# CMPE 663 Project 2

## Servos

Andrei Tumbar

Instructor: Wolfe
TA: Nitin Borhade

By submitting this report, you attest that you neither have given nor have received any assistance (including writing, collecting data, plotting figures, tables or graphs, or using previous student reports as a reference), and you further acknowledge that giving or receiving such assistance will result in a failing grade for this course.

Your Signature: _____

## Analysis/Design

This project looked at implementing a motor controller that could take user input as well as run a predefined set of instructions. Instructions were encoded into a single byte with the first three bits acting as an op-code and the final five bits as a a parameter. The design of the entire system was split into multiple independent modules. An executor was created for executing task modules at various rates. A table of tasks can be found in `main()` called `task_table[]`.

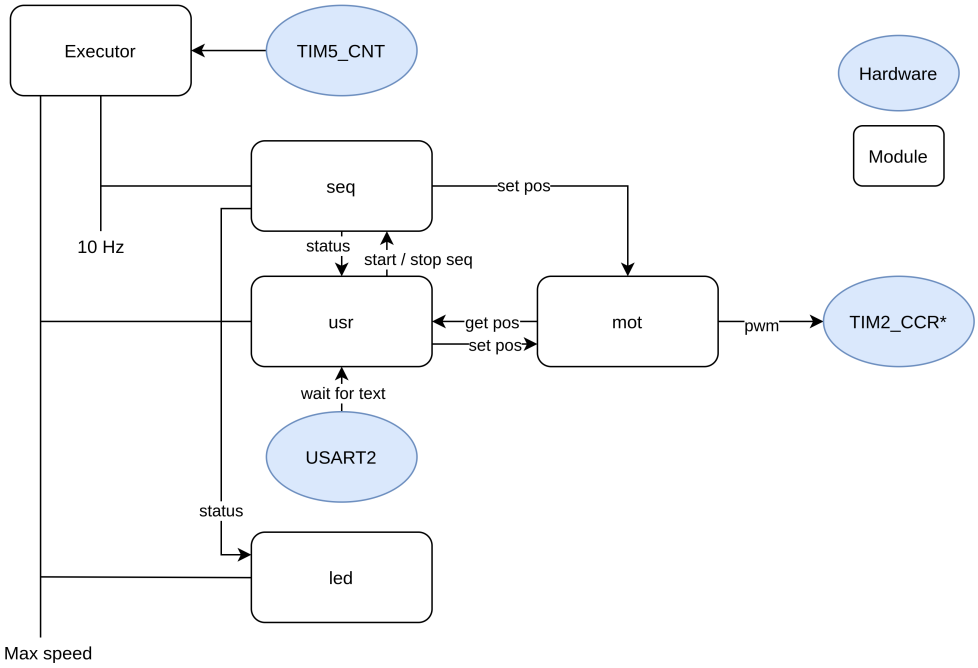


Figure 1: Design of servo control project

The first and most important module is `mot` (Motor). This module is responsible for keeping track of motor position and providing a public interface for getting and setting motor position. This module utilized a PWM conversion table to produce a correct duty-cycle given a target position.

`seq` (Sequence or recipe) is responsible for keeping state of an executing recipe on the specified motor. Two sequence engines were instantiated pointing to either servo motors. The implementation involved a program counter and instruction table to decode from. Each opcode would then dispatch a separate function and could interface with `mot` if needed. Because the `seq` module was implemented as a task (as opposed to `mot` which is a service), it has a executor interface that is expected to be called at 10 Hz.

Finally, `usr` and `led` are also implemented as tasks, however their execution are not delayed

by a hardware timer. Their non-blocking task will be executed at the maximum speed the software can attain. `led` simply looks at the status code of both sequence engines and controls four LEDs to denote each of the four states per motor. `usr` will take input from the user and control `seq` and `mot` as needed.

### Grad-portion

For the grad-portion of the assignment, an extra op-code was implemented called `ERROR_IF`. This opcode would check the position of the opposite motor and cause recipe failure if the motor was at a given position. For example, if `ERROR_IF|3` was running on motor `1` and motor `2` was at position `3`, the recipe on motor `1` would fault out. To easily test this behaviour, an extra set of user command was implemented, any ascii character from `'0'` to `'5'` could directly set the position of the motor in question.

## Test Plan

Testing the code was fairly straighforward. I started by implementing `mot` and testing each motor position on either motor. Once this was working, I wrote the `seq` state machine and made sure the timing provided by `TIM5` was accurate. The extra set of user commands discussed in the grad-portion section of this document also proved very useful for the testing of `usr`, `seq` and `mot`.

## Project Results

The software behaved as expected by the problem statement. Motor 1's recipe `MOV` commands as well as the `WAIT` commands produced the expected delays of $9.3s + 0.2 \cdot \delta_{position}$. User input could be gathered at any point during program execution including during the execution of a recipe. This was done by writing all tasks as a non-blocking interface so as to allow the execution of all tasks.

## Lessons Learned

This project explored creating an embedded system with multiple simultaneous cooperating tasks running. It also tought me how servo motors are controlled with a PWM signal. While the hardware component of this project was fairly straightforward, the software was written to be robust enough to fault execution if invalid branches or parameters were seen. This caught several bugs and sped for project up considerably.