Lesson:7

- **Lesson 7: Debugging**

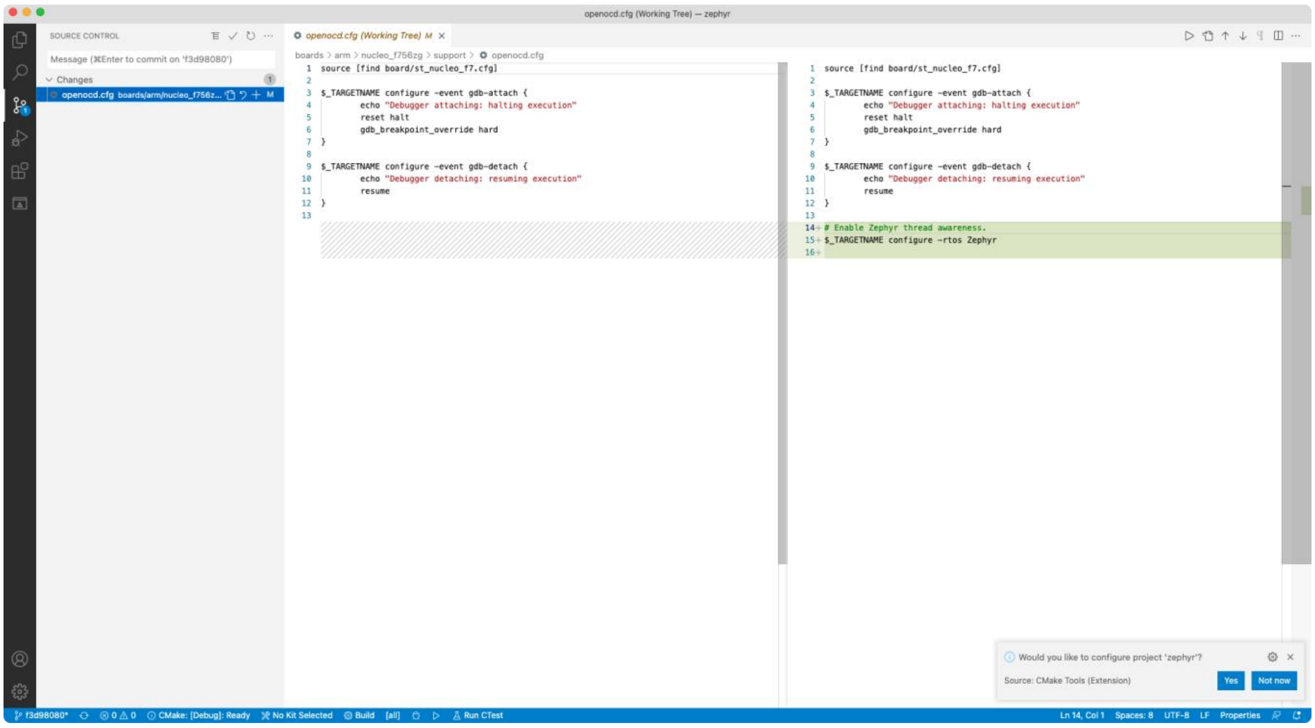
What are some common debugging techniques and when to use them?

The three most common debugging techniques we'll be discussing here are:

- thread aware debugging (debug probe)
- thread analysis
- core dump

How does thread aware debugging work?

To enable thread-aware debugging you'll need to add the line shown below to `zephyr/boards/arm/nucleo_f756zg/support/openocd.cfg`
Enable Zephyr thread awareness
$_TARGETNAME configure -rtos Zephyr



openocd.cfg (Working Tree) — zephyr
SOURCE CONTROL
Message (⌘Enter to commit on 'f3d98080')
Changes
openocd.cfg boards/arm/nucleo_f756z... M
openocd.cfg (Working Tree) M
boards > arm > nucleo_f756zg > support > openocd.cfg

```
source [find board/st_nucleo_f7.cfg]

$_TARGETNAME configure -event gdb-attach {
        echo "Debugger attaching: halting execution"
        reset halt
        gdb_breakpoint_override hard
}

$_TARGETNAME configure -event gdb-detach {
        echo "Debugger detaching: resuming execution"
        resume
}

```

```
source [find board/st_nucleo_f7.cfg]

$_TARGETNAME configure -event gdb-attach {
        echo "Debugger attaching: halting execution"
        reset halt
        gdb_breakpoint_override hard
}

$_TARGETNAME configure -event gdb-detach {
        echo "Debugger detaching: resuming execution"
        resume
}

# Enable Zephyr thread awareness.
$_TARGETNAME configure -rtos Zephyr

```

Would you like to configure project 'zephyr'?
Source: CMake Tools (Extension)

Yes
Not now
f3d98080* 0 0 CMake: [Debug]: Ready No Kit Selected Build [all] Run CTest
Ln 14, Col 1 Spaces: 8 UTF-8 LF Properties

A debug probe is special hardware which allows you to control execution of a Zephyr application running on a seperate board. Debug probes usually allow reading and writing registers and memory, and support breakpoint debugging of the Zephyr application on your host workstation using tools like GDB.

ST-LINK is a serial and debug adapter built into all Nucleo and Discovery boards. It provides a bridge between your computer (or other SUB host) and the embedded target processor, which can be used for debugging, flash programming, and serial communication, all over a simple USB cable.

It is compatible with the following host debug tools:

- OpenOCD Debug Host Tools
- J-Link Debug Host Tools

In this tutorial we'll be using OpenOCD.

How to use OpenOCD?

OpenOCD is available by default on ST-Link and configured as the default flash and debug tool. Flash and debug can be done as follows:

```
west build -b nucleo_f756zg
west flash
west build -b nucleo_f756zg
west debug
```

How does thread analysis work?

The thread analyzer module enables all the Zephyr options required to track the thread information, e.g. thread stack size usage and other runtime thread statistics.

The analysis is performed on demand when the application calls thread_analyzer_run() or thread_analyzer_print().

The output is put on the serial connection.

How does core dump work?

The core dump module enables dumping the CPU registers and memory content for offline debugging. This module is called when a fatal error is encountered, and the data is printed or stored to which backends are enabled. This core dump data can be fed into a custom made GDB server as a remote target for GDB. CPU registers, memory content and stack can be examined in the debugger.

This usually involves the following steps:

1. Get the core dump log from the device depending on enabled backends.

2. Convert the core dump log into a binary format that can be parsed by the GDB server. For example, scripts/coredump/coredump_serial_log_parser.py can be used to convert the serial console log into a binary file.
3. Start the custom GDB server using the script scripts/coredump/coredump_gdbserver.py with the core dump binary log file, and the Zephyr ELF file as parameters.
4. Start the debugger corresponding to the target architecture.

Runtime statistics

The analysis is performed on demand when the application calls one of the following functions:

- thread_analyzer_run
- thread_analyzer_print

OpenOCD

Thread analyzer

CONFIG	Description
THREAD_ANALYZER	enable the module
THREAD_ANALYZER_USE_PRINTK	use printk for thread statistics
THREAD_ANALYZER_USE_LOG	use the logger for thread statistics
THREAD_ANALYZER_AUTO	run the thread analyzer automatically. You do not need to add any code to the application when using this option
THREAD_ANALYZER_AUTO_INTERVAL	the time for which the module sleeps between consecutive printing of thread analysis in automatic mode

CONFIG	Description
THREAD_ANALYZER_AUTO_STACK_SIZE	the stack for thread analyzer automatic thread
THREAD_NAME	enable this option in the kernel to print the name of the thread instead of its ID
THREAD_RUNTIME_STATS	enable this option to print thread runtime data such as utilization (This options is automatically selected by THREAD_ANALYZER)

Core dump

CONFIG	Description
DEBUG_COREDUMP	enable the module
DEBUG_COREDUMP_BACKEND_LOGGING	use log module for core dump output
DEBUG_COREDUMP_BACKEND_NULL	fallback core dump backend if other backends cannot be enabled. All output is sent to null.
DEBUG_COREDUMP_MEMORY_DUMP_MIN	only dumps the stack of the exception thread, its thread struct, and some other bare minimal data to support walking the stack in debugger. Use this only if absolute minimum of data dump is desired.

Exercise 1: Multiple threads debugging

Using the code from the previous section (time-slicing), we'll try to see how debugging works in VSCode.

1. in multi-thread-debug create folder .vscode
2. in .vscode create file launch.json
3. setup your debug config as follows inside launch.json (change as applicable)

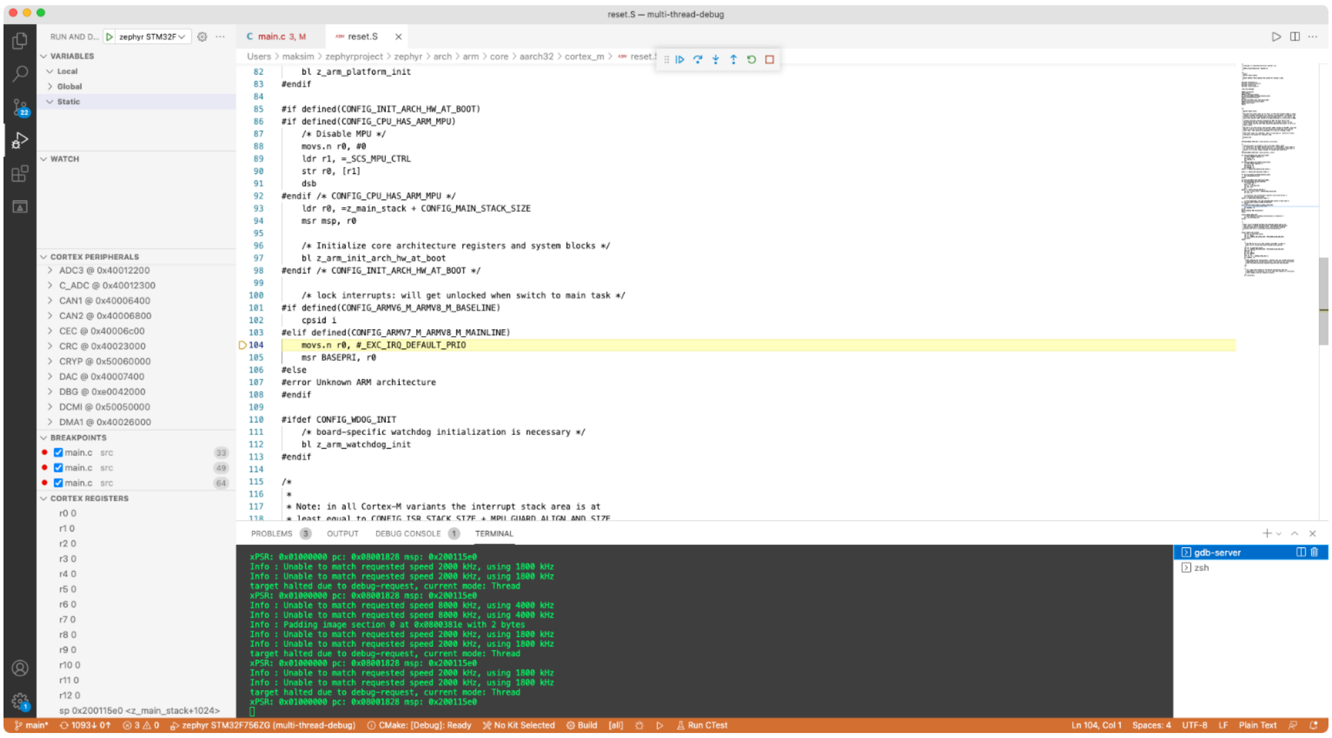
```
{
   "version": "0.2.0",
   "configurations": [
       {
           "type": "cortex-debug",
           "request": "launch",
           "servertype": "openocd",
           "cwd": "${workspaceRoot}",
           "executable": "build/zephyr/zephyr.elf",
           "name": "zephyr STM32F756ZG",
           "configFiles": [
"/Users/maksim/zephyrproject/zephyr/boards/arm/nucleo_f756zg/support/openocd.cfg"
                  ],
           "svdFile": "STM32F756.svd"
       }
   ]
}
```

The svd file can be found on the STM website. Add this file to your folder and mention it in `launch.json`, this way you'll be able to easily read out the different registers/peripherals/memory once you're in debug mode.

4. the svd file is necessary to be able to read registers from that specific stm processor

5. Now go to Run->Start debugging

The first screen should look like this:



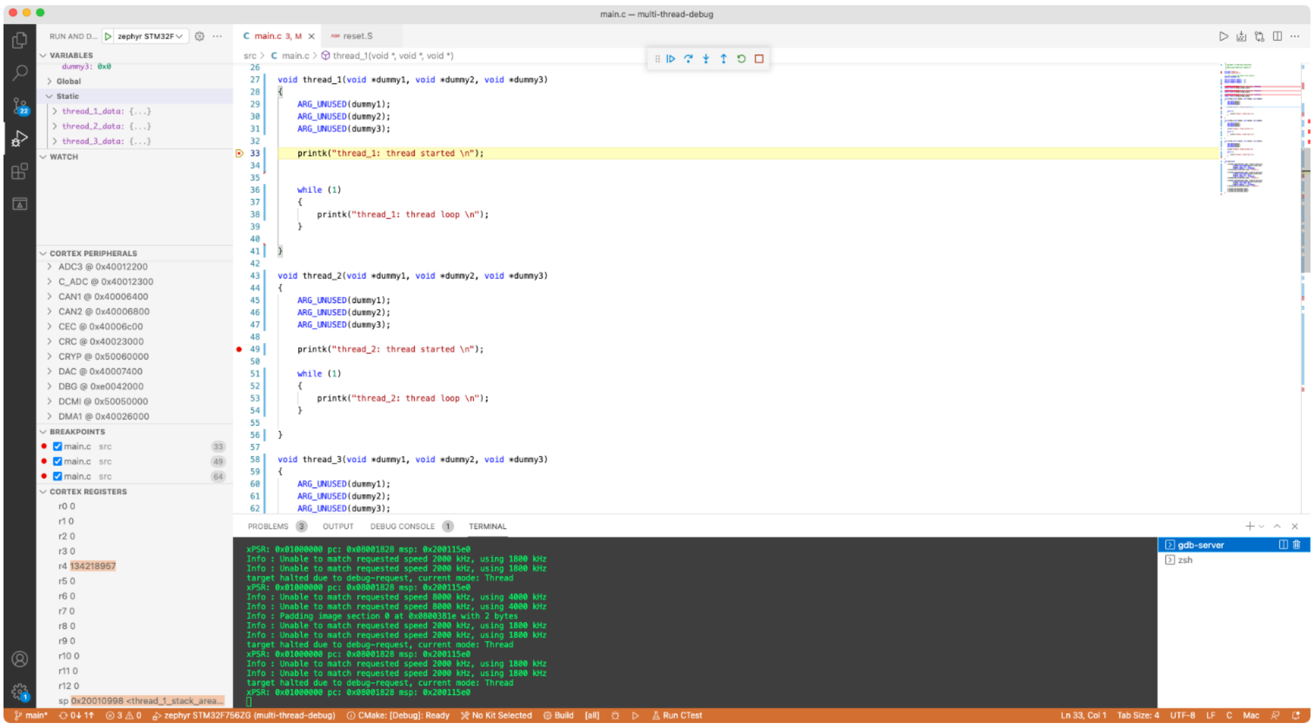
reset.S — multi-thread-debug

Users > maksim > zephyrproject > zephyr > arch > arm > core > aarch32 > cortex_m > reset.S

```
    bl z_arm_platform_init
#endif

#if defined(CONFIG_INIT_ARCH_HW_AT_BOOT)
#if defined(CONFIG_CPU_HAS_ARM_MPU)
    /* Disable MPU */
    movs.n r0, #0
    ldr r1, =_SCS_MPU_CTRL
    str r0, [r1]
    dsb
#endif /* CONFIG_CPU_HAS_ARM_MPU */
    ldr r0, =z_main_stack + CONFIG_MAIN_STACK_SIZE
    msr msp, r0

    /* Initialize core architecture registers and system blocks */
    bl z_arm_init_arch_hw_at_boot
#endif /* CONFIG_INIT_ARCH_HW_AT_BOOT */

    /* lock interrupts: will get unlocked when switch to main task */
#if defined(CONFIG_ARMV6_M_ARMV8_M_BASELINE)
    cpsid i
#elif defined(CONFIG_ARMV7_M_ARMV8_M_MAINLINE)
    movs.n r0, #_EXC_IRQ_DEFAULT_PRIO
    msr BASEPRI, r0
#else
#error Unknown ARM architecture
#endif

#ifdef CONFIG_WDOG_INIT
    /* board-specific watchdog initialization is necessary */
    bl z_arm_watchdog_init
#endif

/*
 *
 * Note: in all Cortex-M variants the interrupt stack area is at
 * least equal to CONFIG_ISR_STACK_SIZE + MPU_GUARD_ALIGN_AND_SIZE
```

```
xPSR: 0x01000000 pc: 0x08001828 msp: 0x200115e0
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08001828 msp: 0x200115e0
Info : Unable to match requested speed 8000 kHz, using 4000 kHz
Info : Unable to match requested speed 8000 kHz, using 4000 kHz
Info : Padding image section 0 at 0x0800381e with 2 bytes
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08001828 msp: 0x200115e0
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08001828 msp: 0x200115e0
```

If you put an interrupt in the first interrupt, the execution will again be stopped at this point



main.c — multi-thread-debug

src > main.c > thread_1(void *, void *, void *)

```c

void thread_1(void *dummy1, void *dummy2, void *dummy3)
{
    ARG_UNUSED(dummy1);
    ARG_UNUSED(dummy2);
    ARG_UNUSED(dummy3);

    printk("thread_1: thread started \n");


    while (1)
    {
        printk("thread_1: thread loop \n");
    }

}

void thread_2(void *dummy1, void *dummy2, void *dummy3)
{
    ARG_UNUSED(dummy1);
    ARG_UNUSED(dummy2);
    ARG_UNUSED(dummy3);

    printk("thread_2: thread started \n");

    while (1)
    {
        printk("thread_2: thread loop \n");
    }

}

void thread_3(void *dummy1, void *dummy2, void *dummy3)
{
    ARG_UNUSED(dummy1);
    ARG_UNUSED(dummy2);
    ARG_UNUSED(dummy3);
```

```
xPSR: 0x01000000 pc: 0x08001828 msp: 0x200115e0
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08001828 msp: 0x200115e0
Info : Unable to match requested speed 8000 kHz, using 4000 kHz
Info : Unable to match requested speed 8000 kHz, using 4000 kHz
Info : Padding image section 0 at 0x0800381e with 2 bytes
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08001828 msp: 0x200115e0
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08001828 msp: 0x200115e0
```

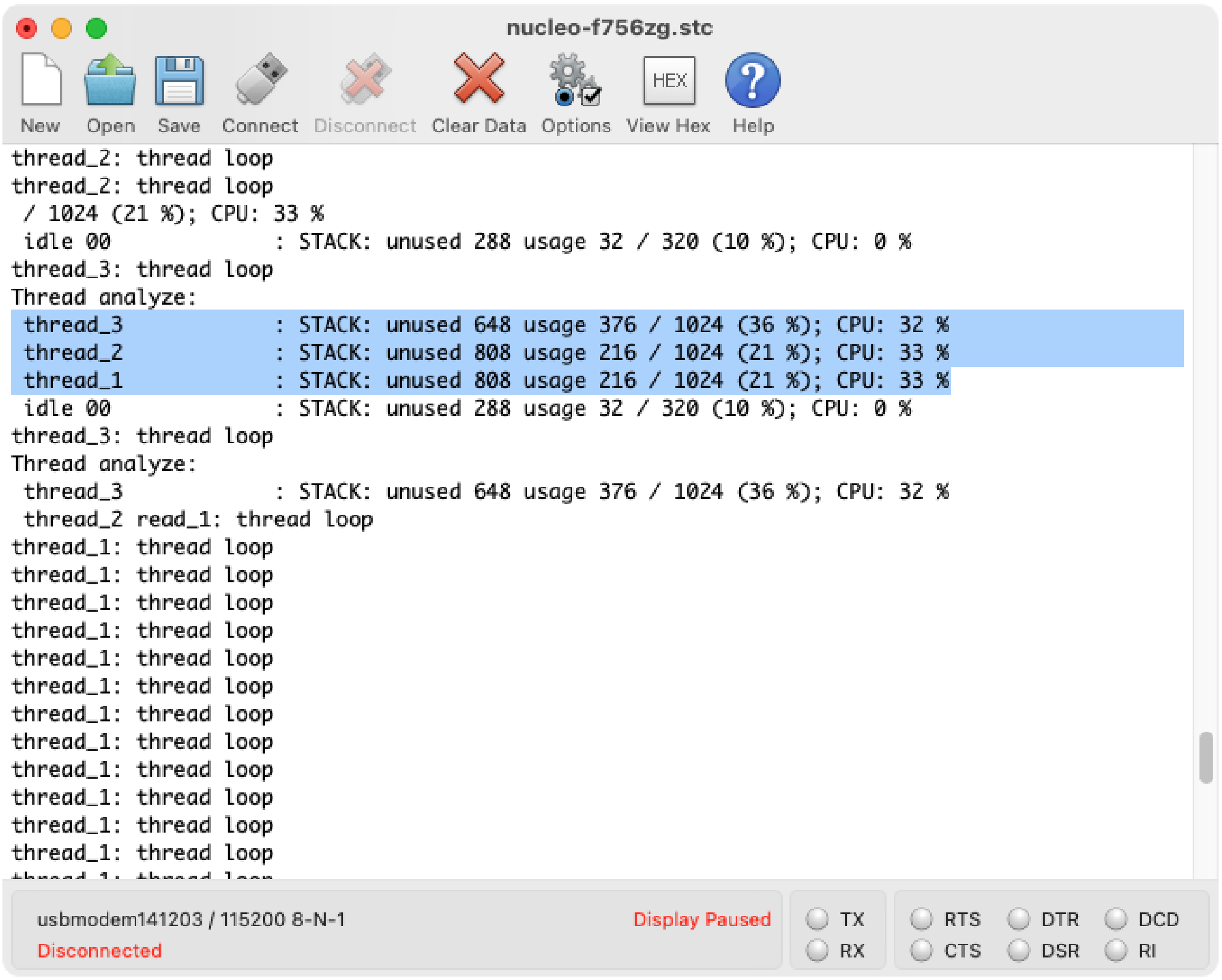
Notice the "Cortex Peripherals" and "Cortex Registers" on the left-hand side.

Exercise 2: Print runtime statistics

1. Go to `exercises/debugging/runtime-statistics`
2. Build & Flash

```
west build -b nucleo_f756zg
west flash
```

3. Observe the output in the serial monitor:



```
thread_2: thread loop
thread_2: thread loop
 / 1024 (21 %); CPU: 33 %
 idle 00            : STACK: unused 288 usage 32 / 320 (10 %); CPU: 0 %
thread_3: thread loop
Thread analyze:
 thread_3           : STACK: unused 648 usage 376 / 1024 (36 %); CPU: 32 %
 thread_2           : STACK: unused 808 usage 216 / 1024 (21 %); CPU: 33 %
 thread_1           : STACK: unused 808 usage 216 / 1024 (21 %); CPU: 33 %
 idle 00            : STACK: unused 288 usage 32 / 320 (10 %); CPU: 0 %
thread_3: thread loop
Thread analyze:
 thread_3           : STACK: unused 648 usage 376 / 1024 (36 %); CPU: 32 %
 thread_2 read_1: thread loop
thread_1: thread loop
thread_1: thread loop
thread_1: thread loop
thread_1: thread loop
thread_1: thread loop
thread_1: thread loop
thread_1: thread loop
thread_1: thread loop
thread_1: thread loop
thread_1: thread loop
thread_1: thread loop
thread_1: thread loop
```

usbmodem141203 / 115200 8-N-1
Disconnected
Display Paused
TX RX RTS CTS DTR DSR DCD RI

Exercise 3: Core dump

Come back to this when I understand GDB better

{% include incomplete.md %}