ELECTRICAL AND ELECTRONICS ENGINEERING DEPARTMENT

Introduction to Interrupts through Stepper Motor Driving

Experiment 5 - Introduction to Interrupts through Stepper Motor Driving

Objectives

The previous laboratory session was dedicated to handling the parallel I/O using the polling technique. Namely, you monitored the ports continuously to check whether an input exists. What would you do if you have to do other tasks while monitoring the I/O port? For instance, stepper motor-driving is one of the most common task performed by the MCUs and it requires periodic signals to be generated. On the other hand, your I/O monitoring software may call delay subroutine, which can violate the periodicity of the generated signals. To handle such cases, there is tool coming with the MCU: Interrupts. The aim of this lab is to to go over the utilization of the MCU with the interrupts and stepper motor driving will be an instrument to experience the aforementioned concepts. In this lab, you will learn,

- The concept of interrupts and their trigger-response structure

- Differences between polling-driven and interrupt-driven I/O

- How to write interrupt service routines (ISR) for the TM4C

- How to generate delays using the System Timer system

- How to drive a stepper motor

1   Background Information

1.1   Interrupts

Lab 3 briefly described the register model for the TM4C123GH6PM. This register model contained the Program Status Register (PSR). Eight of the bits in the PSR (bits 7:0) indicate which interrupt, if any, is currently being handled.

Another special register associated with interrupts is the PRIMASK register. The PRIMASK register contains only one bit (also named PRIMASK) that enables / disables most interrupts. The user cannot turn off a non-maskable interrupt. A maskable interrupt may be turned on or off by the user under program control. Its corresponding interrupt masking bit (I) is set to logic 1 during system reset, which turns off the maskable interrupt system. The user can turn on this system using the Change Processor State Interrupt Enable CPSIE I instruction and turn off this system using the Change Processor State Interrupt Disable CPSID I instruction. The TM4C123GH6PM is equipped with several maskable interrupts, e,g.SysTick, Timer, Pulse accumulator, SPI, and A/D system. In this lab, we will consider only one maskable interrupt, i.e., the System Timer (SysTick). Other interrupts are discussed in future labs. The System Timer generates an interrupt at a user specified interval. This interrupt is very useful in reminding the TM4C123GH6PM to perform a regular, repetitive task such as measuring the temperature in the controlled room at a regular interval.

1.1.1   Interrupt Service Routine

An interrupt service routine (ISR) is similar to a subroutine. It is called by the system when an *interrupt* occurs. Recall from the lectures that there are two main interrupt service approaches which are polling and vectored interrupts. TM4C123GH6PM supports vectored interrupts. In that scheme, each interrupt source has an address dedicated to it and that address contains the address of the ISR. The design of the MCU is in such a way that when an interrupt occurs, the PC fetches the corresponding address of the ISR from the vector map which is the memory block dedicated to the interrupt sources in order to hold the addresses of the ISRs. A complete interrupt vector map for TM4C123GH6PM is listed in the file Startup.s available on ODTUClass course page. This map tells you where the interrupt vector the appropriate interrupt service routine is located for all possible TM4C123GH6PM interrupts.

In general, the main program runs, possibly in an infinite loop, to perform some user defined useful tasks. However, you sometimes need to perform additional tasks that actually depend on internal or external requests. A request can be considered as an interrupt to the main program. When an interrupt occurs, the system runs the ISR. In that manner, an ISR is similar to a subroutine but it responds to a request, i.e. it is called by the system, whereas a subroutine is called because of an instruction in the program.

1.1.2   Coding with Interrupts

There are two main tasks to properly make use of the interrupts. The first task is to write an ISR and the second task is to configure and initialize the interrupt source that you want to use. The latter is to be explained in the following subsection 1.1.3 specific to *System Timer* interrupt. Recall that ISR is actually a subroutine which is called by the system when the related interrupt occurs. Thus, writing an ISR is actually writing a subroutine which is something you know from Experiment-2. There is no difference, all you do is labeling your subroutine by using PROC and ENDP directive doublet and ending your subroutine code with BX LR instruction. The issue is to link your subroutine with the related interrupt so that your subroutine becomes an ISR.

Remember that a 4-Byte memory space is dedicated to each interrupt source. In that space, the address of the corresponding ISR should be placed. Thus, all you need is to place the address where your subroutine starts in the reserved memory space of the interrupt you are using. There are several ways to do that. When you examine the vector map defined in the start up code, *Startup.s*, you will realize that

the reserved memory spaces for the interrupt vectors are filled via assembly directive that uses labels and resolves the address value that a label corresponds:

```
;*******************************************************************************
;
; The vector table.
;
;*******************************************************************************
        EXPORT    __Vectors
__Vectors
        DCD       StackMem + Stack              ; Top of Stack
        DCD       Reset_Handler                 ; Reset Handler
        ...
        DCD       SysTick_Handler               ; SysTick Handler
        ...
```

Therefore, as long as the label of your subroutine matches with the label of the corresponding interrupt source, your subroutine is the ISR. When you further examine the start up code, *Startup.s*, you will see code sections devoted to the interrupt service subroutines:

```
...
;*******************************************************************************
;
; This is the code that gets called when the processor first starts execution
; following a reset event.
;
;*******************************************************************************
        EXPORT    Reset_Handler
Reset_Handler
        ...
        IMPORT    __main
        B         __main
        ...
SysTick_Handler PROC
                EXPORT    SysTick_Handler              [WEAK]
                B         .
                ENDP
        ...
```

Notice that the first ISR is the reset handler. Reset is an unmaskable interrupt and its ISR is called when the system reset occurs. You can see that your main program is called by this ISR with the specific label __main. That is why you have been labeling your main program with the label __main so far. The ISR sections of the interrupt sources follow the reset handler. One option to link your subroutine with the interrupt source you are using is just modifying the start up code, *Startup.s*, and placing your subroutine in the corresponding code section. Yet, you might not want to do that especially when you are planning to have a generic start up code that can be used by other projects that uses the same interrupt sources. In that case, you may follow two approaches. One is implementing your ISR in a separate source file and using EXPORT directive to make its address achievable by the other sources and placing an IMPORT/EXTERN directive before the definition of the vector map in the start up code. In that approach, you have to comment out the code section corresponding the interrupt source you are using in order to prevent redefinition of the same label. For instance, if you are planning to use *System Timer* interrupt, your generic start up code after modification should be something like:

```
;*******************************************************************************
;
;  The  vector  table .
;
;*******************************************************************************
        IMPORT   SysTick_Handler              ;  the  label  is  to  be  imported
        EXPORT   __Vectors
__Vectors
        DCD      StackMem + Stack             ;  Top  of  Stack
        DCD      Reset_Handler                ;  Reset  Handler
        . . .
        DCD      SysTick_Handler              ;  SysTick  Handler
        . . .
        . . .
        . . .
;  The  SysTick_Handler  section  should  be  disabled
;  to  prevent  redefinition  of  the  same  label
; SysTick_Handler PROC
;                 EXPORT   SysTick_Handler              [WEAK]
;                 B        .
;                 ENDP
;        . . .
```

The other approach is calling your subroutine from the ISR implemented in the start up code, *Startup.s*. Again you should use EXPORT directive to make your subroutine achievable and you should use IM-PORT/EXTERN directive to refer your subroutine from the start up code. For instance, if you are planning to use *System Timer* interrupt, your generic start up code for the second approach after modification should be something like:

```
;*******************************************************************************
;
;  The  vector  table .
;
;*******************************************************************************
;  NO  CHANGE  AT  THIS  SECTION
        EXPORT   __Vectors
__Vectors
        DCD      StackMem + Stack             ;  Top  of  Stack
        DCD      Reset_Handler                ;  Reset  Handler
        . . .
        DCD      SysTick_Handler              ;  SysTick  Handler
        . . .
        . . .
        . . .
;  My_SysTick_Handler  subroutine  should  be  implemented  and  EXPORTed
                IMPORT   My_ST_ISR               ;  import  the  starting  address  of  ISR
SysTick_Handler PROC
                EXPORT   SysTick_Handler              [WEAK]
                PUSH     {LR}
                BL       My_ST_ISR
                POP      {LR}
                BX       LR
                ENDP
                . . .
```

By following the aforementioned approaches, you may link your subroutine with the interrupt source you are using and your subroutine becomes the ISR. Now, what is left is the initialization and the

configuration of the interrupt source you are planning to use. In the following section 1.1.3, the *System Timer* is to be explained, since it is interrupt source you will be using in the scope of this lab.

1.1.3 System Timer (SysTick)

From Experiment-4, you know that delays can be written by means of software. However, software delays reduces the utilization of the processor. Namely, no other tasks can be performed during the execution of a delay subroutine. If you want to perform a repetitive task and nothing else, you may use software delay to periodically perform your task. Yet, if you want to simultaneously perform repetitive tasks and more other tasks, then things become more complicated. One way to handle such situations is to make use of the *System Timer*.

There are three important control registers for the SysTick in the TM4C123GH6PM: STCTRL, STRELOAD, STCURRENT. You may refer to page 138 of the datasheet of the MCU [1] for more detailed information about the registers.

The Status and Control register (STCTRL) given by Figure 1 is used to enable the SysTick system (interrupt vector address 0x0000.003C). Bit 0 of this register is the System Timer Enable (ENABLE) bit. This bit when set to 1 enables the SysTick system, and when set to 0, disables the SysTick system. Bit 1 is the Interrupt Enable bit (INTEN) which enables interrupts for the SysTick system. When this bit is set to 1 interrupts are enabled and an ISR will be called when the timer reaches 0. Bit 2 is the Clock Source bit (`CLK_SRC`). The internal oscillator of the TM4C123GH6PM is not the only clock available to use. With additional setup, other more accurate oscillators can be used. However, for now we will simply use the Precision Internal OSCillator (PIOSC) divided by 4. The PIOSC runs at 16 MHz on the TM4C123GH6PM, so PIOSC/4 is 4 MHz. Hence the clock period is 250ns, which enables us creating delays in the order of seconds.

SysTick Control and Status Register (STCTRL)

Base 0xE000.E000
Offset 0x010
Type RW, reset 0x0000.0004

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
								reserved								COUNT
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

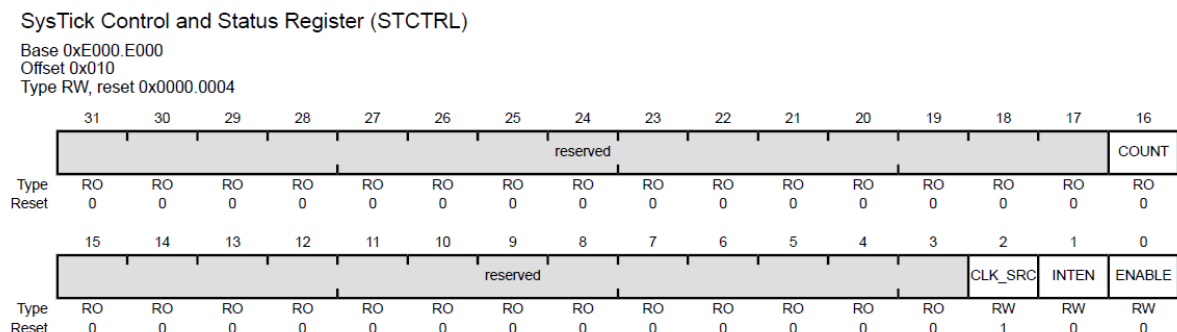|	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
						reserved								CLK_SRC	INTEN	ENABLE
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

Figure 1: Status and Control register (STCTRL) at 0xE000.E010.

The SysTick Reload Value register (STRELOAD) given by Figure 2 selects the time-out period of the SysTick system. RELOAD[23:0] is where the 24-bit starting value for the counter will be stored. The SysTick system will count down from this value to 0, subtracting 1 every clock cycle.

The SysTick Current Value register is a 24-bit register containing the current value of the count down from the RELOAD value.

The SYSPRI register configures the priority level, 0 to 7 of the SysTick exception handler by placing this level into the TICK field (bits 31:29). The lower the value of this field, the higher its priority. Recall from the lectures that a priority level of 1 or greater is required to enable SysTick interrupts. This register is byte addressable.

Once the functions of the registers are comprehended, an initialization and configuration routine can be written for the system timer interrupt. You are advised to write subroutines for initializations always, since the structure of your main code will be simpler and you may use your initialization subroutines in your future projects with slight modifications. Hence, it is better to implement the initialization and the service subroutines in the same file but separate from the main source file and to import the subroutines

SysTick Reload Value Register (STRELOAD)

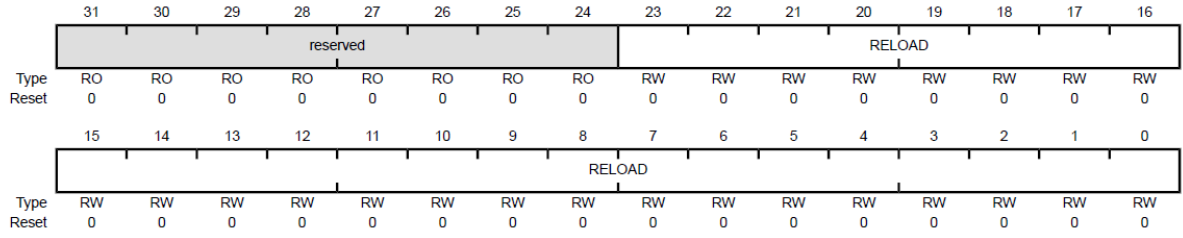Base 0xE000.E000
Offset 0x014
Type RW, reset -

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
				reserved							RELOAD					
Type	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
								RELOAD								
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 2: SysTick Reload Value register (STRELOAD) at 0xE000.E014.

SysTick Current Value Register (STCURRENT)

Base 0xE000.E000
Offset 0x018
Type RWC, reset -

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
				reserved							CURRENT					
Type	RO	RO	RO	RO	RO	RO	RO	RO	RWC	RWC	RWC	RWC	RWC	RWC	RWC	RWC
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

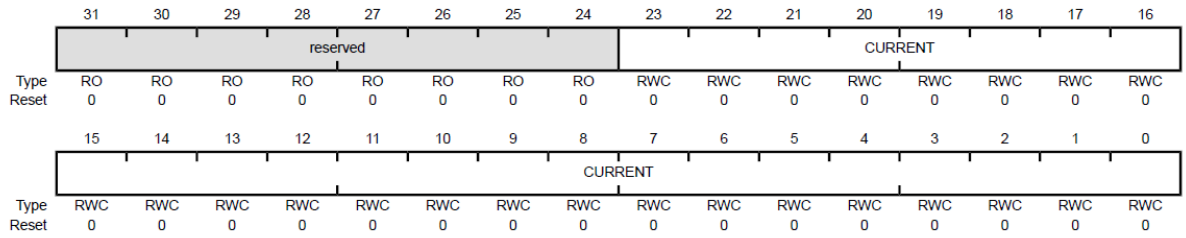|	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
								CURRENT								
Type	RWC	RWC	RWC	RWC	RWC	RWC	RWC	RWC	RWC	RWC	RWC	RWC	RWC	RWC	RWC	RWC
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 3: SysTick Current Value register (STCURRENT) at 0xE000.E018

System Handler Priority 3 (SYSPRI3)

Base 0xE000.E000
Offset 0xD20
Type RW, reset 0x0000.0000

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	TICK			reserved					PENDSV			reserved				
Type	RW	RW	RW	RO	RO	RO	RO	RO	RW	RW	RW	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

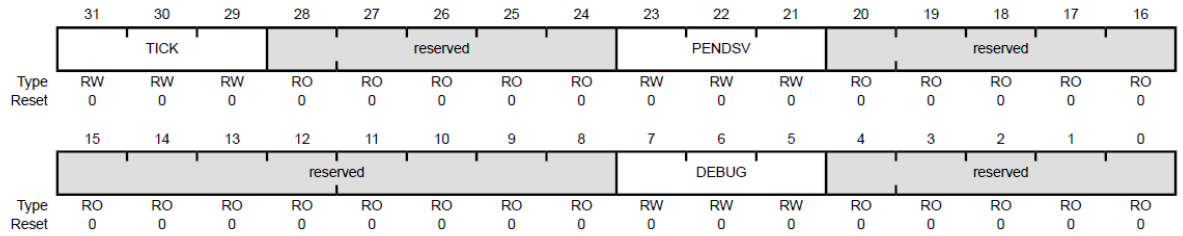|	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved								DEBUG			reserved				
Type	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4: SYSPRI3 register at 0xE000.ED20

to your main program. As an example, the following code implements the initialization subroutine to enable an interrupt that occurs in every $500\,\mu s$ and the user defined ISR. The initialization is to be called from the main program to configure and initialize the timer interrupt. The start up code, *Startup.s*, is assumed to be modified as explained in the previous section 1.1.2 to link the user defined ISR.

```
;*********************************************************
; Your SystemTimer.s source file to implement
; initialization and ISR
;*********************************************************
; Definition of the labes standing for
; the address of the registers
NVIC_ST_CTRL EQU 0xE000E010
NVIC_ST_RELOAD EQU 0xE000E014
NVIC_ST_CURRENT EQU 0xE000E018
SHP_SYSPRI3 EQU 0xE000ED20
; end of the register label definitions


; 0x7D0 = 2000 -> 2000*250ns = 500mus
RELOAD_VALUE EQU 0x000007D0



;*********************************************************
; Initialization area
;*********************************************************
;LABEL              DIRECTIVE          VALUE                   COMMENT

                    AREA               init_isr , CODE, READONLY, ALIGN=2
                    THUMB

                    EXPORT             InitSysTick
InitSysTick        PROC
; first disable system timer and the related interrupt
; then configure it to use isternal oscillator PIOSC/4
                    LDR                R1, =NVIC_ST_CTRL
                    MOV                R0, #0
                    STR                R0, [R1]
; now set the time out period
                    LDR                R1, =NVIC_ST_RELOAD
                    LDR                R0, =RELOAD_VALUE
                    STR                R0, [R1]
; time out period is set
; now set the current timer value to the time out value
                    LDR                R1, =NVIC_ST_CURRENT
                    STR                R0, [R1]
; current timer = time out period
; now set the priority level
                    LDR                R1, =SHP_SYSPRI3
                    MOV                R0, #0x40000000
                    STR                R0, [R1]
; priority is set to 2
; now enable system timer and the related interrupt
                    LDR                R1, =NVIC_ST_CTRL
                    MOV                R0, #0x03
                    STR                R0, [R1]
; set up for system time is now complete
                    ENDP
```

```
;************************************************************
;  SysTick ISR area
;************************************************************
;LABEL            DIRECTIVE        VALUE             COMMENT

                  EXPORT           My_ST_ISR
My_ST_ISR         PROC
                  ...                                ; your routine
                  BX               LR                ; return
                  ENDP


;************************************************************
;  Your main.s source file
;************************************************************
;LABEL            DIRECTIVE        VALUE             COMMENT

                  AREA             main , CODE, READONLY
                  THUMB
                  EXTERN           InitSysTick       ; Reference external subroutine

                  EXPORT           __main
__main            PROC
                  ...                                ; your other initializaitons
                  BL               InitSysTick       ; initialize system timer
                  CPSIE            I                 ; enable interrupts
                  ...                                ; rest of your code
                  ENDP
```

1.2   Stepper Motors

The stepper motor is an electromagnetic device that converts digital pulses into mechanical shaft rotation. Advantages of step motors are low cost, high reliability, high torque at low speeds and a simple, rugged construction that operates in almost any environment. The main disadvantages in using a stepper motor are the resonance effects often exhibited at low speeds and decreasing torque with increasing speed. These motors are commonly used in measurement and control applications. Sample applications include ink jet printers, CNC machines, volumetric pumps, etc.

The inner structure of the stepper motors are beyond the scope of this experiment. For this experiment we will consider stepper motors as black boxes, which convert digital pulses into mechanical shaft rotation. The protocol for this conversion is standard and very simple.

The stepper motor, like most other motor types has coils in its structure, which converts electrical current to magnetic force to rotate the motor shaft. Very simple schematic of a stepper motor with its control circuitry is given in Figure 5.

By energizing the coils respectively in each step, the stepper motor is rotated in one direction with fixed angles, which is prescribed in stepper motor's specifications. Every stepper motor has a fixed stepping angle. While reversing the current in an alternating fashion (windings energized consecutively), the motor makes this fixed angle of rotation. Essentially a digital input pattern with only one high bit from the driver is equivalent to one step. This mode of operation is called the *Full Step Mode* for step motors.

Applying these values to the controller outputs in the ascending order according to step index will make our motor make four steps of fixed angle rotation. Note that the table is cyclic in the sense that Full Step 1 follows Full Step 4. Reversing the direction of the order to descending (the order which the output is applied) will make the motor to rotate in the reverse direction.
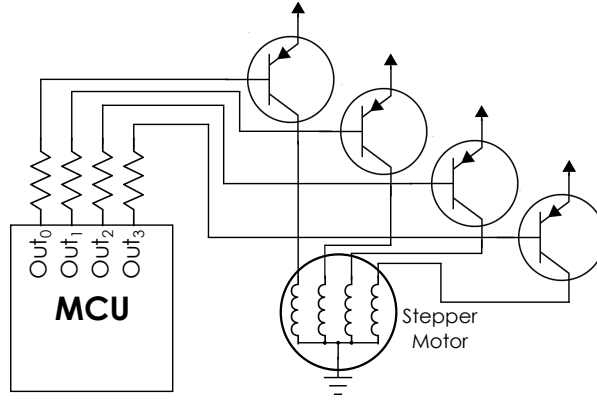
Figure 5: In this circuit, a micro-controller controls the rotation of a motor known as a step motor by sequentially activating one transistor at a time (thus, energizing one motor coil at a time). With each step in the sequence, the motor rotates a fixed number of degrees.

Step No.	Out_0	Out_1	Out_2	Out_3
Full Step 1	1	0	0	0
Full Step 2	0	1	0	0
Full Step 3	0	0	1	0
Full Step 4	0	0	0	1

1.2.1 Motor Driving

In the previous section 1.2, we have learned how to control the motors via applied voltages. But when it comes to the driving circuitry, the motor may require high currents compared to other interface devices. The current output of usual voltage outputs (like PIA) may not suffice this requirement. Because of similar reasons, separate motor driver IC's are used. For this experiment, you will use L293D.

The L293 and L293D are quadruple high-current half-H drivers. The L293 is designed to provide bidirectional drive currents of up to 1 A at voltages from 4.5 V to 36 V. The L293D is designed to provide bidirectional drive currents of up to 600-mA at voltages from 4.5 V to 36 V. Both devices are designed to drive inductive loads such as relays, solenoids, dc and bipolar stepping motors, as well as other high-current/high-voltage loads in positive-supply applications.

1.2.2 Motor Drive Control

Up to now, we have seen how to control and drive the stepper motors. For this experiment, as you may easily guess, we will use TM4C123GH6PM as the controller. For the experimental part you are going to implement the stepper motor driving control. The connections for stepper motor driving circuits are:
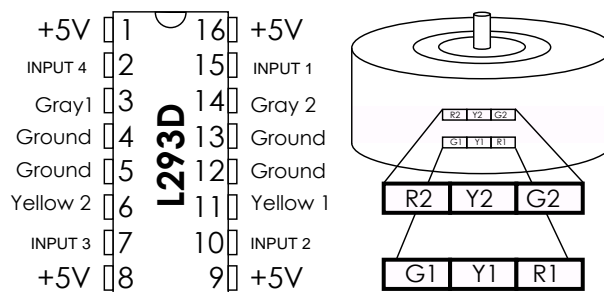
Important: Do not forget to connect Red1 (R1) and Red2 (R2) to +5V!

2   Preliminary Work

Assume you have started your own company and you want to develop an ink jet printer. You made a research and you understood there are different components needed. One of the required tasks for an ink jet printer is to relocate the ink jet throughout the page width. After your preliminary research, you realized that, this task is commonly done by stepper motors. Before you advance in the development of the rest of the printer, you should understand how to drive a stepper motor using a TM4C123GH6PM processor. Hence, you have decided to implement a system that controls the speed and direction of a stepper motor via external push buttons. You will implement your system step by step as follows:

1. Write a subroutine which sends a GPIO port the necessary signals to demonstrate the Full Step Mode in both directions (clockwise or counterclockwise).

2. Now, you will design a system that has two inputs from push buttons and provides a step to the stepper motor upon input. One button is for clockwise rotation and the other is for counterclockwise rotation. You will use 4 switches of the 4x4 Keypad Module introduced in Experiment-4. Draw the necessary connections between TM4C123G, L293D, 4x4 Keypad Module and stepper motor.

3. According to your hardware design in part-2, write a program that, in an infinite loop, gives a step upon the release of one button and gives a step in the opposite direction upon the release of the other button. You may assume that the other button is never pushed until the pressed button is released. The response of the motor should be after the button release. You should be aware of bouncing inherent in the switches.

4. At this stage, you will design a system that has 4 inputs from push buttons to control a stepper motor. One button is for speeding up, one is for slowing down, the other two are for directions. You will use 4 switches of the 4x4 Keypad Module introduced in Experiment-4. Draw the necessary connections between TM4C123G, L293D, 4x4 Keypad Module and stepper motor.

5. According to your hardware design in part-4, write a program that, in an infinite loop, drives a stepper motor speed and direction of which can be controlled by external push buttons. You may assume that the no button is ever pushed until a pressed button is released. The controls should be applied upon releasing the corresponding button. You should be aware of bouncing inherent in the switches.

   For this item, please explain how you attack the problem and draw a flowchart of your algorithm.

3   Experimental Work

1. <u>User Driven Stepper Motor:</u> Construct the circuit that you have prepared in 2nd step of preliminary work (2.2). Test the operation with the program that you have written in 3rd step of preliminary work (2.3). Demonstrate your results to your lab instructor.

2. <u>User Operated MCU Driven Stepper Motor:</u> Construct the circuit that you have prepared in 4th step of preliminary work (2.4). Test the operation with the program that you have written in 5th step of preliminary work (2.5). Demonstrate your results to your lab instructor.

Note that DEES's VCC is to be used for the motor driving circuit, since the VCC of the MCU cannot provide enough current for the coils.

!!!   Important Notice   !!!

When using DEES in your experiments, **DO NOT CONNECT VCC (Pins +3.3V & VBUS) of TM4C123G Board** to DEES's VCC terminal. Also, to make a common voltage reference, <u>CONNECT GND of TM4C123G Board to DEES's GND terminal</u>. Please make sure that you <u>turn all supplies OFF</u> while making any connection throughout laboratory work.

4   Parts List

```
1 x Stepper Motor
1 x L293D
1x 4x4 Keypad Module
DEES
```

References

[1] TI, "Tiva^{TM} tm4c123gh6pm microcontroller data sheet." `http://www.ti.com/lit/ds/spms376e/spms376e.pdf`.