EE3-24 Embedded Systems
Coursework 2

Imperial College London
Department of Electrical and Electronic Engineering

Mateo Sarjanovic - ms6616@ic.ac.uk
Jordi Laguarta Soler - jl9516@ic.ac.uk
Alessandro Serena - as6316@ic.ac.uk

22nd March 2019

1 Introduction

The purpose of this assignment was to implement a multifunctional software for the Nucleo-L432KC developer board. The system, will control a brushless motor and, in parallel thanks to multithreading, it performs a hashing function.

2 Hashing Function

Since hashing is a one-way function, in order to find a desired value (in this case we are looking for values that will produce a hash with 16 leading zeros) the only possible approach is brute forcing different values.

The input to the hashing function is a 64-byte number where the first 48 bytes are predefined. The remaining 16 bytes are associated with a 8-byte *key* that is initialized with value 0 but can be redefined by the user (to see how this is implemented refer to section 4), and an 8-byte *nonce*.
The algorithm consists of progressively increasing the value of the nonce (by 1 at each iteration) and applying the SHA-256 hashing function on the 64-byte number defined above until it produces an outptu that satifies the condition mentioned at the beginning of this section.
Once the desired output has been found, our system outputs the value of the *nonce* that has generated it to the serial interface.

This function is intended to be a proof of the fact that the CPU can allocate time to this computationally-intensive task. In fact there is little to no stall due to memory read-write overhead since the function only needs a very limited amount of memory: only the registers and L1 cache will be needed considering the 512-bit input sequence and the temporary variables generated by the hashing function.
The process runs on the main thread of our software architecture and, given its low requirements in terms of memory allocation, can be quickly pushed to and popped from the stack when the OS switches the execution context to another thread.

The only inter-thread dependency that this the process will encounter while executing this function happens when a correct *nonce* has been found and we want to print it. In this occasion the thread will call the function `putMessage` (described in more detail in section 4) to safely append the message to the message buffer handled by the thread that manages the serial interface. Note that this is a blocking call and it is indeed the only section in this function where the nonce search is interrupted.

3 Motor Control

The motor control is broken up into 5 main components.
There are two interrupt service routines(ISR), and three threads who in conjunction to ensure motor speed stays below the specified maximum, and the motor does not overshoot.

The interrupt service routines serve to drive the motor, and calculate the current position of the motor. Upon initialization of the motor, all three threads are instantiated: the *speed controller*, the *position controller*, and a thread that drives the motor. The *motor controller* varies the current feeding the motor through pulse width modulation (PWM) and calculates the current velocity at a rate of 10Hz. By changing the current, we are able to directly impact the speed of rotations, and ultimately turn it off.

When the R command is input, the serial communication sets a global variable equal to the amount of rotations we are aiming to perform. This value is constantly being monitored by the *position controller*. The *position controller* implements a proportional differential (PD) controller, with the position error (how many revolutions are left until target is met) and velocity as its inputs. The output of this controller is written to a global variable, that is to be handled by the *motor controller*.

When the `V` command is input, this sets a global variable indicating the maximum speed the motor is meant to run at. This value is constantly being monitored by the *speed controller* which implements a proportional integral (PI) controller, with the speed error (distance from max speed) as an input. The output of the controller is written to a global variable which is in turn dealt with by the *motor controller*.

The *motor controller* sets the motor current by alternating between setting the PWM pulse width to the output of the *speed controller* or the *position controller* depending on the current velocity.

Writing to global variables that are shared between processes is done with the use of mutual exclusion variables in order to guarantee safe access during concurrent processes. While the reading of global variables set by the ISRs is done in critical sections in order to disable interrupts and prevent an untimely update, leading to undefined behaviour.

4 I/O over Serial Port

One of the principal advantages of multithreading in an information system is that of avoiding race conditions and deadlocks on a certain resource. If we decide that only one thread in the process

architecture has read-write access to a resource, we can have all other threads to communicate with this when they need access.
Especially when we want to scale the number of parallel threads, ensuring mutual exclusion on a shared resource can be cumbersome and lead to unexpected race conditions. This solution will resolve any issue of concurrent access and provide a safe interface for other threads to use the resource.

In this case, the resource that we want to protect is the serial port. This is used by the Nucleo board in order to read user commands and to print out information about the system status.

The outbound communication from the threads to the serial port are handled by a global*message FIFO buffer*.
Threads can generate structured messages and append them to the buffer thanks to the interface provided by `putMessage`. This ensures that all messages have the same standard format (specifically `number string number`) and lets us have a content-independent message handler.

We decided to have a distinct thread handling the incoming messages generated by the other threads. This operates as defined by the function `Receiver`.
It polls the less recent message written to the FIFO queue and prints its value to the serial port interface.

User input from the serial port is handled in a separate thread from the output in order to guarantee there are no race conditions while concurrently inputting and outputting to the port. The thread receives a command form the port and decodes it, character by character, deciding which action to perform with the first character, and using the rest as parameters. Once the parameters are received, global variables are updated. The dedicated threads handle the rest of the operation from here on forward.

5 Real-Time Behaviour

The system performs its intended functionalities well, with threads operating successfully and safely from each other. The motor is controlled efficaciously with small undershoots or overshoots.
Although this might have been a problem linked specifically with the hardware provided us, one remark that we have it that we noticed that the system has a lower performance during the initial runs. We suggest the tester to let it run and reset it a couple of times before assessing its behaviour.