**Communication Challenge 2: Free RTOS**



A screenshot of a computer

Description automatically generated

*Circuit used: red wires are to simulate power to the component that is connected to its designated pin. The black wires are to simulate being connected to the GND pin. Each LED represents a sprinkler, and they are connected to my ESP32 as show in the above circuit.*

*By: Johnson Domacasse (#4471709)*

*13 Oktober 2023*

# Abstract:

Real-time operating systems uses threads to implement a program that looks like it is running in “real time”. Our normal understanding of code is to use a super loop to run all our functions. These are run after each other. Using threads, we can run all these functions at the same time instead of a super loop. Threads are not completely safe and can be prone to errors or data loss. Some fixes for this include the usage of mutex and/or queues in these threads.

Contents

[Abstract: 1](#_Toc148130392)

[Introduction: 3](#_Toc148130393)

[Procedure: 3](#_Toc148130394)

[Threads: 3](#_Toc148130395)

[Thread implementation: 4](#_Toc148130396)

[Mutex: 4](#_Toc148130397)

[Mutex implementation: 4](#_Toc148130398)

[Queue: 6](#_Toc148130399)

[Queue implementation: 6](#_Toc148130400)

[Results: 6](#_Toc148130401)

[Conclusions: 7](#_Toc148130402)

[References: 7](#_Toc148130403)

# Introduction:

This report was written so we can find out more about RTOS. For the implementations below we will be using Free RTOS. As we go deeper into our implementation, we can discover the differences when we work with threads, when we add mutex into the implementation, and0 when we add queues into the implementation.

In short, the usage of threads is when you want a program with different functions that doesn’t run using a super loop but simultaneously with eachother. This doesn’t completely make it safe. For extra safety we can add mutexes and/or queues to make the program more thread safe and less prone to issues with synchronization.

# Procedure:

I begin by setting up my circuit for this entire challenge. See the above circuit diagram on the front page. See table 1 for pin numbers.

|  |  |
| --- | --- |
| **COMPONENT** | **PIN** |
| LED\_1\_PIN (Sprinkler 1) | D14 |
| LED\_2\_PIN (Sprinkler 2) | D26 |
| LED\_3\_PIN (Sprinkler 3) | D21 |
| LED\_4\_PIN (Sprinkler 4) | D22 |

*Table 1: Hardware Pins*

## Threads:

Based off the requirements that were given, I made the following decisions:

* Each sprinkler own designated LED and a function to control turning the LEDs on and off.
* Store the operations of scenes. They are hardcoded and stored in my application.
* Make a function to execute a scene, by reading these operations of the hardcoded stored scenes and perform said operations.
* This function is first tested without threads.
* Make a thread that only reads the UART repeatedly. If it receives a value (in this case, an integer representing a scene), it echos this number back to my program. This thread should be never ending.
* In that same thread, when a valid number of a Scene is received from the user, it will start another thread that performs the operations of that Scene by calling the function to execute a scene that was defined previously.
* Make sure that that the thread ends itself in a controlled way when the Scene is done.
* Test the program with multiple Scenes running at the same time. For example: first type ‘1’ and a second later type ‘2’. Scenes 1 and 2 should then run together, which can be checked by looking at the LED’s.

### Thread implementation:

Now that the requirements are known, I began to work. I first defined the LED pins to simulate the sprinklers and then made the predefined scenes. These are hardcoded arrays with messages/operations to simulate its behaviour. I chose to do it like this because this is one of the methods I am most familiar with for storing data. I have made a simple function to control the state of the LED by providing it with the LED pin and the state I want the LED to be in. Another function was defined to execute the scenes by providing it with the array and the size of the array. It would then perform the operations of the given array. Finally in the super loop I call that function 3 times, to test all 3 scenes. The only issue I see with this implementation is that I need to implement delays in the loop for it to have my desired behaviour.

Now I will work on the thread part of the implementation of the program.

It was tested in a way so that a user can input values. I tested a case where an invalid scene is provided, an invalid character is provided and the correct one. Some special problems that were found were that the LEDs do turn on per scene, but they do not continue with the array. For example, if 1 was provided, the execute scene task would call the execute scene function and the execute scene function would receive array 1 and its size. The special problem that was encountered was that when the same scene was called twice, it would not execute the scene twice but once. I assume this is due to the lack of mutex usage. More will be researched in the mutex assignment.

## Mutex:

Based off the previous thread requirements and the ones that were now provided, I made the following decisions:

* Make the program in a way that the scenes do not interfere with eachother if both scenes are using the same sprinkler. This needs to be implemented by adding one mutex per sprinkler. (Acquire the mutex before the start and release mutex at the end of operation.)
* Make sure a scene doesn’t execute twice. If I use 2 scenes, then they will run in parallel because each scene has its own thread. If I use a single scene twice then the second scene must wait for the first to finish before executing. Each scene must have 1 mutex. Before a thread executes the first step of a scene, it must acquire that scenes mutex and after the last step it will release it.

### Mutex implementation:

I took the previous thread implementation and from there, implementing the mutex was simple. Based on the design choices above, I have made one mutex for each LED. This will ensure that if 2 scenes use the same LED there will be no issues for the LED being called twice. It will need to take the mutex first, which it can’t since the first one has the mutex of that LED. I have done this for every LED. Now there is no more unpredictable behaviour with the LEDs when I have two scenes running with the same LED. See figure 1 to have a better understanding of how the problem is solved.

A screenshot of a computer

Description automatically generated

*Figure 1: LED problem solved using mutex.*

Now I needed to make sure that when I call a scene twice, that it will run twice sequentially and not simultaneously. Initially, it wouldn’t run twice. The second one would be ignored. I thought this was the case but further investigation made me realize it is still being called, but they run simultaneously. This was a problem that I solved with the mutex for each scene. So, in the scene execution thread, I included two lines to take and give the mutex of that specific scene. I take the mutex of the scene, then call the function to execute the scene, then I give the mutex back before deleting the thread. In the end, if a scene is being called twice, it will now run the first time normally. When it finishes, it will run the second time. See figure 2 to have a better understanding of how the problem is solved.

A screenshot of a screen

Description automatically generated

*Figure 2: scene problem solved using mutex.*

## Queue:

Based off the previous thread and mutex requirements and the ones that were provided, I made the following decisions:

* When a new thread is created to execute a Scene, the steps of that Scene should be sent to that thread through a Queue (so the thread will not access the Scene-array directly anymore but get the steps from the Queue).
* Test if the program is now completely thread safe. In other words, works correct all the time.

### Queue implementation:

Simply put, I would need to repeat the same logic I did to fix the issues of the thread program with a queue instead of a mutex. I took the same code I had in the thread program and started implementing based on the choices above.

First I made changes to my UART reader thread. I define the steps (for the queue) of each scene, depending on which scene I am using. For example, if I use scene 1, I will assign the operations of scene 1 as steps. I then send these steps to the queue. Finally, I create my execute scene thread.

Within the execute scene thread, I will “receive” the steps that were sent to me in the UART reader thread. Finally, I pass these steps into the execute scene function instead of the entire array itself. I still use a mutex to make sure that if a scene is executed twice, it won’t be simultaneous. With this implementation, I am sending the operations of a scene through queue steps instead of reading from the array itself.

As for an answer to the question if my program is now completely thread safe, I have concluded the following: Since I am using 1 queue per scene, it makes the chances less of the data being manipulated by multiple scenes. Further use of the mutex makes it so that the operations (LED toggles in this case) make the data even less prone to being manipulated. My final answer is no, the program is not completely thread safe, because right now I am only focusing on the synchronization aspect of RTOS. If I go a bit deeper, I think I will find more information of how my program can be even more thread safe. For now, this is the safest it will get.

# Results:

A problem I encountered would be that in the beginning I couldn’t iterate through my scenes due to me assigning the wrong values to my added pointer variables. Once I figured this out It was easy to iterate through the array. So, for future uses, I wouldn’t directly assign the array value to the new pointer value since it only assigns the first element in the array.

Other problems include not deleting a task at the right time or not deleting them at all. This would call the while true loop to run indefinitely.

My test cases were quite simple. I had a case where I would execute one scene and see if it would run correctly. I would execute two scenes who use different LEDs at the same time. I would execute two scenes with the same LEDs at the same time. I would execute all 3 scenes at the same time. I executed a scene twice to see the outcome. Finally, I execute a scene twice and an additional scene as well.

I execute these test cases for all 3 implementations. Naturally in the first you will find problems that the second and third implementations fix.

All these test cases had the correct results I was expecting so I thought my implementations were a success.

# Conclusions:

What I learned from this assignment is how threading works, how I can apply both mutex and queues to make them more efficient, and able to run the program simultaneously or sequentially. I learned that threads are a good way to make a program execute multiple tasks at the same time (outside of a super loop), but I should be careful implementing it.

Whether I would use threads with or without a queue or a mutex highly depends on the scale of my program and what I want the program to accomplish. If I am using a program that doesn’t involve the threads using shared data, I would simply just use threads without mutexes. If I am working with shared data, for example an LED in this case, I would need to apply mutexes or queues, so this data doesn’t manipulated.

You could say the usage of a mutex and queue are the same, but they are not. With mutexes you are simply giving a thread access to a shared variable, while other threads cannot access this shared variable yet. With queues, the threads assignments are put in a queue in the form of “steps”. When running the program, it will follow the steps in the queue itself without having to worry about priority.

My final thoughts would be, if possible, to just implement queues all the time, if memory is not an issue. It is a few additional lines of code that will safeguard your usage of threads. If memory is the issue but you need to implement something robust with minimal failure, then mutex would be your best choice.

## References:

[1] – Free RTOS kernel section - <https://www.freertos.org>

[2] – variable manipulation (UART thread) - <https://sentry.io/answers/char-to-int-in-c-and-cpp/#:~:text=C%20and%20C%2B%2B%20store%20characters,the%20value%20of%20%270%27%20>.

[3] – understand mutex and queue difference - <https://www.geeksforgeeks.org/implement-thread-safe-queue-in-c/>

[4] – understanding Free RTOS - <https://www.youtube.com/@simplyexplained>

[5] – additional information of Free RTOS - <https://www.youtube.com/@digikey>

[6] – information on queue and code implementation - <https://www.youtube.com/watch?v=pHJ3lxOoWeI>