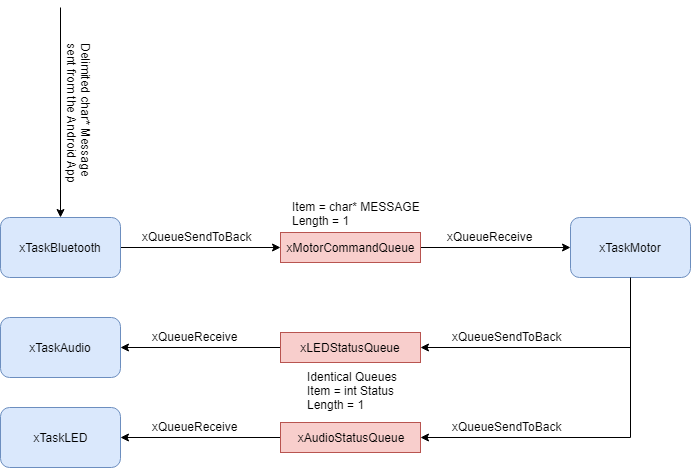
# **RTOS Architecture Report**

## CG2271 | AY2018-19 Semester 2 Saif Uddin Mahmud [A0170896N] & Sean Tan [A0164209H]

The human-controlled, teleoperated robot (endearingly called TeleChicken) was made using **FreeRTOS** as the Real Time Operating System, used with the **default configuration of the Atmega328P port,** and the project was developed in **Eclipse IDE (Mars)**. The architecture is simple-yet-efficient and comprises of **4 tasks**, one each for Bluetooth, Audio, LEDs and Motors. **There were no, non-RTOS global variables used to share information across tasks,** keeping with the spirit of RTOS systems. The tasks use **Rate-Monotonic-Scheduling** (RMS), with the priority in descending order as follows: **xTaskBluetooth (Highest) > xTaskMotor > xTaskAudio > xTaskLED (lowest).**

**Each task has lightweight supplementary functions for interacting with the underlying hardware**, stored in separate cpp files for modularity. You can see a high-level overview of the architecture in the diagram below:



Descriptions of the internal working of the tasks are as follows. This section is meant to document the thought process behind design decisions.

**xTaskBluetooth:** The robot has a Bluetooth Module (HC-06) hooked up to it to receive commands from the Android Application. The Bluetooth module is configured with the default baud rate of 9600 (~ 1 char/second) as we found higher baud rates susceptible to corruption and hence unreliable. The command is 6 chars long and follows a predetermined protocol (Appendix A). Our team decided to go with **heartbeat polling (once every 10 ms) of the RX buffer instead of USART Interrupt**, due to bugs outside the scope of the module (Appendix B). Once the RX buffer has a command, xTaskBluetooth carries out rudimentary error checks and parsing and sends the command to xMotorCommandQueue. Since the app is programmed to sends commands at a rate of 1 command per 20 ms, this ensures no packet loss. **This task is currently the bottleneck of how snappy the robot can be**, but a response time of around 20 ms seems to be performing very well, given that the human response time is around 300 ms.

**xTaskMotor:** xTaskMotor consumes the messages from xMotorCommandQueue and controls the motors appropriately. xMotorCommandQueue has a length of 1 only, since there should never be 1 unhandled message while another command comes in (20 ms apart at least). This is thanks to the very small WERT of all the tasks in the system. The 2 left motors are shorted and receive the same PWM input, and the 2 right motors work the same way. This theoretically means the robot can turn at any angle, barring limitation of the motors used. Once this important task is done, the task sends only the status of the robot (moving, stationary etc) to xLEDStatusQueue and xTaskAudioQueue, both of length 1 as well for the aforementioned reason. Note that both the message queues get the same status.

The task also has a safety feature. The app is supposed to send STOP packets every 20 ms if we’re not fiddling with the controls (joystick like app interface with two sticks). Since we do not specify any distance from the app, the motors would run forever. If the connection was lost after a move command (as might happen from interference in the Bluetooth spectrum), the car would keep moving and eventually crash. The task guards against this, by asking the motors to stop as if we have not received any command in the previous 200 ms.

**The motors themselves are driven by interrupt based pwm using both channels of timer2, which means that this task merely has to set the Duty cycle.**

**xTaskAudio:** xTaskAudio calls the appropriate function to output a square wave of 50% duty cycle at a specific frequency on the pin connected to the buzzer. This square wave is generated by timer0, which is directly connected to pin6 on the Arduino. Since the generation of the square wave is handled by a hardware timer, this task is only required to set the frequency at which the hardware timer will output the square wave. The task has a period of 100ms, and will loop through a predefined set of frequencies in order to play a song regardless of status received from the Message Queue. However, should the status received in the message queue not correspond the current song, the task will switch and begin to loop through another set of frequencies, effectively changing the song it is playing.

**xTaskLED:** xTaskLED calls appropriate functions to control the two sets of LEDs – 8 Red LEDs addressed in the same way (consumes 1 hardware pin) and 8 individually addressed Green LEDs (consumes 3 hardware pins by using a serial-parallel shift register). Functions in ledDriver.cpp are called appropriately depending on the status received from the Message Queue. Note that the task is non-blocking regardless of whether the Queue is empty or not.

**Conclusion**

As all projects go, there are areas of improvement. We would have liked to hunt down the UART bug and do things the right way with a slightly enhanced architecture. We would have also liked to replace some of the parts for better quality and added sensors (LIDAR, Ultrasound, Camera, BLE/Wifi) and actuators (Stepper motors) for a higher performing, more impressive robot. That would require a much more complex architecture with more tasks, and would require more time, better planning and timing analysis. We would also fortify the app with Cybersecurity in mind, since the current system is prone to packet sniffing (no encoding of data) and Man-In-The-Middle attack (no authentication system, except the initial pairing). We would also make the code more resistant to errors, and the foundation of that is laid down in the code.

The hands-on mini project was a fun experience; it enhanced our understanding of RTOS structures as well as made us interested in leveraging different RTOS-es in our other embedded systems projects.

**Appendix A: Command Protocol**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| leftforward | leftbackward | rightfoward | rightbackward | status | delimiter |

**Figure:** Command Protocol

The app can send either leftforward or leftbackward, and the same is true for rightforward and rightbackward. These 4 are of 8 bit unsigned integers (0 through 255) and map to the PWM of the Arduino. Status can be MOVING, STOP, INIT and WON (and are defined as chars in the project). The delimiter is the dot(.) character and is necessary since HC-06 doesn’t use \r or \n and \0 can’t be sent over this serial (Arduino thinks the buffer is empty).

**Appendix B: UART Interrupt Bug**

Our initial design choice was to use the USART\_RX\_Interrupt so as to keep the CPU free and the architecture even more expandable. This would work by having the ISR release a BinarySemaphore for the xTaskBluetooth to start receiving. However, after a considerable amount of trial and error, we understood that FreeRTOS is somehow forcing both Eclipse and the Arduino IDE to import Serial’s definition of the ISR, hence barring us from declaring it even when we’re trying to configure it bare-metal without importing Serial.h.   
  
If you’re adamant on using interrupts and the newly mentioned architecture, you can perhaps solder a connector on the RX LED of the Arduino and connect that to INT0 (pin 2) and use the hardware interrupt on the rising edge.

**Appendix C: Further Reading**

The code used in the project can be found here: <https://github.com/Psyf/CG2271_AY1819S2_TeamSaen>  
You can find the video we made for the project here: <https://youtu.be/wJaD3GtW3Lc>   
A more extensive writeup of the project, helping people rebuild this and learn some RTOS fundamentals along the way, will be coming out here: <https://medium.com/dabbler-in-de-stress>