



LABORATORY MANUAL

SC2107

Microprocessor System Design and Development

Lab Experiment #3

***Exception Handling and
Timer Compare Operations***

**SESSION 2023/2024
SEMESTER 1**

**SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
NANYANG TECHNOLOGICAL UNIVERSITY**

This lab session may be done individually or in groups of 2.

1. OBJECTIVES

- 1.1 Understand exception handling in ARM Cortex M4F in Processor Startup sequence, GPIO port interrupts, SysTick and Timer Operations
- 1.2 Using Timer hardware to generate PWM signals to motor

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. EXCEPTIONS HANDLING

Exceptions arise whenever the normal flow of program is halted temporarily due to the occurrence of an event that needs the immediate attention of the processor. It is commonly known as 'Interrupts'.

In ARM Cortex-M4F, exceptions can be roughly grouped into three categories, system and fault exceptions (1-15 below) and IRQs (16-255). NVIC controls the IRQs. Note that the number of IRQs supported differ for different ARM processors in the market. Refer to Table 6-39 in the MSP432 datasheets [5] (Pg 117) for the full list of IRQs supported by MSP432 and their respective IRQ numbers.

| Exception Num | CMSIS INT Num | Exception Type | Priority level | Function |
|---------------|---------------|----------------|-----------------|--|
| 1 | - | Reset | -3 (Highest) | Reset |
| 2 | -14 | NMI | -2 | Non-Maskable interrupt |
| 3 | -13 | HardFault | -1 | All classes of fault, when the corresponding fault handler cannot be activated because it is currently disabled or masked by exception masking |
| 4 | -12 | MemManage | Settable | Memory Management fault; caused by MPU violation or invalid accesses (such as an instruction fetch from a non-executable region) |
| 5 | -11 | BusFault | Settable | Error response received from the bus system; caused by an instruction prefetch abort or data access error |
| 6 | -10 | Usage fault | Settable | Usage fault; typical causes are invalid instructions or invalid state transition attempts (such as trying to switch to ARM state in the Cortex-M3) |
| 7-10 | - | - | - | Reserved |
| 11 | -5 | SVC | Settable | Supervisor Call via SVC instruction |
| 12 | -4 | Debug monitor | Settable | Debug monitor – for software based debug (often not used) |
| 13 | - | - | | Reserved |
| 14 | -2 | PendSV | Settable | Pendable request for System Service |
| 15 | -1 | SYSTICK | Settable | System Tick Timer |
| 16-255 | 0-239 | IRQ | Settable | IRQ Input #0-239 |

A recap of the power up sequence of the MSP432 can be found in Table 1. Note that the first exception vector fetch is the reset vector. In all the CCS projects, you'll find a startup file (startup_msp432p401r_ccs.c) which contains the entire exception table. The reset vector points to a Handler routine Reset_Handler() which in turn made a jump to a label _c_int00 which is the starting address of the bootup routine that initialise the C environment. For those interested, the boot.asm file which contains the _c_int00 routine can be found in the directory C:\ti\ccs740\ccsv7\tools\compiler\ti-cgt-arm_16.9.6.LTS\lib\src or its equivalent depending on where CCS is installed. This routine prepare the environment for C program to be executed. _c_int00 will call the main() after the initialization.

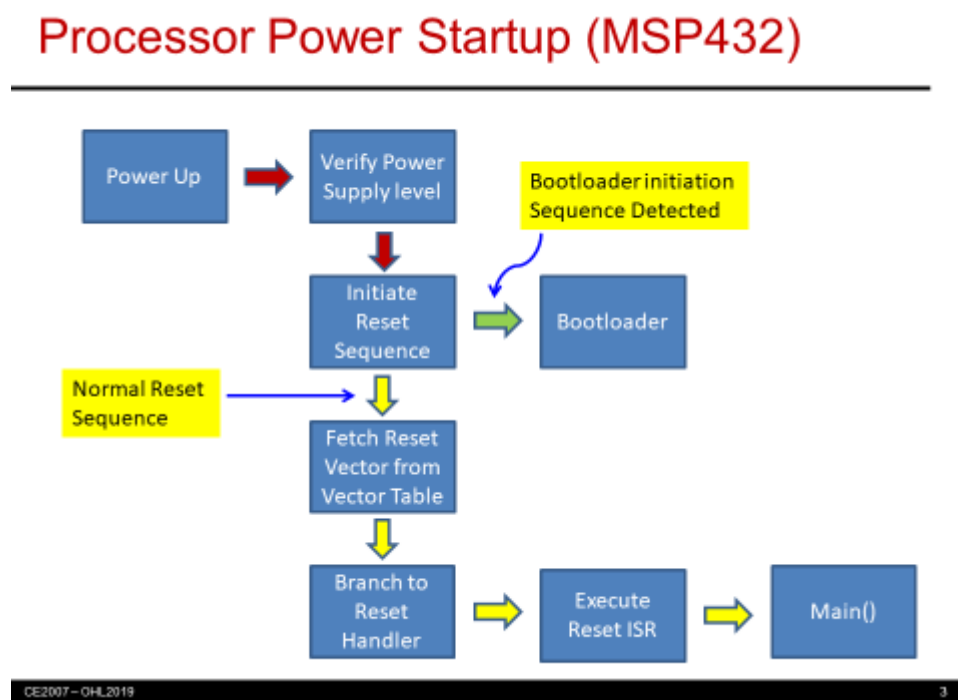


Figure 1: MSP432 Power up sequence

Interrupts mechanism is controlled at three levels: **Global, NVIC and Peripheral**. A few important points on interrupt mechanism to take note:

Peripheral specific configuration

- Depending on the Peripheral in use, some configuration register bits must be initialized to enable the interrupt at the peripheral module level.
- E.g. GPIO module requires configuration of the 'IE', 'IES' bits to enable the GPIO interrupts and to configure whether to trigger the interrupt at rising or falling edge of the input signal.

NVIC configuration

- This is the next level of Interrupt Handling mechanism. To enable a particular interrupt under NVIC, **you need to know the corresponding interrupt position, i.e. IRQ number, within the NVIC controlled interrupt list.** This is processor dependent, for MSP432 the list can be found in Table -39 (Pg 117) of the MSP432 Datasheets, part of the table and relevant register descriptions are shown in Figure 2 below.

- Timer_A0 CCR0 interrupt has an index 8 (list starts with index 0), so to enable Timer_A0 CCR0 interrupt, you will need to set bit 8 of **ISER0** register to '1', ISER0 register description can be found in slau356 (Pg 116).
- Port4 interrupt has an index 38, so it will be bit 7 of **ISER1** register. ISER0 is 32-bit wide so will contain interrupts with index 0-31.

Table 6-39. NVIC Interrupts (continued)

| NVIC INTERRUPT INPUT | SOURCE | FLAGS IN SOURCE |
|----------------------|------------------------|--|
| INTISR[4] | FPU_INT ⁽²⁾ | Combined interrupt from flags in the FPSCR (part of Cortex-M4 FPU) |
| INTISR[5] | FLCTL | Flash Controller interrupt flags |
| INTISR[6] | COMP_E0 | Comparator_E0 interrupt flags |
| INTISR[7] | COMP_E1 | Comparator_E1 interrupt flags |
| INTISR[8] | Timer_A0 | TA0CCTL0.CCIFG |
| INTISR[9] | Timer_A0 | TA0CCTLx.CCIFG (x = 1 to 4), TA0CTL.TAIFG |
| INTISR[10] | Timer_A1 | TA1CCTL0.CCIFG |
| INTISR[11] | Timer_A1 | TA1CCTLx.CCIFG (x = 1 to 4), TA1CTL.TAIFG |
| INTISR[34] | DMA_INT0 | DMA completion interrupt |
| INTISR[35] | I/O Port P1 | P1IFG.x (x = 0 to 7) |
| INTISR[36] | I/O Port P2 | P2IFG.x (x = 0 to 7) |
| INTISR[37] | I/O Port P3 | P3IFG.x (x = 0 to 7) |
| INTISR[38] | I/O Port P4 | P4IFG.x (x = 0 to 7) |
| INTISR[39] | I/O Port P5 | P5IFG.x (x = 0 to 7) |
| INTISR[40] | I/O Port P6 | P6IFG.x (x = 0 to 7) |
| INTISR[41] | Reserved | |

Figure 2-20. ISER0 Register

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|--------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| SETENA | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| R/W-0h | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

Table 2-26. ISER0 Register Field Descriptions

| Bit | Field | Type | Reset | Description |
|------|--------|------|-------|--|
| 31-0 | SETENA | R/W | 0h | Writing 0 to a SETENA bit has no effect, writing 1 to a bit enables the corresponding interrupt. Reading the bit returns its current enable state. Reset clears the SETENA fields. |

ISERx register contains interrupt enable bits for each IRQ (Interrupt request) controlled by NVIC

E.g. bit 0 of ISER0 register correspond to IRQ0. Bit 0 of ISER1 register correspond to IRQ32. (aka ISER0 concatenates ISER1. Set the bit to '1' to enable the interrupt

Figure 2-21. ISER1 Register

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|--------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| SETENA | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| R/W-0h | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

Table 2-27. ISER1 Register Field Descriptions

| Bit | Field | Type | Reset | Description |
|------|--------|------|-------|--|
| 31-0 | SETENA | R/W | 0h | Writing 0 to a SETENA bit has no effect, writing 1 to a bit enables the corresponding interrupt. Reading the bit returns its current enable state. Reset clears the SETENA fields. |

Figure 2: Sample of the NVIC interrupt table, ISER0 and ISER1 register description

- To change the priority of a particular interrupt, you need to modify the IPRx register. ARM Cortex M4F supports priority level from 0-255 so it requires 8 bits to store the priority information. Hence, each IPRx register contain priority info for 4 interrupts. Figure 3 below is extracted from Pg 122 of the slau356h document. Timer_A0 CCR0 interrupt correspond to IRQ8 so its priority info is in bit 7-0 of IPR2 register. Note that for MSP432, only the upper 3 bits is valid, so if you want to set the priority to 2, then a value 0100 0000b is written to bit 7-0.

Figure 2-32. IPR2 Register

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|--------|----|----|----|----|----|----|----|--------|----|----|----|----|----|----|----|--------|----|----|----|----|----|---|---|--------|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| PRI_11 | | | | | | | | PRI_10 | | | | | | | | PRI_9 | | | | | | | | PRI_8 | | | | | | | |
| R/W-0h | | | | | | | | R/W-0h | | | | | | | | R/W-0h | | | | | | | | R/W-0h | | | | | | | |

Table 2-38. IPR2 Register Field Descriptions

| Bit | Field | Type | Reset | Description |
|-------|--------|------|-------|--------------------------|
| 31-24 | PRI_11 | R/W | 0h | Priority of interrupt 11 |
| 23-16 | PRI_10 | R/W | 0h | Priority of interrupt 10 |
| 15-8 | PRI_9 | R/W | 0h | Priority of interrupt 9 |
| 7-0 | PRI_8 | R/W | 0h | Priority of interrupt 8 |

Figure 3: IPR2 register content.

Global

- There are some registers that control the interrupt mechanism behaviour at the global (chip) level. Registers such as PRIMASK and BASEPRI. Fortunately, both these registers' default value upon reset allows interrupts to be triggered, hence they are not initialized for our labs. See lecture notes for details.

Sample interrupt procedure setup

- Create Interrupt Handler routine.
- Link Handler routine to vector table.
- Configure Peripheral related registers (if any)
- Configure Interrupt priority via **IPRx** register (if needed)
- Initialize **ISERx** register to enable the target interrupt.
- For example, to configure Timer_A0 CCR0 interrupt
 - Create Timer_A0_CCR0_ISR() Handler routine
 - Ensure that the Handler name is the same as that specified in the vector table in startup_msp432p401r_ccs.c. Note that the default name of the handler for Timer_A0 CCR0 interrupt in startup_msp432p401r_ccs.c is TA0_0_IRQHandler.
 - Enable interrupt in relevant timer and channel control registers.
 - If required, configure the interrupt priority via the IPRx register.
 - Enable the Timer_A0 CCR0 interrupt by setting the corresponding bit in ISERx register.

Overview of tasks for Lab 3

There are two sections that requires coding.

- **Section 6.3.1.** Uses systick timer to trigger an interrupt periodically. The **ISR triggered will read the reflectance data** and **poll the bump switch status**. The interrupt mechanism's configuration is already done in the code provided but you will need to **code the systick ISR to enable proper reading** of the reflectance sensor data.
- **Section 6.3.2.** The **reflectance data is still read via systick timer** ISR. But the **bump switch status will only be read when there is a physical contact** on one or more bump switches, **via GPIO interrupt mechanism**. The GPIO pin connected to the bump switch output is configured to be interrupt capable. When the **bump switch is closed** physically, a **GPIO port interrupt will be triggered** and the **corresponding ISR is used to read the Bump Switch status**. You will need to **configure the GPIO interrupts (Peripheral, NVIC) to enable the Bump Switch detection**. Do you need to configure the global interrupt registers?

6.1 SysTick Timer

Systick timer is a timer that source its clock from the processor clock and can be configured to generate an interrupt whenever the timer counts down to zero from its configured period.

To use the systick timer, you need to **configure bits 2-0 of STCSR register** and **initialize the period register (STRVR)**.

| Name | Name Description | Type | Address | Description |
|-------|--------------------------|------|------------|--|
| STCSR | SysTick Control & Status | RW | 0XE000E010 | Basic control e.g. enable, clock source, systick interrupt enable/disable. |
| STRVR | SysTick Reload Value | RW | 0XE000E014 | Value to be load current value register when 0 is reached. |
| STCVR | SysTick Current Value | RW | 0XE000E018 | The current value of the count down |

Table 2-54. STCSR Register Field Descriptions

| Bit | Field | Type | Reset | Description |
|-------|-----------|------|-------|--|
| 31-17 | RESERVED | R/W | 0h | |
| 16 | COUNTFLAG | R | 0h | Returns 1 if timer counted to 0 since last time this was read. Clears on read by application of any part of the SysTick Control and Status Register. If read by the debugger using the DAP, this bit is cleared on read-only if the MasterType bit in the AHB-AP Control Register is set to 0. Otherwise, the COUNTFLAG bit is not changed by the debugger read. |
| 15-3 | RESERVED | R/W | 0h | |
| 2 | CLKSOURCE | R | 1h | Clock source. 0b = Not applicable 1b = Core clock |
| 1 | TICKINT | R/W | 0h | |
| 0 | ENABLE | R/W | 0h | Enable SysTick counter 0b (R/W) = Counter disabled |

6.2 Bump Switch

Bump switches allow you to know when the robot has contacted an obstacle. If the internal pull-up resistor is enabled, the switch will report a **logic '1' when there are no obstacles** encountered and a **logic '0' when it hit an obstacle** and closed the switch. Data from the reflectance sensors and bump sensors will be collected periodically using SysTick exception. Using exceptions to handle the reading of sensors provide a resource-efficient solution.

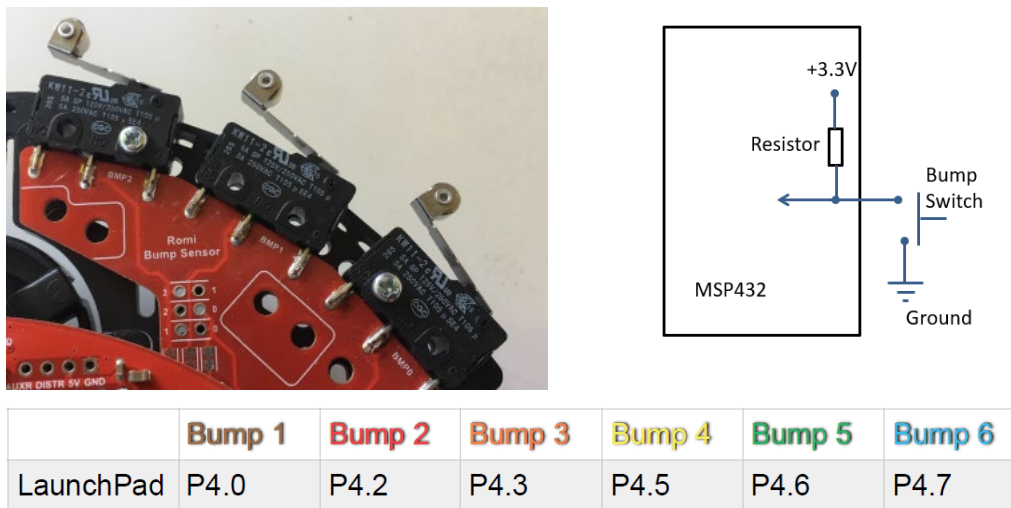


Figure 4: Bump Switches (Logic '0' when contact is closed)

6.3 Reading Reflectance and Bump Switch values

6.3.1 Via SysTick Timer Exception

In this section, you will use the periodic nature of the systick timer exception to read the reflectance sensor and the bump switch value. Open the project Lab3_Bump_Reflectance_Systick. Explore the project to see how the systick timer is being initialized and configured to generate periodic exceptions to the system. The periodic operation that you need is placed in the systick handler routine. Locate that and you should see a few Work-in-Progress functions there.

- Reflectance_Start()
- Reflectance_End()
- Bump_Read()

In Lab2, you developed the Reflectance_Read() function to sample the reflectance sensor data. In Reflectance_Read(), a **spin loop is used to implement the time delay t_b** before the sampling the capacitor discharge status. Spin loops are wasteful as it hogs the CPU. We are going to **implement this delay using systick exception** so CPU can be freed to perform other task during this delay period. So **step 5 of the original procedure will be removed, Step 1-4 will go into Reflectance_Start(), Step 6-7 goes into Reflectance_End()** and the delay in step 5 is tracked using systick interrupts.

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 μ s 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' μ s 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.

Figure 4a: Original procedure from Lab2 for sampling Reflectance Sensor Data.

The reflectance sensor reading is done in the background using SysTick interrupts. If the SysTick interrupts are occurring at 1000 Hz (every 1ms), a counter variable can be used to track the timing of events. `Reflectance_Start()` implements Step1-4. This function is called every tenth time in the SysTick ISR.

1. Set P5.3 and P9.2 high (turn on 8 IR LED)
2. Make the P7.7-P7.0 outputs, and set them all high
3. Wait 10 μ s
4. Make the P7.7-P7.0 inputs

Note that the delay you have used in Lab2 is 1ms, that means `Reflectance_End()` should be called in the subsequent ISR.

6. Read the 8-bit sensor result
7. Turn off the 8 IR LEDs (P5.3) low

Step 8 (Store the data into a shared global variable) will occur in the SysTick ISR itself. If you are sampling the line sensor at 1000 Hz, there will be eight SysTick interrupts during which the software performs no operation, one interrupt that calls `Reflectance_Start()`, and one interrupt that calls `Reflectance_End()`. Verify if the sample code is doing what was described.

It is a good debugging practice to toggle a port pin/LED during each ISR execution so that you could visualise quickly whether the ISRs are being triggered. Place the sensor data in a memory watch window and use the debugger in a similar manner to Lab 2.

You will also need to write one function to initialize the bump sensors, `Bump_Init()`. This function simply sets the appropriate port pins and enables internal resistors as needed. A second function, `Bump_Read()`, reads the switches and returns one 8-bit result. Program the code to sample Bump switches periodically every 10ms. One way to do it is to call `Bump_Read()` every time `Reflectance_End()` is called. The detail connection between the Bump switches and the MSP432 can be found in the RSLK-MAX Construction guide (sekp164.pdf).

#Task:

- Explore the program flow in Lab3_Bump_Reflectance_Systick. Understand how systick is initialized to generate periodic exceptions.
- Complete the functions Reflectance_Start(), Reflectance_End(), Bump_init() and Bump_Read().
- Potential Files to modify
 - Reflectance.c
 - Bump.c

6.3.2 Bump Switch reading via GPIO Port Interrupts

In section 6.3.1, both the reflectance and bump switch status are read within the systick ISR. This is still some form of polling. It is ok for reading the reflectance sensors but is a waste of CPU resource for the bump switch reading. This is because you only need to read the bump switch if one or more of the bump switches are asserted. A more efficient way is to configure the Port4 pins to be interrupt enabled. When any of the bump switches are asserted, a falling edge is detected on the Port4 pin(s) and an interrupt request will be sent to the CPU. The status of the Port4 pins can be read within the Port4 ISR.

#Task:

- Remove the need to do periodic polling for Bump Switch reading. Reading of bump switch should only be required when there is a contact with obstacle. This can be done by migrating the Bump Switch reading function into the Port4 ISR. Note that the bump switches are connected to the Port4 GPIO pins.
- Check the project Lab3ref_EdgeInterrupt for reference on how to configure and use a GPIO port interrupt. This project configured some pins in Port1 for interrupt triggered operations.
- You will need to perform following operation.
 - Modify Bump_Init() in Bump.c to include codes for interrupt configuration. Do not forget to initialize corresponding bit(s) in the NVIC->IP[] and NVIC->ISER[] registers when configuring Port4 for interrupt operations. Check the Lab3ref_EdgeInterrupt project for code examples.
 - Add a PORT4_IRQHandler() to service the interrupts coming from GPIO Port4. Read the bump switch status from the ISR. Remember to clear the IFG register after reading to avoid multiple triggers from the same interrupt request.
 - Note that Port1Task in Lab3ref_EdgeInterrupt project does not have an input argument, so its declaration is “void (*Port1Task)(void);”. If an input argument is required, it will be declared as “void (*Port1Task)(uint_8);”, if the input argument is of ‘uint_8’ data type.
- Potential Files to modify
 - Lab3_Sensorsmain.c
 - Bump.c

6.4 Motors (Code integration)

This section covers the DC motors used to move the robot (Yes, it is finally moving). You can google online for the details of how a DC motor works, essentially, we need to supply power into the motor for it to move. The higher the voltage, the faster the motor moves. In a real system, we typically have a motor driver IC controlling the voltage and current into the DC motor. The hardware interface includes an H-bridge motor driver, for this robot, the Texas Instruments DRV8838 driver IC is used. It allows software to spin each motor forward or backward. The software can vary the electrical power delivered to each motor using pulse width modulation (PWM). Essentially, the average voltage supplied to the motor is a function of two variable: the Supply voltage and the PWM duty Cycle. E.g. if the supply voltage is 6V and duty cycle is 50%, then the average power supplied to the motor would be similar to that using a 3V DC supply.

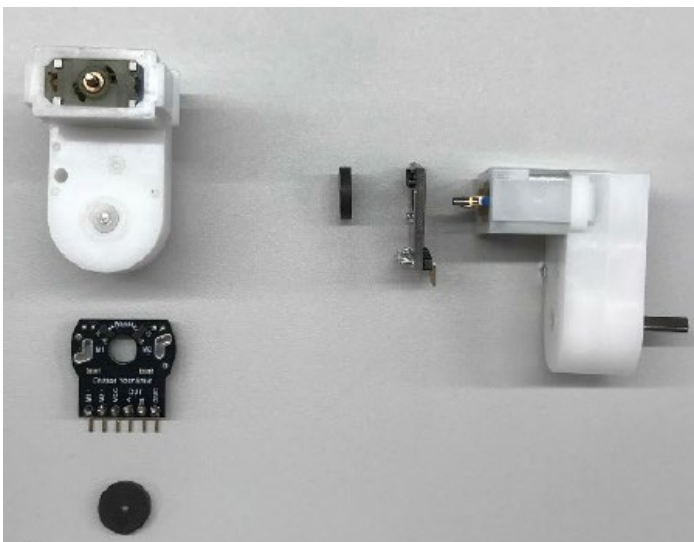


Figure 5: DC motor and Tachometer

Figure 5 shows a photo of the DC motor and the Tachometer used to measure the wheel speed. Figure 6 shows the schematic of the motor circuit. Figure 7 shows the logics used by the motor driver.

- nSLEEP=1 puts the motor driver to sleep mode
- EN=1 enable the motor driver input
- PH pin controls the direction

Typical PWM operation is achieved by pulling the EN pin to logic High or Low according to the required duty cycle. This will gate the voltage supply going into the motor driver IC.

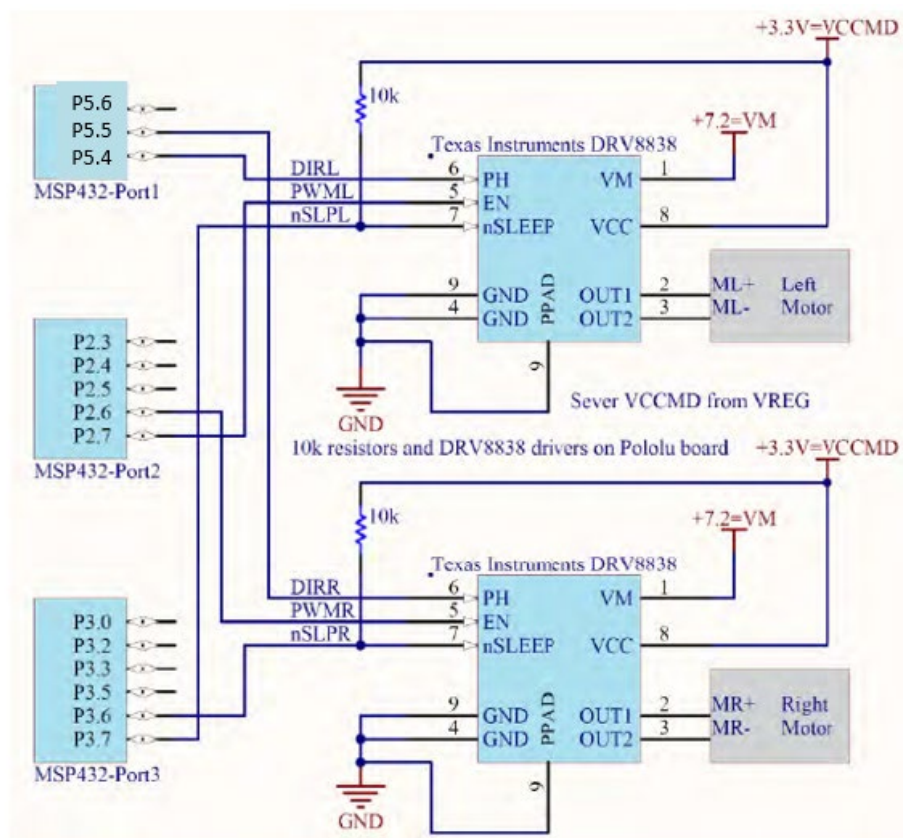


Figure 6: DC Motor Circuit

| nSLEEP | PH | EN | OUT1 | OUT2 | Function (DC Motor) |
|--------|----|----|------|------|---------------------|
| 0 | X | X | Z | Z | Coast |
| 1 | X | 0 | L | L | Brake |
| 1 | 1 | 1 | L | H | Reverse |
| 1 | 0 | 1 | H | L | Forward |

Figure 7: DRV8838 Device logic

| LaunchPad | TI-RSLK chassis board | DRV8838 | Description |
|-----------|-----------------------|---------|-----------------------|
| P5.5 | DIRR | PH | Right Motor Direction |
| P3.6 | nSLPR | nSLEEP | Right Motor Sleep |
| P2.6 | PWMR | EN | Right Motor PWM |
| P5.4 | DIRL | PH | Left Motor Direction |
| P3.7 | nSLPL | nSLEEP | Left Motor Sleep |
| P2.7 | PWML | EN | Left Motor PWM |

Figure 8: Pin connection to MSP432

6.5 Generating PWM via Timers

Timers are very useful peripherals found in many processors. The exact features supported by timer peripheral in different processor will be different but in general, they offer two main features: compare and capture. In this section, you explore how to use the timer compare feature to create periodic tasks and to generate PWM signals for the DC motors.

We are using the Timer_A peripheral of MSP432. The timer peripheral can derive its clock from different sources as shown in Figure 9 below. There are two dividers that can be used to further divide down the clock before it reaches the timer. These are controlled by the ID and IDEX bits in the TAxCTL and TAxEX0 registers. See section 19.3 in slau356h for details.

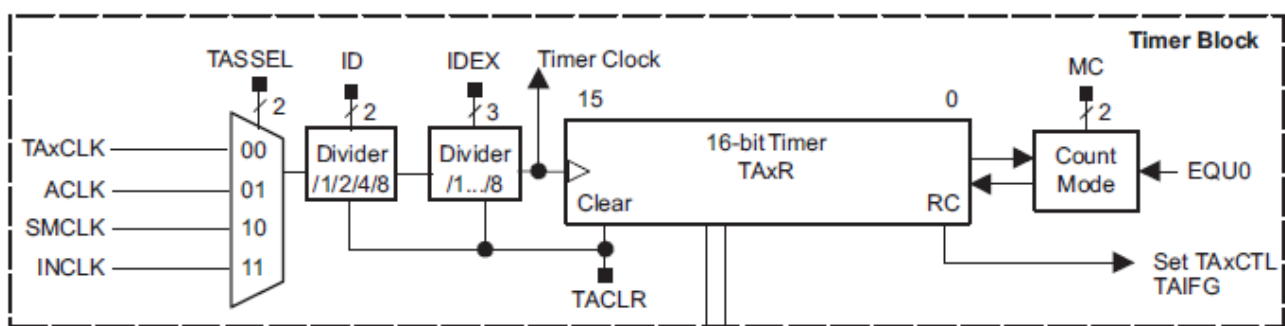


Figure 9: Clock distribution for Timer_A

Timer_A peripheral supports a number of counting modes. We will be using the UP/DOWN mode to generate the PWM signal. Figure 10 below shows the Timer_A in Up/Down counting mode. We will be using Output Mode 2: Toggle/Reset for this section.

The example illustrates the case of using TAxCCR2. When in Toggle/Reset compare mode, the corresponding Timer output pin (TAx.2 for TAxCCR2) is toggled when the timer counts to the TAxCCR2 value. It is reset when the timer counts to the TAxCCR0 value. This also means that the period of the PWM signal is twice that of the timer period (TAxCCR0 value).

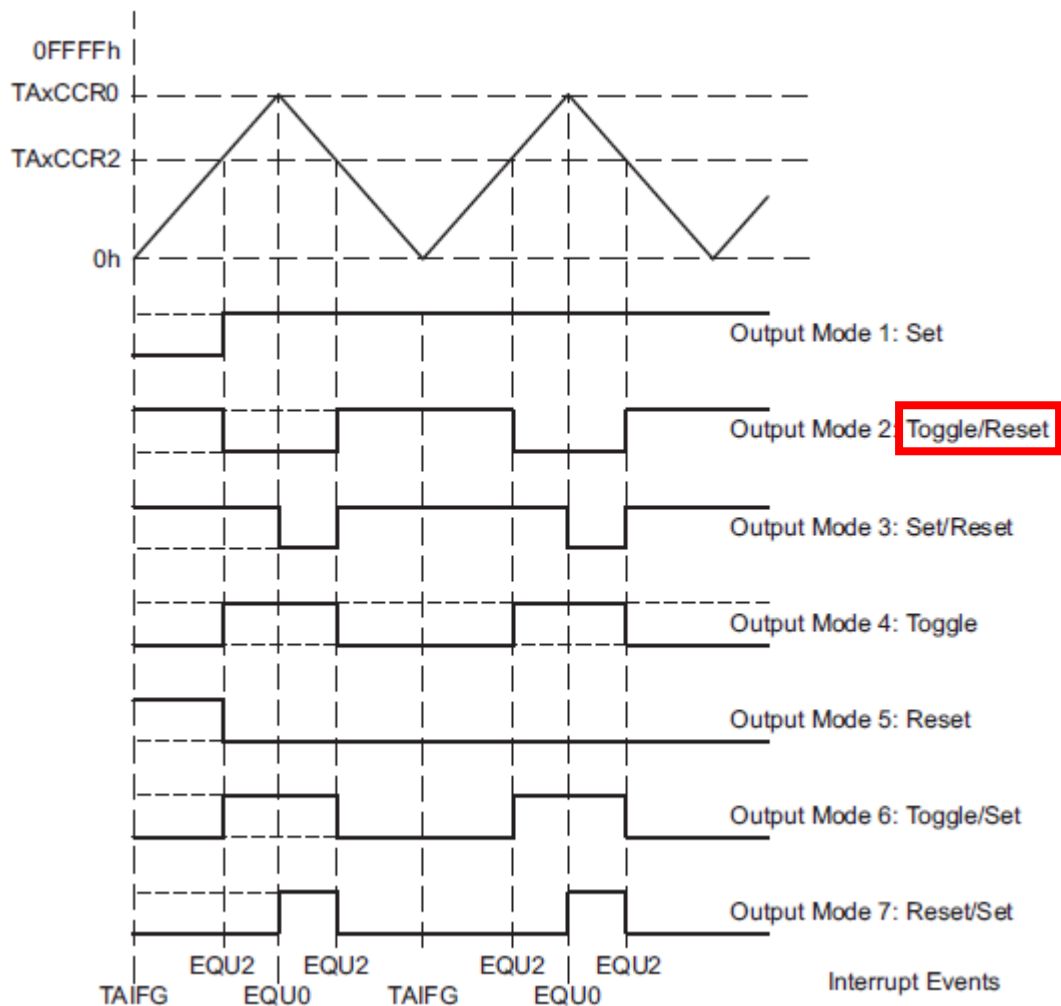


Figure 10: Timer_A Up/Down Mode Compare Operation

#Task:

- Open the project Lab3_TimerCompare_Motor and verify that the robot can move according to the movement configured in the code and is able to stop when one or more bump switches are asserted.
- The motor project uses the code you developed for the bump switches. If the return value of the Bump_read() is not compatible with what the motor code requires (packed 6 bits occupying the last 6 bit position), it is very likely that the motor will not move when the code is first integrated.
- In the downloaded code, the motor will stop if Bump_read() did not return a value of 0x3F. i.e. none of the bump switches are asserted. Check and modify the code according to your own bump switch code behaviour (if required).
- Potential Files to modify
 - Motor.c

In addition, investigate the following.

- Look into PWM.c in the project and look at PWM_Init134(), PWM_Duty3() and PWM_Duty4(). These functions use TA0CCR3 and TA0CCR4 to generate PWM signals. Channel 3 and 4 is used because the EN pin of the motors are connected to P2.6 and P2.7, which is the TA0.3 and TA0.4 output pins of Timer_A0 (See MSP432 Datasheets Pg 147).
- Is interrupt mechanism used in the PWM signal generation via Timers?

The reading of the reflectance sensor and bump switch is done periodically via Timer_A1 interrupts. Look at how the TimerA1 is initialised and how the timer ISR is linked. Each Timer_Ax is served by two interrupt vectors, which one is the Timer_A1 using in this project? Why is that so?

7. **ASSIGNMENT**

- Complete the Lab3 handout and submit before your next lab.