Driver Program Specifications

Contents

Notes:

For methods, an equivalent address “selfTypeDef* self” is assumed to already be included with whatever other arguments are shown. Methods that are common between drivers should be named for each individual driver (e.g. “enable” should be something like “enable_servo” or “enable_motor” respectively).

Servo Driver:

- Overview:

Upon startup, the servo should be able to enable itself and all of the information that is required to operate. It should be able to record its current position and work off of the predetermined zero position. There should be two position setting methods: one to set the absolute angle (relative to the predetermined zero) and one to set the angle relative to the last angle (so that a controller input scaled down by controller gains can be used as an input).

- Methods:
    - void enable()
        - Enables things like timer channels and the servo
    - void disable()
        - Not very required, but disables things like channels and the servo
    - uint32_t get_position()
        - Returns the current absolute angle of the servo (assuming only positive values)
    - void set_position(uint32_t angle)
        - Takes in an angle relative to the absolute zero angle and sets the servo angle to it (will likely require pulse width setting NOT PWM setting)
        - Stores the absolute angle
    - void change_position(int32_t angle)
        - Takes in an angle relative to the last angle and sets the servo angle to be this input difference (should be able to handle negative values)
        - Stores the absolute angle
        - Change_Position could be replaced by something like Change_Pulse_Width or something like that if required.

Motor Driver:

- Overview:

Upon startup, the motor should be able to enable itself and all of the information that is required to operate. It should be able to set the PWM (can't use the same method as servo because it can't do position control like servos do).

This can likely use the same driver that's been used previously.

- Methods:
    - void enable()
        - Enables things like timer channels and the motor
    - void disable()
        - Not very required, but disables things like channels and the motor
    - void set_pwm(int32_t PWM)
        - Takes in a PWM and sets the motor PWM to it

Encoder Driver:

- Overview:

Upon startup, the encoder should be able to enable itself and all of the information that is required to operate. This includes a pair of timers to record position (and handle value overflow either using the 32 bit maximum variable value or synchronously with the encoder timer overflow). This will be used to determine the current position as an input to the control loop for setting position when not using the accelerometer or photoresistors.

- Methods:
    - void enable()
        - Enables things like timer channels and the encoder
        - Will need a total_count variable or something similar that is different from a prev_count variable that just stores the last position in the encoder timer (since the timer will overflow).
    - void disable()
        - Not very required, but disables things like channels and the encoder
    - void set_zero()
        - Sets the recorded encoder value to zero at its current location
    - int32_t read_position()
        - Returns and stores the new current position by comparing the previous position to the new position using the encoder timer.
        - Will need to handle overflow of the encoder timer.

Radio Receiver Driver:

- Overview:

Upon startup, the radio receiver should be able to enable itself and all of the information required to operate. It should take in the pulse width received from the radio receiver and interpret it as a percentage of the maximum value (or do this post). In main, this value will be evaluated for around 50% on or off.

This can likely use the same driver that's been used previously.

- Methods:
    - void enable()
        - Enables things like timer channels and the radio receiver
    - void disable()
        - Not very required, but disables things like channels and the radio receiver
    - uint32_t get_pulse()
        - Returns either the percent or absolute pulse width (then post process)

Photoresistor Driver:

- Overview:

Upon startup, the photoresistor ADC pins should be able to enable itself and all of the information required to operate. It should be able to read the ADC voltage values from the photoresistor pins and compare the horizontal and vertical pairs to find the difference between them. It should handle correct positive/negative values according to how the motor/servo should rotate or tested prior to mounting. This driver can be made for the individual photoresistors, but for this, assume the driver should be made for all 4.

- Methods:
    - void enable()
        - Enables things like 4 ADC pins and photoresistor values
    - void
        - Not very required, but disables things like ADC pins and photoresistor values
    - float get_ADC_value(uint8_t photoresistor_num)
        - Returns the photoresistor's value
        - Watch out for return value resolution, but values should range from 100/120*Vref to 100/200*Vref for illuminated inputs, and 0V for dark input
    - float get_ADC_difference(uint8_t photo_num1, uint8_t photo_num2)
        - Returns the photoresistor value difference between the two input photoresistors
        - Can use get_ADC_value to call the values

IMU Driver:

- Overview:

Upon startup, the IMU should be enabled, including its communication protocol pins, channels, and initializing of addresses required for the IMU to send gyroscope information (accelerometer values can be ignored). The IMU should be able to return angle data from all 3 axis. From testing, these values should be interpreted to determine if they are Euler angles or Quaternion angles and have the angle calculations corrected accordingly.

- Methods:
    - void enable()
        - Enables things the pins, channels and timers, and send initial register values to be able to receive gyroscope values.
    - void disable()
        - Not very required, but disables things like the pins and channels
    - uint32_t get_angle()
        - Returns the angle data using the I2C protocols (see DigiKey tutorial)
        - This data will likely be sent in 6 bytes of information and need to be collected and parsed for the 3 angle values

Controller Driver:

- Overview:

Upon startup, the controllers should be generated with their associated controller gains which can be updated later if required. Each control loop can either use the same controller with new controller gains or create a new controller. The controller should be able to set the gain, set the target position, and be able to take in an input current value to produce an output scaled by gain. This can be done through proportional, derivative, and integral gains, but for simplicity can be done with just proportional gain if needed.

This controller should be usable with any of the prior driver pairs for sensors and actuators. See Appendices for relevant control loops.

- Methods:
    - void enable(uint32_t kp, uint32_t kd, uint32_t ki)
        - Enables the controller with controller gains
    - void disable()
        - Not very required, but resets the controller gains and target position
    - void set_gains(uint32_t kp, uint32_t kd, uint32_t ki)
        - Takes in controller gains and sets them within the controller
    - void set_target(uint32_t OR float target)
        - Takes in either an integer or float, depending on the inputs we plan on using, and sets the target value within the controller
    - int32_t get_output(uint32_t OR float input)
        - Takes in either an integer or float, depending on the inputs we plan on using, and uses the controller gains and target value to calculate the output.
        - If derivative or integral gains are to be used, difference in times will also be required to measure.

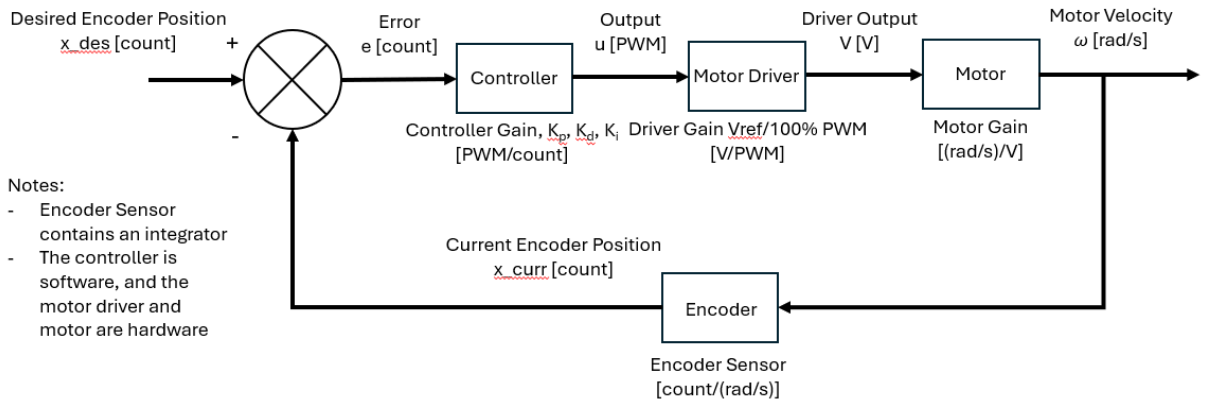
Appendices

Control Loops for various operations.

Appendix A

Motor PWM (speed) driven by encoder position outputs for standard position setting operations.

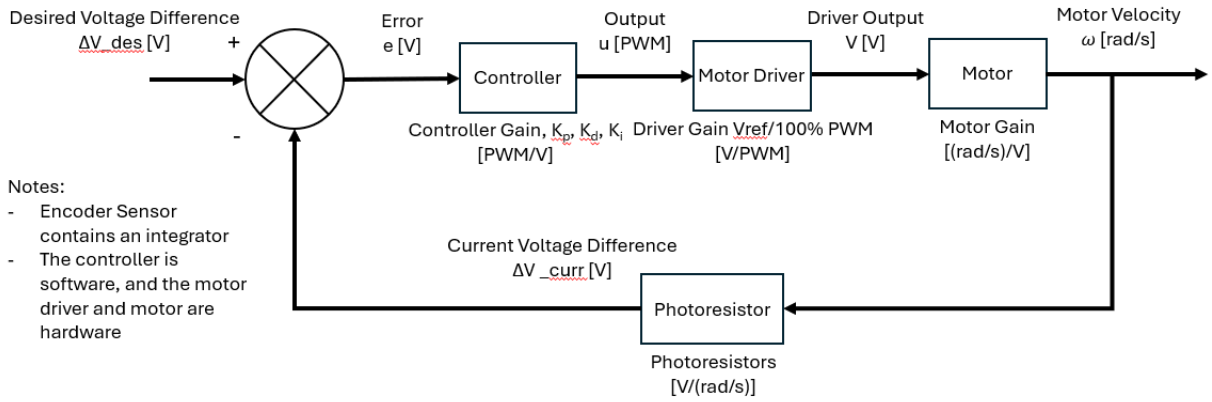
Encoder-Motor Control Loop



Appendix B

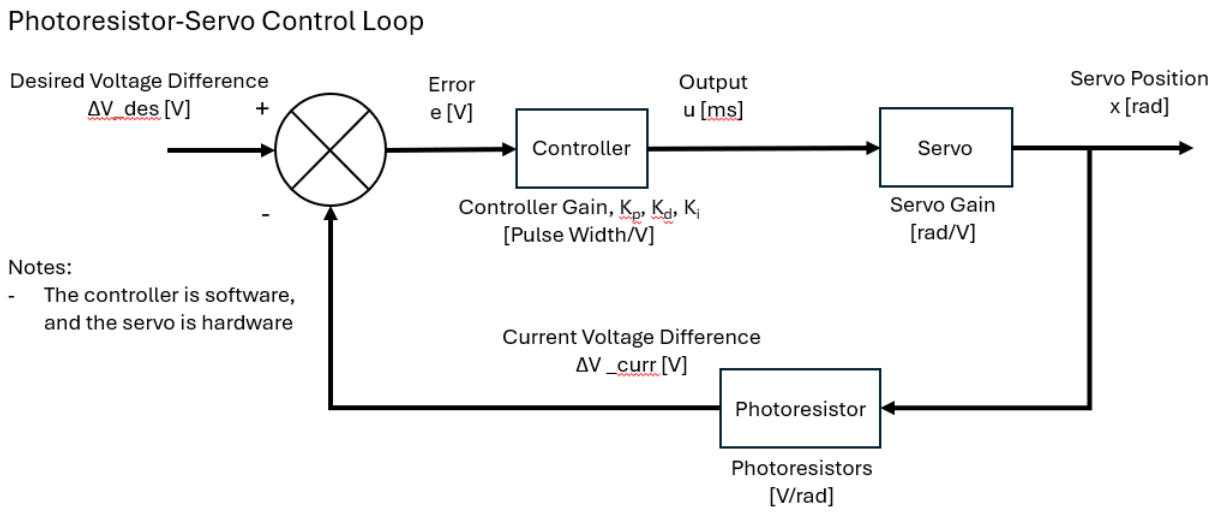
Motor PWM (speed) driven by photoresistor voltage difference outputs for after a starting position has been found to then determine where the light source is located.

Photoresistor-Motor Control Loop



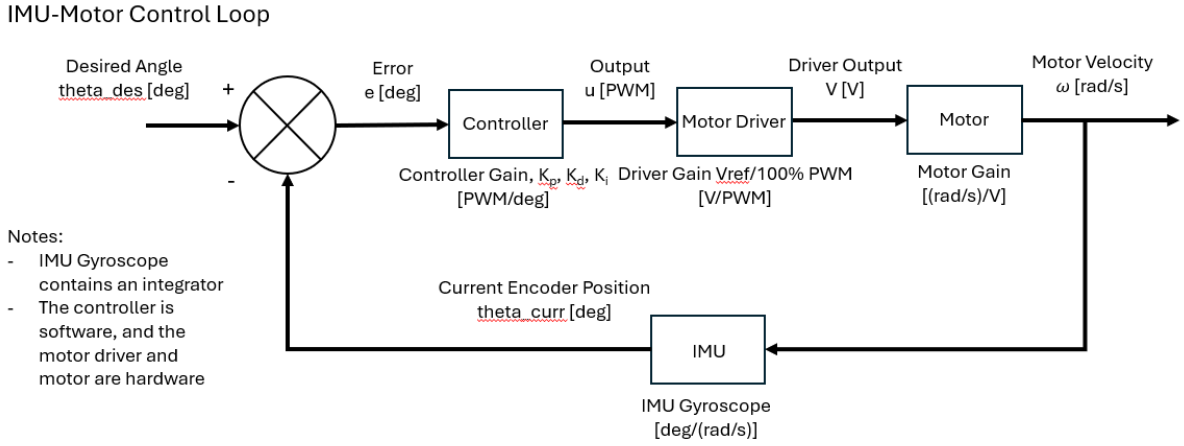
Appendix C

Servo pulse width (position) driven by photoresistor voltage difference outputs for after a starting position has been found to then determine where the light source is located.



Appendix D

Motor PWM (speed) driven by IMU angle outputs for after the light source has been located and a new angle has been calculated.



Appendix E

Servo pulse width (position) driven by IMU angle outputs for after the light source has been located and a new angle has been calculated. This may not be required if the set_position() command is accurate enough at setting the angle.

IMU-Servo Control Loop