B31DG Assignment 2 Report

Conor O'Brien – H00249212

Tasks, Timers and Synchronisation

The solution consists of a set of fixed and non-fixed period tasks implemented using both FreeRTOS task and FreeRTOS timer constructs. All but one of these periodic tasks begin to be scheduled as soon as the FreeRTOS scheduler is started, and repeat indefinitely.

The one exception uses the FreeRTOS timer API and is responsible for toggling the Arduino's onboard LED, which repeats at a 4Hz rate for 3 seconds. This task does not begin with the FreeRTOS scheduler, but rather is triggered every time both motor PWM signals exceed a WARN_VEL threshold and the program's diagnostic mode is enabled.

Of the remaining tasks, those which are periodic are designed to meet timing requirements. These include a task emulating sensor processing for 5ms every 20ms, and a task responsible for monitoring and computing rolling averages for both motor PWM signals with ½ Hz period. Both of these tasks are similarly implemented using the FreeRTOS timer API.

Two other periodic tasks are implemented instead using the FreeRTOS tasks API along with the Arduino's clock and timer1 to trigger interrupts at the desired periods via the millis() function.[1]

The first such task is responsible for indicating the maximum sensed currents computed using the analogue inputs read from the Motor Shield's SENSA and SENSB pins – over 2 second periods and since the last time the program's diagnostic mode was enabled, and only when the diagnostic mode is enabled – via the Arduino's serial port[2].

The final fixed period task involves sending kinematics data – the robot's estimated 2D linear and angular velocity, and position and orientation – periodically every second through the serial port. This duty is carried out within a FreeRTOS task which computes this data using a basic forward kinematic model, and which additionally sends this data every time either of the motor PWM signals changes.

The two remaining tasks are FreeRTOS tasks which repeat indefinitely but are not explicitly periodic, the scheduler alone determines when they take place.

The first of these tasks is responsible for polling the brake pin, engaging and zeroing both motor PWM signals, or disengaging the brakes (no debouncing), but also polling and debouncing the 5 other input pins. This task has higher priority (4) than all of the other FreeRTOS tasks (0), but also higher than each of the FreeRTOS timers (3).

The last remaining task processes events resulting from the debounced inputs, increasing and decreasing both motor input velocities and toggling the diagnostic mode, as well as writing the motor PWM signals and motor directions to their respective pins.

---

[1] This is a by-product of the way I used timers. I effectively implemented my FreeRTOS timers as entire tasks, meaning that I did not have enough stack to place all my "timers" in FreeRTOS timers. In hindsight, and considering that all timers share a stack, a better approach might have been to use timers to set flags in global state for separate FreeRTOS tasks to access or mutate, although this would further degrade synchronisation.

[2] Note that this task is only responsible for sending max sensed current information to the serial port and depends on a separate task to track the maximum analogue inputs when diagnostic mode in enabled and reset them when diagnostic mode is disabled.

Semaphores and Local and Global State

The program stores task states at the local scope where possible, for instance intermediary variables for e.g., position and velocity calculations, rolling average velocities, previous state for timing counters, debouncing or event processing and counting repetitions of the velocity warning task.

Where data sharing between tasks in required, state is stored at the global state. This is the case for button inputs, diagnostic mode and braking flags, motor velocities and increase/decrease flags and current sensing input signals. Associated semaphores are used to control access to critical regions accessing global state to protect against race conditions.

Assumptions

The axle length is defined as a constant of 10 units, and the linear velocity range is assumed to be [-1.0, 1.0] units, with -1.0 representing full velocity reverse for each motor (100% duty cycle PWM and direction pin HIGH), 0.0 representing zero velocity (0% duty cycle PWM and direction pin LOW), and 1.0 representing full velocity forwards (again 100% duty cycle PWM but with direction pin LOW).

Real World System Testing and Validation

Use Case Testing

To emulate the motor shield, LEDs were attached to the motor shield's PWM, direction and brake input pins. Additionally, a potentiometer simulates the motor shield's sensed voltage analogue outputs.

Comprehensive testing was performed by recreating use cases including edge cases by interacting with the system's state using buttons and verifying that the corresponding serial output and oscilloscope measurements meet each system requirement. The tests performed, along with the required response are listed below. A video demonstrating these tests is available at https://youtu.be/FCuon-5mO7w.

Test assertion	State	Expected Response	Pass/ Fail
The system's response to the **initial state** is correct	Initial state: no buttons pressed, diagnostic mode off, brakes disengaged, Unknown input from motor shield sense pins	The system transmits estimated linear/angular velocity and position/orientation via the serial port periodically at 1Hz rate. This estimated kinematic data is zeroed and does not change	Pass
The system responds correctly to an extended **increase left/right motor velocity button press**	Left/right motor velocity increase button down or released after having been pressed for longer than the	- The system updates and transmits kinematic data as soon as the button has been debounced, as well as at a 1Hz rate - Transmitted data reflects a single **20% increase** in velocity - Position and orientation change with time in accordance to forward kinematic model	Pass

	debouncing period	- Associated PWM duty cycle **increases by 20%** if not already at max - Associated direction pin **OFF** only if new linear velocity is **positive**	
The system responds correctly to an extended **decrease left/right motor velocity button press**	Left/right motor velocity **decrease** button down or released after having been pressed for longer than the debouncing period	- The system updates and transmits kinematic data as soon as the button has been debounced, as well as at a 1Hz rate - Transmitted data reflects a single **20% decrease** in velocity - Position and orientation change with time in accordance to forward kinematic model - Associated PWM duty cycle **increases by 20%** if not already at max - Associated direction pin **ON** only if new linear velocity is **negative**	Pass
The system responds correctly to any hint of braking	Brake button is on or off after being pressed for any amount of time	- Braking state change is indicated by a message transmitted via the serial port - Both brake pins are high for the duration of the brake button pressed - Both motor PWM signals are reset to zero duty cycle - Kinematic data reflects new zeroed velocities	Pass
The system responds correctly to diagnostic mode	- Diag mode on	- Current draw info correct, transmitted via serial port at 0.5Hz rate	Pass
	- Diag mode off	- When and while both PWM signals above threshold duty cycle, onboard LED toggles for 3s @ 4Hz	Pass
The system indicates when brakes change state	- Brakes off -> on	- Message indicating that brakes are engaged is transmitted via serial port	Pass
	- Brakes on -> off	- Message indicating that brakes are disengaged is transmitted via serial port	Pass
The system simulates a sensor processing task	Any	- A 20ms periodic task waits for 5ms per period	Fail

Timing Requirements Validation

Timing requirements were validated using an oscilloscope where possible, and the serial monitor otherwise (max current notification). The following figures demonstrate some of these timing validations. The remaining requirements can be verified by looking at the oscilloscope and serial monitor in the previously linked testing and validation video.

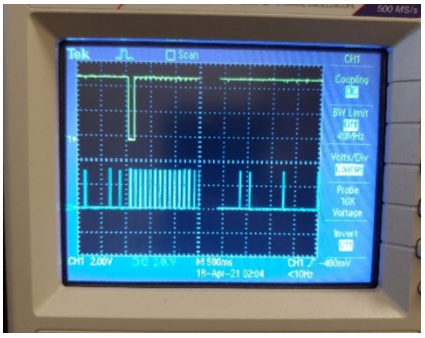


Figure 1: 0% to 20% left motor velocity increase after button press

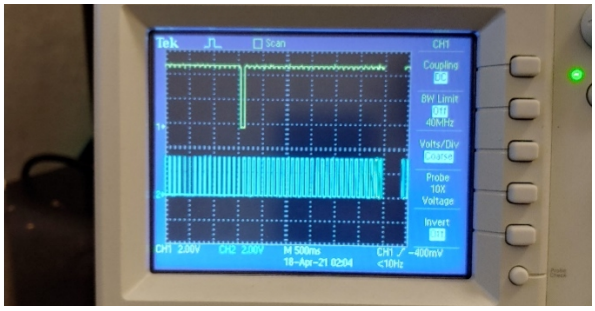


Figure 2: 20% to 40% left motor velocity increase after button press

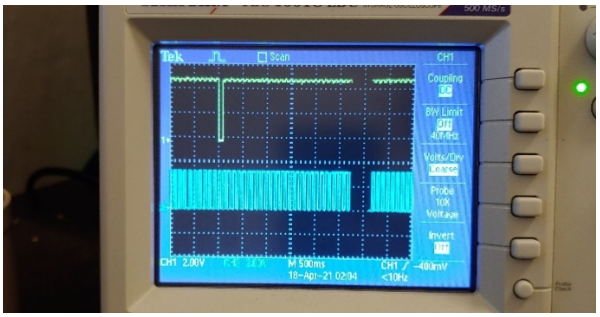


Figure 3: 40% to 60% left motor velocity increase after button press

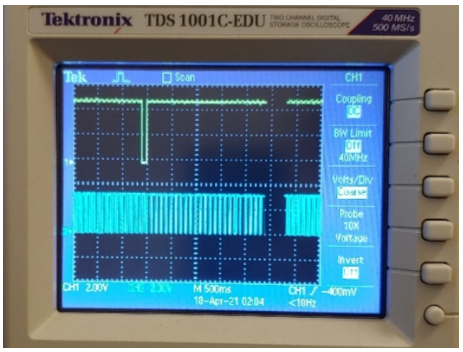


Figure 4: 60% to 80% left motor velocity increase after button press

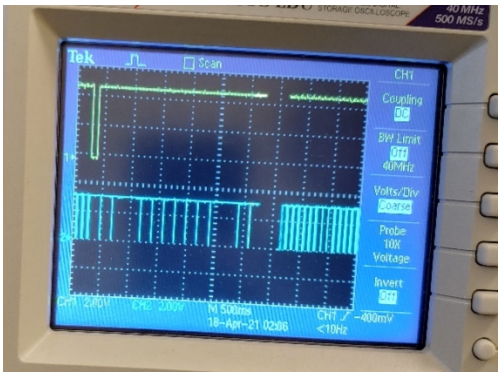


Figure 5: 80% to 100% left motor velocity increase after button press

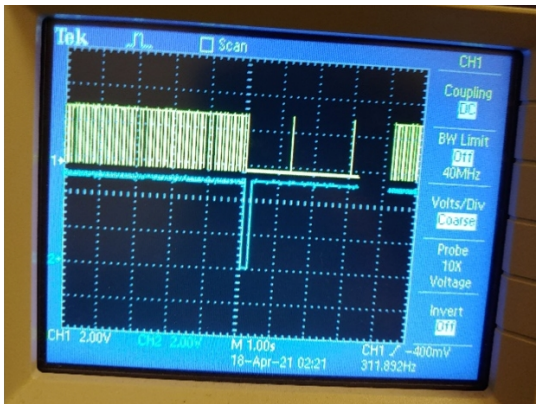


Figure 6:left motor PWM reset at first hint of braking

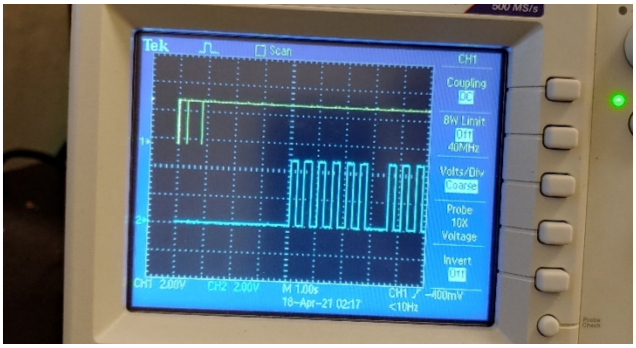
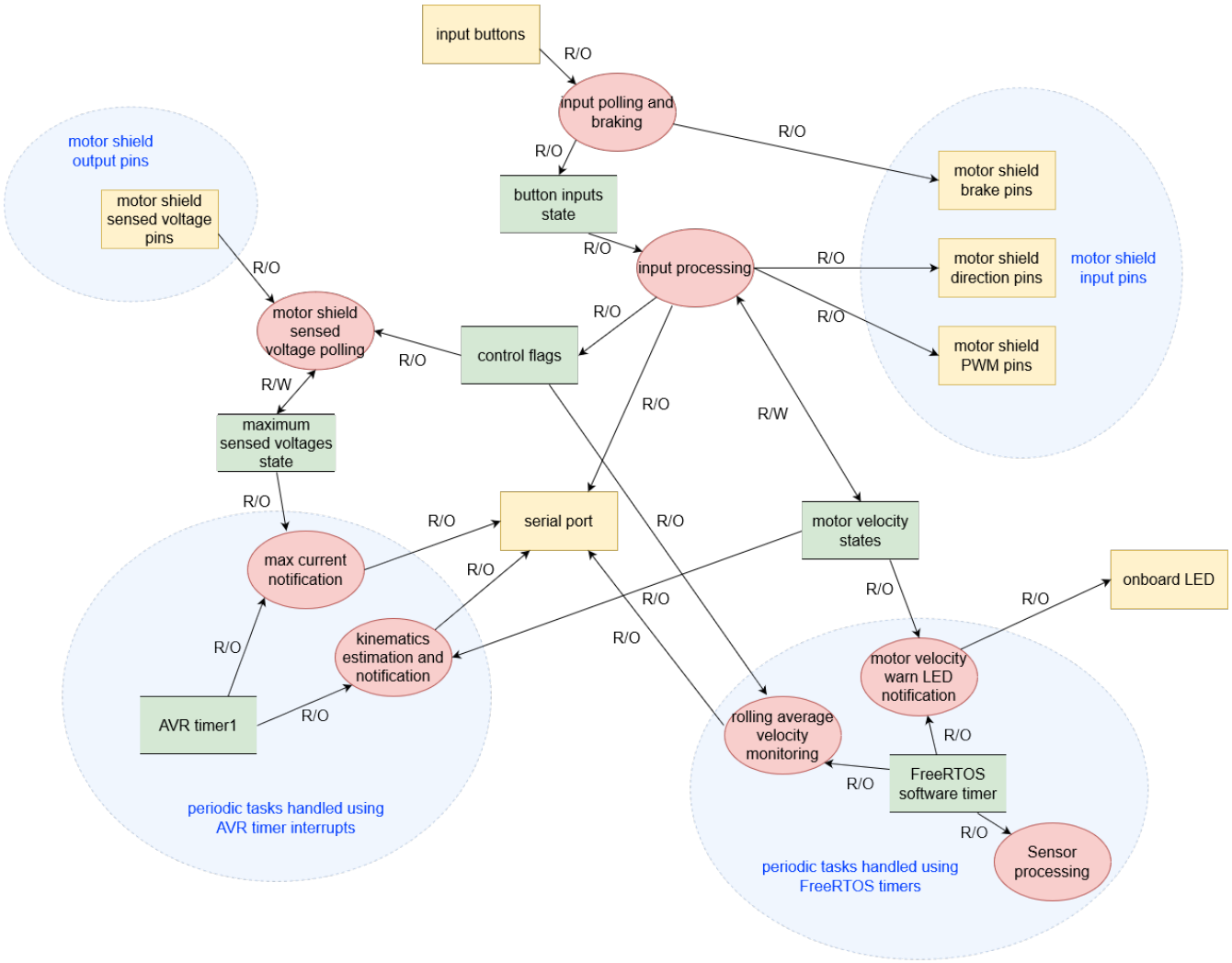


Figure 7: Warn velocity blink cycle

Dataflow diagram



Critical Analysis

Quality of Program and Strengths of Implementation

The binary is 46KB in size. Global variables take up 575 bytes (28% of the Arduino's memory), while the upper bound of the used stack at any given time is calculated as the sum of the allocated stack for each of the FreeRTOS tasks including the FreeRTOS task handling FreeRTOS timers. This equates to an upper bound stack size of 576 words (2304 bytes).

The program is designed to optimize for size where possible, for instance using bit fields as opposed to Booleans for storing button input states. Furthermore, the program is designed such as to maximise temporal and spatial locality, particularly within critical regions. Good temporal and spatial locality are achieved by using structs to group data which are accessed or mutated one after the other in time. Not only does this maximise the effectiveness of caching, avoiding expensive reads from main memory especially during critical regions, but also reduces the risk of deadlock and resource starving where multiple tasks access global state in critical regions concurrently.

Each of the hyperparameters likely to be changed by a user, including all of the timing periods, velocity rolling average circular buffer size, number of velocity warning LED blink cycles, the current factor calculated using the resistance values inside the motor shield, and

velocity increment step sizes, and maximum and threshold velocities are clearly defined and commented as constants at the beginning of the program, meaning that changes to the functionality amount to changing single constants. Additionally, all pins used are defined as constants and given explicit names.

Precautions have been taken to ensure the program is as robust as possible, for instance ensuring all local variables are properly initialised at the beginning of each task to prevent accessing uninitialized memory. This is necessary because the ATMega328P is not memory-mapped. Furthermore, struct copies are performed element-wise as opposed to using pointer manipulation. While this is slower, it removes the danger of undefined behaviour that could be caused by accessing dead pointers in the case of memory fragmentation.

Limitations of the system

Due to stack size constraints and to limit computation time of the high-priority braking and input polling task, button inputs are debounced as a single unit. Indeed, when any of the input button states changes, the debouncing process begins. The cost of this design choice is that buttons pressed after the debounce delay begins may be registered at the end of the debounce delay even though they might not have been delayed long enough for adequate debouncing. This did not constitute any issue for me while testing, but would constitute a concern in a use case calling for multiple button pressed within periods smaller than the debounce delay (50ms).

Another limitation has to do with integer division rounding errors. The 8-bit *velocity* variables used by the program are mapped to PWM duty cycles

$$PWM\ duty\ cyc\,;\ e = 2 * \left| velocity - \frac{maxVelocity}{2} \right| - 1$$

Similarly, linear velocities are mapped from the same 8-bit *velocity* variables are mapped to scaled integers then divided and cast to floats

$$linearVelocity = integerMap\left(velocity\,, 0\,, maxVelocity\,, -1000\,, 1000\right) / 1000$$

Both computations incur rounding errors which translate to PWM duty cycles and linear velocities being affected by errors whose upper bounds are $\frac{1}{2} velocity * maxVelocity$ and $\frac{1}{2} velocity * max\ Linear\ Velocity$ respectively. Depending on the use case, this error is likely less significant than mechanical errors, and I therefore consider the error to be an acceptable cost considering it enables the motors to operate in both forward and reverse directions.