ENGD2103 EMBEDDED SYSTEMS FUNDAMENTALS
Laboratory guidance notes
Week 7: Summative Assignment – Part 1 – Bare Metal and Traffic Lights Sequencer

Author: M A Oliver, J A Gow, T Allidina *v1.1*

Table of Contents

1 Aims and objectives....1
2 Overview....2
3 Task Requirements....2
4 Bare-Metal Port Access....2
5 Converting your HAL code to bare metal....5
6 Further Bare Metal and Lights Sequence....6
7 Traffic Lights Sequencer....8
8 Using the Serial Port for Debugging....10

Version record

Version	Notes	Date	By
1.0	Initial version	13/10/22	MAO
1.1	Correction of minor typo	17/11/22	MAO

PLEASE READ THIS CAREFULLY

This is the first of a series of guidance documents to get you started with your Embedded Systems Fundamentals assignment. They are aligned with the key aspects of the summative 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.

1 Aims and objectives

During the summative assignment, you will be creating a single project capable of performing multiple tasks concurrently. This document outlines the first part: bare metal and traffic lights.

Prerequisite: you should have constructed the breadboard layout and completed all the formative assignment tasks (including the HAL) before commencing this task.

2 Overview

The purpose of this task is to give you experience of:-

- Porting Arduino I/O functions to bare-metal.
- Implementing a traffic light sequencer as a finite state machine (FSM).

3 Task Requirements

In this task, you will be required to convert all your Arduino-specific I/O code to bare metal; this should be a straightforward exercise in modifying your HAL.

You will also be required to create a traffic lights sequencer as a code module capable of concurrent operation.

4 Bare-Metal Port Access

Introduction

So far, you will be familiar with:-

- Using `pinMode()` to set-up an I/O pin as an input or output (in other words, setting-up the data direction).
- Using `digitalWrite()` to set the state of a single output pin to HIGH or LOW.
- Using `digitalRead()` to read the state of a single input pin.

These are fairly high-level techniques for respectively setting data direction, writing outputs and reading inputs and provide an abstract encapsulation of the low-level accesses, also known as register-level accesses, to the programmer. In this exercise you will gain first-hand experience of reading from and writing to registers inside the ATMEGA328P microcontroller to control the digital I/O.

Why take a Bare-Metal Approach?

The `pinMode()`, `digitalWrite()` and `digitalRead()` functions are a feature of the Arduino IDE. Microcontrollers that are supported by the Arduino IDE can use these functions.

In industry however, you are more likely going to encounter one of the myriad of microcontrollers that are not supported by the Arduino IDE. By learning and understanding the principles of bare-metal control of I/O ports, you will be able to apply that knowledge to control the I/O ports of any microcontroller.

There are advantages of using bare-metal port-access:-

- Direct register access is faster than using `digitalWrite()` or `digitalRead()`
- With direct register access, multiple outputs can be written simultaneously, or multiple inputs can be read simultaneously.

Port Registers on the Arduino Uno

The ATmega328P on the Arduino Uno has three I/O ports:-

- PORTB
- PORTC
- PORTD

PORTB

PORTB maps to the Arduino as follows:-

- Bit 0 connects to Digital Pin 8
- Bit 1 connects to Digital Pin 9
- Bit 2 connects to Digital Pin 10
- Bit 3 connects to Digital Pin 11
- Bit 4 connects to Digital Pin 12
- Bit 5 connects to Digital Pin 13
- (Bits 6-7 connect to the crystal.)

It contains three registers:-

`DDRB` – The Port B Data Direction Register (read/write)
`PORTB` – The Port B Data Register (read/write)
`PINB` – The Port B Input Register (read/write)

PORTC

PORTC maps to the Arduino as follows:-

- Bit 0 connects to Analog Pin 0
- Bit 1 connects to Analog Pin 1
- Bit 2 connects to Analog Pin 2
- Bit 3 connects to Analog Pin 3
- Bit 4 connects to Analog Pin 4
- Bit 5 connects to Analog Pin 5
- (Bits 6-7 are not brought out.)

It contains three registers:-

`DDRC` – The Port C Data Direction Register (read/write)
`PORTC` – The Port C Data Register (read/write)
`PINC` – The Port C Input Register (read/write)

PORTD

PORTD maps to the Arduino as follows:-

- Bit 0 connects to Digital Pin 0 (avoid this as it is connected to the UART)
- Bit 1 connects to Digital Pin 1 (avoid this as it is connected to the UART)
- Bit 2 connects to Digital Pin 2
- Bit 3 connects to Digital Pin 3
- Bit 4 connects to Digital Pin 4
- Bit 5 connects to Digital Pin 5
- Bit 6 connects to Digital Pin 6
- Bit 7 connects to Digital Pin 7

It contains three registers:-

`DDRD` – The Port D Data Direction Register (read/write)
`PORTD` – The Port D Data Register (read/write)
`PIND` – The Port D Input Register (read/write)

The DDRx registers determine which pins are inputs and which pins are outputs. If a bit in the DDRx register is set to 1 then its corresponding pin is an output. Conversely, if a bit in the DDRx register is set to 0 then its corresponding pin is an input.

The PINx registers read the states of the pins that have been configured as inputs.

For pins configured as outputs, the PORTx registers control the states of the outputs. If a bit in the PORTx register is 1, then its corresponding output is HIGH. If a bit in the PORTx register is 0, then its corresponding output is LOW.

For pins configured as inputs, the PORTx registers determine whether an internal pull-up resistor is applied. If a bit in the PORTx register is 1, then a corresponding pull-up resistor is applied. If a bit in the PORTx register is 0, then no corresponding pull-up resistor is applied.

At this stage, it would be a worthwhile exercise reviewing Section 2.9 "Bitwise Operators" in "The C Programming Language - 2nd Edition" by Kernighan & Ritchie.

Example 1

Consider the following simple example. The Arduino Nano has an internal LED on Digital Pin 13. Suppose a pushbutton switch is connected between Pin A0 and GND. The LED should illuminate when the switch is depressed, and should go off when the switch is released.

So the LED on Digital Pin 13 resides on PORTB, Bit 5.
The Switch on Pin A0 resides on PORTC, Bit 0.

```
////////////////////////////////////////////////////////
// Simple Example demonstrating how to perform simple
// bare-metal port access.
////////////////////////////////////////////////////////

// The internal LED resides on Digital Pin 13 (i.e. PORTB, bit 5)
// Macros LED_ON and LED_OFF are self-explanatory
#define LED_BIT         B00100000
#define LED_ON          PORTB |= LED_BIT
#define LED_OFF         PORTB &= ~LED_BIT

// The pushbutton switch resides on Digital Pin 8 (i.e. PORT B, bit 0)
// The macro SWITCH_PRESSED gives 1 when the switch is pressed
// The macro SWITCH_RELEASED gives 1 when the switch is released
#define SWITCH_BIT      B00000001
#define SWITCH_PRESSED  !(PINC & SWITCH_BIT)
#define SWITCH_RELEASED !SWITCH_PRESSED


void setup()
```

```
{
    // Set-up PORTB, bit 0 for switch input
    DDRC &= ~SWITCH_BIT;
    PORTC |= SWITCH_BIT;     // Apply an internal pull-up

    // Set-up PORTB, bit 5 for the internal LED
    DDRB |= LED_BIT;
}

void loop()
{
    if (SWITCH_PRESSED)
    {
        LED_ON;
    }
    else
    {
        LED_OFF;
    }
}
```

5 Converting your HAL code to bare metal

You should have written a Hardware Abstraction Layer (HAL) for your formative assignment. If so, you should have definitions for setting each I/O pin high or low (using `digitalWrite()`.

Using the techniques shown in Example 1, try converting your `digitalWrite()` calls to their more efficient bare metal counterparts in your HAL module. Be careful when mapping the Arduino digital pins to their corresponding ATmega328P ports and bits.

Firstly, concentrate on the traffic lights definitions. Once you have performed the mapping of the `digitalWrite()` calls to bare-metal within your HAL module, try running your traffic lights code from the formative assignment. You will be able to use your form1.ino code from the formative assessment as a test harness. If the traffic lights code worked in the formative assignment, the changes you make to the HAL should still result in correct operation; if they don't, you will need to debug the HAL only.

Once this works, don't forget to also convert your `pinMode()` calls to bare-metal.

Next, concentrate on the definitions from the button counter. These will be the `digitalWrite()` calls associated with the CLOCK, DATA and LATCH lines. You will then need to deal with the `digitalRead()` call associated with SW1; Example 1 shows how this can be done and this should be a straightforward change. This technique can now be extended so you can produce the definitions for reading the second switch SW2. Once these definitions have been ported to bare metal, try running your button counter code from the formative assignment. You will be able to use your form2.ino code from the formative assessment as a test harness. If this worked in the formative test, the changes you make to the HAL should still result in correct operation; if they don't, you will need to debug the HAL only.

Once this works, don't forget to convert your `pinMode()` calls to bare-metal. These will include the DATA, CLOCK and LATCH lines as well as the two switches.

Once this task is complete, you should now have a HAL module that will support the fundamental digital I/O requirements for the summative assignment.

6 Further Bare Metal and Lights Sequence

Consider the following example which shows how 6 LEDs can be driven in a single write operation as part of a very simple lights scheduler. The LEDs reside on Arduino Digital Pins 7-2, i.e. Port D bits 7-2 respectively.

This uses the unstructured approach for concurrency to produce a stand-alone demo.

In the main `loop()` function, the code module for sequencing the lights is continually executed. For most of the time, it will be performing timing operations. However, the finite state machine will be run every time the required time delay has elapsed.

```
////////////////////////////////////////////////////////
// Example demonstrating how to drive multiple output
// lines simultaneously
////////////////////////////////////////////////////////

// The LEDs are located on Port D.
// LED1 is connected to Port D Bit 7 (Digital Pin 7)
// LED2 is connected to Port D Bit 6 (Digital Pin 6)
// LED3 is connected to Port D Bit 5 (Digital Pin 5)
// LED4 is connected to Port D Bit 4 (Digital Pin 4)
// LED5 is connected to Port D Bit 3 (Digital Pin 3)
// LED6 is connected to Port D Bit 2 (Digital Pin 2)
#define LEDS_OFF  B00000000
#define LED_1     B10000000
#define LED_2     B01000000
#define LED_3     B00100000
#define LED_4     B00010000
#define LED_5     B00001000
#define LED_6     B00000100


bool init_module0_clock;


void setup() {
  DDRD |= 0b11111100;           // Pins PD7-PD2 as outputs
  init_module0_clock = true;    // Set Module 0 running…..
}

void loop() {
  // put your main code here, to run repeatedly:
```

```
  { // Set Module 0
    static unsigned long module_time, module_delay;
    static bool module_doStep;
    static unsigned char state; // state variable module 0

    if (init_module0_clock) {
      module_delay = 50;
      module_time = millis();
      module_doStep = false;
      init_module0_clock = false;
      state = 0;
    }
    else {
      unsigned long m = millis();
      if ( (m - module_time) > module_delay ) {
        module_time = m;
        module_doStep = true;
      }
      else module_doStep = false;
    }

    if (module_doStep)
    {
      // FSM for module 0.
      switch (state)
      {
        case 0:
          PORTD = LED_1 | LED_6;
          module_delay = 1600;
          state = 1;
          break;

        case 1:
          PORTD = LED_2 | LED_5;
          module_delay = 1200;
          state = 2;
          break;

        case 2:
          PORTD = LED_3 | LED_4;
          module_delay = 800;
          state = 3;
          break;

        case 3:
          PORTD = LEDS_OFF;
          module_delay = 400;
          state = 0;
          break;

        default:
          state = 0;
      }
    }
  }
}
```

There are four states in this Finite State Machine. If you study the code:-

State 0: LEDs 1 and 6 are illuminated (all other LEDs are off)
The module delay is set to 1.6 seconds (i.e. State 0 runs for 1.6s)
The next state will be State 1.

State 1: LEDs 2 and 4 are illuminated (all other LEDs are off)
The module delay is set to 1.2 seconds (i.e. State 1 runs for 1.2s)
The next state will be State 2.

State 2: LEDs 3 and 4 are illuminated (all other LEDs are off)
The module delay is set to 0.8 seconds (i.e. State 2 runs for 0.8s)
The next state will be State 3.

State 3: All LEDs are extinguished
The module delay is set to 0.4 seconds (i.e. State 0 runs for 0.4s)
The next state will be State 0.

You can see how it is possible to drive all 6 LEDs using a single write command. This is efficient, but it does have a disadvantage that it overwrites pins PD1 and PD0 (associated with the serial port).

Also, in this example, you will be able to see how you can change the module delay between states.

7 Traffic Lights Sequencer

Having worked your way through the examples, you should now be able to apply your knowledge to implement a traffic lights sequencer capable of concurrent operation. At this stage concentrate on the “Equal Priority” mode.

Example 2 presents a rather unstructured approach. It is expected that you will improve the structure by using the “`Concurrent`” code module that has been covered in lectures and provided on Blackboard, and use some degree of encapsulation to modularize your project. Please refer to Lectures in Weeks 5 and 7.

As for driving the LEDs: you can either use your existing HAL code to drive the individual LEDs, or drive all 6 LEDs in a single write. Choose which you feel most comfortable with. However `digitalWrite()` and `pinMode()` must be avoided.

The required sequencing for the traffic lights is shown in the table below. The sequence is slightly more involved than the traffic lights controller encountered in the Formative assignment. Also note the differences in timings. Please note that State 0 is a start-up state and it is only run upon initialization. Once State 8 is complete, the next state needs to be State 1.

STATE	LIGHTS SET 1	LIGHTS SET 2	TIME (s) Equal Priority	TIME (s) SET 1 Priority	TIME (s) SET 2 Priority
0			2	2	2
1			1	1	1
2			1	1	1
3			**6**	**8**	**4**
4			2	2	2
5			1	1	1
6			1	1	1
7			**6**	**4**	**8**
8			2	2	2

Before embarking on the traffic lights sequencer, it is advised that you start a brand new project. Once you have created the project, close down your Arduino IDE. Now copy your HAL files into the project folder. You might also want to add the Concurrent.h and Concurrent.cpp files as well. When you restart the Arduino IDE, you should now see these additional files. You can now write your traffic lights sequencer module; it is advised to use encapsulation for modularizing and improving the structure of your code.

Once you have completed this task, the rest of the summative assignment involves adding further code modules to the project.

8 Using the Serial Port for Debugging

When debugging FSMs, you might like to temporarily add:-

```
Serial.println(state);
```

just before the `switch` statement in the FSM. By observing this output, you will be able to see whether or not the FSM is running the states in the correct order. If there is an issue, this will provide you with a good point to start targeting your debugging.