NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

LABORATORY MANUAL

CE2107
Microprocessor System Design and Development

Lab Experiment #2

General Purpose Input/Output (GPIO) and Finite State Machine Design

SESSION 2021/2022
SEMESTER 1

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
NANYANG TECHNOLOGICAL UNIVERSITY

This lab session will be done in groups of 2. The group will work together to complete the lab exercises.

1. OBJECTIVES

1.1 Memory Map Registers access in C
1.2 Interface reflectance sensors via GPIO pins
1.3 Design a finite state machine for a line follower robot

2. LABORATORY

This experiment is conducted at the **Hardware Lab 2** at **N4-01b-05 (Tel: 67905036)**.

3. HARDWARE EQUIPMENT

- A Windows-based computer (PC) with a Universal Serial Bus (USB) port.
- Texas Instruments Robotic System Learning Kit (RSLK-MAX)
- A USB A-to-MicroB cable.
- Oscilloscope.

4. ACKNOWLEDGEMENT

This lab reference and leverage from the work done by Dr Jonathan Valvano on the RSLK-MAX. The original source (sans solution) can be downloaded online under filename slac799a.zip. Students can also download the original lab notes (slay052a.pdf) for reference but note that adaptation had been made so there are differences in information, instructions and tasks. The original RSLK Max workshop is also available online at https://university.ti.com/en/faculty/ti-robotics-system-learning-kit/ti-rslk-max-edition-curriculum

5. REFERENCES (Can be found in the Doc sub-folder)

[1] Wk1-5 Lecture Notes

[2] RSLK-MAX Construction Guide (sekp164.pdf)

[3] MSP432 Launchpad UG (slau597f.pdf)

[4] MSP432 TRM (slau356h.pdf)

[5] MSP432 Datasheets (msp432p401r.pdf)

[6] ARM Optimizing Compiler UG (spnu151r.pdf)

[7] ARM Assembly Language Tools UG (spnu118u.pdf)

[8] Cortex M3/M4F Instruction Set (spmu159a.pdf)

6. GENERAL PURPOSE INPUT/OUTPUT (GPIO)

The simplest I/O port on a microcontroller is the parallel port, or general purpose input output (GPIO). A parallel I/O port is a mechanism that allows the software to interact with external devices. It is called parallel because multiple signals can be accessed all at once.

Ports 1–10 are 8 bits wide each, meaning each port module can handle up to 8 pins. However, due to limited number of pins on the physical IC, not every port on the MSP432 LaunchPad has all 8 pins.

To make the microcontroller more marketable, the ports on most microcontrollers can be software-specified to be either inputs or outputs. Microcontrollers use the concept of a direction register to determine whether a pin is an input (direction register bit is 0) or an output (direction register bit is 1).

When the GPIO Port is configured to input mode, it allows software to read external digital signals. For example, a read from Port1 input data register i.e. P1->IN, returns the logic levels of the input signals of all Port 1 pins at that instance. A write to P1->IN register usually produces no effect.

Note that pins on the MSP432 are not 5-V tolerant, meaning the input voltages must be between 0 and 3.3 V.

If a GPIO Port pin is configured to output mode, a write to P1->OUT register will affect the values on the output pins of Port 1. Writing a ‘1’ will cause a Logic ‘1’ to be output from the P1 output pin.

You can refer to your lecture notes for more details on the GPIO Port.

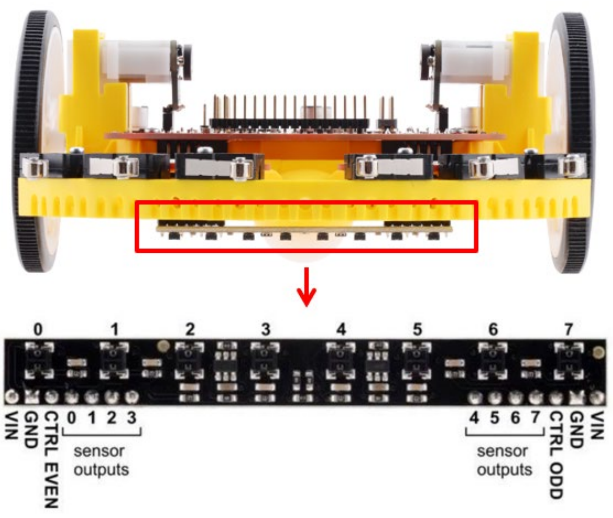


Figure 1: Reflectance Sensors mounted at the bottom of the robot.

6.1 Exploration on GPIO operation

#Task:

- **Explore two projects Lab1ref_InputOutput and to understand how to setup GPIO port pins.**

Essentially, a few key attributes have to be initialized

- Pin mode selection via PSEL0 and PSEL1 register
- I/O pin direction via DIR register
- Pullup/Pulldown resistor mode via REN and OUT register

Take note of the way the registers are initialized. Preferred way is to only modify the bits required while keeping all other bits in the register unchanged. This is done using bitwise logical operation in the sample code.

6.2 Low-level reflectance sensors drivers

In this section, you will explore the characteristics of the reflectance sensors and develop a program that allows a robot to follow a line via its reflectance sensors, as illustrated in Figure 2 below. Robot has eight sensors on a PCB and will try to keep the line in the middle of the sensor array.

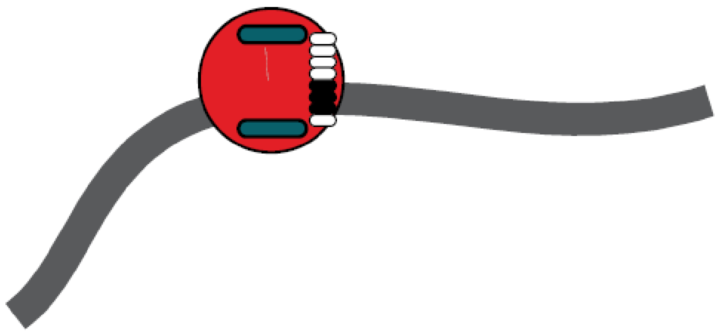


Figure 2: Line Following Robot

Each sensor is binary, returning 1 if it sees black, and 0 if it sees white.

- If the robot is properly positioned on the line, the middle sensors will see the black line.
- If the robot is a little off to the left or right, one or more outer sensors will see the black line.
- If the robot is completely off the line, all sensors will see white.

We will use sensor integration to combine the eight binary readings into a single position parameter. We define position of the robot as the distance from the sensor to the line. The sensor we will use is about 66 mm wide (with about 9.5mm between sensors), so we should be able to estimate the robot position of -33 to +33 mm from the center of the line.

The optimal sensing distance for this sensor strip is 3 mm (0.125"). You will need to fix the distance between the line sensor and the floor to about 3 mm. Fortunately, the board has been mounted properly for you.

The reflectance sensors are connected to Port 7 of the MSP432 and P5.3 and P9.2 pin is used to enable the firing of the sensor, P5.3 enable the even sensors while P9.2 takes care of the odd sensors. Note that MSP432 pins are not 5V tolerant so you need to power the sensors with only 3.3V supply. This is done in the current hardware. See Figure 3 below for details.

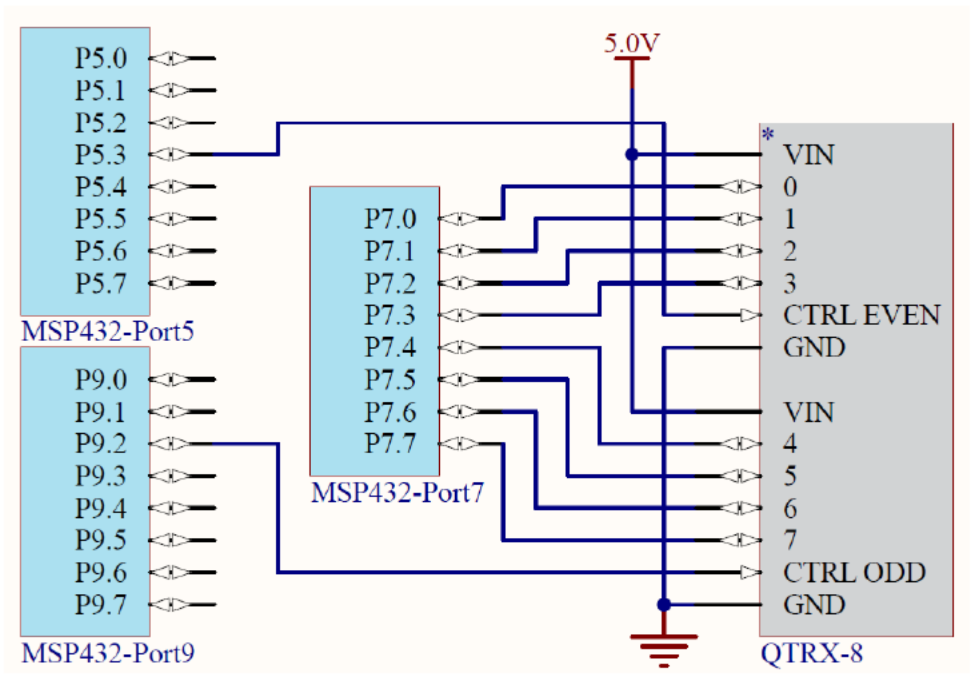


Figure 3: Reflectance Sensor Connection to MSP432

As shown in Figure 4, P5.3 is configured as an output and controls the firing of IR LED (even numbered sensors) in the reflectance sensor array. The reflectance of the surface will affect the "effective resistance" of the transistor which in turn affects the decay rate of the voltage on Port7 pins. A more proper explanation is that the white surface reflects more IR wave to the base of the transistor, causing more current to flow through the collector-emitter and discharge the capacitor at a faster rate.

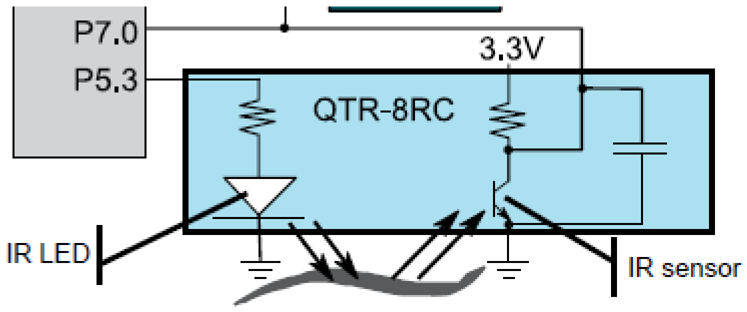


Figure 4: Reflectance Sensor Hardware Internals

Figure 5 illustrate the decay rate when a black or white surface is detected. The black background diagram on the right is the actual scope capture. The voltage on Port 7 pins decay slower when the surface is black, hence a logic ‘1’ will be read after time t_b. By the same concept, a white surface will give a logic ‘0’ after the same time delay t_b.

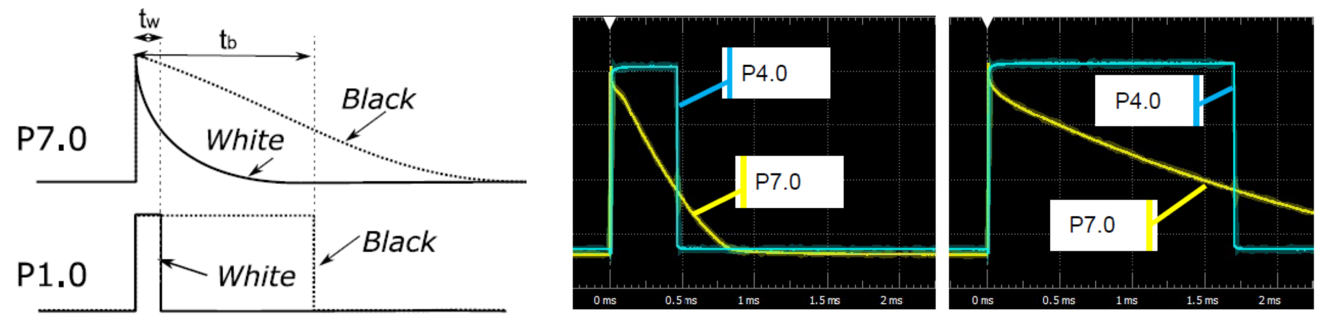


Figure 5: Decay rate of Voltages at GPIOs connected to Reflectance Sensors

Now that you understand how the sensor works, you will create a low-level software driver that measures all eight sensors at the same time. You will need to connect all 8 sensor inputs as shown in Figure 6. You can assume the MSP432 is running with a 48 MHz clock. The initialization includes:

1. Configure P5.3 and P9.2 as output and set them to low initially.
2. Configure P7.7 – P7.0 to input mode.

Create a C function that measures all eight sensors. Let time be a parameter passed into this function, choosing time between *tb* and *tw*. Basically, if we wait 1000 μs, then the capacitor voltage will have decayed to zero if the surface is white. The capacitor voltage will still be at Logic high if the surface is black. This allows us to differentiate between white and black surfaces.

Perform these 8 steps in this sequence:

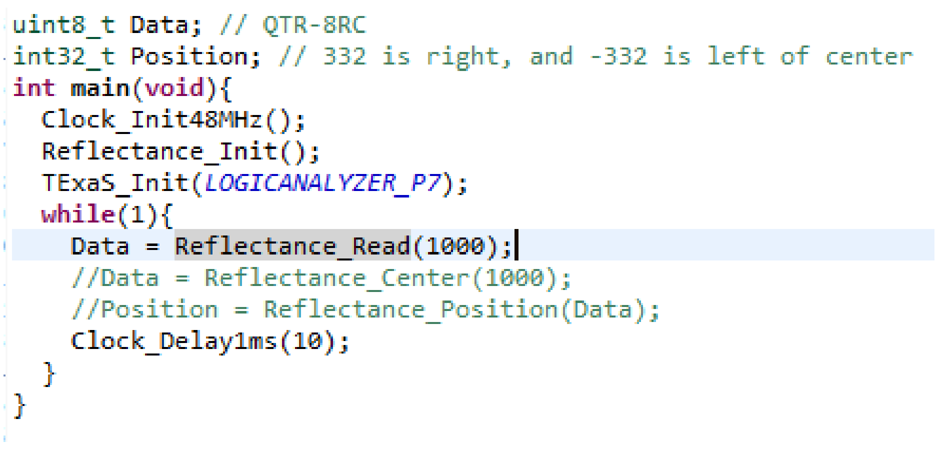
1. Set P5.3 and P9.2 high (turn on IR LED).
2. Configure P7.7 – P7.0 as output pins. Set the pins to Logic ‘1’. This will charge the capacitor in Figure 4.
3. Wait for 10 μs by calling Clock_Delay1us(10);
4. Configure P7.7 – P7.0 as input pins. Capacitor in Figure 4 will start to discharge.
5. Wait for ‘time’ μs by calling Clock_Delay1us(time).
6. Read P7.7 – P7.0 digital inputs.
7. Set P5.3 and P9.2 low. This will turn off the IR LED to save power.
8. Return 8-bit binary value read in step 6. This will show the status of each reflectance sensor.

The delay time t_b is about 1000 µs and is derived by observing the decay rate of the capacitor voltages on the oscilloscope. You are welcomed to check that out. Note however that this delay duration may vary if the hardware environment is different.

#Task:

- Open Lab2_ReflectanceGPIO project and complete the Lab2 related functions in the file reflectance.c.
- Potential Files to modify:
    - Lab2_GPIOmain.c
    - Reflectance.c

Read the comments in the reflectance.c file for details. Upon completion, you can test out your code with the main() in Lab2_GPIOmain.c, as shown below. Add the variable **Data** to the expression window and step through the code to see if the reflectance value changes as surface colour below your robot changes from white to black and vice versa.



```
uint8_t Data; // QTR-8RC
int32_t Position; // 332 is right, and -332 is left of center
int main(void){
  Clock_Init48MHz();
  Reflectance_Init();
  TExaS_Init(LOGICANALYZER_P7);
  while(1){
    Data = Reflectance_Read(1000);
    //Data = Reflectance_Center(1000);
    //Position = Reflectance_Position(Data);
    Clock_Delay1ms(10);
  }
}
```

6.3 High-level reflectance sensor value interpretation

In this section, you will combine the eight binary measurements into a single parameter representing the amount of which the robot is away from the centre of the line. We will assume the sensor S1 (P7.0) is positioned on the robot's right, 33.4 mm from midline. Furthermore, we assume the sensor S8 (P7.7) is positioned on the robot's left, -33.4 mm from midline. As mentioned earlier, another goal is to make this parameter insensitive to angle. If the sensor is operating properly, the 8-bit binary pattern stored in Data falls into four categories:

1. <all 0's> (off the line or on white surface)
2. <some 0's, some 1's>, e.g., 00000111 (off to the left)
3. <some 0's, some 1's, some 0's>, e.g., 00110000 (over the line)
4. < some 1's, some 0's>, e.g., 11110000 (off to the right)

Figure 6 below shows the sensors position from the centre of the Reflectance sensor PCB, with the centre of the robot between sensors 4 and 5 (P7.3 and P7.4)

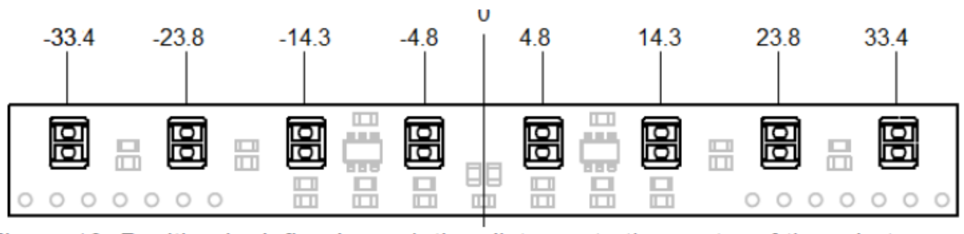


Figure 12. Position is defined as relative distance to the center of the robot.

Figure 6: Reflectance Sensor Board

Define an array W whose elements represent eight distance values in 0.1mm units, Wi for i = 0 to 7. i.e. W = {334, 238, 143, 48, -48, -143, -238, -334}

Define bi to be 0 (white) or 1 (black) for each binary bit returned by the Reflectance_Read() function. One possible sensor integration function is to calculate a weighted average of the eight sensor readings or binary results. Assuming there is at least one black reading, the estimated distance **d** of the black line from the centre of the Robot is given by the following expression.

$$d = \frac{\sum_{i=0}^{7} b_i w_i}{\sum_{i=0}^{7} b_i}$$

This is what the function Reflectance_Position() is supposed to return. A function which you need to code as part of the task for this section.

#Task:

- Complete the function Reflectance_Position() in Reflectance.c
- Compare the actual distance and the estimated distance derived from Reflectance_Position() to calibrate for any offset.
- Potential File to modify: Reflectance.c
- Hint: One way of going about computing the mathematical expression above is to extract individual bit b0, b1, b2 etc from the 8-bit binary value measured. This can be done via shifting and masking appropriate bits. Shifting in C is done via ‘<<’ and ‘>>’. The individual bit (b0, b1, …) is multiplied with the respective weights (w1, w2, …) to derive d.

7. FINITE STATE MACHINE (code integration)

Second section to the lab is a code integration and validation exercise. You will explore a 3-state and 11-state FSM design of a line following robot. Details below.

7.1 Introduction

Software abstraction allows us to define a complex problem with a set of basic abstract principles. We can then construct a system solution using these abstract building blocks. Using the abstraction gives us a better understanding of both the problem and its solution. This is because we can separate what the system does (policies) from the details of how the system works (mechanisms). This separation simplifies the design process by first describing what the system does, and then we can translate the description into a system that implements that description. Abstraction provides for a proof of correct function and simplifies both extensions and customization. The abstraction presented in this section is the Finite State Machine (FSM). The abstract principles of FSM development are the inputs, outputs, states, and state transitions. The FSM state transition graph (STG) defines the time-dependent relationship between its inputs and outputs. If we can take a complex problem and map it into an FSM model, then we can solve it with simple FSM software tools. Our FSM software implementation will be easy to understand, debug, and modify.

7.2 Finite State Machine Implementation

In this section, we will use a finite state machine to create a controller for a simple line following robot. Inputs will come from the center two reflectance sensors you tested in section 6. Since we have not developed the motor code yet, the output of the FSM will go to two LEDs to simulate two motors on the robot. The goal of the controller is to follow the line.

The original Lab2_FSM project implements a 3-state FSM shown in Figure 7 below, which we could use to implement a line-following robot. The 500 is the time to wait in each state in ms. On the real robot, we set these delay times to be much shorter, depending on how fast the mechanical robot responds to actuator commands. However, in this lab, the 500 ms is chosen to make it easy to see the output with our eyes

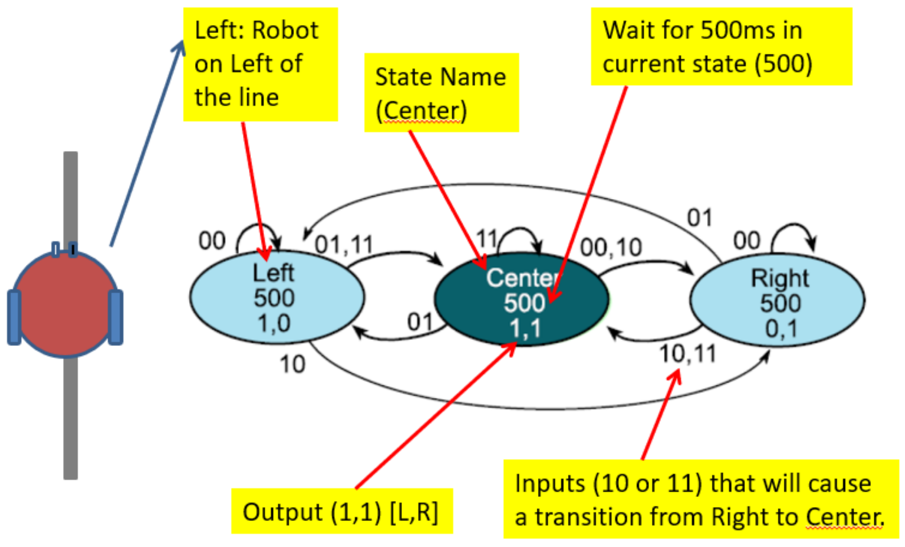

Left: Robot on Left of the line
State Name (Center)
Wait for 500ms in current state (500)
00
01,11
11
00,10
01
00
Left 500 1,0
Center 500 1,1
Right 500 0,1
01
10
10,11
Output (1,1) [L,R]
Inputs (10 or 11) that will cause a transition from Right to Center.

Figure 7: 3-state FSM

The robot has two sensors to detect the line, see Figure 8 below. If the robot is properly positioned on the line, both sensors will read 1. If the robot is a little off to the left or right, one sensor reads 1, and the other sensor reads 0. If the robot is completely off the line, both sensors will read 0.

The robot has two motors, if the software outputs high to both motors, the robot moves forward in a straight line. If the software outputs high to just one motor, it will turn. If the software outputs low to both motors, it will stop.

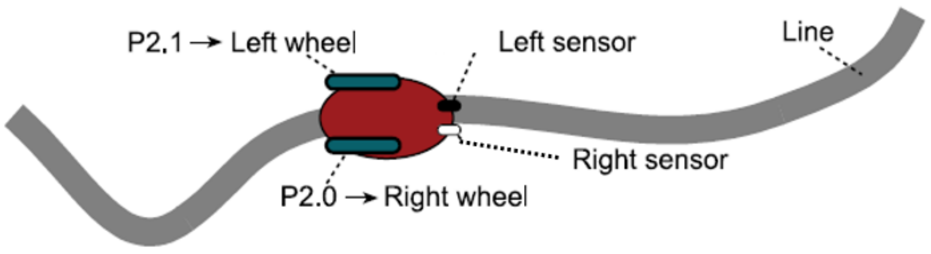


Figure 8: LED emulation of robot

The connection on the MSP432 Launchpad is shown in Figure 9 below. Table1 and Table2 shows the simulation of the SW1/SW2 and LED colours for the line sensors and Robot motor. You can test the behaviour of the 3-stage FSM with the SW1/SW2 and LEDs, before proceeding to integrate the actual 11-stage FSM in section 7.3.

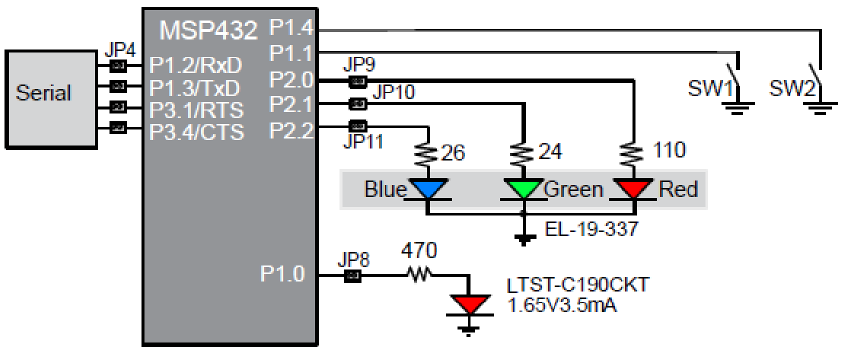


Figure 9: MSP432 Launchpad SW and LED connection

SW2	SW1	LaunchPad_Input	Meaning
pressed	pressed	1,1 = 0x03	On line
pressed	not	1,0 = 0x02	Right of line
not	pressed	0,1 = 0x01	Left of line
not	not	0,0 = 0x00	Off the line

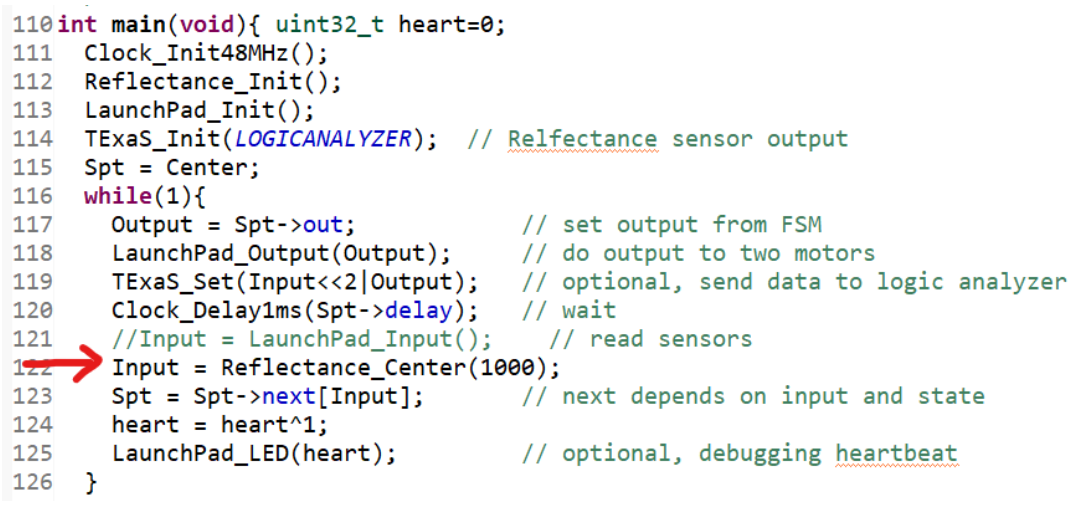
Table 1. Switches simulate line sensors.

P2.1	P2.0	LaunchPad_Output	LED	Meaning
off	off	0,0 = 0x00	black	Stop
off	on	0,1 = 0x01	red	Turn left
on	off	1,0 = 0x02	green	Turn right
on	on	1,1 = 0x03	yellow	Straight

Table 2. LEDs simulate robot motor

7.3 FSM Design

The 3-state FSM design in Figure 7 does not take care of certain scenarios (to be explained during the lab). To fix this issue, a 11-state FSM illustrated in Figure 10 is used instead. The FSM design in Lab2-FSMmain-11states.c solution file in the CCS project is supposed to conform to the FSM design shown in Figure 10. Notice also that the input to the 11-state FSM is taken directly from the Reflectance_Center() function, so make sure the corresponding reflectance sensor data code is in place.



```
int main(void){ uint32_t heart=0;
  Clock_Init48MHz();
  Reflectance_Init();
  LaunchPad_Init();
  TExaS_Init(LOGICANALYZER);  // Relfectance sensor output
  Spt = Center;
  while(1){
    Output = Spt->out;            // set output from FSM
    LaunchPad_Output(Output);     // do output to two motors
    TExaS_Set(Input<<2|Output);   // optional, send data to logic analyzer
    Clock_Delay1ms(Spt->delay);   // wait
    //Input = LaunchPad_Input();    // read sensors
    Input = Reflectance_Center(1000);
    Spt = Spt->next[Input];       // next depends on input and state
    heart = heart^1;
    LaunchPad_LED(heart);         // optional, debugging heartbeat
  }
```

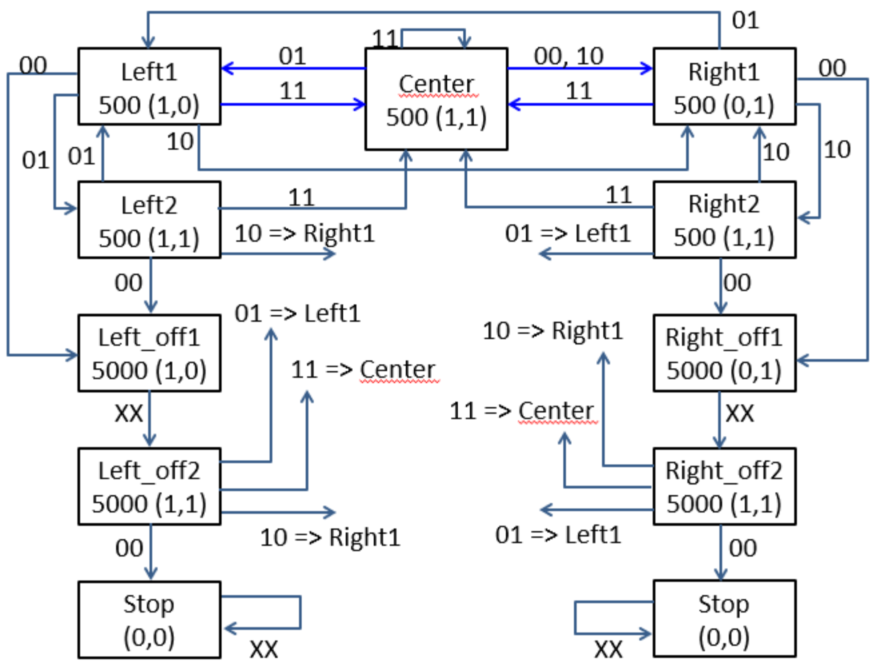
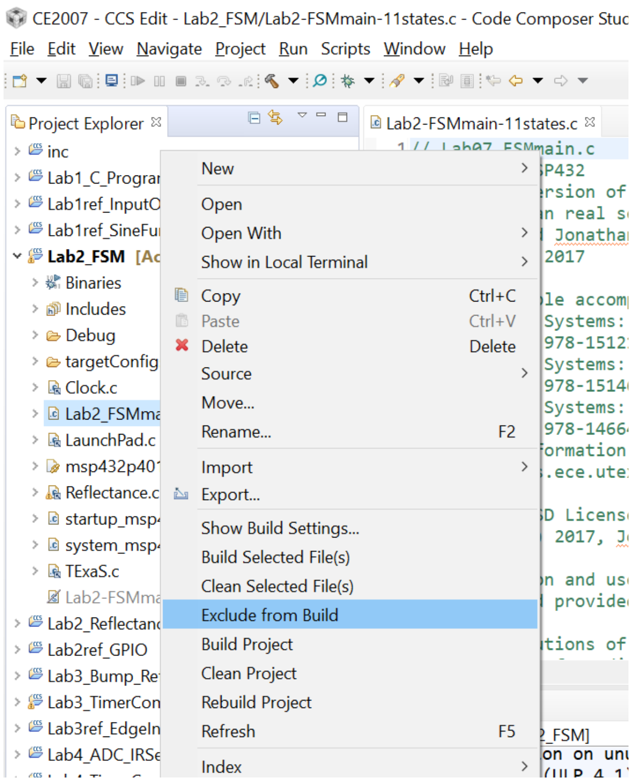


Figure 10: Final FSM design

#Task:

- The FSMmain-11states.c file is already included in the Lab2_FSM project but excluded from compilation. Open the project, exclude Lab2-FSMmain-3states.c and include Lab2-FSMmain-11states.c in the project compilation, rebuild the project to verify if the code function as per FSM diagram in Figure 10. Check Figure 11 below to see how to include and exclude specific files into a project. It is the same menu item that toggles the exclude/include option.
  - Sensor input to the FSM is obtained from real Reflectance sensor
  - Verify that the LEDs (simulating Motors) response to lines position below the robot
  - Correct any bugs found in the FSM array wrt the FSM design in Figure 10.
- Note that motor actions are simulated by LEDs in this project. See Figure 9.
- Potential File to modify: Lab2_FSMmain.c

Exclude File



Include File

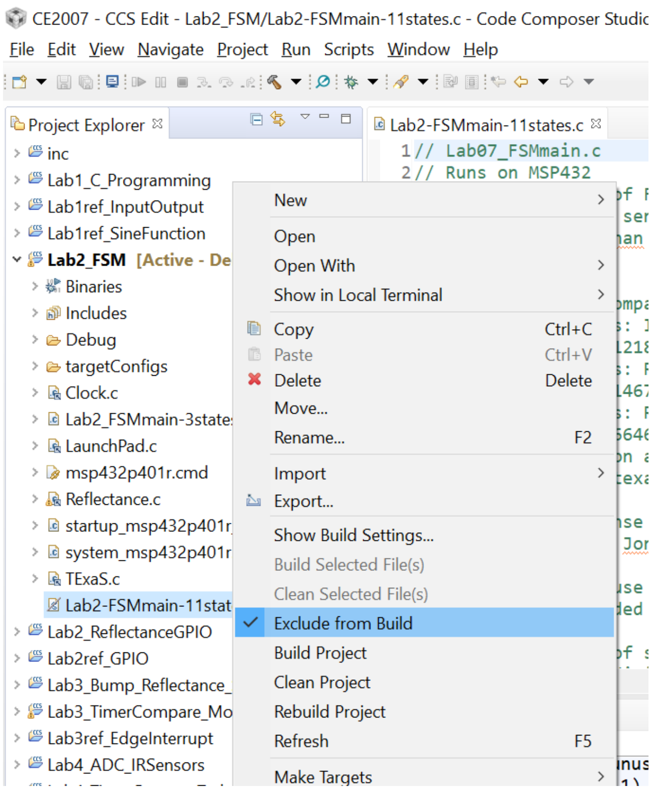


Figure 11: Include/Exclude Files from CCS project compilation

8. ASSIGNMENT

- Complete the Lab2 handout and submit before your next lab.

9. Optional Lab (Bit-Banding)

Bit-Banding

In ARM processor, the 1Mbyte SRAM memory is aliased to a 32Mbyte range in the processor memory map. And user can access each bit in the SRAM memory in an atomic manner, by accessing its alias address. For example, the bit0 in byte location 0x2000_0000 in SRAM is aliased to a 4-byte address range 0x2200_0000 to 0x2200_0003. Writing a '1' to location 0x2200_0000 is equivalent to setting bit0 of 0x2000_0000 to '1'. See Figure 1-3 below.

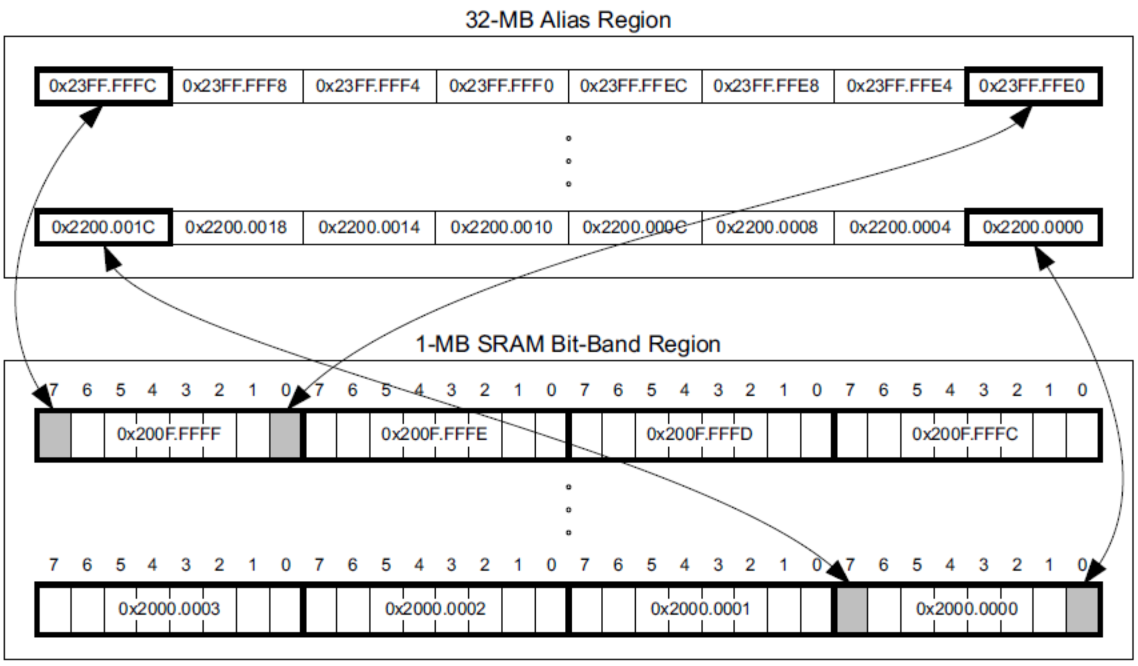


Figure 1-3. Bit-Band Mapping

In MSP432, the address map where the peripheral registers reside is also aliased. See figure below.

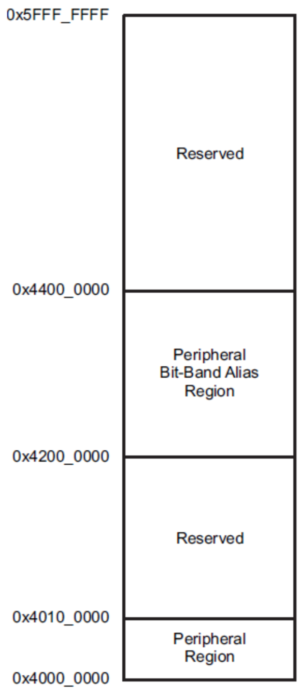
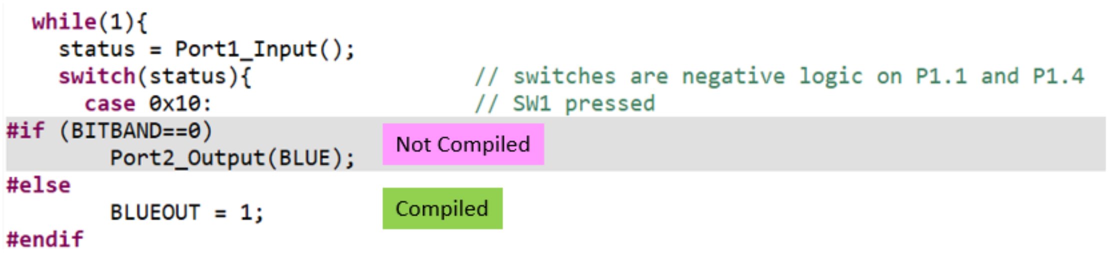


Figure 6-4. Peripheral Zone Memory Map

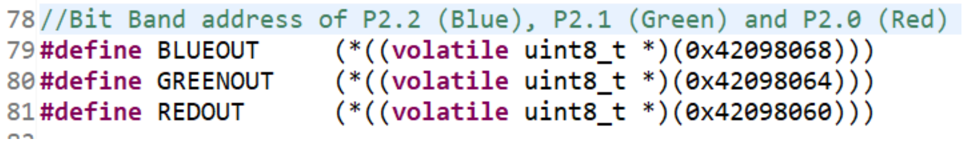
Similarly, Bit0 of address 0x4000_0000 is mapped to alias region 0x4200_0000 to 0x4200_0000. Each bit in the original memory is mapped to 4 bytes in the alias region.

Search the main.c for a "**#define BITBAND 0**" and modify the BITBAND macro from '0' to '1'. Rebuild the project. Compiler will compile the lines "BLUEOUT=1;" instead of "Port2_Output(BLUE);".



```
  while(1){
    status = Port1_Input();
    switch(status){                     // switches are negative logic on P1.1 and P1.4
      case 0x10:                        // SW1 pressed
#if (BITBAND==0)
        Port2_Output(BLUE);
#else
        BLUEOUT = 1;
#endif
```

Check the values assigned to BLUEOUT and other LEDs. Use the following information to check if the you are able to derive the same alias addresses.



```
//Bit Band address of P2.2 (Blue), P2.1 (Green) and P2.0 (Red)
#define BLUEOUT      (*((volatile uint8_t *)(0x42098068)))
#define GREENOUT     (*((volatile uint8_t *)(0x42098064)))
#define REDOUT       (*((volatile uint8_t *)(0x42098060)))
```

Address of Port2 output data register (P2->OUT) is 0x4000_4C03. P2.0 (RED LED) is bit0 of 0x4C00_4C03. Note that each bit in the original memory is mapped to a 4-byte location in the alias region.

Two key points in this sub-section:

- How to use #define directive to selectively compile codes in your program.
- Verifying bit-banding operation in ARM processor.

.