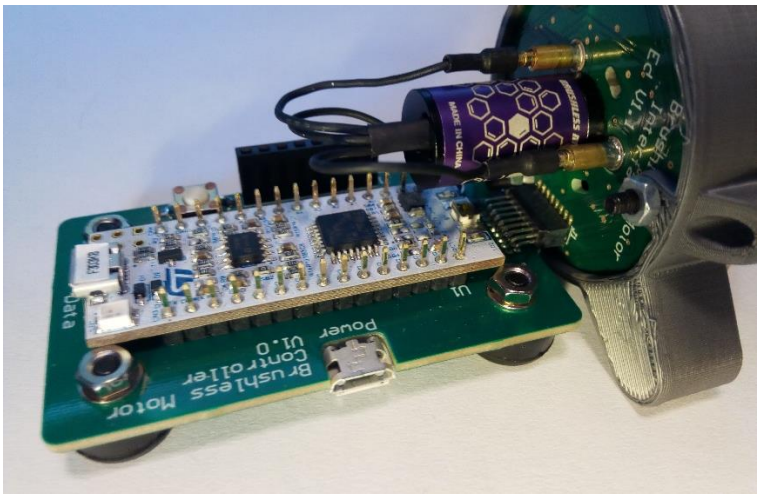
# Lab instructions – Coursework 2, Part 1

Getting started, using interrupts and Bitcoin mining

## Getting Started

Compile and download the starter code to get the motor running.

1. Get the starter code
    a. Log in to the mbed online compiler at https://os.mbed.com/compiler/ Create an account if you don't have one already.
    b. Find the starter code project at https://developer.mbed.org/users/estott/code/ES_CW2_Starter/ Click 'Import into Complier' to fork the project into your own Workspace
    c. Select the project. Check the target device is NUCLEO-L432KC – this is shown in the top right hand corner of the online IDE. If you don't have this target available, add it to your compiler using the link on this page: https://os.mbed.com/platforms/ST-Nucleo-L432KC/
2. Assemble and connect the motor kit
    a. Connect the motor module to the control board.




⚠ The pins are delicate. Take care!

⚠ Disconnect the motor module and control board when transporting. Protect the header pins from damage during transit.

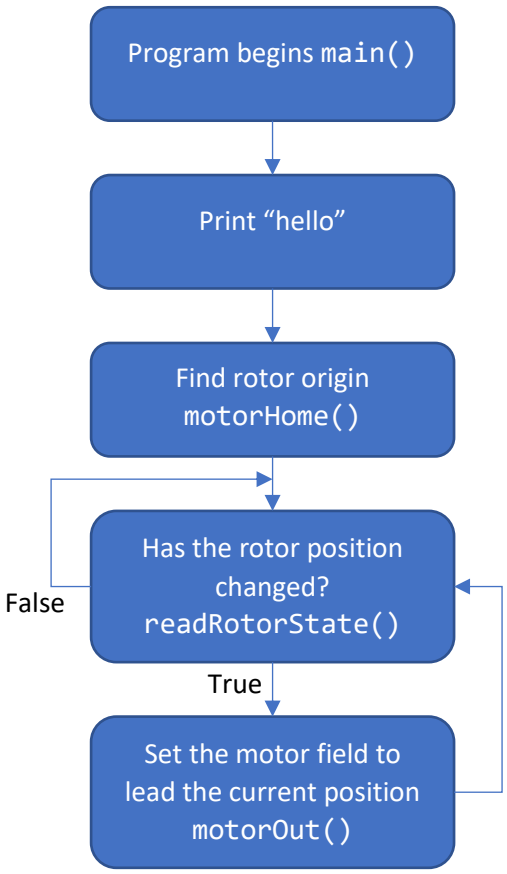
⚠ There are no alignment features on the connector. Check carefully that the connector parts are not transposed. Misalignment of the connector may cause damage.

    b. Connect the USB socket labelled 'data' to your computer. It should enumerate as three separate USB devices: a mass storage device (USB drive), a serial terminal (COM or tty port) and a proprietary ST-LINK debug device. You may need to install drivers for the serial terminal from here: http://www.st.com/content/st_com/en/products/development-tools/software-development-tools/stm32-software-development-tools/stm32-utilities/stsw-stm32102.html

    The ST-LINK device is not required for use with the mbed compiler

c. Connect the USB socket labelled ‘power’ to your laptop or a plug-in 5V power supply

3. Compile and run the starter code
   a. Click compile in the mbed online compiler. The project should compile and then offer you a .bin file download. Save it directly to the USB drive created when you plugged in the microcontroller module.
   b. The code should begin running straight away. You can also press the push button on the controller board to reset the microcontroller. If the power is connected, the motor will begin turning after a few seconds.
   c. Open a terminal to the device using the COM or tty port created when you plugged in the microcontroller. The baud rate is 9600. Reset the microcontroller and you will see some startup messages.
   d. Inspect main.cpp to see what the code does. The program flow looks like this:




   e. The code begins by setting the motor into state 0 and recording the position state of the rotor. This offset depends on the position of the rotor disc during assembly and is constant for each motor module. If the rotor is spinning when the code starts the incorrect offset may be recorded, so wait for the disc to stop before releasing the reset button. Even though the offset is constant, always leave the `motorHome()` routine in your code so that it works with different motors.
   f. The motor field position is calculated by reading the rotor position, correcting for the offset and adding a lead value so that the motor field leads the rotor and a torque is applied. The lead value is defined as a constant; try changing it from positive to negative to reverse the direction of rotation.

## Use Interrupts to initiate the motor sequencing task

The starter code works well but it is difficult to add functionality because it must constantly poll the rotor position to keep the motor field updated and the motor turning. This can be fixed using interrupts.

1. Change the type of the photointerrupter inputs I1,I2 and I3 from `DigitalIn` to `InterruptIn`
2. Move the code which checks the rotor position and sets the motor field into a new function, which will be the interrupt service routine (ISR). There is no need to check for a change in rotor position anymore because the ISR will only be executed when an input changes.
3. Attach the ISR to rising and falling edge events on each photointerrupter input using the `rise()` and `fall()` methods of `InterruptIn.` The API for `InterruptIn` is here: https://os.mbed.com/docs/mbed-os/latest/apis/interruptin.html
4. Your `main()` function should now end with an empty loop forever.
5. Check it works, the motor should spin as before but it may need a push because the ISR is only called if the motor is already moving. How can you fix this?

## Bitcoin mining

You are asked to run an encryption task with lowest priority on your microcontroller to use excess CPU time and prove that the real-time behaviour is correct. Such proof-of-work tasks are the core of blockchain systems like Bitcoin.

The encryption task is to carry out a *preimage attack* on the SHA-256 hashing algorithm. SHA-256 generates a 256-bit hash from a sequence of input data. A preimage attack tries to find a sequence of input data that generates a specific hash, which (if the hashing algorithm is good) is only possible by trying many sequences through brute force. In this case, you are looking for a sequence that produces a SHA-256 hash that begins with 16 binary zeros (this is similar to Bitcoin mining, but the number of leading zeros required there is much greater). Finding suitable sequences, which are easy to verify, proves you have done a certain amount of computational work.

The sequence is made up of 48 bytes of constant payload (this would be the transaction details in a crypto-currency), 8 bytes provided by the host (a key) and 8 bytes (a 'nonce') that you can alter to search for a matching hash. Each successful 'nonce' is sent back to the host as proof-of-work. With 16 binary zeroes required in the hash, a suitable 'nonce' will be found, on average, once every $2^{16}$=65536 attempts.

For now, run the mining tasks in the main loop as fast as possible. Later, you will need to regulate it to meet the throughput specification of 5000 iterations per second.

1. Import the crypto library by François Berder from here: https://os.mbed.com/users/feb11/code/Crypto/
2. Create an instance of the class SHA256 (from SHA256.h) in your main loop.
3. Declare and initialise the input sequence and hash as shown below. `key` and `nonce` are initialised as pointers to locations within `sequence[]` so they can be accessed independently.

```
uint8_t sequence[] = {0x45,0x6D,0x62,0x65,0x64,0x64,0x65,0x64,
     0x20,0x53,0x79,0x73,0x74,0x65,0x6D,0x73,
     0x20,0x61,0x72,0x65,0x20,0x66,0x75,0x6E,
     0x20,0x61,0x6E,0x64,0x20,0x64,0x6F,0x20,
     0x61,0x77,0x65,0x73,0x6F,0x6D,0x65,0x20,
     0x74,0x68,0x69,0x6E,0x67,0x73,0x21,0x20,
     0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
     0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};
uint64_t* key = (uint64_t*)&sequence[48];
uint64_t* nonce = (uint64_t*)&sequence[56];
uint8_t hash[32];
```

4. There is no efficient method for searching for the 'nonce', so just start at zero and increment by one on each attempt. `key` will eventually be supplied from the host and you can leave it as zero for now. Each value of key will result in a unique set of 'nonces' that generate the desired hash.
5. Use the `computeHash()` method of SHA256 to calculate the hashes. Its arguments are a pointer to the hash output, a pointer to the sequence input and the length of the sequence (64 bytes). Check the prototype in `SHA256.h` for details.
6. Test for both `hash[0]` and `hash[1]` equal to zero to indicate a successful 'nonce'. When you have found one, report it to the host by printing it on the serial port as 16 hexadecimal digits. You should find a 'nonce' once every few seconds.
7. Every second, report the current computation rate in hashes per second. Use the mbed `Timer` class to measure the time and simply count the number of hashes executed in each interval.
8. Print the hash rate on the serial port once every second. The motor should be running using the ISR so unplug the power cable to stop it. The microcontroller will continue running and you should see an increase in hash rate as the rotor stops turning and the CPU overhead from the photointerrupter interrupts stops.