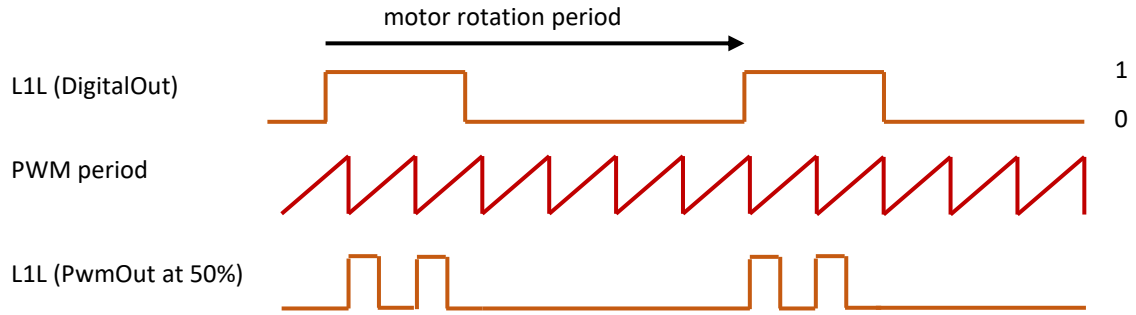
Lab instructions – Coursework 2, Part 3

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 10% would reduce the average motor current to 10%. Since the PWM will be replacing the standard digital outputs, the duty cycle must be changed on every interrupt to generate the coil energising sequence.



Timing diagram showing one motor output pin before and after conversion to PWM

1. Replace the class `DigitalOut` with `PWMOut` for the pin objects `L1L`, `L2L` and `L3L`. Only the low-side drivers will be controlled by PWM – the high side drivers will be switched as digital outputs as before. This works because both a high-side and a low-side driver need to be turned on for current to flow, so modulating the low-side only is sufficient. https://os.mbed.com/handbook/PwmOut
2. Set up the PWM period in your initialisation statements in `main()`.Set the PWM period with class method `PwmOut::period_us(int us)`

   ⚠ There is a hardware bug on the motor PCB that constrains the use of the PWM. You need to use a workaround, which is to set the PWM period to 2000µs and to limit the PWM duty cycle to the range 0-50%. Above 50% there is a non-linear relationship between duty cycle and motor speed.
3. Modify the `motorOut()` function to set the PWM. Add an `uint32_t` argument to the function prototype, which will be used to pass the motor torque variable. Replace each assignment to `L1L`, `L2L` or `L3L` with a call to the method `PwmOut:: pulsewidth_us(int us)`. Where the output is turned off, continue to do so by setting the pulse width to zero. Where the output is turned on, set the pulse width to the torque variable you have passed to the function.
4. 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 won't be able to protect the shared variable for motor torque with a mutex this time because you cannot use a mutex in the ISR where the variable will be read. However, if the variable is type `uint32_t` we can be confident that the accesses to the variable will be atomic; i.e. they take place in a single CPU instruction and cannot be interrupted part way through. Ideally, you would check the assembly output or write the accesses with inline assembler instructions to guarantee this behaviour.

Update the calls to `motorOut()` to pass the additional motor torque argument. The call in `motorHome()` should request maximum duty cycle and the call in the photointerrupter ISR should use the value written by the command interpreter.

Measuring motor position and velocity

Keeping track of motor position is best carried out in the photointerruptor ISR since any alternative would require a high-priority thread to count rotation events and the overhead of this would be high. Each interrupt represents one sixth of a 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. We will 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. The new ISR looks something like this:

```
int32_t motorPosition;
void motorISR() {
    static int8_t oldRotorState;
    int8_t rotorState = readRotorState();
    motorOut((rotorState-orState+lead+6)%6,motorPower);
    if (rotorState - oldRotorState == 5) motorPosition--;
    else if (rotorState - oldRotorState == -5) motorPosition++;
    else motorPosition += (rotorState - oldRotorState);
    oldRotorState = rotorState;
    }
```

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. The default stack size is 2K bytes and you will probably run out of RAM if you try to create more than 2 threads of this size – the stack sizes must be reduced.
    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. If the position variable is a 32 bit integer or smaller then you can

consider a read access to be atomic. But you must make sure there is just one access – if you use the position in more than one statement then copy its value to a local variable first.

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 velocity

Motor velocity 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:

$$y_s = k_p(s - |v|)$$

y_s is the speed controller output, s is the maximum speed setpoint (set by the serial port command) and v is the measured velocity. Since v can be negative, depending on the direction of rotation, it must be converted to an absolute value. k_p should be set experimentally to avoid oscillation – try a value of 25 to start with.

> ⚠ In mbed, floating point type `float` is 32 bits and type `double` is 64 bits. Accesses to a `double` will not be atomic and should be protected from concurrent access.

c. y_s will be the motor power, 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 y_s is valid because it represents a reverse torque but the PWM pulse width cannot be negative. Instead, if y_s is negative the value of `lead` should be set to -2 instead of 2.
   II. Apply a limit so that the PWM pulse width cannot exceed the maximum value (1000µs in this case due to the hardware limitation).

   The extra processing is best done within the photointerrupter ISR so that the existing variable `motorPower` is used for the controller output y_s. This adds extra computation to the ISR but the advantage is that we retain a single variable to pass information from the motor control thread to the ISR and we can continue to rely on atomic data access to ensure thread safety. For this reason, there should be just one access of `motorPower` in each loop of `motorCtrlFn()`.

d. If the command `V0` is given, motor power should be set to maximum (positive for now, because no direction control is implemented).
e. 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. Also observe how quickly the motor slows down when the speed is reduced. The motor should slow more quickly than it would naturally thanks to reversal of `lead`.
f. Choosing k_p is a trade-off. Reducing it will increase the *steady-state error*, meaning that the motor will spin slower than requested. Increasing it may cause the speed to fluctuate, especially at slow speeds, due to the lag of the controller.

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.

$$y_r = k_p E_r + k_d \frac{dE_r}{dt}$$

Here, k_p and k_d are the tuning parameters. E_r is the position error: the setpoint position minus the current position. k_d controls the system damping: too small and the motor will overshoot, too large and the movement will take longer than necessary. Try $k_d = 20$

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:

$$y_s = k_p(s - |v|)\mathrm{sgn}(E_r)$$

3. Position and speed control must be combined by choosing to apply y_s or y_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.

$$y = \begin{cases} \max(y_s, y_r), & v < 0 \\ \min(y_s, y_r), & v \geq 0 \end{cases}$$

For example, if the position setpoint is a long way from the current position in the positive direction, y_r will be a large positive value but y_s will fluctuate around zero once the maximum speed is reached. y_s should be chosen because it is smaller and this will apply the maximum speed.
Once the position setpoint is near, y_r will become negative as the motor slows down and y_s will increase because the speed is below target. Now y_r should be selected.