ECEN-361 Lab-07:Queues and IPC

# NAME: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Introduction and Objectives of the Lab

This lab will introduce the concept of interprocess communication mechanisms, particularly queues and task messages. This will be done in the framework of FreeRTOS middleware as supplied by the STM32 ecosystem.

Queues

Queues are a well-known data structure, serving a FIFO strategy to store/forward data. The student has likely programmed with queues in other classes. We take this data structure and move it into an environment where access to the queue is available multiple producers (tasks that add to the queue), and consumers (tasks that take first-off data from the queue.) Simple everyday analogous systems utilizing this type of structure can be seen in many operations: A tech-support call-center producing many incoming calls at a time, putting them in the queue to be help for service, while many operators pull the calls off to consume and process them.

In FreeRTOS, the queue structure is global in nature, and available to be written-to and read-from most any other process. In general processes will be dedicated to handing one or the other operation – either producing or consuming. Events (interrupts) can be setup to signal on various states of the queue: EMPTY, FULL, FILLED-to-a-LEVEL, etc. can a be the source of interrupts and exceptions in processing.

For this lab we will build a small queue pipeline with three producers, and 1 consumer. Elements that go into the queue pipeline from the superset of all ASCII characters, but each producer has the unique quality that they deliver only subsets of the full character set, so:

Producers

1. Random lowercase letters [a..z] One character issued per 400mS
2. Random symbols, punctuation, etc. [!..=] One character issued per 700mS
3. Typed Uppercase Letters (from keyboard – PuTTY) One character per keystroke – about 200mS

Consumer:

1. Reads the queue once every second and displays:
   1. Number of bytes left on the queue or empty “----” or FuLL”
   2. The state of the queue on the TTY Terminal (PuTTY)

Follows is a simple diagram of a queue being fed by all three sources. It’s not quite full and no priority scheme is implemented between the producing tasks.

A diagram of a computer code

Description automatically generatedColor-coding of the item shown in the queue corresponds with the source of that item.

There is also control on the production:. A dedicated button start/stops each of the following producers.

* The Random Character Producing Task Button\_1🡪 Start/Stop
* The Random lowercase number Production Task Button\_2 🡪 Start/Stop

The consumer is an outputting process who waits for data availability then sends that data to the USART2 out. (USB TTY via PuTTY).

Note that keystrokes typed into the UART keyboard (via the TTY Terminal) are ‘filtered’ before being put onto the queue. This type of pre-processing Is typical and could be likened to a real-world example of a tech-support site sending customers with different types of issues into different waiting queues.

In this code, all keystrokes typed are checked to see that they are strictly alphabetic [‘A’ … ‘z’] and then added to the queue only as uppercase. This allows us to determine the source of the item in the queue, by the type of data: uppercase, lowercase, and special-characters all can come only from their unique source producer.

# Lab Instructions

## Examine the existing code and operation

## 2 Pts.

1. How are the interrupts for the buttons used to start/stop the respective producers for the Queue?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Part 1: Understand the ADC, Read/Output the voltage

## 2 Pts.

1. How are the interrupts for the buttons used to start/stop the respective producers for the Queue?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### Accept the Assignment, Download the repo, Run it

## Extra Credit (5 pts.max for any of these completed)

1. Currently, the data is taken out of the queue only once every “**read\_pacing\_delay**” milliSeconds. Add an option to change that queue read speed by way of user input. If you add little decoding in the routine where a typed character is received, you could speed-up or slow-down the read pacing. (That routine is: **HAL\_UART\_RxCpltCallback()** and already converts any character to upper case. For example, you could make the “+” key make it go faster by lowering the pacing by 100mS.

Write about what you did – and what keys do what now:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Currently Switch S3 just takes out a single byte on demand. Change the switch so that it stops further input and dumps the entire queue.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Add another producer that adds other “extended characters into the queue, randomly chosen, like the **Add\_Random\_Symbols\_to\_Queue()**. For this producer, output characters that are in the other half of the ASCII table – 0x80 … 0xFF ;

Paste in a small screenshot of what these characters look like on your TTY. You may want to change character set in PuTTY to be Translation/ISO-8859

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. You’ll notice that the ‘random’ sequences always start the same after every reset. Look into using the **srand(unsigned seed)** routine to give a more random seed so that your routine is different on every start up. Maybe use a counter or a button press?

Ideas? Get something that works?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_