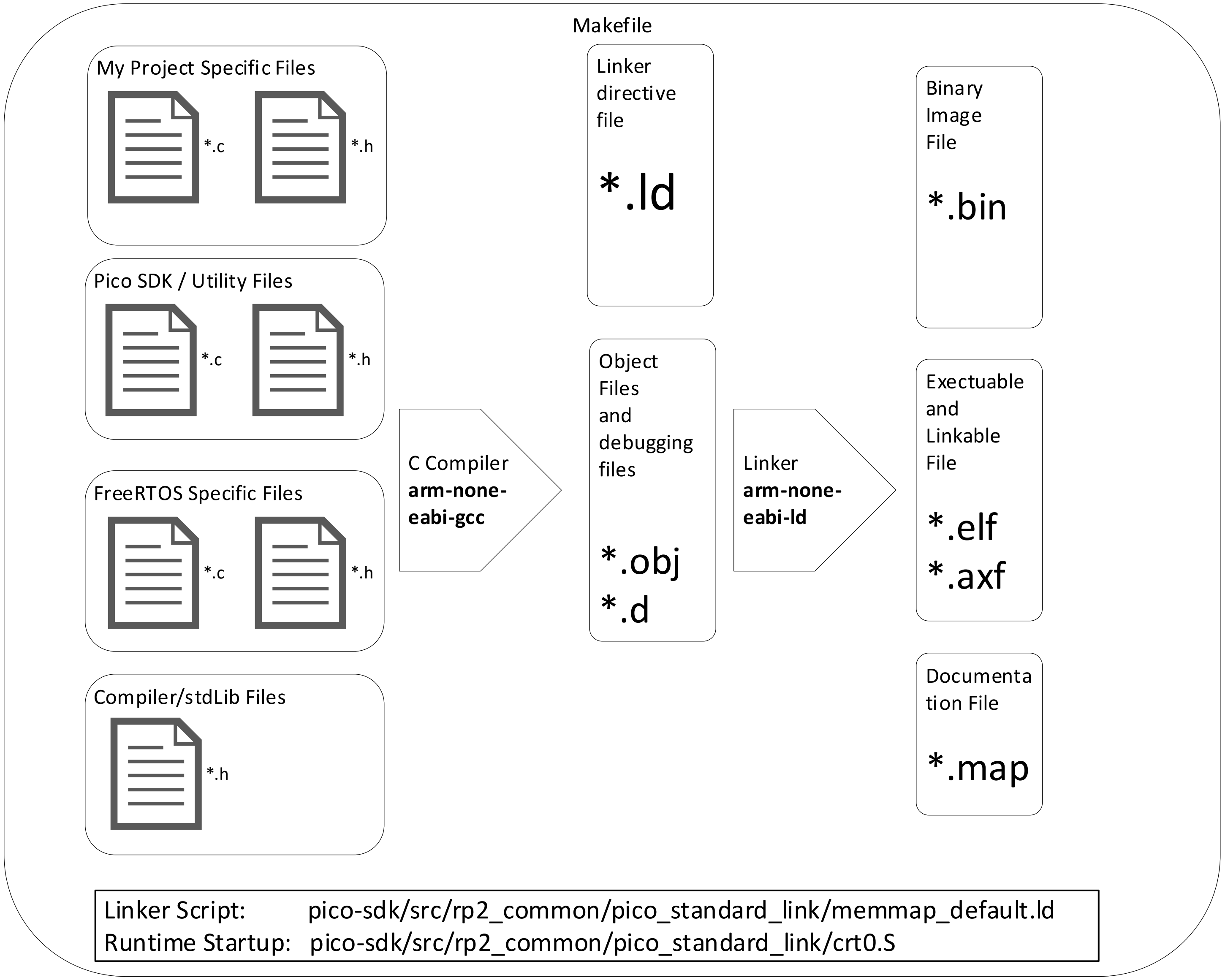
CS466 Lab03 Supplement

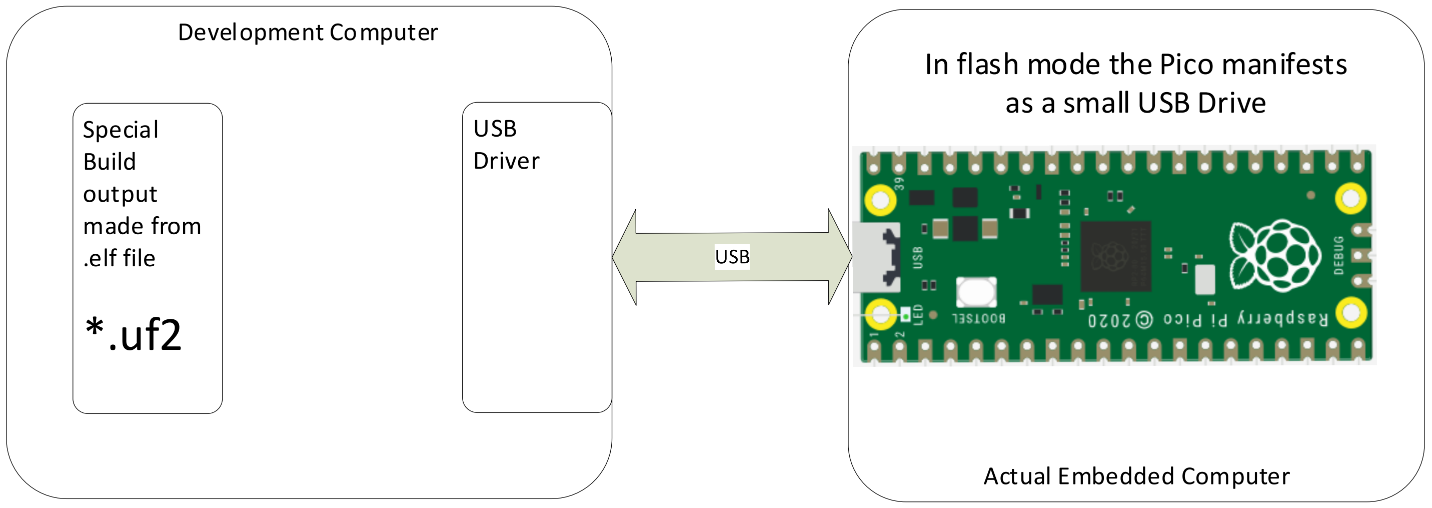
Build and Debug of lab3.c
on Pi_Pico using a second Pi-Pico as your CMSIS-DAP probe.
If you find errors in this deck please let me know.

Build Files


Makefile
My Project Specific Files
*.c
*.h
Pico SDK / Utility Files
*.c
*.h
FreeRTOS Specific Files
*.c
*.h
Compiler/stdLib Files
*.h
C Compiler **arm-none-eabi-gcc**
Linker directive file *.ld
Object Files and debugging files *.obj *.d
Linker **arm-none-eabi-ld**
Binary Image File *.bin
Exectuable and Linkable File *.elf *.axf
Documentation File *.map

Linker Script: pico-sdk/src/rp2_common/pico_standard_link/memmap_default.ld
Runtime Startup: pico-sdk/src/rp2_common/pico_standard_link/crt0.S

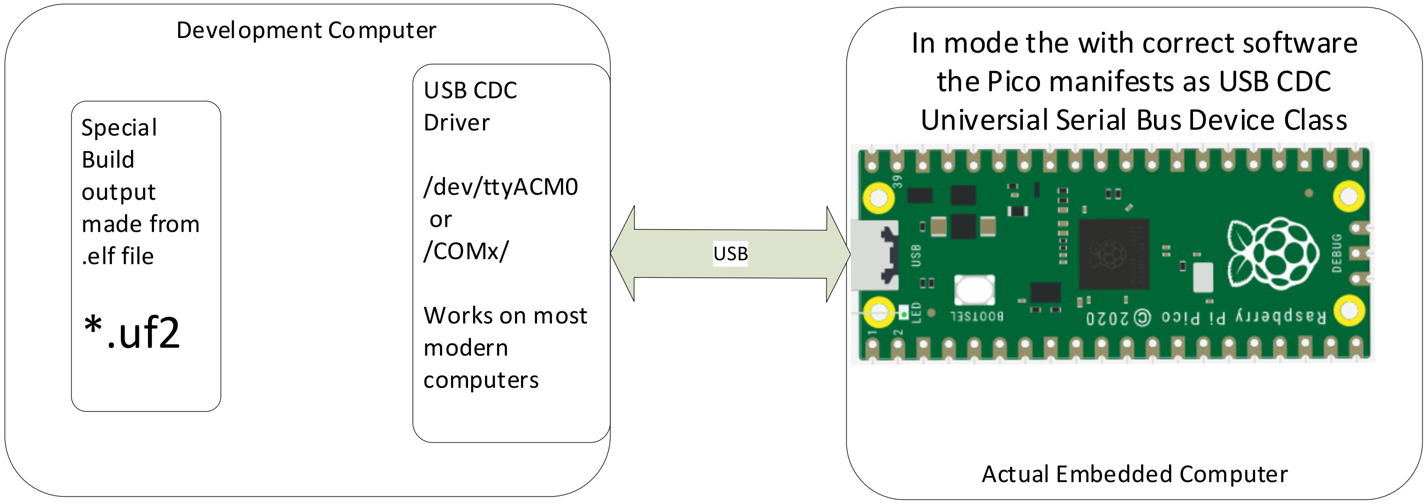
Normal Pseudo-Drive Flashing


Development Computer
Special Build output made from .elf file
*.uf2
USB Driver
USB
In flash mode the Pico manifests as a small USB Drive
Actual Embedded Computer

- The prg.uf2 file is created from some cmake rules
- The Pico in reflash-mode appears to the development computer as a USB drive
- To flash you just copy the .uf2 file to the pseudo-drive.
- The Pico will detect the .uf2 file, reflash the Pico and reboot into Run mode with your new image running.

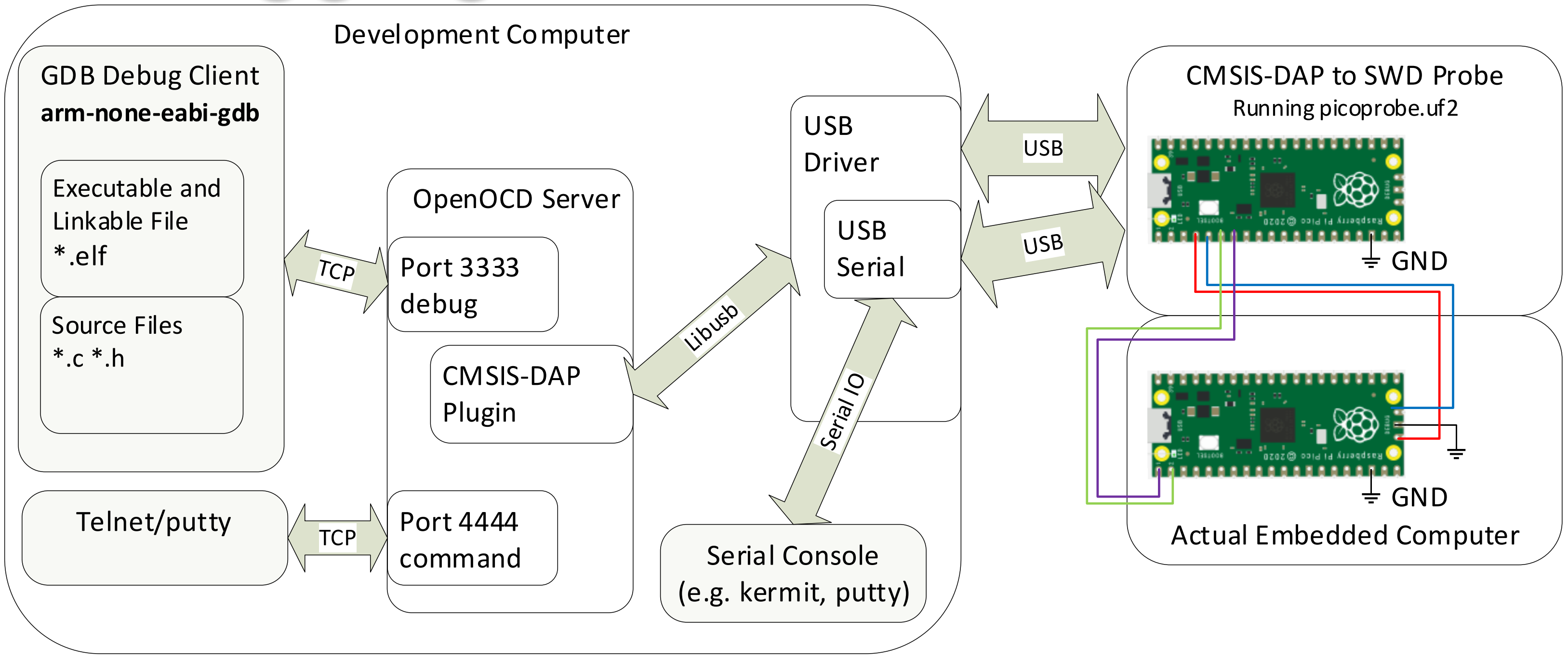
  - Note: USB-Drives are not supposed to change except under control of the computer they are plugged into. The Pico drive may cause some software on some computers to act strange.

Serial over USB


Development Computer
Special Build output made from .elf file
*.uf2
USB CDC Driver
/dev/ttyACM0
or
/COMx/
Works on most modern computers
USB
In mode the with correct software the Pico manifests as USB CDC Universial Serial Bus Device Class
Raspberry Pi Pico © 2020
BOOTSEL
USB
DEBUG
LED
Actual Embedded Computer

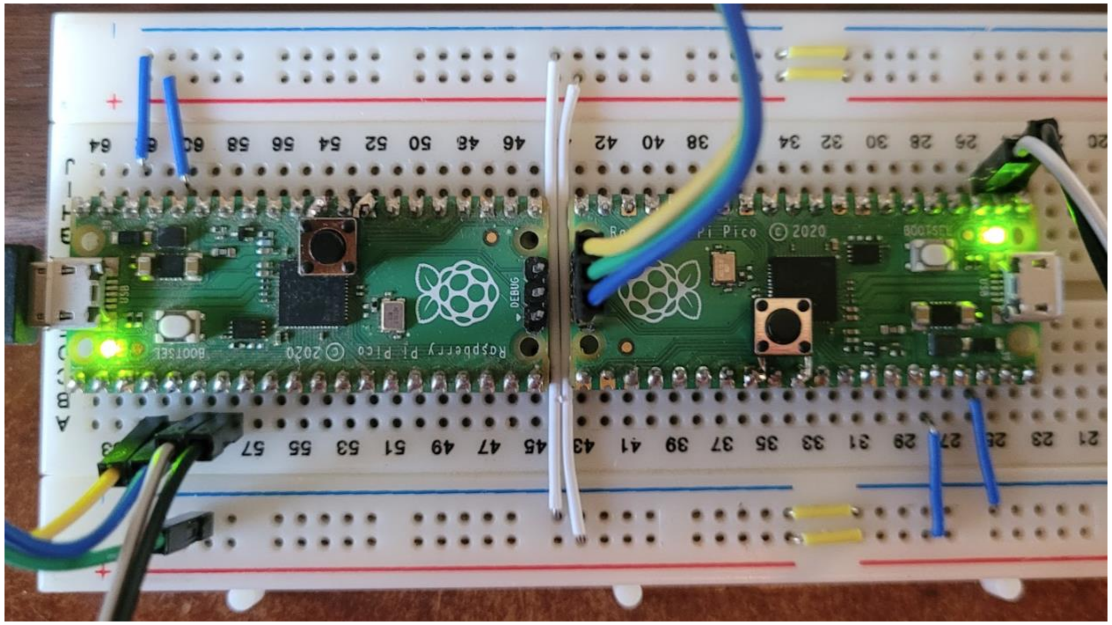
- If you build and include the TinyUSB code (added in lab2) the Pico SDK will present the USB device as a Universal Serial Bus device.
- The downside of this software solution is that the serial port will disconnect when you boot the Pico into Reflash Mode but it does save you from dedicating pins to serial data.
- There are lots of cheap Serial->USB devices that you can use that will keep the connection open across Pico boots. (e.g. available on Amazon)

Debugging over USB/CMSIS-DAP



- The Blue and Red lines are the Single Wire Debug (SWD) ARM Debug Interface.
    - (One may argue that it's not a single wire but that's what it's called)
- The Violet and Green lines are the Actual Serial between target and probe.
- Probe Pico is running picoprobe.uf2
- The OpenOCD server program and CMSIS-DAP Probe combine to emulate older probes.
- I generally run OpenOCD in its own window and leave it running
- I generally run GDB in its own window and through it Build and Reflash the target bypassing the .uf2 copy exercise.

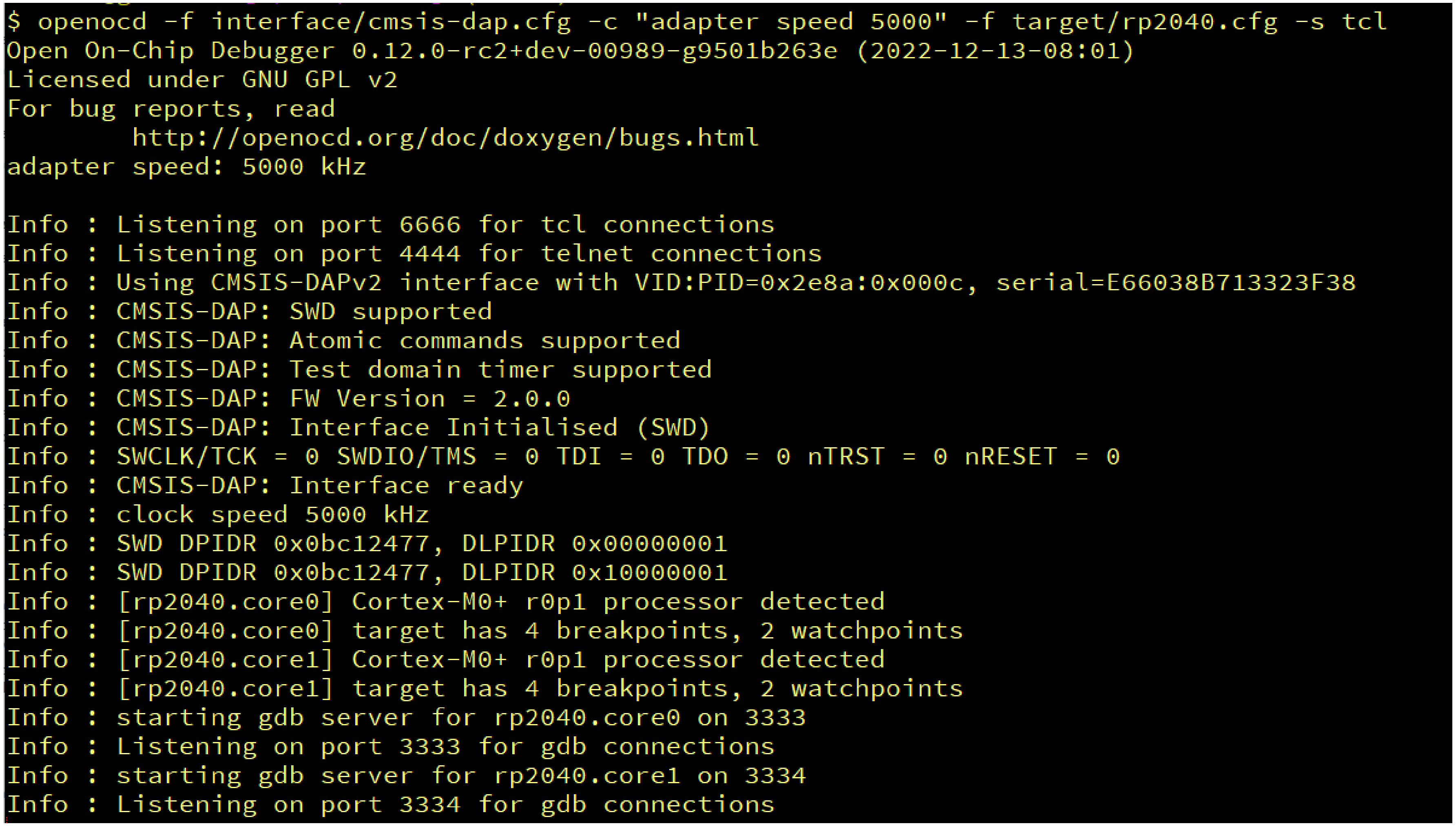
What my setup looks like



- The Pico on the left is my probe and target is on the right.
- From the Probe-pico I power the breaboard rails 3.3v power signal. Both boards are grounded and I have tied the top and bottom power busses together.
- Yellow is SWClock and Blue is SWData (this is close to I2C signaling bit is single-master and slightly different hardware layer descriptions)
- Physical target serial is connected to Probe Serial.
- You will need to load picoprobe.uf2 onto the Pico that you use as a probe. It is available here or in your lab3 folder.

Starting OpenOCD Server

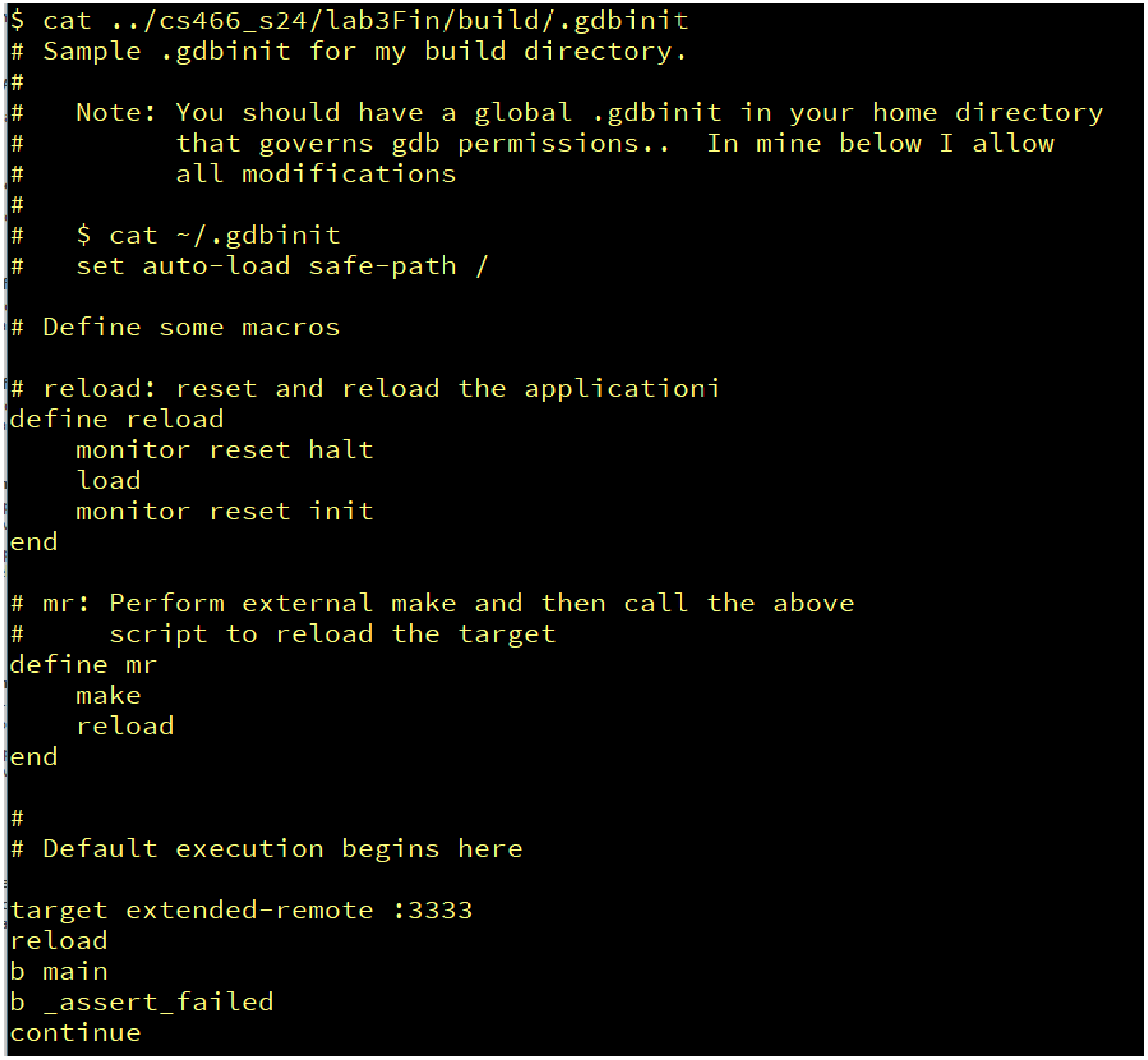
Power and electrical connections must be correct for this to work



```
$ openocd -f interface/cmsis-dap.cfg -c "adapter speed 5000" -f target/rp2040.cfg -s tcl
Open On-Chip Debugger 0.12.0-rc2+dev-00989-g9501b263e (2022-12-13-08:01)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
adapter speed: 5000 kHz

Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : Using CMSIS-DAPv2 interface with VID:PID=0x2e8a:0x000c, serial=E66038B713323F38
Info : CMSIS-DAP: SWD supported
Info : CMSIS-DAP: Atomic commands supported
Info : CMSIS-DAP: Test domain timer supported
Info : CMSIS-DAP: FW Version = 2.0.0
Info : CMSIS-DAP: Interface Initialised (SWD)
Info : SWCLK/TCK = 0 SWDIO/TMS = 0 TDI = 0 TDO = 0 nTRST = 0 nRESET = 0
Info : CMSIS-DAP: Interface ready
Info : clock speed 5000 kHz
Info : SWD DPIDR 0x0bc12477, DLPIDR 0x00000001
Info : SWD DPIDR 0x0bc12477, DLPIDR 0x10000001
Info : [rp2040.core0] Cortex-M0+ r0p1 processor detected
Info : [rp2040.core0] target has 4 breakpoints, 2 watchpoints
Info : [rp2040.core1] Cortex-M0+ r0p1 processor detected
Info : [rp2040.core1] target has 4 breakpoints, 2 watchpoints
Info : starting gdb server for rp2040.core0 on 3333
Info : Listening on port 3333 for gdb connections
Info : starting gdb server for rp2040.core1 on 3334
Info : Listening on port 3334 for gdb connections
```

Note the information about both cores and core0 is ready on port 3333

Setting up gdb (using .gdbinit)

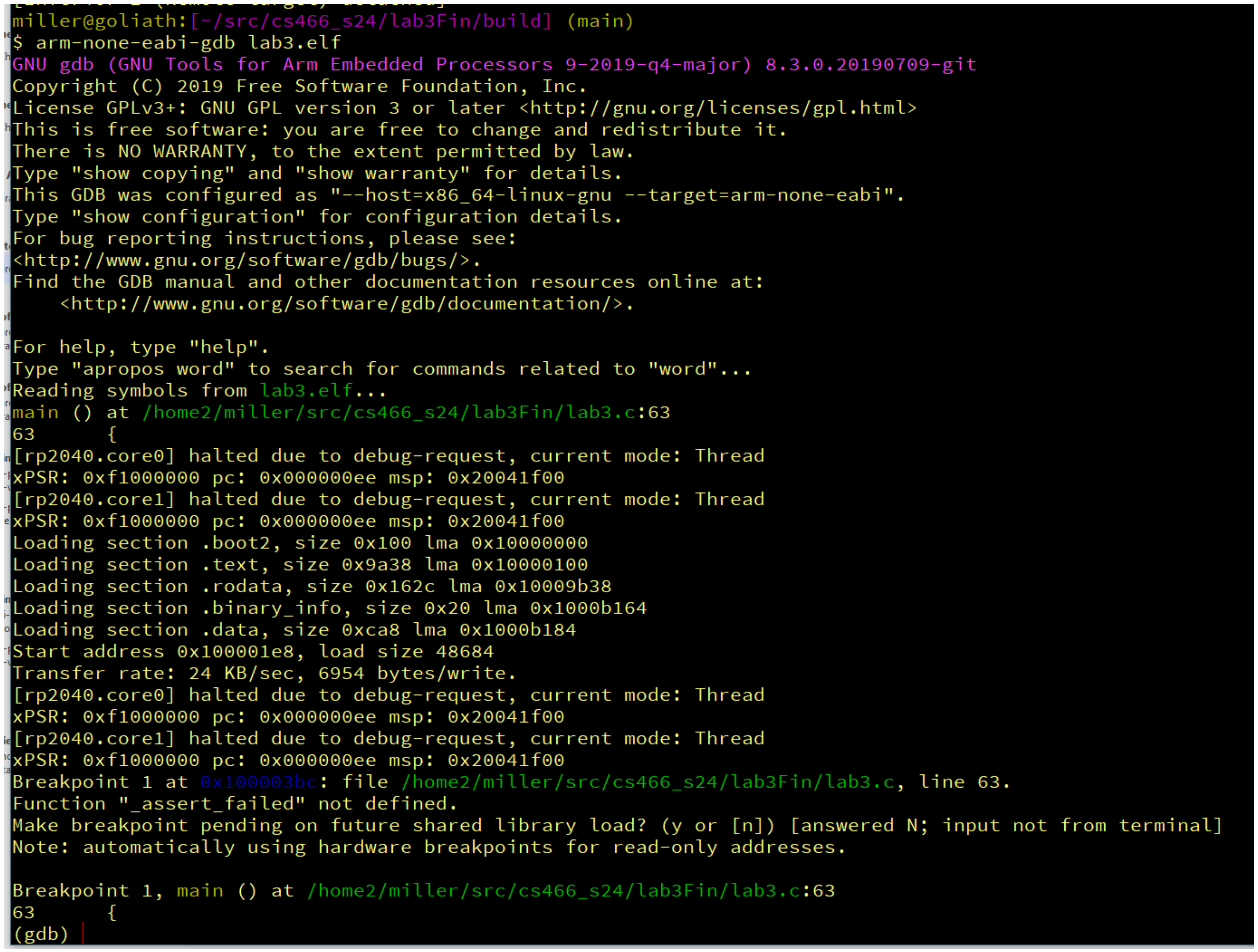


```
$ cat ../cs466_s24/lab3Fin/build/.gdbinit
# Sample .gdbinit for my build directory.
#
#   Note: You should have a global .gdbinit in your home directory
#         that governs gdb permissions..  In mine below I allow
#         all modifications
#
#   $ cat ~/.gdbinit
#   set auto-load safe-path /

# Define some macros

# reload: reset and reload the applicationi
define reload
    monitor reset halt
    load
    monitor reset init
end

# mr: Perform external make and then call the above
#     script to reload the target
define mr
    make
    reload
end

#
# Default execution begins here

target extended-remote :3333
reload
b main
b _assert_failed
continue
```

- There usually must be a global .gdbinit file
- Using macros under gdb makes life easier for repeated operations
- Look at the two macros I define here
- Using the ‘mr’ macro I do an external make and reload avoiding the whole .uf2/drive game

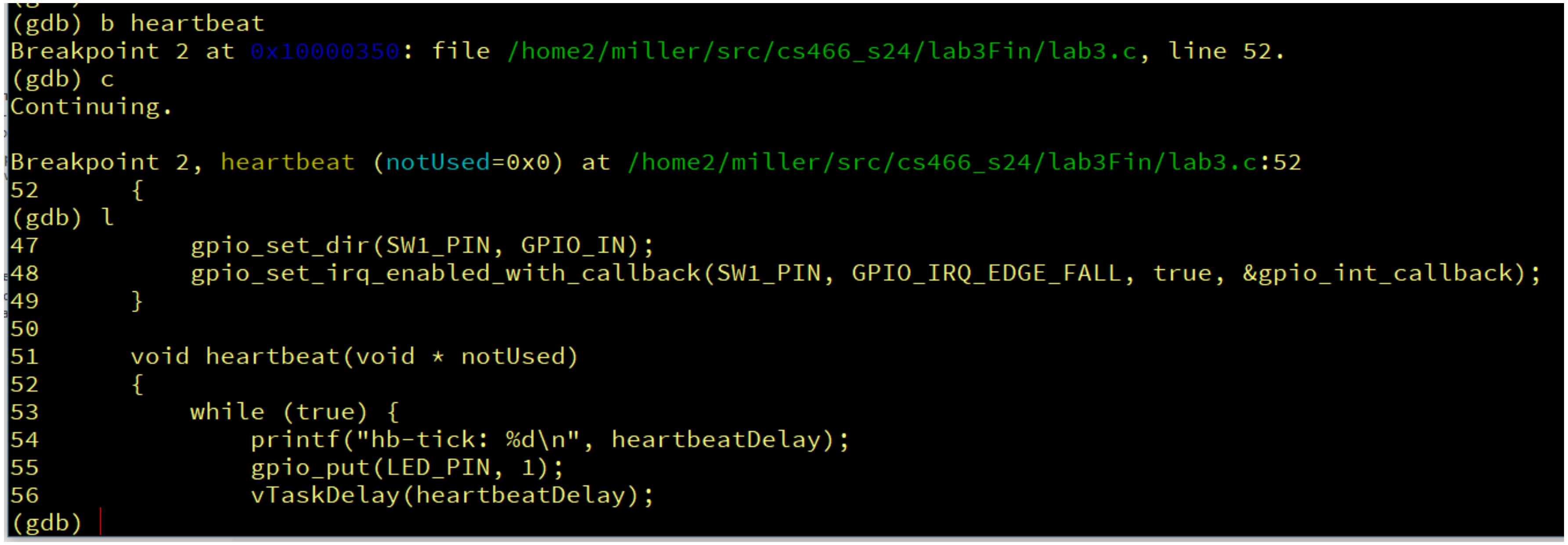
Starting gdb

(Runs until main() entry from startup code…)



```
miller@goliath:[~/src/cs466_s24/lab3Fin/build] (main)
$ arm-none-eabi-gdb lab3.elf
GNU gdb (GNU Tools for Arm Embedded Processors 9-2019-q4-major) 8.3.0.20190709-git
Copyright (C) 2019 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-linux-gnu --target=arm-none-eabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://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"...
Reading symbols from lab3.elf...
main () at /home2/miller/src/cs466_s24/lab3Fin/lab3.c:63
63      {
[rp2040.core0] halted due to debug-request, current mode: Thread
xPSR: 0xf1000000 pc: 0x000000ee msp: 0x20041f00
[rp2040.core1] halted due to debug-request, current mode: Thread
xPSR: 0xf1000000 pc: 0x000000ee msp: 0x20041f00
Loading section .boot2, size 0x100 lma 0x10000000
Loading section .text, size 0x9a38 lma 0x10000100
Loading section .rodata, size 0x162c lma 0x10009b38
Loading section .binary_info, size 0x20 lma 0x1000b164
Loading section .data, size 0xca8 lma 0x1000b184
Start address 0x100001e8, load size 48684
Transfer rate: 24 KB/sec, 6954 bytes/write.
[rp2040.core0] halted due to debug-request, current mode: Thread
xPSR: 0xf1000000 pc: 0x000000ee msp: 0x20041f00
[rp2040.core1] halted due to debug-request, current mode: Thread
xPSR: 0xf1000000 pc: 0x000000ee msp: 0x20041f00
Breakpoint 1 at 0x100003bc: file /home2/miller/src/cs466_s24/lab3Fin/lab3.c, line 63.
Function "_assert_failed" not defined.
Make breakpoint pending on future shared library load? (y or [n]) [answered N; input not from terminal]
Note: automatically using hardware breakpoints for read-only addresses.

Breakpoint 1, main () at /home2/miller/src/cs466_s24/lab3Fin/lab3.c:63
63      {
(gdb)
```

gdb break at heartbeat()



```
(gdb) b heartbeat
Breakpoint 2 at 0x10000350: file /home2/miller/src/cs466_s24/lab3Fin/lab3.c, line 52.
(gdb) c
Continuing.

Breakpoint 2, heartbeat (notUsed=0x0) at /home2/miller/src/cs466_s24/lab3Fin/lab3.c:52
52          {
(gdb) l
47              gpio_set_dir(SW1_PIN, GPIO_IN);
48              gpio_set_irq_enabled_with_callback(SW1_PIN, GPIO_IRQ_EDGE_FALL, true, &gpio_int_callback);
49          }
50
51          void heartbeat(void * notUsed)
52          {
53              while (true) {
54                  printf("hb-tick: %d\n", heartbeatDelay);
55                  gpio_put(LED_PIN, 1);
56                  vTaskDelay(heartbeatDelay);
(gdb)
```

- b sets a breakpoint
- c continues from a breakpoint
- r restarts the application from start
- s steps into a function
- n steps over a function
- l lists the relevant lines of code
- See the GDB cheat sheet for lots more commands. Google can find you many more

GDB Continued

- The command line/text interface is rough but with a little practice it becomes second nature.
- Most commercial packages provided by device manufacturers use this as a back end with a clunky GUI built on top of it.
- This article gives instruction how to use vscode as a front end.
- I normally get a GDB connection to an embedded target before I get a serial connection..
- If you wanted to monitor a variable in a loop you can do something like..
  - set a breakpoint at a line and attach it to a macro
  - macro can dump one or more variables or memory locations
  - at the end of the macro you can include the ‘continue’ command to automatically restart.. It’s slow but meets the needs of most serial debugging.