This report examines the RTOS architecture and usage of Global Variables for the robotic car project.

1. RTOS Architecture

A total of five threads consisting of four tasks and one interrupt, were created to execute the respective functions. The flow between these threads can be seen in the figure below.

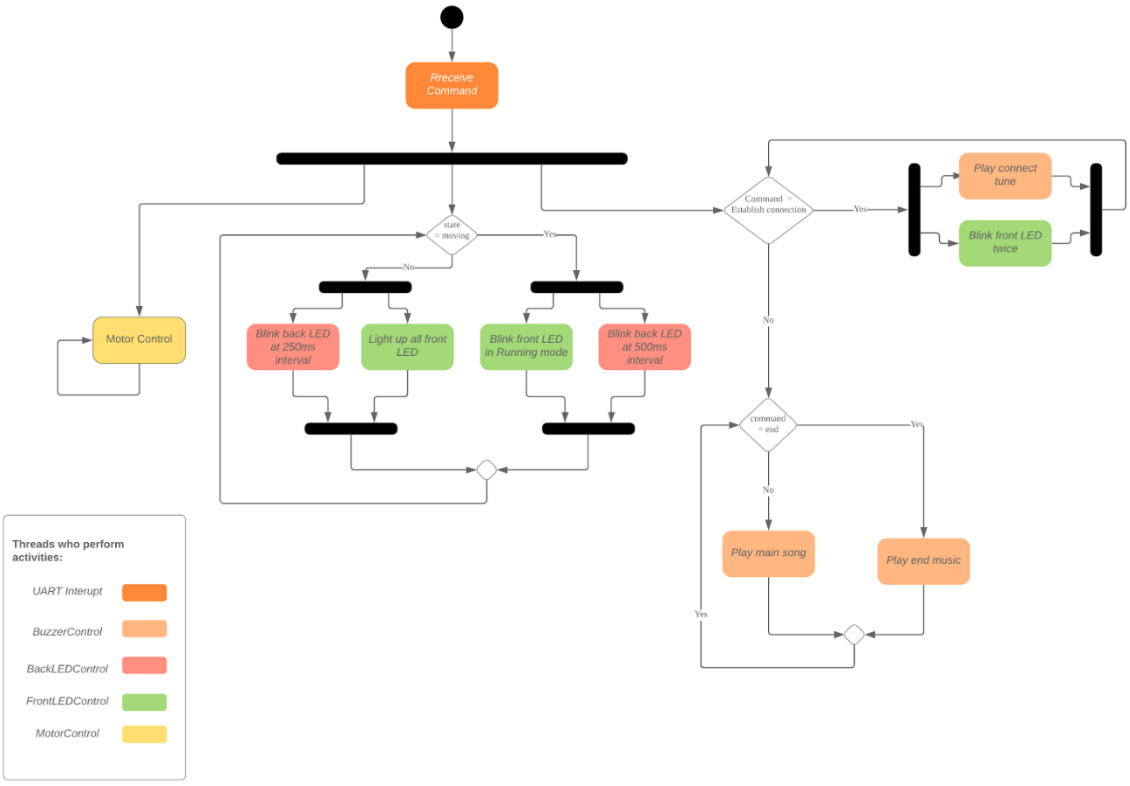

Figure 1: Overall activity diagram for the RTOS-based robotic car

1.1. Global variables

The two global variables that were created and utilized, are as below:

`volatile uint8_t data;`

- Variable to contain the data received from the Bluetooth module
- Needs to be `volatile` for other threads to get the updated command from the app and do the right action accordingly

`volatile int state;`

- Variable to indicate the current state of the bot
- `state` = 0 when the bot is stationary & = 1 when the bot is moving
- Since `state` is being checked in multiple threads, it should be `volatile` to ensure that the data is updated whenever it is changed.

1.2. Threads/Tasks

The four tasks implemented in the project and their corresponding priority levels are as follows:

a) `MotorControl (High Priority)`
- Performs the necessary action with respect to the data received which includes changing the PWM signal accordingly to enable the robotic car to move in different directions and causing the front LED to blink twice upon the establishment of a Bluetooth connection
- Highest priority given to ensure that the bot response to the command is almost real time

b) `FrontLEDControl (Low1)`
- Controls the front green LEDs according to the state of the bot as highlighted below:
  - `state` = 0 (bot is stationary): Front LEDs are lighted up continuously
  - `state` = 1 (bot is moving): Front LEDs are in a Running Mode (1 LED at a time)

c) `BackLEDControl (Low)`
- Controls the rear red LEDs according to the state of the bot as highlighted below:
  - `state` = 0: Rear LEDs flash continuously at a rate of 250ms ON, 250ms OFF
  - `state` = 1: Rear LEDs flash continuously at a rate of 500ms ON, 500ms OFF

d) `BuzzerControl (Normal)`
- Plays notes of the respective music tune by changing the PWM signal accordingly
- Second highest in priority to ensure that music played is at the right pitch on the correct tempo

The timing diagram of these threads is illustrated in the below diagram:

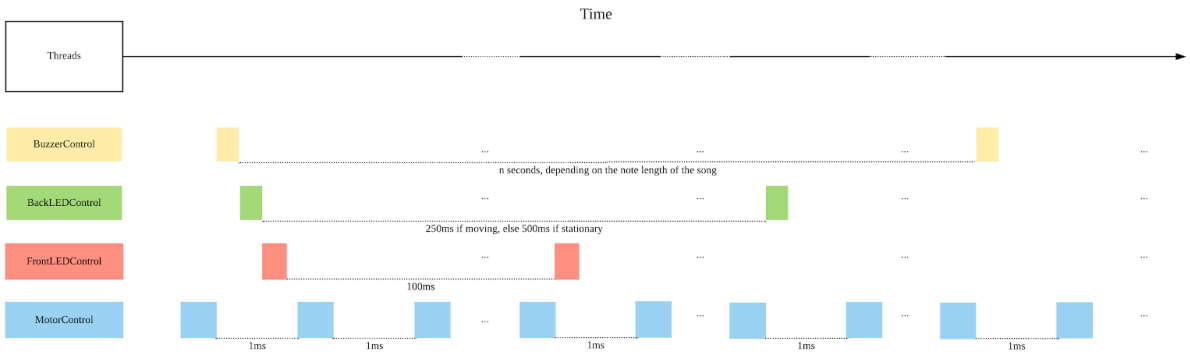

Figure 2: Timing diagram of threads

As seen in figure 2, the `MotorControl` thread runs every millisecond. Meanwhile, the `BackLEDControl` thread would toggle the LED every 250ms if the bot is stationary, or 500ms if the bot is moving while the `FrontLEDControl` thread "scrolls" the LED every 100ms if the bot is moving, or lights all of the LEDs up otherwise. Lastly, the `BuzzerControl` thread would change the PWM frequency of the buzzer with a variable period, depending on the length of the note.

1.3. Usage of Mutex

As explained in 1.2, both `MotorControl` and `FrontLEDControl` threads make use of the shared resource of the front green LEDs. `MotorControl` have to ensure that the green LEDs blink twice upon the establishment of a Bluetooth signal while `FrontLEDControl` have to ensure that the green LEDs are either lighted up or in a running order depending on whether the bot is moving. In order to ensure they don't conflict, a mutex, `ledMutex` was created. This way, when one task is in possession of the mutex, the other would be unable to acquire it until it gets released from the first task.

1.4. Usage of Interrupt

`UART2_IRQHandler` acts as an UART2 interrupt on receive and reads from the UART register into the global variable `data` and sets the `state` of the bot either 1 or 0.

2. Detailed Explanation of Code

2.1. Initialization

On startup, we initialize our PWM, LED and UART through `initPWM()`, `initLED()` and `initUART2()`, and then initialize the kernel, create the threads as stated in section 1.2., and start the kernel.

The first thread to run is `MotorControl` since it has the highest priority. As no data packet has been received yet, it does nothing and gives up the CPU for 1ms using `osDelay(1)`. This causes the next highest priority thread, `BuzzerControl`, to run which starts to play the respective notes of the main song tune. The call of osDelay() between successive notes enables `FrontLEDControl` to run which turns on all the green LEDs since the global variable `state` is 0 as the bot is stationary. It then gives up CPU for 100ms and finally allows `BackLEDControl` to run which toggles the back red LEDs and gives up the CPU for 250ms.

2.2. Upon Establishment of a Bluetooth Connection:

On connect, the mobile app would send the data packet, `0x08`, to the Bluetooth module. The ISR, `UART2_IRQHandler`, would hence get triggered and read the data packet into the global variable `data`.

When `MotorControl` runs, upon checking that the data packet received is `0x08`, it would try to acquire the `ledMutex` to ensure that the front LEDs blink by first turning the LEDs OFF. After which, it would sleep for 1s by calling `osDelay(1)`.

Next, `BuzzerControl` runs and upon checking that the data received is `0x08`, it would play the connect tone, giving up the CPU after each note for the duration of the note, and setting the frequency of the note by setting the PWM frequency. After the end of the connect tone, it would once again start to play the main tune.

Meanwhile, after 1s, when motor control is ready to run again, it would turn the LED on, give up the CPU again for 1s. This toggling cycle would repeat for another time before changing the global variable data to `0x09` and releasing `ledMutex`.

During this process, when the code reaches `FrontLEDControl`, it would try to switch the LEDs on. However, it would be unable to do so as it is unable to acquire `ledMutex`, and it would enter a blocked state. It would only run and turn the LEDs ON after `ledMutex` has been released. Meanwhile, `BackLEDControl` would continue to ensure that the red LEDs blink at the correct rate.

2.3. On Data Packet:

The app sends a data packet when a respective button corresponding to an action that needs to be done, is pressed. The table below shows the corresponding packet that is sent.

Data packet	App action
0x01	Play end music
0x02	Forward
0x03	Backward
0x04	Rotate Left
0x05	Rotate right
0x06	Curve Leftwards
0x07	Curve Rightwards
0x08	Signal to indicate connection has been established
0x09	Stop Moving
0x10 - 0x6E	Sets the power level of the motor
0xCD - 0xE6	Sets the curvature radius of rotation

As for direction commands, the app would send a single packet corresponding to the respective direction that we want the bot to move, when we press and continue to hold the respective button. Once the button is lifted up, the app would send a single stop packet. As such, the data `0x09` is not represented by any buttons and would be sent automatically whenever the buttons corresponding to the directions are released.

When the '`Set Power`' button in the app is pressed, an integer value of 10 to 110 is sent, which corresponds to the speed at which we want the bot to move. This value is the nearest integer percentage of the speed slider bar + 10. The larger the integer value, the faster the bot would move. Likewise, an integer value between 205 - 230 would be sent if the curvature slider bar had been modified which corresponds to the angle at which we want the bot to curl. Similar to the speed cursor bar, a larger value represents a wider curvature angle and vice versa.

When UART2 interrupt is triggered, the data packet is read from the UART register and stored in the global variable `data`. It also sets the global variable `state` to 1 if the data packet corresponds to the bot moving (which is anything between `0x02` and `0x07`).

When `MotorControl` runs, it would take action based on the `data`, and if it corresponds to a direction command, it sets the PWM duty cycle of the motor respectively, multiplied by the power and curvature that we have set accordingly. If `data` is between 10 - 100, we know that it is a power packet, and we calculate and set the desired `power` variable in the task. When the task runs the next round 1ms later, the PWM duty rate would be then updated. Similarly, if `data` is above 200, we know that it is a curvature value, and set it accordingly, which would update the PWM duty rate in the next iteration. At the end of each loop, the CPU is given up for 1ms in order to let other threads to run.

Thus, this allows for the `BuzzerControl` to run when it needs to change the notes of the song accordingly. When `FrontLEDControl` and `BackLEDControl` run, they would check the global variable `state` and flash the LEDs accordingly as stated in 1.2.

2.4. Upon Reaching the End of the Maze

Upon finishing the maze, the `Stop` button in the app (which is different from the data packet `0x09` that corresponds to 'Stop Moving') needs to be pressed to indicate that the bot has finished the maze. This would send a data packet of `0x01`. When the data being read is `0x01`, the `BuzzerControl` thread will set its local variable, `end`, to 1. Thus, on all subsequent runs then after, upon checking `end` = 1, `BuzzerControl` would play the ending song tune instead of the earlier music tune.