**Electronics and Control Chapter**

**Section 2 – Software Levels and Layout**

Shape, arrow

Description automatically generated After Selecting the microcontrollers to be used in the bionic arm, we aim to program them to do specific functions to control the various system motors and sensors. We had many options to program the ARM microcontrollers as they were compatible with abstract APIs like Embed. However, we chose to start from scratch on Keil with our custom-made drivers. This choice was made because we as a Low – Level Control team wanted to dive deeper into the ARM architecture and increase our exposure to embedded system coding.

The software layout was divided into three main layers. The first layer is the microcontroller layer (MCAL) that is responsible for interfacing all the necessary MCU peripherals. This layer has drivers that activate peripherals like GPIO, ADC, UART, I2C, and Timers.

The second layer is known as the Hardware Abstraction Layer (HAL layer). It is responsible for interfacing all sensors and motors connected to the system. HAL Drivers use drivers from the MCAL layer to operate various sensors like FSRs, IMU, Potentiometers, Encoders, and Temperature sensors. Motors used are both DC motors and Off the Shelf Servo Motors.

The third layer is the Application Layer. It uses the HAL layer to specifically operate the bionic arm system flowchart. All drivers are used to make functions suitable to the usage of sensors in the bionic arm sensor. Interrupts and Queues for communication are defined for operating all microcontrollers are working simultaneously to achieve the desired poses set by the high-level controller.

**ADC Driver Logic and Algorithm**

Text, letter

Description automatically generatedThe Analog to Digital Converter in Core M4 microcontrollers is a 12bit ADC with multiple features that focuses on low power consumption and reliable sensor readings. The driver aims to simplify the initialization process for the user by allowing configuration features set by the user without showing the user what is going on under the hood. An initialization function applies all the settings in the configuration file. While another function is used to get a single reading from a specified analog channel. Settings the user can configure include ADC prescaler, mode, or interrupt usage.

The analog interface functions are like those of Arduino to simplify coding for the user. Although we only utilize a single read from the analog module, there are other features that can be implemented like continuous analog conversion or multiple consecutive channel reading. Interrupt and DMA can be also used after scanning the selected channels for efficient processor usage.

Text

Description automatically generated with low confidence

**Advanced PWM Timers Driver Logic and Algorithm**

Timers are necessary in all microcontrollers because of their important roles in counting and measuring capabilities. Timers can be used in multiple modes including input capture, output compare, and PWM modes. They can be used as periodic interrupt generators or counters of external triggers too. In this driver, we aim to reduce contact of the user with configuring module registers as all is done through settings made by the user. PWM is generated by setting the duty cycle in the channel counter, while the time-period is placed in the timer counter. Other useful features like stopping and resuming the PWM signal on the pin can be made too.

Timers are also used to interface encoders when set to work in encoder mode. In this mode, two timer channels and depending on which channel input leads the other, encoder direction is determined by the module. In trigger mode, two timers are linked in which one timer is set to measure the time between two rising edges on the counter of the other timer. The rising edges act as a trigger to start measuring time. This is useful in encoder interfacing because it allows for speed and acceleration measurements, not only position measurement by counting the number of rising edges. “Get” functions are used to return important measured values from module registers without bothering the user about the location of these registers in memory or how to interface them.

A close-up of a paper

Description automatically generated with medium confidence

Text

Description automatically generated

**Encoders Driver Logic and Algorithm**

Drivers of the HAL layer use MCAL drivers to not deal directly with the microcontroller registers. Also, the HAL Drivers should not bother the user with initialization needed for any MCU peripheral. Following this concept, we implemented the encoder driver to be abstract as the user only specifies the pins on which the encoder is connected. All the heavy lifting to initialize timers, GPIO, and clocking is done by the driver.

Text

Description automatically generatedOther driver functions are related to the encoder operation such as speed and rotary position with direction. Each encoder instance created by the user is saved in a structure that emulates classes but with C language capabilities.

By utilizing passing by reference, we made each function as a class function that changes in the parameters stored in the encoder instance structure. This abstract level of coding is built over the MCAL layer drivers to make it simple in debugging and coding.

Text

Description automatically generated Some configurations are necessary to be added by the user. These include encoder properties as pulses per revolution. A limitation that can be improved in future work would be the integration of multiple encoder types into the driver as it currently works with incremental encoders with double channels. Also, configuration can be made easier in the future to add multiple encoders with different pulses per revolution as currently all added encoder instances should have the same pulses per revolution.

**PID Controller Driver Logic and Algorithm**

Not all motors are easy to control like off the shelf servo motors with internal controllers that require only a PWM signal to specify position of the motor shaft. DC motors require a reliable controller to maintain position or speed with a feedback loop from a sensor like potentiometers or encoders. A feasible controller is the PID controller in which the sensor data of a potentiometer, for instance, represents the current position of a motor shaft. To calculate the error, the PID controller checks for the difference between the current position and the desired position and aims to reduce this error as time passes. This is done by proportional, integral, and differential error parameters. A properly tuned PID Controller should be responsive enough and with minimal error based on application requirements.

Text

Description automatically generatedTo discretize the continuous PID controller discussed, we need to specify a sampling time interval at which sensor data is being read. This is important because it sets a relation to the maximum frequency at which the sensor value can change and be measured reliably “as in Nyquist Theorem”. Solving this sampling time problem can be made by increasing the sampling frequency of the ADC but this comes at the cost of lower resolution. A sweet spot between the two edge cases is configured for the ADC.

Following the driver implementation of the HAL layer, we made an abstract layer in which the user only specifies the potentiometer pin and other PID related parameters. The output of the PID controller can be returned by using a function that uses previously defined parameters in the PID Controller datatype. Each instance of PID controller has its own parameters as seen in this structure.

Icon

Description automatically generatedIt is necessary to add other sensor controls like encoders and IMU to enable PID control over elbow and Wrist motions, respectively. This is easily done by replacing the potentiometer “Get Angle” function of the current position with Encoder function “Get position or speed” or IMU “Get orientation XYZ”. Error calculation and other PID Driver functions are the same as the error is calculated from a desired variable based on the sensor measurements type.