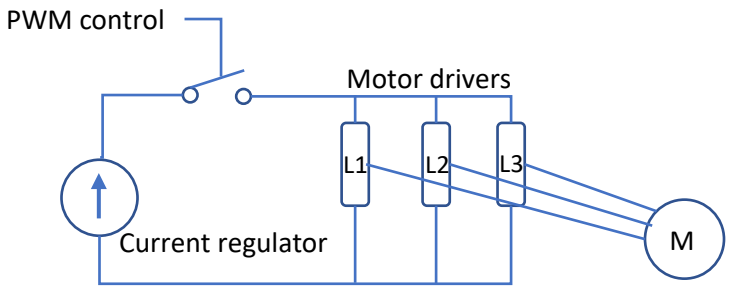
Lab instructions – Coursework 2, Part 3

Motor control

Controlling motor torque

Motor torque (turning force) is created when the magnetic field generated by the motor windings is not aligned with the magnetic field generated by the permanent magnet of the spindle. So far, the ISR sets the winding field to lead the spindle field by two rotor segments, which is 120°. The motor turns through one segment (60°) before the next interrupt, by which time the lead has reduced to 60°. This generates the maximum torque possible for a motor of this configuration.

To control the torque we will modulate motor current with PWM. The microcontroller contains PWM hardware that can produce rectangular waves on the output. Setting the duty cycle to 50% would reduce the average motor current to 50%. The PWM signal controls the current regulator, switching the motor current on and off. The current regulator in the circuit is gated with a switch that allows current to all the motor drive circuitry to be switched on and off



1. Create a `PWMOut` class to control pin PWMpin (pin D9). The class reference is here: https://os.mbed.com/docs/mbed-os/v5.11/apis/pwmout.html
2. Set up the PWM period and initial duty cycle in your initialisation statements in `main()`. Set the period to be 2ms, but this will change if you implement the tune generator.
   ⚠ Make sure the PWM is initialised and the duty cycle set to maximum before calling function motorhome(), otherwise there won't be enough motor current to reach the origin position.
3. Add a new serial command that will let you set the motor torque. This will be superseded by a controller in time, but it will be useful to test that it is working. You should be able to vary the speed of the motor.

Measuring motor position and velocity

Keeping track of motor position is best carried out in the photointerrupter ISR since any alternative would require a high-priority thread to count rotation events and the overhead of this would be high. Every sixth interrupt represents one revolution and it is straightforward to increment or decrement a counter each time.

The question is: which way is the rotor turning? You cannot assume it is turning in the direction that it is being driven. To do this you need to compare the current rotor state to the previous.

Velocity is best calculated on a fixed interval by differentiating the position. Do this in a new thread that will also report the position and velocity to the host and, eventually, it will become the controller.

Time (ms)	0	20	40	60	80	100	120	140	160	180	200
Rotor State	0	1	2	3	4	5	0	1	2	3	4
Position	0	1	2	3	4	5	6	7	8	9	10
Velocity	-	-	-	-	-	50	-	-	-	-	50

Example of position and velocity measurements over time. Velocity is calculated every 100ms

1. Add code to the photointerrupter ISR to compare new and previous rotor states and find the direction of movement. Increment or decrement a global variable as appropriate. Make the variable an integer and consider it to represent revolutions times 6, rather than whole revolutions, to avoid using costly floating point arithmetic. That is, a count of 6 will equal one revolution.
2. Create a thread containing a task that will execute every 100ms. The method for this is to use the `Ticker` class to set up a timer interrupt. The interrupt will send a signal to the thread that will tell it to run the task.
    a. Create a new thread. This time, we will use the non-default constructor that allows us to specify the priority of the thread and the maximum stack size. You may wish to change thread priorities later, but begin by using the default, which is `osPriorityNormal`.
    Each thread has an independent stack in a block of memory allocated from the heap when the thread is created. There must be enough stack space to store all the local variables required at the deepest (or otherwise worst-case) level of function calls. To work out the necessary stack size exactly would require studying the compiled object files, but 1024 bytes should be sufficient. The RTOS will usually write an error message to the serial terminal if it runs out of memory.

```
Thread motorCtrlT(osPriorityNormal,1024);
```

    b. The thread will run a function containing an infinite while loop. Before the while loop starts, create a ticker object and attach a callback function `motorCtrlTick()` that will run every 100ms:

```
void motorCtrlFn(){
    Ticker motorCtrlTicker;
    motorCtrlTicker.attach_us(&motorCtrlTick,100000);
    while(1){
       ...
```

c. `motorCtrlTick()` will be an ISR triggered by the Ticker. The motor control task is a lower priority than the photointerrupter ISR task so, to avoid blocking the CPU, `motorCtrlTick()` will not do any computation itself. Instead it will just send a signal back to the motor control thread:

```
void motorCtrlTick(){
    motorCtrlT.signal_set(0x1);
    }
```

d. The RTOS function `Thread::signal_wait()` causes a thread to block until the requested signal number (in this case 0x1) is set. Therefore, the contents of the while loop will run once per ticker event. The signal is automatically cleared once it is received.

```
while(1){
     motorCtrlT.signal_wait(0x1);
     ...
```

3. After the thread has been released by the signal, it should calculate the velocity. Simply find the difference between the current position and the position on the last iteration and multiply by 10 (since the interval is 0.1 seconds). This method can be inaccurate if the function doesn't run exactly every 100ms, maybe because another task is running. So, if you want to improve the accuracy add a timer and measure the exact interval.

   ⚠ Consider the concurrent access of the position variable, which could be updated by the ISR at any time. Use a critical section, which temporarily disables interrupts, to access variables that could be accessed simultaneously by an ISR. Keep the critical section as short as possible:

```
core_util_critical_section_enter();
//Access shared variables here
core_util_critical_section_exit();
```

4. Count the iterations of the velocity calculation task. Every tenth iteration (once per second) create messages on the output channel that report the position and velocity.

Controlling motor speed

Motor speed is easier to control than position because velocity is the first integral of acceleration. That means you can get reasonable results by using simple proportional control.

a. Decode the serial port command for setting rotation speed.
b. Implement proportional speed control in the motor controller thread:

$$T_s = k_{ps} e_s$$

Where $e_s = s_{\text{max}} - \left|\frac{de_r}{dt}\right|$, the difference between the maximum speed setpoint (set by the serial port command) and the current speed. T_s is the speed controller output. Since $\frac{de_r}{dt}$ can be negative, depending on the direction of rotation, it must be converted to an absolute value. k_{ps} should be set experimentally to the maximum that avoids oscillation – try a value of 25 to start with.

T_s will be the motor torque, but it will need a little conversion to make it into a PWM pulse width:

I. Negative values should be made positive. A negative value of T_s is valid because it represents a reverse torque but the PWM pulse width cannot be negative. Instead, if T_s is negative the value of `lead` should be set to -2 instead of 2.

II. Apply a limit so that the PWM duty cycle cannot exceed 100%.

c. Test the speed controller. Start with a fast speed (e.g. 50 rotations per second) and then find the slowest speed that the motor will run reliably. If you reduce the speed setting you should see the rotations slow quickly as the controller reverses the torque direction.

d. You may observe a steady-state error, where the motor speed stabilises at a value below that requested due to friction. You can remove it by adding an integral term to the controller (see lecture notes).

Next steps

1. Add position control. Overshoot is important here because the motor cannot stop instantly when it reaches its target. Don't try and fix this by using heuristics that turn the motor off a bit early; use a differential term in the controller. The differential term is not quite the same as the velocity; it's the rate of change of the position error not rate of change of position. The magnitude is the same, but the sign may be different.

$$T_r = \left(k_{pr} e_r + k_{dr} \frac{de_r}{dt}\right)$$

Here, k_{pr} and k_{dr} are the tuning parameters. e_r is the position error: the setpoint position minus the current position. k_{dr} controls the system damping: too small and the motor will overshoot, too large and the movement will take longer than necessary. Try $k_{dr} = 20$

Try the position controller independently before combining it with the speed controller.

2. The speed controller must be modified because now the motor can be spinning in either direction. This is done by multiplying by the sign of E_r. So if the motor needs to go forward, the velocity limit is positive and vice versa:

$$T_s = \left(k_{ps} e_s + k_{is} \int e_s \, dt\right) \operatorname{sgn}\left(\frac{de_r}{dt}\right)$$

3. Position and speed control must be combined by choosing to apply T_s or T_r. Do this by calculating both and choose the value which is lowest or most negative, with respect to the direction of rotation. The most conservative controller output takes precedence.

$$T = \begin{cases} \min(T_r, T_s), & \frac{de_r}{dt} \geq 0 \\ \max(T_r, T_s) & \frac{de_r}{dt} < 0 \end{cases}$$

For example, if the position setpoint is a long way from the current position in the positive direction, T_r will be a large positive value but T_s will be less than one once the maximum speed is reached. T_s should be chosen because it is smaller and this will limit the speed to the maximum.

Once the position setpoint is near, T_r will become negative as the motor needs to slow down and T_s will increase because the speed is below target. Now T_r should be selected.