Embedded Systems Design Laboratory Manual

ICOM4217

Electrical & Computer Engineering Department
University of Puerto Rico at Mayagüez
Mayagüez, PR 00681-9000

Danilo Rojas Luís Francisco Manuel Jiménez

May 15, 2018

Electrical and Computer Engineering Department
University of Puerto Rico at Mayagüez

ACKNOWLEDGMENT

The authors would like to thank Cesar A. Aceros, for his help creating the Latex template for the manual.

DISCLAIMER

Table Of Contents

Laboratory Rules

Usage of the Microprocessor Interfacing Lab facilities is subject to the abidance of the rules listed below:

1. The entrance to the facilities is reserved exclusively for students enrolled in the **ICOM 4217** course or approved projects. Any authorization for the use of facilities shall be designated by the Laboratory Director. Persons authorized to access the laboratory facilities and resources shall follow the rules and procedures established for the University of Puerto Rico system.

2. Students projects are performed in groups. Each group will be assigned a workspace with a computer and resources for the development of their project. It is the responsibility of each group to maintain and return those resources in good condition. The lab assistant will perform periodic inventory checks to ensure the integrity of loaned laboratory resources.

3. The University maintains a limited stock of resources to support project development. Resources requests can be made via a "Materials Request Form". Requests are granted subject to availability. It is the responsibility of each work group to acquire any materials that could not be provided from the laboratory stock. Materials and resources brought into the lab for the performance of a project must be removed from the premises at the end of the project performance period.

4. The laboratory is monitored 24/7 by security cameras . Anyone agreeing using the laboratory also agrees to the monitoring and recording their behavior by the security cameras.

5. Each authorized user will have access to the facilities using his or her provided access card. The access log will be used as a user registry. This registry could be used to establish responsibilities, if necessary.

6. The laboratory will remain open as long as a lab assistant is present. In his/her absence, authorized students can stop by the campus security office to request that the laboratory wooden door be opened. A student requesting such a service must present his or her student card at the office. The requester will be registered as the person in charge of the lab. If the person in charge leaves the laboratory, he or she must close the lab wooden door and notify the professor via email. If another student wants to stay in the lab, the outgoing student must transfer the responsibility by notifying the campus security office

or the professor via email. Any incident must be reported by the person in charge. The published schedule indicates the availability of a lab assistant in the facilities.

7. The consumption of beverages and/or food within the laboratory facilities is prohibited. This prohibition includes depositing waste food or drinks into the lab trash cans.

8. It is the responsibility of each group to maintain their work space neat and organized. The cutting, grinding or machining of materials that generate particulate is prohibited within the laboratory facilities.

9. Dress code: Students in the laboratory facilities should use proper attire for a sensitive electronics laboratory and particularly clothing that minimizes the generation of static electricity. Refrain from using vinyl clothing, rubber shoes (eg. Crocks) or other parts known to generate high levels of static electricity.

10. The entry of pets into the facilities is prohibited. This restriction excludes guide dogs used by blind people.

11. Removing any resource from the laboratory without written permission of the laboratory director or department director is prohibited.

12. It is forbidden to temporarily or permanently add or bring any type of resource to the lab without prior authorization of the laboratory director or department head. This rule excludes the use of laptop computers for personal use provided they are with their owner, and electronic components used in projects prototypes.

13. Accessing the network must always be done wirelessly. Plugging personal computers to the wired network is a violation of the Laboratory Regulations and will carry penalties.

14. The usage of the laboratory printer is subject to the institutional rules established for computer center printers. The system will deduct the number of printed pages from your print quota.

15. The transfer (loan) of accounts among students is strictly prohibited. If this were the case, the loaned account could be deactivated.

16. Any software installation requires prior authorization from the system administrator. The use, installation, or storage of programs or resources that violate current copyright law is not allowed.

17. Unnecessary noise within the laboratory is prohibited. Using external computer speakers is prohibited, except for projects that require so. In such cases, moderation is advised. The use of hearing aids will be permitted provided that their volume is moderate and does not disrupt the work environment.

18. The act of locking computers is limited to a maximum of ten minutes. If the computer were not unlocked before the timer expires the work session will be automatically terminated and the logged user logged-out without notice.

19. Students must respect the workspaces of their peers and refrain from assessing restricted access areas in the laboratory. Under no circumstances should a student sabotage or modify in any way the project area of other groups or access unauthorized areas.

20. The laboratory has designated seats for people who require special accommodations. Such individuals will have priority in using such resources.

21. It is the duty of every student to report any violation to the rules established herein. Violation of the dispositions contained in this regulation will be sufficient cause to initiate a disciplinary action against the offender; including denial of access to resources, removal from the facilities, and/or any other applicable legal action.

Students with special needs or requiring reasonable accomodation, please contact the laboratory coordinator, the class professor, or the laboratory teaching assistant.

Experiment 1

High-Voltage Safety

Objectives

- Understanding the concept of high voltage and its implications to the human body
- Recognizing dangerous current and voltages levels for the human body and their effects
- Computing the human body impedance and understanding its role in the levels of current flowing trough the human body
- Identifying the potential electric hazards in a laboratory space
- Learning and applying safe practices in a laboratory environment
- Learning how to apply emergency procedures in case an electric shock

Duration

- 2 Hours in the laboratory and extra time for complementary tasks

1.1 Introduction

Common electrical and electronic devices require the use of electrical energy to operate. The very same energy that makes them work can reach levels that could become harmful or even lethal for humans. Electric and electronic devices are commonly enclosed, but in some cases, this enclosure needs to be opened to make adjustments or to perform repair procedures, exposing areas that might be subjected to harmful. Moreover, when working in the development and prototyping of new applications and circuits, such activities expose developers to the same type of risks.

This experiment has an objective creating awareness of the levels of voltages and currents that could cause harm, how to identify hazards situations, minimize the risk of accidents, and how to responds in the event of an accident.

1.2 Terms and Definitions

The term **High Voltage** refers to electrical energy at a voltage high enough to cause injury or death. In a formal definition *"High Voltage is any voltage exceeding 1000V rms or 1000V dc with current capability exceeding 2mA as or 3mA dc, or for an impulse voltage generator having a stored energy in excess of 10mJ"* according to the *IEEE Trans.Power App . Sys., vol PAS-97, no. 6, 2243, November, 1978.* Although, in a relative sense 50Volts might not be considered strictly a high voltage, it represents the threshold where harmful effects occur in a adult body. For this reason, 50Volts is considered as a danger high voltage for the human body.

High voltages can be found in form of AC (Alternating Current), DC (Direct Current), or momentary pulsed signals. These signals can injure the human body depending on their voltage and current levels, being the AC voltage at 60Hz the worst possible voltage type and frequency for humans. At this frequency the human body is 5 times more sensitive than to direct current. Also, keep in mind that, although some voltage levels in DC are not supposed to be dangerous for the human, in AC, those levels can be fatal.

Other important definitions suitable for electrical and laboratory applications include:

- **Moderate Voltage**: Refers to voltages greater than 120 V rms or 120V dc but less than 1000V. With current capabilities of 2mA ac and 3mA dc respectively.
- **Temporary Setups**: System assembled generally for measurement over a period of time that not exceeds three months.
- **Troubleshooting**: Temporarily procedure carried out to repair or diagnose problems in a device or circuit energized with any voltage level.
- **Bare Conductor**: A conductor without covering or electrical insulation.
- **Covered Conductor**: A conductor enclosed within a material not necessarily electrical insulation.
- **Insulated Conductor**: A conductor enclosed within electrical insulation material.

- **Exposed Conductor**: Refers to parts that are not suitable guarded, isolated, or insulated.
- **Enclosed**: An object surrounded by a case, housing, fence, or walls that prevents persons to enter in contact with energized parts.

1.3 Human Body Impedance

The human body has his own resistance, allowing us to interact in some cases with electricity without suffering any type of damage. But, when this resistance is overcome, a current flow can pass through the entire body causing external or internal injuries. Although, on average, the skin resistance has a value between 1,000 ohms and 100,000 ohms, the internal body resistance is lower with values between 25 and 1,000 ohms. Table 1.1 shows the different skin resistance values depending of the part of the body and some specific conditions.

Table 1.1: Human skin resistance (*Source*: Electric Safety Manual, Berkley Laboratory)

Condition	**Resistance (Ohms)**	
	Dry	**Wet**
Finger touch	40,000 to 1,000,000	4,000 to 15,000
Hand holding wire	15,000 to 50,000	3,000 to 6,000
Finger-thumb gasp	10,000 to 30,000	2,000 to 5,000
Hand holding pliers	5,000 to 10,000	1,000 to 3,000
Palm touch	3,000 to 8,000	1,000 to 2,000
Hand around 1.5 in pipe or drill handle	1,000 to 3,000	500 to 1,500
Two hands around 1.5 in pipe	500 to 1,500	250 to 750
Hand immersed	–	200 to 500
Foot immersed	–	100 to 300
Human body, internal, excluding skin ohms	200 to 1,000	

The skin resistance depends mainly of the parameters that include:

- Area of contact
- Pressure applied
- Amount of current

- Waveform of the current (AC, DC)
- Duration of the shock
- Environmental conditions such as humidity, temperature, and pressure, among others.

The internal body resistance is affected by factors such as:

- Body mass (weight & height)
- Age
- Diseases
- Tissue type and amount

Once the factors that could affect the body resistance are known, the total body resistance can be estimated. The total resistance in the human body can be calculated as:

$$R_{total} = R_{skin(in)} + R_{internal} + R_{skin(out)}, \tag{1.1}$$

where $R_{skin(in)}$ denotes the skin resistance where the electric current enters the body, $R_{internal}$ refers to the resistance where the current flows, and $R_{skin(out)}$ is the resistance where the current leaves the body.

1.4 Current Flow Through the Human Body

When a person receives an electric shock or is electrocuted, it is because the human body works in that moment as a conductor. Whenever a person comes in contact with an energized bare conductor while also in contact with a grounded surface, a conduction path is established and current passes through his or her body.

The current can flow through the body affecting or not the organs in it's path. Although, some organs could be affected due to the current, there are some paths more dangerous than others. One of the most lethal paths is when current passes through the chest affecting the heart. This is usually the worst case scenario as it might interfere with electrical impulses of the heart making it stop. Current through the body can also cause severe injuries such as internal burns resulting the heat generated by the current flow. Table 1.2 shows different injuries caused by different current values. These values are not the same for every person due physiology and environmental factors.

Table 1.2: Electric current effects on the human body (*Source*: High Voltage Safety Manual, Colorado State University)

Effect/feeling	**Direct Current (mA)**		**Alternating Current (mA)**				**Incident Severity**
			60 Hz		**10,000 Hz**		
	150 lb	**115 lb**	**150 lb**	**115 lb**	**150 lb**	**115 lb**	
Slight sensation	1	0.6	0.4	0.3	7	5	None
Perception threshold	5.2	3.5	1.1	0.7	12	8	None
Shock not painful	9	6	1.8	1.2	17	11	Minor
Shock painful	62	41	9	6	55	37	Spasm, indirect injury
Muscle clamps source	76	51	16	10.5	75	50	Possibly fatal
Respiratory arrest	170	109	30	19	180	95	Frequently fatal
≥0.03-s vent. fibril.	1300	870	1000	670	1100	740	Probably fatal
≥3-s vent. fibril.	500	370	100	67	500	340	Probably Fatal
≥5-s vent. fibril.	375	250	75	50	375	250	Probably fatal
Cardiac arrest	–	–	4000	4000	–	–	Possibly fatal
Organs burn	–	–	5000	5000	–	–	Fatal if it is a vital organ

To have an idea of how severe a voltage shock can be, consider the following scenario. Let's suppose that our hands are sweaty and we touch with one hand an energized circuit with 50V and accidentally with the other hand a ground surface (Lethal path!). With the hands sweaty, the skin's resistance can drop to 1,000 ohms. Using the ohms law and supposing an internal resistance of 200Ω, we can calculate a current of:

$$I = \frac{V}{R_{total}} = \frac{V}{R_{skin(in)} + R_{internal} + R_{skin(out)}} = \frac{50V}{1000\Omega + 200\Omega + 1000\Omega} = 22.7mA \tag{1.2}$$

This current level as we can see in Table 1.2 can be potentially harmful for us because it can case us a cardiac arrest.

1.5 Risk Mitigation

To try to avoid electrical risks and hazards we need to take into consideration not only behavioral aspects but also our own senses. They can help us detect possible electric hazards in different ways. Typical indications, depending on the type of sense, include:

Visual indicators:

- High voltage warning signs & labels
- Flashes, arcs, corona discharge
- Cables with damaged insulation
- Burn marks on circuit
- Tripped breakers or GFCIs
- Dim or flickering lights

Audible indicators:

- Sizzles
- Buzzes

Tactile indicators:

- Tingling sensation
- Hot or burning wires, connectors, junctions, or other components

Odor indicators:

- Smell of burning wire or other components

Also, observing safety precautions during the assembly, testing, and debugging process, helps in the prevention of possible hazards. Some tips that could help you during the laboratory work and prototype construction are mentioned below:

- Before energizing your circuit:
  - Make all connections and configurations
  - Have someone else inspect your circuit
  - Locate all breakers and workstation power switches
- When energizing your circuit:
  - Observe: For arcs, sparks, smoke, or sings of heat

  - Listen: For cracking sounds, pops, or hisses
  - Smell: Odor to smoke or burning electronics
  - check: Current and voltage levels into the power supply. Recall the signs of a short circuit. $I \to \infty$ and $V \to 0$.

- Work with energized circuits only for debugging and testing purpose
  - Always be careful

General Measures include:

- Maintain an illuminated and organized workstation
- Keep the floor in your workspace completely dry
- Avoid exposed connections
- Set up your work area away from possible grounds that you may accidentally contact
- Do not reach for something you cannot see (Within an energized circuit or panel)
- Avoid working alone
- Never ignore high voltage warning signs
- Never enter alone into an area containing exposed electrical energy sources
- Wear personal protective equipment associated with the voltage you are handling
  - Goggles
  - Gloves if needed, specially to work with high voltage
  - Insulated shoes and tools
  - Do not wear conductive jewelry
- Use the buddy system
  - It is best to work with someone that has knowledge about what you are doing
- One hand in a pocket rule

  - Never touch an energized circuit with both hands

- Know your equipment's limitations
  - Do not exceed your equipment's insulation capabilities
  - Locate your equipment's power switches
  - Use and electrostatic discharge wrist wrap

- Connect/disconnect any test leads with the equipment unpowered and unplugged. Use clips leads or solder temporary wires to reach cramped locations or difficult to access locations

- Perform as many tests as possible with the power off and the equipment unplugged

- Use only the test instruments, and insulated tools rated for the voltage and current specified

- Keep all electrical cords away from areas where they may be pinched, such as off the floor, out of walkways, and out of doorways.

- Know the emergency procedures to follow in case of an accident

1.6 Emergency Response Procedure

In case of an emergency, there are some step procedures that you have to follow in order to guarantee, not only the life of the affected person, but also your own life. Follow this steps in order:

- Shut off the power source
  - Breakers, power strips, etc
  - Never touch a victim with bare hands before shutting off power the source. Use a nonconductive rod or something similar

- Call for help

- Pry the victim from the circuit

- Use CPR if you are trained or find help

Experiment 2

IDE, GPIOs, and LCD

Objectives

- Understanding the process of assembling, debugging, and executing a program with your microcontroller's IDE
- Identifying the basic structure of an assembly or C program
- Identifying and understanding your microcontroller architecture and main features
- Using I/O ports to interface the MCU to different electronics components
- Interfacing and using an LCD with a microcontroller

Duration

- 2 Hours in the laboratory and extra time for complementary tasks

Materials

Table 2.1: Bill of materials for completing Lab. 2

Item #	Qty	Description	Reference
1	1	Development board	
2	1	IDE application	
3	1	LCD display: 2 lines, 16 characters	W/HD 44780 Controller
4	2	Light Emitting Diode	5mm Red LED
5	2	1/4W Carbon fill resistor	330 Ω
6	2	1/4W Carbon fill resistor	4.7 KΩ
7	2	Momentary switch	Pusbutton

2.1 Introduction

2.1.1 Microcontroller IDE

IDE stands for Integrated Development Environment also called Integrated Design Environment or Integrated Debugging Environment.

For all upcoming experiments, the **IDE** of your chosen microcontroller will be used as the compiler, assembler, linker, and code debugger. Some Microcontroller Units (MCU) have more than one IDE choice. Choosing an IDE depends on the main language used to program your microcontroller (assembly, C or another language). Although the main languages to be used will be assembly and C, some IDEs allow you to add compilers for different languages.

Normally an IDE for programming microcontrollers consist of the following tools:

- **A code editor**: The code editor is a type of text editor used to enter and modify the source code in a programming language. It is basically the text processor in an IDE and provides, in the majority of the cases, cross references to the elements in the code.

- **A compiler or/and assembler**: A compiler is a program that translates/-transforms a source code (written typically in high-level) to a target code (typically in low-level). The target code in our case refers to an executable code in machine language that governs the behavior in our MCU.

- **A debugger**: Is a software program used to test other programs and find bugs (errors). The debugger warns the programmer about what types of errors it finds and the exact line number where they are found. Also, it allows to run the source code step by step to help determining execution and logic errors.

- **A download tool to program the MCU flash memory**.

- **Built-in automation options**.

When an IDE is selected, it is important to know the maximum limit of code you can write on it. Most demo versions of IDEs limit the code size to only a few kilobytes of length. Also, it is important to know if your IDE supports the type of JTAG or programming tool available in your MCU. The JTAG interface is used for debugging your code in your embedded system.

2.1.2 General Purpose Input/Output (GPIO)

To communicate with the external world, microcontrollers use input and outputs pins known as **General Purpose Input/Outputs (GPIOs)** that are part of the Peripheral Subsystem. GPIOs have the capability of exchanging information in the form of digital signals (0 or 1) to other devices or systems.

GPIO pins in an MCU are grouped as ports commonly made of 8 pins. Each I/O pin can be programmed independently as an input or output. Generally, each port has a couple of associated registers that allow for configuring the function of the pin (input or output) and determining the logic level that it is reading-in or sending-out. Some ports have specialized capabilities to perform other functions such as internal and external oscillator options, timers functions, hardware for pulse width modulation (PWM), watchdog timer, USART, SPI, I^2C, data converters, and brownout reset circuitry, among others.

An input port always transfers data towards the CPU. Input ports can be either buffered or latched. A buffered input only reads the current status present in the input. A latched input uses a latch to hold the input data until it is read by the CPU. Output ports are in most cases is latched, holding the output data until the next output operation is executed. Figure 2.1 shows the general structure of an I/O pin. The Port Direction latch is used to select the pin direction (through the P_Dir bit) and the latch Output is used to determine the output value (through the P_Out bit). The P_In signal is used to read the data present in the pin. $\overline{\text{Dir\_En}}$ and $\overline{\text{Data\_En}}$ allowing for a physical connection to the processor bus line.

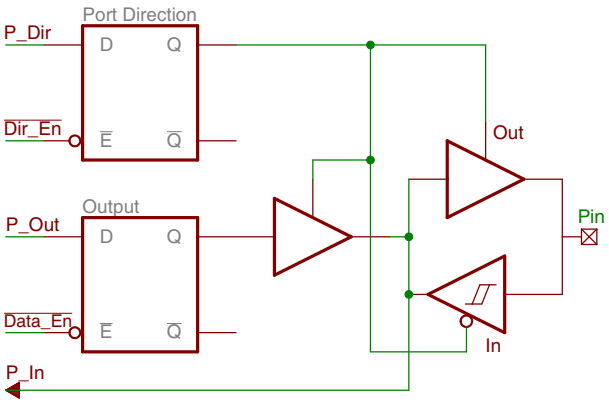


Figure 2.1: Basic structure of an Input/Output pin driver (*Source*: Introduction to Embedded Systems, M. Jiménez, R. Palomera, I. Couvertier)

GPIOs have electrical characteristics that define the currents and voltages that can be safely managed by the pin as either input or output. These characteristics and

descriptions include:

- V_{IL} **Input-low voltage**. Establishes the maximum voltage level that can be interpreted as a low by the pin input buffer.
- V_{IH} **Input-high voltage**. Represents the minimum voltage level that can be interpreted as a high by the I/O pin.
- V_{OH} **Output-high voltage**. The voltage level used to represent a logic "High" on an output pin.
- V_{OL} **Output-low voltage**. The voltage level used to represent a logic "low" on an output pin.

These characteristics must be taken into consideration when you are designing interfaces to be connected to I/O pins. Failing to observe such limits can cause malfunction or irreparable damage to the port electronics. See class's book Section 8.2.1 (Electrical Characteristic in I/O pins) for a detailed and extra explanation about electrical pins characteristics.

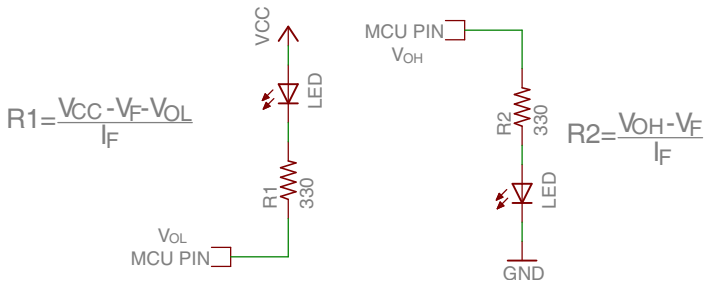
2.2 Basic Exercises

2.2.1 Blinking LED

Make an LED turn On and Off intermittently. The delay time to keep the LED On and OFF shall be about 100ms.

Follow the steps outlined below:

1. Identify the Port and Pin of your MCU that will be used to connect to the LED.
2. Connect the LED to the pin in one of the two possible configurations presented in Figure 2.2. Be sure to use a resistor that limits the current through the LED to a value below its current capacity. Take into account that when the LED is connected as illustrate in Figure 2.2(a) the MCU pin is sinking current, while when connected in as in Figure 2.2(b) the MCU pin is sourcing current. Be sure also, to not exceed the GPIO pin current limitations in terms of the maximum current the pin can source or sink.
3. Open your IDE.



(a) Pull-Up LED (b) Pull-Down LED

Figure 2.2: Pull-up vs. pull-down LED

4. Set the selected port and pin as output. If your are using configuration in 2.2(b), you should send a logic '1' to your pin to turn On the LED and '0' to turn it Off. Use a delay value that allows you to see the LED blinking. Use the inverse logic if you decided to use the configuration Figure 2.2(a). You can use the pseudocode in Listing 2.1 as a guide.

Listing 2.1: Blinking LED Pseudocode

```
;-------------------------------------------------------
;   Program Start
;   INIT RESET VECTOR
;   INIT STACK POINT, WDT
;-------------------------------------------------------
Port_bit = output                ;Set port pin as output
Port_pin = 0                     ;Initialize pin to ``low''

While   TRUE
    Port_pin = NOT(Port_pin)     ;Toogle pin
    wait = 2000h
    While wait>0                 ;Delay loop
        wait = wait - 1
    Endwhile
Endwhile
;-------------------------------------------------------
```

5. Assemble or compile your code and verify that it does not contain errors.

6. To run the code, select 'Run → Debug active project'. The progress information is displayed while the code downloads. Once the download is completed, the debug perspective shall open automatically.

7. Select 'Run' from the 'Run menu' and verify that the LED is blinking.

8. Pause the running programm and use the debugger options to place a break point in the instruction where the pin is toggled. Run the program, and see how the LED state changes.

9. Using the debugging tool to run your code step by step and see the hardware's reaction. Examine the contents of the count register and the port output register as you step through your program.

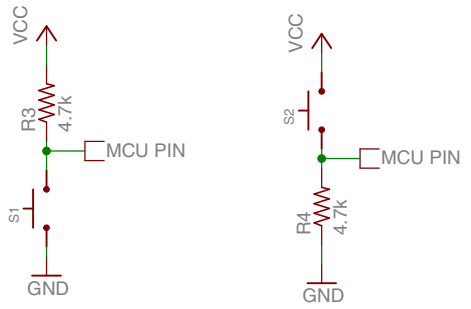
2.2.2 Polling a Switch

Read the state of an input Pin and make your LED turn On and Off accordingly.

Follow the steps outlined below:

1. Identify the Port and Pin of your MCU that will be used to connect the pushbutton.

2. Connect the pushbutton to the selected pin according to the configuration in Figure 2.3. Taking into account that when the pushbutton is connected as shown in Figure 2.3(a) the pin it reads a logic '1' while not depressed and "0" when depressed. The connection in Figure 2.3(b), it will revert the logic values read.



(a) Pull-Up Resistor (b) Pull-Down Resistor

Figure 2.3: Pull-up vs. pull-down resistor

3. Open your IDE.

4. Set the selected port and pin as input. In configuration (a) you should receive a '0' when the pushbutton is depressed, in that moment you have to turn the LED On and keep it in that state while the pushbutton is depressed. Once the

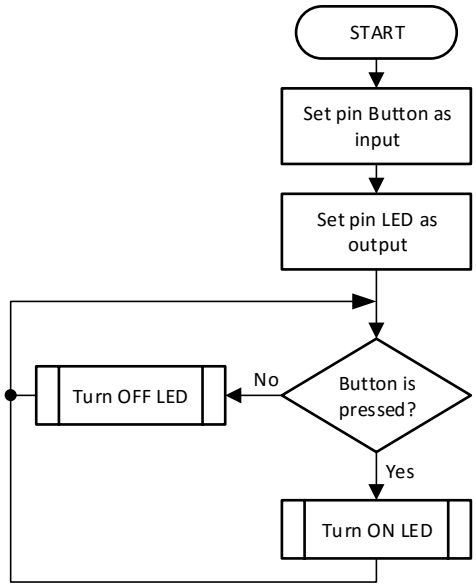


Figure 2.4: Polling a switch flow diagram

pushbutton is released, it should turn Off the LED. You can use the flowchart in the Figure 2.4 as a guide to write your code.

5. Compile your code and verify it does not contain errors.

6. Run your code and verify if the pushbutton is working.

7. Now, connect a second pushbutton and LED to your MCU. For the second pushbutton use configuration (b) and repeat from step 4, making the respective changes in the code for working with this pushbutton configuration.

2.2.3 LCD Configuration

Connect and configure the LCD (LM016L) to work with your MCU.

Follow the steps outlined below:

1. Using the LCD part number search on the web for the LCD's datasheet. Open it and find the device timing diagram.

2. Try to understand the signal sequences, commands, and timing metrics for your LCD and verify it's requirements. Ask your instructor in case of difficulty understanding the datasheet.

3. Connect your MCU to the LCD according to the following block diagram (Figure 2.5). Read the datasheet in order to get a better understanding of the LCD pins meaning.

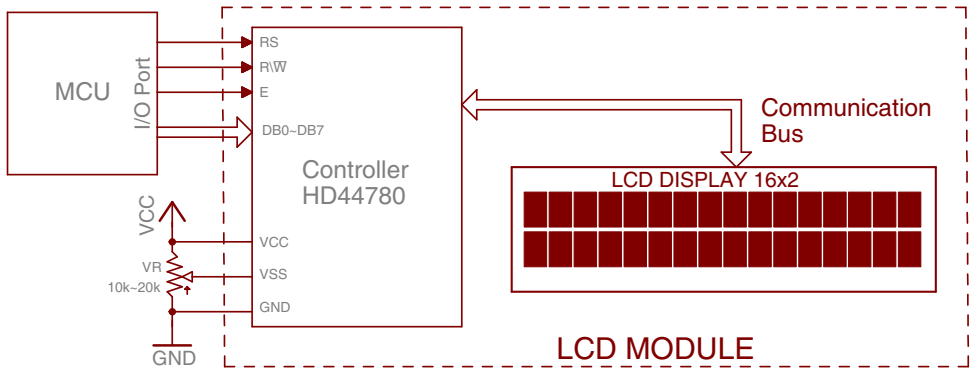


Figure 2.5: LCD block diagram connection

4. Open your IDE.

5. First, write a code to initialize the LCD with: Display ON, two line, 5x8-dot character font,and blinking cursor position character. You can use the flowchart presented in Figure 2.6 as a guide to write your code.

6. Compile your code and verify it does not contain errors.

7. Run your code and verify if the LCD is working as expected. At this point, you should see a blinking cursor on the LCD.

8. Create subroutines for each one of the LCD commands. The subroutines have to perform the following operations:

   - Clear the LCD
   - Set the Cursor to a Position
   - Write a Character
   - Write a Command
   - Write a Message (Use write a character function)

9. Test all your LCD subroutines.

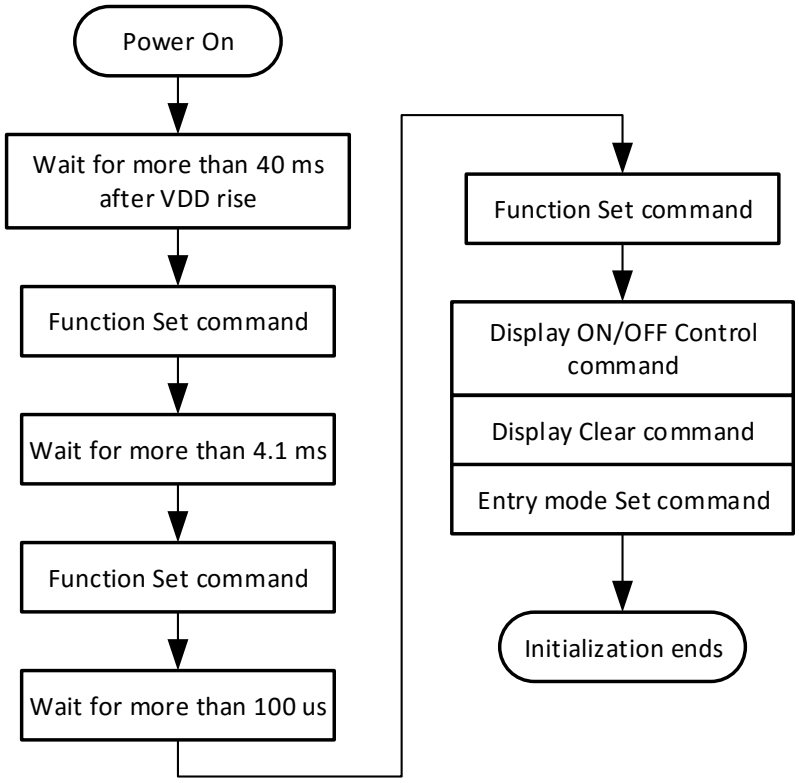


Figure 2.6: LCD initialization procedure (*Source*: HD44780U (LCD-II), Hitachi)

2.3 Complementary Tasks

2.3.1 Scrolling List

The activity consists of generating a circular scrolling list of messages and display them on the LCD. The list must consist of a minimum of 16 messages. For scrolling through the list you shall provide two pushbuttons connected to the MCU as shown in Figure 2.7. Two consecutive messages have to be displayed at the same time on the LCD; use the first line of the LCD for the first message and the second line for the following message.

Presentation and Report

Each basic exercise and complementary task must be presented to the TA before the initiation of the next laboratory experiment. The demonstration must be made personally in the lab and including the hardware and software components needed

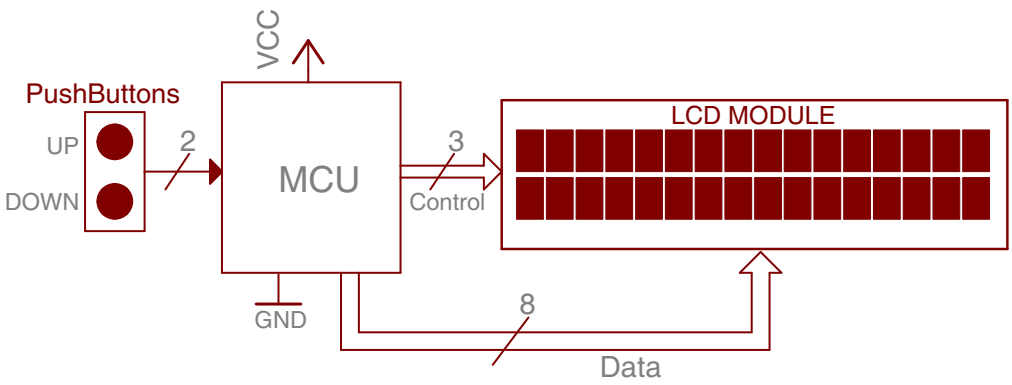


Figure 2.7: Scroll list connection diagram

for its completion.

An electronic report that includes the following information about the complementary task must be presented to the TA:

- Software plan and explanation (pseudocode or flowchart)
- Connection schematic with relevant component calculations (current, voltage, timing values, etc.)
- Code listing with comments.
- Additional information used to complete the task (Web pages, datasheets, books, etc.)

Experiment 3

Interrupts, Switch Debouncing, and Keypad

Objectives

- Understanding the bouncing phenomena and the issues related
- Identifying the main differences between hardware and software debouncing techniques
- Understanding how an interrupt process is carried out in an MCU
- Using interrupt to read keys
- Interfacing and using keypads with a microcontroller

Duration

- 2 Hours in the laboratory and extra time for complementary tasks

Materials

Table 3.1: Bill of materials for completing Lab. 3

Item #	Qty	Description	Reference
1	1	Development board	
2	1	IDE application	
3	1	LCD display: 2 lines, 16 characters	W/HD 44780 Controller
4	1	1/4W Carbon fill resistor	220 Ω
5	1	1/4W Carbon fill resistor	2.7 KΩ
6	1	1/4W Carbon fill resistor	3.3 KΩ
7	2	1/4W Carbon fill resistor	4.7 KΩ

Table 3.1: Continued

8	2	1/4W Carbon fill resistor	12 KΩ
9	1	Polarized electrolytic capacitor	4.7 uF
10	2	Momentary switch	Pusbutton
11	1	3 columns by 4 rows Buttons array	Keypad 3x4
12	2	Optoswitch	RPR-220
13	1	Schmitt trigger array	74LS14N

3.1 Introduction

3.1.1 Interrupts

An interrupt is an asynchronous signal produced by an external or internal event in a device that generates an interruption in the execution of a program. At the moment when an interrupt is executed, the processor executes a jump to an interrupt handler routine defined by the programmer. This routine is responsible for serving the interruption. Once the interruption is served, the processor returns to the execution of the interrupted program in the same position where the interrupt was executed. For this reason, interrupts provide one of the most useful features in microprocessors.

Interrupts provide an efficient mechanism to handle the service request from peripheral devices and external events in a computer system, allowing for a much more efficient use of the CPU when compared to polling. In addition, interrupt servicing also provides the following advantages:

- **Compact and modular code**: An Interrupt Service Routine (ISR) induces software modularity and software reusability.
- **Reduce energy consumption**: As ISRs lead to less CPU cycles, this reduces the energy consumed by the application.
- **Faster response time**: Provide a quick response to the triggering event.

To configure interrupts in a typical MCU it is important to follow these steps:

- **Setup the Stack**: If you are using assembly language, you will need to allocate stack space in memory and initialize the stack pointer (this is also necessary when you use call instructions). You do not have to do this if your MCU has a hardware stack or if you are programming in C language (the compiler makes this for you).

- **Write the ISR**: This is the code executed by the CPU to serve the interrupt. Avoid loops or calling subroutines from inside an ISR. Just write a simple and short code. It is important when you use ASM to write the ISR to keep register transparency (push all registers used in the ISR onto the stack at the beginning and pop them at the end of the ISR). If you are using C language the compiler does this for you.

- **Set-up the interrupt table**: Once your ISR is written, enter the ISR location in the interrupt table. This is how the CPU will know where the ISR is located. All MCUs have a table with entries for each interrupt source. In assembly, all you need to do is take the label from the ISR and write an absolute jump instruction to this label in the corresponding table entry.

- **Enable Interrupts**: Make sure that you have enabled the CPU global interrupt flag and the particular enable flag of the service. First set the flags corresponding to each device and then enable the CPU global interrupt flag. Many devices require re-enabling their interrupt flags at the end of the ISR to ensure it can be triggered again.

Interrupts are mainly triggered by hardware events knows as **hardware interrupts** such as a push-button depression, a threshold reached, and timer expiration. Their occurrence is asynchronous, making it impossible to know when it may occur. In contrast, **software interrupts** are predictable and become part of the normal program sequence. These are triggered by software instructions within a program.

Due to the number of different mechanisms able to trigger interrupts in an MCU, they have levels of priorities to determine which interrupt will be served first in the case of two or more interrupts request simultaneously arrive to the CPU. See class's book Section 7.1 (Fundamental Interrupt Concepts) for a detailed and deep explanation about Interrupts.

3.1.2 Switch Bouncing

Switch contacts are usually made of springy metals that are forced into contact by an actuator. When the contacts strike together, their momentum and elasticity act together, causing a bounce phenomena. The result is rapidly sequenced electrical pulses instead of a clean transition from 0 to a logical 1 as we can see in Figure 3.1. The problem may occur in switch closures and openings. The maximum time taken by the contacts in a switch to reach the steady state, is called switch bounce time.

Bouncing causes that a single switch throw be interpreted as multiple operations, causing, in many cases, incorrect system operation particulary when managed by

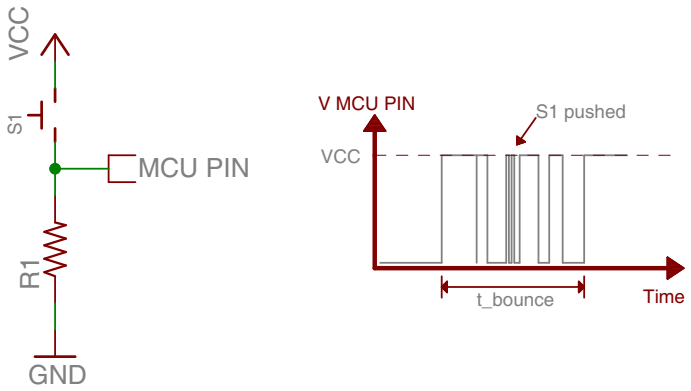


Figure 3.1: Bouncy behavior of a mechanical switch (*Source*: Introduction to Embedded Systems, M. Jiménez, R. Palomera, I. Couvertier)

interrupts. However, even when a polling technique is used to read the switch, the process might be affected by bouncing if the polling interval were shorter than the bounce time. To try to avoid or reduce this problem, hardware or software techniques can be implemented.

A **hardware technique** is implemented through the insertion of circuit components such as filters or some form of digital delay to suppress the transient pulses. The implementation of these techniques depends mainly on the type of switch used and the characteristics of the application. The most commonly used techniques include:

- An SR debouncing circuit: Consists of a set-reset (SR) latch between the switch and the digital pin.
- An RC debouncing circuit: Is a cost-effective solution that consists of a Resistor-capacitance network to implement a delay in the switch line.
- An IC debouncer circuit: Consists of a commercial integrated-circuit (IC) such as the MC14490 (Hex contact bounce eliminator). This IC contains six independent debouncing circuits for an equal number of switches.

Software techniques are implemented through the use of extra code lines (subroutines) instead of external components. This code uses CPU cycles to remove the bouncy portion from the switch signal. The most common used techniques include:

- A polling debouncer: This is a simple technique that polls the switch port with a constant polling period longer than the expected switch bouncing time.
- A counter debouncer: This technique consists of assuming the contacts have settled if they have not bounced for a certain number of samples.

See class's book Section 8.3 (Interfacing Switches and Switch Arrays) for a detailed and deep explanation about the bounce phenomena in switches and the techniques used to minimize its effects.

3.2 Basic Exercises

3.2.1 Read a Key Using Interrupts

Read the input state of a pin through an interrupt service routine. The interrupt has to increment the value of an internal variable. This value must be displayed constantly on the LCD screen.

Follow the steps outlined below:

1. Use the set-up developed in Experiment 2 that connects an LCD screen and a switch to build your circuit.
2. Verify if the I/O port, where the switch is connected, has interrupt capabilities. If not, move the switch to an interrupt capable input port.
3. Open your IDE.
4. Create a main program to setup and initialize the stack pointer (if necessary).
5. Write the code for your ISR. The ISR only needs to increment the value of a (global) variable each time it is executed. Remember to make your ISR register transparent. You can use the flowchart in Figure 3.2 as a guide to writing your code.
6. Identify which entry in the MCU jump table corresponds to the port where the switch is connected. Use the label of your ISR filling the interrupt table entry.
7. Insert instructions in your main program to enable interrupts: To do so, first, clear the interrupt flag, enable the PORT interrupt, and then enable the CPU global interrupts.
8. Modify your main program so that every time the global variable is incremented, it's value is displayed on the LCD. You can use the flowchart in Figure 3.3 as a guide to write your code.
9. Compile your code and verify that it does not contain errors.

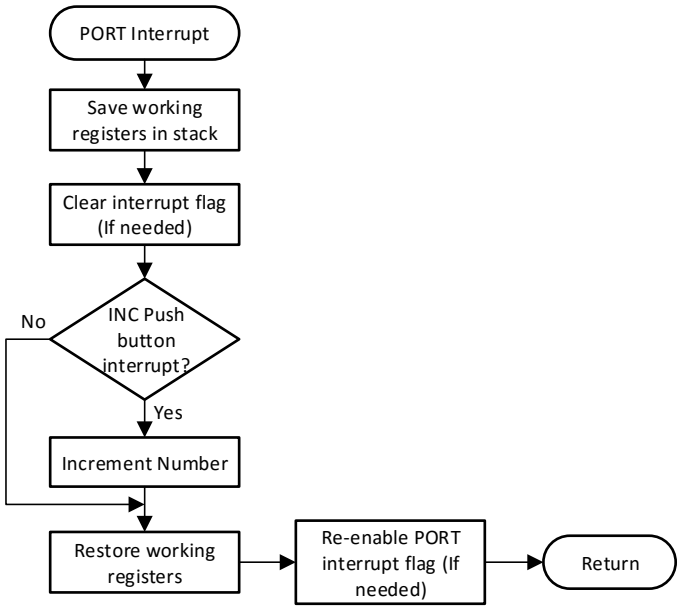


Figure 3.2: ISR flowchart

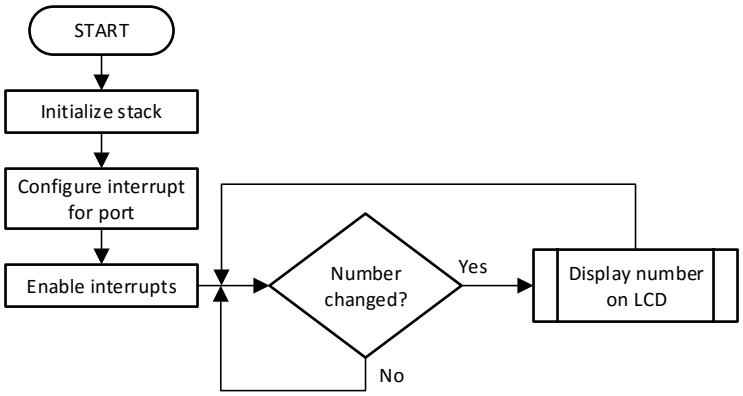


Figure 3.3: Flowchart for the main loop in read a key using interrupts

10. Run your code and verify if the number is incremented when you depress the pushbutton.

3.2.2 Hardware Debouncing

Read the input state of a pin using a hardware debouncing technique.

Follow the steps outlined below:

1. Assemble the circuit shown in Figure 3.4, with R1=2.7KΩ, R2=3.3KΩ and C1=4.7uF. These values were chosen assuming that the push button used, has a bouncing period of about 20ms. Take into account that the inverter buffer has a Schmitt triggered input. If your MCU has Schmitt triggers in its input ports, the inverter is not needed; otherwise, provide one externally. See class's book Section 8.3.7 (Hardware Debouncing Techniques) for a detailed explanation in how calculate the resistors and capacitors values for the RC debouncing circuit.

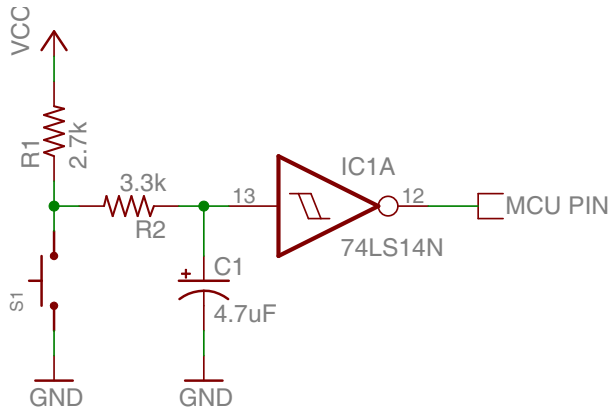


Figure 3.4: Schematic for hardware debouncing circuit

2. Replace the switch used in the previous exercise with the circuit shown in step 1.

3. Open your IDE and run the code created in the previous exercise.

4. Verify if the bounce effect of the switch disappears. If not, determine new values for R1 and R2 using a larger capacitor C1.

3.2.3 Software Debouncing

Read the input state of a pin using a software debouncing technique.

Follow the steps outlined below:

1. Restore to the circuit used in the first Basic Exercise (Pushbutton without hardware debouncing).

2. Open your IDE.

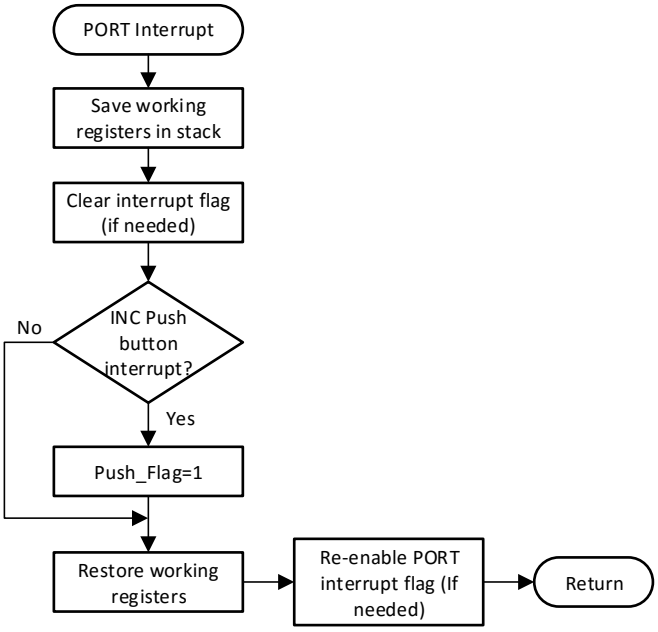


Figure 3.5: Software debouncing ISR flowchart

3. Write a code for your ISR to set a flag when an interrupt is generated. You can use the flowchart in Figure 3.5 as a guide.

4. Modify your main program so that when it confirms that the Push_Flag is set, it waits 30ms and then resets the Push_Flag. Finally, the program has to increment the value and display it on the LCD. You can use the flowchart in Figure 3.6 as a guide to write your code. This routine can be optimized using a timer interrupt to count the 30ms. You will learn this technique in the next experiment.

5. Compile your code and verify that it does not contain errors.

6. Run your code and verify that the number is incremented when you depress the pushbutton.

3.2.4 Reading Keypads Through Interrupts

Read the input state of a keypad using a scan algorithm and display it on the LCD. Follow the steps outlined below:

1. Identify the ports and pins of your MCU that will be used to connect to the

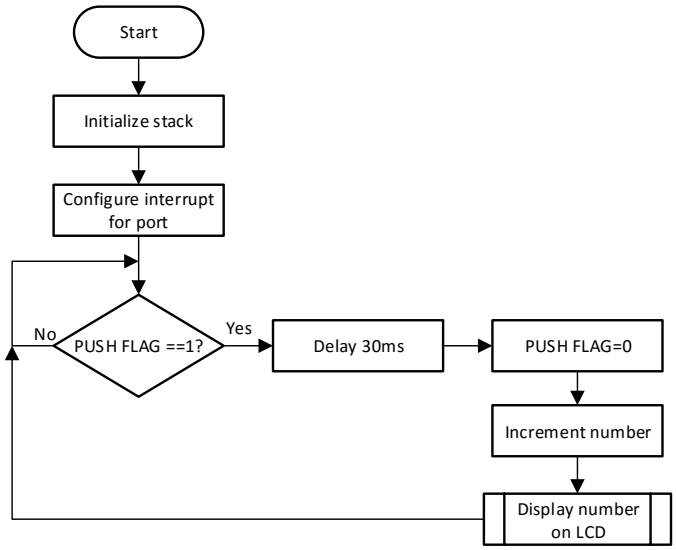


Figure 3.6: Flowchart for the main loop in software debouncing

keypad. Be sure that the pins used in the columns (inputs) have interrupt capabilities. If not, move the connections to an interrupt capable input port.

2. Connect the keypad according to the block diagram shown in Figure 3.7. You can configure and use the pull-down resistors of your MCU instead of the external resistors (You have to verify if your MCU has this capability).

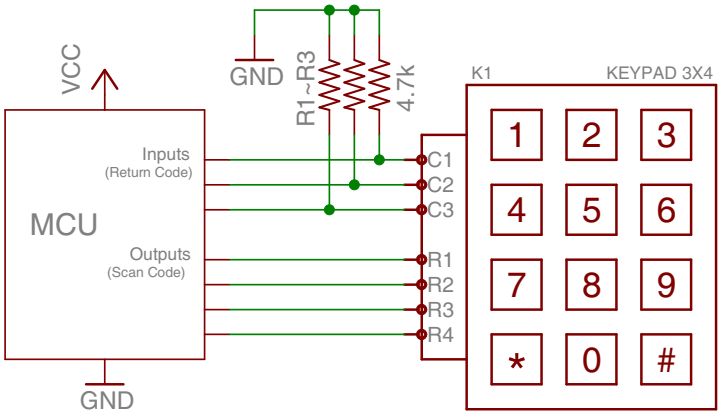


Figure 3.7: Block diagram for keypad connection

3. Open your IDE.

4. Set the pins connected to the rows as outputs and the pins connected to the columns as inputs.

5. Write a code for your ISR which once a key has been depressed performs the scanning keypad algorithm to read the key. You can use the flowchart in Figure 3.8 to write the scanning algorithm. Read the note section at the end of this exercise to have a better understanding on how the scanning method works.

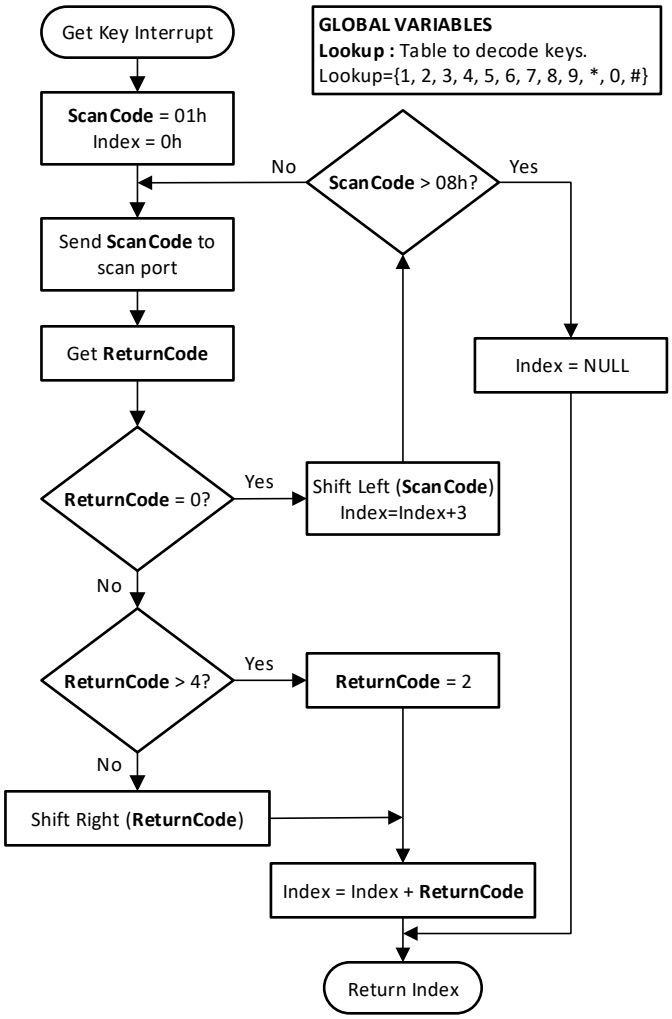


Figure 3.8: Read keypad Flowchart (*Source*: Introduction to Embedded Systems, M. Jiménez, R. Palomera, I. Couvertier)

6. Insert instructions in your main program to enable interrupts: PORT interrupt and CPU global interrupts.

7. Initialize the port configured as output to have its lines in high. This procedure will ensure the activation of the interrupts independently of the key depressed. Be sure to maintain the output pins in high once the interrupt have been served.

8. Write a main code to display the keys obtained from the interrupt on the LCD. Each new number has to be displayed on the LCD without erasing the previous number.

9. Write a function for the "$*$" key. This function has to clear the LCD.

10. Write a function for the "#" key. This function has to change the line in the LCD in which the numbers are being displayed.

11. Compile your code and verify that it does not contain errors.

12. Run your code and verify the functionality of your keypad.

Note: The scanning method for keypads consists in setting a scan code with a "Logic high" to the row (Rx) we want to scan and "logic low" the rest of the rows (ScanCode). All columns are read at once (ReturnCode). If a logic 1 is detected in a particular column line (Cy) it means that the key in position (Rx, Cy) is depressed. At the end of the cycle, the returned "Index" will contain a binary number between 00h and 0Bh that represent the key depressed in the keypad. Where the binary numbers represent the keys in the order "123456789*0#". The "Null" character is assigned by the user. These steps are sequentially repeated for each column until the entire keypad is scanned. If several keystrokes are expected, the scanning algorithm is executed in a loop until all keystrokes are received. This explanation assumes pull-down resistors are connected in the return lines. See class's book Section 8.3.11 (Interfacing Keypads) for a detailed explanation about the scanning algorithm.

3.3 Complementary Tasks

3.3.1 Scrolling List With Wheel

The activity consists of generating a scrolling list of messages and display them on the LCD (similar to the Experiment 2 but using a scrolling wheel instead of switches). To detect the movement direction, build an encoder wheel (an optical encoder can convert the angular movement and direction of the wheel into a set of digital pulses). For the wheel encoding pattern use the diagram shown in the Figure

3.11 and affix the pattern to one side of the wheel. Mount the wheel in a base as shown in Figure 3.9. Connect two opto-switches aligned vertically with the two sets of marks in the wheel. You can use the schematic in Figure 3.10 as a reference to connect the opto-switches. Also, investigate how to read a quadrate encoder using a look up table technique or read the Appendix A at the end of the document.

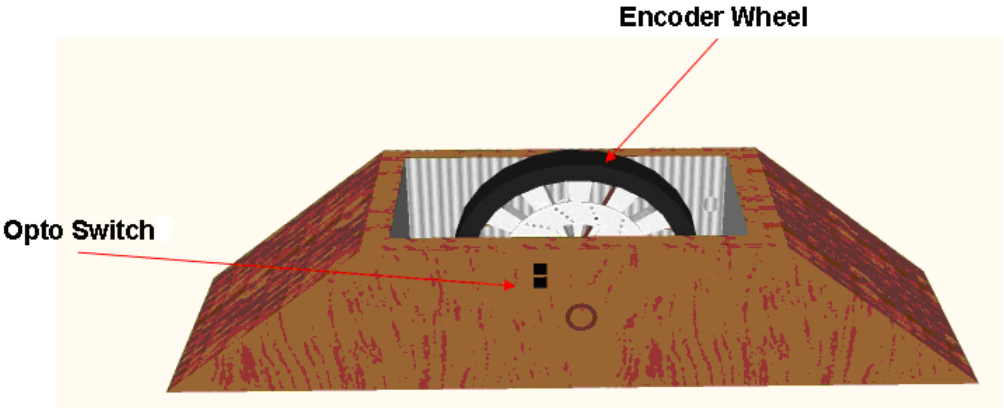


Figure 3.9: Suggested mounting base for encoding wheel

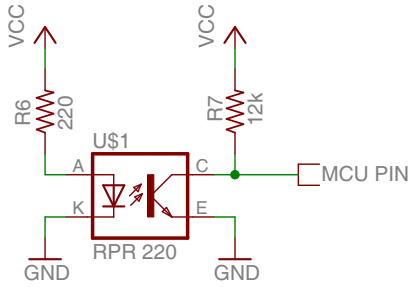


Figure 3.10: Schematic for hardware debouncing circuit

Presentation and Report

Each basic exercise and complementary task must be presented to the TA before the initiation of the next laboratory experiment. The demonstration must be made personally in the lab and including the hardware and software components needed for its completion.

An electronic report that includes the following information about the complementary task must be presented to the TA:

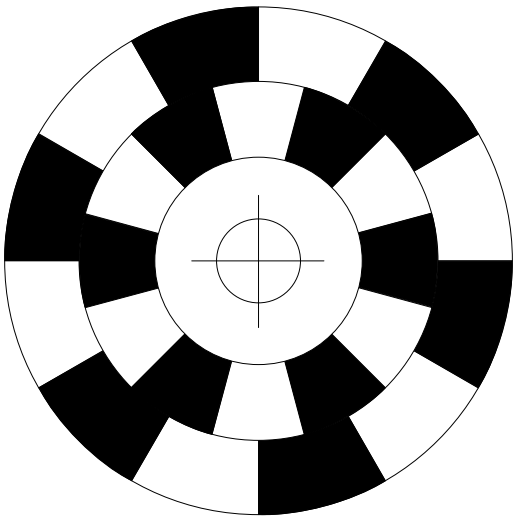


Figure 3.11: Layout of quadrature encoding patterns to be attached to the wheel

- Software plan and explanation (pseudocode or flowchart)
- Connection schematic with relevant component calculations (current, voltage, timing values, etc.)
- Code listing with comments.
- Additional information used to complete the task (Web pages, datasheets, books, etc.)

Experiment 4

Timers and LEDs

Objectives

- Understanding the uses of timers in embedded applications
- Identifying and understand timer architectures and operating modes
- Configuring and using the timer modules
- Interfacing 7-segment displays to microcontrollers
- Implementing software techniques to display information in 7-segment displays modules

Duration

- 2 Hours in the laboratory and extra time for complementary tasks

Materials

Table 4.1: Bill of materials for completing Lab. 4

Item #	Qty	Description	Reference
1	1	Development board	
2	1	IDE application	
3	1	Dual 7-Segment Common Anode	DA56-11EWA
5	10	1/4W Carbon fill resistor	220 Ω
6	3	1/4W Carbon fill resistor	1 KΩ
7	1	1/4W Carbon fill resistor	4.7 KΩ
8	2	1/4W Carbon fill resistor	12 KΩ
9	1	Momentary switch	Pusbutton
10	2	Optoswitch	RPR-220

Table 4.1: Continued

11	1	Piezoelectric buzzer	Buzzer
12	2	BJT PNP transistor	2N3906

4.1 Introduction

4.1.1 Timers

In its most basic form, a timer is a counter driven by a know clock signal that increase its count with each clock cycle. When the count reaches its maximum value $(2^n - 1)$ the timer generates an overflow signal and restarts counting at 0. The overflow signal can be polled by software or used to trigger an interrupt request (Timer Overflow). The Figure 4.1 shows the basic structure of a timer. Timers are important in MCU systems because they can be used to: implement time-bases, count time between events, an to develop real-time clocks, watchdog timers, pulse-width modulators, and baud rate generation, among many others applications. Microcontrollers may include one or more configurable timer modules among their peripherals.

Timer modules can be found in different number of bits such as 8-, 16-, 24-bit, etc. The number of bits in the timer's counter determines the maximum value it can count to, i.e., the maximum value the timer count register can hold.

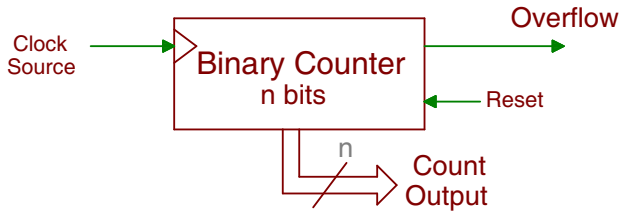


Figure 4.1: Timer overflow structure

As CPU clock frequencies are considered high for most practical applications, timers also include a pre-scaler. A pre-scaler is just a chain of flip-flops that can divide the source clock frequency by values specified through a configuration register. Most typical values are 1, 2, 4, and 8, although this might change from one timer to another. Pre-scalers are useful when you need to extend the length of time between timer overflows.

Another important part of a timer module is the terminal count register. This register, typically of n bits as the timer's binary count, can be loaded with any value $\leq 2^{n-1}$. When the timer's Binary Counter reaches *Compare Register*, a reset signal

is generate that restart the binary counter and a *Top* count signal is generated. The *Top* signal can be polled by software or configured to generate an interrupt request. This is a versatile feature that can be used for many applications, such as generating periodic signals or pulses of predetermined width. Usually, timer modules include a clock multiplexer that allows to select a clock source from a set of internal or external clock sources through a selector signals (clock source selector). Figure 4.2 shows a complete timer block diagram with both, pre-scaler and terminal count registers.

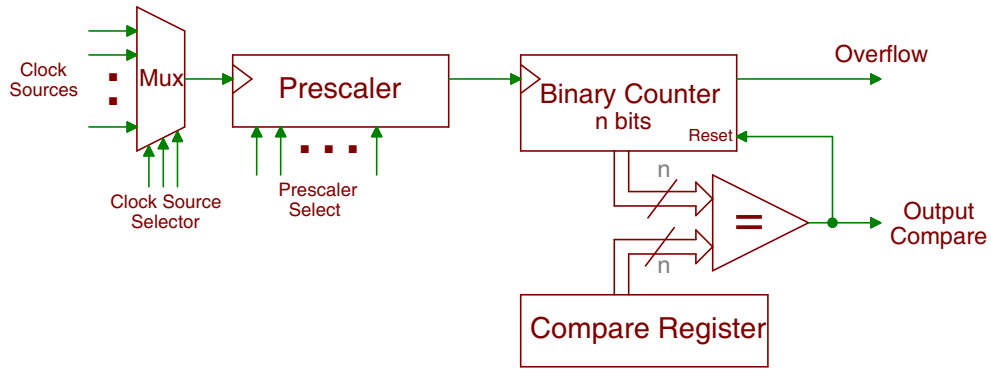


Figure 4.2: Timer basic structure (*Source*: Introduction to Embedded Systems, M. Jiménez, R. Palomera, I. Couvertier)

Timers have two basic modes of operations used in the majority of application: **Event counter** and **Interval timer**. When operated as an event counter, a timer simply counts the number of events it detects in its clock input. This clock signal does not necessarily have a periodic behavior. But, when the input clock signal is periodic with a frequency f, the timer can be used to measure time intervals between two events See class's book Section 7.4.2 (Fundamental Operation: Interval Timer vs. Event Counter) for a detailed explanation. Furthermore, if the frequency is f Hz, then the period will be:

$$PERIOD \qquad T = \frac{1}{f} seconds \tag{4.1}$$

Therefore, when the counter shows k pulses, it has registered a duration of $kT = k/f$ seconds.

4.2 Basic Exercises

4.2.1 Timer by Polling

Produce an audible sound using delays generated by polling the timer's Top count flag. Read the note section at the end of this exercise to understand other timer architectures.

Follow the steps outlined below:

1. Connect the buzzer according to the schematic shown in Figure 4.3. Choose R to not exceed your MCU's pin current capacity.

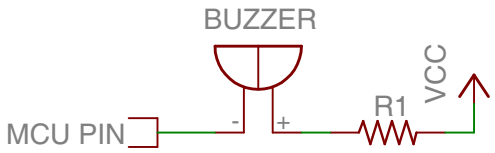


Figure 4.3: Schematic for buzzer connection

2. Open your IDE.
3. First, calculate the compare register value that produces a delay to generate an audible frequency (f=1000Hz). Take into account in your calculation that the signal to be produced must have a duty cycle of 50%.
4. Look into the architecture of your MCU Timer to determine the appropriate timer operation mode to generate the above frequency, and produce a code to configure the timer in that mode.
5. Next, load the compare register value calculated into the timer's configuration registers required.
6. Write a code that continuously is checking the Top count flag and when it is detected, toggle the buzzer pin. You can use the flowchart in the Figure 4.4 as a guide to write your code.
7. Compile your code and verify that it does not contain any errors.
8. Run your code and verify if the buzzer produces an audible sound.
9. After the completion of Table 4.2, modify your code to produce the frequencies listed on the table. To change between the frequencies use a pushbutton connected to the MCU.

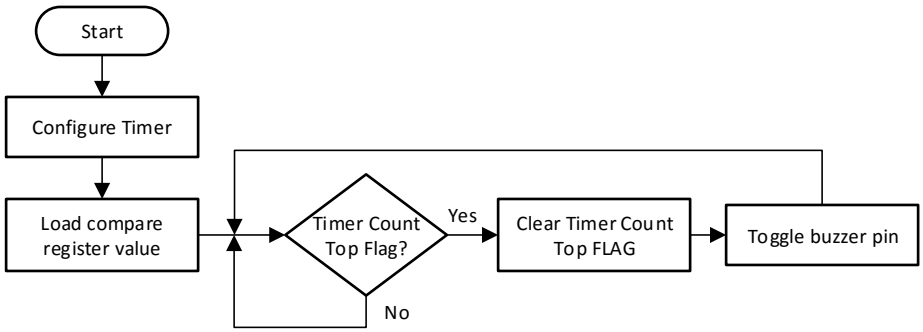


Figure 4.4: Timer by polling flowchart

Complete Table 4.2. Specify the clock period, number of timer bits, and the prescaler currently used.

Table 4.2: Timer MCU values

Clock Frequency:			
Timer's Bits:			
Preescaler used:			
Frequency	Period	Period/2	Compare register value nedded
500 Hz			
1 KHz			
1.5 KHz			
2 KHz			
3 KHz			

Note: The timer architecture presented is commonly used in MSP430 and ARM MCU architectures but other types of MCUs possess different timer architectures. These other architectures allows to pre-load a initial value in the timer's binary count that modify the quantity of clock cycles needed to reach it maximum value and restart in instead of using a value in the compare register to generate a reset in the timer.

4.2.2 Timer by Interrupt

Produce an audible sound using delays generated by the timer's interrupt.

Follow the steps outlined below:

1. Use the same setup as the basic exercise developed before "Timer By Poling".

2. Open your IDE.

3. First, calculate the terminal count value that produces a delay to generate an audible frequency (f=1000Hz). Take into account in your calculation that the signal to be produced must have a duty cycle of 50%.

4. Configure the timer mode necessary to generate the frequency calculate before and load the terminal count value calculated.

5. Write a timer ISR code to generate the audible frequency on the buzzer. You can use the flowchart in the Figure 4.5(a) as a guide to write your code.

6. Next, enable timer and global interrupts.

7. Compile your code and verify that it does not contain any errors.

8. Run your code and verify if the buzzer produces a sound.

9. Now, modify your code to produce the frequencies listed in Table 4.2. To change between the frequencies, use a pushbutton connected to the MCU.

4.2.3 7-Segment Display

Generate a counter from 0 to F displaying the number on a 7-segment display.

Follow the steps outlined below:

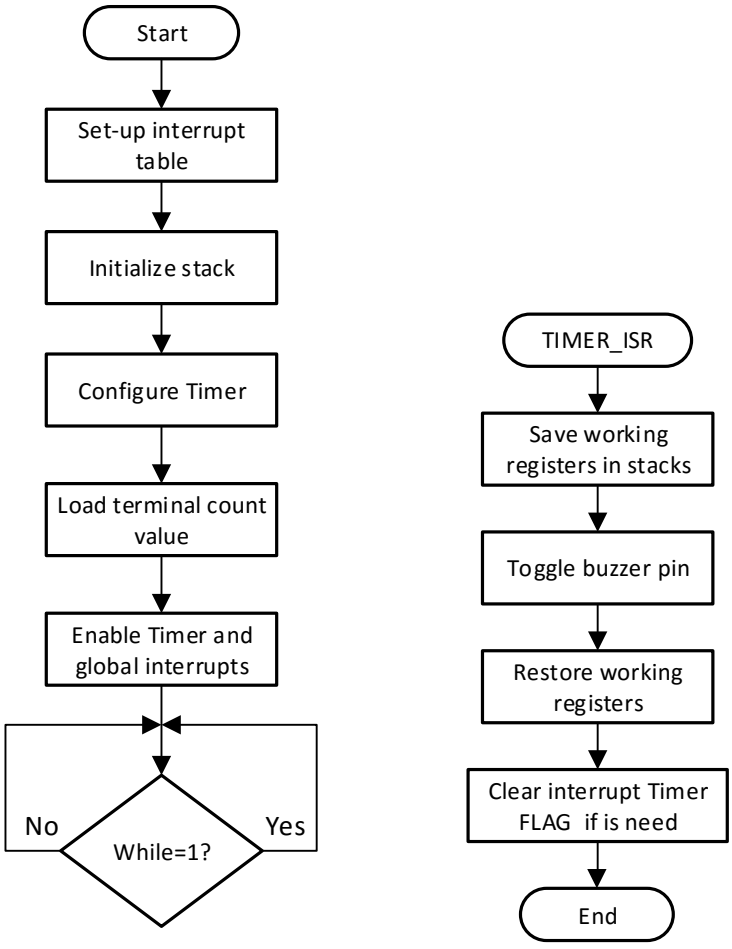
1. Connect a 7-segment display according to the schematic shown in Figure 4.6.

2. Calculate the value of the series resistor to limit the current through the segment to not overload your MCU I/O pin and satisfy the 7-segment requirements. Also, calculate the transistor base resistor to ensure it would saturate with the maximum 7-segment current.

3. Make a table to decode the digits from 0 to F into 7-segment codes. Remember, that if you are using a common anode 7-segment, a logic low value is needed on the data line to power-on the segment. Complete the table 4.3 using as reference the 7-segment labels shown in Figure 4.7.

4. Open your IDE.

5. Make a look-up table with the data completed in Table 4.3.



(a) Main Program (b) Service Routine (ISR)

Figure 4.5: Timer by polling flowcharts

6. Write a program that sends the appropriate code to the 7-segment port to display the digits between 0 and 9 every 1 second. Use a timer ISR to produce a delay time of 1 second and increment the value that will count the numbers. Use the pseudocode listed below as a guide to your main code:

Listing 4.1: 7-segment display Pseudocode

```
1 ;-------------------------------------------------------
2 ;    Program Start
3 ;    INIT RESET VECTOR
4 ;    INIT STACK POINT, WDT
5 ;-------------------------------------------------------
6 lookup = C0h, F9h,....
```

```
7  Number = 0
8
9  While   TRUE
10     Port = @lookup + number
11     Pin_control = 0
12 Endwhile
13 ;-------------------------------------------------------
```

7. Compile your code and verify that it does not contain any errors.

8. Run your code and verify if the numbers in the 7-segments are appearing.

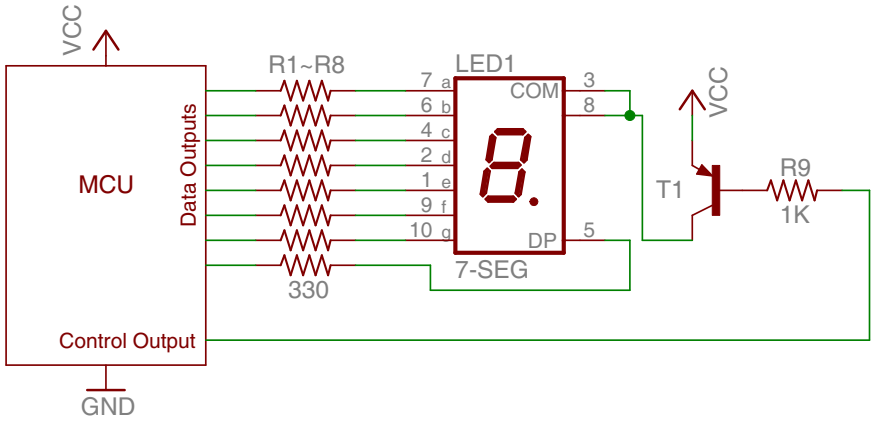


Figure 4.6: Schematic for 7-segment

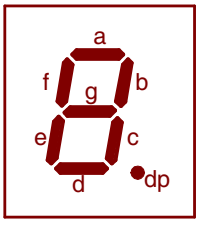


Figure 4.7: Segments names

4.2.4 Multiplexed Display Using a dual 7-segment display

The objective of this section is to generate a counter from 00 to FF using dynamic display techniques. Read the note section at the end of this exercise to have a better understanding on how the dynamic display works and its implications.

Table 4.3: Codes for 7-segment display of digits from 0 to F

#	dp	g	f	e	d	c	b	a	7-seg
0	1	1	0	0	0	0	0	0	C0h
1	1	1	1	1	1	0	0	1	F9h
2									
3									
4									
5									
6									
7									
8									
9									
A									
B									
C									
D									
E									
F									

Follow the steps outlined below:

1. Connect two 7-segment displays according to the schematic shown in Figure 4.8 taking into account the electrical consideration explained before for a 7-segment connection.

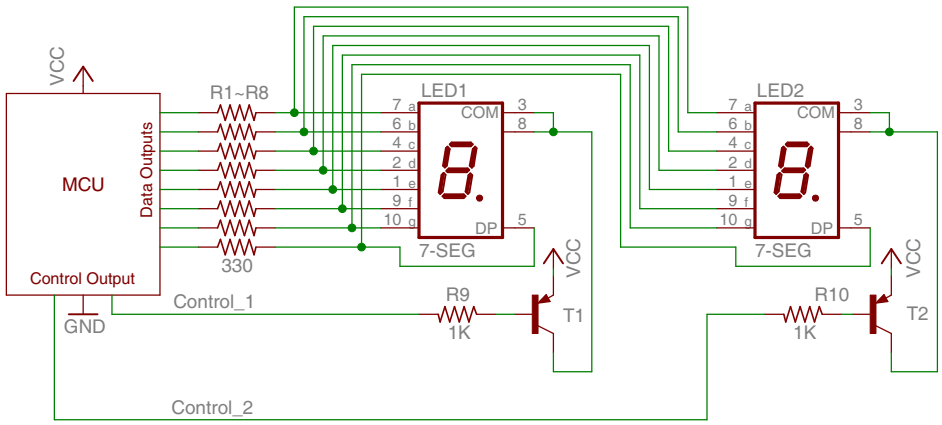


Figure 4.8: Schematic two 7-segments displays

2. Open your IDE.

3. Make a code to implement a dynamic visualization technique to display the two numbers into the 7-segment displays. Use the following steps and pseudocode to produce the your main code:

   (a) Turn off the control signal for both 7-segments

   (b) Send the data to appear in the first 7-segment

   (c) Turn on the first 7-segment control signal

   (d) Delay loop

   (e) Turn off the control signal for both 7-segment

   (f) Send the data to appear in the second 7-segment

   (g) Turn on the second 7-segment control signal

   (h) Delay loop

   (i) Back to step a

Listing 4.2: Dynamic display Pseudocode

```
;-------------------------------------------------------
;   Program Start
;   INIT RESET VECTOR
;   INIT STACK POINT, WDT
;-------------------------------------------------------
lookup = 2Fh, 06h,....
Num_7Seg1=0
Num_7Seg2=0

While   TRUE
    Pin_control_1 = Pin_control_2  = 0  ;turn off both displays
    Port = @lookup + Num_7Seg1          ;obtain 7-seg code for
                                        ;upper digit
    Pin_control_1 = 1                   ;turn on display
    call delay_loop                     ;delay loop

    Pin_control_1 = 0                   ;turn off display
    Port = @lookup + Num_7Seg2          ;obtain 7-seg code for
                                        ;lower digit
    Pin_control_2 = 1                   ;turn of display
    call delay_loop                     ;delay loop
Endwhile
;-------------------------------------------------------
```

4. Implement the delay loop using a timer ISR to refresh the two 7-segments with a refresh rate of 60Hz. The number to be displayed in the 7-segments must be increased every 1 second.

5. Compile your code and verify that it does not contain any errors.
6. Run your code and verify if the numbers in the 7-segments are appearing.

Note: The dynamic display technique is a software technique that allows controlling one or more 7-segment displays at time using the same segment lines for all the display. Through a constantly blinking process in each display, the technique generate a visual effect that allows to see all the displays in On at the same time. This technique is commonly used in display applications because reduce the amount if MCU pins to control serval display quantities. A drawback of this method is the loss of brightness in the display due the display is not in On all the time.

4.3 Complementary Tasks

4.3.1 Digital Tachometer

The activity consists of an implementation of a tachometer for the encoder wheel built in Experiment 3. Use the LCD for displaying in the first line a message with the speed: "Speed=####RPM" (Four units to represent the speed value must be used). In the second line indicate whether the rotation direction is clockwise or counterclockwise. Internally, use the timer and interrupts to determine the speed of rotation of the wheel and its direction.

Presentation and Report

Each basic exercise and complementary task must be presented to the TA before the initiation of the next laboratory experiment. The demonstration must be made personally in the lab and including the hardware and software components needed for its completion.

An electronic report that includes the following information about the complementary task must be presented to the TA:

- Software plan and explanation (pseudocode or flowchart)
- Connection schematic with relevant component calculations (current, voltage, timing values, etc.)
- Code listing with comments.

- Additional information used to complete the task (Web pages, datasheets, books, etc.)

Experiment 5

Low-Power Modes and PWM

Objectives

- Understanding how low-power modes help to reduce the energy consumption of embedded system
- Using low-power modes to improve the power performance of an embedded application
- Identifying and understanding PWM architectures and operating modes
- Using a PWM module to control electronic devices such as LEDs

Duration

- 2 Hours in the laboratory and extra time for complementary tasks

Materials

Table 5.1: Bill of materials for completing Lab. 5

Item #	Qty	Description	Reference
1	1	Development board	
2	1	IDE application	
3	1	LCD display: 2 lines, 16 characters	W/HD 44780 Controller
4	1	Light Emitting Diode	5mm Red LED
5	1	Light Emitting Diode	5mm RGB LED
6	1	1/4W Carbon fill resistor	220 Ω
7	3	1/4W Carbon fill resistor	510 Ω
8	4	1/4W Carbon fill resistor	4.7 KΩ
9	1	Momentary switch	Pusbutton
10	1	3 columns by 4 rows Buttons array	Keypad 3x4
11	1	Multimeter	Fluke 179

5.1 Introduction

5.1.1 Low-Power Modes

Power consumption in embedded systems is an important design factor that affects a wide range of aspects, from battery life in portable applications to issues such as reliability, cost, size, and environmental impact. Therefore, using MCUs with low-power modes and learning how and when to activate and use those modes becomes of utmost importance in the design of embedded systems.

The activation and use of low-power modes in a microcontroller unit involves minimization of the individual current consumption of its internal peripherals, minimization of the CPU activity, and optimization of the code running during active periods. Depending on the particular MCU, the activation of a low-power mode can involve: disabling the CPU by turning it off or sending it to a standby mode, reducing the CPU clock frequency and, changing the clock source, among other strategies.

An effective way of incorporating low-power modes into an application is by configuring the code running on the CPU to operate using interrupts. Figure 5.1 shows a flowchart of a program that uses low-power modes. Basically, the program starts with the initialization of peripherals and system components, continues enabling interrupts of expected events and finally sends the CPU into a sleep mode. Every time an enabled interrupt is triggered, it will wake-up the CPU to serve the event and go back to sleep. This procedure saves more energy because instead of continuously polling for the expected event to occur, the CPU is only active when the interrupts mark the event that needs to be served.

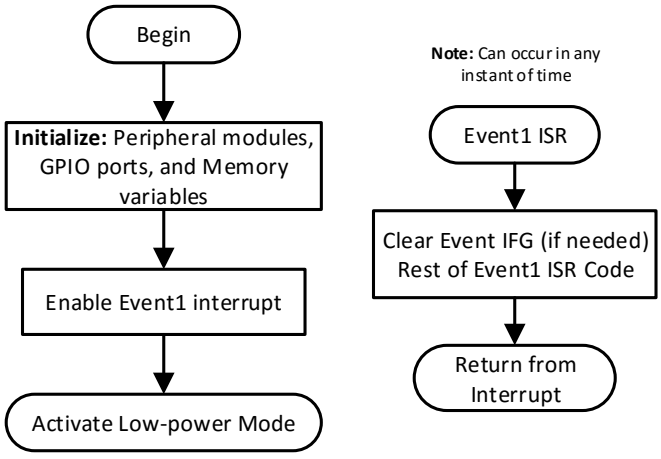


Figure 5.1: Main Program using low-power mode and a single event ISR

Low-power modes are of utmost importance in battery-powered applications due to the dramatic reduction in power consumption they induce. As illustrated in Figure 5.2, depending on the low-power mode activated, the system could achieve different levels of power consumption.

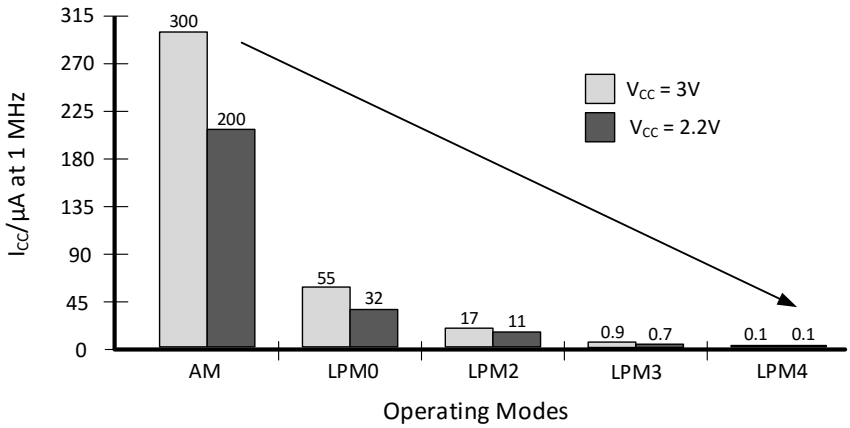


Figure 5.2: Current Consumption of MSP430F21x1 devices in different operating modes (*Source*: Introduction to Embedded Systems, M. Jiménez, R. Palomera, I. Couvertier)

5.1.2 Pulse Width Modulation

Pulse Width Modulation (PWM) is another useful timer application in embedded systems. A PWM module produces a square wave signal with a predefined and controlled duty cycle, allowing the generation of a signal with different pulse widths in different time periods as we can see in Figure 5.3.

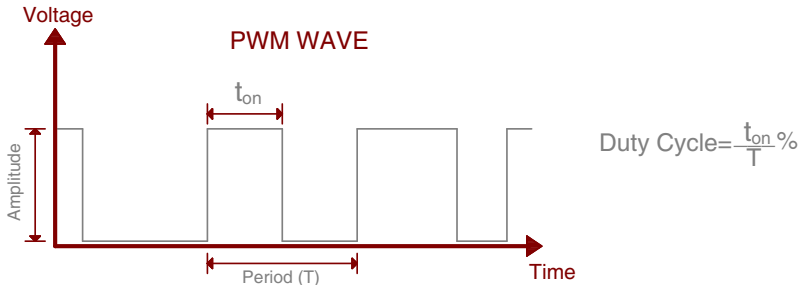


Figure 5.3: PWM signal parameters

A basic PWM module structure is shown in Figure 5.4. This structure is similar to the basic timer structure presented in the previous lab, with the difference

that a PWM module uses a High Count register together with an n-bit hardware comparator instead of the compare register to generate the square wave signal. It fundamentally contains an n-bit timer (with clock selector and Prescaler), whose count is compared in hardware to the contents of a "high-count register" (*hc*). Figure 5.5 illustrate the behavior of a PWM module in which, while the timer has a value less than *hc* the PWM output is high, otherwise, the output is low. This mode of operation depends basically on the PWM module architecture. Some MCUs incorporate an enhanced PWM architecture to allow the user to use different modes of operation and interrupt sources.

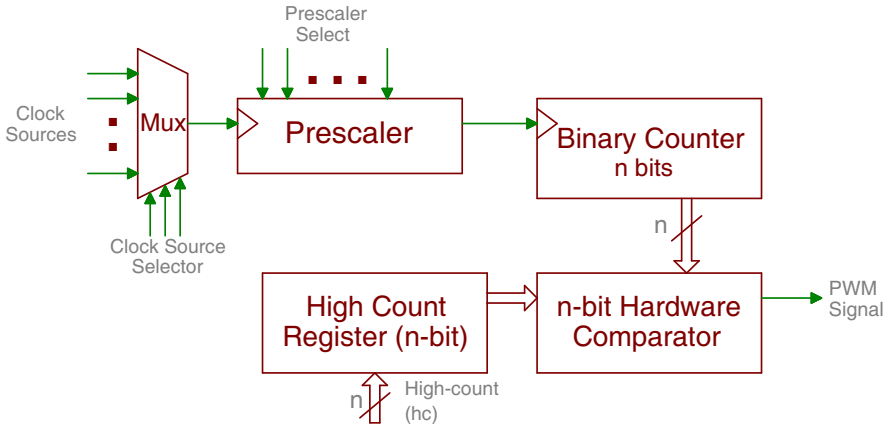


Figure 5.4: PWM basic architecture (*Source*: Introduction to Embedded Systems, M. Jiménez, R. Palomera, I. Couvertier)

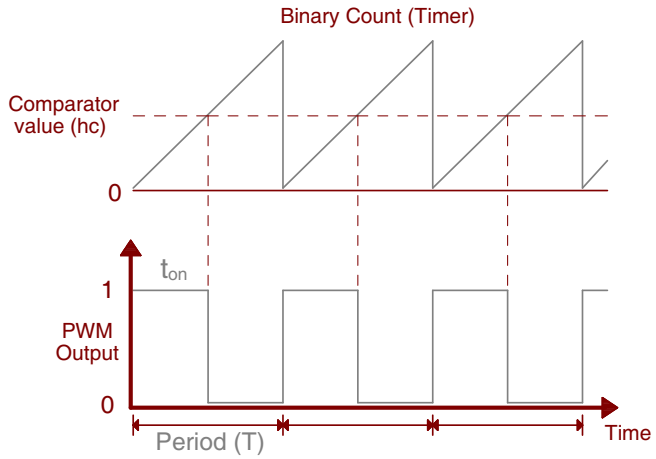


Figure 5.5: PWM signal parameters

Applications like DC motor speed, heater's temperature, light intensity in LEDs,

and even musical tones can be implemented with PWM. These many applications make PWM modules a useful addition to the list of MCU peripherals. See class's book Section 7.4.3 (Signature Timer Applications) for a detailed explanation about the PWM architecture and applications.

5.2 Basic Exercises

5.2.1 Low-Power Modes

Compare the low-power current consumption of your MCU versus that in normal mode.

Follow the steps outlined below:

1. Assemble the setup used for the scrolling list developed in Experiment 2 to scroll messages in an LCD display using two keys.

2. Load the software version of your code that reads the switches by polling. Make sure that your development board is only connected to the host computer through the programming adapter, with no other external power supply connected to it through the downloading process.

3. Remove the programming adapter and all connections to the host computer. Your program will stay safely in the MCU flash memory.

4. Connect your development board with an external power supply allowing the connection of an amp-meter to measure the MCU's current consumption. Connect the LCD to the power supply such that its current does not pass through the MCU amp-meter. Make sure the multi-meter is in current mode and the leads are connected to the amp-meter inputs. Use the block diagram shown in Figure 5.6 as a guide to make the necessary connections.

5. Power-Up your development board and LCD, then measure the MCU current several time and take the average value.

$$Average\ I_{CC}1: \_\_\_\_\_$$

6. Scroll the list up/down a few times to see the average load current measured by the amp-meter.

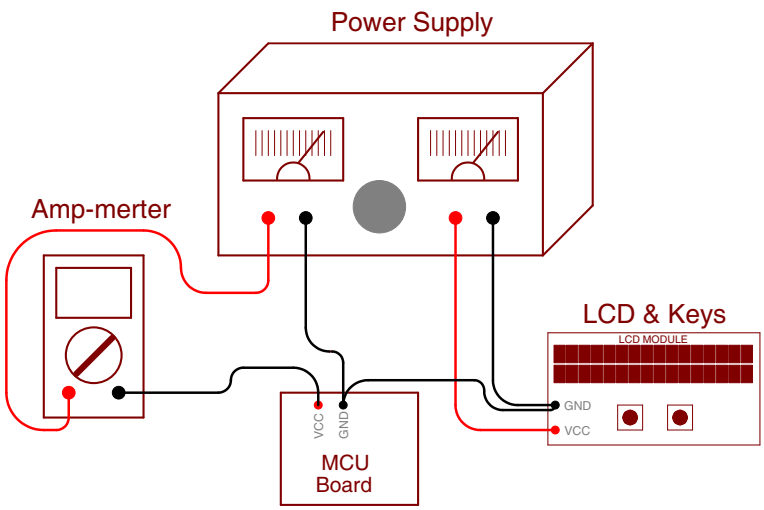


Figure 5.6: Connection diagram for MCU with power supply and amp-meter

7. Turn the power supply off, remove the power supply cables from the board and re-attach the programming adapter.

8. Edit your code to serve the keys by interrupts. Also, modify the main program to make the MCU enter a low-power mode, right after the system set-up is completed. Remember to enable global and peripheral interrupts.

9. Download the code to your system and with the programming adapter still connected (do not connect your MCU to the power supply yet!), debug your code and make sure it works as expected.

10. Once you have a working version of your code, remove the programming adapter and re-connect the external power supply to the boards, making sure to connect the multimeter as illustrate in Figure 5.6.

11. Measure the MCU's current consumption using the amp-meter (again, make multiple measurements and take the AVG).

$$Average\ I_{CC}2: \_\_\_\_\_$$

12. Compare the two current values measured, did you notice any change between the two measurement? For the two current values measured calculate the expected battery life for a 1500 mAh battery.

5.2.2 PWM Signal Generation

Produce a square wave signal using a PWM module.

Follow the steps outlined below:

1. Connect the oscilloscope to your MCU according to the schematic shown in Figure 5.7.

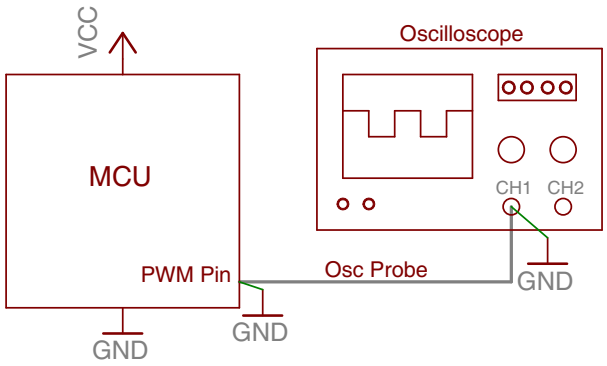


Figure 5.7: Block diagram for oscilloscope connection

2. Verify that the I/O port where the oscilloscope is connected to has PWM capabilities, if not, move the oscilloscope probe to an appropriate pin.

3. Open your IDE.

4. Identify the PWM module to be configured according to the pin previously selected and configure it to use one of the MCU's clock sources. Look for the register associated with the Period and Pulse width in your PWM module.

5. Calculate the terminal count value that produces a signal with frequency of 1000Hz, this value must be loaded into the period register. Also, calculate the value to be loaded in the pulse width register to produce a duty cycle of 50%.

6. Determine the appropriate operating mode of your PWM module, and configure the PWM registers necessary.

7. Next, enable your PWM module.

8. Compile your code and verify that it does not contain any errors.

9. Run your code and verify that a signal is visible on the oscilloscope.

10. Extract the waveform signal from the oscilloscope. Obtain the period and duty cycle of the signal on the oscilloscope.

11. After the completion of Table 5.2, modify your code to produce each frequency listed in the table. Use the oscilloscope to save each waveform observed. Do not forget measuring the duty cycle and period of each signal.

Complete the following Table 5.2. Specify the clock period and the count values for the PWM registers. Later, complete Table 5.3 with the values measured for each signal and calculate the % of error in the duty cycle.

Table 5.2: Timer MCU values

Frequency	Period (T)	Period register value	50% Duty cycle (DC) register value
500 Hz			
1 KHz			
2 KHz			
4 KHz			
8 KHz			

Table 5.3: Timer MCU values

Frequency	Measured T	Measured DC	% Error DC
500 Hz			
1 KHz			
2 KHz			
4 KHz			
8 KHz			

5.2.3 Generating colors with an RGB LED

The purpose of this section is generating different colors using an RGB LED with PWM signals.

Follow the steps outlined below:

1. Connect the RGB LED according to the schematic shown in Figure 5.8. Be sure that your MCU has three pins with PWM capabilities.

2. Calculate the series resistor value to obtain the maximum brightness possible of each LED color without overloading the pin's current.

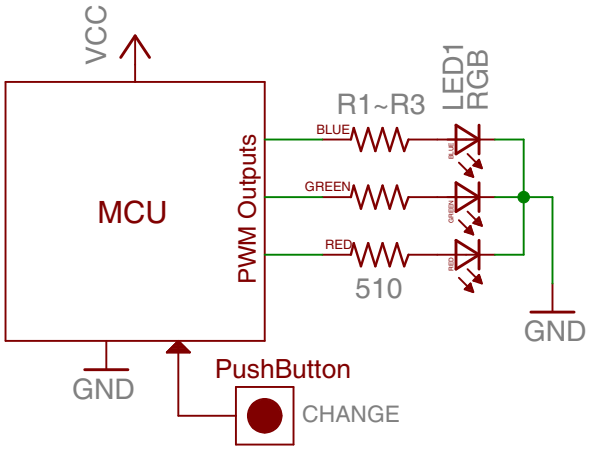


Figure 5.8: Schematic RGB LED Connection

3. Open your IDE.

4. Configure the PWM modules to produce the signals with a frequency of 1000Hz.

5. Make a look-up table with the duty cycle values of RED, BLUE, and GREEN corresponding to each color shown in Table 5.4. Take into account that a value of 255 in the color represents a 100% duty cycle for your output signal and a value of 0 represents 0% duty cycle.

Table 5.4: RGB Color Values

	R	G	B
1	0	0	255
2	0	255	0
3	255	0	0
4	255	30	217
5	30	222	252
6	240	200	40
7	255	123	33
8	255	255	255

6. Write an ISR code such that every time a pushbutton is depressed, changes the values in the PWM signals to produce a different color.

7. Enable the interrupt flags and PWM modules.

8. Now, write a main code to make the MCU enter a low-power mode right after the system set-up is completed and the interrupts have been enabled.

9. Compile your code and verify that it does not contain any errors.

10. Run your code and verify if the LED color changes when you depress the pushbutton.

5.3 Complementary Tasks

5.3.1 Digital Dimer

The activity consists of making a digital dimer for an LED. The system must allow changing the LED brightness from 0% (LED Turn-Off) to 100% (Maximum brightness), in increments of 10%, for a total of 11 levels of luminosity. The level is selected by the user through a keypad where each key corresponds to one level (e.g. $0 \rightarrow 0\%$, $1 \rightarrow 10\%$, $2 \rightarrow 20\%$, $\cdots$, $9 \rightarrow 90\%$, and $\# \rightarrow 100\%$) once the key is depressed. The LCD must display the current level selected by the user and its corresponding percentage of brightness. The LED must by driven by a PWM signal generated from the MCU. Use the block diagram shown in Figure 5.9 as a reference to connect your system.

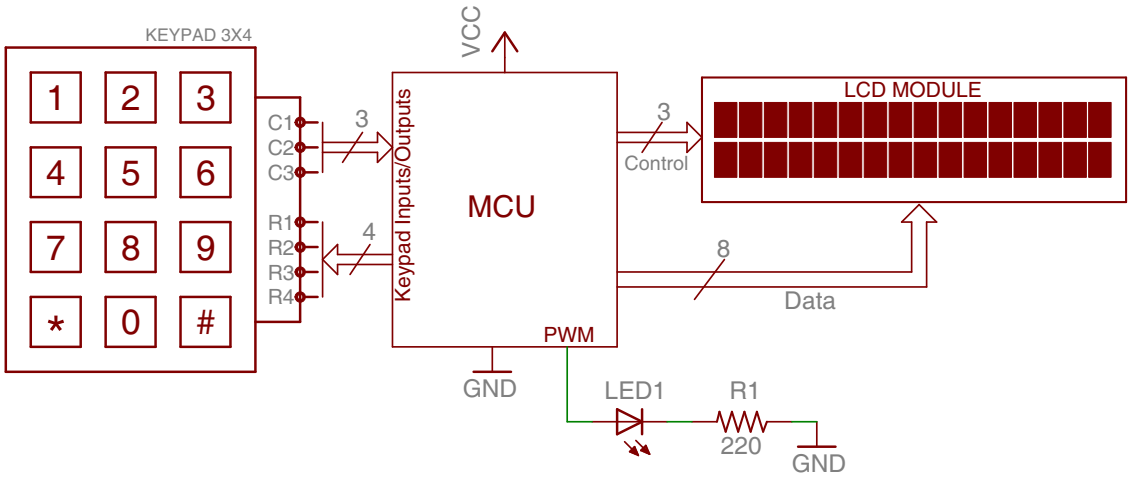


Figure 5.9: Digital dimmer connection diagram

Presentation and Report

Each basic exercise and complementary task must be presented to the TA before the initiation of the next laboratory experiment. The demonstration must be made personally in the lab and including the hardware and software components needed for its completion.

An electronic report that includes the following information about the complementary task must be presented to the TA:

- Software plan and explanation (pseudocode or flowchart)
- Connection schematic with relevant component calculations (current, voltage, timing values, etc.)
- Code listing with comments.
- Additional information used to complete the task (Web pages, datasheets, books, etc.)

Experiment 6

Motor Interfacing

Objectives

- Understanding the operating principles of electric motors used in embedded applications
- Recognizing electromechanical characteristics of DC motors, stepper motors, and servomotors
- Employing H-bridges and current drivers to control DC and Stepper motors
- Employing PWM signals to control servomotor position angle

Duration

- 2 Hours in the laboratory and extra time for complementary tasks

Materials

Table 6.1: Bill of materials for completing Lab. 6

Item #	Qty	Description	Reference
1	1	Development board	Tool
2	1	IDE application	Tool
3	1	LCD display: 2 lines, 16 characters	W/HD 44780 Controller
4	2	1/4W Carbon fill resistor	22 Ω
5	4	1/4W Carbon fill resistor	330 Ω
6	2	1/4W Carbon fill resistor	4.7 KΩ
7	2	1/4W Carbon fill resistor	12 KΩ
8	4	P-N junction diode	1N4004
9	2	BJT NPN transistor	MPSA42
10	2	BJT PNP transistor	MPSA92

Table 6.1: Continued

11	2	Momentary Switch	Pushbutton
12	2	Optocoupler	4N25
13	1	Half H-bridge Driver	L293D
14	1	Darlington Transistor Array	ULN2803
15	2	Optoswitch	RPR-220
16	1	DC Motor, 6VDC, 9100rpm, 0.14Oz-in	711 Motor
17	1	Stepper, 5VDC, Unipolar, 11.25° step angle	28BYJ-48
18	1	Servo, 6VDC, 38Oz-in, 180° Range	900-00005

6.1 Introduction

An electric motor can be defined as an electromechanical device capable of transforming electrical power into mechanical power. This kind of devices is extensively used in embedded systems applications where precise mechanical movement is required. Some examples include plotters, inkjet printers, and CNC (computer numerical control) machines. Depending on the type of application, different types of electric motors can be found. The three most common include: DC motor, servomotors, and stepper motors. Although these three types of motors all transform electricity into mechanical power, there are fundamental differences among them in terms of how they can be controlled.

6.1.1 Direct Current Motors

A direct current (DC) motor continuously spins when energy is applied to their electrical terminals. The speed of a DC motor is generally a function of the applied voltage. Thus, they can be used in applications where only the spinning speed is important.

Due to the electrical limitations of MCU's pins in terms of voltage and current, a direct connection between an MCU and a motor can rarely be done. Motor drivers are required to manage the motor load, speed, and the direction of rotation. These motor drivers can be implemented using discrete components or acquired in the form of integrated circuits (IC). See class's book Section 8.10.1 (Working with DC Motors) for a detailed explanation about the characteristic and interfacing of DC motors.

A common motor driver is provided by an H-Bridge. A H-Bridge is composed of NPN and PNP transistors as illustrated in Figure 6.1. The rotation direction is

determined by the signal combination used to turn-on and turn-off the transistors. When signal Rev is low and Fwd is high, transistors Q1 and Q3 are simultaneously turned On, allowing the current to flow from left to right in the motor. Making Rev high and Fwd low will activate transistor Q2 and Q4 instead, reversing the current direction and, therefore, reversing the polarity of the voltage applied to the motor. This causes a change in the direction of rotation. The transistors used to construct an H-Bridge are selected to satisfy the motor specifications (current and voltage).

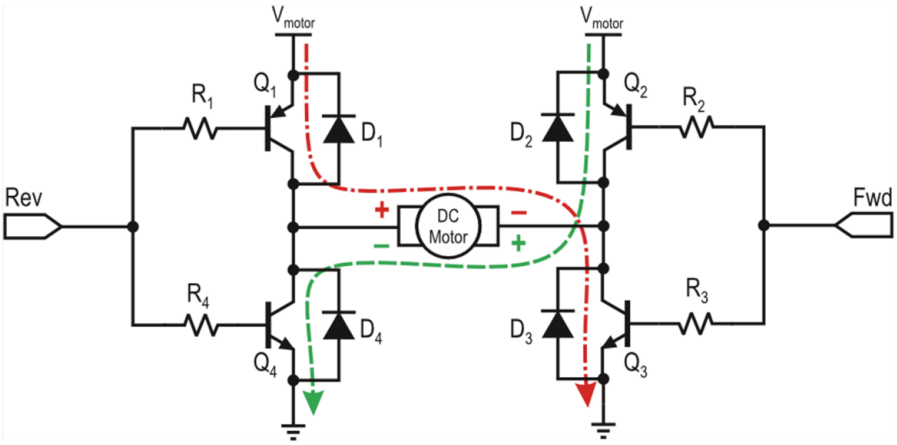


Figure 6.1: Transistor H-Bridge (*Source*: Introduction to Embedded Systems, M. Jiménez, R. Palomera, I. Couvertier)

A L293D is a motor driver with four channels capable of supplying a current up to 600mA per channel. This driver incorporates diode protection in each channel to avoid damage by the inductive turn-off transient generated by the motor. Each channel is controlled by TLL signals and each pair of channels has an enabling signal to connect and disconnect the channels.

The DRV8833 is another IC driver composed by dual CMOS H-bridges. This chip contains two full H-bridges capable of supplying a current up to 3A at a voltage up to 10.8V. This driver also provides short circuit protection, thermal shutdown, and supports low power modes.

6.1.2 Servo-Motors

A servo-motor is a DC motor with a feedback control that allows for a precise position control. A servo-motor is composed of four main components as illustrated in Figure 6.2: a DC motor that provides the basic electromechanical conversion, a control board housing the feedback electronics, a set of gears that slow-down the

DC motor rotation speed and increases the torque, and a position sensor, typically a potentiometer.

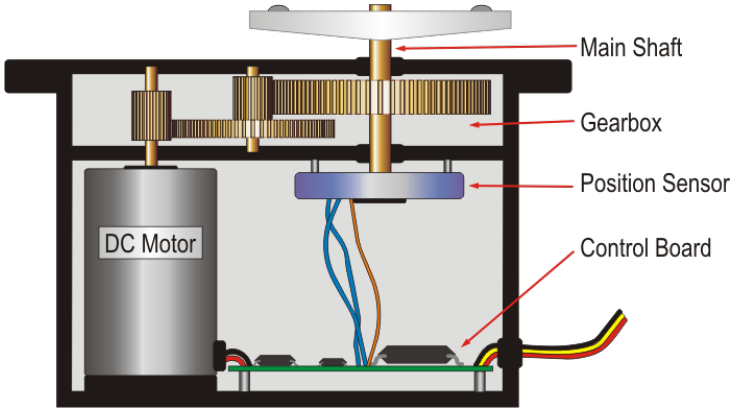


Figure 6.2: Servo-motor internal composition (*Source*: Introduction to Embedded Systems, M. Jiménez, R. Palomera, I. Couvertier)

The operating voltage of a servo-motor is generally in the range of 4 to 8 volts. A servo is controlled by a pulse-width modulation (PWM) signal that determines the position of the servo. Basically, the duration of the signal in high (duty cycle) determines the angular position of the motor, as illustrated in Figure 6.3. The position sensor in the servo continuously indicates the shaft angular position to the control board. A servo motor has a restricted travel angle of about 200° or less (typically of 180°) due the gearbox attached to the DC motor.

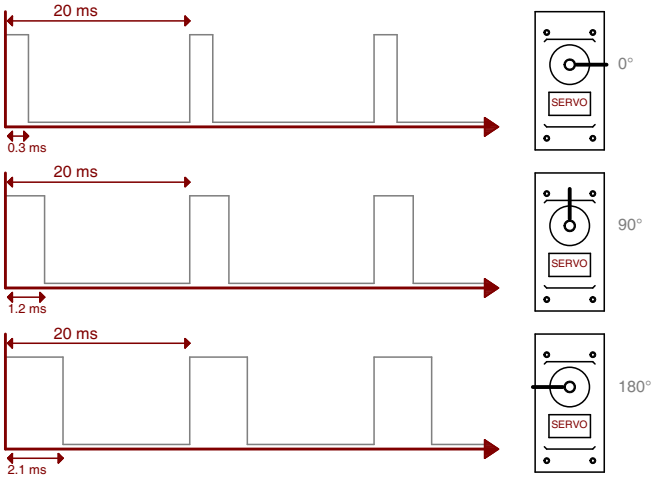


Figure 6.3: Servo-motor position determined by the signal pulse width (*Source*: Microcontrolador PIC16F84. Desarrollo de Proyectos, E. Palacios)

The external interface of a servo-motor has only three wires: V_{DD}, GND, and a $Control$ signal. Power is applied trough the V_{DD} and GND terminals while the desired position is specified through the $Control$ pin via a PWM signal. Each servo-motor has his own operation range that corresponds to the maximum and minimum pulse-width that the servo understands. See class's book Section 8.10.2 (Servo Motor Interfacing) for a detailed explanation on how to interface and control servo-motors.

6.1.3 Stepper Motor

A stepper motor is a type of electric motor that possesses a shaft which moves in discrete increments. The movement is the product of digital pulse sequences applied from a controller. Each pulse produces a precise angular displacement known as a step. Rotation increments or steps are measured in degrees. The structure of a stepper motor is different from of a DC motor, as it incorporates multiple windings to make possible the stepping behavior. Depending on how the rotor and stator are designed, stepper motors are classified in three types: **variable reluctance**, **permanent magnet**, and **hybrid**. See class's book Section 8.10.3 (Stepper Motor Interfaces) for a detailed explanation about the types of stepper motors.

A variable reluctance stepper motor has a soft Iron, non-magnetized, multitoothed rotor and a wounded stator with three to five windings (unipolar). The number of poles in the stator is larger than the number of teeth in the rotor. Torque is developed when the poles and teeth seek to minimize the length of the magnetic flux path between the stator poles and rotor teeth. In a permanent magnet stepper, the rotor is built using permanent magnets without teeth and the stator is constructed using multiple windings. In this case torque occurs when the excited stator poles attract opposite magnet poles in the rotor while repulsing similar poles. Hybrid steppers combine features from variable reluctance and permanent magnet motors. Hybrid motors have the ability of producing high torque at low and high speeds through the use of two multi-toothed, soft iron disks with a permanent magnet between them. See class's book Section 8.10 (MCU Motor Interfacing) for a detailed explanation.

Stepper motors come in a variety of step resolutions, ranging from 0.72° to 22.5° per step (500 and 16 steps per revolution).

The most important parameters specifying stepper motors include:

- **Working Torque**: The maximum momentum that the motor can reach while responding to an impulse excitation. If the torque of the load is larger than the working torque, the motor will not move.
- **Dynamic Torque**: The torque that the motor possess at a defined speed.

This torque may vary depending on the load attached to the motor and the driver used to control the motor.

- **Holding Torque**: Is the amount of torque needed to move the motor when the windings are energized but the motor's rotor is not moving.
- **Maximum pull-in/out**: Is defined as the maximum number of steps per second that the motor can perform.
- **Step resolution**: Is the angular displacement experienced by the motor with each excitation pulse, measured in degrees. This parameter can also be specified as the number of full steps per revolution. Table 6.2 shows common stepper motors angles. To calculate the number of steps for a stepper motor, the Equation 6.1 can be used:

$$SN = \frac{360}{\alpha}, \tag{6.1}$$

where SN is the number of steps per revolution and α is the step angle.

Table 6.2: Common resolution in commercial stepper motors

Degrees per excitation pulse	N° steps per revolution
0.72°	500
1.80°	200
3.75°	96
7.50°	48
15.00°	24

6.2 Basic Exercises

6.2.1 DC Motor Driven with Transistors

In this exercise, you will implement a DC motor driver using discrete components such as transistors. The control signals, to define the motor rotation direction, will come from a set of pushbuttons.

Follow the steps outlined below:

1. Connect the DC motor according to the schematic in Figure 6.4. This schematic is an isolated H-drive that incorporates two optocouplers in the control signals

to prevent propagating the noise generated by the motor into the MCU. Set VCC according to the DC motor specifications. Two complementary transistors were chosen with a collector current (I_C) of 0.5Amp. This I_C is required to withstand the current peaks generated by the motor when an instantaneous change of direction is required or when a load is applied to the motor. Verify each connection twice before powering the circuit.

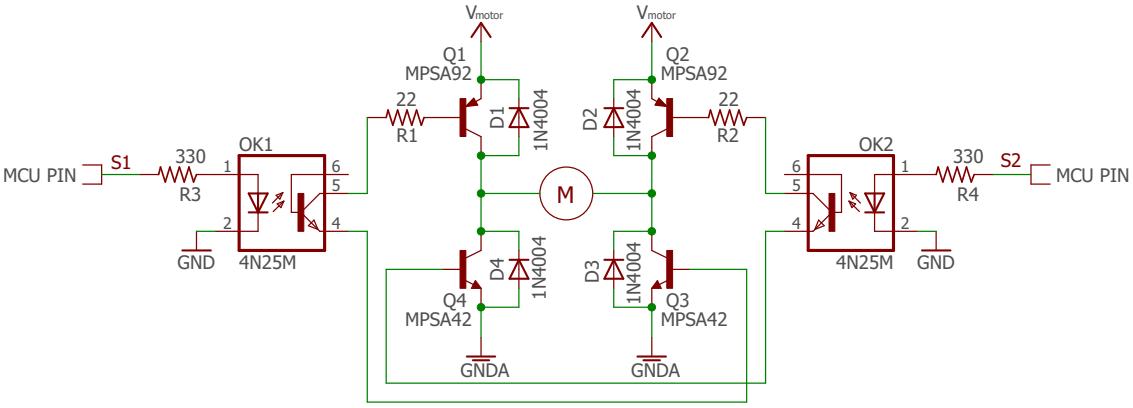


Figure 6.4: Isolated transistor H-Bridge schematic

2. Open your IDE.
3. Configure the I/O pins connected to the optocouplers as outputs.
4. Now, connect three pushbuttons and an LCD to the MCU. Each button must perform one of the functions described in Table 6.3. The LCD must be used to display the current motor state and the pushbutton depressed. S1 and S2 represent the logic values to be sent through the outputs connected to the optocouplers.

Table 6.3: Motor States

Motor			
Button	MotorAction	S1	S2
1	Stop Free	0	0
2	Rot. Left	0	1
3	Rot. Right	1	0
X	Stop Forced	1	1

Do not attempt to send this command as it creates a short circuit in the H-drive and burns ALL transistors

5. Make a program to continuously perform the function selected by each button until another function is selected. The program must start with the motor in the "stop free" condition.

6. Compile your code and verify that it does not contain any errors.

7. Run your code and verify if the DC motor performs the function selected.

6.2.2 DC Motor Controlled Through Driver IC

The objective of this exercise is controlling the direction of rotation and velocity of a DC motor through a L293D IC driver. The L293D is an IC driver that contains four half H-bridges designed to provide bidirectional drive currents up to 600mA.

Follow the steps outlined below:

1. Connect the DC motor according to the schematic in Figure 6.5. Set the VCC voltage according to the DC motor specifications. The IC is designed to work with 3.3V or 5V logic.

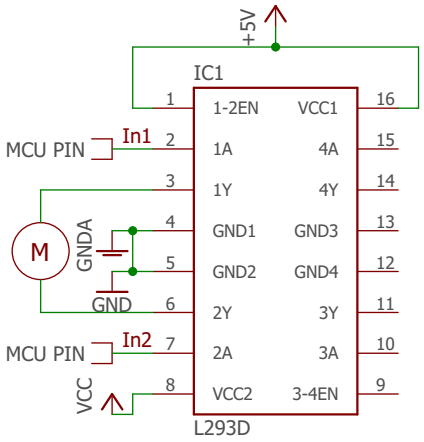


Figure 6.5: L293D H-Bridge schematic

2. Connect In1 to the pin where the S1 signal was generated in the previous exercise. Repeat the same for In2 and S2.

3. Open your IDE and perform the steps, 2 through 5, describe in the basic exercise 6.2.1.

4. Run your code and verify if the DC motor performs the function selected.

5. Now, modify your program to increment or decrement the velocity of the motor in steps of 20% using the first and second pushbutton. The third pushbutton must be used to toggle between the motor states. The change in velocity must be allowed in both directions of rotations. Hint: Use two PWM channels instead of the GPIOs selected.

6.2.3 Servo-motor Interfaces

The purpose of this section is controlling a servo motor using a PWM signal.

Follow the steps outlined below:

1. Connect a 900-00005 servo-motor according to the schematic in Figure 6.6.

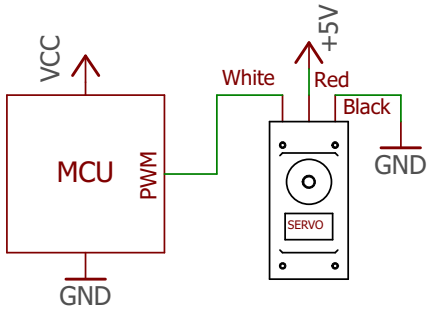


Figure 6.6: Servo-motor connection schematic

2. Verify that the I/O port where the servo-motor signal is connected has PWM capabilities. Otherwise move it to a pin able to produce PWM signal from an internal MCU timer.

3. Open your IDE.

4. Configure the PWM module to produce a square signal of 50Hz. Do not forget to configure the Timer associated to the PWM module.

5. Make a program to produce the angle displacements listed in Table 6.4. Use a timer function to change between the positions every 2 seconds. Complete the missing information of the table. Remember do not exceeds the pulse-widths values for the 0° (0.375ms) and 180° (2.1ms) in order to avoid damages and excessive current consumption in the servomotor.

6. The system must start with the servomotor in 0° position. Take into account that you have to permanently send the pulse width that corresponds to the

Table 6.4: Servo-motor angles routine

Angle Displacement	Servo Angle position	Necessary Pulse-width
22.5 to the left		
90.0 to the left		
22.5 to the right		
45.0 to the left		
90.0 to the right		
135.0 to the left		
22.5 to the right		
90.0 to the right		
67.5 to the right		

position desired to hold the servo-motor in that position. Design a setup with marks to verify if the servo reaches the desired angle.

7. Compile your code and verify that it does not contain any errors.

8. Run your code and verify if the servo performs the programmed routine.

6.2.4 Stepper Motor Interfaces

In this part, you will implement a full step and a half step sequence to control a unipolar, two-winding stepper motor.

Follow the steps outlined below:

1. Search information related to unipolar stepper motors and how they work. Also, search information about the 28YBJ-48 stepper motor and its characteristics, and IC driver ULN2803.

2. Connect the stepper motor according to the schematic in Figure 6.7.

3. Open your IDE.

4. Configure the four MCU I/O pins connected to the ULN2803 as outputs.

5. Write a program to perform the signal sequence described in Table 6.5. Take into account that the ULN2803 inverts the logic of its input signals (a high voltage in the input produces a low voltage at the output). The sequence is a one-phase activation that allows moving the motor shaft while saving energy in comparison with a two-phase activation. See class's book section

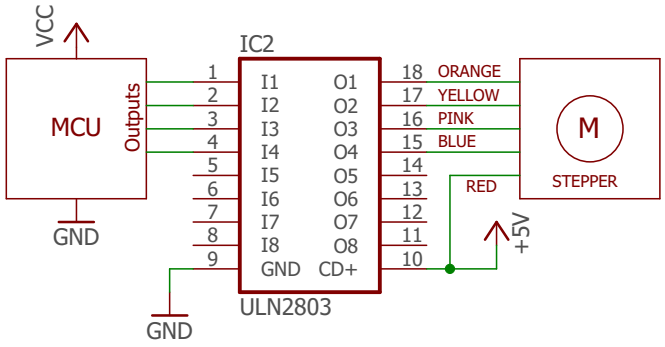


Figure 6.7: Stepper motor connection schematic

8.10.5 (Permanent Magnet Stepper Motors) for a detailed explanation about one-phase and two-phase activation.

Table 6.5: Full-Step Sequence

	Motor Coils			
Step	(4) Orange (A)	(3) Yellow (B)	(2) Pink (A')	(1) Blue (B')
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

Read Note section for full-step sequence explanation

6. Use a delay of 10ms between steps.

7. Compile your code and verify that it does not contain any errors.

8. Run your code and verify that the stepper motor works as expected.

9. Now, modify your code to implement the signal sequence describe in Table 6.6 that corresponds to a Half-step sequence.

10. Use the same delay established previously (10ms).

11. Run your code and verify if the stepper motor works as expected.

12. Finally, modify your code to rotate the stepper motor 270 to the left, later on 180 to the left, and finally 90 to the right. Use marks to verify if the stepper reaches the desired angle.

Table 6.6: Half-Step Sequence

	Motor Coils			
Step	(4) Orange (A)	(3) Yellow (B)	(2) Pink (A')	(1) Blue (B')
1	1	0	0	0
2	1	1	0	0
3	0	1	0	0
4	0	1	1	0
5	0	0	1	0
6	0	0	1	1
7	0	0	0	1
8	1	0	0	1

Read Note section for half-step sequence explanation

Note: In stepper motors, a full-step sequence allows the motor to advance in angular increments equal to its nominal resolution. In the case of a half-step sequence, the motor resolution is double because the motor advances only half of its nominal angular resolution. If a motor has an angular resolution of 11.25° per step, with a half-step sequence the displacement will be 5.625° per step.

6.3 Complementary Tasks

6.3.1 Stepper Motor Characterization

The activity consists in determining the maximum input signal frequency in which the stepper motor can work without missing steps. The system shall use keys UP and DOWN to increase and reduce the motor speed, and START/STOP to turn the motor on and off.

An oscilloscope shall be used to observe and measure the motor input signal. The control signals must start at the 1st value in Table 6.7 as the initial speed of the motor. When the START/STOP key is depressed the motor must perform two complete rotations (720°) at the selected speed, based on the # of steps required for a revolution. A mark must be placed on the motor shaft to observe is the motor is able to perform the two revolutions, see Figure 6.9 for an example on how to do the angle measurements. Once the two rotations are completed, the following input signal period shall be selected using the UP key. The input signal periods for the motor tests are defined in Table 6.7. The missing information in Table 6.7 must be completed and calculated for each test. The motor speed must be derived from the input signal frequency. Plot a graph where the relationship between the motor

velocity and input period signal can be observed. Be careful while the motor is being tested; if the motor emits inappropriate sounds or the angular movement is irregular, the test must be stopped using the STARt/STOP key and the frequency should be marked unsuccessful in the Table (Critical Frequency column). You can use the block diagram shown in Figure 6.8 as a reference to connect your system.

Table 6.7: Frequencies to be tested

Sig. Period (ms)	Sig. Frequency (Hz)	Motor Speed (rpm)	Critical Frequency (Yes/No)
100			
50			
20			
10			
5			
2			
1			
0.5			
0.2			
0.1			

Note: Sig. means Signal

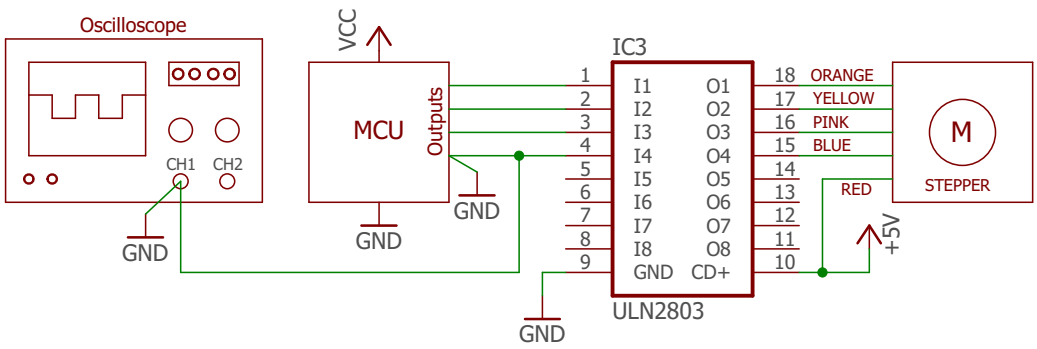


Figure 6.8: Stepper motor input frequency measurement diagram

Presentation and Report

Each basic exercise and complementary task must be presented to the TA before the initiation of the next laboratory experiment. The demonstration must be made

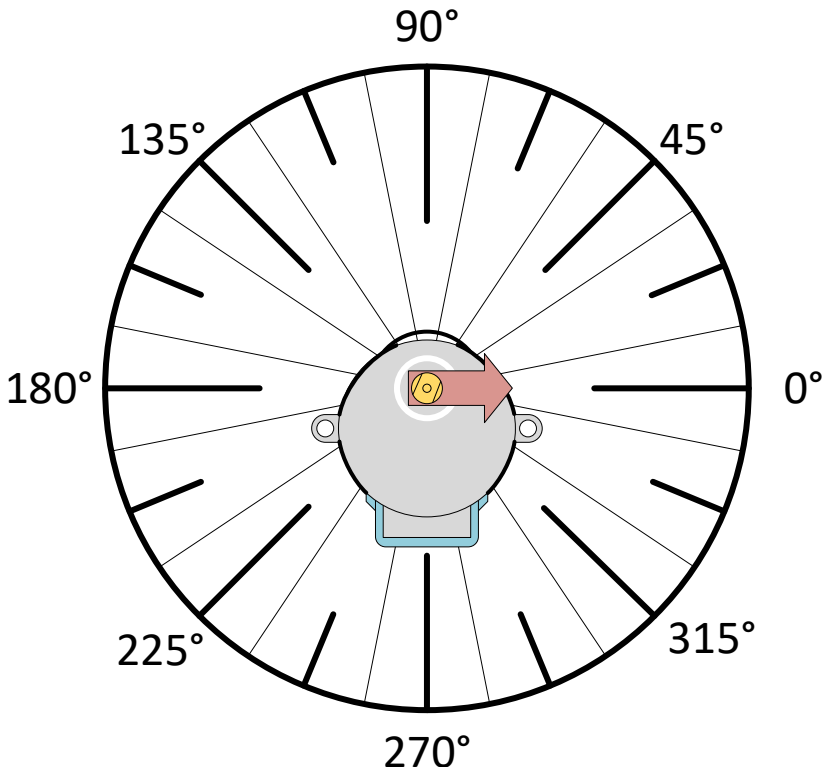


Figure 6.9: Stepper motor measurement circle example

personally in the lab and including the hardware and software components needed for its completion.

An electronic report that includes the following information about the complementary task must be presented to the TA:

- Software plan and explanation (pseudocode or flowchart)
- Connection schematic with component calculations
- Code listing with comments.
- Additional information used to complete the task (Web pages, datasheets, books, etc.)

Experiment 7

Serial Communication

Objectives

- Identifying and understanding the standard formats for serial communications
- Recognizing the differences between synchronous and asynchronous serial communications
- Understanding the physical requirements and operation of an USART interface
- Using an USART interface to transmit and receive information to/from a personal computer and your MCU
- Understanding the connection and operation of an I^2C interface
- Employing an I^2C protocol to share data with a real-time clock device

Duration

- 2 Hours in the laboratory and extra time for complementary tasks

Materials

Table 7.1: Bill of materials for completing Lab. 7

Item #	Qty	Description	Reference
1	1	Development board	
2	1	IDE application	
3	1	LCD display: 2 lines, 16 characters	W/HD 44780 Controller
4	1	1/4W Carbon fill resistor	1 KΩ
5	5	1/4W Carbon fill resistor	4.7 KΩ
6	3	Momentary Switch	Pusbutton
7	1	I^2C Real-Time clock calendar	DS1307

Table 7.1: Continued

8	1	Quartz crystal	32.768KHz
9	1	3V Lithium Battery	CR2032
10	1	Piezoelectric buzzer	Buzzer
11	1	USB-To-UART converter cable	FTDI TTL-232R

7.1 Introduction

Serial channels are extensively used to establish communications between devices and computers, working under different formats and protocols in a wide range of applications, and transmitting information between points that can be from a few centimeters to hundred or thousand of kilometers apart.

In a serial communication channel, data is sequentially transmitted, one bit at a time, over a single data line. Each bit transmitted over a serial channel takes a predetermined amount of time ($tbit$) to be transmitted. Thus, transmitting an $n - bit$ character will take $n \cdot tbit$.

The number of bits transmitted in a serial channel per unit of time determines the transmission rate of the channel. Two commonly used metrics include:

- Bit rate: Number of bits-per-seconds (bps) transmitted over the channel.
- Baud rate: Number of symbols per seconds transmitted over the channel.

When the channel modulation scheme assigns one bit to each signal transmitted over the channel, the terms bit and baud rate are interchangeable. Modern modulation schemes commonly assign multiple bits per signal. In such cases, the terms have different meanings.

The most basic structure of serial channel calls for a transmit signal (TxD), a receive signal (RxD), a ground reference, and some form of clock synchronization, as illustrated in Figure 7.1. Depending on the protocol, additional signals might also be necessary for handshaking, synchronization, or data flow regulation. Common serial protocols and physical standards include RS-232, RS-485, SPI, I^2C, CAN-BUS, and 1-wired.

7.1.1 Types of Serial Channels

The serial channels could be Simplex, Half Duplex, or Full Duplex where the difference between them is the connectivity used to sent the information:

Simplex serial channel: The transmission process is only in one direction.

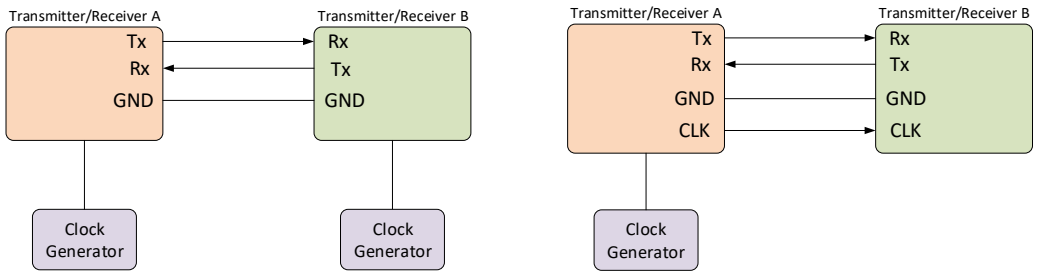
Half duplex serial channel: The transmission process could be in either direction, but only in one direction at a time.

Full Duplex serial channel: The transmission process could be in either direction simultaneously due to the usage of separate links (one for transmitting and one for receiving).

See class's book Section 9.2 (Types of Serial Channels) for a detailed explanation on this topic.

7.1.2 Synchronous Vs. Asynchronous Serial Communication

Generally, a serial channel requires the usage of a clock signal to synchronize the data transmission and reception ends. Depending on how the transmission clock is handled, the communication can be asynchronous or synchronous. In asynchronous communications, individual clock generators are used on each end of the channel, as shown in Figure 7.1(a). In synchronous communications, the clock signal is transmitted through the channel, as illustrate in Figure 7.1(b).



(a) Asynchronous (b) Synchronous

Figure 7.1: Synchronous vs asynchronous serial communication.

In both cases, asynchronous and synchronous channels, the transmitted message is divided into fundamental units known as packets or datagrams. Commonly, a packet contains three parts:

- A header that indicates the beginning of the packet.

- A body that contains the information or message being transmitted.
- A footer that delimitates the data. In most cases, the footer also includes redundant information that can be used for error checking & in some cases error correction.

7.1.3 Serial Interfaces

A Universal Synchronous/Asynchronous Receiver/Transmitter (USART) controller is a fundamental module that can generate the necessary signals for synchronous or asynchronous communication, one mode at a time. In asynchronous mode, an USART becomes an UART module. In synchronous mode, it can support one of several synchronous protocols, such as SPI, I^2C, and others.

USARTs contain multiple functional units and registers, which may vary from one architecture to another. However, there are a few basic components, as illustrated in Figure 7.2, that are fundamental to its operation. These includes:

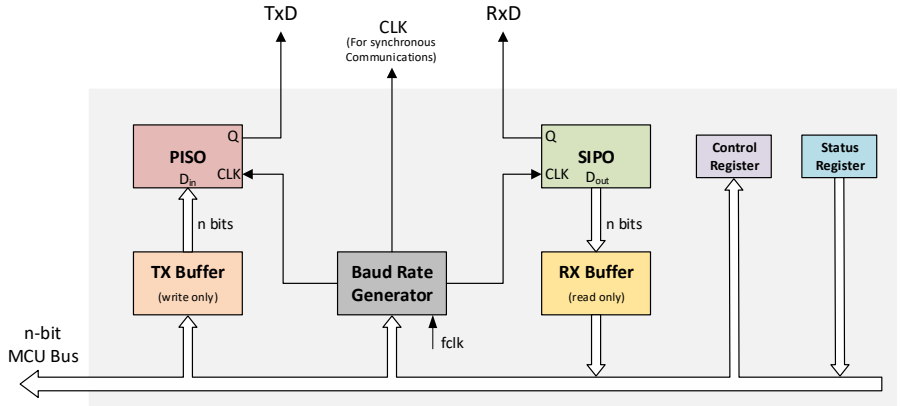


Figure 7.2: Minimum USART components

- A Baud Rate Generator: A timer that generates the clock frequency necessary for setting the transmission and reception speed (baud rate). The clock signal may be transmitted through the channel or generated at each end, depending on whether the channel is synchronous or asynchronous.
- Parallel Input Serial Output Shift Register (PISO): Converts n-bit parallel data from the CPU into a serial stream.
- Transmit Buffer (TX Buffer): Holds the data to be transmitted. Also called "Data-out register".

- Serial Input Parallel Output Shift Register (SIPO): Converts the serial input stream into parallel data.
- Receive Buffer (RX Buffer): Accommodates newly received characters for the CPU to read. Also called "Data-in register".

Two additional registers are necessary to operate a USART. These include a **control register** and a **status register**. A control register allows configuring the USART in the desired operating mode. For example, it allows choosing either synchronous or asynchronous mode, enabling the transmitter or receiver, enabling USART interrupts, setting number of bits to be transmitted, selecting error check, etc. The status register contains information that indicates the current USART status. Indicators, such as when the TX Buffer can be written, when the RX Buffer can be read, or when an error has occurred.

Status bits TxR (Transmitter ready) and RxR (Recover ready) are essential for a UART operation. TxR signals are used when the Data-out register is empty in order to accept new characters for transmission. RxR indicates that a new character has been received and is ready for reading in the Data-in buffer. These flags can be used in a polled fashion to operate the channel or be enabled to trigger interrupts, alluding to a more efficient way to operate the UART.

The baud rate generator, as any timer, is configured using the system clock frequency, a divider (n), and a prescaler value. The desired baud rate is obtained as:

$$BaudRate = \frac{fclk}{PrescalerValue * (n + 1)} \tag{7.1}$$

UART Operation

An asynchronous frame is composed of a start bit, multiple data bits (five- to eight-bit characters), an optional parity bit, and a stop bit as illustrated in Figure 7.3. The parity check is a simple mechanism for detecting errors in the channel. See class's book Section 9.3.5 (UART Structure and Functionality) and Section 9.3.7 (UART configuration and Operation) for a detailed explanation of UART hardware interface and operation.

I^2C

The Inter-Integrated Circuit bus (I^2C) is a synchronous serial protocol developed by Philips in the early 1980s to support board-level interconnections.

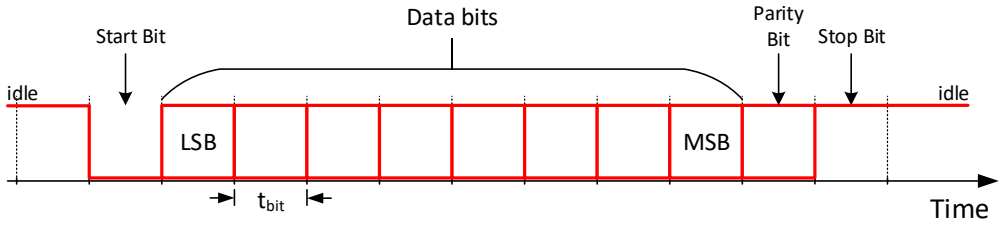


Figure 7.3: Typical asynchronous serial transmission frame (*Source*: Introduction to Embedded Systems, M. Jiménez, R. Palomera, I. Couvertier)

I^{2}C uses two signal lines to connect with other devices: Serial DAta line (SDA) and Serial CLock line (SCL), both ground (GND) referenced. The SCL line synchronizes all bus transfers while SDA carries the transferred data. Both SDA and SCL are open collector lines, requiring external Pull-up resistors. Figure 7.4 shows a basic I^{2}C topology with two masters and three slaves interconnected.

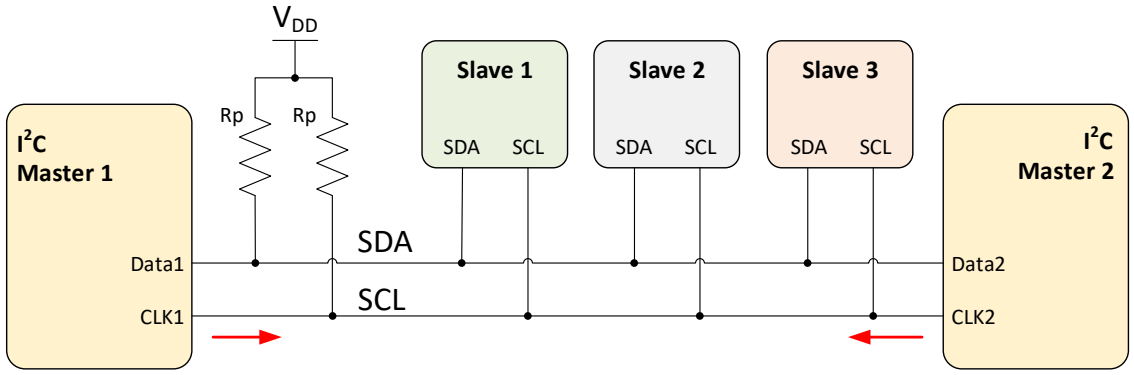


Figure 7.4: I^{2}C multimaster-multislave structure (*Source*: Introduction to Embedded Systems, M. Jiménez, R. Palomera, I. Couvertier)

In an I^{2}C protocol, the devices are software addressable, through a 7- or 10-bit address field. These number denote the maximum quantity of devices that can be accommodate on the bus but also, the number of devices is limited by the total capacitance of the bus. This capacitance also limits the maximum speeds that can be reached by the bus. Some predefined speeds for the protocol: are the standard speed (100Kbps) and the fast speed (400Kbps).

In I^{2}C, the master device controls the communication process. It defines the slave to communicate with, whether the data will be transmitted or received, and generating the necessary clock signals for the data transmission.

A typical structure of an I^{2}C packet is shown in Figure 7.5. The Figure denotes the

start condition sent by the master, the slave address field followed by the bit that indicates the communication direction (read or write). Then, the ACK sent by the slave, the data packets sent by the master, the ACK response from the slave in each packet received, and the stop condition.

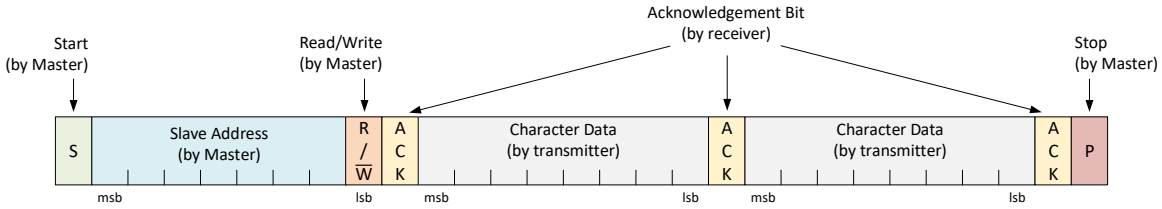


Figure 7.5: I^{2}C message structure (*Source*: Introduction to Embedded Systems, M. Jiménez, R. Palomera, I. Couvertier)

See class's book Section 9.4.2 (The Inter-Integrated Circuit Bus: I^{2}C) for a detailed and deep explanation about an I^{2}C architecture and interface.

7.2 Basic Exercises

7.2.1 Asynchronous Serial Communication (UART)

In this exercise, we will use a USB-to-UART cable for sending characters to a personal computer (PC) using asynchronous serial communication.

Follow the steps outlined below:

1. Locate the TX and RX signal pins in your MCU and connect them to the computer using a USB-to-UART cable. Be sure that the voltage logic of the cable corresponds to the operating voltage of your MCU. Do not forget to connect the GND to the USB-to-UART cable.
2. Open a HyperTerminal on your computer or other RS-232 communication program (Putty) and configure it for:
   - Baud rate: 9600 bauds
   - Data bits: 8
   - Parity bit: none
   - Flow control: none
3. Open your IDE.

4. First, configure the baud rate on your MCU (The serial configuration in the MCU must match those of the other device we wish to communicate to):
    - Configure the baud control register; that is, select the clock source and specify the prescaler and divider values.
5. Next, configure the UART for asynchronous transmission:
    - In the control register specify the mode to be asynchronous.
    - Enable the transmitter and global USART module.
    - Configure the pins selected on your MCU to work with the UART module.
6. Write a program to write a character into the UART transmitter via polling. You can use the flowchart in Figure 7.6 as a guide for your code.

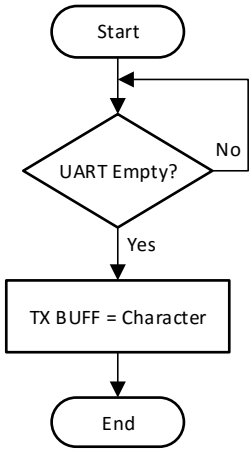


Figure 7.6: UART data transmit flowchart (polling)

7. Compile your code and verify that it does not contain any errors.
8. Run your code and verify if the PC received the transmitted character.
9. Now, modify your code to send over the UART the string: "Hello World!". Develop a subroutine for your code. Use a delimiting character to denote the end of the message (eg. EOT or CR).

7.2.2 Sending and Receiving Characters via UART

In this part, you will send and receive characters from a PC using an asynchronous serial communication.

Follow the steps outlined below:

1. Use the same setup as the basic exercise developed before "Asynchronous Serial Communication (UART)".
2. Connect an LCD to your MCU as you did in previous experiments.
3. Open your IDE.
4. Modify the configuration procedure outlined in the previous section to enable, in the control register, the receiver side of the UART module.
5. Write a program that receives a character using interrupts and displays them on the LCD. To complete this task, you could write the received character directly onto the LCD upon reception. Another way could be using a memory buffer, where received characters are stored in the buffer and having a function to dump the buffer contents into the LCD.
6. Compile your code and verify that it does not contain any errors.
7. Run your code and verify if that the PC is able to send a character to your MCU and the LCD shows the received character.
8. Now, modify your code to receive a 16-character-lower-case message from the PC and display it on the LCD. The message must be returned to the PC via UART in upper case.

7.2.3 Synchronous Serial Communication (I^2C)

In this section you will use an I^2C channel to read the time registers from a Real-Time Clock (RTC) device (DS1307). The DS1307 is a real-time clock-calendar chip that communicates with the MCU through I^2C.

Follow the steps outlined below:

1. Identify and available I^2C port in your MCU and connect the RTC to it according to the schematic shown in Figure 7.7.
2. Open your IDE.
3. Configure the baud rate of your MCU in *low speed* mode (10 kbps):
    - Configure the baud control registers with the prescaler value and baud rate clock source.

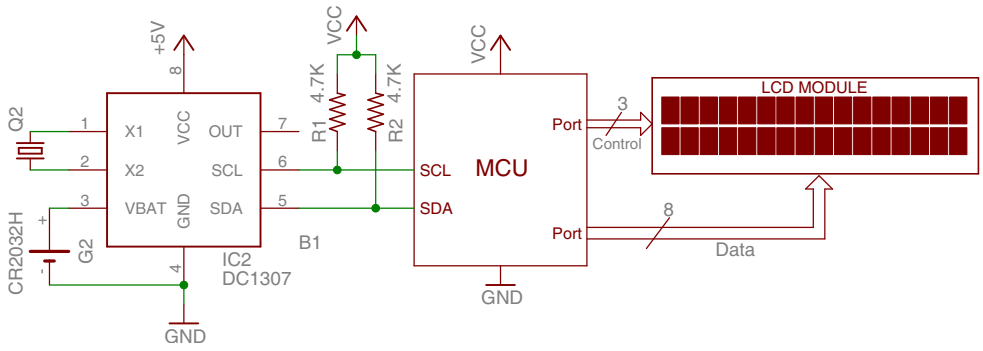


Figure 7.7: Interface for a DS1307 to your MCU via I^{2}C bus.

- Configure the baud rate generator number necessary for the desired baud rate. If your MCU uses a timer for generating the baud rate, programming that timer.

4. Configure the USART for synchronous I^{2}C communication:
   - In the USART control registers choose a synchronous operating mode.
   - Program the corresponding pins to work with I^{2}C protocol.
   - Configure your MCU in Master Mode.
   - Note that to read data from the DS1307 it is necessary to initially send the device's address byte with bit0 in 1. Then, send the register address to be read. Figure 7.8 shows a timing diagram of the reception of a byte from a slave device.
   - Note that to write data on the DS1307 it is necessary to send the device's address byte with bit0 in 0. Then send the register address to be modified and the data that will be stored in the register. Figure 7.9 shows a timing diagram of the transmission of a byte to a slave device.
5. Now, write a program to ask the DS1307 for its current time and display it constantly on the LCD in format "HH:MM:SS". Note that this exercise did not set the time (and date) on the chip, therefore the time value read will correspond to the time elapsed after the last power-up.
6. Compile your code and verify that it does not contain any errors.
7. Run your code and verify if the LCD displays the read time.
8. Later, modify your code to show the time in the first line and the date in the second line of your LCD.

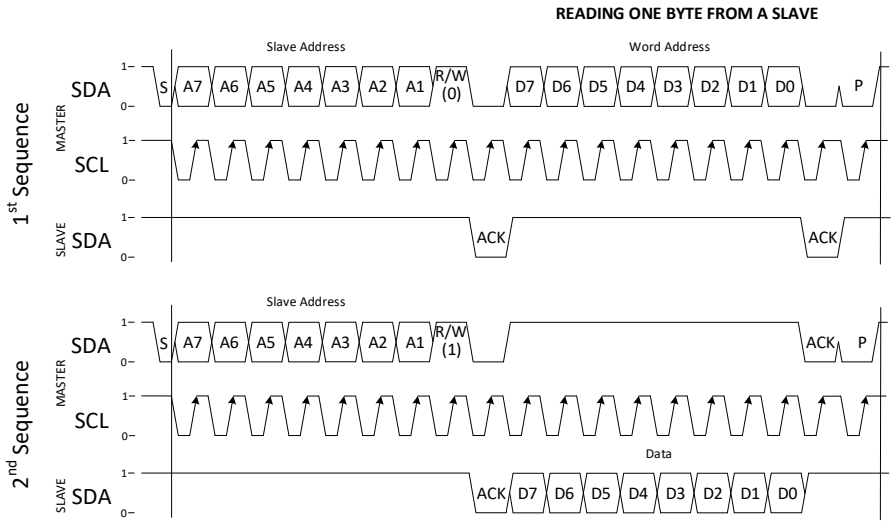


Figure 7.8: Timing diagram to read a byte from a slave (*Source*: Mastering the I^{2}C Bus, V. Himpe)

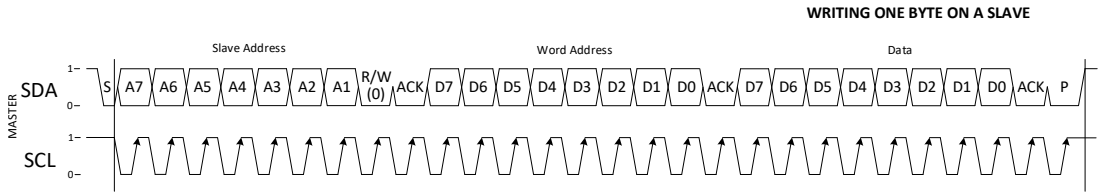


Figure 7.9: Timing diagram to write a byte to a slave (*Source*: Mastering the I^{2}C Bus, V. Himpe)

7.3 Complementary Tasks

7.3.1 Digital Alarm Clock

The activity consists of using a DS1307 to make a programmable alarm clock. The clock shall use keys UP, DOWN, and ENTER to configure the current time, date, and desired alarm upon reset. When the system turns-on, a user shall be able to configure the alarm. Upon setup, the current date shall be displayed on the top line of the LCD and the time on the bottom line. Use the ENTER key to toggle between alarm and current time&date. When the current time matches the alarm time, an audible sound must be produced through a buzzer to indicate that the set time has been reached. The ENTER key must turn the alarm sound off. You can use the block diagram shown in Figure 7.10 as a reference to connect your system.

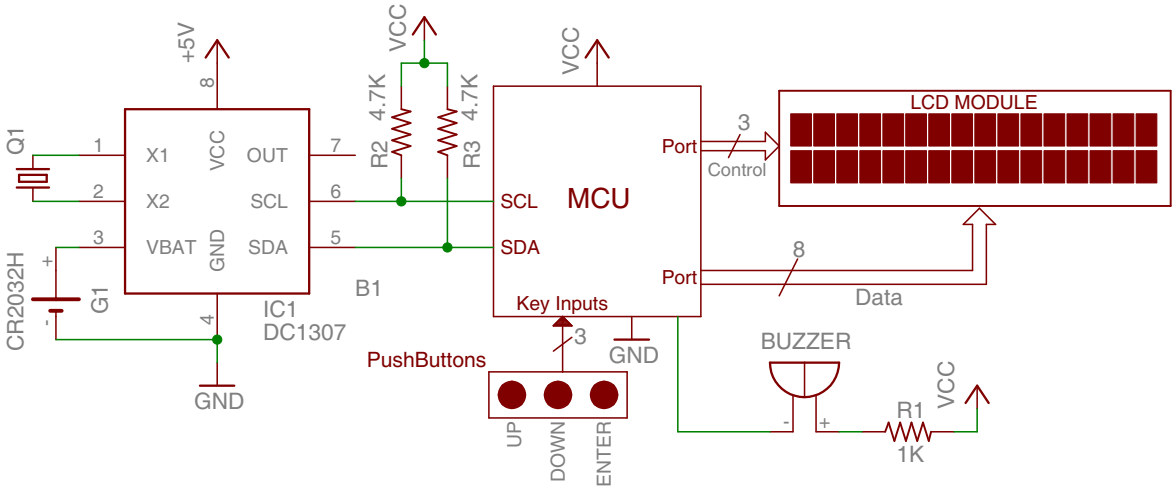


Figure 7.10: Digital Alarm Clock Diagram

Presentation and Report

Each basic exercise and complementary task must be presented to the TA before the initiation of the next laboratory experiment. The demonstration must be made personally in the lab and including the hardware and software components needed for its completion.

An electronic report that includes the following information about the complementary task must be presented to the TA:

- Software plan and explanation (pseudocode or flowchart)
- Connection schematic with component calculations
- Code listing with comments.
- Additional information used to complete the task (Web pages, datasheets, books, etc.)

Experiment 8

Data Converters (DAC & ADC)

Objectives

- Understanding the uses of DAC and ADC in embedded applications
- Understanding how to operate a DAC
- Using a DAC to generate different voltages and signal waveforms
- Identifying and understanding the architecture of an ADC and how to operate it from an MCU
- Using an ADC module to read analog signals that comes from electronic devices such as sensors

Duration

- 2 Hours in the laboratory and extra time for complementary tasks

Materials

Table 8.1: Bill of materials for completing Lab. 8

Item #	Qty	Description	Reference
1	1	Development board	
2	1	IDE application	
3	1	LCD display: 2 lines, 16 characters	W/HD 44780 Controller
4	1	Light Emitting Diode	5mm Red LED
5	1	1/4W Carbon fill resistor	330 Ω
6	4	1/4W Carbon fill resistor	2.4 KΩ
7	1	1/4W Carbon fill resistor	4.7 KΩ
8	1	1/4W Carbon fill resistor	10 KΩ

Table 8.1: Continued

9	1	Non-polarized ceramic capacitor	0.1nF
10	1	Momentary switch	Pushbutton
11	1	Operational amplifier	LM358
12	1	Analog temperature sensor	LM35
13	1	Digital-to-analog converter	DAC0808

8.1 Introduction

8.1.1 Data Converters

Data converters allow for interfacing digital systems to the analog world. Many embedded applications need to interact with analog processes to either receive information about their status, level, or behavior; or to control their status, level or how they behave. In either case, the discrete nature of a digital system requires converting the data format from or to the analog domain to enable operation. This requirement calls for the usage of data converter circuits. Figure 8.1 illustrates a typical signal processing chain denoting the position of the required data converters.

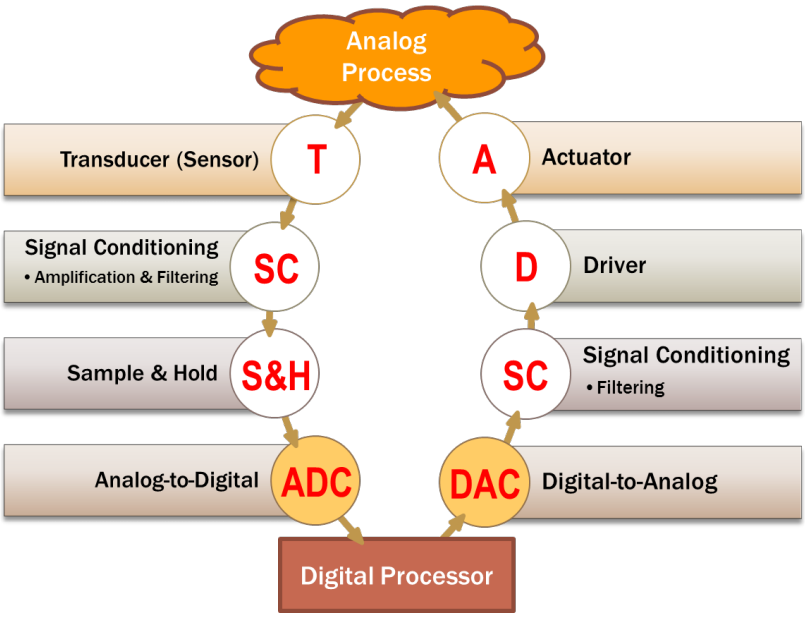


Figure 8.1: Signal Processing Chain

Data converters can be found in the form of Digital-to-Analog Converter or Analog-to-Digital Converter where the first is used for transforming a digital code into a

analog voltage and the second is used for converting an analog voltage into a digital code.

8.1.2 Digital-To-Analog Converters (DAC)

A digital-to-analog converter (DAC) is a device that converts a binary code presented to its input into a discrete voltage value. The specific voltage resulting from a particular digital code will depend on the DAC resolution and the reference voltage it uses. Embedded microcontrollers usually do not include DAC modules. DACs are usually provided via external ICs interfaced to MCUs. A block representation for an n-bit DAC is presented in Figure 8.2 where a digital input is transformed into an analog output.

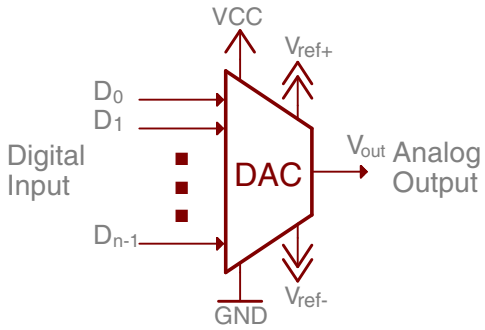


Figure 8.2: DAC Block Diagram

Internal DAC Structure

The internal structure of a DAC is conceptually simple. It includes a voltage reference, a resistor network to break down the reference voltage into binary-weighted voltage values, and some form of analog accumulation to add up the binary-weighted voltages according to the binary code being converted into analog. The most intuitive way of implementing a DAC is provided by a binary-weighted resistor network.

A **Binary-Weighted resistor** DAC uses an operational amplifier in adder configuration to perform the accumulation of the voltages provided by the resistor network as shown in Figure 8.3. In this configuration, the output voltage is composed of the sum of the input voltages where each input uses a different resistor value to represent the binary weights. The output voltage is:

$$Vo = \frac{2R_f}{R} V_R N, \tag{8.1}$$

where N is the digital code represented by the inputs in the DAC and V_R the reference voltage. The major problem with this configuration is the wide range of resistors needed for its construction.

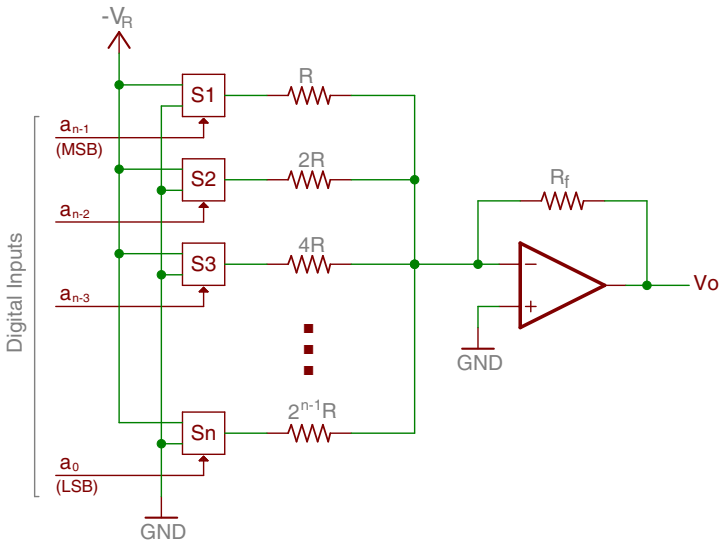


Figure 8.3: Binary-Weighted resistor diagram

An **R-2R or resistor ladder** DAC uses a resistor network made-up of only two different resistor values as shown in Figure 8.4. The set of switches $S_1 \ldots S_n$ allow for connecting each resistor to the Op-Amp adder inputs to generate an output voltage that corresponds to the digital input code specified with lines $a_0 \ldots a_{n-1}$. In this configuration the output voltage is:

$$Vo = \frac{V_{Ref}}{2^n}\left(2^{n-1}a_{n-1} + 2^{n-2}a_{n-2} + \cdots + 2a_1 + a_0\right) \tag{8.2}$$

The advantage of this configuration with respect to the Binary-Weighted resistor circuit is the use of only two resistor values independently from the number of digital inputs.

8.1.3 Analog-To-Digital Converters (ADC)

An analog-to-digital converter (ADC) is a device that converts an input voltage into a digital code. The specific code resulting from a particular analog voltage will depend on the ADC resolution and the reference voltage it uses. Figure 8.5 shows a basic block representation for an ADC module where an analog input, selected through a multiplexer, is converted into a n-bit digital code. The ADC module uses

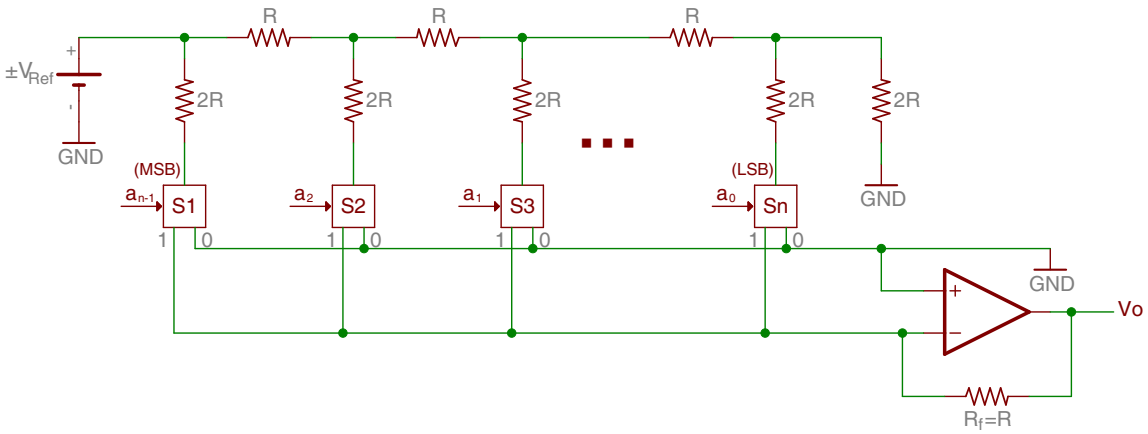


Figure 8.4: R-2R diagram

an input signal to start the conversion (START) and it generates a flag when the conversion has finished (EOC). Most microcontrollers include multi-channel ADC modules among its embedded peripherals but commercial off-the-shelf ADC chips are also available to be interfaced with MCUs.

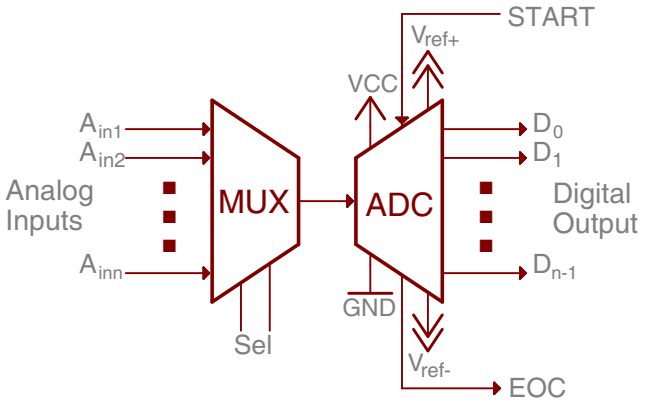


Figure 8.5: ADC Block Diagram

ADC Topologies

ADCs have been implemented using different types of circuits topologies where each topology has its own characteristics, advantages, and disadvantages. For example, Flash ADCs are the fastest ones but consume a large amount of power due the usage of $2^n - 1$ comparators. Likewise, there are other topologies such as two-steps ADC, pipeline ADCs, Slope ADC, sigma-delta converters, among others.

One of the most popular ADC architecture, embedded as a module in many microcontrollers, is the **Successive Approximation (SA) ADC**. This module is composed of a successive approximation register (SAR), a clock signal, a DAC module, and an operational amplifier in comparative voltage mode, as shown in Figure 8.6. In a SA ADC, the n-bits are determined from MSB to LSB in n steps by comparing the analog input with mid levels of successive region intervals.

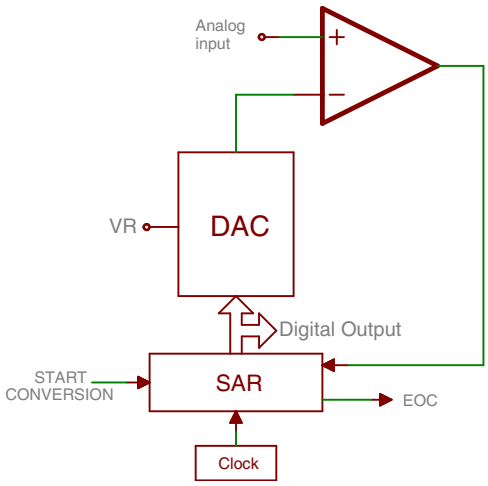


Figure 8.6: Successive Approximation ADC block diagram

The main advantage of the SA ADC is that the circuit complexity and power dissipation are less than those found in most other types of ADC topologies. One of its drawbacks is that eventually the comparator must do a comparison within 1LSB of precision, and precautions must be taken to deal with noise.

The sequential steps in a SA DAC to convert a voltage value is illustrated in Figure 8.7, which corresponds to 4-bit DAC. In step 1, the input voltage is compared with the mid level 1000 of the full scale region. Due to the input voltage being lower than the DAC output, the MSB is turned to 0. In step 2, the DAC output corresponds to a 1/4 of the scale (mid level of the already determined region 0100) and as the input voltage is greater, the third bit remains as 1. In step 3, the input voltage is less than the DAC output (0110) converting the second bit from 1 to 0. Finally, in step 4, as the input voltage is greater that the DAC output (0101), the LSB remains 1. This indicates that an N-bit SAR ADC will require N comparison periods. The increment or decrement rate of the bits is controlled by the clock.

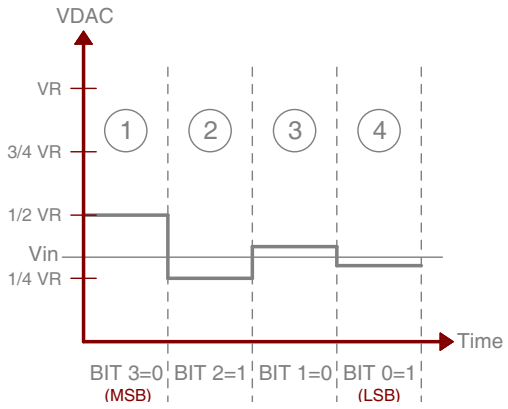


Figure 8.7: SAR Operation

8.2 Basic Exercises

8.2.1 Generating Voltages Using a DAC

The purpose of this exercise is to demonstrate the operation of a DAC by generating different voltage levels using the DAC0808. A DAC0808 is a 8-bits digital-to-analog converter with an analog output current.

Follow the steps outlined below:

1. Connect the DAC to your MCU according to the schematic shown in Figure 8.8. Define the reference voltages for the DAC with VCC as the positive voltage and GND for the negative. The circuit is composed by a DAC0808 that requires a 5V, -15V, and a 8-bit input signal for its operation. An operational amplifier, in current to voltage converter configuration, must be connected to the DAC output to convert the output current in a voltage value.

2. Open your IDE.

3. Configure the MCU pins connected to the DAC as outputs.

4. Make a look-up table with the hexadecimal values in Table 8.2.

5. Write a program that sends the appropriate hexadecimal value to the DAC using a timer function. The timer must change the value that appears in the output port each one second. To handled the timer, you must created an ISR. The binary value sent to the DAC must be displayed on the LCD.

6. Compile and verify that your code does not contain any errors.

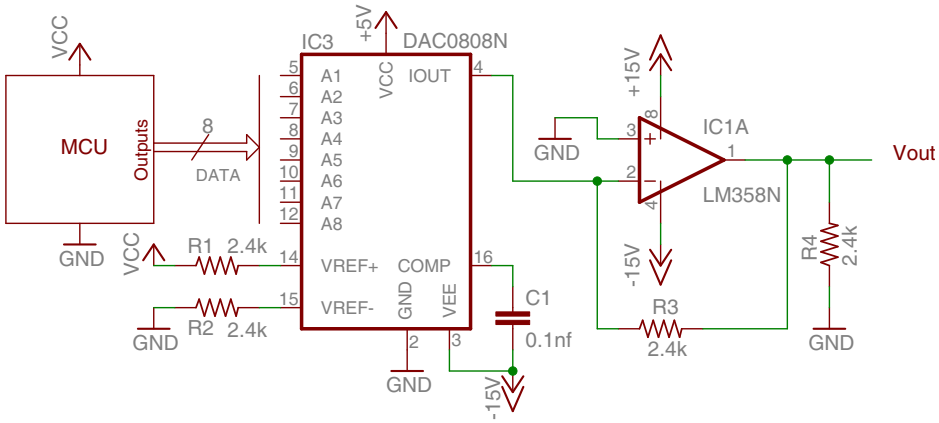


Figure 8.8: Block diagram for DAC connection

7. Run your code and verify if the voltage in the DAC changes.

8. Measure the DAC output voltage in each case and compare it with the expected value from the DAC according to binary number presented in its input. Calculate the percentage of error for each measurement. Complete the Table 8.2 and plot the V_out. Investigate how to calculate the expected output voltage from the DAC ($V_{out} = f\{A_i, VREF\}$).

Table 8.2: DAC Values

Hex Value	Expected Voltage	Measured Voltage	% Error
00			
17			
2E			
45			
5C			
73			
8A			
A1			
B8			
CF			
E6			
FF			

9. Now, modify your code to produce a sinusoidal wave with a frequency of 500Hz and peak-to-peak voltage of 3.3V.

8.2.2 Reading Voltages

In this part, you will Read different voltages from analog devices such as potentiometers using the MCU internal ADC peripheral.

Follow the steps outlined below:

1. Connect the potentiometer to your MCU according to the block diagram in Figure 8.9. The voltage source to be connected to the potentiometer must match the voltage range selected for your MCU ADC module.

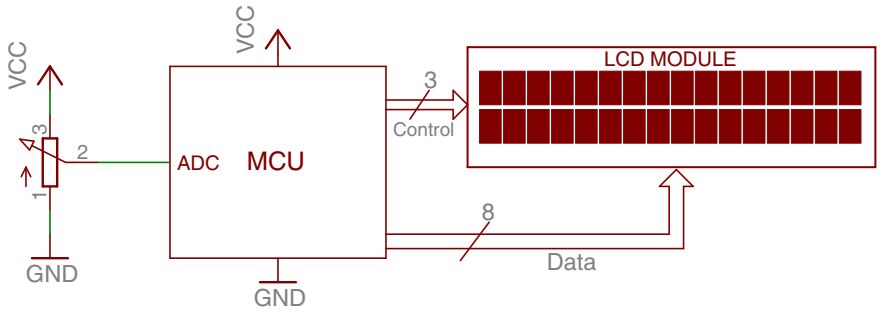


Figure 8.9: Schematic Potentiometer to ADC

2. Verify if the I/O port where the potentiometer is connected has ADC capabilities, if not, move it to an input that does.

3. Open your IDE.

4. Configure the ADC to use full resolution. Take into account the number of bits of your ADC module.

5. Write a code to read the ADC and display the read value in hexadecimal format into the LCD. Use a refresh ratio of 1 second to read the ADC value.

6. Compile and verify that your code does not contain any errors.

7. Run your code and verify if the value displayed in the LCD change when you turn the potentiometer.

8. Now, modify your code to display in the second line of the LCD the decimal voltage value that corresponds to the hexadecimal value being read. Complete the Table 8.3 and compare the voltage that appears on the LCD with the ADC input voltage for each case. Calculate the percentage of error for each measurement and explain the mismatch.

Table 8.3: ADC Values

Input Voltage	Decimal Value	Measured Voltage	% Error
VCC*0.1			
VCC*0.2			
VCC*0.3			
VCC*0.4			
VCC*0.5			
VCC*0.6			
VCC*0.7			
VCC*0.8			
VCC*0.9			
VCC*1.0			

8.2.3 Analog-Digital Dimmer

The objective of this part is controlling the brightness of an LED using a reference voltage from a potentiometer.

Follow the steps outlined below:

1. Connect the potentiometer and the LED according to the schematic in Figure 8.10. Take into account the same consideration, mentioned in the previous exercise, for the voltage source to be connected to the potentiometer.

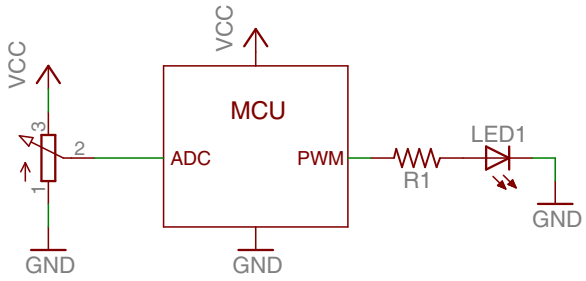


Figure 8.10: Schematic Potentiometer and LED to the MCU

2. Verify if the I/O port where the LED is connected has PWM capabilities, if not, move the LED to an output pin that has.

3. Open your IDE.

4. Configure the PWM module to produce a square signal of 1000Hz. Do not

forget to configure the Timer associated and the correct PWM mode of operation.

5. Configure the ADC to use the full resolution. Take into account the number of bits of your ADC Module.

6. Produce a code that modifies the LED brightness proportionally to the voltage value being read by the ADC module in your MCU. When the ADC reads a decimal value of 0V, the LED brightness must be set to 0% and when it reads a value that corresponds to VCC, the brigthness must be set to 100%.

7. Compile your code and verify that it does not contain any errors.

8. Run your code and verify if the brightness in the LED changes with the corresponding reference voltage.

9. Now, modify your code and circuit to include an LCD. The LED brightness level must appear on the first line and a warning message must appear on the second line when the lower or maximum brightness level are reached.

8.3 Complementary Tasks

8.3.1 Digital Temperature Meter

The activity consists of using the ADC module in your MCU and the temperature sensor LM35 to create a digital temperature meter. The system must have the capability of displaying the current ambient temperature into an LCD in Celsius (°C) or Fahrenheit (°F) units. To select the temperature unit, provide a pushbutton that allows toggling between the two temperature units. Take into consideration that the temperature range for the application is between 0°C (32°F) to 45°C (113°F). You must use the full range of your ADC to measure the temperature; The minimum value of your ADC must correspond to the lowest temperature and the maximum value must correspond to the highest temperature. You shall use a signal conditioner to adapt the signal from the temperature sensor to your ADC range. Investigate how to implement a signal conditioner (sc) with operational amplifiers. You can use the block diagram shown in Figure 8.11 as a reference for connecting your system.

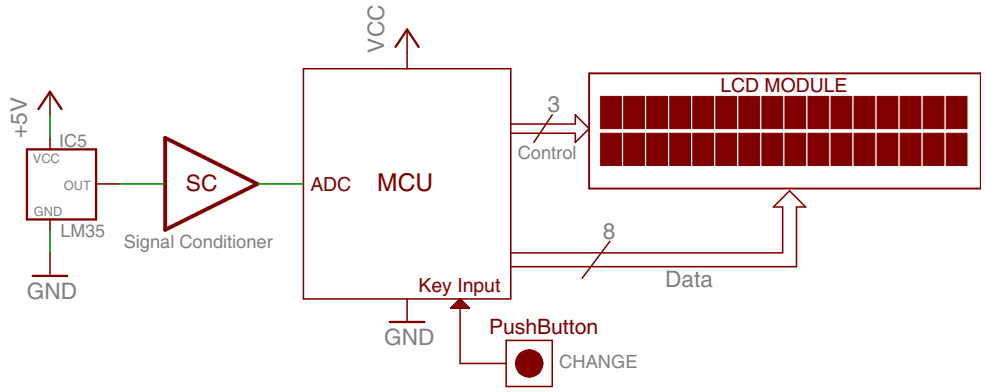


Figure 8.11: Digital temperature meter diagram

Presentation and Report

Each basic exercise and complementary task must be presented to the TA before the initiation of the next laboratory experiment. The demonstration must be made personally in the lab and including the hardware and software components needed for its completion.

An electronic report that includes the following information about the complementary task must be presented to the TA:

- Software plan and explanation (pseudocode or flowchart)
- Connection schematic with component calculations
- Code listing with comments.
- Additional information used to complete the task (Web pages, datasheets, books, etc.)

Appendix A

Using an MCU to Read Incremental Encoders

An incremental encoder is an electromechanical device that can be used to measure the movement, position, and displacement of a rotational or linearly moving mechanical component. In rotations parts, an incremental encoder can measure rotation directions and converts angular position and angular displacement into digital pulses. An incremental encoder has two outputs in quadrature, i.e., two outputs with a 90° phase shift between them. These outputs results from the optical or mechanical detection of two patterned tracks with a 90° geometric shift between them. Sample tracks and pulse trains are illustrate in Figure A.1.

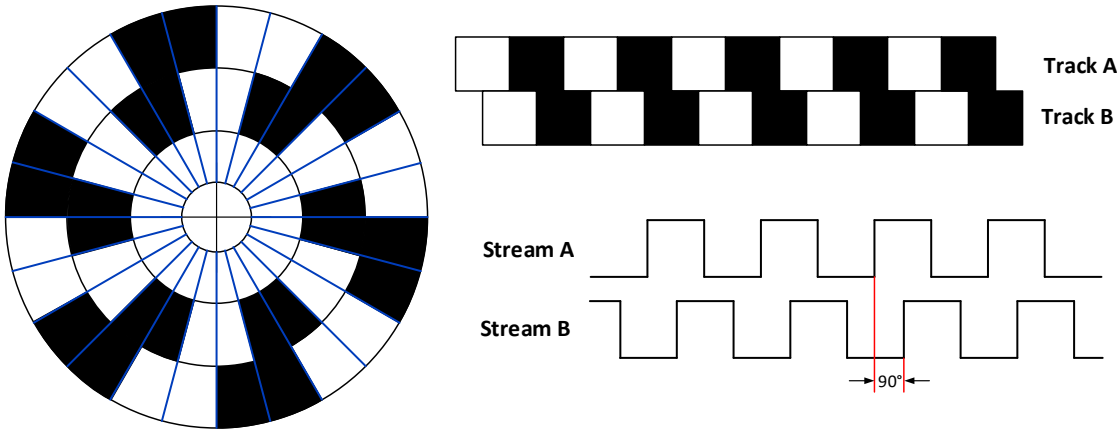


Figure A.1: Output waveforms of an incremental encoder

Detecting movement in either direction from the signal stream only requires determining the order of signal edges in streams A and B and encoding them with a binary code. The positions of the shaded regions generate a Gray binary code.

Figure A.2 shows the waveform streams A and B labeled for rotations sequences in counter clockwise (CCW) and clockwise (CW) directions. Red and blue arrows denote the edge sequences for each direction.

A sequence begins when signals A and B are in the same state. For example, in

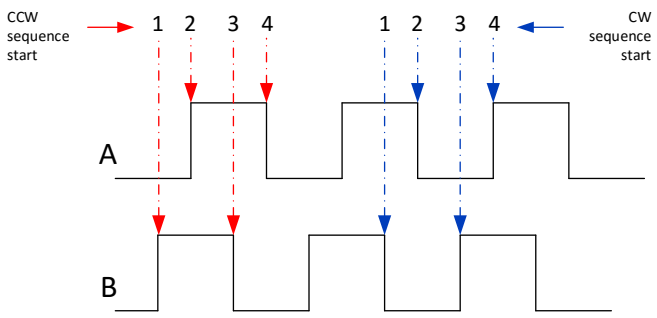


Figure A.2: Rotation sequence for CCW and CW directions

Figure 2, the CCW sequence begins at state '00' while the CW begins at '11'. New sequence values are detected through signal edges in either of the streams. A sequence ends when its four states have been generated. At this point a new sequence begins by repeating the codes from the initial states. Measuring the time between state changes allows obtaining the rotation speed. Tables A.1 and A.2 below show the sequences for CCW and CW rotations.

Table A.1: CCW sequence values

A	B	Value
0	0	0
0	1	1
1	1	3
1	0	2

Table A.2: CW sequence values

A	B	Value
1	1	3
0	1	1
0	0	0
1	0	2

To read the state values and detecting the sequence changes with an MCU it requires using two interrupts enabled i/O lines. The I/O lines levels indicate the state code and the interrupt capability allows detecting code changes. The interrupt needs to be configured so that it is triggered by any change in the encoder lines. This shall allow detecting both, the rising and falling edges of either signal. If the rotational speed were also interest, a timer could be used to measure the time between edges. Multiplying the time between edges (t_{edge}) by the number of the steps in the wheel yields the rotational period. For the sample wheel illustrated in Figure 1, two consecutive interrupts represent 1/24 of the wheel rotation, thus 24 times t_{edge} is one revolution.

To determine the rotation direction it is required to know the state of A and B before and after the occurrence of an edge. Combining the two two-bit codes, a four-bit identifier is obtained, which provides for any possible result in the sequence. Tables A.3 and A.4 list the values for the CCW and CW sequences. Not that each 4-bit code represents a state change in which only one bit changes before and after the edge. Recall that the encoder produces a Gray sequence, and therefore only one

bit is allowed to change between consecutive states.

Table A.3: 4-bit codes for CCW sequence

A_{old}	B_{old}	A_{new}	B_{new}	#(old:new)
0	0	0	1	1
0	1	1	1	7
1	1	1	0	14
1	0	0	0	8

Table A.4: 4-bit codes for CW sequence

A_{old}	B_{old}	A_{new}	B_{new}	#(old:new)
1	1	0	1	13
0	1	0	0	4
0	0	1	0	2
1	0	1	1	11

To write a software function for determining the rotation direction and wheel position, we could use a lookup table (LUT). The 4-bit values resulting from the before and after codes could be used as to index the table, and the entries would be either +1, -1, or 0 for representing CW, CCW or no movement conditions, respectively. Table A.5 shows such a LUT. Assuming the wheel started moving form a known "home" position (index hole and detector might be needed), adding the value fetched from the lookup table on each edge interrupt, we can have the absolute wheel position any time.

Table A.5: Lookup table to detect the direction of rotation and absolute position

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Value	0	+1	-1	0	-1	0	0	+1	+1	0	0	-1	0	-1	+1	0

Bibliography

[1] Enrique Palacios Municio, Fernando Remiro Domínguez, and Lucas J López Pérez. *Microcontrolador PIC16F84: desarrollo de proyectos.* México, DF:. Alfaomega;, 2009.

[2] Manuel Jimenez, Rogelio Palomera, and Isidoro Couvertier. *Introduction to Embedded Systems using Microcontrollers and the MSP430.* Springer, 2014.

[3] Eduardo García Breijo. *Compialdor C CCS y simulador Proteus para Microcontroladores PIC.* España, Barcelona: Marcombo S.A., 2009.

[4] Maxim Integrated. Understanding sar adcs: Their architecture and comparison with other adcs. Technical report, 2001.

[5] Firas Mohammed Ali. *Experiments in Computer and Microcontroller Applications.* Department of Electrical Engineering, University of Technology, 2013.

[6] Berkley Lab. *Electrical Safety Manual.* Lawrence Berkley National Laboratory, 2015.

[7] EUV. *High Voltage Safety Manual.* Colorado State University.

[8] NIST. *EEEL Safet Rules for Moderate and High Voltages.* National Institution of Standars and Technology, 2008.

[9] Environmental Health & Safety. *Electrical Hazards.* Standford University, 2004.

[10] Samuel M. Goldwasser. *Safety Guidlines for High Voltage and/or Line Powered Equipment*, 2010.

[11] Raymond M Fish, Leslie Alexander Geddes, and Charles F Babbs. *Medical and bioengineering aspects of electrical injuries.* Lawyers & Judges Publishing Company, 2003.

[12] American Heart Association, International Liaison Committee on Resuscitation, et al. Guidelines 2000 for cardiopulmonary resuscitation and emergency cardiovascular care, an international consensus of science. *Circulation*, 102, 2000.

[13] Department of Physics and Astronomy. *Electric Shock*. Georgia State University, 2014.

[14] Hd44780u (lcd-ii). HITACHI. https://www.sparkfun.com/datasheets/LCD/HD44780.pdf. Accessed: 2016-03-25.

[15] Lcd 16x2 (wh1602b2-tm1-et#). http://www.mouser.com/ds/2/272/-364177.pdf. Accessed: 2016-03-25.

[16] Ps1240p02ct3. TDK. https://product.tdk.com/info/en/catalog/datasheets/ef532_ps.pdf. Accessed: 2016-03-25.

[17] Dc56-11ewa. KINGBRIGHT. http://www.us.kingbright.com/images/catalog/spec/DC56-11EWA.pdf. Accessed: 2016-03-25.

[18] Wp154a4sureqbfzw. KINGBRIGHT. https://www.kingbrightusa.com/images/catalog/spec/WP154A4SUREQBFZGW.pdf. Accessed: 2016-03-25.

[19] Rpr-220. ROHM. http://rohmfs.rohm.com/en/products/databook/datasheet/opto/optical_sensor/photosensor/rpr-220.pdf. Accessed: 2016-03-25.

[20] Ds1307. MAXIM INTEGRATED. http://datasheets.maximintegrated.com/en/ds/DS1307.pdf. Accessed: 2016-03-25.

[21] Max3232. MAXIM INTEGRATED. http://pdfserv.maximintegrated.com/en/ds/MAX3222-MAX3241.pdf. Accessed: 2016-03-25.

[22] Parallax standard servo (#900-00005). PARALLAX. https://www.parallax.com/sites/default/files/downloads/900-00005-Standard-Servo-Product-Documentation-v2.2.pdf. Accessed: 2016-03-25.

[23] L293d. TEXAS INSTRUMENTS. http://www.mouser.pr/ProductDetail/Texas-Instruments/L293DNE/?qs=sGAEpiMZZMtYFXwiBRPs0wSafWlCmJbc. Accessed: 2016-03-25.

[24] Dac0808. TEXAS INSTRUMENTS. http://www.ti.com/lit/ds/symlink/dac0808.pdf. Accessed: 2016-03-25.

[25] Lm35. TEXAS INSTRUMENTS. http://www.ti.com/lit/ds/symlink/lm35.pdf. Accessed: 2016-03-25.