



FIRST EDITION – Chapter 3

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 within an STM32F401CC6 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-Connetor-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 1: hello, world

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

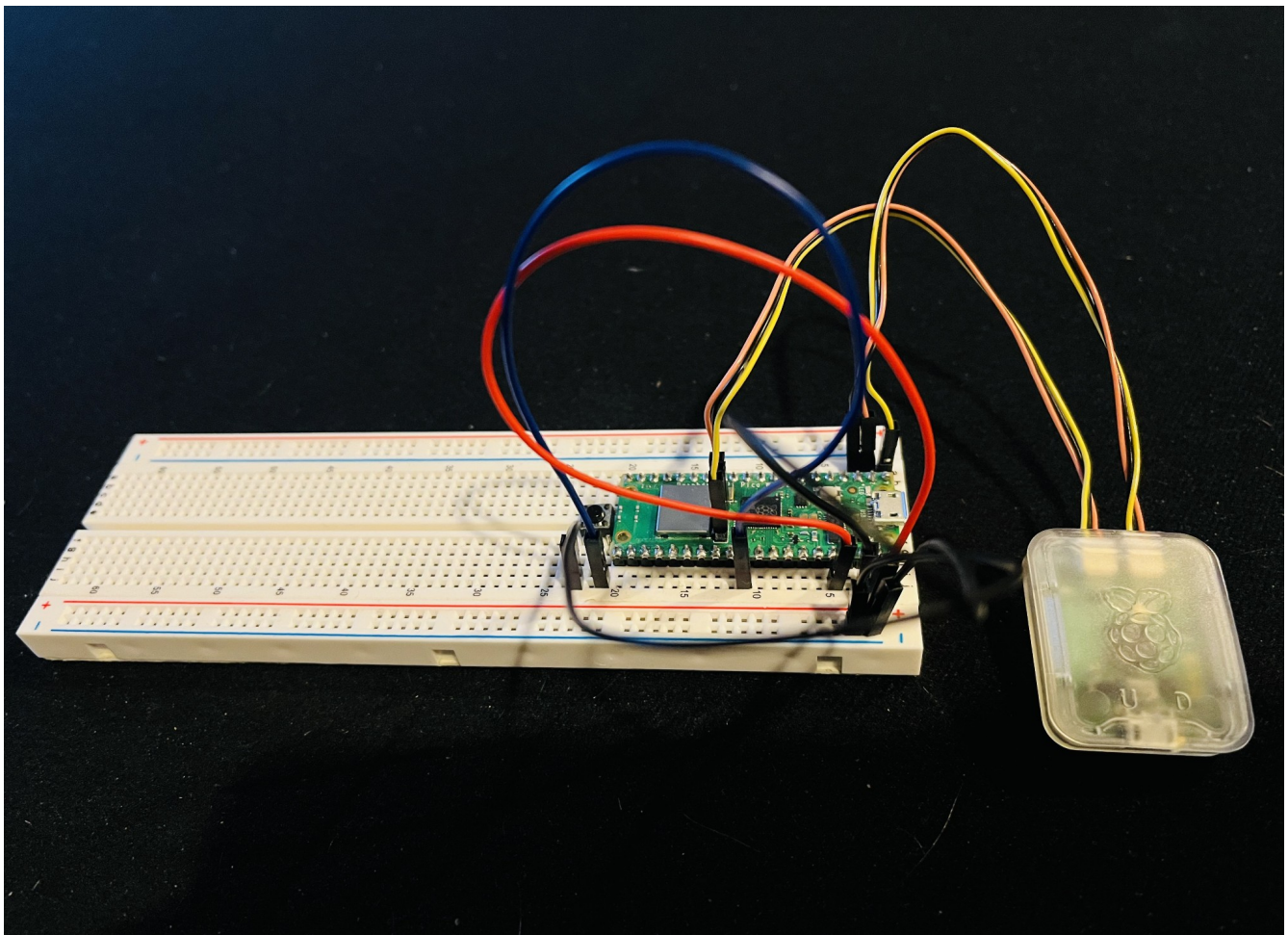
We will then reverse engineer each binary in GDB.

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.

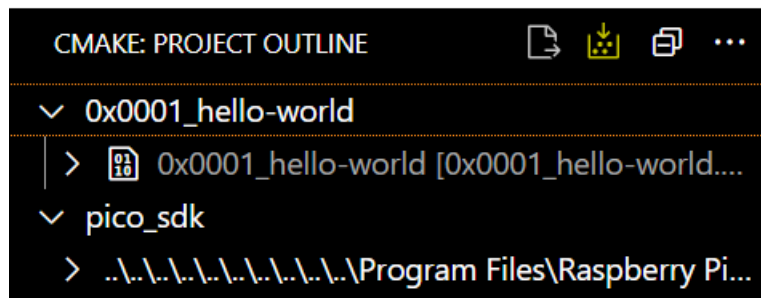
```
git clone https://github.com/mytechnotalent/Embedded-Hacking.git
```

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.

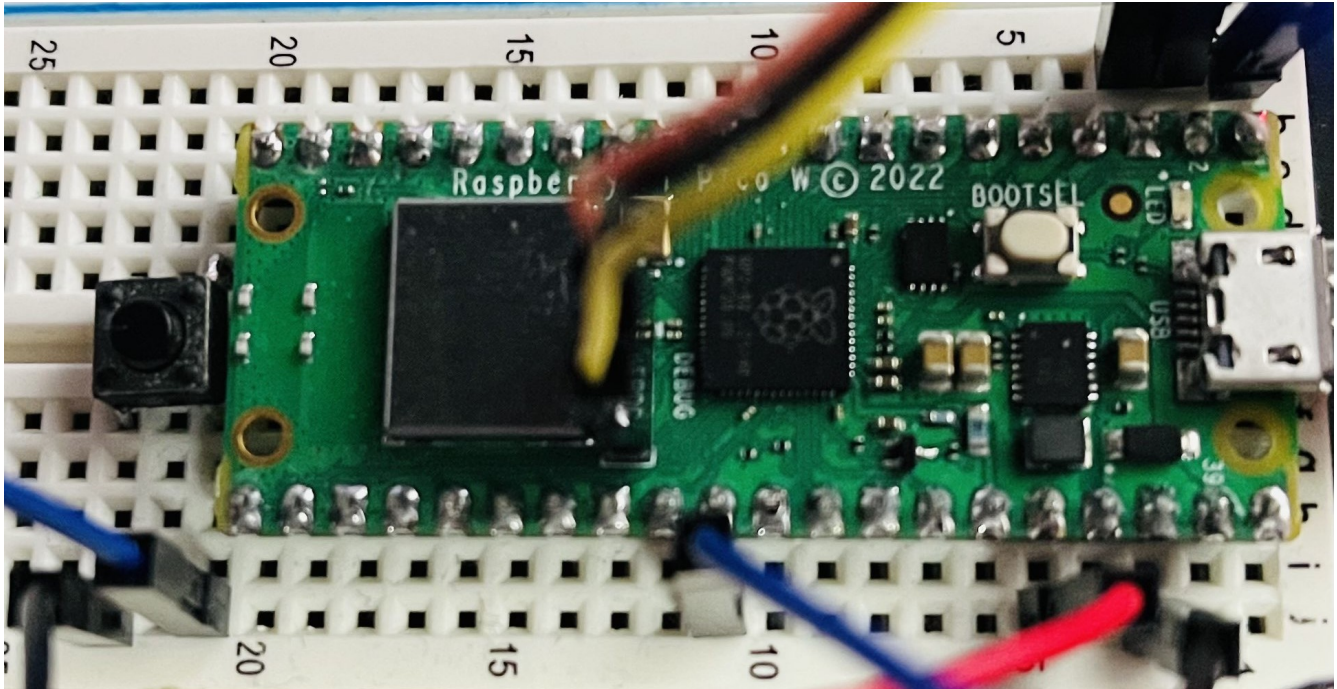
It may ask you for an active kit of which you will choose Pico ARM GCC.

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

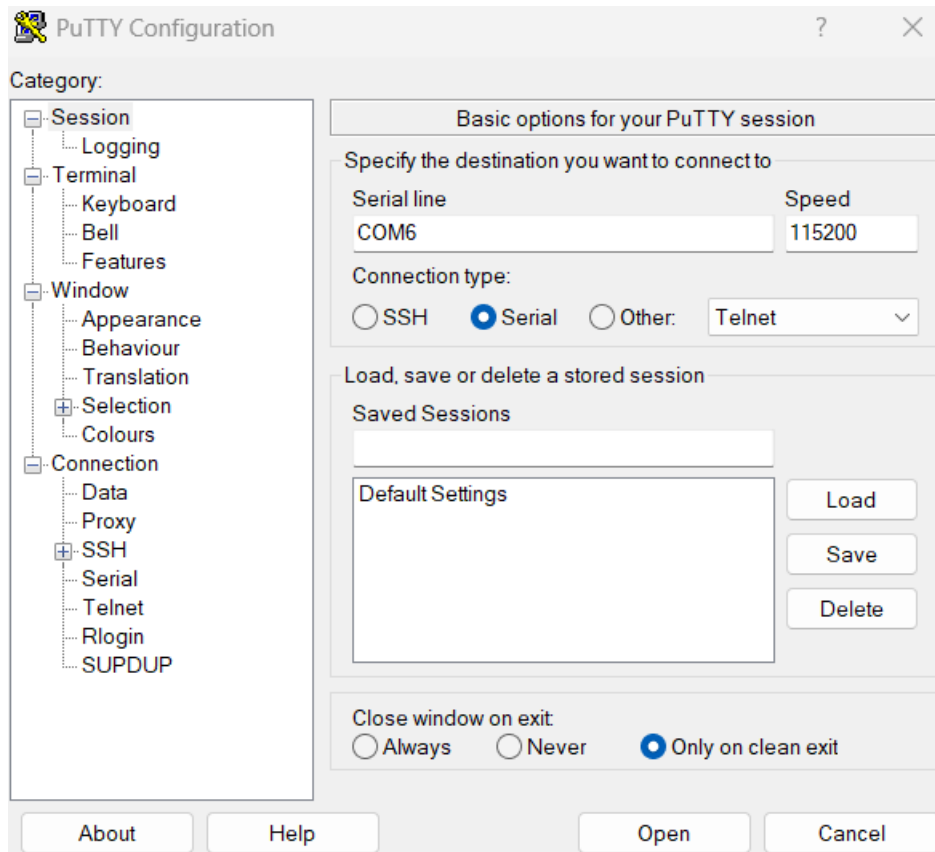
We need to download a serial monitor to interact with our Pico W. If you are on Windows download PuTTY as the link is below.

<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)
```

This 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.

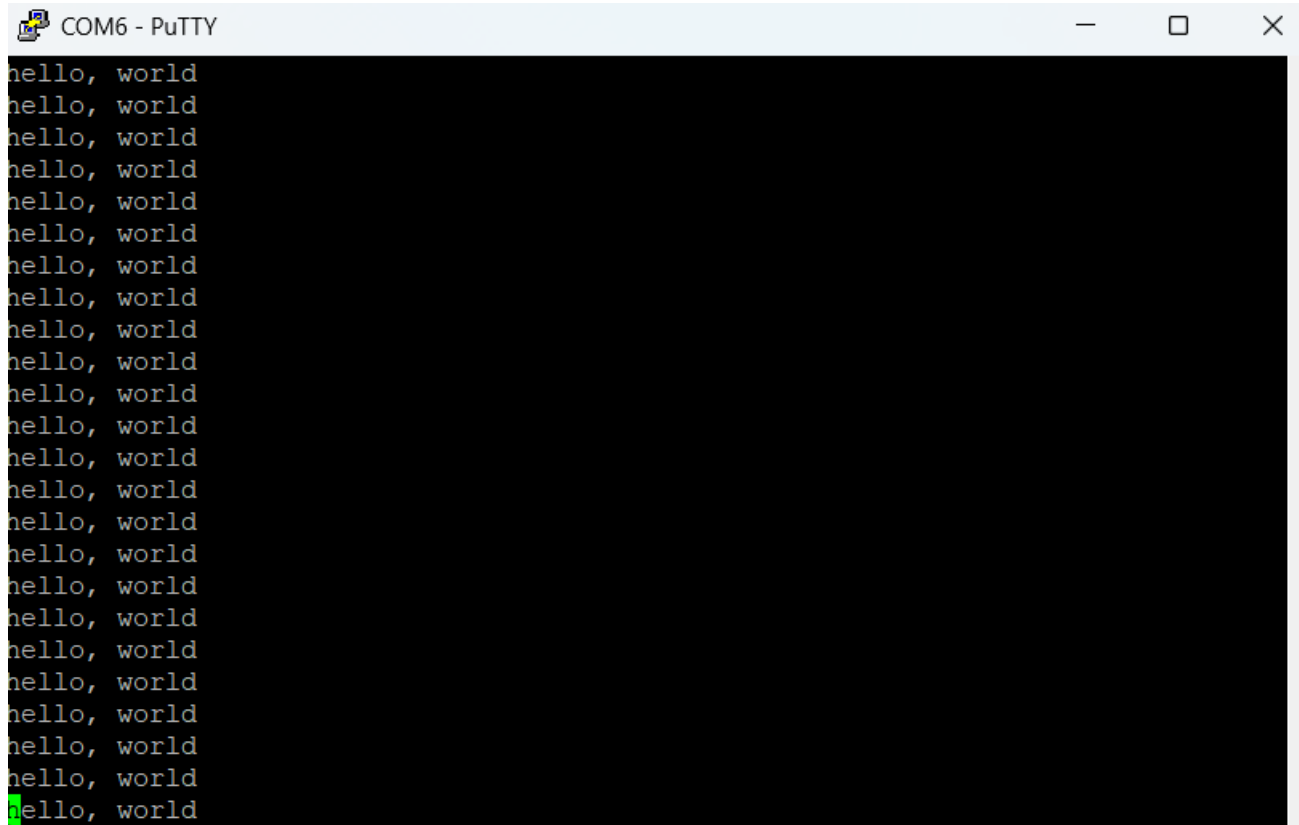
```
    while (true)
```

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 control buttons (minimize, maximize, close). The terminal area has a black background with yellow text. The text "hello, world" is printed on approximately 25 lines, with a green cursor visible at the end of the last line.

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.

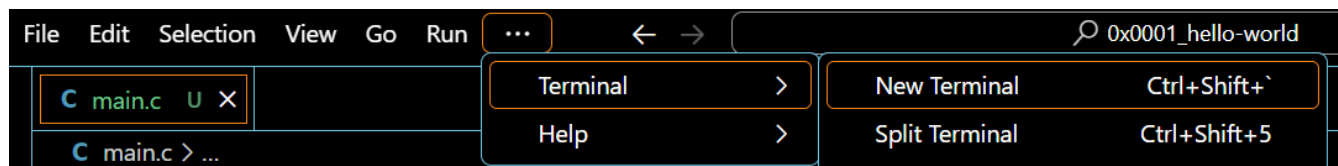
Assembler is not natural to everyone and I have another FREE book and primer on Embedded Assembler below if you feel you need a good primer. PLEASE take the time to read this book if you are new to this so that you can get the full benefit of this book.

<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.

```
(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 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/mytechtalent/Embedded-Assembler>

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

Chapter 3: Hacking hello, world

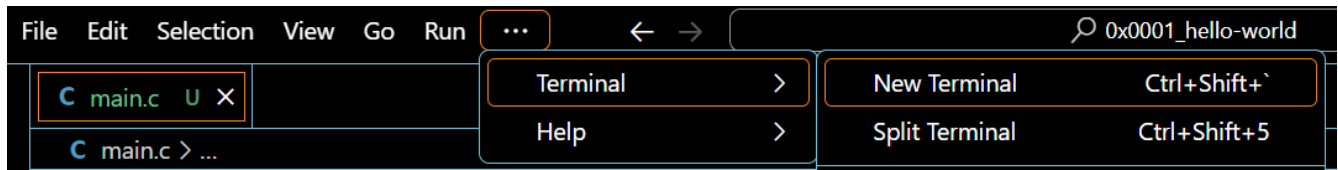
Today we hack!

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.

```

(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 have an introduction to variables.