_oe_clr = mask;
        }
    }
#endif
}
```

We have worked with Ghidra and GDB. Let’s take another perspective working with our GPIO and LED by first looking at that raw code.

We have **0x0008_unitialized-variables-a.c** so let’s review.

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

#define LED_PIN 16

int main(void)
{
    gpio_init(LED_PIN);
    gpio_set_dir(LED_PIN, GPIO_OUT);

    while (true) {
        gpio_put(LED_PIN, 1);
        sleep_ms(500);

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

Let’s take a step deeper.

We have **0x0008_unitialized-variables-b.c** so let’s review.

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

#define LED_PIN 16

int main(void)
{
    //   gpio_init(LED_PIN);
    gpio_set_dir(LED_PIN, GPIO_IN);
    gpio_put(LED_PIN, 0);
    gpio_set_function(LED_PIN, GPIO_FUNC_SIO);

    //   gpio_set_dir(LED_PIN, GPIO_OUT);
    gpioc_bit_oe_put(LED_PIN, GPIO_OUT);

    while (true) {
        //   gpio_put(LED_PIN, 1);
        gpioc_bit_out_put(LED_PIN, 1);
        //   sleep_ms(500);
        sleep_us(500 * 1000ull);

        //   gpio_put(LED_PIN, 0);
        gpioc_bit_out_put(LED_PIN, 0);
        //   sleep_ms(500);
        sleep_us(500 * 1000ull);
    }
}
```

Here we see some additional internal sdk functions. Let’s dig deeper to have a better understanding of what is going on.

We have **0x0008_unitialized-variables-c.c** so let’s review.

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

#define LED_PIN 16

int main(void)
{
    //   gpio_init(LED_PIN);
    ///  gpio_set_dir(LED_PIN, GPIO_IN);
    gpioc_bit_oe_put(LED_PIN, GPIO_OUT);
    ///  gpio_put(LED_PIN, 0);
    gpioc_bit_out_put(LED_PIN, 0);
    ///  gpio_set_function(LED_PIN, GPIO_FUNC_SIO);
    hw_write_masked(&pads_bank0_hw->io[LED_PIN],
                   PADS_BANK0_GPIO0_IE_BITS,
                   PADS_BANK0_GPIO0_IE_BITS | PADS_BANK0_GPIO0_OD_BITS
    );
    io_bank0_hw->io[LED_PIN].ctrl = GPIO_FUNC_SIO << IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB;
    hw_clear_bits(&pads_bank0_hw->io[LED_PIN], PADS_BANK0_GPIO0_ISO_BITS);

    //   gpio_set_dir(LED_PIN, GPIO_OUT);
    gpioc_bit_oe_put(LED_PIN, GPIO_OUT);

    while (true) {
        //   gpio_put(LED_PIN, 1);
        gpioc_bit_out_put(LED_PIN, 1);
        //   sleep_ms(500);
        sleep_us(500 * 1000ull);

        //   gpio_put(LED_PIN, 0);
        gpioc_bit_out_put(LED_PIN, 0);
        //   sleep_ms(500);
        sleep_us(500 * 1000ull);
    }
}
```

Here we see some more lower-level activity that we will review at the assembler level.

We have **0x0008_unitialized-variables-b.d** so let’s review.

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

#define LED_PIN 16

int main(void)
{
    //   gpio_init(LED_PIN);
    ///  gpio_set_dir(LED_PIN, GPIO_IN);
    //// gpioc_bit_oe_put(LED_PIN, GPIO_OUT);
    pico_default_asm_volatile ("mcrr p0, #4, %0, %1, c4" : : "r" (LED_PIN), "r"
(GPIO_OUT));
    ///  gpio_put(LED_PIN, 0);
    //// gpioc_bit_out_put(LED_PIN, 0);
    pico_default_asm_volatile ("mcrr p0, #4, %0, %1, c4" : : "r" (LED_PIN), "r"
(GPIO_OUT));
    ///  gpio_set_function(LED_PIN, GPIO_FUNC_SIO);
    //// hw_write_masked(&pads_bank0_hw->io[LED_PIN],
    ////                PADS_BANK0_GPIO0_IE_BITS,
    ////                PADS_BANK0_GPIO0_IE_BITS | PADS_BANK0_GPIO0_OD_BITS
    //// );
    //// hw_xor_bits(addr, (*addr ^ values) & write_mask);
    pico_default_asm_volatile (
        "ldr r2, [%0]\n"                     // load current pad register
        "eor r2, r2, %1\n"                   // xor with IE bit
        "and r2, r2, %2\n"                   // mask with (IE|OD)
        "eor r2, r2, %1\n"                   // recombine (hw_xor_bits logic)
        "str r2, [%0]\n"                     // write back
        :
        : "r" (&pads_bank0_hw->io[LED_PIN]),
        "r" (PADS_BANK0_GPIO0_IE_BITS),
        "r" (PADS_BANK0_GPIO0_IE_BITS | PADS_BANK0_GPIO0_OD_BITS)
        : "r2", "memory"
    );

    ///  io_bank0_hw->io[LED_PIN].ctrl = GPIO_FUNC_SIO << IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB;
    pico_default_asm_volatile (
        "str %1, [%0]\n"
        :
        : "r" (&io_bank0_hw->io[LED_PIN].ctrl),
        "r" (GPIO_FUNC_SIO << IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB)
        : "memory"
    );
    ///  hw_clear_bits(&pads_bank0_hw->io[LED_PIN], PADS_BANK0_GPIO0_ISO_BITS);
    pico_default_asm_volatile (
        "ldr r2, [%0]\n"                     // load current register value
        "bic r2, r2, %1\n"                   // clear the ISO bits (bit clear)
        "str r2, [%0]\n"                    // write back
        :
        : "r" (&pads_bank0_hw->io[LED_PIN]),
        "r" (PADS_BANK0_GPIO0_ISO_BITS)
        : "r2", "memory"
    );
    //   gpio_set_dir(LED_PIN, GPIO_OUT);
    ///  gpioc_bit_oe_put(LED_PIN, GPIO_OUT);
    pico_default_asm_volatile ("mcrr p0, #4, %0, %1, c4" : : "r" (LED_PIN), "r"
(GPIO_OUT));

    while (true) {
        //   gpio_put(LED_PIN, 1);
        ///  gpioc_bit_out_put(LED_PIN, 1);
        pico_default_asm_volatile ("mcrr p0, #4, %0, %1, c0" : : "r" (LED_PIN), "r" (1));
        ///  sleep_ms(500);
        sleep_us(500 * 1000ull);

        //   gpio_put(LED_PIN, 0);
        ///  gpioc_bit_out_put(LED_PIN, 0);
        pico_default_asm_volatile ("mcrr p0, #4, %0, %1, c0" : : "r" (LED_PIN), "r" (0));
        ///  sleep_ms(500);
        sleep_us(500 * 1000ull);
    }
}
```

Here we start to dive into assembler, lets review in our next and final deeper dive.

We have **0x0008_unitialized-variables-e.e** so let’s review.

```
int main(void) {
    __asm__ volatile (
        //   gpio_init(LED_PIN);
        ///  gpio_set_dir(LED_PIN, GPIO_IN);
        //// gpioc_bit_oe_put(LED_PIN, GPIO_OUT);
        "movs r4, #0x10\n"                   // GPIO16
        "movs r5, #0x01\n"                   // bit 1; used for OUT/OE writes
        "mcrr p0, #4, r4, r5, c4\n"         // gpioc_bit_oe_put(16, 1); p102

        //   gpio_set_function(LED_PIN, GPIO_FUNC_SIO);
        ///  hw_write_masked(&pads_bank0_hw->io[LED_PIN],
        ///                 PADS_BANK0_GPIO0_IE_BITS,
        ///                 PADS_BANK0_GPIO0_IE_BITS | PADS_BANK0_GPIO0_OD_BITS
        ///  );
        //// hw_xor_bits(addr, (*addr ^ values) & write_mask);
        "ldr r3, =0x40038044\n"             // &pads_bank0_hw->io[16]; p785, p796
        "ldr r2, [r3]\n"                    // load current config
        "bic r2, r2, #0x80\n"               // clear OD; output disable
        "orr r2, r2, #0x40\n"               // set IE; enable input buffer
        "str r2, [r3]\n"                    // store updated config
        ///  io_bank0_hw->io[LED_PIN].ctrl = GPIO_FUNC_SIO <<
IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB;
        "ldr r3, =0x40028084\n"             // &io_bank0_hw->io[16].ctrl; p603, p637
        "ldr r2, [r3]\n"                    // load current config
        "bic r2, r2, #0x1f\n"               // clear FUNCSEL bits [4:0]
        "orr r2, r2, #5\n"                  // set FUNCSEL = 5 (SIO)
        "str r2, [r3]\n"                    // store updated config
        ///  hw_clear_bits(&pads_bank0_hw->io[gpio], PADS_BANK0_GPIO0_ISO_BITS);
        "ldr r3, =0x40038044\n"             // &pads_bank0_hw->io[16]; p785, p796
        "ldr r2, [r3]\n"                    // load current config
        "bic r2, r2, #0x100\n"              // clear ISO bit (bit 8) un-isolate pad
        "str r2, [r3]\n"                    // store updated config

        //   gpio_set_dir(LED_PIN, GPIO_OUT);
        ///  gpioc_bit_oe_put(LED_PIN, GPIO_OUT);
        "movs r4, #0x10\n"                   // GPIO16
        "movs r5, #0x01\n"                   // bit 1; used for OUT/OE writes
        "mcrr p0, #4, r4, r5, c4\n"         // gpioc_bit_oe_put(16, 1); p102

        //   while (true)
        "1:\n"                              // loop start

        //   gpio_put(LED_PIN, 1);
        ///  gpioc_bit_out_put(LED_PIN, 1);
        "movs r4, #0x10\n"                   // GPIO16
        "movs r5, #0x01\n"                   // bit 1; used for OUT/OE writes
        "mcrr p0, #4, r4, r5, c0\n"         // gpioc_bit_out_put(16, 1)
        //   sleep_ms(500);
        ///  sleep_us(500 * 1000ull);
        "ldr r2, =0x17D7840\n"              // r2 = ~8.4M cycles
        "2:\n"                              // delay loop
        "subs r2, r2, #1\n"                 // decrement counter
        "bne 2b\n"                          // repeat until zero
        //   gpio_put(LED_PIN, 1);
        ///  gpioc_bit_out_put(LED_PIN, 1);
        "movs r4, #0x10\n"                  // GPIO16
        "movs r5, #0x00\n"                  // bit 0; used for OUT/OE writes
        "mcrr p0, #4, r4, r5, c0\n"         // gpioc_bit_out_put(16, 0)
        //   sleep_ms(500);
        ///  sleep_us(500 * 1000ull);
        "ldr r2, =0x17D7840\n"              // r2 = ~8.4M cycles
        "3:\n"                              // delay loop
        "subs r2, r2, #1\n"                 // decrement counter
        "bne 3b\n"                          // repeat until zero

        //    jmp
        "b 1b\n"                            // repeat forever
    );
}
```

This code demonstrates a complete GPIO blink implementation for the RP2350 microcontroller using inline assembly with GPIO coprocessor instructions. The program initializes GPIO pin 16 as an output and creates an infinite loop that toggles the pin state every 500 milliseconds. The code leverages RP2350-specific features, particularly the GPIO coprocessor accessible through `mcrr p0, #4` instructions, which provides efficient hardware-accelerated GPIO control that wasn't available on the earlier RP2040 chip.

The initialization sequence follows the standard GPIO setup pattern but uses direct register manipulation for maximum performance. First, it configures the pad control register at address `0x40038044 (PADS_BANK0_BASE + 0x44 for GPIO16)` to enable input buffering and disable output disable functionality. Then it sets the GPIO function to SIO (Software I/O) by writing value 5 to the `IO_BANK0` control register at `0x40028084`, and removes pad isolation by clearing bit 8. Finally, it uses the GPIO coprocessor to enable output mode for the pin.

The main blink loop demonstrates precise timing control using a software delay loop calibrated for the RP2350's 150MHz system clock. The delay value `0x17D7840` (approximately 25 million iterations) is calculated to produce a 500ms delay, accounting for the loop overhead of roughly 3 CPU cycles per iteration. The loop alternates between setting GPIO16 high and low using the coprocessor instructions `mcrr p0, #4, r4, r5, c0`, where the coprocessor efficiently handles the bit manipulation without requiring traditional memory-mapped I/O operations. This approach provides deterministic timing and minimal CPU overhead compared to using the standard SDK GPIO functions.

In our next chapter we will hack this.