ENGT5259 EMBEDDED SYSTEMS
Laboratory guidance notes
Part 1: Intro

Author: J A Gow, M A Oliver *v1.0*

Table of Contents

Version record

Version	Notes	Date	By
1.0	Initial version	21/01/22	JAG/MAO

PLEASE READ THIS CAREFULLY

This is the first of a series of guidance documents to get you started with your embedded systems project. They are aligned with the key aspects of the coursework specification, and thus this document should be read in conjunction with the coursework specification document. These documents will start off with some fairly close guidance - however as you develop your skills the notes will provide less and less detailed guidance and become more of an overview.

Note that the implementation is yours, and **you are encouraged to experiment**, or to try different approaches. As long as (a) you clearly understand the code you have written and

(b) your result complies with the specification, you will pass. There is no 'right answers' in this module. The benchmarks are: (a) have you hit the learning outcomes and (b) does your final project code comply with the specification.

Note that this module is not a C/C++ programming course. Most embedded system development is carried out in C/C++ and you should have had exposure to these languages as part of your undergraduate studies. If you are rusty, or are not familiar with this language there are plenty of tutorials available online (some links are available on the Blackboard shell) that can guide you. It is possible to learn C/C++ alongside this project. They are not especially 'difficult' languages to learn when you are using them to solve a problem and there are some 'templates' available to help.

1 Aims and objectives

This document covers the initial setup of the environment, the use of the provided software templates and an introduction to the implementation of the requirement of section 4 of the coursework specification. This should take you about 2 weeks to work through.

1.1 Hardware and microcontroller platform

The hardware and microcontroller platform you will use for your design is the DMU-designed ATMega328P Microcontroller Development Board. These are available in Queens Q2.01 and Q1.01. This board is based around an ATMega328P microcontroller, and this Atmel AVR-based ATMega328P microcontroller is the target platform for your firmware. In this set of guidance notes you will be implementing code that allows you to communicate with and drive the 7-segment display.

1.2 Schematic

Embedded systems invariably involve interfacing hardware and firmware. To enable you to develop the firmware you will need to view the schematic to determine how the connections are made between the peripheral (in this case a 7-segment display) and the pins on the microcontroller that drive it. A schematic for the development board is available and may be downloaded from the Blackboard shell ('Lab guidance notes'->'Datasheets and schematics').

1.3 Template

The first exercise is aimed to familiarize you with the environment. To this end, a template is provided that performs some basic functions. It contains code to flash the built-in LED on the board at a constant rate (this is not part of the specification, but is there to show you how specific tasks may be performed). It also contains the basic set-up and some test code to provide you with the structure (but not the implementation) to implement requirement section 4 (7 segment display) without having to do this from scratch.

2 First task: download the template, compile and run.

These instructions will get you started. They will involve downloading and installing the kernel in your environment, downloading the template, compiling the firmware and running it on the development boards. You will not actually have to do any programming to complete this task. If successful, once the code has been compiled, the binary program has been downloaded to and is running on the ATMega328P microcontroller on the development board, you will see the LED flash on the development board on and off at approximately 3/4 of a second.

2.1 Download and install the kernel

Download the kernel .zip file from Blackboard ('Firmware templates'->'Kernel library').

Do not unzip this file at this stage, but place it in a suitable location where it can be accessed.

Start up the Arduino environment and you will see a window something like Figure 1. Then select 'Sketch'->'Include Library'->'Add .ZIP library' (Figure 2).

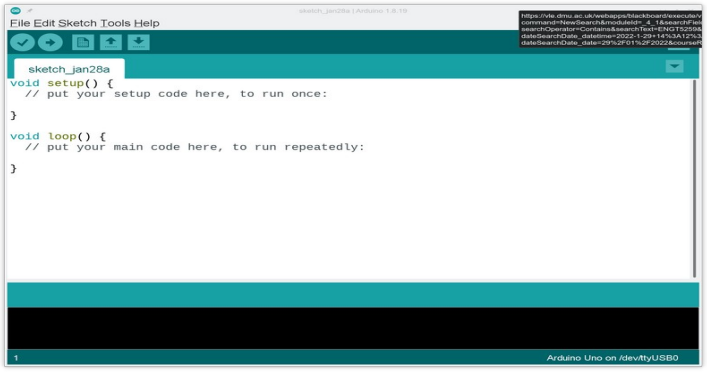


Figure 1: 'Arduino' environment at startup

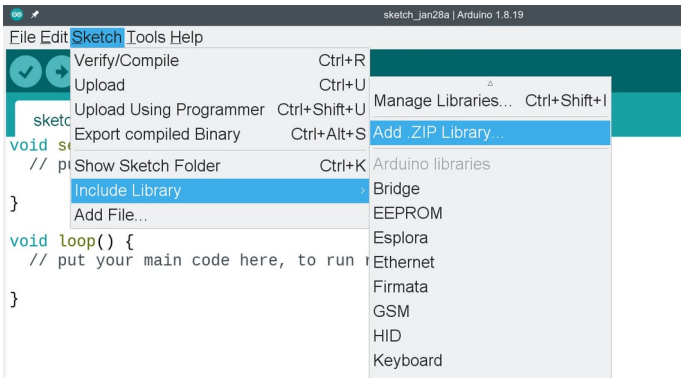


Figure 2: Add .ZIP library

A file select dialog will appear. Navigate to the 'Kernel.zip' file you downloaded from Blackboard and select it. The Arduino environment then should add the library to the list of libraries (the meaning of 'library' in this context will become clear as we go through the module).

2.2 Download and open the template

Download the '7SegTemplate-ArduinoEnv.zip file from Blackboard ('Firmware templates'->'7SegTemplate-ArduinoEnv').

Place this file in a suitable location, and expand the .zip file. It will expand into a subdirectory called 'SevenSeg'. In the Arduino environment, navigate to this folder and open the 'SevenSeg.ino' file.

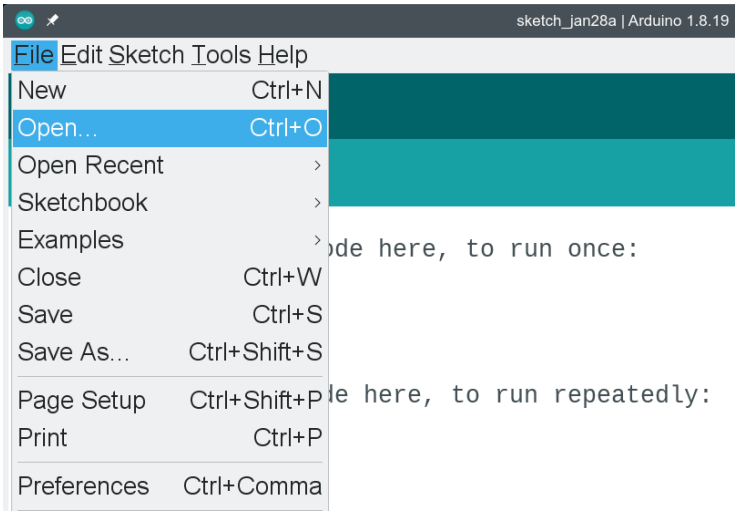


Figure 3: Opening the template code

2.3 Compile and run the template

Ensure the ATMega328 development board is connected to the host system and is powered. Click on the 'right arrow' icon in the environment (Figure 4). This will compile the C/C++ code contained in the several files associated with the firmware, and if there are no errors (there should not be at this stage), will transfer the binary firmware to the flash memory on the ATMega328P microcontroller on the development board.

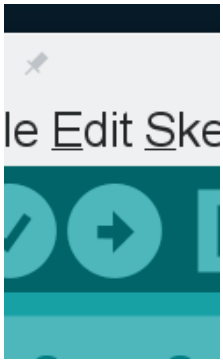


Figure 4: Upload button

Once this process is complete, observe the LED on the development board. It should be flashing at a rate of approximately 1.25 times per second.

If any errors have been observed at this point, please consult your lab tutor.

2.4 Observations and explanations

Look carefully at the code. Start with the file SevenSeg.ino. When the microcontroller is first programmed with your firmware, or if first switched on with your firmware downloaded (remember the firmware is stored in nonvolatile flash memory on the microcontroller), then some initialization tasks are carried out before the function 'UserInit' is called. This function forms part of the firmware you will write. In the case of this code, it calls three functions.

Locate the files that contain these functions. Note that there are two files associated with each: a 'header file' (with the extension .h), and an implementation file (with the extension .cpp). This is an example of 'modularity' and 'encapsulation'. The functions for a specific task are grouped together in a unit called a 'module'. The header file contains only declarations - in other words it alone does not compile to actual binary code. Its purpose is to define the structure of the module and define the functions and variables that can be accessed from outside the module. The implementation - the part that actually compiles to code - may be found in the associated .cpp file.

The UserInit function calls three initialization functions, then exits. The first of these is LEDInitializeDriver. We can see that this is declared in 'leddriver.h' and implemented in 'leddriver.cpp'. We can also see that the function LEDInitializeDriver only does two things: it first arranges the digital IO peripheral on the ATMega328, specifically Port B, to have Port B bit 5 as an output. This is the physical port with a pin connected to the onboard LED (see the schematic diagram) and allows us to switch the LED on and off by writing a bit to the Port B data register. The second thing is that the function calls Kernel::OS.MessageQueue.Subscribe() to register a message handler with the operating system's message queue. This allows other parts of the code to send messages to the message queue with the specified ID. It then exits.

When a message with the ID of MSG_ID_CHANGE_LED (see the header file 'common.h') is posted by another function, the function LEDControlMessageHandler is called, with an argument (parameter) of 'context'. Now although this parameter is declared as a void pointer, such a variable is of 16 bits in length (in the ATMega328 architecture) and there is nothing to say it actually has to *be* a pointer. In this case we are using this variable to contain a simple boolean variable. If it is nonzero, the LED is turned on (see the body of the function). If it is zero, the LED is turned off. Thereby, another function elsewhere in the code can control the LED by simply posting a message! It does not need to re-implement the actual port manipulation (which is encapsulated - in other words 'can not be seen') by code outside this module. The message queue, and the use of modularity and encapsulation is a very common way of avoiding programming errors in embedded systems where otherwise the same peripheral would be accessed at multiple places in the code.

So where do the messages originate from? Look at control.h and control.cpp. In the function CONTROLInitialize, we do two things. Firstly, we create a structure containing two pointers to timer objects. Creating this structure (line 55) doesn't actually create the timer objects - we need to do this separately (line 56 and 57). In line 56 and 57, two new timer objects are created and the pointers to them are placed in the convenient structure we created earlier.

The last thing the CONTROLInitialize() function does is to register a task handler (line 67) and pass a pointer to the structure containing the timer pointers to the task handler. This is a bit different to the message queue. The operating system implements a 'round-robin' scheduler. The user (i.e. your firmware) may register one or more 'tasks' (functions) which will be called sequentially. When all functions are called, the operating system starts at the beginning again and calls all the registered task functions again, sequentially. Thus this is known as a 'round-robin' scheduler.

Therefore, the function ControlTask will be called repeatedly. In this task, we have two conditional 'if' statements that check for the expiry of the timers. Note that the timers in and of themselves do not 'do' anything unless you, as the user of the operating system services, check for their expiry. Since we can not guarantee exactly when ControlTask() will be executed (run), we can not state that the timers will *accurately* expire after the timeout period specified. All we can say is that the timers will expire at or some short time after the specified timeout has elapsed. They thus can not be used where accurate timing is required - they are not 'real time'. This is a subject of a later lecture. However, for this purpose they are perfectly satisfactory.

Just to note, line 91 simply recovers the pointer to the structure containing the timers from the 'context' parameter so we can access it.

We will consider line 93 and the block within it here. This block is very simple. If the timer has expired, the 'if' statement will have a true condition and the block will be executed. In this, the static variable 'ledstate' will be set to zero if it is 1, or 1 if it is zero. This means that 'ledstate' will change from a 0 to a 1 or a 1 to a zero each time the timer expires. Then, in line 114, we call Kernel::OS.Messagequeue.Post() to post a message to the message queue, the 'context' of which is simply the value of 'ledstate'. As 'ledstate' is declared 'static', it is stored on the heap rather than the stack, so it retains its value between calls to ControlTask.

Finally, in line 121, we simply reset the timer, so it once again times out at 750ms.

Thus, the ControlTask simply posts a message every 750ms, alternately with the value of 0 or 1 in its 'context'. This is then subsequently picked up by LEDControlMessageHandler, which does the 'donkey work' of actually changing the LED state!

You can experiment with this. For example, try changing the timeout and see what happens?

This is a very simplistic example that uses almost all the functions of our rudimentary operating system and yet fits in a very low resource microcontroller (remember, this little chip has only 2K of RAM so we have to be mindful of how we use storage when programming in embedded environments. (Yes, that's 'K', not megabytes or gigabytes!). Some embedded microcontrollers have only 128 bytes of RAM!

2.4.1 Encapsulation

The style of the code may seem complex to you on first glance. After all, we could have written all of this in a single .cpp file. So why not? Well the answer lies in the principle of encapsulation. This is good programming practice.

In the above example, the code that actually drives the LED is placed in a separate .h/.cpp combination (leddriver.h/leddriver.cpp). The leddriver.h file contains nothing that will compile to actual code, but contains only the declarations of one function LEDInitializeDriver(). This function is required to be called at initialization, so it is 'exported'

from the module so that our system initialization module. The implementation file 'leddriver.cpp' contains the actual code to respond to the messages.

Our control process, in control.cpp, communicates with the LED driver via messages, posted via a message queue. Other processes can also send messages to the LED driver. This means that the control process (or any other process that uses Kernel::OS.MessageQueue.Post() to send a message with the message ID MSG_ID_CHANGE_LED) does not need to know the technical details of how to manipulate the ports to turn the LED on and off. The subsystem that deals with the manipulation of the interface directly with the hardware can be implemented and tested independently from the rest of the specification.

If you look at the specification, there are a number of requirements to implement. Most of these can be implemented in their own module, using the message queue and/or shared memory to communicate. The control.cpp module I have provided does not implement section 8 of the specification at present. What it is doing is providing a test harness - a function that allows each subsystem to be exercised in order to test that it meets the requirements of the specification.

Therefore the project as a whole can be built up in independent stages, with each stage being tested independently. This maximizes the likelihood of error - or worse, of nothing at all working and the system becoming very difficult to debug.

3 The kernel

A minimal low-footprint 'operating system' is provided. This is documented elsewhere and provides memory management, message queues and a round-robin task manager: common services provided by most embedded operating systems. It is there to help you, and the source code is provided. (You can unzip the kernel .zip file that you downloaded from Blackboard into a directory unconnected from your project, from which you can see the source code for the kernel). This should be made use of in your final design. Source code to the kernel is available and can be downloaded from Blackboard, along with a template project to get you started.

Note that all objects within the operating system are implemented within the 'Kernel' namespace. This means that when accessing them, the prefix Kernel:: should be used with all entities within this namespace.

In this exercise, as well as the demo above, you will use the static objects Kernel::OS.TaskManager and Kernel::OS.MessageQueue, along with creating and using objects of type Kernel::OSTimer

4 The next task: implementing the 7 segment display

Your next task is to implement the seven segment display, according to section 4 of the specification. There is a framework already present in the demo code to help get you started.

4.1 7 segment test code and requirements

Look at lines 127-178 of control.cpp. These use the message queue to send a different message, with a different ID of MSG_ID_CHANGE_7SEG as defined in the header file common.h. Note that the parameter of this message differs from that used in the LED test code earlier. An incrementing integer is sent. Since the specification calls for the display to be able to display the numerals 0-9, and the two special letters 'E' and 'b', there can be a simple mapping where the bits of the 16-bit integer passed with the message are as follows:

Bits	15-5	4	3	2	1	0
Value	0	DP	Value displayed			

where bit 0 is the least significant bit.

Bit 4 represents the state of the decimal point. If a '1', the decimal point is illuminated. If a '0', the decimal point is clear (not illuminated).

The decimal range 0-31 can be encoded in the five bits from 0-5. Each value represents a digit on the display as follows:

Value	Display required
0	'0'
1	'1'
2	'2'
3	'3'
4	'4'
5	'5'
6	'6'
7	'7'
8	'8'
9	'9'
10	'b'
11	'E'
12	blank
13	blank
14	blank
15	blank

In addition, if any of the bits 15 down to 5 are set to anything other than zero, the display should be blank (but the decimal point will still be set according to the state of bit 4).

The code in control.cpp starting at line 127 sends the value of the variabl;e 'v' to the message queue with the ID of MSG_ID_CHANGE_7SEG once the separate timer 'SevenSegTimer' in TIMERSTRUCT expires. It then increments the value of 'v'. If the value of 'v', after increment, is greater than 0x1b (or 27 decimal), then the value of 'v' is reset to zero. This allows for testing of all values in the table above. Then, finally, the timer is reset to 300ms before the task function exits.

4.2 Implementation

This is where your task lies. Have a look at ssegdriver.cpp. In here, we have already created a function SSEGInitializeDriver in which we have assigned three different port pins to outputs. More on this later. Then we subscribe the function SSEGControlMessageHandler() to the message queue so it can receive the posted messages from the control task.

Your code needs to go in the function SSEGControlMessageHandler. This code must, for each call to SSEGControlMessageHandler when a message has been posted:

- interpret the value passed to SSEGControlMessageHandler() according to the mapping in section 4.1 above. I have already cast out the void pointer to an integer called 'value' which you can use.
- manipulate the port pins on the microcontroller in such a manner as to pass the correct signals to the hardware to display the desired
- the code in the function SSEGControlMessageHandler() must NOT block, delay or otherwise take excessive time to execute. If it does, it will hold up other tasks in the system. Remember we are cooperatively multitasking, not pre-emptively multitasking. The function must cooperate!

4.2.1 Hardware

To do this, you will need to become conversant with the actual hardware that is connected to the microcontroller; to do this you will need to look at the schematic diagram for the development board. This is available on the Blackboard shell. The first step here is to look for the seven segment display:

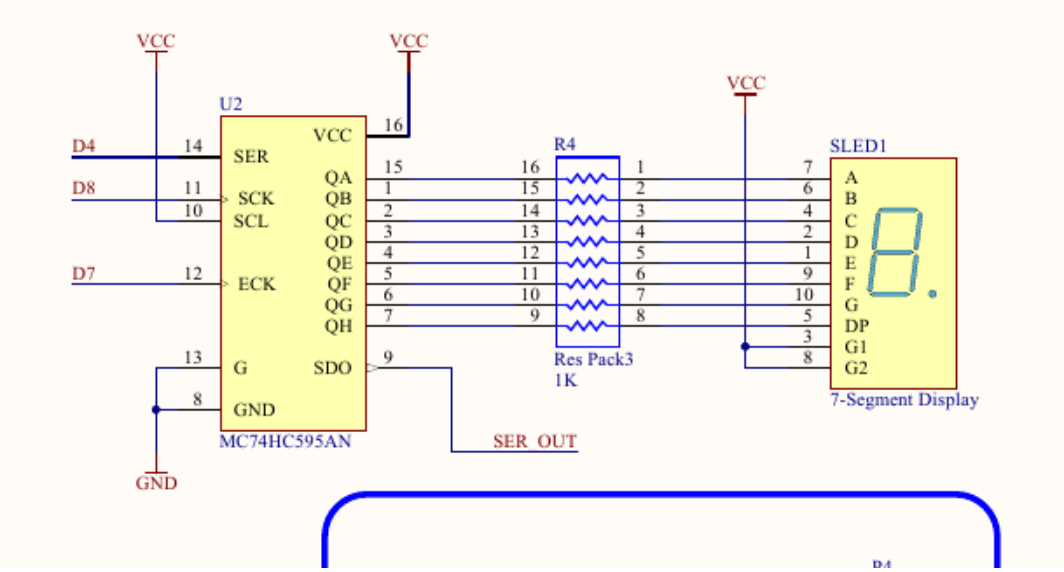


Figure 5: Portion of schematic showing seven segment display

You can see that the seven segment display is:

- common anode (that means the segment pin associated with each segment must be taken to ground to light the segment - in other words it is active low).
- connected, via current limiting resistors, to an integrated circuit 74HC595. This is a shift register, or a serial to parallel converter. The inputs to this integrated circuit are connected to pins D4, D8 and D7
- the shift register is clearly 8 bits wide.
- connected so that the segments are in descending order from the most significant bit. This means that segment A corresponds to the most significant bit, while the decimal point corresponds to the least significant bit.

So, some points to ponder:

- do the outputs of the 74HC595 need to go high or low to light a segment?
- what output would you need to display the value '0', for example?
  - start by thinking which segments needs to be lit
  - then translate that to outputs of the 74HC595 - which lines need to be low and which ones high?
  - so what 8-bit value needs to be loaded by your firmware into the 74HC595 in order to display this value?

To determine where pins D4, D8 and D7 go, you need to scrutinize the schematic and you can see they are connected to pins of the same name on the daughterboard (Figure 6):

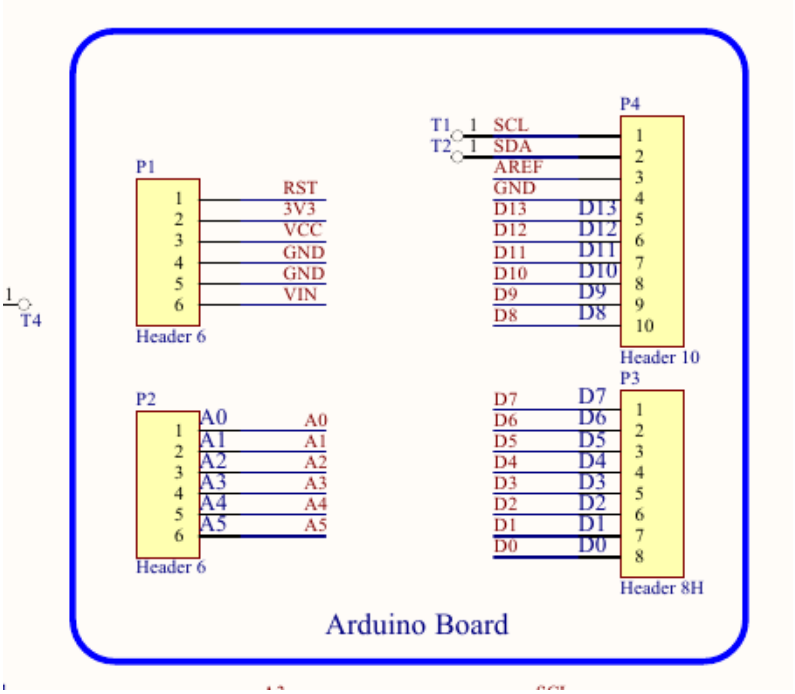


Figure 6: Portion of schematic showing connections to the daughterboard

Observation of the daughterboard on the physical development board shows that the daughterboard is an 'Arduino Uno' - a common microprocessor development platform.

However, we need to know exactly what pins on the microprocessor integrated circuit itself the pins D4, D8 and D7 are connected to. To do this, we need to do some research.

Have a look at https://docs.arduino.cc/hacking/hardware/PinMapping168 - this link will take you to the official pin mapping (and some other information besides). The link talks about the ATMega168, however the ATMega328 we are using has similar pinning.

We can see from the link that the pins D4, D8 and D7 are connected to pins labelled PD4, PB0 and PD7. These are connected to the lines on the 74HC595 as follows:

- PD4: 'SER' (or DATA)
- PB0: 'SCK' (or CLOCK)
- PD7: 'ECK' (or LATCH)

By consulting the ATMega328P datasheet, we can see that:

- PD4 is Port D bit 4
- PB0 is Port B bit 0
- PD7 is Port D bit 7

The peripheral associated with the digital IO ports on the ATMega328 has two registers that allow us to control the port pins: DDRx and PORTx (where 'x' is the port letter). DDRx is the Data Direction Register: set the corresponding bit to a 1 and the port pin becomes an output; set it to '0' and it becomes an input. It is a '0' by default hence lines 39 and 40 in ssegdriver.cpp set the relevant pins to outputs. PORTx is the value of the port - set a bit to '1' and the corresponding output pin on the microcontroller is driven high. Set it to '0' and the pin will be driven low.

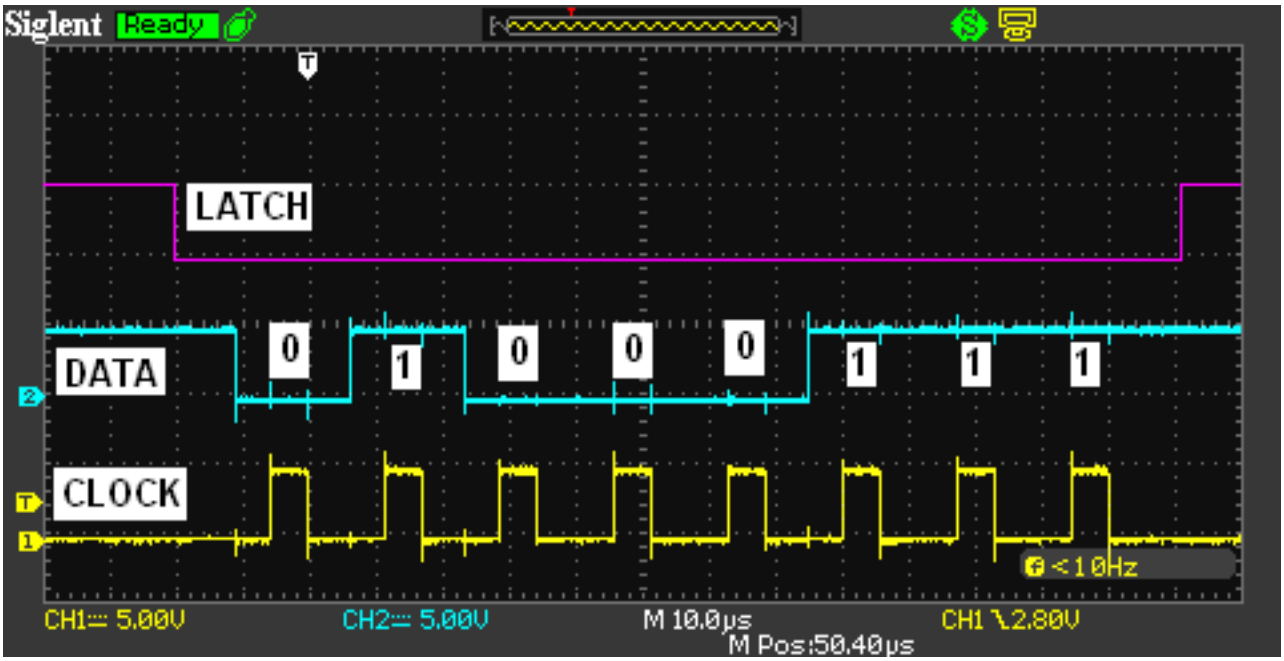
So how do we drive these pins to send in data to the 74HC595? We need to refer to the datasheet for the 74HC595. From this we can deduce how we may be able to write data to the shift register.

4.2.2 Driving the 74HC595

The following procedure is needed to write an 8-bit quantity held in a hypothetical variable called `value` to the shift-register:-

- Pull the LATCH line LOW
- Declare an 8-bit 'mask' variable, and set the most significant bit high.
  i.e. set `mask` to 0x80 (or 128 in decimal, or B10000000 in binary)
- For each of the 8 bits in `value`
  - If `value` bitwise-ANDed with `mask` is non-zero then
    - Pull the DATA line HIGH
  - Otherwise
    - Pull the DATA line LOW
  - Pull the CLOCK line HIGH
  - Either left-shift `value` 1 place, **or** right-shift `mask` 1 place
  - Pull the CLOCK line LOW
- Once all 8-bits are clocked, pull the LATCH line HIGH

The following oscilloscope trace shows how the `value` 71 (i.e. 0x47 in hexadecimal, or 01000111 in binary) is clocked MSB into the shift-register. It is worthwhile spending some time studying this in conjunction with the method described above.



4.2.3 Why use the 74HC595

You may be wondering: why do we use the 74HC595? This seems to add complexity, after all wouldn't it be simpler to just connect the display directly, through current limiting resistors, to the microcontroller pins and drive it directly? Well - this could be done, but it would use eight microcontroller pins instead of three. This means we would have fewer pins available to use elsewhere. This would mean a complex system would either need a more expensive microcontroller with more pins, or would limit the number of external peripherals we could use for a given microcontroller.

4.3 The task

Work out what you need to do to drive the appropriate port pins in accordance with the information in section 4.2.2. Then to start with, try and display something on the display. Once you have worked out how this could be achieved, the last part of the exercise is to interpret the value of 'value' on line 68 correctly, so when SSEGControlMessageHandler is called, the correct display is made.

And then you have already completed and implemented requirement 4 of the specification!