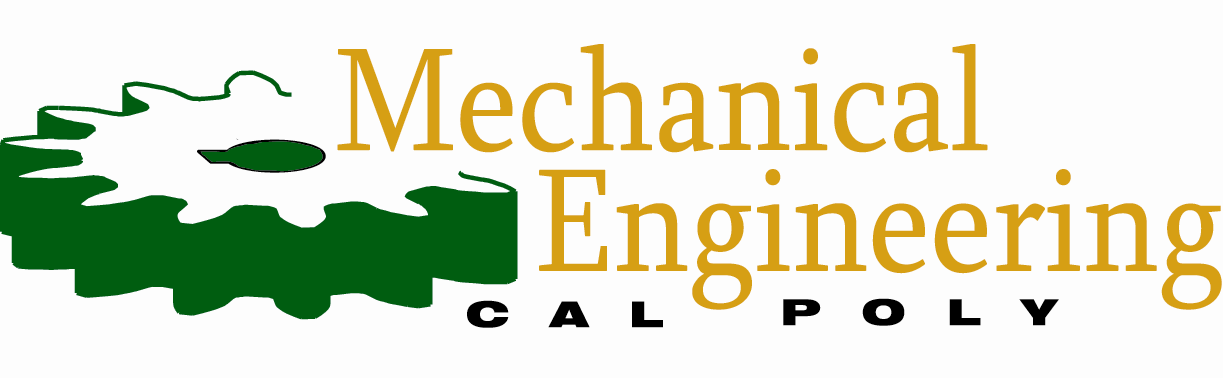
**MEMORANDUM**

|  |
| --- |
|  |

|  |  |  |
| --- | --- | --- |
| **To:** | Kennedy, Jane E. | |
|  | jekenned@calpoly.edu | |
| **From:** | Edward Ruano | Robert Trujillo |
|  | eruano@calpoly.edu | rotrujil@calpoly.edu |
|  | Drew Isaak | Anthony Lombardi |
|  | aisaak@calpoly.edu | atlombar@calpoly.edu |
| **Date:** | April 21th, 2016 | |
| **RE:** | **Lab 2: Creating Dual Tasks with Motor Drivers** | |
|  |  | |

We implemented a design similar to that requested on the lab in that each task\_motor has a corresponding motor\_driver that it controls. In order to keep the code robust and generic as mentioned in the lab, pointers to motor\_drivers are passed into the tasks that control them. These drivers are initialized in the main() loop and therefore can be given any register or pin values.

Our task\_user class received a lot of attention as it was altered to remove most of the code in the main case statement and spread it out among several helper methods. In order for the task\_user class to print out statuses about either motor at any time, local variables were created to store the values being passed into the created *TaskShare* variables.

This way we can keep the amount of *TaskShare* variables declared to a minimum and a bit more like what our project will ultimately need since we are only using one motor. We used a total of three different TaskShare variables: one to keep track of the power level, another to keep track of the directive of the motor, and the last was a selector which dictates which motor was being affected at a certain time. This way both motors could be run simultaneously since the task\_motor ignores setting the registers of the motor\_driver that is not selected.

To test the implementation, we decided that having a printDashBoard() method in task\_user would be beneficial since it would handle giving us a recap of all the important conditions related to the driver at the time of refresh. Since the user can control when the printDashBoard() method refreshes, the user can change something and quickly see if the task is reacting accordingly. During potentiometer mode, the user can print various snapshots of data to compare against other data like the actual A/D readings.

Much of the time was spent in the task\_user class. The way the case statements are set up in the run() method are set up so that they loop infinitely so instead of containing useful logic in a confusing case statement, we placed it in protected methods. This way we conserve the case states for top level operations such as the *Main Motor Control Module.* For instance, the logic given to receive inputted numbers from the user was contained in a state where the task was waiting for input. Instead of having all that logic run continuously in a state, we consolidated it into a method that could be used over and over even within the same state in order to not lose our place in the logic.