



FIRST EDITION – Chapter 9

Kevin Thomas
Copyright © 2023 My Techno Talent

Forward

I remember when I started learning programming to which my first language was 6502 Assembler. It was to program a Commodore 64 and right from the beginning 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.

Everything we did we had an absolute mastery of 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 W microcontroller.

Below are items you will need for this book.

Raspberry Pi Pico W

<https://www.amazon.com/Pre-Soldered-Raspberry-Dual-Core-Processor-Dual-core/dp/B0BLZ26S6>

Raspberry Pi Pico W Debug Probe

<https://www.amazon.com/GeeekPi-Raspberry-Connector-RP2040-Microcontroller/dp/B0C5XNQ7FD>

Electronics Soldering Iron Kit

<https://www.amazon.com/Electronics-Adjustable-Temperature-ControlledThermostatic/dp/B0B28JQ95M?th=1>

Premium Breadboard Jumper Wires

<https://www.amazon.com/Keszoox-Premium-Breadboard-Jumper-Raspberry/%20dp/B09F6X3N79>

Breadboard Kit

<https://www.amazon.com/Breadboards-Solderless-BreadboardDistribution-Connecting/dp/B07DL13RZH>

6x6x5mm Momentary Tactile Tact Push Button Switches

<https://www.amazon.com/DAOKI-Miniature-Momentary-Tactile-Quality/dp/B01CGMP9GY>

WS2812 Neopixel Array

<https://www.amazon.com/BTF-LIGHTING-Individual-Addressable-Flexible-Controllers/dp/B088BTXHRG>

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
Chapter	2: Debugging hello, world
Chapter	3: Hacking hello, world
Chapter	4: Embedded System Analysis
Chapter	5: Intro To Variables
Chapter	6: Debugging Intro To Variables
Chapter	7: Hacking Intro To Variables
Chapter	8: Uninitialized Variables
Chapter	9: Debugging Uninitialized Variables

Chapter 1: hello, world

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

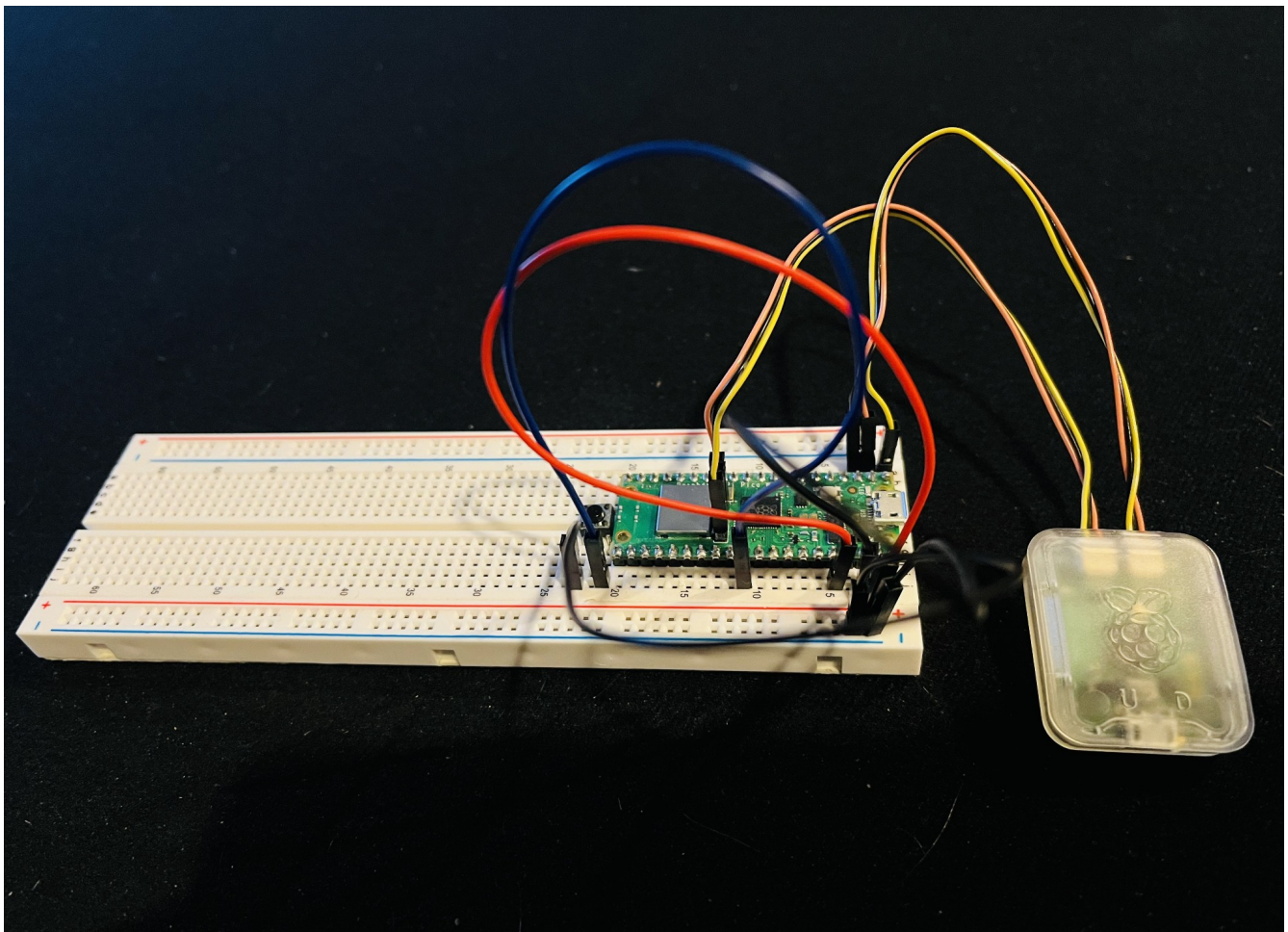
We will then reverse engineer each binary in GDB.

To setup our environment we will follow the details in the link below which covers all operating systems.

<https://www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html#technical-specification>

The next thing we will setup is 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#about-the-debug-probe>



connect a red wire to the 3V3 pin on the Pico W then to the red power rail
connect a black wire to the GND pin on the Pico W then to the black ground rail
connect a blue wire to the RUN pin on the Pico W then to the right button pin
connect a black wire to the black ground rail then to the left button pin
connect the female yellow wire from the Pico Debug Probe to male bottom debug pin
connect the female black wire from the Pico Debug Probe to male center debug pin
connect the female orange wire from the Pico Debug Probe to the male top debug pin
connect the male yellow wire from the Pico Debug Probe to the Pico W GP0 pin
connect the male orange wire from the Pico Debug Probe to the Pico W GP1 pin
connect the male black wire from the Pico Debug Probe to the Pico W GND pin

A **PicoW-A4-Pinout.pdf** file exists in the GitHub repo as well to help find the respective pins.

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>

Clone the repo to whatever folder you prefer.

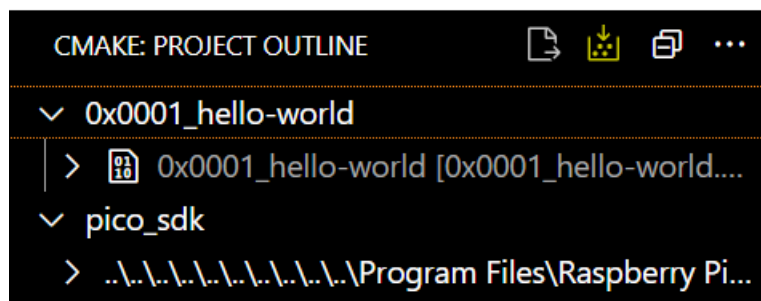
```
git clone https://github.com/mytechnotalent/Embedded-Hacking.git
```

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

Give it a few minutes to initialize.

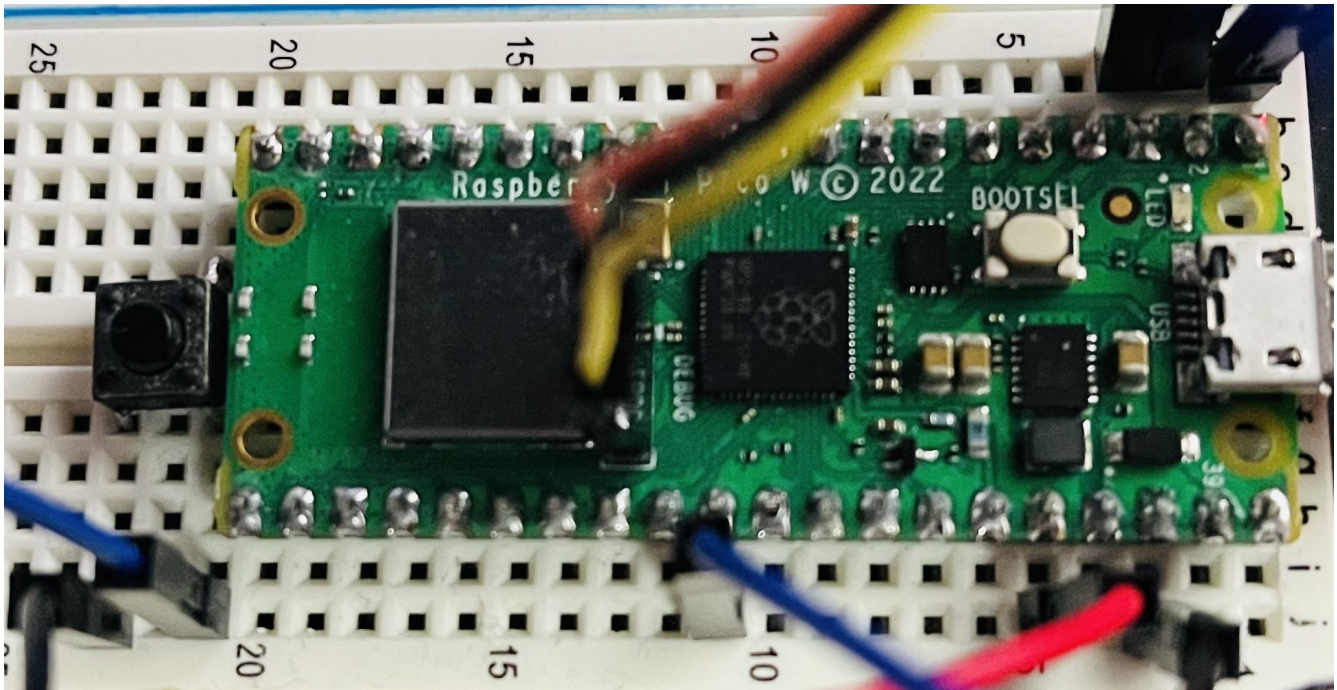
It may ask you for an active kit of which you will choose Pico ARM GCC.

On the left-hand side of the VS code window is a Cmake button. It looks like a triangle with a wrench through it. Click on the center icon to build all projects (highlighted in yellow below).



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

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



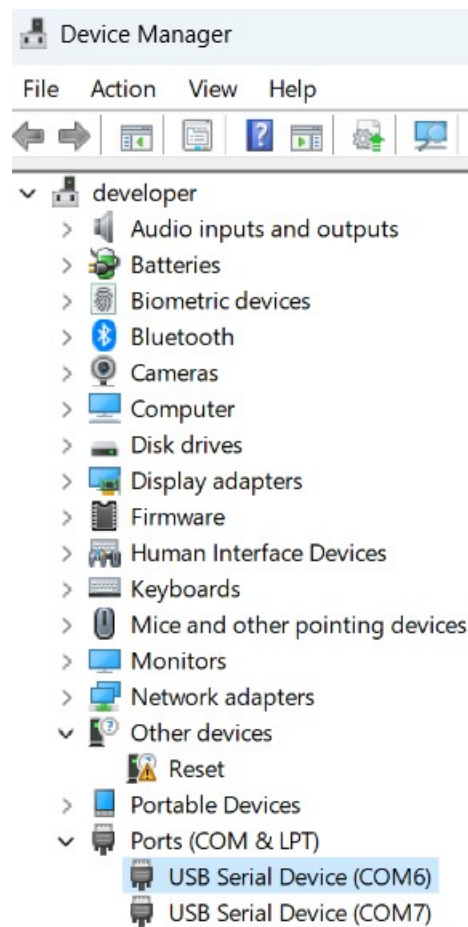
This will open up a file explorer window to copy our **0x0001_hello-world.uf2** firmware into the **RPI-RP2** drive.

drag and drop 0x0001_hello-world.uf2 file from the build directory to the RPI-RP2

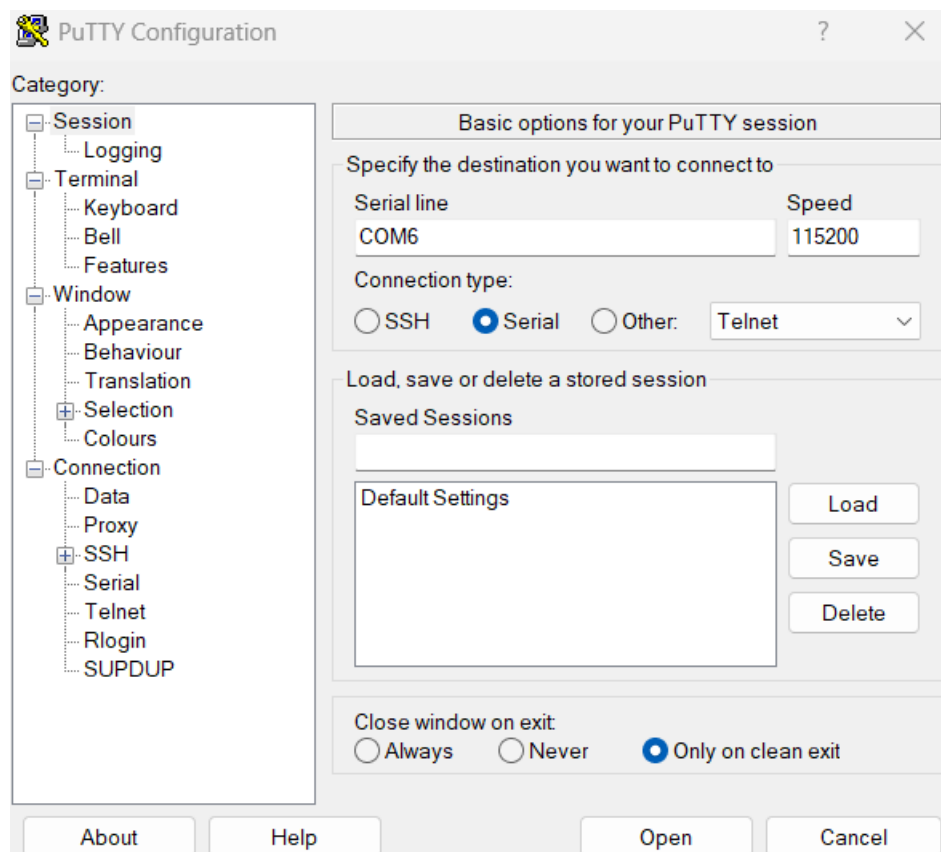
We need to download a serial monitor to interact with our Pico W. If you are on Windows download PuTTY as the link is below.

<https://www.putty.org>

If you are on Windows you can open up 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 W UART and the other for the Pico Debug Probe. Try both and one of them will be UART that we are looking for.



Next step is to run PuTTY.



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

If you are on MAC or Linux you can follow the instructions in the below link to use minicom.

<https://www.raspberrypi.com/documentation/microcontrollers/debug-probe.html#serial-connections>

Now let's review our **main.c** file as this is located in the main folder.

```
#include <stdio.h>
#include "pico/stdlib.h"
#include "pico/cyw43_arch.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.

```
#include "pico/cyw43_arch.h"
```

This line includes the `pico/cyw43_arch.h` header file, which contains declarations for Raspberry Pi Pico W-specific functions, such as those related to Wi-Fi. NOTE: We will not setup or use Wi-Fi until much later in this book.

```
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.

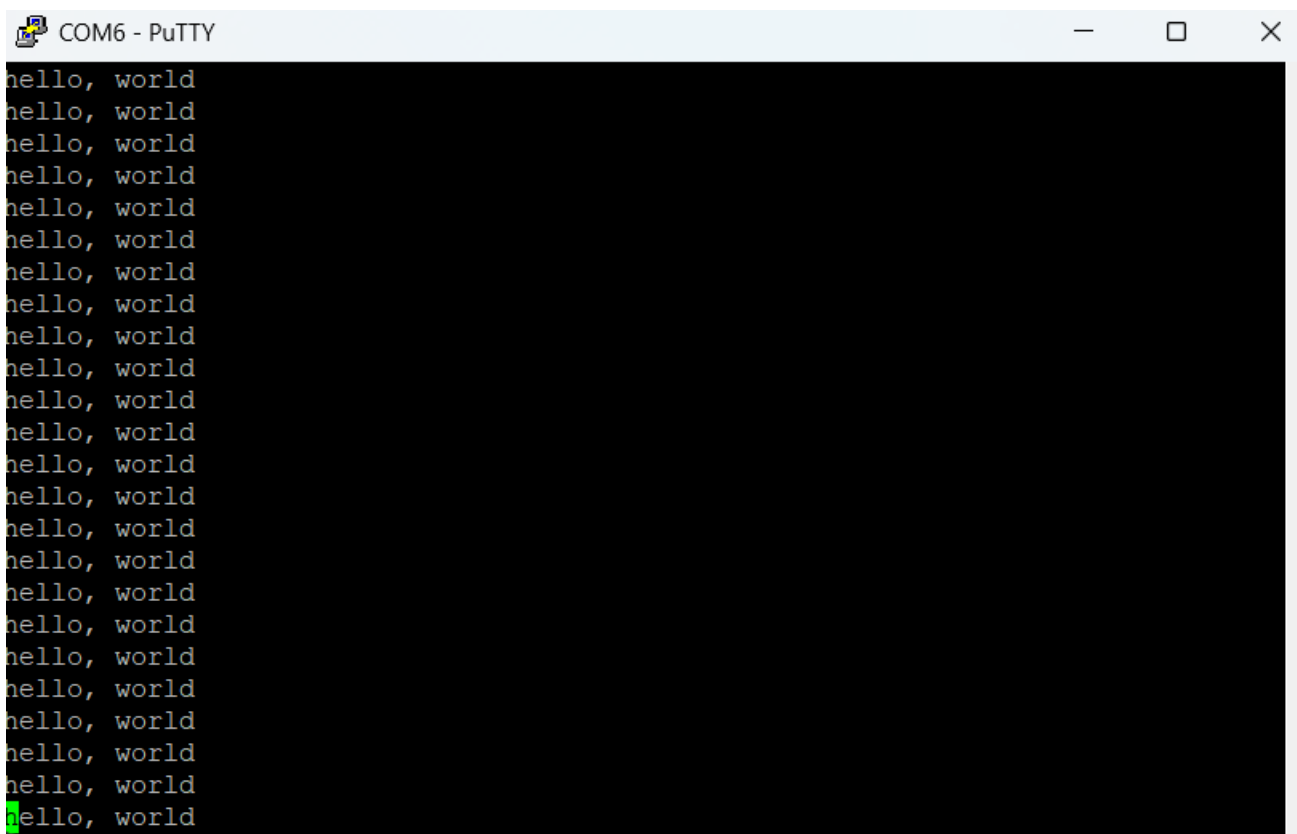
```
while (true)
```

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.

When we open up our terminal we will see hello, world as expected being printed over and over in the terminal.

A screenshot of a PuTTY terminal window. The title bar at the top reads "COM6 - PuTTY" and includes standard window controls (minimize, maximize, close). The terminal area has a black background with white text. The text "hello, world" is printed on approximately 25 lines, with each line starting with a green cursor character. The text is repeated continuously, demonstrating the output of the program.

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

Chapter 2: Debugging hello, world

Today we debug!

Before we get started, we are going DEEP and I mean DEEP! Please do not get discouraged as I will take you through literally every single step of the binary but we need to start small.

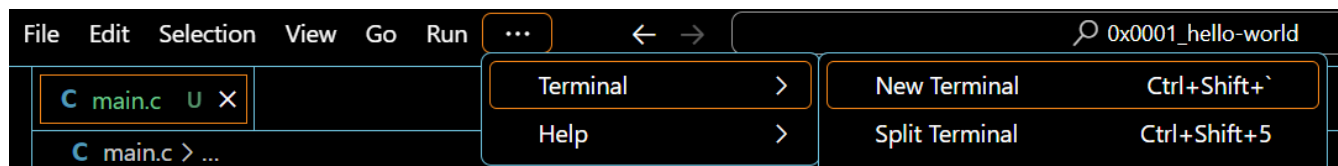
Assembler is not natural to everyone and I have another FREE book and primer on Embedded Assembler below if you feel you need a good primer. PLEASE take the time to read this book if you are new to this so that you can get the full benefit of this book.

<https://github.com/mytechnotalent/Embedded-Assembler>

I am going to work within Windows as the majority of people I have polled for the book operate within Windows.

Lets run OpenOCD to get our remote debug server going.

Open up a terminal within VSCode.



```
cd ..  
.\openocd.ps1
```

```
PS C:\Users\mytec\Documents\Embedded-Hacking\0x0001_hello-world> cd ..  
PS C:\Users\mytec\Documents\Embedded-Hacking> .\openocd.ps1  
Open On-Chip Debugger 0.12.0-g4257276 (2023-01-27-10:19)  
Licensed under GNU GPL v2  
For bug reports, read  
    http://openocd.org/doc/doxygen/bugs.html  
Info : Hardware thread awareness created  
Info : Hardware thread awareness created  
adapter speed: 5000 kHz  
  
Info : Listening on port 6666 for tcl connections  
Info : Listening on port 4444 for telnet connections
```

If you are on MAC or Linux, simply run the below command.

```
openocd -f interface/cmsis-dap.cfg -f target/rp2040.cfg -c "adapter speed 5000"
```

Open up a new terminal and cd `.\build\ dir` and then run the following.

```
arm-none-eabi-gdb .\0x0001_hello-world.bin
```

```
PS C:\Users\mytec\Documents\Embedded-Hacking\0x0001_hello-world> cd .\build\  
PS C:\Users\mytec\Documents\Embedded-Hacking\0x0001_hello-world\build> arm-none-eabi-gdb .\0x0001_hello-world.bin  
GNU gdb (GNU Arm Embedded Toolchain 10.3-2021.10) 10.2.90.20210621-git  
Copyright (C) 2021 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=i686-w64-mingw32 --target=arm-none-eabi".  
Type "show configuration" for configuration details.  
For bug reporting instructions, please see:  
<https://www.gnu.org/software/gdb/bugs/>.  
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"...  
"016ffbd4s": not in executable format: file format not recognized  
(gdb)
```

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

```
target remote :3333
```

We need to halt the currently running binary.

```
monitor reset halt
```

```
(gdb) target remote :3333  
Remote debugging using :3333  
warning: No executable has been specified and target does not support  
determining executable automatically. Try using the "file" command.  
warning: multi-threaded target stopped without sending a thread-id, using first non-exited thread  
0x000000ea in ?? ()  
(gdb) monitor reset halt  
[rp2040.core0] halted due to debug-request, current mode: Thread  
xPSR: 0xf1000000 pc: 0x000000ea msp: 0x20041f00  
[rp2040.core1] halted due to debug-request, current mode: Thread  
xPSR: 0xf1000000 pc: 0x000000ea msp: 0x20041f00  
(gdb)
```

We need to touch base on what XIP is within the RP2040 MCU or microcontroller. This is the actual chip that powers the Pico W.

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.

We see our pc or program counter is currently at 0x000000ea which is very low in memory.

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 lets examine 1000 instructions and look for a *push {r4, lr}* instruction which would indicate our main stack frame being called.

NOTE: ADDRESSES WILL VARY FROM MACHINE TO MACHINE

```
(gdb) x/1000i 0x10000000
...
0x10000304:  push    {r4, lr}
```

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

We first need to have an understanding of how memory is layed out within the Pico W and specifically the RP2040 chip. The RP2040 uses a dual-core ARM Cortex-M0+ processor. We begin with the concepts of the stack and heap as they are fundamental to understanding memory management in embedded systems.

The stack is a region of memory used for managing function calls and local variables. It grows and shrinks automatically as functions like main are called and return. Each time a function is called, a stack frame is created to store local variables and return addresses. The stack pointer (SP) register keeps track of the current position in the stack.

The RP2040 has a dedicated stack for each core, as it is a dual-core processor. The stack size is typically defined in the linker script or project configuration and is limited by the available RAM.

Push: Adding data to the stack (e.g., pushing function parameters).
Pop: Removing data from the stack (e.g., popping values after a function call).

The stack pointer is a register that keeps track of the current position in the stack. It is automatically adjusted during function calls and returns.

If the stack grows beyond its allocated size, it can lead to a stack overflow, causing unpredictable behavior or crashes. In contrast, the heap is a region of memory used for dynamic memory allocation. It is managed by the programmer, and memory must be explicitly allocated and deallocated.

Dynamically allocated memory using functions like `malloc()` or `new` in C or C++. It is useful for managing variable-sized data structures. The heap on the RP2040 is typically part of the RAM region. The size of the heap is not fixed and can be adjusted based on application requirements.

We can allocate and deallocate memory on the heap. Allocation: Obtaining a block of memory from the heap. Deallocation: Returning a block of memory to the heap.

Over time, as memory is allocated and deallocated, the heap may become fragmented, making it challenging to find contiguous blocks of memory.

The RP2040 uses the C standard library's memory allocation functions (`malloc()`, `free()`) to manage the heap. The size of the heap is often defined in the linker script or project configuration. Both the stack and heap are typically located in RAM.

The RP2040 has a limited amount of RAM, so careful management is crucial. Code is stored in Flash memory, and it's executed directly from there.

There are also general-purpose registers that are essential for program execution, data manipulation, and control flow. Below is an overview of these registers:

ARM Cortex-M0+ General-Purpose Registers:

1. r0 - r12 (Register 0 - Register 12):

These are general-purpose registers used for temporary storage of data during program execution.

r0 is often used as a scratch register or for holding function return values.

r1 to r3 are also commonly used for passing function arguments.

r4 to r11 are generally used for holding variables and intermediate values.

2. r13 - Stack Pointer (SP):

r13 is the Stack Pointer (SP), which points to the current top of the stack.

The stack, as previously discussed, is used for storing local variables and managing function calls.

3. r14 - Link Register (LR):

r14 is the Link Register (LR), used to store the return address when a function is called.

Upon a function call, the address of the next instruction to be executed is stored in LR.

4. r15 - Program Counter (PC):

r15 is the Program Counter (PC), which holds the memory address of the next instruction to be fetched and executed.

When a branch or jump instruction is encountered, the new address is loaded into PC.

5. Application Program Status Register (APSR):

Contains status flags such as zero flag (Z), carry flag (C), negative flag (N), etc.

The APSR reflects the status of the ALU (Arithmetic Logic Unit) after arithmetic operations.

6. Program Status Register (PSR):

Combines the APSR with other status information.

Contains information about the current operating mode and interrupt status.

7. Control Register (CONTROL):

Contains the exception number of the current Interrupt Service Routine (ISR).

Bit 0 is the privilege level bit, determining whether the processor is in privileged or unprivileged mode.

Let's discuss usage and considerations below.

Function Calls:

r0 to r3 are used to pass arguments to functions.
LR is used to store the return address.
The stack (SP) is used to store local variables.

Branch and Jump:

PC is updated to the new address during branch or jump instructions.
Status Registers:

APSR flags are used for conditional branching and checking the outcome of arithmetic/logic operations.

Stack Usage:

The stack (SP) is used for managing function calls and local variables.

Now let's examine our main function.

```
(gdb) x/5i 0x10000304
0x10000304: push    {r4, lr}
0x10000306: bl      0x1000406c
0x1000030a: ldr     r0, [pc, #8]    ; (0x10000314)
0x1000030c: bl      0x10003ff4
0x10000310: b.n     0x1000030a
```

let's set a breakpoint to our main function.

```
(gdb) b *0x10000304
Breakpoint 1 at 0x10000304
Note: automatically using hardware breakpoints for read-only addresses.
(gdb) c
Continuing.
```

Thread 1 "rp2040.core0" hit Breakpoint 1, 0x10000304 in ?? ()

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 0x10000304
=> 0x10000304: push    {r4, lr}
0x10000306: bl      0x1000406c
0x1000030a: ldr     r0, [pc, #8]    ; (0x10000314)
0x1000030c: bl      0x10003ff4
0x10000310: b.n     0x1000030a
```

We are about to start off pushing the r4 register and the lr register to the stack.

Keep in mind, the base pointer (BP) is not a register in the RP2040 ARM Cortex-M0+ architecture, and therefore, it is not present as a dedicated register like in some other architectures such as x86. Instead, the ARM Cortex-M0+ architecture relies on the use of the stack pointer (SP) and the link register (LR) for managing the stack during function calls.

In ARM Cortex-M0+, the stack pointer (SP or r13) is typically used to point to the top of the stack, and it is adjusted dynamically as functions are called and return. The link register (LR or r14) is used to store the return address when a function is called. The base pointer, as seen in some other architectures like x86 (EBP), is not explicitly used or available in the same way.

In the context of the RP2040 and ARM Cortex-M0+, you would primarily rely on the stack pointer (SP) and link register (LR) for managing the stack and tracking return addresses during function calls. The base pointer concept is not part of the standard conventions for this architecture.

As stated, the lr register contains the return address to return to after main finishes. Keep in mind, we are using a micro-controller so main will be in an infinite loop and will never return.

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

```
(gdb) x/10x $sp
0x20042000: 0x00000000 0x00000000 0x00000000 0x00000000
0x20042010: 0x00000000 0x00000000 0x00000000 0x00000000
0x20042020: 0x00000000 0x00000000
```

Now lets step-into which means take a single step in Assembler.

```
(gdb) si
0x10000306 in ?? ()
(gdb) x/5i 0x10000304
0x10000304: push    {r4, lr}
=> 0x10000306: bl      0x1000406c
0x1000030a: ldr     r0, [pc, #8] ; (0x10000314)
0x1000030c: bl      0x10003ff4
0x10000310: b.n     0x1000030a
```

Now let's review our stack.

```
(gdb) x/10x $sp
0x20041ff8: 0x10000264 0x10000223 0x00000000 0x00000000
0x20042008: 0x00000000 0x00000000 0x00000000 0x00000000
0x20042018: 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 r4 and lr.

```
(gdb) x/x $r4
0x10000264:      0x00004700
(gdb) x/x $lr
0x10000223:      0x00478849
```

Keep in mind, the stack grows downward so we first see the lr pushed to the top of the stack. Our stack pointer is currently at 0x20041ff8.

```
(gdb) x/x $sp+4
0x20041ffc:      0x10000223
```

We see the stack pointer was first at 0x20041ffc and then it was pushed DOWN to 0x20041ff8.

```
(gdb) x/x $sp
0x20041ff8:      0x10000264
```

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

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.

```
stdio_init_all();
```

Because we are working with a binary without any symbol info we need to do two steps which are si and then ret to return out of the function call.

```
(gdb) x/5i 0x10000304
0x10000304: push    {r4, lr}
=> 0x10000306: bl      0x1000406c
0x1000030a: ldr      r0, [pc, #8]    ; (0x10000314)
0x1000030c: bl      0x10003ff4
0x10000310: b.n     0x1000030a
(gdb) si
0x1000406c in ?? ()
(gdb) ret
Make selected stack frame return now? (y or n) y
#0 0x1000030a in ?? ()
(gdb) x/5i 0x10000304
0x10000304: push    {r4, lr}
0x10000306: bl      0x1000406c
=> 0x1000030a: ldr      r0, [pc, #8]    ; (0x10000314)
```

```
0x1000030c:  bl      0x10003ff4
0x10000310:  b.n     0x1000030a
```

Now we are about to load the value INSIDE of a memory address at 0x10000314 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) x/x $r0
0x10006918:      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
0x10006918:      "hello, world\r"
```

This has been quite a bit of information but please take the time and work through this several times and I again encourage you to read the FREE Embedded Assembler if you want a deeper dive into this.

<https://github.com/mytechnotalent/Embedded-Assembler>

In our next lesson we will hack this hello, world program!

Chapter 3: Hacking hello, world

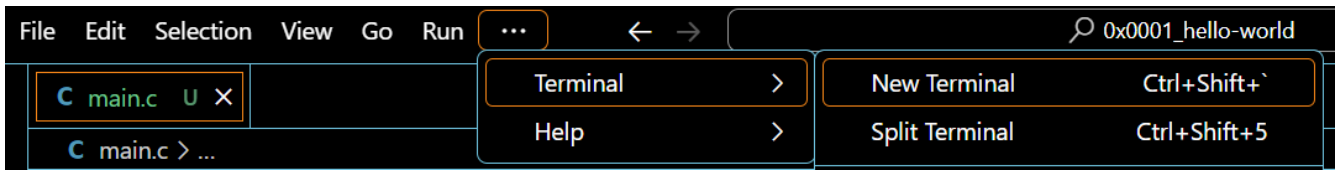
Today we hack!

Lets run OpenOCD to get our remote debug server going.

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

[illegible]

Open up a terminal within VSCode.



```
cd ..  
.\openocd.ps1
```

```
● PS C:\Users\mytec\Documents\Embedded-Hacking\0x0001_hello-world> cd ..  
○ PS C:\Users\mytec\Documents\Embedded-Hacking> .\openocd.ps1  
Open On-Chip Debugger 0.12.0-g4257276 (2023-01-27-10:19)  
Licensed under GNU GPL v2  
For bug reports, read  
    http://openocd.org/doc/doxygen/bugs.html  
Info : Hardware thread awareness created  
Info : Hardware thread awareness created  
adapter speed: 5000 kHz  
  
Info : Listening on port 6666 for tcl connections  
Info : Listening on port 4444 for telnet connections
```

If you are on MAC or Linux, simply run the below command.

```
openocd -f interface/cmsis-dap.cfg -f target/rp2040.cfg -c "adapter speed 5000"
```

Open up a new terminal and cd `.\build\ dir` and then run the following.

```
arm-none-eabi-gdb .\0x0001_hello-world.bin
```

```

PS C:\Users\mytec\Documents\Embedded-Hacking\0x0001_hello-world> cd .\build\
PS C:\Users\mytec\Documents\Embedded-Hacking\0x0001_hello-world\build> arm-none-eabi-gdb .\0x0001_hello-world.bin
GNU gdb (GNU Arm Embedded Toolchain 10.3-2021.10) 10.2.90.20210621-git
Copyright (C) 2021 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=i686-w64-mingw32 --target=arm-none-eabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
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"...
"016ffbd4s": not in executable format: file format not recognized
(gdb)

```

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

```
target remote :3333
```

We need to halt the currently running binary.

```
monitor reset halt
```

```

(gdb) target remote :3333
Remote debugging using :3333
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
warning: multi-threaded target stopped without sending a thread-id, using first non-exited thread
0x000000ea in ?? ()
(gdb) monitor reset halt
[rp2040.core0] halted due to debug-request, current mode: Thread
xPSR: 0xf1000000 pc: 0x000000ea msp: 0x20041f00
[rp2040.core1] halted due to debug-request, current mode: Thread
xPSR: 0xf1000000 pc: 0x000000ea msp: 0x20041f00
(gdb)

```

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

Let's re-examine main.

NOTE: ADDRESSES WILL VARY FROM MACHINE TO MACHINE

```
(gdb) x/5i 0x10000304
0x10000304: push    {r4, lr}
0x10000306: bl      0x1000406c
0x1000030a: ldr     r0, [pc, #8]    ; (0x10000314)
0x1000030c: bl      0x10003ff4
0x10000310: b.n     0x1000030a
```

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

```
(gdb) set $pc = 0x1000030c
(gdb) b *0x1000030c
Breakpoint 1 at 0x1000030c
Note: automatically using hardware breakpoints for read-only addresses.
(gdb) c
Continuing.
```

```
Thread 1 "rp2040.core0" hit Breakpoint 1, 0x1000030c in ?? ()
```

The next thing we need to do is hijack the value of hello, world from 0x10000314 and create our own data within SRAM and fill it with a hacked malicious string ;)

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

Now to need to hijack the address inside r0 which is 0x10000314 and change it to our hacked address in SRAM and verify our hack.

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

Now let's continue and execute our hack!

```
(gdb) c
Continuing.
```

```
Thread 1 "rp2040.core0" hit Breakpoint 1, 0x1000030c in ?? ()
```

Let's verify our hack!

[illegible]

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.

Let's first review the RP2040 datasheet and examine a few areas of interest before we start to examine our hello world binary.

On page 25 of the RP2040 datasheet, we see area 2.2. Address Map.

We see that ROM begins at 0x00000000 and something called XIP begins at 0x10000000. We also see SRAM starting at 0x20000000 and our microcontroller peripherals start at 0x40000000.

Let's break these down.

Base Address: 0x00000000

ROM is where the initial firmware, often referred to as "boot ROM" or "mask ROM," resides. It contains the code executed at startup. This code typically initializes essential system components and sets up the system for further program execution. Since it is read-only, this memory is used for storing permanent and unchangeable instructions.

Base Address: 0x10000000

XIP is a mechanism that allows the microcontroller to execute code directly from external Flash memory. In systems with XIP capability, the microcontroller fetches and executes instructions directly from the external Flash, eliminating the need to load the entire program into RAM first. This can be advantageous in embedded systems where RAM is limited.

Base Address: 0x20000000

SRAM is volatile memory used for storing data that needs to be accessed quickly during program execution. Unlike Flash memory (used for ROM and XIP), SRAM loses its contents when power is turned off. It is used for variables, stack, and other dynamic data during program execution.

Base Address: 0x40000000

Microcontroller peripherals include various hardware components like GPIO (General Purpose Input/Output), UART (Universal Asynchronous Receiver-Transmitter), SPI (Serial Peripheral Interface), I2C (Inter-Integrated Circuit), timers, and other controllers. These peripherals are memory-mapped, meaning they have specific addresses in the microcontroller's address space. By reading from and writing to these

addresses, you can configure and interact with the various hardware features of the microcontroller.

Understanding these memory regions is crucial for bare-metal programming because you'll be directly manipulating these memory addresses to control the behavior of the microcontroller. For example, configuring GPIO pins, setting up communication peripherals, or managing interrupts involves writing to specific addresses within the peripheral memory space.

When writing a "hello world" program in bare-metal, you typically need to set up the stack, initialize necessary peripherals, and write your program code. Since there's no operating system, you have full control over the hardware and must handle everything from initialization to execution.

Another important area of the datasheet is 2.8.3. Bootrom Contents.

Address	Contents	Description
0x00000000	32-bit pointer	Initial boot stack pointer
0x00000004	32-bit pointer	Pointer to boot reset handler function
0x00000008	32-bit pointer	Pointer to boot NMI handler function
0x0000000c	32-bit pointer	Pointer to boot Hard fault handler function
0x00000010	'M', 'u', 0x01	Magic
0x00000013	byte	Bootrom version
0x00000014	16-bit pointer	Pointer to a public function lookup table
0x00000016	16-bit pointer	Pointer to a public data lookup table
0x00000018	16-bit pointer	Pointer to a helper function

Let's break down these items in more detail and the bootrom contents exist between 0x00000000 and 0x0000001F.

Initial Boot Stack Pointer (0x00000000):

This 32-bit pointer indicates the initial stack pointer value when the device boots. The stack pointer is a register that points to the top of the stack, and it's crucial for managing function calls and storing local variables.

Pointer to Boot Reset Handler Function (0x00000004):

This 32-bit pointer indicates the address of the function that will be executed when the microcontroller is reset. The reset handler typically initializes essential system components and sets up the environment for the main program.

Pointer to Boot NMI Handler Function (0x00000008):

NMI stands for Non-Maskable Interrupt. This pointer points to the function that will handle non-maskable interrupts, which are interrupts that cannot be disabled or ignored.

Pointer to Boot Hard Fault Handler Function (0x0000000C):

This pointer indicates the address of the function that will handle hard faults. Hard faults occur when the microcontroller encounters an error that cannot be handled by the normal program flow.

Magic and Bootrom Version (0x00000010 to 0x00000013):

The values 'M', 'u', 0x01 represent a magic number that helps verify the integrity of the bootrom. This is a common technique to ensure that the bootloader or bootrom code is valid.

Bootrom version information is also provided.

Pointer to Public Function and Data Lookup Tables (0x00000014 to 0x00000017):

These pointers lead to lookup tables that likely contain information about public functions and data. Public functions could include services provided by the bootloader that are accessible to user code.

Pointer to Helper Function (0x00000018):

This pointer points to a helper function that may provide essential services during the boot process.

The Vector Table is a critical part of the microcontroller's startup process. It contains pointers to various exception and interrupt handlers. In ARM Cortex-M microcontrollers, the Vector Table is located at the beginning of the program memory.

The first entry in the Vector Table is the initial stack pointer. The second entry is the address of the reset handler function. Subsequent entries are usually addresses of exception and interrupt handlers.

These handlers include the NMI handler, Hard Fault handler, and others. When an exception or interrupt occurs, the microcontroller looks up the corresponding address in the Vector Table and jumps to that location to execute the associated handler code. Understanding and, if necessary, customizing the Vector Table is crucial for bare-metal programming, as it allows you to define how the microcontroller responds to various events during its operation.

There are a number of tools to examine our firmware.

"Firmware" and a "regular application" are terms that refer to different types of software, and their distinction lies in their intended use and functionality.

Firmware is a specialized type of software that is embedded in hardware devices to control their specific functionalities. It is designed to interact closely with the hardware components of a device, providing low-level control and management. Firmware is typically stored in non-volatile memory (such as Flash memory) and is responsible for initializing the hardware and facilitating the communication between hardware and higher-level software.

In the context of the RP2040 microcontroller, firmware refers to the low-level software that is executed on the microcontroller itself. This includes the initial boot code, device initialization, and drivers for various peripherals. The firmware for RP2040 is often responsible for setting up the system, configuring peripherals, and providing a foundation for higher-level software to run on the microcontroller.

A regular application, on the other hand, is a software program designed to perform specific tasks or functions on a computing device. Unlike firmware, applications are generally written to run on top of an operating system and are more abstracted from the hardware. They leverage the services provided by the operating system and interact with the hardware through well-defined APIs (Application Programming Interfaces).

For RP2040, a regular application would be a program written to run on the microcontroller but not involved in the low-level control of hardware. Instead, it interacts with the hardware through APIs provided by the firmware. This application might perform specific tasks, such as sensor data processing, communication, or control functions, utilizing the capabilities of the microcontroller.

The Executable and Linkable Format (ELF) is a common file format for executables, object code, shared libraries, and even core dumps. It is a standard format used for binaries in many software development environments.

In the context of RP2040 development, the ELF format is often used to represent the compiled binary of both firmware and regular applications. The ELF file contains sections like text (executable code), data (initialized and uninitialized data), and various other

sections that define the structure and layout of the binary. Tools in the development process, such as compilers and linkers, generate and manipulate ELF files.

In summary, firmware on RP2040 refers to the low-level software responsible for managing and controlling hardware, while a regular application is a higher-level program designed to perform specific tasks. Both firmware and regular applications for RP2040 can be represented in the ELF format, providing a standardized way to organize and distribute executable binaries.

We will revisit our hello world binary and take a deeper look into how this all works. Keep in mind we debugged with the .bin file which had no symbols or helpful locators of where things are as in the real-world of reverse engineering you will rarely get symbols.

I bring this up as to not confuse you as we are going to examine the elf binary which is a .bin file with symbols in it's simplest explanation.

Let's do some firmware analysis!

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

Give it a few minutes to initialize.

It may ask you for an active kit of which you will choose Pico ARM GCC.

Let's open up a terminal and take a look at our first tool.

The `arm-none-eabi-objdump` command is a tool used in the ARM development environment to display information about object files. The `-h` option specifies that you want to display the section headers of the ELF (Executable and Linkable Format) file.

When you run the `arm-none-eabi-objdump -h` command on a specific ELF binary file (in this case, `.build\0x0001_hello-world.elf`), it provides detailed information about the various sections in that ELF file.

Let's break down what this command will show and how it relates to the RP2040 ELF binary:

Section Headers:

Sections are parts of the ELF file that organize and hold specific types of data. Section headers contain information about each section, such as its name, type, virtual address, size, and other attributes.

Output Format:

The output of the **arm-none-eabi-objdump -h** command typically includes a table with columns representing different attributes of each section.

The columns may include:

Name: The name of the section.

Size: The size of the section in bytes.

VMA (Virtual Memory Address): The virtual memory address at which the section is loaded into memory.

LMA (Load Memory Address): The load memory address, which is the same as VMA for most sections.

Offset: The offset of the section in the file.

Align: The alignment requirement for the section.

For the RP2040 microcontroller, the ELF binary would contain sections that represent different parts of the program, including code sections, data sections, and potentially sections related to microcontroller-specific features.

The ELF file for RP2040 would likely include sections for the bootloader, firmware, and potentially user applications. The bootloader, for example, might reside at a specific address in the memory space, and the ELF file's section headers would provide details about the location and size of the bootloader code and data.

The information provided by the section headers is crucial for understanding how the ELF file is mapped into the memory space of the RP2040. This includes the starting addresses of various sections and their sizes, which are essential for programming and debugging embedded systems.

In summary, running **arm-none-eabi-objdump -h** on the RP2040 ELF binary will show detailed information about the sections within the binary, helping developers understand how different parts of the program are organized in memory and how they relate to the RP2040 microcontroller's architecture.

Let's run the following.

NOTE: ADDRESSES WILL VARY FROM MACHINE TO MACHINE

arm-none-eabi-objdump -h .\build\0x0001_hello-world.elf

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

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.boot2	00000100	10000000	10000000	00001000	2**0
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
1	.text	00006818	10000100	10000100	00001100	2**3
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
2	.rodata	00001758	10006918	10006918	00007918	2**3
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
3	.binary_info	00000028	10008070	10008070	00009070	2**2
	CONTENTS, ALLOC, LOAD, DATA					
4	.ram_vector_table	000000c0	20000000	20000000	00009cb8	2**2
	CONTENTS					
5	.data	00000bf8	200000c0	10008098	000090c0	2**4
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
6	.uninitialized_data	00000000	20000cb8	20000cb8	00009d78	2**0
	CONTENTS					
7	.scratch_x	00000000	20040000	20040000	00009d78	2**0
	CONTENTS					
8	.scratch_y	00000000	20041000	20041000	00009d78	2**0
	CONTENTS					
9	.bss	00000d94	20000cb8	20000cb8	00009cb8	2**3
	ALLOC					
10	.heap	00000800	20001a4c	20001a4c	00009d78	2**2
	CONTENTS, READONLY					
11	.stack_dummy	00000800	20041000	20041000	0000a580	2**5
	CONTENTS, READONLY					
12	.ARM.attributes	00000028	00000000	00000000	0000ad80	2**0
	CONTENTS, READONLY					
13	.comment	00000049	00000000	00000000	0000ada8	2**0
	CONTENTS, READONLY					
14	.debug_info	00025b3b	00000000	00000000	0000adf1	2**0
	CONTENTS, READONLY, DEBUGGING, OCTETS					
15	.debug_abbrev	00006d3c	00000000	00000000	0003092c	2**0
	CONTENTS, READONLY, DEBUGGING, OCTETS					
16	.debug_aranges	00001400	00000000	00000000	00037668	2**3
	CONTENTS, READONLY, DEBUGGING, OCTETS					
17	.debug_ranges	00004bd0	00000000	00000000	00038a68	2**3
	CONTENTS, READONLY, DEBUGGING, OCTETS					
18	.debug_line	0001804f	00000000	00000000	0003d638	2**0
	CONTENTS, READONLY, DEBUGGING, OCTETS					
19	.debug_str	00007638	00000000	00000000	00055687	2**0
	CONTENTS, READONLY, DEBUGGING, OCTETS					
20	.debug_frame	00002ac0	00000000	00000000	0005ccc0	2**2
	CONTENTS, READONLY, DEBUGGING, OCTETS					
21	.debug_loc	00018ed3	00000000	00000000	0005f780	2**0
	CONTENTS, READONLY, DEBUGGING, OCTETS					

1. .boot2 Section:

Size: 256 bytes (0x100)

VMA (Virtual Memory Address): 0x10000000

LMA (Load Memory Address): 0x10000000

File Offset: 0x1000

Attributes: CONTENTS, ALLOC, LOAD, READONLY, CODE

This section likely contains the bootloader code. It is read-only, loaded into memory at the specified address (0x10000000), and contains executable code.

2. .text Section:

Size: 26680 bytes (0x6818)

VMA: 0x10000100

LMA: 0x10000100

File Offset: 0x1100

Attributes: CONTENTS, ALLOC, LOAD, READONLY, CODE

This is the main code section (.text) of your program. It is read-only, loaded into memory at 0x10000100, and contains the executable instructions of your program.

3. .rodata Section:

Size: 5976 bytes (0x1758)

VMA: 0x10006918

LMA: 0x10006918

File Offset: 0x7918

Attributes: CONTENTS, ALLOC, LOAD, READONLY, DATA

This section likely contains read-only data (constants, strings, etc.) used by your program. It is loaded into memory at 0x10006918.

4. .binary_info Section:

Size: 40 bytes (0x28)

VMA: 0x10008070

LMA: 0x10008070

File Offset: 0x9070

Attributes: CONTENTS, ALLOC, LOAD, DATA

This section contains binary information. It is loaded into memory at 0x10008070.

5. .ram_vector_table Section:

Size: 192 bytes (0xC0)

VMA: 0x20000000

LMA: 0x20000000

File Offset: 0x9CB8

Attributes: CONTENTS

This section likely contains the vector table for interrupts. It's loaded into RAM at 0x20000000.

6. .data Section:

Size: 3064 bytes (0xBF8)

VMA: 0x200000C0

LMA: 0x10008098

File Offset: 0x90C0

Attributes: CONTENTS, ALLOC, LOAD, READONLY, CODE

This is the initialized data section. It includes initialized global and static variables. It is loaded into RAM at 0x200000C0.

7. .uninitialized_data Section:

Size: 0 bytes

VMA: 0x20000CB8

LMA: 0x20000CB8

File Offset: 0x9D78

Attributes: CONTENTS

This section is for uninitialized data. It's likely that the .bss section will fulfill the same purpose.

8-11. .scratch_x, .scratch_y, .bss, .heap Sections:

These sections represent scratch memory, uninitialized data, heap, and stack.

12-20. Debug Sections:

These sections contain debugging information

(.ARM.attributes, .comment, .debug_info, .debug_abbrev, .debug_abbrevs, .debug_ranges, .debug_line, .debug_str, .debug_frame, .debug_loc).

Understanding these sections is crucial for debugging and memory management. The .text section contains the main code, .data contains initialized data, and .bss is for uninitialized data.

The .ram_vector_table is essential for interrupt handling, and other sections provide additional details and support for debugging.

When programming the RP2040, it's important to know where each section is loaded in memory, especially when dealing with limited resources in embedded systems. The memory map and section headers provide insight into how your code and data are organized in the memory space of the microcontroller.

Let's examine our next tool.

The -D option in the arm-none-eabi-objdump command stands for "disassemble." When used, it instructs the objdump tool to display the disassembly of the entire ELF file, including all sections. Let's break down how this option works and why it's relevant.

```
arm-none-eabi-objdump -D .\build\0x0001_hello-world.elf | less
```

`arm-none-eabi-objdump`: The main command for disassembling and inspecting binary files in the ARM embedded toolchain.

`-D`: This option tells `objdump` to disassemble the contents of the ELF file. It means that the tool will convert the machine code instructions in the binary file into human-readable assembly language instructions. The disassembly output will include not only the code sections but also information about other sections like data, symbols, and more.

`.\build\0x0001_hello-world.elf`: This is the path to the ELF file that you want to analyze. Replace this with the actual path to your compiled program.

`| less`: The pipe (`|`) operator is used to send the output of `objdump` to the `less` command. `less` is a pager program that allows you to scroll through large amounts of text one screen at a time.

The `-D` option is crucial for gaining a comprehensive understanding of the program. It disassembles the entire content of the ELF file, providing insights into the machine-level instructions generated by the compiler.

Disassembling all sections means that you get information about not only the code sections (`.text`), but also other sections like data (`.data`, `.bss`), symbols, and potentially debug information. This can be valuable for debugging, analyzing memory usage, and understanding the structure of your program.

The Vector Table, which contains addresses of interrupt and exception handlers, is included in the disassembly. By disassembling all sections, you can locate the Vector Table and understand how the microcontroller responds to different events.

Memory Layout Visualization:

The disassembly output includes information about the memory layout, showing where different sections are loaded in memory. This is particularly important in embedded systems where memory usage is critical.

Disassembly is a powerful tool for debugging and optimization. It allows you to inspect the generated machine code, identify potential issues, and optimize the code for size or performance. By using the `-D` option in `objdump`, you obtain a detailed disassembly of your program, providing a deep insight into how the high-level code you wrote is translated into low-level machine instructions for the target microcontroller. This is invaluable for embedded systems

development, especially when working with resource-constrained devices like the RP2040.

The `-D` option in `arm-none-eabi-objdump` instructs the tool to disassemble the contents of the ELF file, providing a human-readable representation of the machine code instructions. However, disassembling data as if it were instructions can result in nonsense output because the tool interprets arbitrary data as if it were executable code. Let's explore why this can happen:

One of the issues with this tool is that the data sections are misinterpreted. The `-D` option doesn't discriminate between code sections (text) and data sections. It attempts to disassemble all sections in the ELF file. This means that sections containing non-executable data, such as initialized or uninitialized variables, will be disassembled as if they were instructions.

Data sections might contain values that, when interpreted as instructions, result in nonsensical or invalid opcodes. For example, a sequence of bytes representing ASCII characters or numeric values might not make sense when treated as executable instructions.

When disassembling data, you might see output that appears to be assembly instructions, but these are essentially meaningless in terms of program execution. It can be misleading and confusing, especially if you're expecting only code sections to be disassembled.

Symbol Confusion:

The disassembler might attempt to interpret data values as symbolic instructions, leading to the creation of pseudo-instructions that don't correspond to any valid machine code. This can make it challenging to distinguish between actual instructions and arbitrary data.

Example:

Consider a simple data section that contains an array of 32-bit integers:

```
// Data section
int data_array[] = {0x12345678, 0xAABBCCDD, 0xDEADBEEF};
```

When disassembling this data section with `-D`, the tool might interpret the data values as instructions and attempt to disassemble them. However, these values don't represent valid ARM instructions, leading to nonsense output.

Another important part of this analysis is the Vector Table. The Vector Table is a critical part of ARM Cortex-M microcontrollers. It

contains addresses of interrupt service routines (ISRs) and is usually located at the beginning of the program memory. Let's discuss how you can find and interpret the Vector Table in the disassembly output:

Scroll through the disassembled code using the arrow keys. The Vector Table is typically located at the beginning of the disassembly output.

In the Vector Table, you should see a list of addresses at the beginning of the disassembly, each corresponding to a specific interrupt or exception.

The first entry in the Vector Table is the address of the Reset Handler, which is the starting point of your program. It's the address where the microcontroller jumps to when it is powered on or reset.

Following the Reset Handler, you'll find addresses for other exception handlers, such as NMI, Hard Fault, and various interrupts. Understand the Addresses:

Each address in the Vector Table points to the corresponding handler function or routine that should be executed when the associated interrupt or exception occurs.

For example, if you see an address like 0x10000100 in the disassembly output, it likely corresponds to the Reset Handler. The exact details will depend on your specific program and the configuration of your microcontroller.

By examining the disassembly output with objdump, you can gain insights into how your code is translated into machine instructions, identify key sections such as the Vector Table, and understand the flow of control in your program.

```
arm-none-eabi-objdump -D .\build\0x0001_hello-world.elf | less
```

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

Disassembly of section .boot2:

```
10000000 <__boot2_start__>:
10000000:      4b32b500      blmi    10cad408 <__flash_binary_end+0xca4778>
10000004:      60582021      subsvs  r2, r8, r1, lsr #32
10000008:      21026898      ; <UNDEFINED> instruction:
0x21026898
1000000c:      60984388      addsvs  r4, r8, r8, lsl #7
10000010:      611860d8      ldrsbvs r6, [r8, -r8]
```

```

10000014:      4b2e6158      blmi    10b9857c <__flash_binary_end+0xb8f8ec>
10000018:      60992100      addsvs  r2, r9, r0, lsl #2
1000001c:      61592102      cmpvs   r9, r2, lsl #2
10000020:      22f02101      rscscs  r2, r0, #1073741824      ; 0x40000000
10000024:      492b5099      stmdbmi fp!, {r0, r3, r4, r7, ip, lr}
10000028:      21016019      tstcs   r1, r9, lsl r0
1000002c:      20356099      mlascs  r5, r9, r0, r6
10000030:      f844f000      ; <UNDEFINED> instruction:
0xf844f000
10000034:      42902202      addsmi  r2, r0, #536870912      ; 0x20000000
10000038:      2106d014      tstcs   r6, r4, lsl r0
1000003c:      f0006619      ; <UNDEFINED> instruction:
0xf0006619
10000040:      6e19f834      mrcvs   8, 0, APSR_nzcv, cr9, cr4, {1}
10000044:      66192101      ldrrvs  r2, [r9], -r1, lsl #2
10000048:      66182000      ldrrvs  r2, [r8], -r0
1000004c:      f000661a      ; <UNDEFINED> instruction:
0xf000661a
10000050:      6e19f82c      cdpvs   8, 1, cr15, cr9, cr12, {1}
10000054:      6e196e19      mrcvs   14, 0, r6, cr9, cr9, {0}
10000058:      f0002005      ; <UNDEFINED> instruction:
0xf0002005
1000005c:      2101f82f      tstcs   r1, pc, lsr #16 ; <UNPREDICTABLE>
10000060:      d1f94208      mvnsls  r4, r8, lsl #4
10000064:      60992100      addsvs  r2, r9, r0, lsl #2
10000068:      6019491b      andsvs  r4, r9, fp, lsl r9
1000006c:      60592100      subsvs  r2, r9, r0, lsl #2:
...

```

The contents are quite large but when you run this on your own you will see the extent of the result. The key is this is everything and you need to ignore the data representation of attempted Assembler code as noted above.

Let's examine our next tool.

The `-d` flag in the `arm-none-eabi-objdump` command specifies that you want to disassemble the executable sections of the ELF file. Here's what the `-d` flag does:

arm-none-eabi-objdump -d .\build\0x0001_hello-world.elf | less
 arm-none-eabi-objdump: The main command for disassembling and inspecting binary files in the ARM embedded toolchain.

`-d`: This option stands for "disassemble." It instructs `objdump` to disassemble the contents of the ELF file. Unlike the `-D` option, which disassembles all sections, the `-d` option specifically focuses on the code sections (text sections) of the program.

.\build\0x0001_hello-world.elf: This is the path to the ELF file that you want to analyze. Replace this with the actual path to your compiled program.

| less: The pipe (|) operator is used to send the output of objdump to the less command. less is a pager program that allows you to scroll through large amounts of text one screen at a time.

The -d option is useful when you specifically want to focus on disassembling the code sections of your program. This is typically the section where the machine code instructions of your program's executable code reside (commonly named .text).

The output of the -d option is human-readable assembly code. It provides a translation of the machine code instructions into mnemonic instructions that are easier for a human to understand.

Disassembling the code helps you understand the flow of your program at the assembly level. You can see the instructions that make up each function and analyze the control flow between different parts of your code.

The disassembled code is often used for debugging and analysis. It allows you to inspect the generated machine code, identify potential issues, and understand how the high-level code you wrote is translated into low-level instructions for the target microcontroller.

Example:

If you have a simple C program like:

```
#include <stdio.h>
#include "pico/stdlib.h"
#include "pico/cyw43_arch.h"

int main(void)
{
    stdio_init_all();

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

This command below will output the disassembled code for the .text section, showing the assembly instructions generated by the compiler for your main function and any other code in the program.

The -d option is particularly useful when you are interested in examining the assembly representation of the machine code instructions in the code sections of your ELF file.

```
arm-none-eabi-objdump -d .\build\0x0001_hello-world.elf | less

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

Disassembly of section .boot2:

```
10000000 <__boot2_start__>:
10000000:      4b32b500      .word    0x4b32b500
10000004:      60582021      .word    0x60582021
10000008:      21026898      .word    0x21026898
1000000c:      60984388      .word    0x60984388
10000010:      611860d8      .word    0x611860d8
10000014:      4b2e6158      .word    0x4b2e6158
10000018:      60992100      .word    0x60992100
...
```

Like our previous command, the results are many pages long but will be good to run in your terminal to get an idea of how our binary looks which starts at 0x10000000 XIP as we discussed prior.

Let's examine our next tool.

The arm-none-eabi-objdump -s command is used to display the content of sections in a binary file. The -s option specifies that you want to view the contents of sections, and it is often used for inspecting the raw data within different sections of an ELF (Executable and Linkable Format) file. Let's break down what the command does:

arm-none-eabi-objdump -s .\build\0x0001_hello-world.elf | less

arm-none-eabi-objdump: The main command for disassembling and inspecting binary files in the ARM embedded toolchain.

-s: This option tells objdump to display the contents of sections in a binary file. Unlike the -d option, which disassembles the code sections into human-readable assembly language, the -s option displays the raw data within the specified sections.

.\build\0x0001_hello-world.elf: This is the path to the ELF file that you want to analyze. Replace this with the actual path to your compiled program.

| less: The pipe (|) operator is used to send the output of objdump to the less command, allowing you to scroll through large amounts of text one screen at a time.

Relevance of -s and Displaying Section Contents:

The `-s` option is used to view the raw binary data within specific sections of the ELF file. This can include code sections, data sections, symbol tables, and more.

By inspecting the contents of sections, you gain insights into how different types of data are organized in memory. This is crucial for understanding the memory layout of your program and for debugging purposes.

For non-code sections, such as `.data` or `.rodata`, the `-s` option allows you to inspect the actual data values stored in these sections. This is valuable for understanding the initialized and constant data used by your program.

The `-s` option can also be used to inspect symbol tables and other debug-related sections. This is important for debugging tools and for understanding the relationship between high-level source code and low-level machine code.

Inspecting section contents is helpful for low-level analysis, especially when you need to understand how specific data or code is laid out in memory. It complements the information provided by disassembly (`-d`) by giving you a more direct view of the raw binary data stored in different sections of your executable.

```
arm-none-eabi-objdump -s .\build\0x0001_hello-world.elf | less

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

```
Contents of section .boot2:
 10000000 00b5324b 21205860 98680221 88439860 ..2K! X`.h.!.C.`
 10000010 d8601861 58612e4b 00219960 02215961 .`.aXa.K.!.`.!Ya
 10000020 0121f022 99502b49 19600121 99603520 !!".P+I.`!.`5
 10000030 00f044f8 02229042 14d00621 196600f0 ..D..".B...!.f..
 10000040 34f8196e 01211966 00201866 1a6600f0 4..n.!.f. .f.f..
 10000050 2cf8196e 196e196e 052000f0 2ff80121 ,..n.n.n. ../.!
 10000060 0842f9d1 00219960 1b491960 00215960 .B...!.`.I.`.!Y`
 10000070 1a491b48 01600121 9960eb21 1966a021 .I.H.`!.`.!.f.!
 10000080 196600f0 12f80021 99601649 14480160 .f.....!.`.I.H.`
 10000090 01219960 01bc0028 00d00047 12481349 .!.`...(...G.H.I
```

...

Like many of the previous commands, we are only showing the first page.

Let's explore our next command.

The `arm-none-eabi-readelf` command is a tool that displays information about ELF (Executable and Linkable Format) files. The `-a` option in the command stands for "all," and it instructs `readelf` to display all available information about the ELF file. The `| less` part of the

command pipes the output through the less command, allowing you to scroll through the information one screen at a time.

Here's a breakdown of what the command does:

arm-none-eabi-readelf -a .\build\0x0001_hello-world.elf | less

arm-none-eabi-readelf: This is the command for reading and displaying information about ELF files in the ARM embedded toolchain.

-a: This option tells readelf to display all available information about the ELF file. This includes information about the ELF header, program headers, section headers, symbol tables, and more.

.\build\0x0001_hello-world.elf: This is the path to the ELF file that you want to analyze. Replace this with the actual path to your compiled program.

| less: The pipe (|) operator is used to send the output of readelf to the less command, allowing you to scroll through large amounts of text one screen at a time.

The -a option provides a comprehensive overview of the ELF file. It includes information about the ELF header, program headers, section headers, symbol tables, and more. This is valuable for understanding the structure and contents of the binary.

The ELF header contains essential information about the binary, such as the architecture, entry point, program header table offset, section header table offset, and other metadata.

Program Headers:

Program headers provide information about how the ELF file should be loaded into memory. This includes details about the different segments of the program, such as code and data segments.

Section headers provide information about the various sections in the ELF file, including code sections, data sections, symbol tables, and more. This helps in understanding the layout of data and code in the binary.

Symbol tables contain information about symbols (functions, variables) present in the binary. This is crucial for debugging and understanding the relationship between high-level source code and low-level machine code.

When running the command, you will see a detailed output that includes information about the ELF header, program headers, section

headers, symbol tables, and other relevant information about the structure and content of your ELF file.

This command is particularly useful for gaining a deep understanding of the ELF file, especially when you need detailed information beyond what is provided by tools like `objdump`. It's a versatile tool for inspecting various aspects of the binary, making it valuable in the context of embedded systems development and low-level programming.

```
arm-none-eabi-readelf -a .\build\0x0001_hello-world.elf | less
```

ELF Header:

```
Magic:  7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00
Class:                                     ELF32
Data:                                     2's complement, little endian
Version:                                 1 (current)
OS/ABI:                                 UNIX - System V
ABI Version:                             0
Type:                                    EXEC (Executable file)
Machine:                                ARM
Version:                                0x1
Entry point address:                     0x100001e9
Start of program headers:                 52 (bytes into file)
Start of section headers:                 524904 (bytes into file)
Flags:                                    0x5000200, Version5 EABI, soft-float ABI
Size of this header:                      52 (bytes)
Size of program headers:                  32 (bytes)
Number of program headers:                 3
Size of section headers:                  40 (bytes)
Number of section headers:                 26
Section header string table index: 25
```

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.boot2	PROGBITS	10000000	001000	000100	00	AX	0	0	1
[2]	.text	PROGBITS	10000100	001100	006818	00	AX	0	0	8
[3]	.rodata	PROGBITS	10006918	007918	001758	00	A	0	0	8
[4]	.binary_info	PROGBITS	10008070	009070	000028	00	WA	0	0	4
[5]	.ram_vector_table	PROGBITS	20000000	009cb8	0000c0	00	W	0	0	4

...

Like the rest, this is just a small section you can try this out on your own to get the full contents.

There are two more useful commands to review.

The `arm-none-eabi-nm` command is a tool that displays symbol information from object files or executables. The command you provided, `arm-none-eabi-nm .\build\0x0001_hello-world.elf | less`, is using `nm` to display symbol information for the specified ELF

(Executable and Linkable Format) file and piping the output through the less command for easier navigation.

```
arm-none-eabi-nm .\build\0x0001_hello-world.elf | less
```

arm-none-eabi-nm: This is the command for displaying symbol information from object files or executables in the ARM embedded toolchain.

.\build\0x0001_hello-world.elf: This is the path to the ELF file that you want to analyze. Replace this with the actual path to your compiled program.

| less: The pipe (|) operator is used to send the output of nm to the less command, allowing you to scroll through large amounts of text one screen at a time.

The nm displays information about symbols in the ELF file. Symbols include functions, variables, and other program entities. This information is crucial for understanding the structure of your program at the symbol level.

For each symbol, nm provides its address in memory. This is useful for understanding where functions and variables are located within the program's memory space.

Symbols are categorized into different types, such as functions, variables, or sections. This helps you quickly identify the role of each symbol in your program.

Symbol information is essential for debugging. When you encounter issues in your program, knowing the addresses and types of symbols can aid in identifying problems and understanding the program's behavior.

nm is often integrated into build systems or used as part of the development process to inspect and analyze ELF files. It can be part of scripts or automation workflows for examining symbols and addresses.

Example Output:

You will see an output that lists symbols along with their addresses and types. Here's a simplified example:

```
08000100 T _start
08000104 T main
08000120 T printf
08000200 D data_variable
T: Indicates a code symbol (function).
```

D: Indicates a data symbol (variable).

08000100, 08000104, etc.: The address in memory.

By inspecting the symbol table, you can gain insights into the structure of your program, understand the memory layout, and use the information for debugging and analysis. It's a valuable tool for low-level programming and embedded systems development.

```
arm-none-eabi-nm .\build\0x0001_hello-world.elf | less
```

```
10006888 t __aeabi_idiv0_veneer
10006858 t __aeabi_ldiv0_veneer
20000b30 t __assert_func_veneer
20000b20 t __wrap__aeabi_lmul_veneer
20000b60 t __wrap_memcpy_veneer
20000b50 t __wrap_memset_veneer
10006908 t __hw_endpoint_buffer_control_update32_veneer
10003534 t __aeabi_bits_init
10003a76 t __aeabi_dfcmlpe_guts
10003bc0 T __aeabi_double_init
10003d00 T __aeabi_float_init
20000b08 W __aeabi_idiv0
20000b08 W __aeabi_ldiv0
10003d88 T __aeabi_mem_init
10002290 T __assert_func
20000b90 t __best_effort_wfe_or_timeout_veneer
10007c8c r __bi_188.0
100002ac t __bi_22
100002a0 t __bi_30
10007be8 r __bi_33.4
10007bf4 r __bi_34.5
10000294 t __bi_38
10007b7c r __bi_44
10000288 t __bi_50
10007b88 r __bi_75
10007b94 r __bi_81
10008094 d __bi_ptr188.4
10008070 d __bi_ptr22
10008074 d __bi_ptr30
...
```

When you run on your own you will see the entire contents.

Our final command is the strings command.

The arm-none-eabi-strings command is used to extract printable character sequences (strings) from binary files. When applied to an ELF (Executable and Linkable Format) file, it can reveal human-readable strings embedded within the binary. The | less part of the command pipes the output through the less command, allowing you to scroll through large amounts of text one screen at a time.

```
arm-none-eabi-strings .\build\0x0001_hello-world.elf | less
```

arm-none-eabi-strings: This is the command for extracting printable strings from binary files in the ARM embedded toolchain.

`.\build\0x0001_hello-world.elf`: This is the path to the ELF file that you want to analyze. Replace this with the actual path to your compiled program.

| less: The pipe (|) operator is used to send the output of strings to the less command, allowing you to scroll through large amounts of text one screen at a time.

The strings command helps you identify and inspect human-readable text within the binary. This includes strings used in your code, such as debug messages, error messages, and other informative text.

Extracted strings can be useful for debugging and analysis. By reviewing the strings present in the binary, you may gain insights into the behavior of the program and identify specific points of interest.

It allows you to verify that constants or string literals in your code are present in the binary as expected. This is particularly useful for confirming that messages and constants defined in your source code are correctly included in the compiled binary.

Analyzing strings in a binary is also a common practice for security analysis. It can help identify sensitive information or hardcoded credentials that should not be exposed in a production binary.

You'll see a mix of strings that are part of your code, standard library functions, and potentially other information embedded in the binary. By examining these strings, you can gain a better understanding of the content and functionality of the compiled program. Keep in mind that not all strings extracted may be directly visible in your source code; some might be library or system-related.

```
arm-none-eabi-strings .\build\0x0001_hello-world.elf | less
```

```
2K! X`  
aXa.K  
pG      H  
`SL{  
RPT"  
""@      *  
K#+p  
"iF(  
FWNFEEF  
FdDch
```

```
#hZ@
`CFc`BF
!hy@
-J.H
*K,JZb
`'K[l
XpGD
h@"b@
F`DA`
FaD      J
K#CC
FWNFEEF
FRFAFHF
VKVJ
VKSJ
GKCJ
+K+J
KZFAFR
RASC
...
```

Keep parsing through the output. You should see our hello, world string in there.

You can use grep to find specifically what you are looking for like this.

```
arm-none-eabi-strings .\build\0x0001_hello-world.elf | grep -i "hello"
```

```
hello, world
0x0001_hello-world
C:/Users/mytec/Documents/Embedded-Hacking/0x0001_hello-world
C:/Users/mytec/Documents/Embedded-Hacking/0x0001_hello-world/main.c
C:\Users\mytec\Documents\Embedded-Hacking\0x0001_hello-world\build
```

I know this was a longer chapter but please take the time and understand these tools and getting a better understanding of what firmware is and in this case how it is specific to the M0+ and RP2040 MCU.

In our next lesson we will have an intro to variables.

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 layed 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 `main.c` file as this is located in the main folder.

```
#include <stdio.h>
#include "pico/stdlib.h"
#include "pico/cyw43_arch.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. 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;
```

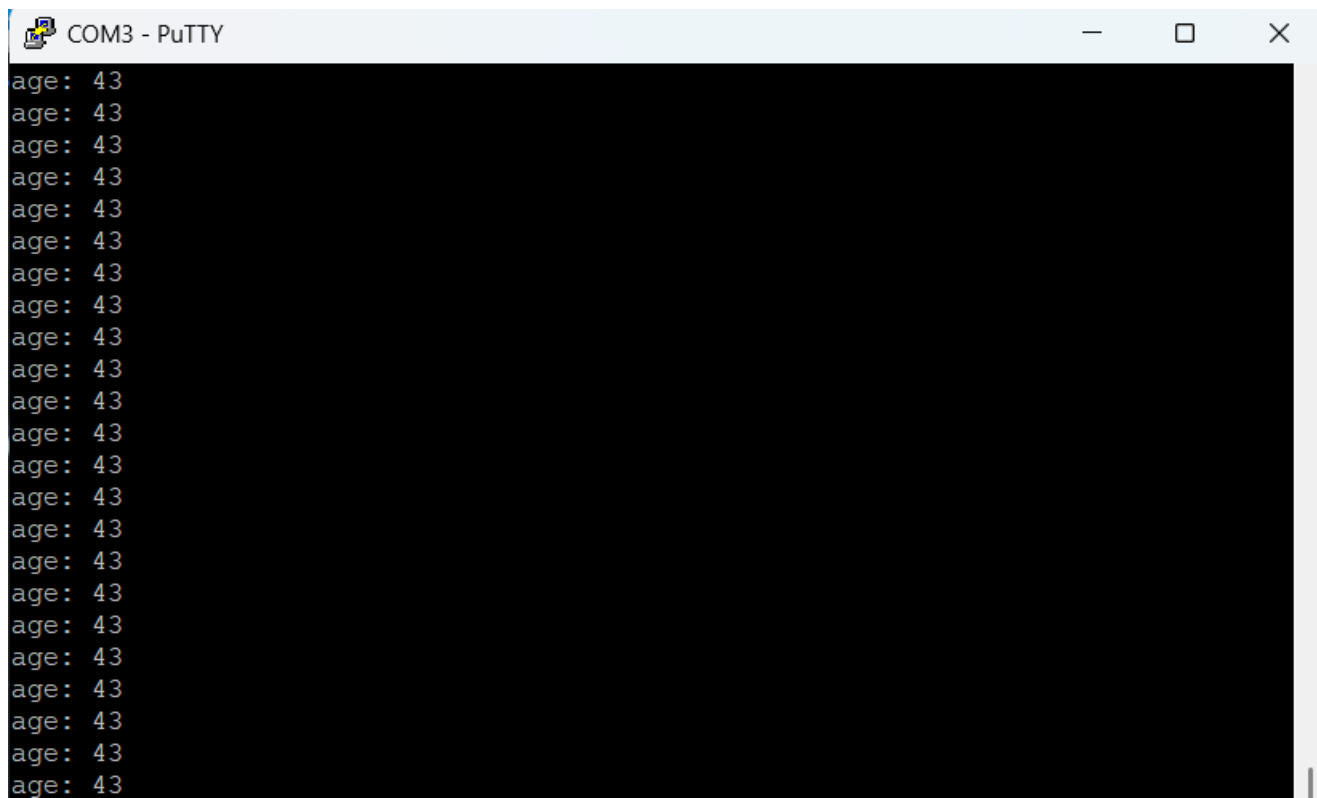
We start by declaring and initializing the variable to hold a 1 byte unsigned integer and assign the value of 42 to it.

```
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.



```
COM3 - PuTTY
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
```

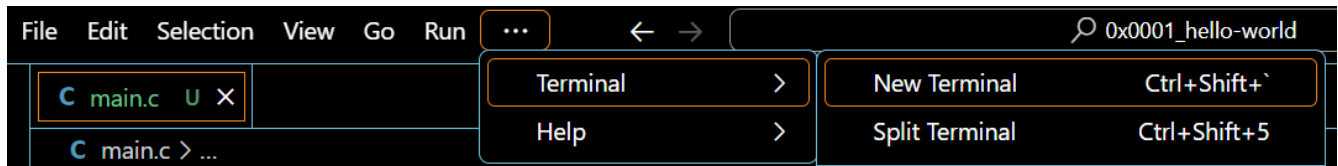
In our next chapter we will debug this.

Chapter 6: Debugging Intro To Variables

Today we debug!

Lets run OpenOCD to get our remote debug server going.

Open up a terminal within VSCode.



```
cd ..  
.\openocd.ps1
```

If you are on MAC or Linux, simply run the below command.

```
openocd -f interface/cmsis-dap.cfg -f target/rp2040.cfg -c "adapter speed 5000"
```

Open up a new terminal and cd `.\build\ dir` and then run the following.

```
arm-none-eabi-gdb .\0x0005_intro-to-variables.bin
```

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

```
target remote :3333
```

We need to halt the currently running binary.

```
monitor reset halt
```

I want to make sure you REALLY understand what is going on. The XIP is the start of FLASH at `0x10000000`. The entry point to our vector table begins with the MSP or master stack pointer and this will be at `0x10000100`.

```
(gdb) x/x 0x10000100  
0x10000100:      0x20042000
```

This should make sense if not please re-read the last chapter.

We are dealing with a raw binary with no symbols so we have to find our main function.

Let's examine instructions from the beginning of our vector table.

```
x/1000i 0x10000100
```

NOTE: ADDRESSES WILL VARY FROM MACHINE TO MACHINE

We see something interesting here.

```
0x10000308: push    {r4, lr}
=> 0x1000030a: bl      0x10004064
0x1000030e: ldr     r0, [pc, #8] ; (0x10000318)
0x10000310: movs    r1, #43 ; 0x2b
0x10000312: bl      0x1000404c
0x10000316: b.n     0x1000030e
```

Let's do the following.

```
set $pc = 0x10000308
b *0x10000308
c
```

Then we can look at a few instructions.

```
(gdb) x/6i $pc
=> 0x10000308: push    {r4, lr}
0x1000030a: bl      0x10004064
0x1000030e: ldr     r0, [pc, #8] ; (0x10000318)
0x10000310: movs    r1, #43 ; 0x2b
0x10000312: bl      0x1000404c
0x10000316: b.n     0x1000030e
(gdb)
```

We see 43 decimal being moved into r1 however we do not see the 42 initialized value as the compiler optimized that away as it then assigned 43 into it to be called by the printf function.

In our next lesson we will hack this!

Chapter 7: Hacking Intro To Variables

Let's pick up where we were in our last lesson and set a breakpoint on the line after the assignment of decimal 43.

```
(gdb) b *0x10000312
Breakpoint 2 at 0x10000312
(gdb) c
```

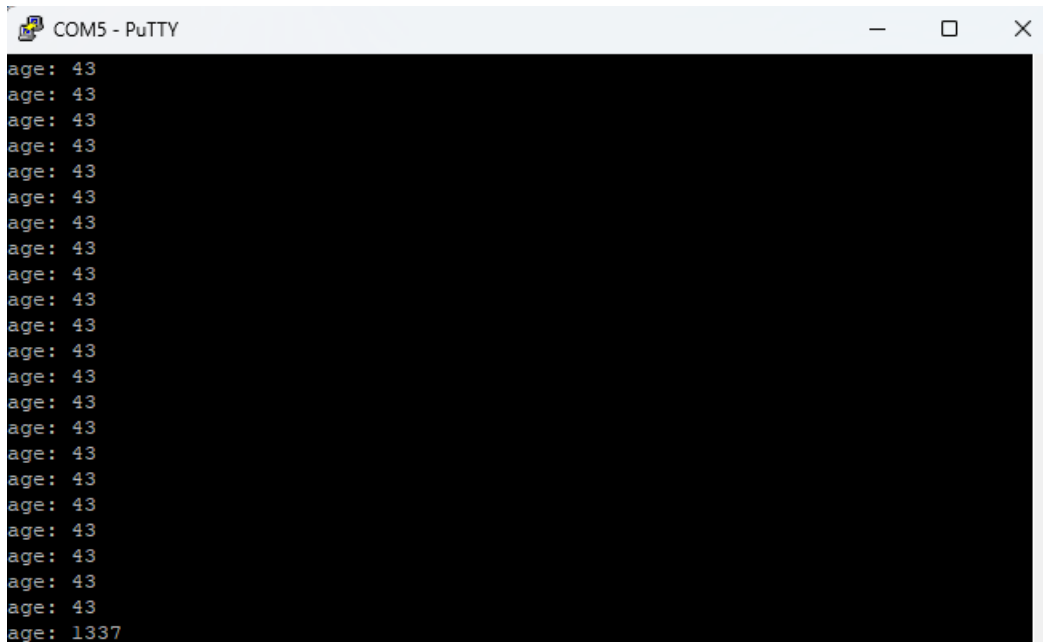
Let's examine what is in r1.

```
(gdb) x/x $r1
0x2b: 0x701c18d1
```

Let's hack r1!

```
(gdb) set $r1 = 1337
(gdb) x/x $r1
0x539: 0xc3702a35
```

Let's open PuTTY or our terminal emulator of choice.



```
COM5 - PuTTY
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: 1337
```

Success! We hacked it! In our next lesson we will look at uninitialized variables.

Chapter 8: Uninitialized Variables

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

Let's open up our folder **0x0008_uninitialized-variables**.

Now let's review our **main.c** file as this is located in the main folder.

```
#include <stdio.h>
#include "pico/stdlib.h"
#include "pico/cyw43_arch.h"

int main(void)
{
    uint8_t age;

    stdio_init_all();

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

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.

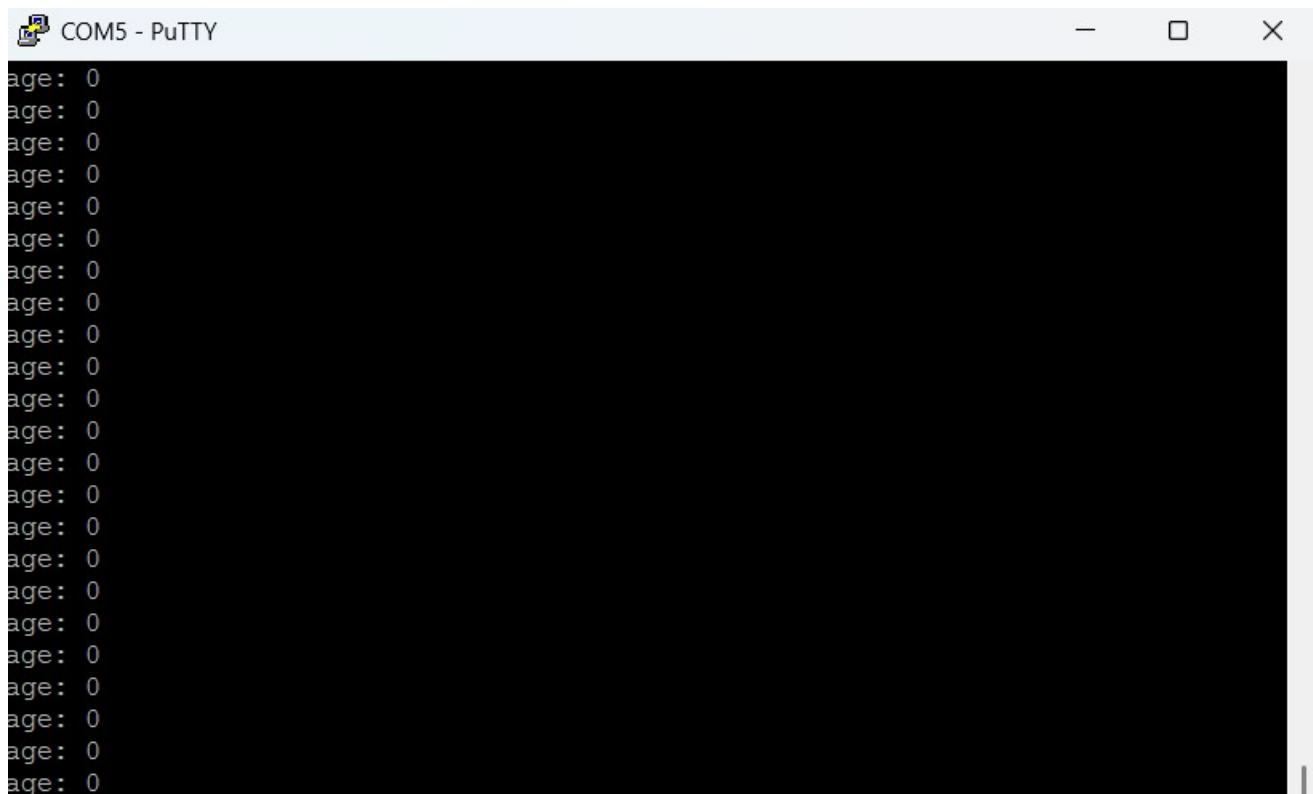
The entire `.bss` section is assigned an address in RAM via the linker and does not reside in the binary or flash.

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 flash and examine the binary.



```
COM5 - PuTTY
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
```

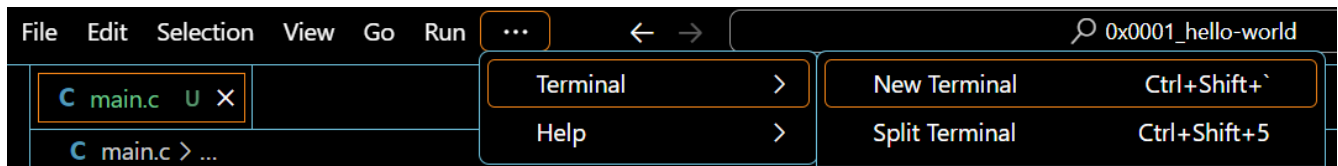
In our next lesson we will debug this.

Chapter 9: Debugging Uninitialized Variables

Today we debug!

Lets run OpenOCD to get our remote debug server going.

Open up a terminal within VSCode.



```
cd ..  
.\openocd.ps1
```

If you are on MAC or Linux, simply run the below command.

```
openocd -f interface/cmsis-dap.cfg -f target/rp2040.cfg -c "adapter speed 5000"
```

Open up a new terminal and cd `.\build\` dir and then run the following.

```
arm-none-eabi-gdb .\0x0008_uninitialized_variables.bin
```

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

```
target remote :3333
```

We need to halt the currently running binary.

```
monitor reset halt
```

Remeber, the XIP is the start of FLASH at `0x10000000`. The entry point to our vector table begins with the MSP or master stack pointer and this will be at `0x10000100`.

```
(gdb) x/x 0x10000100  
0x10000100:      0x20042000
```

This should make sense if not please re-read the last few chapters.

We are dealing with a raw binary with no symbols so we have to find our main function.

Let's examine instructions from the beginning of our vector table.

```
x/1000i 0x10000100
```

NOTE: ADDRESSES WILL VARY FROM MACHINE TO MACHINE

We see something interesting at offset 304.

```
0x10000304: push    {r4, lr}
0x10000306: movs    r4, #0
0x10000308: bl      0x10004070
0x1000030c: movs    r1, r4
0x1000030e: ldr     r0, [pc, #8]    ; (0x10000318)
0x10000310: bl      0x10004058
0x10000314: b.n     0x1000030c
0x10000316: nop
```

We know that 308 offset is our C SDK init which we have seen before.

Let's set a breakpoint after the ldr instruction and see what is going on.

```
(gdb) b *0x10000310
Breakpoint 1 at 0x10000310
Note: automatically using hardware breakpoints for read-only addresses.
(gdb) c
Continuing.
Thread 1 "rp2040.core0" hit Breakpoint 1, 0x10000310 in ?? ()
(gdb) x/x $r0
0x10006920:    0x3a656761
(gdb) x/s $r0
0x10006920:    "age: %d\r\n"
```

We see our age variable however what does it store? If you look at the movs r4, #0 it tells us our answer. In the last chapter we talked about how initialized variables were init to 0.

In our next chapter we will hack this!