**CS466 Lab 3 -- Simple Queues, Producer/Consumer, Serial IO and gdb debugging**
Due by Midnight Sunday 2-21-2025.

**!!! Must use provided lab format!!!**

You may hand in a team lab per, Individual or Two members to a team only.

Overview:

- This lab is similar in thread structure to Lab 2, I suggest that you start with a working lab2 solution.

Lab Preparation:

- Review your Lab 2 thread structure.
- Read over the FreeRTOS API documentation for queues (http://www.freertos.org/a00018.html). Specifically look at the documentation for xQueueCreate(), xQueueSend(), and xQueueReceive() in detail. Like semaphores you have to create the queue structure before you can call the queue api operations.
- Look at the xTaskCreate api and note how priority works as well.
- Look at the use of the assert() function. Since the pico environment defines assert() so we define myAssert(). See https://www.tutorialspoint.com/c_standard_library/c_macro_assert.htm For some explanation.

Objective:

- To work a very simple Queue example.
- To add Serial debugging IO to your code.
- To run the same code in two separate tasks
- To analyze assert failures in your code.
- To connect and use using a debugger on your code

Lab Work

1. Copy your Lab02 code to this lab03. Name the main program lab3.c. You will need to modify the CmakeLists.txt to adjust the lab2 code file name to lab3. Delete all the interrupt and semaphore code that dealt with switches, keep the heartbeat task.

2. Rename your green thread to 'heartbeat' and have it run at priority 1 (One step above idle priority). This thread should always run and keep the green LED blinking at around 1 Hz if SW1 and SW2 are not being pressed.

    a) Lab-report-question-1: How much effort did it take to get this basic heartbeat-only task working?

3. □ Put a printf() in your heartbeat task that outputs the text 'tick <count> and current button state for SW1 and SW2. Verify that you can receive the text using a serial terminal of your choice.

4. □ Add the myAssert.c to you project in the CMakeLists.txt file and include the header myAssert.h in your lab3.c file.

5. □ test calling the myAssert function a while after your main() program.. Note that you have to have the printf() working if you want to see the assert output. See if you can make the assert print the programmed failure message…
   a) Lab-report-question-2: What does failing the assert do?

6. □ What does the Linux command do (assumiong you are using kermit)?

```
$ make && cp lab3.uf2 /media/miller/RPI-RP2/ && sleep 2 \
  && kermit ~/kermACM0
```

7. · Create a Queue with 20 entries before you start the scheduler.
   a) For now, we will just be passing a uint32_t through the queue. Normally you would use a structure as your queue entry element.

8. · Add a consumer thread and a producer thread. Pass the handle of the queue to each thread as part of its thread parameters, do not use global variables. (to do this you will need to cast the queue handle to a (void *) and pass as pvParameters)

9. · Make the consumer thread block on queue receive so and if SW1 is pressed, momentarily light the led whenever a message is received. Use a frequency that works visually.

10. · Make your producer thread block for a random delay then send a message, If the queue is full assert(). Try to hit about 10 messages per second for a starting average rate. If you assert you are probably holding the LED on too long and overflowing the queue.

11. · Move all of your producer characterization data (thread-name, mean-delay, priority} to a single structure definition and initialize a struct with the data in main. Instead of just passing the queue handle as you did with the consumer thread, pass a pointer to the struct to the producer.

12. · Add a second producer that will also insert messages into the queue. When the consumer thread receives a consumer2 message light the LED if SW2 is pressed momentarily. Also time the random message generation rate to average

about 10 Hz. Assert if the queue send fails. This second producer thread should use the same function as the first.

**13.** · Re-arrange the priorities so that the consumer thread has a lower priority then the two producer threads. If the program does not assert after a while, increase the period of the LED indication in the consumer to slow it down. Why do I expect an assert here?

a) Lab-report-question-3: Describe your verification for SW1 and SW2 reporting the two producer rates using only the single LED?

b) Lab-report-question-4: Describe the usefulness of using your own assert code as opposed to the default pico implementation.

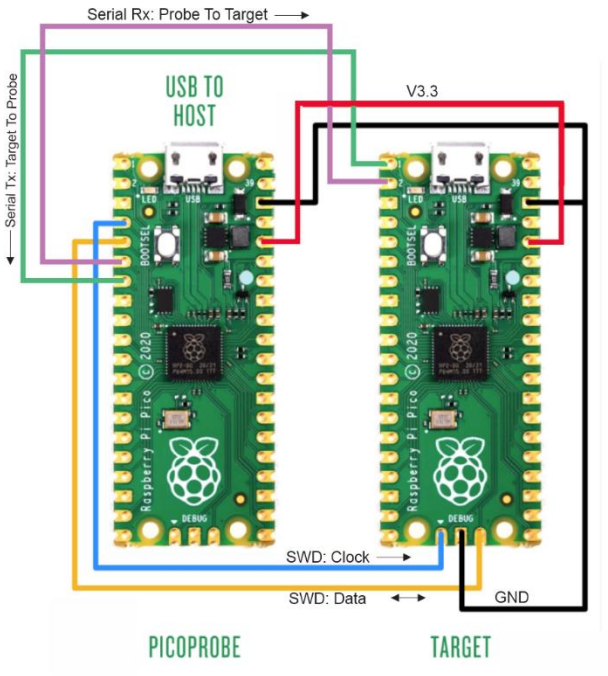
**14.** Run a few tests to make sure that the assert() function is printing what it should.

**15.** Add a counter to your heartbeat() function and add an assert(count<100);

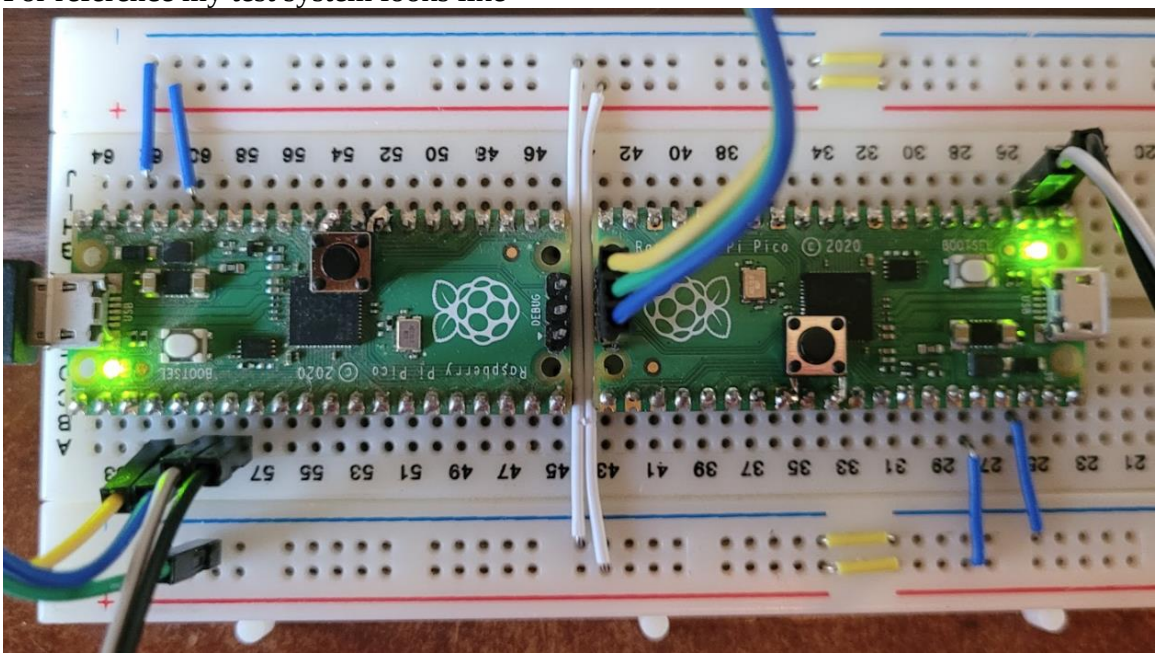
Debugging Overview:

When debugging generally you have a probe and a target. Your 'Target' is the Pico that you are programming functionality into. Your 'Probe' is another pico running a special probe image that will be used to host communications between your development workstation and the target.

We will connect the target to the probe as shown in the image:



This allows both Picos to be powered by the host serial cable but that cable is not attached to the target anymore.  The probe will stay attached to the host and powered all the time.
For reference my test system looks like



We also need to change the serial configuration that we are using and the method of flashing.
It is a little complex getting setup in the debugger but it solves a couple problems and for a longer work session much more efficient.
Pros:

+ Make/flash is a single command line, No fooling with drives coming and going
+ You can step through code and check contents of variables and memory
+ Easy to watch variables.
+ Serial will not disconnect all the tiem

Cons:

- Takes a little time to setup
- More Wires, More complexity to get going

Changes that need to be made to your working lab3:

1. Output to actual serial as opposed to using the pico virtual serial
2. Include actual serial in the cmake file

**16.** Modify your CMakeLists.txt to include the physical serial library:

```
#
# tell the pico library that you will be using usb
# serial and not an actual uart on the processor
#
pico_enable_stdio_usb(${PROJECT_NAME} 1)
pico_enable_stdio_uart(${PROJECT_NAME} 1)
```

**17.** Modify your program to include the serial library (top of file) and call the proper init code.

```
#define ACTUAL_SERIAL

#ifdef ACTUAL_SERIAL
#include "pico/stdio_uart.h"
#endif
```

```
int main()
{
#ifdef ACTUAL_SERIAL
    stdio_uart_init();
#else
    stdio_init_all();
```

**18.** Make sure that you have openocd vsrsion 0.12 or greater installed. You can get it via apt-get if you need to install it.

**19.** Copy and rename the gdbinit file from the lab3 directory to the name build/.gdbinit
look at the file and make sure that a file .gdbinit also resides in your home directory as described in the lab3/gdbinit comments in the top of the file.

**20.** Using our button-dance put the pico-Probe into reflash mode and copy the picoprobe.uf2 to it's faux drive.

**21.** Using the Debugging pdf and the new pico today in lab setup and get openocd and gdb running.
Note: that the openocd should be run alone in a new window.
Note: also, that you want to run arm-none-eabi-gdb not your hoist computers native gdb.

**22.** GDB Tasks for your report (reference the GDB Cheat sheet for help here):

a) The reset vector is stored at 0x00000004, what address contains the startup reset function?

b) What debugger command will show a stack backtrace?
c) What is a backtrace?
d) Write a macro to
    1. dump the value of the count variable in your heartbeat function that counts the number of iterations.
    2. Have the macro ‘continue’ at it’s end.
    3. Attach the macro to a breakpoint that fires when the delay function is called in Heartbeat,
    4. Add some of the output in your report

**23.** Modify your program to fail an assert().. You should have the the rapid blinky light… If you are running under the debugger you can
a) break execution (Ctrl-C)
b) ask for a stack backtrace (bt)
c) You can change the current stack context from the fail context using the (up/down) commands. This allows you to examing automatic variables from each stack frame.

**24.** Write a function that intentionally causes a processor fault. Call it 20 or so cycles through your heartb4eat code. Using the same steps as in #23 use gdb to isolate it to the failed instructions.