



COMSATS University Islamabad, Lahore Campus
Department of Electrical & Computer Engineering

CPE-342 – Microprocessor Systems & Interfacing

Lab Manual for Spring 2024 & Onwards

Lab Resource Person

Engr. Muhammad Usman

Theory Resource Person

Dr. Abbas Javed

Ms. Wajeeha Khan

Supervised By

Dr. M. Naeem Awais

Name: _____ Registration Number: *CIIT/* _____ - _____ - _____ */LHR*

Program: _____ Batch: _____

Semester _____

Revision History

Sr. No.	Update	Date	Performed by
1	Lab Manual Preparation	Sep-08	Engr. Muhammad Usman Rafique Engr. Muhammad Ali Raza
2	Lab Manual Review	Sep-08	Engr. Muhammad Usman Rafique
3	Layout modifications	Sep-08	Engr. Nauman Noor M.
4	Lab Manual Review	Sep-13	Dr. Muhammad Naeem Awais
5	Lab Manual Review	Dec-13	Dr. Muhammad Naeem Awais
6	Lab Manual Updating/modifications	Aug-15	Engr. Arslan Khalid
7	Lab Manual Review	Aug-15	Engr. Muhammad Usman Rafique Engr. Moazzam Ali Sahi
8	Lab Manual Review	Aug-16	Engr. Noman Naeem
9	Lab Manual Review	Sep-17	Engr. Mayyda Mukhtar
10	Lab Manual Preparation	Feb-18	Engr. Zainab Akhtar Engr. Noman Naeem
11	Lab Manual Review	Mar-18	Dr. Muhammad Naeem Shehzad
12	Lab Manual Format Changes	Sep-18	Engr. Muzamil Ahmad Zain
13	Lab Manual Review	Sep-19	Engr. Faisal Tariq
14	Lab Manual Update (removal of erroneous tables and programs)	Feb-20	Engr. Muhammad Usman Rafique Engr. Moazzam Ali Sahi
15	Rearrangement of Lab Experiments	Feb-21	Engr. Moazzam Ali Sahi Engr. M. Hassan Aslam
16	Lab Manual Modification	August-21	Engr. Nesruminallah
17	Rearrangement of Lab Experiments	Jan-22	Engr. Moazzam Ali Sahi Engr. M. Hassan Aslam

Preface

This course is intended to teach sixth semester undergraduate students of Electrical (Telecommunication) Engineering, Computer Engineering and Electrical (Electronics) Engineering about the working, architecture, programming instructions, memory system design and hardware interfacing with any general-purpose microprocessor. Course covers the Intel's 8086-88 microprocessors as example devices, but concepts can be applied to any general-purpose microprocessor. At the end of course, students must submit a semester project reflecting their understanding of the course contents. Project should also depict application of knowledge to solve some practical real-life problem. The project and associated viva-voce exam would carry considerable marks in the lab evaluation.

In this course, students shall have moderate understanding of working of microprocessor-based system on software and hardware basis. This shall be accomplished with the learning of assembly language of 8086-88 CPUs using EMU8086 self-assembler and emulator. Basic 8086-88 instructions such as data transfer, arithmetic, logic and control instructions are discussed, and example programs are designed as lab exercises to develop skills in students. In addition to that, students are also intended to get familiarized with STM32F4 microcontroller system. STM32F4 development board is an advanced level ARM-based embedded hardware that includes STM32F407 microcontroller, 3.5 mm audio jack, a USB interface to connect with host computer and a rich set of I/O ports to exchange digital data between STM32F4 and external hardware. Use of STM32F4 as a learning tool for grasping the concepts of microcontroller-based systems provide students opportunity to design, debug and implement solutions of real-life design problems. Design exercises are included as integral part in this manual that would test the ability to analyze, design and implement the solutions.

Books

Textbooks

1. The Intel Microprocessors by Barry B. Brey, 8th Edition, Pearson
2. Assembly Language Programming and Organization of the IBM PC by Ytha Yu and Charles Marut, 1992, 1st Edition, McGraw-Hill
3. Embedded Systems: Introduction to Arm® Cortex™-M Microcontrollers, by Jonathan W Valvano, Vol 1, 5th Edition, 2019, CreateSpace
4. The Intel Microprocessor Family Hardware and Software Principles and Applications By James L. Antonakos, 1st Edition, 2006, Delmar Cengage Learning

Reference Books for Manual

5. Hand-outs provided by instructor

Learning Outcomes

Theory CLOs

After successfully completing this course, the students will be able to:

1. CLO-1: To write the Intel-assembly code using the knowledge of programmer model, addressing mode and assembly language programming concepts. (PLO3-C5)
2. CLO-2: To integrate the memory, timer, I/O and PPI with microprocessor using address decoding techniques. (PLO3-C5)
3. CLO-3: To design digital system based on microprocessor using the knowledge of architecture, memory, timer, I/O and PPI interfacing. (PLO3-C5)

Lab CLOs

After successfully completing this course, the students will be able to:

4. CLO4: To explain and reproduce the Intel-assembly and STM32F407VG C-Programming codes using software and hardware platforms. (PLO5-P3)
5. CLO5: To design digital system using the knowledge of STM32F407VG C-Programming and peripherals. (PLO3-C5)
6. CLO6: To write effective report(s) of the assigned project. (PLO10-A2).
7. CLO7: To describe the impact of digital system on our society and environment using existing industrial standards (PLO7-C6)
8. CLO8: To justify the significance of designed project to the society using existing engineering practices (PLO6-C6).

CLOs – PLOs Mapping

CLO \ PLO	PLO1	PLO2	PLO3	PLO5	PLO6	PLO7	PLO10	Cognitive Domain	Affective Domain	Psychomotor Domain
CLO1			X					C5		
CLO2			X					C5		
CLO3			X					C5		
CLO4				X						P3
CLO5			X					C5		
CLO6							X		A2	
CLO7						X		C6		
CLO8					X			C6		

Lab CLOs – Lab Experiment Mapping

CLO \ Lab	Lab 1	Lab 2	Lab 3	Lab 4	Lab 5	Lab 6	Lab 7	Lab 8	Lab 9	Lab 10	Lab 11	Lab 12
CLO 4	P2	P2	P2	P2	P2	P2	P3	P3	P3	P3	P3	P3

Grading Policy

Complex Engineering Problem Lab [CEP]

Lab Assignments: <ul style="list-style-type: none"> i. Lab Assignment 1 Marks (Lab marks from Labs 1-3) ii. Lab Assignment 2 Marks (Lab marks from Labs 4-6) iii. Lab Assignment 3 Marks (Lab marks from Labs 7-9) iv. Lab Assignment 4 Marks (Lab marks from Labs 10-12) 	25%
Lab Mid Term = $0.5 * (\text{Lab Mid Term exam}) + 0.5 * (\text{average of lab evaluation of Lab 1-6})$	25%
Lab Terminal = $0.5 * (\text{Complex Engineering Problem}) + 0.375 * (\text{average of lab evaluation of Lab 7-12}) + 0.125 * (\text{average of lab evaluation of Lab 1-6})$	50%

Total (lab)

100%

List of Equipment

- STM32F4 Microcontroller
- STM32F407 Discovery Training Board
- Push buttons
- LEDs
- 4x3 keypad
- 16 X 2 LCD
- Buzzer
- STLM20 Temperature Sensor
- ORP-12 LDR

Software Resources

- EMU8086 emulator for writing, debugging and testing the assembly language programs
- Keil™ Compiler
- Microsoft Windows 7 Operating System

Lab Instructions

- This lab activity comprises of three parts: Pre-lab, Lab Tasks, Lab Report and Conclusion and Viva session.
- The students should perform and demonstrate each lab task separately for step-wise evaluation.
- Only those tasks that are completed during the allocated lab time will be credited to the students.
- Students are however encouraged to practice on their own in spare time for enhancing their skills.

Lab Report Instructions

All questions should be answered precisely to get maximum credit. Lab report must ensure following items:

- Lab Objectives
- Methodology
- Conclusion

Safety Instructions

1. Log-on with your username and password for your use only. Never share your username and password.
2. Chewing gum, food, drinks or apply cosmetics are not allowed in the computer lab or anywhere near a computer.
3. Respect the equipment. Do not remove or disconnect parts, cables, or labels.
4. Do not reconfigure the cabling/equipment without prior permission.
5. Internet use is limited to teacher assigned activities or classwork.
6. Personal Internet use for chat rooms, instant messaging (IM), or email is strictly prohibited. (This is against our Acceptable Use Policy.) Teachers must instruct students in Internet Safety.
7. Do not download or install any programs, games, or music. (This is against our Acceptable Use Policy.)
8. No Internet/Intranet gaming activities allowed.
9. Do not personalize the computer settings. (This includes desktop, screen saver, etc.)
10. Ask permission to print.
11. If by mistake you get to an inappropriate Internet site, turn off your monitor immediately and raise your hand.
12. CD-ROMs, thumb drives, or other multimedia equipment are for work only. Do not use them for playing music or other recreational activities.
13. Do not run programs that continue to execute after you log off.
14. Log-off, leave the computer ready for the next person to use. Pick-up your materials and push in the chair.
15. Do not leave a workstation or a login unattended. Do not leave processes in the background without prior approval from the Systems Manager. Do not lock your workstation for more than 20 minutes.
16. Lecturer/Lab Engineer must always remain in the lab and is responsible for discipline.

Table of Contents

Revision History	i
Preface	ii
Books	iii
Learning Outcomes	iii
CLOs – PLOs Mapping	iv
Lab CLOs – Lab Experiment Mapping	iv
Grading Policy	iv
List of Equipment	v
Software Resources	v
Lab Instructions	v
Lab Report Instructions	v
Safety Instructions	vi
LAB # 1	10
To Explain the Syntax of 8086-8088 and Show the Output of Data Transfer Instructions using EMU8086 Software Tool	10
Objectives	10
Pre-Lab Exercise	10
In-Lab Exercise	13
LAB # 2	16
To Explain and Show the Output of Arithmetic Instructions using EMU8086 Software Tool	16
Objectives	16
Pre-Lab Exercise	16
In-Lab Exercise	20
LAB # 3	23
To Explain and Show the Output of Logical Instructions using EMU8086 Software Tool	23
Objectives	23
Pre-Lab Exercise	23
In-Lab Exercise	28
LAB # 4	31
To Explain and Show the Output of Jump and Control Instructions using EMU8086 Software Tool	31
Objectives	31
Pre-Lab Exercise	31

In-Lab Exercise	35
LAB # 5.....	37
To Explain and Show the Output of BIOS Interrupt Programming using EMU8086 Software Tool	37
Objectives	37
Pre-Lab Exercise.....	37
In-Lab Exercise	40
LAB # 6.....	43
To Explain the Procedure for Using STM32F407, Keil™, MDK-ARM and STM32F407 Trainer Board	43
Objectives	43
Pre-Lab Exercise.....	43
LAB # 7.....	53
To Explain and Reproduce the Working of Switches and LED's on STM32F407 Trainer Board using C Programming.	53
Objectives	53
Pre-Lab Exercise.....	53
In-Lab Exercise	58
LAB # 8.....	61
To Explain and Reproduce the Working of 7- Segment on STM32F407 Trainer Board using C Programming.....	61
Objectives	61
Pre-Lab Exercise.....	61
In-Lab Exercise	62
LAB # 9.....	66
To Explain and Reproduce the Working of 16x2 LCD on STM32F407 Trainer Board using C Programming.	66
Objectives	66
Pre-Lab Exercise.....	66
In-Lab Exercise	69
LAB # 10.....	72
To Explain and Reproduce the Working of 4x3 Keypad on STM32F407 Trainer Board using C Programming. ..	72
Objectives	72
Pre-Lab Exercise.....	72
In-Lab Exercise	73
LAB # 11.....	78
To Explain and Reproduce the Working of Stepper and DC motor on STM32F407 Trainer Board using C Programming.	78
Objectives	78
Pre-Lab Exercise.....	78
In-Lab Exercise	82

LAB # 12.....	84
To Explain and Reproduce the Working of Temperature Sensor and LDR on STM32F407 Trainer Board using C Programming.	84
Objectives	84
Pre-Lab Exercise.....	84
In-Lab Exercise	86

LAB # 1

To Explain the Syntax of 8086-8088 and Show the Output of Data Transfer Instructions using EMU8086 Software Tool.

Objectives

- To explain the use of 8086-88 CPU Assembly Language Syntax.
- To show program output on EMU8086 Software.
- To display and manipulate the output of MOV Instruction using emu8086 software tool.
- To display and manipulate the output of XCHG Instruction using emu8086 software tool.

Pre-Lab Exercise

Read the details given below in order to comprehend the Assembly Language Syntax. Have Emu8086 software program installed on your PC and review how to compile/simulate designs within it.

Assembly Language Syntax

An assembly instruction can be divided into two fields named “operation” and “operand”. Operation refers to “*what to do*” and operand(s) refer to “*with which to do*”.

For Example:

MOV AX, 5498

In this example, MOV is the operation (also termed as instruction) and AX and 5498 are called operands. Operands are separated with “,” between them while instruction and its operand are separated with space in between them.

One point is to be noted that assembly language is NOT case-sensitive. It means an instruction written either in uppercase or lowercase letters is same. For example, above instruction can be written as.

mov ax,5498.

Representing numbers and characters in assembly language

A numerical value can be represented by specifying its radix. In general, letter B (or b) is appended with stream of 1 and 0 to represent it as binary value. Similarly, letter H (or h) represents that number it follows is a hexadecimal (also termed as ‘hex’) number. Decimal numbers do not follow any letter. One thing is to be noted that in assembly language, a hex number that starts with A to F must have a 0 on the left side. For example, representing A456H will be invalid while 0A456H will be the valid value.

Examples are 1100101B, 5645, A45DH (should be written as 0A45DH in an instruction) etc.

Every information in the microprocessor is dealt in binary. Same is true for characters. A character, such as A, a, &, %, +, t, 8, etc., are represented with an 8-bit binary code called “ASCII” codes. These one-byte long codes represent each of the character seen on the keyboard of the computer. In

In addition to that, every key on the keyboard has its associated ASCII code. As it is difficult to memorize all ASCII codes (256 in total), there is an easy way to do that. Any character that is enclosed in “ ” will be treated as its corresponding ASCII. For example, digit 6 alone represents a numerical value 6 that we use in routine. But when it is used as enclosed in “ ” like ‘6’, it becomes the ASCII for digit 6 which is 54.

Assembler directive

An assembler directive is an English word that appears in assembly language programs. It is not an instruction in assembly language but facilitates the program to complete some specific tasks through assembler. Assembler directives may vary from assembler to assembler, but instructions remain unchanged for a given microprocessor on its specific assembler. Examples of assembler directives are EQU, DB, and DW etc.

Defining variables, arrays and constants

Variables

The syntax of assembler directives that define variables is given below:

```
name DB initial_value
```

The assembler directive DB in above statement (note that it is not an instruction) stands for “*Define Byte*”. Hence a variable with name ‘name’ is declared with one-byte storage. Consider defining a variable with name VAR1 as a one-byte variable and its initializing value is set to be 100. Then following statement will do it.

```
VAR1 DB 100
```

Similarly, a 16-bit variable can also be defined using assembler directive ‘DW’ (*Define Word*).

For example:

```
VAR1 DW 1897H
```

```
MY_VAL DW 101101110B
```

Important Note

There are some rules that must be followed when selecting the variable name. Variable name should not be an instruction such as AND, XOR, CALL etc. Variable name should not start with symbols such as +, -, % etc., though _ (underscore can be used as starting character). Variable names should not have space in between. For example, MY VARIABLE is an incorrect variable name. If label name is to be separated for easy reading and understanding, then _ should be used. For example, MY_VARIABLE will be a valid replacement of MY VARIABLE.

Arrays or Strings

In assembly language, an array (also called string) is just a sequence of memory bytes or words. For example, in order to define a 3-byte array with name ‘arr_3byte’ whose initial values are 10h, 20h, 30h, we can write as:

```
arr_3byte DB 10h,20h, 30h
```

Note that all the values in an array have the same size.

Constant

To assign a name to a constant, assembler directive EQU (equate) is used. The syntax is:

```
name EQU constant
```

For example, the following statement associates the name LF with constant 0AH.

```
LF EQU 0AH
```

Now LF could be used in place of constant 0AH which is ASCII value of Line Feed character.

Comments

Comments in the assembly language have the same understanding as they are in any other programming language. They help programmer identify the purpose of statement, instruction or part of the program. In assembly language, anything followed by the symbol ';' will be a comment. In assembly language comments are always single line. Hence in order to extend the comment to next line, it must be started with symbol ';'.

For example:

```
MOV BX, 5698 ; BX is loaded with 5698
```

Read the details given below in order to comprehend the basic operation of MOV and XCHG instructions. Study in detail and become familiar with the various ways and combinations in which these two instructions can be used.

MOV Instruction

Pronounced as 'move', MOV transfers a constant, contents of memory or register to a register or memory. MOV operates on two operands, called source and destination. Format of the MOV is

MOV destination, source

Some examples of MOV with different nature of operands are given below.

MOV AX, DX	Transfers contents of DX to AX, leaving DX unchanged
MOV AL, CH	Transfers contents of CH to AL, leaving CH unchanged
MOV BL, CX	Not allowed
MOV AH, 56	56 decimal is transferred to AH
MOV BX, 0ABCDH	Transfers ABCD hexadecimal to BX
MOV 67H, BL	Not allowed
MOV DS, 3467H	Not allowed
MOV DS, BX	Loads DS with contents of BX, leaving BX unchanged
MOV [2345H], AH	Transfers contents of AH to the memory location
MOV [DI], 56H	Transfers 56H to the memory whose offset is stored in DI
MOV [DI], [SI]	Memory to memory move is not allowed

XCHG Instruction

Pronounced as ‘exchange’, XCHG swaps the contents of its operands. Operand can both be registers or a register and a memory location. Examples are:

XCHG AX, BX

XCHG AL, CL

XCHG 32H, BH

Invalid operation

XCHG [2345H], [SI]

Invalid operation

Table 1-1: Legal Combinations for MOV instruction

MOV Source Operand	Destination Operand			
	General Register	Segment Register	Memory Location	Constant
General Register	Yes	Yes	Yes	No
Segment Register	Yes	No	Yes	No
Memory Location	Yes	Yes	No	No
Constant	Yes	No	Yes	No

Table 1-2: Legal Combinations for XCHG/ADD & SUB instructions

XCHG Source Operand	Destination Operand	
	General Register	Memory Location
General Register	Yes	Yes
Memory Location	Yes	No
ADD/SUB Source Operand	Destination Operand	
	General Register	Memory Location
General Register	yes	yes
Memory Location	yes	no
Constant	yes	yes

In-Lab Exercise

Task 1: Fill the table after compiling the code

Write the following program in editor, compile and emulate it. Also fill the table.

- | | | |
|------------------|-----------------|-------------------|
| 1. MOV AX, 16H | 6. MOV CX, SI | 11. MOV AX, 1200H |
| 2. MOV BH, 36 | 7. MOV DS, CX | 12. MOV DS, AX |
| 3. MOV SI, 2345H | 8. MOV AH, 0CDH | 13. MOV |
| 4. MOV BL, 24H | 9. XCHG AH, BH | [0004H], 78H |
| 5. MOV AX, BX | 10. XCHG CX, BX | |

S. No	Instruction	AX		BX		CX		DS	SI
		AH	AL	BH	BL	CH	CL		
1.	MOV AX, 0A9H								
2.	MOV BH, 36								
3.	MOV SI, 2345H								
4.	MOV BL, 24H								
5.	MOV AX, BX								
6.	MOV CX, SI								
7.	MOV DS, CX								
8.	MOV AH, 0CDH								
9.	XCHG AH, BH								
10.	XCHG CX, BX								
11.	MOV AX, 1200H								
12.	MOV DS, AX								
13.	MOV [0004H], 78H								

Task 2: Without using XCHG command, write 8086/8088 Assembly language program that swap the contents of

- SP and DI registers
- SS and DS registers
- Memory locations ABCDH: 2345H and 1234H: 78DEH

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____ **Date:** _____

LAB # 2

To Explain and Show the Output of Arithmetic Instructions using EMU8086 Software Tool

Objectives

- To explain and show the output of ADD, SUB, MUL, and DIV Instruction using emu8086 software tool.
- To explain and show the output of INC, DEC, and NEG Instruction using emu8086 software tool.

Pre-Lab Exercise

Read the details given below in order to comprehend the basic operation of Arithmetic instructions. Arithmetic instructions includes addition, subtraction, multiplication and division. Study in detail and become familiar with the various ways and combinations in which these instructions can be used.

ADD Instruction

ADD instruction adds values existing in two registers, register and memory, immediate and memory, immediate and register. To add two values existing in AL and BH register, ADD is used as:

```
ADD  AL,  BH
```

This instruction adds the contents of AL and BH registers, leaving sum in AL and BH unchanged. ADD is applied, for example, on CX and DX in the same way. Consider CX is containing 18 and DX containing 25 before executing the following instruction.

```
ADD  CX,  DX
```

After this instruction is executed, CX is left with 43, DX with 25 and CF with 0.

Any 8-bit register (AH, AL, BH, BL, CH, CL, DH, DL) can be used with ADD. For 16-bit addition, ADD can be applied on AX, BX, CX, DX, SI and DI. You are encouraged to check the validity of ADD instruction for another 16-bit registers yourself.

ADC Instruction

ADC, pronounced as Add with Carry, instruction adds the content of operands along with value in carry flag that exists in it at the instant instruction ADC is executed. Consider SI contains 15678, DI contains 325 and CF is set. After executing following instruction,

```
ADC  SI,  DI
```

SI is left with 16004 (3E84H), DI with 325 (145H) and CF is cleared as the latest result does not produce overflow condition.

SUB Instruction

SUB instruction subtracts the contents of operand2 from operand1 and leaves the difference in operand1, with operand2 unchanged.

SUB operand1, operand2

Consider content of AH are to be subtracted from content of DL then SUB can be used as,

SUB DL, AH

For 16-bit subtraction,

SUB DX, DI

SUB BX, AX

Any 8-bit register (AH, AL, BH, BL, CH, CL, DH, DL) can be used with SUB. For 16-bit subtraction, SUB can be applied on AX, BX, CX, DX, SI and DI. You are encouraged to check the validity of SUB instruction for another 16-bit registers yourself.

SBB Instruction

SBB, pronounced as Subtract with Borrow, subtracts the contents of operand2 and CF from the content of operand1 and leaves the result in operand1. Operand2 is left unchanged, and CF depends upon whether most significant bit in operand1 required a borrow bit or not. If borrow was required CF is set, otherwise cleared. Format of SBB is,

SBB operand1, operand2

Consider CF is set, BX = 67ABH and DX = 100H before executing following instruction.

SBB BX, DX

After execution, BX is left with 66AAH, DX with 100H and CF is cleared.

INC Instruction

Increment instruction INC adds 1 to the contents of its operand. It can be applied on any 8-bit register (AH, AL, BH, BL, CH, CL, DH, DL). For 16-bit increment, INC can be applied on AX, BX, CX, DX, SI and DI. You are encouraged to check the validity of INC instruction for other 16-bit registers yourself.

Consider AL contains 97H before execution of following intrusion.

INC AL

After execution, AL is left with 98H.

DEC Instruction

Decrement instruction DEC subtracts 1 from the contents of its operand. It can be applied on any 8-bit register (AH, AL, BH, BL, CH, CL, DH, DL). For 16-bit decrement, DEC can be applied on AX, BX, CX, DX, SI and DI. You are encouraged to check the validity of DEC instruction for other 16-bit registers yourself.

Consider DI contains 55968 before execution of following instruction.

```
DEC    DI    ; after execution, DI is left with 55967.
```

NEG Instruction

NEG instruction is single operand instruction that leaves the operand with 2's complement of its original data. For example, register AX contains 12656 (3170H). After executing NEG on AX, new contents of AX will be CE90H which is -12656. NEG can be applied on 8-bit registers, 16-bit registers or memory location. Consider,

```
NEG    AX
```

MUL Instruction

MUL carry out multiplication on two operands in 8086-88 CPU. MUL is a single operand instruction whereas other operand is assumed to be in a specified register. MUL can be applied on two 8-bit values producing 16-bit result, and on 16-bit values producing 32-bit result.

For 8-bit multiplication, one operand is assumed in AL register whereas other is the part of instruction. For example, in order to multiply 230 with 165, one of these values must be in AL register. Other operand can be in any 8-bit register or in memory location. Consider following code.

```
MOV    AL, 230
```

```
MOV    BL, 165
```

```
MUL    BL
```

After executing these three instructions, 16-bit result (37950 in this case) will be found in AX register, while content of BL is left unchanged.

For 16-bit multiplication, one operand is assumed in AX register whereas other is the part of instruction. For example, in order to multiply 22330 with 10365, one of these values must be in AX register. Other operand can be in any 16-bit register. Consider following code.

```
MOV    AX, 22330
```

```
MOV    BX, 10365
```

```
MUL    BX
```

After executing these three instructions, 32-bit result (231450450 in this case) will be found in DX-AX registers. DX register contains most significant 16 bits (from bit 16 to bit 31) and AX contains least significant 16 bits (from bit 0 to bit 15). Content of BX are left unchanged.

DIV Instruction

To do division in 8086-88, DIV instruction is provided by the instruction set of 8086-88 CPU. Like MUL, DIV can be done on 8-bit data and on 16-bit data. In 8-bit division, dividend (numerator) is stored in AX register while 8-bit divisor (denominator) is stored in 8-bit register or in memory. After executing DIV, 8-bit quotient moves in AL while 8-bit remainder moves in AH.

Consider the following code that divides 2334 by 167.

```
MOV  AX, 2334
MOV  BH, 167
DIV  BH
```

After executing above three instructions, AL contains DH (13 decimal, the quotient) and AH contains A3H (163 decimal, the remainder).

In 16-bit division, 32-bit dividend (numerator) is stored in DX-AX registers such that most significant 16 bits are in DX and least significant 16 bits are in AX. 16-bit divisor (denominator) is stored in a register. After executing DIV, 16-bit quotient moves in AX while 16-bit remainder moves in DX.

Consider the following code that divides 235634 (39872H) by 45667 (B263H).

```
MOV  AX, 9872H
MOV  DX, 3H
MOV  BX, 0B263H
DIV  BX
```

After executing above three instructions, AX contains 5H and DX contains 1C83H.

Note:

An interrupt is occurred if division is achieving such that ‘divided by zero’ is carried out or a very large number is divided by a very small number that cause quotient greater than the range of register.

In-Lab Exercise

Task 1: Fill the table after compiling the code

Type in the following program in editor of EMU8086 and fill in the table.

Instruction	AH	AL	BH	BL	DH	DL	CF
MOV AX, 1456							
MOV BX, 238							
ADD AX, BX							
SUB AX, 134							
MOV AL, 33H							
MOV AH, 54H							
MUL AH							
ADC AH, AL							
MOV BH, 14							
MUL BH							
DIV BH							

Task 2: Write an assembly language code to compute the cube of a number

Write an assembly language program that cube the 8-bit number found in DL. Load DL with 7 initially, and make sure that your result is a 16-bit number.

Task 3: Write an assembly code that converts the Fahrenheit reading of temperature into the equivalent Celsius reading

Write an assembly language program that converts the Fahrenheit reading of temperature into the equivalent Celsius reading. Celsius reading is found in AH register and Fahrenheit reading is to be stored in AL register.

Hint: Formula for conversion is as follows

$$C = 5/9 \times (F - 32)$$

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____ **Date:** _____

LAB # 3

To Explain and Show the Output of Logical Instructions using EMU8086 Software Tool

Objectives

- To explain and show the output of AND, OR, NOT, and XOR Instruction using emu8086 software tool.
- To explain and show the output of RCL, ROR, ROL, and RCR Instruction using emu8086 software tool.
- To explain and show the output of SHL, SHR, and SAL Instruction using emu8086 software tool.

Pre-Lab Exercise

Read the details given below in order to comprehend the basic operation of logical instructions. It includes different operations such as AND, OR, XOR, NOT, Shifting and rotating of data. Study in detail and become familiar with the various ways and combinations in which these instructions can be used.

AND Instruction

AND instruction do logical AND on two 8-bit data or on two 16-bit data. For example, in order to logically AND contents of BH and DL, AND is used as:

```
AND  BH, DL
```

This instruction logically ANDs the contents of BH and DL registers, leaving result in BH and DL remains unchanged. AND can be applied on 16-bit data stored in registers in the similar way. Any 8-bit register (AH, AL, BH, BL, CH, CL, DH, DL) can be used with AND. For 16-bit AND, instruction can be applied on AX, BX, CX, DX, SI and DI. You are encouraged to check the validity of AND instruction for other 16-bit registers yourself.

OR Instruction

OR instruction executes logical OR on two 8-bit data or on two 16-bit data. For example, in order to logically OR contents of BH and DL, OR is used as:

```
OR   BH, DL
```

This instruction logically ORs the contents of BH and DL registers, leaving result in BH and DL remains unchanged. OR can be applied on 16-bit data stored in registers in the similar way. Any 8-bit register (AH, AL, BH, BL, CH, CL, DH, DL) can be used with OR. For 16-bit OR, instruction can be applied on AX, BX, CX, DX, SI and DI. You are encouraged to check the validity of OR instruction for other 16-bit registers yourself.

XOR Instruction

XOR instruction executes exclusive OR on two 8-bit data or on two 16-bit data. For example, in order to exclusive OR contents of BH and DL, XOR is used as:

```
XOR  BH, DL
```

This instruction XORs the contents of BH and DL registers, leaving result in BH and DL remains unchanged. XOR can be applied on 16-bit data stored in registers in the similar way. Any 8-bit register (AH, AL, BH, BL, CH, CL, DH, DL) can be used with XOR. For 16-bit XOR, instruction can be applied on AX, BX, CX, DX, SI and DI. You are encouraged to check the validity of XOR instruction for other 16-bit registers yourself.

NOT Instruction

NOT instruction takes one complement of operand that can be 8-bit data or 16-bit data. For example, in order to apply NOT on AH register, following syntax is used.

```
NOT  AH
```

After executing this instruction, AH is left with complement of its contents prior to execute this instruction.

Any 8-bit register (AH, AL, BH, BL, CH, CL, DH, DL) can be used with NOT. For 16-bit NOT, it can be applied on AX, BX, CX, DX, SI and DI. You are encouraged to check the validity of SUB instruction for other 16-bit registers yourself.

RCL Instruction

There are four different types of rotate instructions. RCL instruction stands for “Rotate Left through Carry” positions the bits in a register or in memory location according to following scenario.

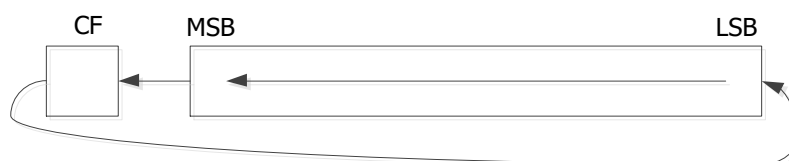


Figure 3-1 Rotate left through carry

Previous value of CF goes to LSB of operand and MSB of operand goes into the CF. Bits in the operand are shifted left by one bit at a time.

Applying RCL on AH, for example, is:

```
RCL  BL, 6
```

Above instruction accomplish RCL operation on BL for six times. If number of shift operations is a variable value, then it is to be placed in CL. Result is stored back in BL.

ROL Instruction

The rotate instruction ROL stands for “Rotate Left”. ROL positions the bits in a register or in memory location according to following scenario.

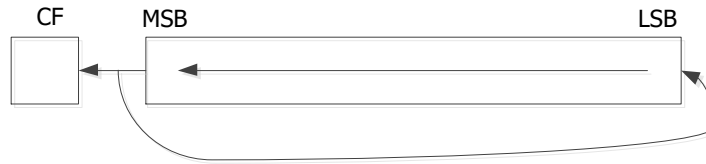


Figure 3-2 Rotate Left operation

Previous value of MSB of operand goes into the CF and the same also enters in to the LSB position, with each bit in operand shifting left by one bit at a time.

Applying ROL on SI, for example, is:

```
RCL SI, 14
```

Above instruction executes ROL operation on SI for fourteen times. If number of shift operations is a variable value, then it is to be placed in CL. Result is stored back in SI.

RCR Instruction

RCR, “Rotate Right through Carry”, behaves in reverse of RCL. Function of RCR is represented in following diagram.

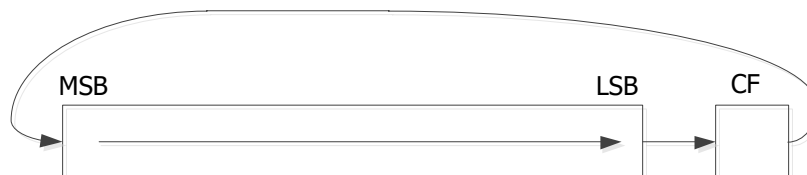


Figure 3-3 Rotate right through carry

Previous value of CF goes to MSB of operand and LSB of operand goes into the CF. Bits in the operand are shifted right by one bit at a time.

Consider following instruction.

```
RCR AH, CL
```

After execution, original content of AH register is rotated right times the value in CL.

Result is stored back in AH.

ROR Instruction

The rotate instruction ROR stands for “Rotate Right”. ROR positions the bits in a register or in memory location according to following scenario.

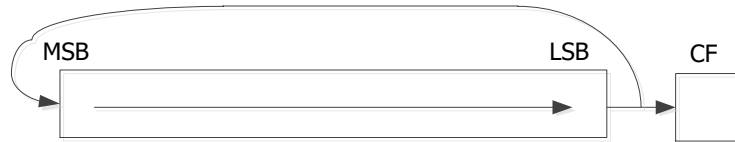


Figure 3-4 Rotate right instruction

Previous value of LSB of operand goes into the CF and the same also enters in to the MSB position, with each bit in operand shifting right by one bit at a time.

Consider following instruction.

```
ROR [23ABH], CL
```

After execution, original content of memory location with offset address 23ABH are rotated right though carry times the value in CL. Result is stored back in memory location with offset address 23ABH.

SHL/SAL Instructions

Like rotate instructions, shift instructions are four in number. Instruction SHL, abbreviated from “Shift Left”, shifts the bits in operand left one bit at a time. Instruction SAL “Shift Arithmetic Left” behaves exactly identical to SHL as the MSB, that reflects the positivity or negativity of value is altered in shift left operation.

Bit shifted out from MSB of operand enters into the CF. 0 is entered in the LSB position for a single SHL or SAL operation. This operation is diagrammatically represented below.



Figure 3-5 Shift left instruction

Consider following instructions.

```
SHL BX, 4
```

```
SAL DI, 6
```

The instruction SHL BX, 4 shifts the contents of BX four bits left, leaving result in BX. If number of shifts is a variable value, then it is to be stored in CL. SHL can be done on 8-bit register, 16-bit register or on a memory location.

The instruction SAL DI, 6 behaves exactly similar to above discussed instruction with difference that content of DI are shifted six bits left now.

SHR Instruction

Instruction SHR, abbreviated of “Shift Right”, shifts the bits in operand right one bit at a time. Bit shifted out from LSB of operand enters into the CF. 0 is entered in the MSB position for a single SHR operation. This operation is diagrammatically represented below.



Figure 3-6 Shift right instruction

Consider following instruction.

```
SHR    DX, CL
```

After execution, original content of DX is shifted right times the value in CL. Result is stored back in DX register.

SAR Instruction

Instruction SAR, abbreviated of “Shift Arithmetic Right”, shifts the bits in operand right one bit at a time. Bit shifted out from LSB of operand enters into the CF with MSB remains unchanged. This retains the positivity or negativity of value in operand. This operation is diagrammatically represented below.

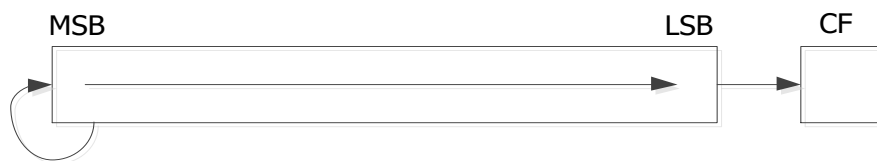


Figure 3-7 Shift right arithmetic

Consider following instruction.

```
SAR    DH, CL
```

After execution, original content of DH is shifted right times the value in CL with MSB preserved. Result is stored back in DH register.

In-Lab Exercise

Task 1: Write the following assembly program in editor and execute it

Instruction	AH	AL	BH	BL	DH	DL	CL	CF
MOV AX,1456								
MOV BX,0FF00H								
XOR AX,AX								
MOV BX,2300H								
MOV DS,BX								
MOV [1601H],25H								
OR BH,[1601H]								
MOV CL,3								
MOV DX,23DCH								
SHL DX,CL								
SHR DX,3								
SAL DX,1								
MOV AH,250								
ADD AH,10								
RCR AH,2								

Task 2: Write and assembly program to count the number of 1's

Write an assembly language program that counts the number of '1's in a byte residing in CL register. Store the counted number in DH register.

Task 3: Bits' manipulation

Write an assembly language program to perform the following tasks:

- a) Set the leftmost 4 bits of AX
- b) Clear the rightmost 3 bits of AX
- c) Invert the bits 5,7 and 9 of AX.

Task 4: Bits' masking

Write an assembly language program that clears any bit (from bit0 to bit15) in AX register, leaving other bits unchanged. Number of bit that is to be cleared is stored in CL register.

Hint: If the number 9 is stored in CL register, it means 9th bit of AX should be cleared

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____ **Date:** _____

LAB # 4

To Explain and Show the Output of Jump and Control Instructions using EMU8086 Software Tool

Objectives

- To explain and show the output of conditional and unconditional jump Instruction using emu8086 software tool.
- To organize the assembly programs based on subroutines using emu8086 software tool.
- To explain and show working of label using assembly language program.

Pre-Lab Exercise

Read the details given below in order to comprehend the basic operation of Jump and Control, Loop and Call instructions. Study in detail and become familiar with the various ways and combinations in which these instructions can be used.

Labels

When an instruction is stored in ROM, it is stored in individual location out of many available locations. Each location has its unique identification code called its address. Label is basically the address on which subsequent instruction is stored in ROM. For convenience, programmer can use an alphanumeric string to represent the address of any instruction in program. Consider following piece of assembly code written for 8088 CPU.

```

                MOV    AX,  BX
MSI :
                MOV    BX,  0ABCDH
                OR     BL,  35H
                AND    BH,  BL
                HLT

```

The string MSI is label, and its associated instruction is MOV BX, 0ABCDH. Label name must end with ‘:’ character. When this program would be assembled, label MSI will be translated to the address of MOV BX, 0ABCDH instruction, whatever it might be. Note that instructions in ROM are placed in the same order as they appear in program code. There are some rules that must be followed when selecting the label name. Label name should not be an instruction such as AND, XOR, CALL etc. Label name should not start with symbols such as +, -, % etc., though _ (underscore can be used as starting character). Label names should not have space in between. For example, MY LABEL is an incorrect label name. If label name is to be separated for easy reading and understanding, then _ should be used. For example, MY_LABEL will be a valid replacement of MY LABEL.

Label names should be self-descriptive. They should depict the purpose for which they are defined.

Subroutines

A subroutine is a chunk of code that executes a special operation in entire program. A subroutine starts with a label name and end in RET (return) instruction. Consider following piece of code.


```

MY_SUB:
    MOV    AX, 5623H
    ADD    AX, 1200H
    MOV    DX, 2255H
    OR     DX, AX
    RET

```

The above piece of code is written to demonstrate the concept of subroutine. It may not be executing some useful operation but points to consider are the start of subroutine with label MY_SUB and its end with RET instruction.

JMP Instruction

Abbreviation of jump, JMP instruction directs the program flow to the instruction associated with the label name that follows the JMP instruction. For example, consider following use of JMP instruction.

```
JMP    CIIT
```

This instruction diverts the program flow to the instruction which is labeled as CIIT. Program starts executing from that instruction and flows in a normal sequential way onwards. Consider following piece of code.

```

    MOV    CL, BH
    AND    BH, 32H
    JMP    MSI_LAB
    XOR    AX, AX
    DEC    BX
MSI_LAB:
    MOV    AH, 16
    MOV    CL, 3
    SHR    AH, CL

```

Program execution starts from line 1 and goes to line 3 according to normal flow. In line 3, instruction JMP is encountered that directs the program flow to instruction in line 7. Remember that a label name is not an instruction. Once program flow is directed to line 7 then it continues execution subsequently by executing instructions in line 8 and then in line 9 and so on. Redirecting the program from one instruction to another which is not in subsequence of previous instruction is called jumping. As in this example program jumps to MSI_LAB label with no dependency on results of last instruction, it is called unconditional jumping.

CMP Instruction

CMP (compare) instruction compares its operands through subtraction. CMP does not modify the operands rather it updates the flag register only. Consider use of CMP given below.

```
CMP    BL, AL
```

This instruction subtracts the contents of AL register from contents of BL register, but result is not stored anywhere. Only flags are updated according to one of three possible outcomes which are “Is AL greater than BL”, “Is AL less than BL” and “Is AL is equal to BL”.

JZ Instruction

JZ (Jump if Zero) is conditional jump instruction. It is used as follows.

JZ Label name

If result of last operation (operation completed just before using JZ instruction) is zero, zero flag (ZF) would have been set, otherwise cleared. This instruction checks the ZF and if it is found set, then program is directed to the instruction associated with label name that is used in JZ instruction. Otherwise, jumping is skipped and instruction following the JZ is executed. Consider following piece of code.

```

MOV  AX, 3456
MOV  BX, AX
SUB  AX, BX
JZ   ZERO
AND  BX, 2345H
MOV  SI, BX
ZERO:
MOV  DI, SI
AND  DI, AX

```

When instruction in line 4 is to be executed, it will be checked if ZF is set or not. As due to the instructions in line 1 to line 3 cause $ZF = 1$, jump to label ZF will be taken and instruction in line 8 shall be executed after instruction in line 4. If ZF would have not been set at the time of execution of instruction in line 4, then next instruction would be of line 5. Note that as label name is not an instruction, in above code, instruction in line 8 would be executed after instruction in line 6.

JNZ Instruction

JNZ (jump if Not Zero) behaves oppositely of JZ. Jump would be taken to specified label if ZF is cleared and will not be taken if ZF is set at the time of execution of JNZ.

JC Instruction

JC (Jump if Carry) directs the program flow to the specified label if CF is set at the time of execution of JC instruction. Jumping will be skipped otherwise.

JNC Instruction

JNC (Jump if No Carry) directs the program flow to the specified label if CF is cleared at the time of execution of JNC instruction. Jumping will be skipped otherwise.

JG Instruction

JG (Jump if Greater) instructions deals operands of CMP instruction as signed numbers. This instruction is generally used in conjunction with CMP instruction. Upon comparison, if operand 1 is greater than operand 2, then JG will direct the flow to the label associated with it.

```
CMP operand 1, operand 2
JG label
```

JGE Instruction

JGE (Jump if Greater or Equal) instructions also deals operands of CMP instruction as signed numbers. This instruction is generally used in conjunction with CMP instruction. Upon comparison, if operand 1 is greater than or equal to operand 2, then JG will direct the flow to the label associated with it.

```
CMP operand 1, operand 2
JGE label
```

JL Instruction

JL (Jump if Less) instructions also deals operands of CMP instruction as signed numbers. This instruction is generally used in conjunction with CMP instruction. Upon comparison, if operand 1 is less than operand 2, then JL will direct the flow to the label associated with it.

```
CMP operand 1, operand 2
JL label
```

JLE Instruction

JLE (Jump if Less or Equal) instructions also deals operands of CMP instruction as signed numbers. This instruction is generally used in conjunction with CMP instruction. Upon comparison, if operand 1 is less than or equal to operand 2, then JLE will direct the flow to the label associated with it.

```
CMP operand 1, operand 2
JLE label
```

CALL Instruction

CALL is used to direct program to a subroutine. Consider following piece of code.

```
MOV CL, BH
AND BH, 32H
CALL MSI_LAB
XOR AX, AX
DEC BX
MSI_LAB:
MOV AH, 16
MOV CL, 3
RET
```

When program flow executes CALL instruction in line 3, it is directed to the instruction in line 7 from where it starts execution sequentially. When flow encounters the RET instruction in line 9, it directs program back to the instruction following immediately after the most recent executed CALL instruction. In this example, after executing RET instruction in subroutine MSI_LAB, instruction in

line 4 is executed and program flows onwards normally. This is called “returning from call”. Another CALL instruction can be used without returning from a call. This is called “nested calling”

RET Instruction

RET (Return) instruction, normally placed at the end of a subroutine to terminate it, brings the control back to the instruction that follows immediately after the CALL instruction using which the current subroutine was called.

LOOP Instruction

LOOP instruction moves to prescribed label iteratively. Value in the CX register determines the number of iterations. With each LOOP execution, CX decrements automatically. LOOP keeps on executing as long as CX reaches zero.

Consider following code.

```

        MOV    CX, 100
        XOR    AX, AX
HERE:
        ADD    AX, 1
        LOOP   HERE

```

In line 1, register CX is loaded with 100 count that determines the desired iterations of LOOP instruction. Line 2 clears the AX register. Note that this instruction has nothing to do with LOOP iterations. When line 5 is executed, flow jumps to the label HERE, CX is decremented, instruction in line 4 is executed and then LOOP HERE executes again. Now CX is decremented again (leaving 98 in it now), instruction in line 4 is executed and LOOP HERE executes again. This procedure is repeated unless CX becomes 0.

In-Lab Exercise

Task 1: Write and assembly program to count the number of 1's using LOOP

Write an assembly language program that counts the number of ‘1’s in a byte residing in CL register. Store the counted number in DH register.

Task 2: To identify an odd number

Write an assembly language program, that check the contents of AH register and place a 1 in BL register if the number is Odd or a 0 otherwise.

Task 3: Write an assembly language program using subroutines

Write an assembly language program to calculate the average of four 8-bit numbers that uses a subroutine named ‘ADDITION’. The numbers are stored in AL, AH, BL and BH registers and their average value should be left in CX. For adding the numbers, you are required to make use a subroutine” ADDITION”

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____ **Date:** _____

LAB # 5

To Explain and Show the Output of BIOS Interrupt Programming using EMU8086 Software Tool

Objectives

- To explain the working of BIOS interrupt programming and EXE Program Structure using assembly language program.
- To show the String and Character on output screen using BIOS Interrupt programming.

Pre-Lab Exercise

Read the details given below in order to comprehend the basic operations of BIOS programming and different routines. Study in detail and become familiar with the various ways and in which these instructions and routines can be used.

Memory Models

Machine language programs consist of code, data and stack. Each part occupies a memory segment. The same organization is reflected in an assembly language program. This time the code data and stack are structured as program segments. Each program segment is translated into a memory segment by the assembler.

The size of code and data a program can have been determined by specifying a memory model using the. MODEL directive. The syntax is

. MODEL memory_model

The most frequently used memory models are.

Model	Description
SMALL	code in one segment data in one segment
MEDIUM	code in more than one segment data in one segment
COMPACT	code in one segment data in more than one segment
LARGE	code in more than one segment data in more than one segment
HUGE	no array larger than 64K bytes code in more than one segment data in more than one segment arrays may be larger than 64K bytes

Memory model must be specified before any segment definition.

Program structure for .EXE files

The general form of a SMALL model program is given below. Unless there is a lot of code or data, the appropriate model is SMALL. When this program is assembled, assembler will generate executable file with extension .EXE.

```
.MODEL SMALL
.STACK 100h
.DATA
    ; Variables and constants are defined here
.CODE
MAIN PROC
    ; Instructions will be written here
MAIN ENDP
    ; Procedures other than MAIN will be written here
END MAIN
```

The END directive in last line of the program specifies the entry point of code. The code will start its execution from start of MAIN procedure. The name after END directive specifies the name of procedure that must be executed first.

DOS I/O service routines

CPU communicates with peripherals through I/O registers called I/O ports. There are two instructions, IN and OUT, that access the ports directly. These instructions are used when a program needs to directly communicate with peripherals like when fast I/O is needed. However most, most applications programs do not use IN and OUT because it's much easier to program I/O with the service routines provided by manufacturer.

There are two categories of I/O service routines.

- a. BIOS routines and
- b. DOS routines

BIOS routines are stored in ROM and communicate directly with the I/O ports.

The DOS routines use BIOS routines to execute direct I/O operation. They can carry out the more complex tasks; for example, printing a character string, getting input from keyboard etc.

INT instruction

To invoke DOS or BIOS routine, the INT (interrupt) instruction is used. It has the format

```
INT interrupt_number
```

Where interrupt_number is a number that specifies a routine. For example, INT 16h invokes a BIOS routine that performs keyboard input. In this Lab you will use a particular DOS routine, INT 21h. INT 21h may be used to invoke a large number of DOS functions. A particular function is requested by placing function number in the AH register and invoking INT 21h. Some of the commonly used

interrupt numbers are given below

Table 5-1: Commonly used interrupt numbers

AH value (Hex)	Function
1	Character input (with echo)
2	Character output
7	Character input (without echo)
9	String Output
A	String Input
2A	Get System Date
2C	Get System Time
4C	return control to the operating system (stop program).

Read a character input from keyboard

To get input from keyboard execute the following instructions

```
MOV AH, 1 ; input key function
INT 21h ; get ASCII code in AL
```

When a character key is pressed, AL will contain its ASCII code. If any other key is pressed, such as an arrow key or F1-F10 and so on, AL will contain 0.

Display a character on screen

To display a character 'A' on screen execute the following instructions, actually DL must contain the ASCII code of character that is to be displayed. In this example DL will contain ASCII code of 'A'.

```
MOV AH, 2 ; display character function
MOV DL, 'A' ; character is 'A'
INT 21h ; display character
```

If DL contains ASCII code of any of the control characters, INT 21h instruction causes control function to be executed. ASCII codes for principal control characters are as follows.

The above code will print 'A' on the screen and cursor will move to next position on the same line. To move cursor to new line we need to execute two control functions i.e. carriage return(CR) and line feed(LF).

Display a string on screen

To display a string execute the following instructions, DX register must contain the starting offset address of string that is to be displayed

```
LEA DX, MSG ; get offset address of MSG
MOV AH, 9 ; string display function
INT 21h ; display MSG
```

Where MSG is string defined in data segment as


```
MSG db "Hello World !", '$'
```

The last character of string must be '\$', in order to indicate the end of string.

Read a string input

Following lines of code will input a string from keyboard and save it to a buffer whose offset address must exist in register DX and first location of buffer must contain size of buffer. This function does not allow entering more characters than the specified buffer size.

```
LEA DX, buffer; get offset address of buffer
MOV AH, 0A; input string function
INT 21h ; input string
```

When above code executes, first byte of buffer will contain size of buffer as specified in buffer declaration, 2nd byte will contain actual number of characters read.

'\$' must be appended at end of buffer to print the contents of buffer as string.

Buffer of size 10 can be defined as:

```
buffer db 10,?, 10 dup(' ')
```

In-Lab Exercise

Task 1: Assembly language program

Write an assembly language program to

- Display “?”
- Read two decimal digits between 0 to 4
- Display the result as per the following format

Sample execution:

```
?24
```

The sum of 2 and 4 is 6

Hint: INT 21h always read input values as ASCII characters

Task 2: Write an assembly language program to convert the case of a string

Write a complete assembly program that converts a user-input lower-case string into corresponding upper-case string.

Sample execution:

Input: hello world

Output: HELLO WORLD

Table 5-2: ASCII Table

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Source: www.LookupTables.com

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____

Date: _____

LAB # 6

To Explain the Procedure for Using STM32F407, Keil™, MDK-ARM and STM32F407 Trainer Board

Objectives

- To identify the procedures related to configuration of I/O ports using
- To explain the use of STM32F407 trainer board used in lab Tasks.

Pre-Lab Exercise

Read this experiment in its entirety to become familiar with the STM32F4 microcontroller and STM32F407 Discovery board. You should install another software that will use in this course: the Microcontroller Development Kit (MDK-ARM), which supports software development for and debugging of ARM7, ARM9, Cortex-M, and Cortex-R4 processor-based devices. MDK combines the ARM RealView compilation tools with the Keil μVision Integrated Development Environment (IDE). You should also review how to compile/simulate designs within it.

Introduction

The STM32F407 Evaluation board helps you to discover the STM32F407 Cortex M4 Core high-performance features and to develop your applications. It is based on an STM32F407VGT6 and includes an ST-LINK/V2 embedded debug tool interface, ST MEMS digital accelerometer, ST MEMS digital microphone, audio DAC with integrated class D speaker driver, LEDs, push buttons, LCD, 4x3 keypad, Temperature Sensor, LDR sensor, ADC variable and a USB OTG micro-AB connector. The STM32F4 Trainer is a low-cost and easy-to-use development kit to quickly evaluate and start a development with an STM32F4 high-performance microcontroller.

Getting started with STM32F4

Follow the sequence below to configure the STM32F407 training board and launch the DISCOVER application:

- Check jumper position on the board, JP1 on, CN3 on (DISCOVERY selected)
- Connect the STM32F407 DISCOVERY board to a PC with a USB cable ‘type A to mini-B’ through USB connector CN1 to power the board. Red LED LD2 (PWR) then lights up
- Connect Power Adapter to DC input Connector J1
- To study or modify the DISCOVER project related, visit www.st.com/stm32f4-discovery and follow the tutorial
- Discover the STM32F4 features, download and execute programs proposed in the list of projects
- Develop your own application using available examples

Development toolchain supporting the STM32F4

- Altium, TASKING™ VX-Toolset
- Atollic TrueSTUDIO®
- IAR Embedded Workbench® for ARM (EWARM)

- Keil™, MDK-ARM

We are using Keil™, MDK-ARM for the rest of the labs.

Features

The STM32F4 DISCOVERY offers the following features:

- STM32F407VGT6 microcontroller featuring 1 MB of Flash memory, 192 KB of RAM in an LQFP100 package
- On-board ST-LINK/V2 with selection mode switch to use the kit as a standalone ST-LINK/V2 (with SWD connector for programming and debugging)
- Board power supply: through USB bus or from an external 5V supply voltage
- External application power supply: 3V and 5V
- LIS302DL or LIS3DSH, ST MEMS motion sensor, 3-axis digital output accelerometer
- MP45DT02, ST MEMS audio sensor, omnidirectional digital microphone
- CS43L22, audio DAC with integrated class D speaker driver
- Eight LEDs: – LD1 (red/green) for USB communication – LD2 (red) for 3.3V power on – Four user LEDs, LD3 (orange), LD4 (green), LD5 (red) and LD6 (blue) – 2 USB OTG LEDs LD7 (green) VBus and LD8 (red) over-current
- Eight LEDs: – LED1 to LED8(red)
- 16 x 2 Character LCD with Backlight
- 3 Digit 7 Segment led Display
- 4 x 3 Numeric Keypad
- Two push buttons PB1 and PB2
- Two pushbuttons (user and reset)
- Buzzer interface
- Temperature Sensor STLW75 analog interface
- LDR Analog interface
- Multi turn potentiometer for ADC interface
- IR receiver for IR interface
- USB OTG with micro-AB connector
- Extension header for LQFP100 I/Os for quick connection to prototyping board and easy probing

STM32F407VGT6 Microcontroller

This ARM Cortex-M4 32-bit MCU with FPU has 210 DMIPS, up to 1 MB Flash/192+4 KB RAM, USB OTG HS/FS, Ethernet, 17 TIMs, 3 ADCs, 15 comm. interfaces and a camera.

This device provides the following benefits.

- 168 MHz/210 DMIPS Cortex-M4 with single cycle DSP MAC and floating point unit providing: Boosted execution of control algorithms, more features possible for your applications, ease of use better code efficiency, faster time to market elimination of scaling and saturation easier support for meta-language tools
- Designed for high performance and ultra-fast data transfers; ART Accelerator, 32-bit, 7- layer AHB bus matrix with 7 masters and 8 slaves including 2 blocks of SRAM, Multi DMA controllers: 2 general purpose, 1 for USB HS, 1 for Ethernet, 1 SRAM block dedicated to the

core, providing performance equivalent to 0-wait execution from Flash Concurrent execution and data transfers and simplified resource allocation

- Outstanding power efficiency; Ultra-low dynamic power, RTC <1 μ A typical in VBAT mode, 3.6 V down to 1.7 V VDD, Voltage regulator with power scaling capability, providing extra flexibility to reduce power consumption for applications requiring both high processing and low power performance when running at low voltage or on a rechargeable battery
- Superior and innovative peripherals providing new possibilities to connect and communicate high speed data and more precision due to high resolution
- Extensive tools and software solutions providing a wide choice within the STM32 ecosystem to develop your applications

Using ST-LINK/V2 to program/debug the STM32F4 on board

To program the STM32F4 on board, simply plug in the two jumpers on CN3, as shown in *Figure 6-1*, but do not use the CN2 connector as that could disturb communication with the STM32F407VGT6 of the STM32F4 DISCOVERY.

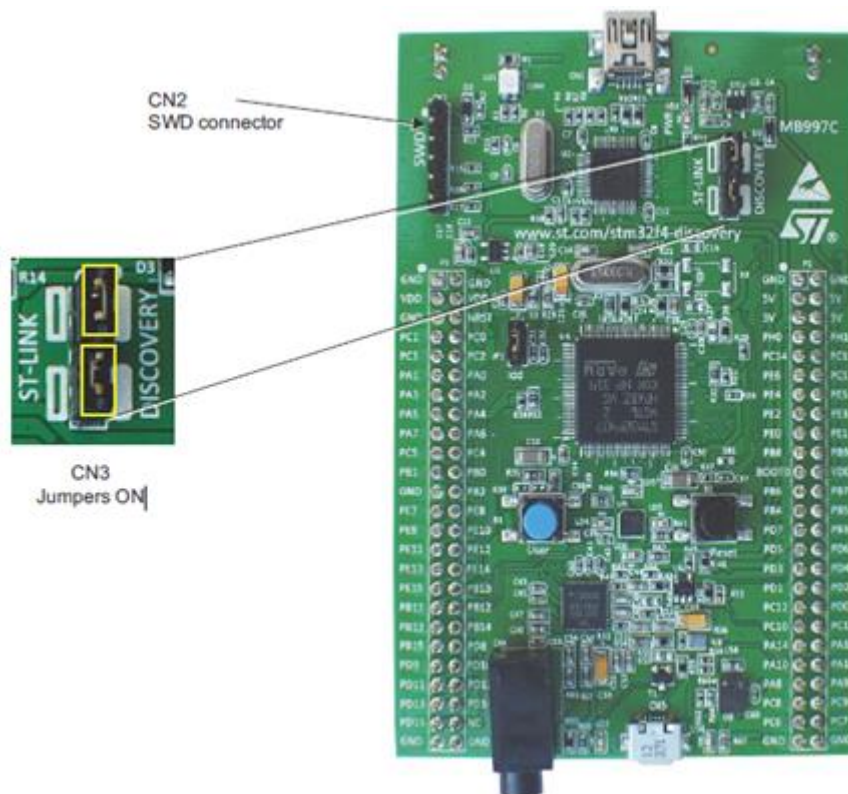


Figure 6-1: STM32F4 Discovery Board

Power supply and power selection

The power supply is provided either by the host PC through the USB cable to Debugger and Microcontroller, and by an external 12V power supply adapter. The D1 diodes for protection.

Using MDK-ARM Microcontroller Development Kit by Keil™

Building an existing MDK-ARM project

Follow these steps to build an existing MDK-ARM project.

- Open the MDK-ARM μ Vision4 IDE, debugger, and simulation environment. *Figure 6-2:* shows the basic names of the windows referred to in this section.

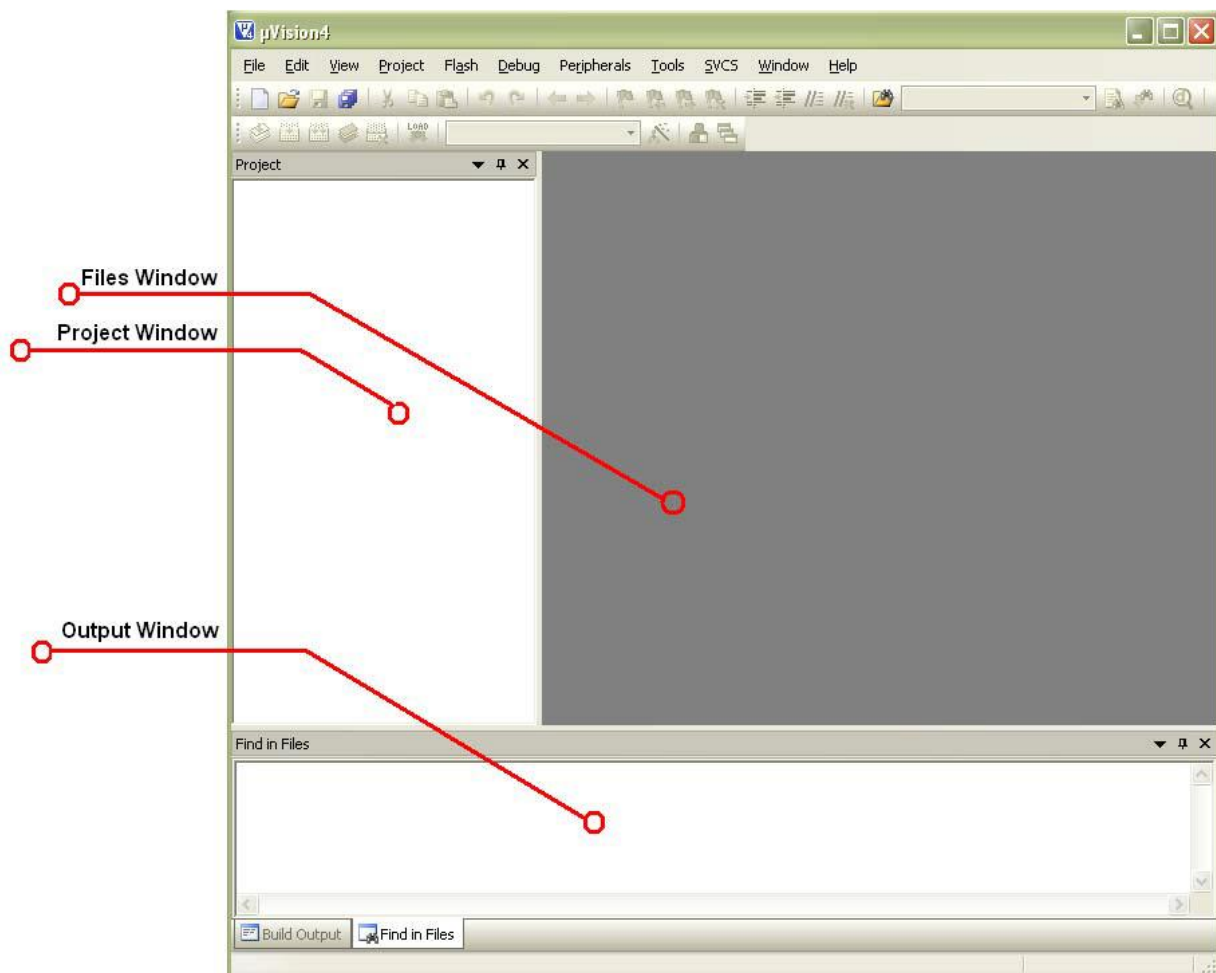
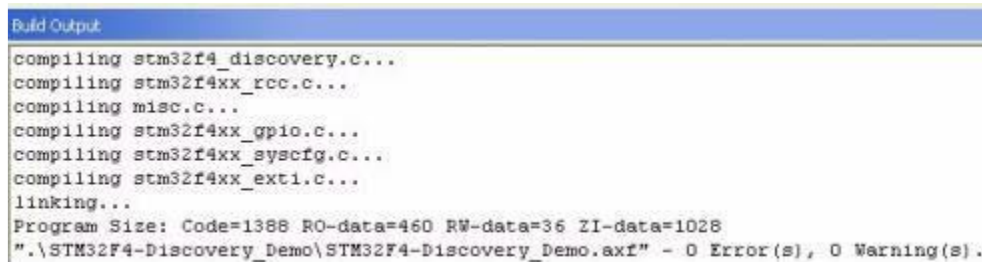


Figure 6-2: MDK-ARM μ Vision4 IDE environment

- In the **Project** menu, select **Open Project...** to display the Select Project File dialog box. Browse to select the *STM32F4-Discovery.uvproj* project file and click **Open** to launch it in the Project window.
- In the **Project** menu, select **Rebuild all target files** to compile your project.
- If your project is successfully compiled, the following **Build Output** window *Figure 6-3:* is displayed.



```

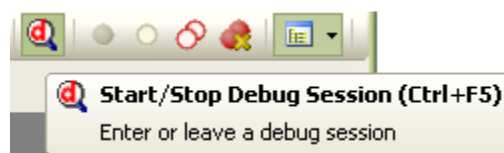
Build Output
compiling stm32f4_discovery.c...
compiling stm32f4xx_rcc.c...
compiling misc.c...
compiling stm32f4xx_gpio.c...
compiling stm32f4xx_syscfg.c...
compiling stm32f4xx_exti.c...
linking...
Program Size: Code=1388 RO-data=460 RW-data=36 ZI-data=1028
".\STM32F4-Discovery_Demo\STM32F4-Discovery_Demo.axf" - 0 Error(s), 0 Warning(s).

```

Figure 6-3: Build Output

Debugging and running your MDK-ARM project

In the MDK-ARM μ Vision4 IDE, click the magnifying glass to program the Flash memory and begin debugging as shown below in Figure 6-4.

Figure 6-4: Starting a MDK-ARM μ Vision4 debugging session

The debugger in the MDK-ARM IDE can be used to debug source code at C and assembly levels, set breakpoints, monitor individual variables and watch events during the code execution as shown below in Figure 6-5.

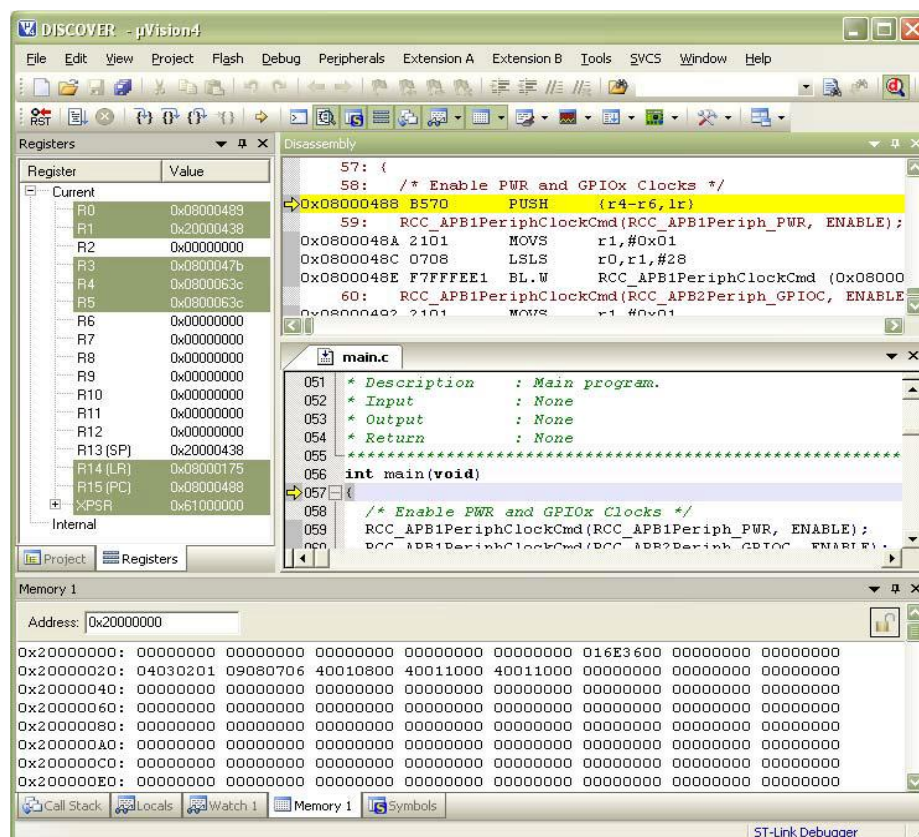


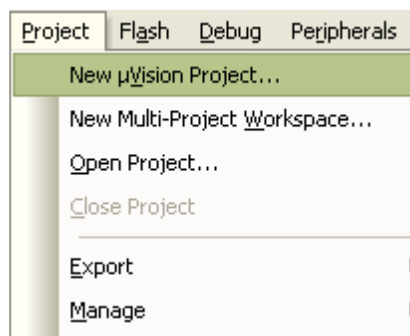
Figure 6-5: MDK-ARM IDE workspace

Creating your first application using the MDK-ARM toolchain

Managing source files

Follow these steps to manage source files.

- In the **Project** menu, select **New μ Vision Project...** to display the Create Project File dialog box. Name the new project and click **Save**.



- When a new project is saved, the IDE displays the *Figure 6-6*. Select the device used for testing. In this example, we will use the STMicroelectronics device mounted on the STM32F4DISCOVERY board. In this case, double-click on **STMicroelectronics**, select the **STM32F407VGT6** device and click **OK** to save your settings.

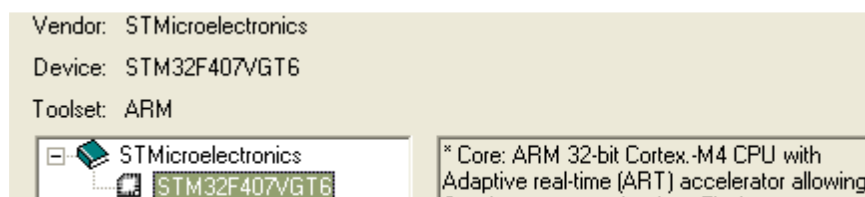


Figure 6-6: Device selection dialog box

- Click **Yes** to copy the STM32 Startup Code to the project folder and add the file to the project as shown in *Figure 6-7*.

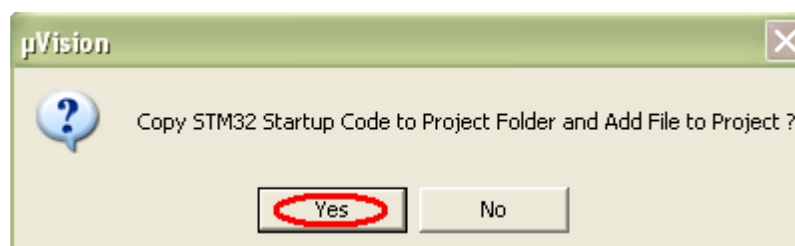


Figure 6-7: Copy the STM32 Startup Code dialog box

To create a new source file, in the **File** menu, select **New** to open an empty editor window where you can enter your source code.

The MDK-ARM tool chain enables C color syntax highlighting when you save your file using the dialog **File > Save As...** under a filename with the *.c extension. In this example the file is saved as **main.c**.

main.c example file

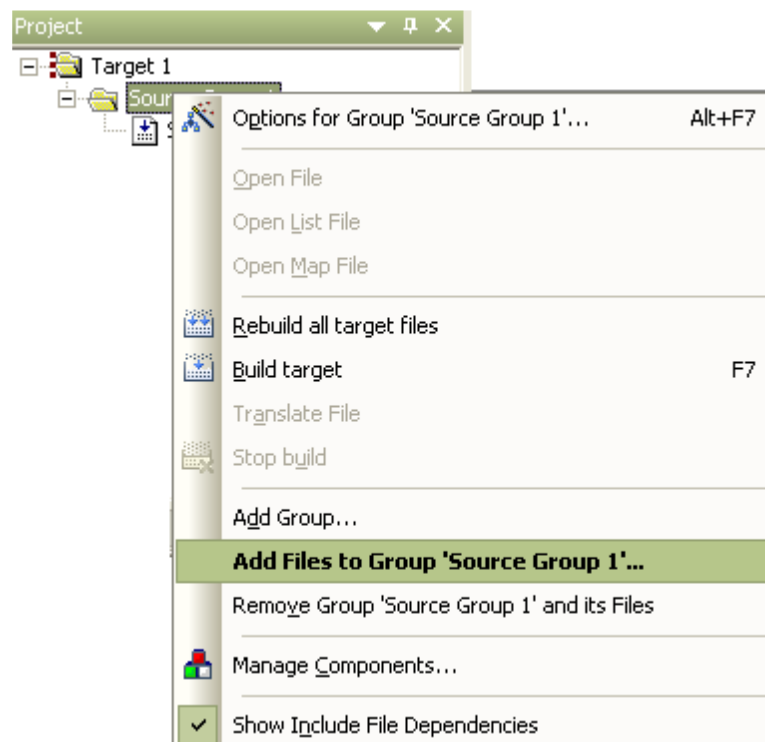
```

1
2
3 int main (void)
4 {
5     return (0) ;
6 }

```

MDK-ARM offers several ways to add source files to a project. For example, you can select the file group in the **Project Window > Files** page and right-click to open a contextual menu. Select the **Add Files...** option and browse to select the *main.c* file previously created.

Adding source files



If the file is added successfully, the following window is displayed.

New project file tree structure



Configuring project options

- In the **Project** menu, select **Options for Target 1** to display the Target Options dialog box.
- Open the **Target** tab and enter IROM1 and IARM1 start and size settings as shown in *Figure 6-8*.
- Open the **Debug** tab, click **Use** and select the **ST-Link Debugger**. Then, click **Settings** and select the **SWD** protocol. Click **OK** to save the ST-Link setup settings.
- Select **Run to main()**.
- Open the **Utilities** tab, select **Use Target Driver for Flash Programming** and select the **ST-Link Debugger** from the drop-down menu.
- Verify that the **Update Target before Debugging** option is selected.
- Click **OK** to save your settings.

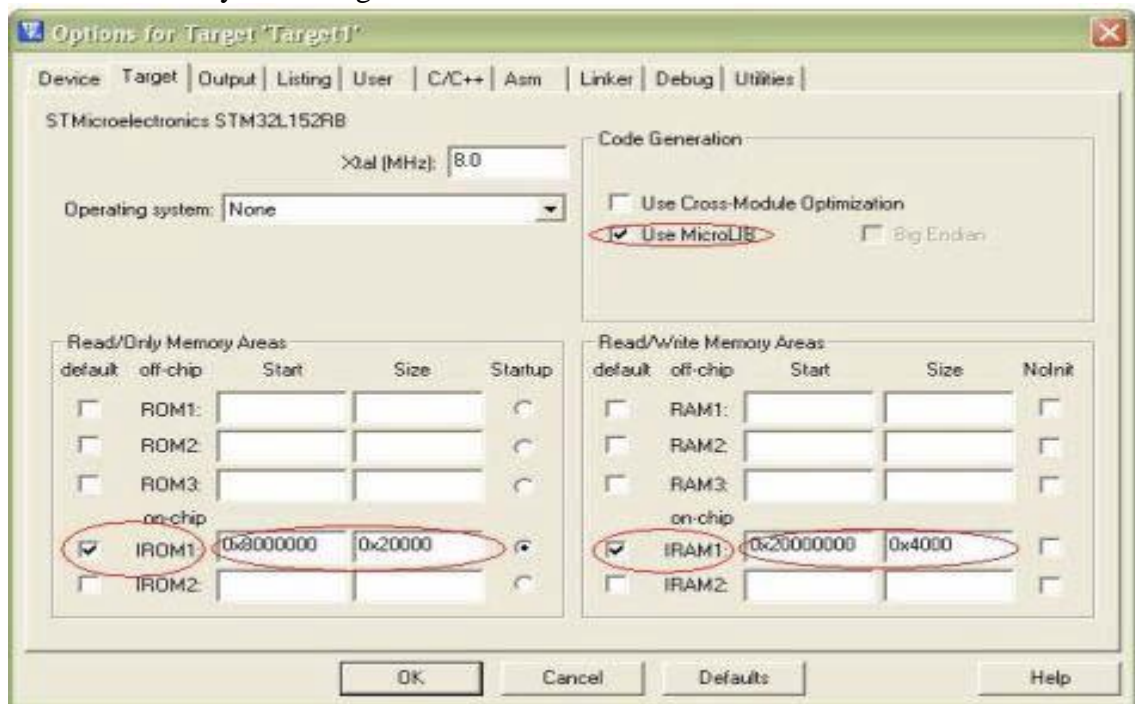


Figure 6-8: Target Options dialog box - Target tab

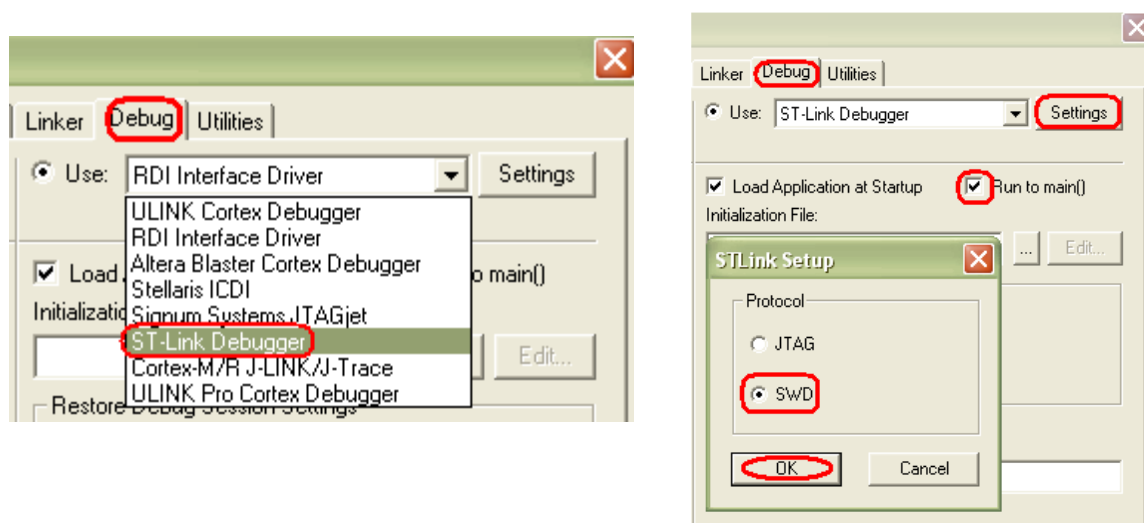


Figure 6-9: ST-Link Setup

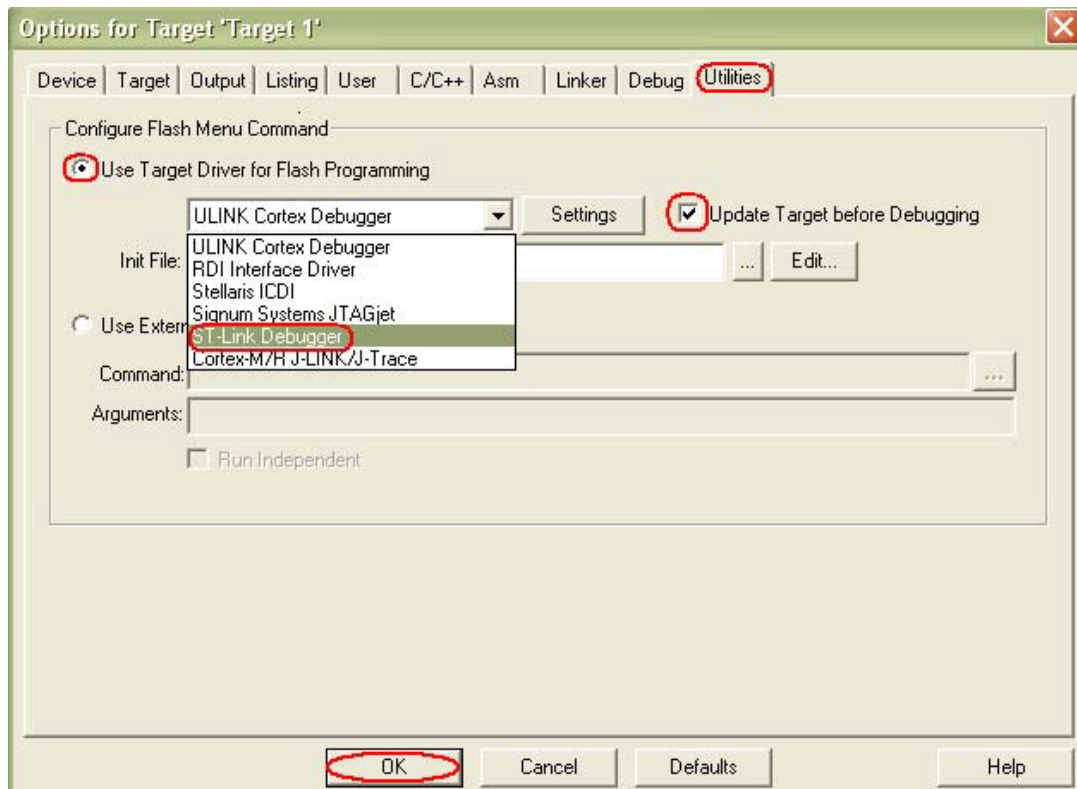


Figure 6-10: Target Options dialog box - Utilities tab

- In the **Project** menu, select **Build Target**.
- If your project is successfully built, the following window is displayed.

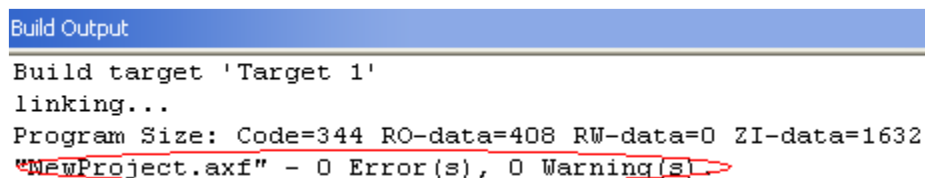
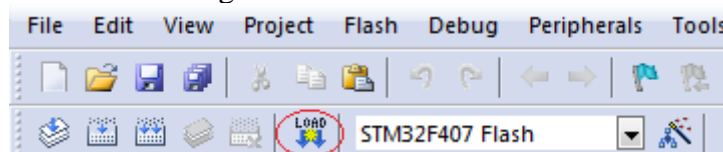


Figure 6-11: MDK-ARM μ Vision4 project successfully built

Download and Burn Hex File in MCU:

- Select **Load** button circled in figure.



- In **Flash Menu** and select option of **Download** as shown in Figure 6-12.

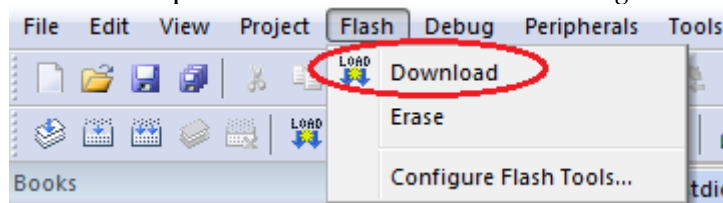


Figure 6-12: Download and Burn HEX file

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____ **Date:** _____

LAB # 7

To Explain and Reproduce the Working of Switches and LED's on STM32F407 Trainer Board using C Programming.

Objectives

- To explain the working of switch and led interfacing with the STM32F407 using trainer board.
- To modify the program and reproduce its output on STM32F407 trainer board using C programming.

Pre-Lab Exercise

Read the details given below in order to comprehend the basic operation of an LED and push button. Interfacing of LED with microcontroller. It can be interfaced to perform different tasks in different combinations.

Introduction

Light Emitting Diodes are the mostly commonly used components in many applications. They are made of semiconducting material. A **light-emitting diode (LED)** is a two-lead semiconductor light source. It is a pn-junction diode, which emits light when activated. When a suitable voltage is applied to the leads, electrons are able to recombine with electron holes within the device, releasing energy in the form of photons. This effect is called electroluminescence, and the color of the light (corresponding to the energy of the photon) is determined by the energy band gap of the semiconductor.

1.9 to 2.1 V for red, orange and yellow,

3.0 to 3.4 V for green and blue,

2.9 to 4.2 V for violet, pink, purple and white.

We interface LED with microcontroller in two configurations (Source and Sink) as shown in the *Figure 7-1*.

The push-button, switch is assumed to be bounce free, implying that when it is pressed and then released, a single pulse is produced. Push Button can interface in two types of configurations i.e. Pull up and Pull Down as shown in the *Figure 7-2*. We use the terms pull-up or pull-down resistor, this is a simple resistor usually between 10k and 100k, to define the input state when no signal source is connected, this way overriding the noise signal, common sense dictates that when you have potentially larger noise then a smaller resistor is needed, but don't be careless about it, don't place a 100Ω resistor because your signal source must be able to "defeat" the pull-up (down) resistor. A rule of thumb is to use at least 10x larger pull-up (down) resistor then your signal source impedance.

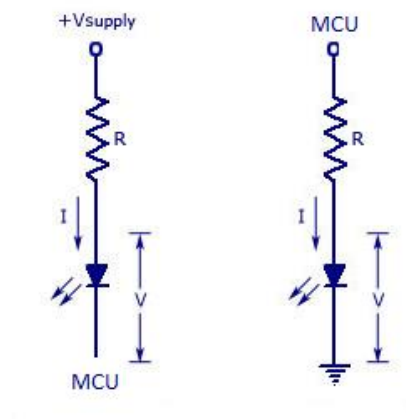


Figure 7-1: LED interfacing with microcontroller

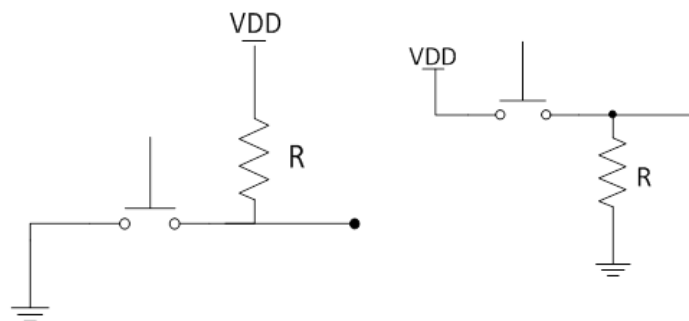


Figure 7-2: Push Button Configurations

Programmer's Model of STM

Peripheral Clock Enable Register (RCC_AHB1ENR)

The details of RCC_AHB1ENR register is as follows:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved	OTGHS ULPIEN	OTGHS EN	ETHMA CPTPE N	ETHMA CRXEN	ETHMA CTXEN	ETHMA CEN	Reserved			DMA2EN	DMA1EN	CCMDATA RAMEN	Res.	BKPSR AMEN	Reserved
	rw	rw	rw	rw	rw	rw				rw	rw			rw	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CRCEN	Reserved			GPIOIE N	GPIOH EN	GPIOGE N	GPIOFE N	GPIOEEN	GIOD EN	GPIOC EN	GPIOB EN	GPIOA EN
			rw				rw	rw	rw	rw	rw	rw	rw	rw	rw

GPIOx_MODER

The GPIOx_MODER register is used to select the I/O direction (input, output, AF, analog) and the details are as follows:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW

Bits 2y:2y+1 **MODERy[1:0]**: Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O direction mode.

00: Input (reset state)

01: General purpose output mode

10: Alternate function mode

11: Analog mode

GPIOx_OTYPER

GPIOx_OTYPER register IS used to select the output type (pushpull or open-drain) and the details are as follows:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **OTy[1:0]**: Port x configuration bits (y = 0..15)

These bits are written by software to configure the output type of the I/O port.

0: Output push-pull (reset state)

1: Output open-drain

GPIOx_OSPEEDR

GPIOx_OSPEEDR register is used to select the speed (the I/O speed pins are directly connected to the corresponding GPIOx_OSPEEDR register bits whatever the I/O direction) and its details are as follows:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
OSPEEDR15[1:0]		OSPEEDR14[1:0]		OSPEEDR13[1:0]		OSPEEDR12[1:0]		OSPEEDR11[1:0]		OSPEEDR10[1:0]		OSPEEDR9[1:0]		OSPEEDR8[1:0]	
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OSPEEDR7[1:0]		OSPEEDR6[1:0]		OSPEEDR5[1:0]		OSPEEDR4[1:0]		OSPEEDR3[1:0]		OSPEEDR2[1:0]		OSPEEDR1[1:0]		OSPEEDR0[1:0]	
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW

Bits 2y:2y+1 **OSPEEDRy[1:0]**: Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O output speed.

00: 2 MHz Low speed

01: 25 MHz Medium speed

10: 50 MHz Fast speed

11: 100 MHz High speed on 30 pF (80 MHz Output max speed on 15 pF)

GPIOx_PUPDR

The GPIOx_PUPDR register is used to select the pull-up/pull-down whatever the I/O direction and its details are as follows

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[1:0]	
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW

Bits 2y:2y+1 PUPDRy[1:0]: Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O pull-up or pull-down

00: No pull-up, pull-down

01: Pull-up

10: Pull-down

11: Reserved

I/O port data registers

Each GPIO has two 16-bit memory-mapped data registers:

GPIOx_IDR:

The data input through the I/O are stored into the input data register (GPIOx_IDR), a read-only register and the details are as follows

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 IDRy[15:0]: Port input data (y = 0..15)

These bits are read-only and can be accessed in word mode only. They contain the input value of the corresponding I/O port.

GPIOx_ODR.

It stores the data to be output, it is read/write accessible and the details are as follows:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 ODRy[15:0]: Port output data (y = 0..15)

These bits can be read and written by software.

Note: For atomic bit set/reset, the ODR bits can be individually set and reset by writing to the GPIOx_BSRR register (x = A..I).

GPIOx_BSRR

The bit set reset register (GPIOx_BSRR) is a 32-bit register which allows the application to set and reset each individual bit in the output data register (GPIOx_ODR).

The bit set reset register has twice the size of GPIOx_ODR.

The details of BSRR register are as follows

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

- Bits 31:16 BRy: Port x reset bit y (y = 0..15)
 - These bits are write-only and can be accessed in word, half-word or byte mode. A read to these bits returns the value 0x0000.
 - 0: No action on the corresponding ODRx bit
 - 1: Resets the corresponding ODRx bit
 - Note: If both BSx and BRx are set, BSx has priority.
- Bits 15:0 BSy: Port x set bit y (y= 0..15)
 - These bits are write-only and can be accessed in word, half-word or byte mode. A read to these bits returns the value 0x0000.
 - 0: No action on the corresponding ODRx bit
 - 1: Sets the corresponding ODRx bit

Program flow for STM projects

- 1) SystemCoreClockUpdate(); // Get Core Clock Frequency
- 2) Setup RCC_AHB1ENR
- 3) Setup the appropriate GPIO port
 - a. Set the appropriate value of GPIOx_MODER
 - b. Set the appropriate value of GPIOx_OTYPER
 - c. Set the appropriate value of GPIOx_OSPEEDR
 - d. Set the appropriate value of GPIOx_PUPDR
- 4) Program logic goes here

STM Code for LED Interfacing

With the help of this program, we will know how to interface LED with the microcontroller. The STM32F4 training board has an LED driven by I/O pin PB11. In order to configure this pin, clocks must first be enabled for General Purpose Input Output (GPIO) Port B with the following command.

```
RCC->AHB1ENR |= (1 << 1)
```

After enabling the clocks, it is necessary to configure any required pins. In this case, a single pin (PB11) must be configured as an output.

```
GPIOB->MODER = 0x55444444;
GPIOB->OTYPER = 0x00000000;
GPIOB->OSPEEDR = 0xAAAAAAAA;
```

```
GPIOB->PUPDR      =      0x00000000;
```

Once configuration is complete, the pin can be set with the following code:

```
GPIOB->BSRRL = (1 << 11);
```

The pin can be reset with the following code:

```
GPIOB->BSRRH = (1 << 11)
```

In-Lab Exercise

Task 1: LED Blinking

Complete the C program that blinks the LED repeatedly with some delay, whereas the LED is attached at I/O pin PB11 of STM32F4 training board.

Program

```
#include <stdio.h>
#include <stm32f4xx.h>

volatile uint32_t msTicks;                                /* counts 1ms timeTicks */
void SysTick_Handler(void)
{
    msTicks++;
}
/*-----
delays number of tick Systicks (happens every 1 ms)
*-----*/
void Delay (uint32_t dlyTicks)
{
    uint32_t loop=0,dly=0,loope=0;
    dly = dlyTicks ;
    for(loop=0;loop<dly;loop++)
    {
        for(loope=0;loope<29000;loope++)
        {
            __nop();
        }
    }
}
/*-----
MAIN function
*-----*/
int main (void)
{
    SystemCoreClockUpdate();                               // Get Core Clock Frequency
    RCC->AHB1ENR |= (1 << 1) ;                               // Enable GPIOB clock
    GPIOB->MODER = 0x55444444; // Setting Direction of Port B
    GPIOB->OTYPER = 0x00000000; // To configure the output type of Port B
    GPIOB->OSPEEDR = 0xAAAAAAA; // To configure the speed of Port B
    GPIOB->PUPDR = 0x00000000; // To configure the Port B Pull-up or Pull-down
    while(1) {                                              // Infinite Loop
        add your code here
    }
}
```

Task 2: LED activation with push button

Complete the C program that continuously monitor push button. When button is activated, a LED will turn on otherwise remain off. whereas the LED and switch is attached at I/O pin PB11 and PC6 respectively of STM32F4 training board.

Program

```
#include <stdio.h>
#include <stm32f4xx.h>
volatile uint32_t msTicks; /* counts 1ms timeTicks */
volatile uint32_t bt=0;
void SysTick_Handler(void)
{
    msTicks++;
}
void Delay (uint32_t dlyTicks)
{
    uint32_t loop=0,dly=0,loope=0;
    dly = dlyTicks ;
    for(loop=0;loop<dly;loop++)
    {
        for(loope=0;loope<29000;loope++)
        {
            __nop();
        }
    }
}
/*-----
MAIN function
-----*/
int main (void)
{
    SystemCoreClockUpdate(); // Get Core Clock Frequency
    RCC->AHB1ENR |= (1 << 1) ; // Enable GPIOB clock
    RCC->AHB1ENR |= (1 << 2) ; // Enable GPIOC clock
    GPIOB->MODER = 0x55444444; // Setting Direction of Port B
    GPIOB->OTYPER = 0x00000000; // To configure the output type of Port B
    GPIOB->OSPEEDR = 0xAAAAAAAA; //To configure the speed of Port B
    GPIOB->PUPDR = 0x00000000 // To configure the Port B Pull-up or Pull-down
    GPIOC->MODER = 0x00000000; // Setting Direction of Port C
    GPIOC->OTYPER = 0x00000000; // To configure the output type of Port C
    GPIOC->OSPEEDR = 0xAAAAAAAA; // To configure the speed of Port C
    GPIOC->PUPDR = 0x00000000; // To configure the Port C Pull-up or Pull-down
    while(1) { // Infinite Loop
        /*-----
Add your code here
-----*/
    }
}
```

Task 3

Write a program that will continuously monitor two push buttons and generates the output as per the following conditions:

- Initially all the four LEDs are OFF.
- When push button 1 is pressed, the number of ON LEDs will increase by 1
- When push button 2 is pressed, the number of ON LEDs will decrease by 1

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____ **Date:** _____

LAB # 8

To Explain and Reproduce the Working of 7- Segment on STM32F407 Trainer Board using C Programming.

Objectives

- To explain the working of 7-segment interfacing with the STM32F407 using trainer board.
- To modify the program and reproduce its output on STM32F407 trainer board using C programming.

Pre-Lab Exercise

Read the details given below in order to comprehend the basic operation of seven-segment display, its types and interfacing of seven-segment with microcontroller. It can be interfaced to do different tasks in different combinations.

Introduction

Seven-segment displays are now widely used in almost all microprocessor-based devices. A seven-segment display can display the digits from 0 to 9 and the hex digits A to F. Display is composed of seven LEDs that are arranged in a way to allow the display of different digits using different combinations of LEDs.

There are two types of seven-segment displays, common anode and common cathode as shown in *Figure 8-1*. In common anode, all the anodes of the LEDs are connected together in a common point. By connecting this point to the supply, we can activate the display. The data bits are connected to the cathodes of the LEDs. So, if one of the bits is low; current will flow through the corresponding LED and turn it ON. For the common cathode display, the common cathode is connected to the GND and the data bits are connected to the anodes of the LEDs.

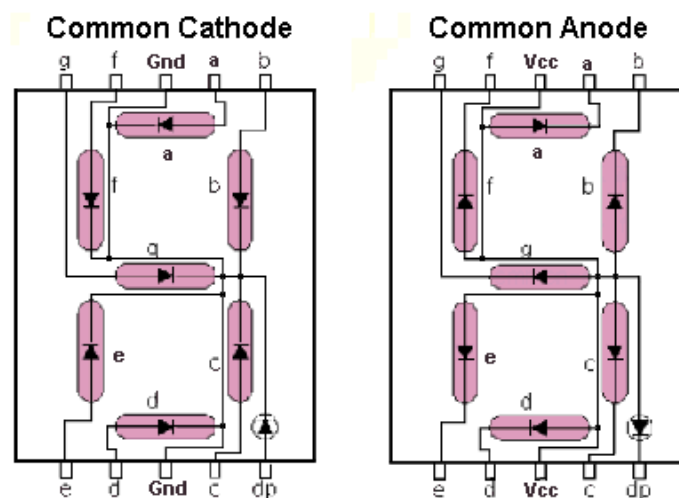


Figure 8-1: Types of 7-Segment displays

Depending upon the decimal digit to be displayed, the particular set of LEDs is forward biased. For instance, to display the numerical digit 0, we will need to light up six of the LED segments corresponding to a, b, c, d, e and f. Then the various digits from 0 through 9 can be displayed using a 7-segment display as shown in *Figure 8-2*.

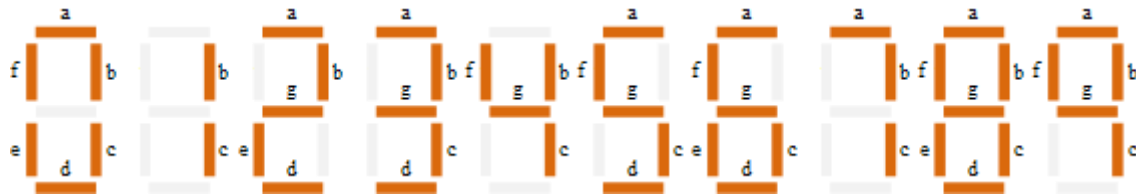


Figure 8-2: 7-Segment Display Segments for all Numbers

Multiplexing

Single seven segment display can display only one digit but what if we want to display more than one digit. There is a solution named Multiplexing. Consider a panel with 3 displays and number to be displayed is 123. Without using multiplexing we need at least $3 \times 7 = 21$ bits for the 3 displays. Multiplexing means that only one display will have current flowing through it at one time. By activating one display at a time, we can display 3 digits turn by turn on their corresponding 7 segment display. If the multiplexing frequency is high enough, our eyes will not be able to detect the switching in the displays and they will seem to be active at the same time. Using this technique same data bits will be shared by all displays.

In-Lab Exercise

Task 1: 7 segment interfacing

Complete the program that will continuously monitor a push button. When a push button is activated; number will be incremented by one and this incremented value will be displayed on 7-Segment display.

Hint: The STM32F4 training board has buttons driven by I/O pin PC6 and PC7 and 7-Segment by I/O Port B, C, D and E so you have to configure these ports for the interfacing.

Program

```
#include <stdio.h>
#include <stm32f4xx.h>

#define seg_a_off    GPIOE->BSRRL = ((1 << 2))
#define seg_b_off    GPIOE->BSRRL = ((1 << 5))
#define seg_c_off    GPIOC->BSRRL = ((1 << 8))
#define seg_d_off    GPIOE->BSRRL = ((1 << 6))
#define seg_e_off    GPIOD->BSRRL = ((1 << 0))
#define seg_f_off    GPIOC->BSRRL = ((1 << 9))
#define seg_g_off    GPIOE->BSRRL = ((1 << 4))
#define seg_dot_off   GPIOC->BSRRL = ((1 << 13))

#define seg_a_on      GPIOE->BSRRH = ((1 << 2))
#define seg_b_on      GPIOE->BSRRH = ((1 << 5))
#define seg_c_on      GPIOC->BSRRH = ((1 << 8))
#define seg_d_on      GPIOE->BSRRH = ((1 << 6))
#define seg_e_on      GPIOD->BSRRH = ((1 << 0))
#define seg_f_on      GPIOC->BSRRH = ((1 << 9))
#define seg_g_on      GPIOE->BSRRH = ((1 << 4))
```

```

#define seg_dot_on  GPIOC->BSRRH = ((1 << 13))

#define dig_1_on    2GPIOB->BSRRH = ((1 << 5))
#define dig_2_on    GPIOB->BSRRH = ((1 << 4))
#define dig_3_on    GPIOB->BSRRH = ((1 << 7))

#define dig_1_off   GPIOB->BSRRL = ((1 << 5))
#define dig_2_off   GPIOB->BSRRL = ((1 << 4))
#define dig_3_off   GPIOB->BSRRL = ((1 << 7))
unsigned int counter=0 , bt=0;
void Delay (uint32_t dlyTicks);
void SysTick_Handler(void);
void display_value(unsigned int value);
void display_digit(unsigned int sel_digit,unsigned int display_digit);
void display_digit(unsigned int sel_digit,unsigned int display_digit)
{
switch(sel_digit){
    case 1:dig_1_on;dig_2_off;dig_3_off;break;
    case 2:dig_1_off;dig_2_on;dig_3_off;break;
    case 3:dig_1_off;dig_2_off;dig_3_on;break;
} // End switch(sel_digit)
switch(display_digit){
    case 0:
        seg_a_on;
        seg_b_on;
        seg_c_on;
        seg_d_on;
        seg_e_on;
        seg_f_on;
        seg_g_off;
        seg_dot_off;
        break;
    case 1:
        .
        .
        .

```

Add your code to complete the cases

```

    } // End switch(display_digit)
} // End void display_digit(unsigned int sel_digit,unsigned int display_digit)

void display_value(unsigned int value){
    unsigned int seg1=0,seg2=0,seg3=0,seg=0;
    seg3=value%10;
    seg2=(value%100)/10;
    seg1= value/100;
    display_digit(1,seg1);
    Delay(3);
    display_digit(2,seg2);
    Delay(3);
    display_digit(3,seg3);
    Delay(3);
} // End void display_value(unsigned int value)
volatile uint32_t msTicks; /* counts 1ms timeTicks */
/*-----
SysTick_Handler
*-----*/
void SysTick_Handler(void) {
    msTicks++;
} // End void SysTick_Handler(void)
void Delay (uint32_t dlyTicks) {
    uint32_t loop=0,dly=0,loope=0;
    dly = dlyTicks ;
    for(loop=0;loop<dly;loop++){
        for(loope=0;loope<29000;loope++){
            _nop();
        } // End for(loope=0;loope<29000;loope++)
    }
}

```



```

} // End for(loop=0;loop<dly;loop++)
} // End void Delay (uint32_t dlyTicks)
/*-----
  MAIN function
  *-----*/
int main (void) {
  SystemCoreClockUpdate();           // Get Core Clock Frequency
  RCC->AHB1ENR |= (1 << 0);           // Enable GPIOA clock
  RCC->AHB1ENR |= (1 << 1);           // Enable GPIOB clock
  RCC->AHB1ENR |= (1 << 2);           // Enable GPIOC clock
  RCC->AHB1ENR |= (1 << 3);           // Enable GPIOD clock
  RCC->AHB1ENR |= (1 << 4);           // Enable GPIOE clock

Setup Port B,C,D,E according to the design
  GPIOB->MODER = 0x-----;           // Setting Direction of Port B
  GPIOB->OTYPER = 0x-----;           // To configure the output type of Port B
  GPIOB->OSPEEDR = 0x-----;           // To configure the speed of Port B
  GPIOB->PUPDR = 0x-----;           // To configure the Port B Pull-up or Pull-down

  GPIOC->MODER = 0x-----;           // Setting Direction of Port C
  GPIOC->OTYPER = 0x-----;           // To configure the output type of Port C
  GPIOC->OSPEEDR = 0x-----;           // To configure the speed of Port C
  GPIOC->PUPDR = 0x-----;           // To configure the Port C Pull-up or Pull-down

  GPIOD->MODER = 0x-----;           // Setting Direction of Port D
  GPIOD->OTYPER = 0x-----;           // To configure the output type of Port D
  GPIOD->OSPEEDR = 0x-----;           // To configure the speed of Port D
  GPIOD->PUPDR = 0x-----;           // To configure the Port D Pull-up or Pull-down

  GPIOE->MODER = 0x-----;           // Setting Direction of Port E
  GPIOE->OTYPER = 0x-----;           // To configure the output type of Port E
  GPIOE->OSPEEDR = 0x-----;           // To configure the speed of Port E
  GPIOE->PUPDR = 0x-----;           // To configure the Port E Pull-up or Pull-down
  while(1) {                          // Infinite Loop
  /*-----
    Add your code here
    *-----*/
  } //End while(1)
  return 0;
} // End int main(void)

```

Task 2: Write a C program to interface Up/Down counter

In this exercise we will interface 7-Segment display and two push buttons PB1 and PB2 to STM32F4 microcontroller. 7-Segment display will be used to display a 3 digit number. When push button PB1 is activated; number will be incremented by one and this incremented value will be displayed on 7-Segment display. When push button PB2 is activated; number will be start decrementing by one and this decremented value will be displayed on 7-Segment display. When both the push buttons are pressed simultaneously then the counter will reset to ZERO.

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____ **Date:** _____

LAB # 9

To Explain and Reproduce the Working of 16x2 LCD on STM32F407 Trainer Board using C Programming.

Objectives

- To explain the working of 16x2 LCD interfacing with the STM32F407 using trainer board.
- To modify the program and reproduce its output on STM32F407 trainer board using C programming.

Pre-Lab Exercise

Read the details given below in order to comprehend the basic operation of LCD and interfacing of LCD with microcontroller. It can be interfaced to perform different tasks in different combinations.

Introduction

In this section, we will learn about the operation modes of LCD, and then describes how to program and interface an LCD to a STM32F407.

LCD is used in all the electronics projects to display the status of the process. A 16x2 alphanumeric LCD shown in *Figure 9-1* is most widely used module of LCD now a days. There are several others type of LCD available in market also.

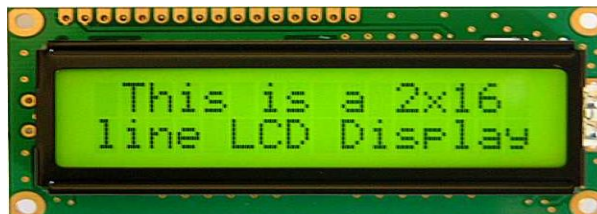


Figure 9-1: 16 x 2 LCD

LCD Operation

In recent years LCD has been finding widespread use replacing LEDs. This is due to following reasons:

- The declining prices of LCDs
- The ability to display numbers, characters, and graphics. This is in contrast to LEDs, which are limited to numbers and a few characters
- Ease of programming for characters and graphics

16x2 LCD has 2 horizontal line which comprising a space of 16 displaying character. It has two type of register inbuilt that is:

- Command Register
- Data Register

Command register is used to insert a special command into the LCD. While Data register is used to insert a data into the LCD. Command is a special set of data which is used to give the internal command to LCD like Clear screen, move to line 1 character 1, setting up the cursor etc.

The HD44780 interface supports two modes of operation, 8-bit and 4-bit. Using the 4-bit mode is more complex, but reduces the number of active connections needed. The operation mode must always be set using the Function Set command, it is not defined at power-up whether the chip is in 8-bit or 4-bit mode. For this reason, in 4-bit mode a command is sent in two operations. To enable 4-bit mode the Function Set command must be sent three times. Once in 4-bit mode, character and control data are transferred as pairs of 4-bit "nibbles" on the upper data pins, D4-D7.

Register Select/Control Pin: This pin toggles among command or data register, used to connect a microcontroller unit pin and obtains either 0 or 1 (0 = data mode, and 1 = command mode).

Read/Write/Control Pin: This pin toggles the display among the read or writes operation, and it is connected to a microcontroller unit pin to get either 0 or 1 (0 = Write Operation, and 1 = Read Operation).

Enable/Control Pin: This pin should be held high to execute Read/Write process, and it is connected to the microcontroller unit & constantly held high.

The details of remaining pins are mentioned in Table 9.1

Table 9-1: LCD Functions

Pin No	Function	Name
1	Ground (0V)	Ground
2	Supply voltage; 5V (4.7V – 5.3V)	V _{CC}
3	Contrast adjustment; through a variable resistor	V _{EE}
4	Selects command register when low; and data register when high	Register Select
5	Low to write to the register; High to read from the register	Read/write
6	Sends data to data pins when a high to low pulse is given	Enable
7	8-bit data pins	DB0
8		DB1
9		DB2
10		DB3
11		DB4
12		DB5
13		DB6
14		DB7
15	Backlight V _{CC} (5V)	Led+
16	Backlight Ground (0V)	Led-

LCD Commands

There are some preset commands instructions in LCD, which we need to send to LCD through some microcontroller. Some important command instructions are shown in Table 9.2:

Table 9-2: LCD Commands

Hex Code	Command to LCD Instruction Register
0F	LCD ON, cursor ON
01	Clear display screen
02	Return home
04	Decrement cursor (shift cursor to left)
06	Increment cursor (shift cursor to right)
05	Shift display right
07	Shift display left
0E	Display ON, cursor blinking
80	Force cursor to beginning of first line
C0	Force cursor to beginning of second line
38	2 lines and 5×7 matrix
83	Cursor line 1 position 3
3C	Activate second line
08	Display OFF, cursor OFF
C1	Jump to second line, position 1
OC	Display ON, cursor OFF
C1	Jump to second line, position 1
C2	Jump to second line, position 2

In-Lab Exercise

Task 1: Write a program that will display string on a LCD

Complete the program that will display string on a LCD.

Hint: The STM32F4 training board has LCD driven by I/O pins PE7 to PE15, so you have to configure these pins for the interfacing.

Program

```
#include <stdio.h>
#include <stm32f4xx.h>
void lcd_ini(void);
void lcd_data(char j);
void lcd_cmd(char i);
volatile uint32_t msTicks;                                /* counts 1ms timeTicks */
/*-----
   SysTick_Handler
   -----*/
void SysTick_Handler(void) {
    msTicks++;
}
void Delay (uint32_t dlyTicks)
{
    uint32_t loop=0,dly=0,loope=0;
    dly = dlyTicks ;
    for(loop=0;loop<dly;loop++)
        {
            for(loope=0;loope<29000;loope++)
                {
                    __nop();
                }
        }
}
unsigned long LCDDATA=0;
/*-----
   MAIN function
   -----*/
int main (void) {

    SystemCoreClockUpdate();                                // Get Core Clock Frequency
    /*-----
       Add your code to enable Ports that can be used for this program.
       -----*/

    GPIOB->BSRRH = ((1 << 0) );    // LCD RW -> 0
    lcd_ini();
    lcd_cmd(0x80); // line 1
    lcd_data('M'); // S
    lcd_data('S'); // T
    lcd_data('I'); // SP
    lcd_data(' '); // T
    lcd_data('L'); // D
    lcd_data('A'); // D
    lcd_data('B'); // D
    while(1){
    }
}
void lcd_ini(void)
{
    Delay(10);
    lcd_cmd(0x38);
    lcd_cmd(0x0C);
    lcd_cmd(0x01);
    Delay(10);
}
```

```

}
void lcd_cmd(char i)
{
    unsigned long r=0;
    char loop=0;
    r |= i;
    for(loop=0;loop<=7;loop++)
    {
        r = r << 1;
    }
    GPIOB->BSRRH = ((1 << 1) );
    LCDDATA = r;
    GPIOE->ODR &= 0x000000FF;
    GPIOE->ODR |= LCDDATA;
    GPIOE->BSRRL = ((1 << 7) );
    Delay(100);
    GPIOE->BSRRH = ((1 << 7) );
}
void lcd_data(char j)
{
    unsigned long r=0;
    char loop=0;
    r |= j;
    for(loop=0;loop<=7;loop++)
    {
        r = r << 1;
    }
    GPIOB->BSRRL = ((1 << 1) );
    LCDDATA = r;
    GPIOE->ODR &= 0x000000FF;
    GPIOE->ODR |= LCDDATA;
    GPIOE->BSRRL = ((1 << 7) );
    Delay(100);
    GPIOE->BSRRH = ((1 << 7) );
}

```

Task 2: Write a complete C program that display two strings

Write a complete C program that display two strings (e.g. Your Name and Your Reg #.) on LCD in such a fashion that the first message will scroll from left to right at the upper line of LCD and the second message will scroll from right to left at the lower line of LCD.

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____ **Date:** _____

LAB # 10

To Explain and Reproduce the Working of 4x3 Keypad on STM32F407 Trainer Board using C Programming.

Objectives

- To explain the working of 4x3 keypad interfacing with the STM32F407 using trainer board.
- To modify the program and reproduce its output on STM32F407 trainer board using C programming.

Pre-Lab Exercise

Read the details given below in order to comprehend the basic operation of Keypad, and interfacing of keypad with microcontroller. Keypad can also be interfaced with LCD or seven segment to perform different tasks in different combinations.

Introduction

Using slide switches and push buttons as inputs is ideal for situations where limited information needs to be fed into a system. In order to introduce more detailed information, such as alphanumeric characters into a system, we normally make use of a keypad or keyboard. As an example, think of the keyboard connected to a computer or the keypad on a telephone.

Matrices

For slide switches and push buttons we have used individual port lines to monitor their status. For keypads and keyboards, it would not be feasible to do this to dedicate a line to each switch. Imagine if a keypad had 12 keys, we would have to use 12 dedicated port lines.

Therefore, keyboards are organized in a matrix of rows and column at the lowest level.

The CPU access both rows and columns through ports; therefore, with two 8-bit ports, an 8 x 8 matrix of keys can be connected to a microprocessor. When a key is pressed, a row and column make a contact; otherwise, there is no connection between rows and columns.

Figure 10-1 shows that each switch of the matrix is connected to a row and a column. If the first left hand switch of the matrix is pressed, Row 0 and Column 1 would be electrically connected.

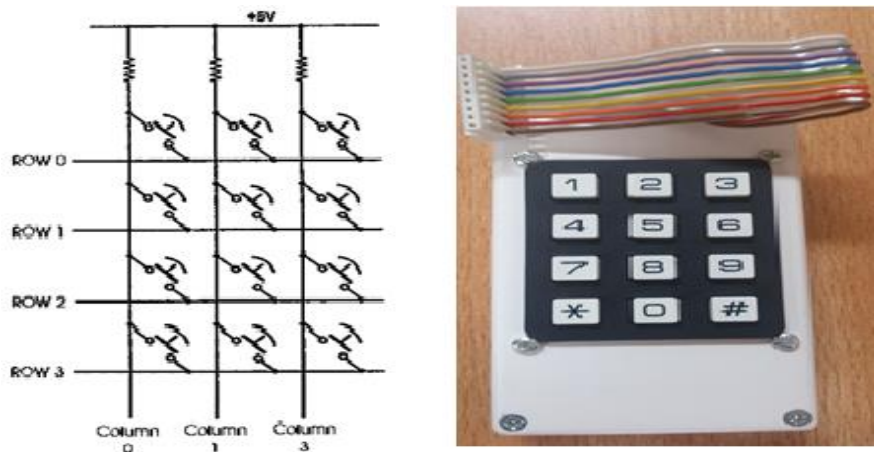


Figure 10-1: Keypad interfacing

The keypad module used in this chapter is constructed in the form of a matrix. It has 12 keys configured in 4x3 matrix. Due to construction of the keypad, the row and column connections are different from the external appearance on the board. The diagram in above figure shows actual electrical connection of the rows and columns. A special routine can be used in the software to take into account these row and column connections.

In its normal state the input will be connected to +5V. The resistor is included to prevent the supply from shorting out when the key has been pressed.

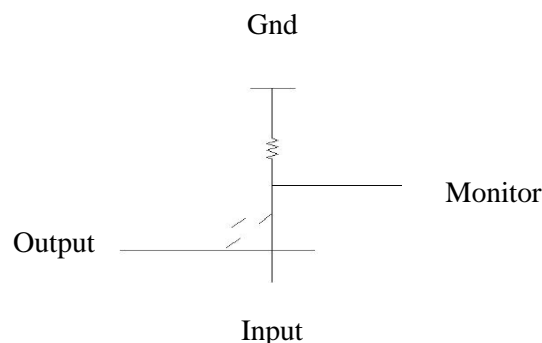


Figure 10-2: Basic Configuration of Switch

In order to determine whether a key has been pressed, logic 0 is sent along each of the row connections. The column connections are then monitored to see if any go low. By knowing which combination of row and column has been connected, it is then possible to determine which switch has been pressed.

In-Lab Exercise

Task 1: Keypad Scanning

Complete the program that continuously scans the keypad and display the value of pressed key on the LCD.

Hint: The STM32F4 training board has keypad driven by I/O port D, so you have to configure this port for the interfacing.

Program

```
#include <stdio.h>
#include <stm32f4xx.h>
void lcd_ini(void);
void lcd_data(char j);
void lcd_cmd(char i);
char keypad(void);
volatile uint32_t msTicks;                                /* counts 1ms timeTicks */
/*-----
   SysTick_Handler
   -----*/
void SysTick_Handler(void) {
    msTicks++;
}
void Delay (uint32_t dlyTicks) {
    uint32_t loop=0,dly=0,loope=0;
    dly = dlyTicks ;
    for(loop=0;loop<dly;loop++){
        for(loope=0;loope<29000;loope++){
            __nop();
        }
    }
}
unsigned long LCDDATA=0;
unsigned long op=0;
int main (void) {
    SystemCoreClockUpdate();                               // Get Core Clock Frequency
    /*-----
       Add your code to enable Ports that can be used for this program.
       -----*/
    GPIOB->BSRRH = ((1 << 0) );    // LCD RW -> 0
    lcd_ini();
    lcd_cmd(0x80); // line 1
    lcd_data(' '); // SP
    lcd_data(' '); // SP
    lcd_data(' '); // SP
    lcd_data('K'); // K
    lcd_data('E'); // E
    lcd_data('Y'); // Y
    lcd_data('P'); // P
    lcd_data('A'); // A
    lcd_data('D'); // D
    lcd_data(' '); // SP
    lcd_data('4'); // 4
    lcd_data('x'); // X
    lcd_data('3'); // 3
    lcd_data(' '); // SP
    lcd_data(' '); // SP
    lcd_data(' '); // SP
    while(1){
        keypad();
    }
}
void lcd_ini(void)
{
    Delay(10);
    lcd_cmd(0x38);
    lcd_cmd(0x0C);
    lcd_cmd(0x01);
    Delay(10);
}
void lcd_cmd(char i)
{
    unsigned long r=0;
    char loop=0;
    r |= i;
```

```

for(loop=0;loop<=7;loop++)
{
    r = r << 1;
}
GPIOB->BSRRH = ((1 << 1) );
        LCDDATA = r;
GPIOE->ODR &= 0x000000FF;
GPIOE->ODR |= LCDDATA;
GPIOE->BSRRL = ((1 << 7) );
Delay(100);
GPIOE->BSRRH = ((1 << 7) );
}
void lcd_data(char j)
{
    unsigned long r=0;
    char loop=0;
    r |= j;
    for(loop=0;loop<=7;loop++)
    {
        r = r << 1;
    }
    GPIOB->BSRRL = ((1 << 1) );
        LCDDATA = r;
GPIOE->ODR &= 0x000000FF;
GPIOE->ODR |= LCDDATA;
        GPIOE->BSRRL = ((1 << 7) );
        Delay(100);
        GPIOE->BSRRH = ((1 << 7) );
}
char keypad(void){
    char key_pressed=0;
    /*Row 1 = 1 & scanning of Col 1 , Col 2 , Col 3*/
        GPIOD->BSRRL = ((1 << 7) );
    GPIOD->BSRRH = ((1 << 6) );
    GPIOD->BSRRH = ((1 << 4) );
    GPIOD->BSRRH = ((1 << 2) );
        if((GPIOD->IDR & 0x00000002) == 0x00000002) {
            key_pressed = 1 ;
            lcd_cmd(0xC7); // line 2
            lcd_data('1');
            Delay(100);
            GPIOD->IDR = 0x00000000;
        }// End if( bt =(1 << 1))
        else if((GPIOD->IDR & 0x00000008) == 0x00000008) {
            key_pressed = 2 ;
            lcd_cmd(0xC7); // line 2
            lcd_data('2');
            Delay(100);
            GPIOD->IDR = 0x00000000;
        }// End if( bt =(1 << 1))
        else if((GPIOD->IDR & 0x00000008) == 0x00000008) {
            key_pressed = 3 ;
            lcd_cmd(0xC7); // line 2
            lcd_data('3');
            Delay(100);
            GPIOD->IDR = 0x00000000;
        }// End if( bt =(1 << 1))

    /*-----
    Add your code that monitors row 2,3,4 and scan continuously column 1, 2 and 3.
    Also display digit on LCD if any key pressed from row 2.
    *-----*/
}

```

Task 2: Write a complete C program to interface a keypad with STM32F4

Write a complete C program that will interface a keypad with STM32F4 microcontroller and continuously scans it. Whenever a key is pressed, the corresponding character will be displayed on the 7-Segment. Verify your program by running it on the STM32F4 Discovery board.

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____ **Date:** _____

LAB # 11

To Explain and Reproduce the Working of Stepper and DC motor on STM32F407 Trainer Board using C Programming.

Objectives

- To explain the working of Stepper and DC motor with the STM32F407 using trainer board.
- To modify the program and display its output on STM32F407 trainer board using C programming.
-

Pre-Lab Exercise

Read the details given below in order to comprehend the basic operation of buzzer and interfacing of buzzer with microcontroller.

Introduction to Motors

Motors (Stepper and DC) convert electric energy to mechanical motion. The type of motor chosen for an application depends on the characteristics needed in that application. These include:

- How fast you want the object to move?
- The weight, size of the object to be moved.
- The cost and size of the motor.
- The accuracy of position or speed control needed

Stepper motor

It offers slow, precise rotation and easy control. It has advantage over servo motors or DC motors in positional control where servos require a complicated feedback mechanism. A stepper motor has positional control via its nature of rotation by fractional increments. Suited for 3D printers and similar devices where position is fundamental.

Stepper has many electromagnets. The rotor carries a set of permanent magnets, and the stator has the coils. Stepper controlled by sequential turning on and off of electromagnet coils. Each pulse moves another step, providing a step angle. It can be moved to and held in a desired position. It can be rotated continuously at a controlled speed.

- For 4 steps per revolution
 - $360/4 = 90^\circ$ step angle
- If motor of 100 steps per revolution, then
 - $360/100 = 3.6^\circ$ step angle
 - Because Rotor not only has Only 2-sided Magnet, it has multiple teeth
 - After 4 steps, rotor moves only one tooth pitch
- In Stepper Motor with 100 steps per revolution, rotor would have 25 teeth
 - $4 \times 25 \text{ teeth} = 100 \text{ steps needed to complete one revolution}$

A stepper drive is the driver circuit that controls how the stepper motor operates. Stepper drives work by sending current through various phases in pulses to the stepper motor. There are four types: wave drive, full step drive, half step drive and microstepping drive.

Wave drive works with only one phase turned on at a time. Consider the illustration in Figure 11.1. When the drive energizes pole A (a south pole) shown in green, it attracts the north pole of the rotor. Then when the drive energizes B and switches A off, the rotor rotates 90° and this continues as the drive energizes each pole one at a time.

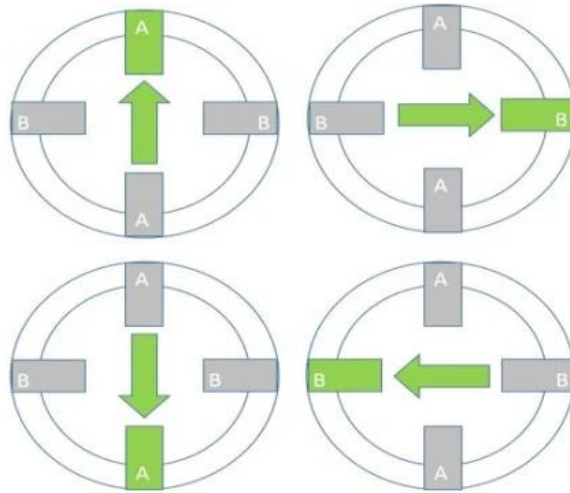


Figure 11-1: Wave drive

Engineers rarely use wave driving it is inefficient and provides little torque, because only one phase of the motor engages at a time.

Full step drive has its name because two phases are on at a time. As seen in Figure 11.2, if the drive energizes both A and B poles as south poles (shown in green), then the rotor's north pole attracts to both equally and aligns in the middle of the two. As the energizing sequence continues on like this, the rotor continuously ends up aligning in-between two poles.

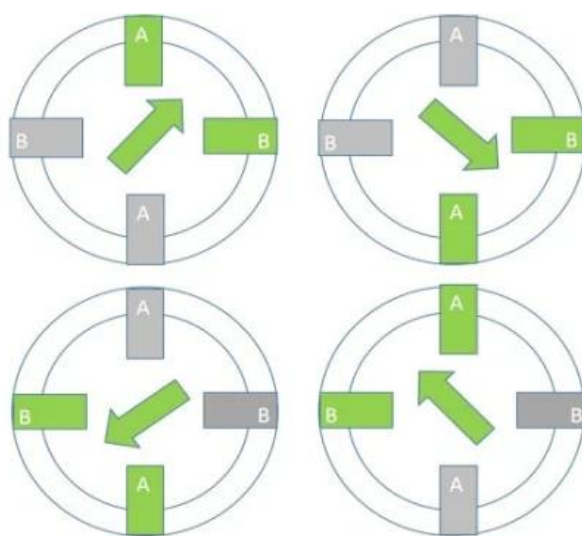


Figure 11-2: Full step drive

Two-phase-on driving gets no finer resolution than one-phase on, but it does produce more torque.

Half step drive has its name for the way the drive energizes either 1 or 2 phases at any specific time. In this driving method, also known as half-stepping, the drive energizes pole A (shown in green) ... then energizes poles A and B ... then energizes pole B ... and so forth. This can be seen in Figure 11.3

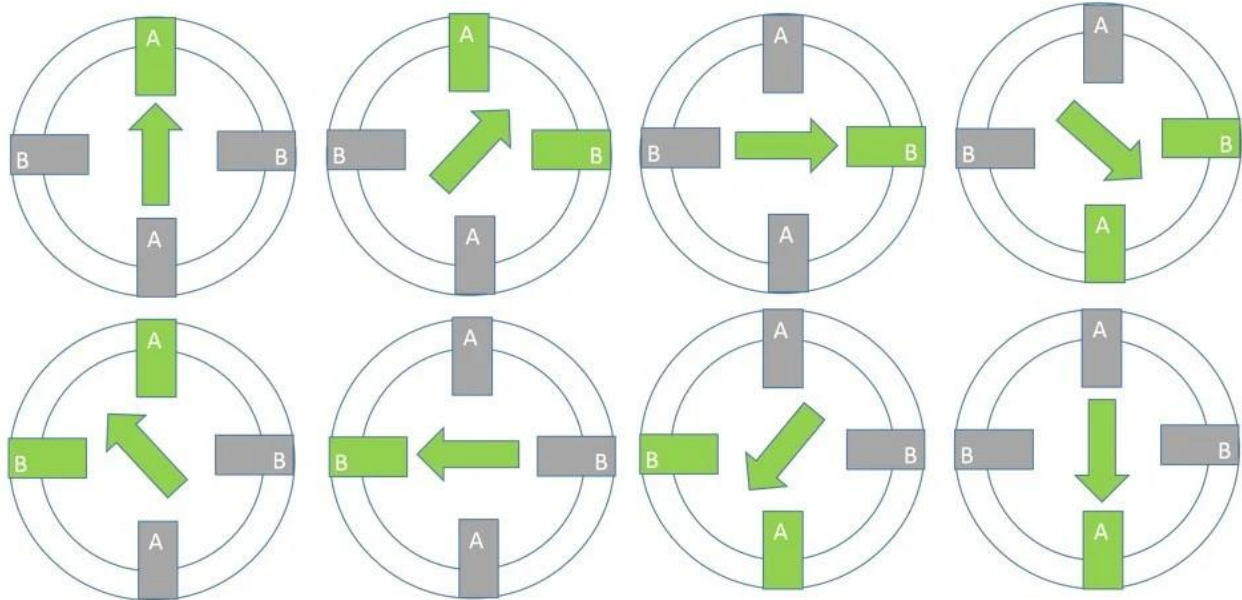


Figure 11-3: Half step drive

DC motor

The electric motor operated by dc is called dc motor. It is the most common and cheapest motor available for use. This is a device that converts DC electrical energy into a mechanical energy. Figure 11.4 shows the physical motor and the conversion phenomena. The important features of a DC motor are as follows:

- Powered with two wires from source
- Draws large amounts of current
- Cannot be wired straight from Controller
- Does not offer accuracy or speed control
- Fast, continuous rotation motors
- Used for anything that needs to spin at a high RPM e.g., car wheels, fans etc.

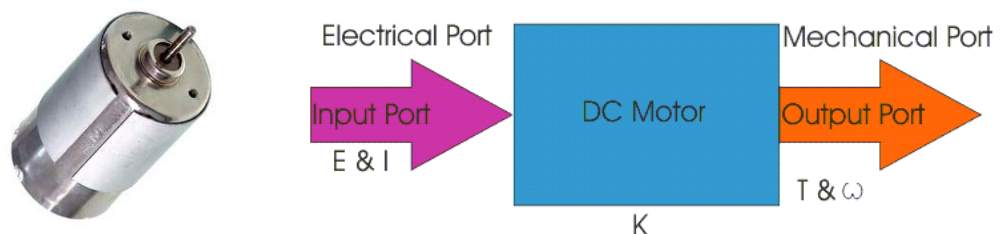


Figure 11-4: DC Motor

The interfacing of DC motor with microcontroller requires a bridge circuitry in between called as an H-Bridge. It makes use of the switching transistors and allows the bi-directional rotation of the DC motor through switching contacts. Figure 11.5 shows the internal circuitry of the H-bridge constructed with

MOSFET switches and is further demonstrated with two directional current flow in Figure. 11.6 with switching scheme in Table 11-1.

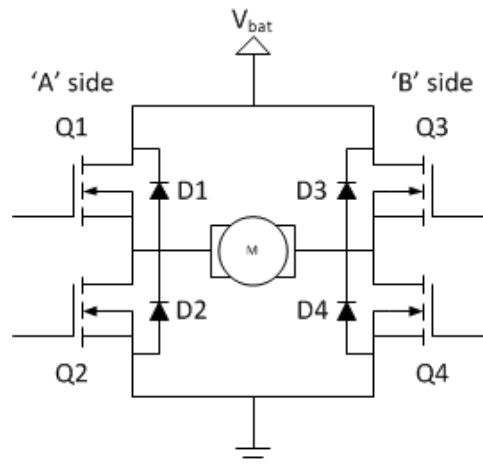


Figure 11-5: H-Bridge internal circuitry

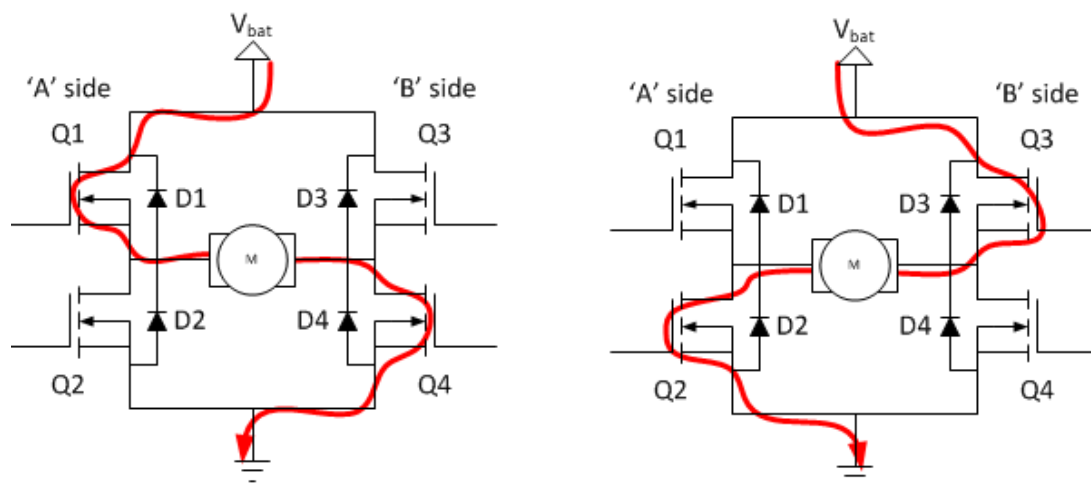


Figure 11-6: H-Bridge directional flow

Table 11-1: H-Bridge mode

Q1	Q2	Q3	Q4	mode
close	open	open	open	motor coasts
close	open	open	close	forward motoring / move right
close	open	close	open	motor brakes (zero potential voltage)
open	close	open	open	motor coasts
open	close	open	close	motor brakes (zero potential voltage)
open	close	close	open	reverse motoring/ move left
open	open	open	open	motor coasts
open	open	open	close	motor coasts
open	open	close	open	motor coasts

In-Lab Exercise

Task-1 Write a complete C program to interface a stepper motor with STM32F4

Complete the following code that shows how a Stepper motor can be interfaced with STM32F4 discovery. It rotates the stepper motor connected to Port B in one direction with fixed speed using wave drive.

```
#include <stdio.h>
#include <stm32f4xx.h>
volatile uint32_t msTicks;                                /* counts lms timeTicks */
/*-----
SysTick_Handler
-----*/
void SysTick_Handler(void) {
    msTicks++;
}
void Delay (uint32_t dlyTicks) {
    uint32_t loop=0,dly=0,loope=0;
    dly = dlyTicks ;
    for(loop=0;loop<dly;loop++){
        for(loope=0;loope<29000;loope++){
            __nop();
        }
    }
}
int main (void) {
    SystemCoreClockUpdate();                            // Get Core Clock Frequency
    /*-----
    Add your code to enable Ports that can be used for this program.
    -----*/
    while(1) { // Infinite Loop

        GPIOB->BSRRL = ((1 << 11) );
        GPIOB->BSRRH = ((1 << 12) );
        GPIOB->BSRRH = ((1 << 13) );
        GPIOB->BSRRH = ((1 << 14) );
        Delay(100);
        /*-----
        Add your code to verify aforementioned 3 driving modes.
        -----*/
    }
}
```

Task-2

Two push buttons are connected with STM32, which are used to rotate a DC motor. Design a scenario which rotates the DC motor according to the following criteria

Push Button 1	Push Button 2	DC Motor Status
LOW	LOW	STAND STILL
LOW	HIGH	CLOCKWISE
HIGH	LOW	ANTI-CLOCKWISE
HIGH	HIGH	PULSE (2 sec) STOP (2 sec) REPEAT

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____ **Date:** _____

LAB # 12

To Explain and Reproduce the Working of Temperature Sensor and LDR on STM32F407 Trainer Board using C Programming.

Objectives

- To explain the working of temperature sensor and light dependent resistor (LDR) interfacing with the STM32F407 using trainer board.
- To modify the program and display its output on STM32F407 trainer board using C programming.

Pre-Lab Exercise

Read the details given below in order to comprehend the basic operation of temperature sensor (STLM20) and interfacing of STLM20 with microcontroller. STLM20 can be interfaced with LCD to complete different tasks in different combinations.

Also read the details given below in order to comprehend the basic operation of Light Dependent Resistor (LDR) and interfacing of LDR with microcontroller. LDR can be interfaced with LCD to perform different tasks in different combinations.

Introduction to ADC

Analog-to-digital converters are among the most widely used devices for data acquisition. Digital computers use binary values, but in the physical world everything is analog. Temperature, pressure, humidity, and velocity are a few examples of physical quantities that we deal with every day. A physical quantity is converted to electrical signals using a device called a transducer. Transducers are also referred to as sensors. Sensors for temperature, velocity, pressure, light and many other natural quantities produce an output that is voltage or current. Therefore, we need an analog-to-digital converter to translate the analog signals to digital numbers so that the microcontroller can read and process them. The basic process is shown in the *Figure 12-1*:

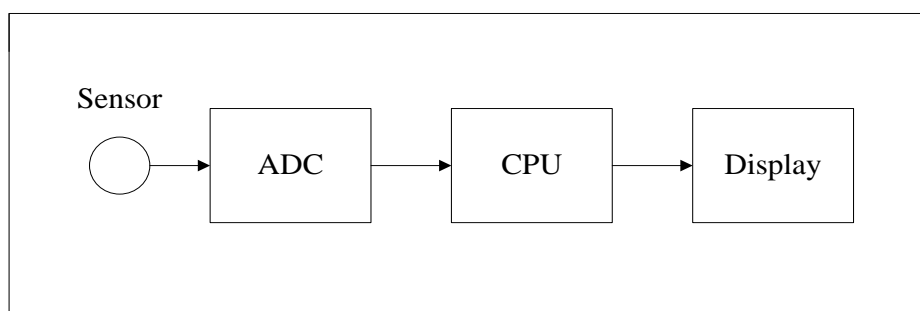


Figure 12-1: Microcontroller connection to sensor via ADC

Some of the major characteristics of the ADC are as follows.

Resolution

ADC has n-bit resolution; where n can be 8, 10, 12, 16, or even 24 bits. The higher resolution ADC provides a smaller step size, where step size is the smallest change that can be discerned by an ADC. We can control the step size with the help of what is called V_{ref} . This is discussed below.

V_{ref}

V_{ref} is an input voltage use for the reference voltage. The voltage connected to this pin, along with the resolution of the ADC chip, dictate the step size. For an 8-bit ADC, the step size is $V_{\text{ref}} / 256$ because it is an 8-bit ADC, and 2 to the power of 8 gives us 256 steps.

Digital data output

In the 12-bit ADC, we have a 12-bit data output of D0—D11. To calculate the output voltage, we use the following formula:

$$D_{\text{out}} = V_{\text{in}} / \text{step size}$$

Where D_{out} = digital data output (in decimal), V_{in} = analog input voltage, and step size is the smallest change, which is $V_{\text{ref}}/256$ for an 8-bit ADC. The result of a conversion may be optionally aligned left or right in the 16 bit result register. Only 12 bits of the result are significant. In regular mode, the other bits are filled with zeros.

STLM20 Temperature Sensor

The STLM20 is a precision analog output temperature sensor for low current applications where maximizing battery life is important. It operates over a $-55\text{ }^{\circ}\text{C}$ to $130\text{ }^{\circ}\text{C}$ (grade 7) or $-40\text{ }^{\circ}\text{C}$ to $85\text{ }^{\circ}\text{C}$ (grade 9) temperature range. The power supply operating range is 2.4 V to 5.5 V. The maximum temperature accuracy of the STLM20 is $\pm 1.5\text{ }^{\circ}\text{C}$ ($\pm 0.5\text{ }^{\circ}\text{C}$ typ) at an ambient temperature of $25\text{ }^{\circ}\text{C}$ and V_{CC} of 2.7 V. The temperature error increases linearly and reaches a maximum of $\pm 2.5\text{ }^{\circ}\text{C}$ at the temperature range extremes. The temperature range is affected by the power supply voltage. For the temperature grade 7 device, a power supply voltage of 2.7 V to 5.5 V, the temperature range extremes are $+130\text{ }^{\circ}\text{C}$ and $-55\text{ }^{\circ}\text{C}$ (decreasing the power supply voltage from 2.7 V to 2.4 V changes the low end of the operating temperature range from $-55\text{ }^{\circ}\text{C}$ to $-30\text{ }^{\circ}\text{C}$, while the positive remains at $+130\text{ }^{\circ}\text{C}$). The STLM20 has a maximum quiescent supply current of 8 μA . Therefore, self-heating is negligible.

Transfer function

The STLM20's transfer function can be described in different ways, with varying levels of precision. A simple linear transfer function, with good accuracy near $25\text{ }^{\circ}\text{C}$ is expressed as:

Equation (first order linear equation)

$$VO = (\text{ADC Value} / 4095) \times 3.0$$

$$T = 85.543 * (1.8663 - VO)$$

Introduction to LDR

A photo resistor or light-dependent resistor (LDR) or photocell is a light-controlled variable resistor. The resistance of a photo resistor decreases with increasing incident light intensity; in other words, it exhibits photoconductivity. A photo resistor can be applied in light-sensitive detector circuits, and light- and dark-activated switching circuits.

A photo resistor is made of a high resistance semiconductor. The most commonly used LDR is ORP-12. In the dark, an ORP-12 can have a resistance as high as several mega ohms ($M\Omega$), while in the light, a photo resistor can have a resistance as low as a few hundred ohms. If incident light on a photo resistor exceeds a certain frequency, photons absorbed by the semiconductor give bound electrons enough energy to jump into the conduction band. The resulting free electrons conduct electricity, thereby lowering resistance. The resistance range and sensitivity of a photo resistor can substantially differ among dissimilar devices. Moreover, unique photo resistors may react substantially differently to photons within certain wavelength bands.

In-Lab Exercise

Task 1: Design a thermostat

Design a thermostat system using STM32F4 microcontroller. The system measures the room temperature and after performing Analog to Digital Conversion display the measured temperature on LCD.

Hint: The STM32F4 training board has STLM20 driven by I/O port C, so you have to configure this port for the interfacing.

Program

```
#include "stm32f4_discovery.h"
#include <stdio.h>
void lcd_ini(void);
void lcd_data(char j);
void lcd_cmd(char i);
void Delay (uint32_t dlyTicks);
void display_value(unsigned int value);
void Delay (uint32_t dlyTicks) {
uint32_t loop=0,dly=0,loope=0;
dly = dlyTicks ;
for(loop=0;loop<dly;loop++){
    for(loope=0;loope<29000;loope++){
        __nop();
    }
}
}
unsigned long LCDDATA=0;
float T=0 , V =0;
__IO uint16_t ADC3ConvertedValue = 0;
__IO uint32_t ADC3ConvertedVoltage = 0;
void ADC3_CH13_Config(void);
int main(void)
{
    RCC->AHB1ENR  |= (1 << 0) ;           // Enable GPIOA clock
    RCC->AHB1ENR  |= (1 << 1) ;           // Enable GPIOB clock
    RCC->AHB1ENR  |= (1 << 2) ;           // Enable GPIOC clock
    RCC->AHB1ENR  |= (1 << 3) ;           // Enable GPIOD clock
    RCC->AHB1ENR  |= (1 << 4) ;           // Enable GPIOE clock
    GPIOA->MODER   = 0xA8000000;
    GPIOA->OTYPER  = 0xA8000000;
    GPIOA->OSPEEDR = 0xA800AAAA;
    GPIOA->PUPDR   = 0xA8000000;
    GPIOB->MODER   = 0x00000005;
    GPIOB->OTYPER  = 0x00000003;
    GPIOB->OSPEEDR = 0xAAAAAAAA;
    GPIOB->PUPDR   = 0x00000000;
    GPIOD->MODER   = 0x00000000;
    GPIOD->OTYPER  = 0x00000000;
```

```

        GPIOD->OSPEEDR = 0xAAAAAAAA;
        GPIOD->PUPDR = 0x00000000;
        GPIOE->MODER = 0x55555555;
        GPIOE->OTYPER = 0x0000FF00;
        GPIOE->OSPEEDR = 0xAAAAAAAA;
        GPIOE->PUPDR = 0x00000000;
    ADC3_CH13_Config();
    ADC_SoftwareStartConv(ADC3);
    GPIOB->BSRRH = ((1 << 0) );    // LCD RW -> 0
    lcd_ini();
    lcd_cmd(0x80); // line 1
    lcd_data('M'); // M
    lcd_data('S'); // S
    lcd_data('I'); // I
    lcd_data(' '); // SPACE
    lcd_data('L'); // L
    lcd_data('A'); // A
    lcd_data('B'); // B
        lcd_cmd(0x01);
    lcd_cmd(0x80); // line 1
    lcd_data('T');
    lcd_data('E');
    lcd_data('M');
    lcd_data('P');
        lcd_data('=');
    while (1)
    {
        ADC3ConvertedValue = ADC_GetConversionValue(ADC3);
    }
    /*-----
Add your code here that gets value from ADC and convert into voltage
    -----*/

}
}
void ADC3_CH13_Config(void)
{
    ADC_InitTypeDef      ADC_InitStructure;
    ADC_CommonInitTypeDef ADC_CommonInitStructure;
    GPIO_InitTypeDef      GPIO_InitStructure;
    /* Enable ADC3, DMA2 and GPIO clocks *****/
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA2 | RCC_AHB1Periph_GPIOA, ENABLE);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC3, ENABLE);
    /* Configure ADC3 Channel 13 pin as analog input *****/
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL ;
    GPIO_Init(GPIOC, &GPIO_InitStructure);
    /*ADC Common Init *****/
    ADC_CommonInitStructure.ADC_Mode = ADC_Mode_Independent;
    ADC_CommonInitStructure.ADC_Prescaler = ADC_Prescaler_Div2;
    ADC_CommonInitStructure.ADC_DMAAccessMode = ADC_DMAAccessMode_Disabled;
    ADC_CommonInitStructure.ADC_TwoSamplingDelay = ADC_TwoSamplingDelay_10Cycles;
    ADC_CommonInit(&ADC_CommonInitStructure);
    /* ADC3 *****/
    ADC_InitStructure.ADC_Resolution = ADC_Resolution_12b;
    ADC_InitStructure.ADC_ScanConvMode = DISABLE;
    ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
    ADC_InitStructure.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_None;
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
    ADC_InitStructure.ADC_NbrOfConversion = 1;
    ADC_Init(ADC3, &ADC_InitStructure);
    /* ADC3 regular channel12 configuration *****/
    ADC_RegularChannelConfig(ADC3, ADC_Channel_13, 1, ADC_SampleTime_3Cycles);
    /* Enable ADC3 */
    ADC_Cmd(ADC3, ENABLE);
}
#ifdef USE_FULL_ASSERT
void assert_failed(uint8_t* file, uint32_t line)

```



```

    while (1)/* Infinite loop */
    {
    }
}

void lcd_ini(void)
{
    Delay(10);
    lcd_cmd(0x38);
    lcd_cmd(0x0C);
    lcd_cmd(0x01);
    Delay(10);
}

void lcd_cmd(char i)
{
    unsigned long r=0;
    char loop=0;
    r |= i;
    for(loop=0;loop<=7;loop++)
    {
        r = r << 1;
    }
    GPIOB->BSRRH = ((1 << 1) );
    LCDDATA = r;
    GPIOE->ODR &= 0x000000FF;
    GPIOE->ODR |= LCDDATA;
    GPIOE->BSRRL = ((1 << 7) );
    Delay(100);
    GPIOE->BSRRH = ((1 << 7) );
}

void lcd_data(char j)
{
    unsigned long r=0;
    char loop=0;
    r |= j;
    for(loop=0;loop<=7;loop++){
        r = r << 1;
    }
    GPIOB->BSRRL = ((1 << 1) );
    LCDDATA = r;
    GPIOE->ODR &= 0x000000FF;
    GPIOE->ODR |= LCDDATA;
    GPIOE->BSRRL = ((1 << 7) );
    Delay(100);
    GPIOE->BSRRH = ((1 << 7) );
}

void display_value(unsigned int value){
    unsigned int seg1=0,seg2=0,seg3=0,seg=0;
    seg3=value%10;
    seg2=(value%100)/10;
    seg1= value/100;
    lcd_cmd(0x85); // line 1
    lcd_data(seg1 + 0x30 ); // S
    lcd_data(seg2 + 0x30); // S
    lcd_data(seg3 + 0x30); // S
} // End void display_value(unsigned int value)

```

Task 2: Write a program that continuously scans the intensity of the light using LDR

Complete the program that continuously scans the intensity of the light using LDR and after performing Analog to Digital Conversion display the measured intensity on LCD.

Hint: The STM32F4 training board has ORP-12 driven by I/O port C, so you have to configure this port for the interfacing.

Program

```
#include "stm32f4_discovery.h"
#include <stdio.h>
void lcd_ini(void);
void lcd_data(char j);
void lcd_cmd(char i);
void Delay (uint32_t dlyTicks);
void display_value(unsigned int value);
void Delay (uint32_t dlyTicks) {
uint32_t loop=0,dly=0,loope=0;
dly = dlyTicks ;
for(loop=0;loop<dly;loop++){
    for(loope=0;loope<29000;loope++){
        __nop();
    }
}
}
unsigned long LCDDATA=0;
__IO uint16_t ADC3ConvertedValue = 0;
__IO uint32_t ADC3ConvertedVoltage = 0;
void ADC3_CH1_Config(void);
int main(void)
{
    RCC->AHB1ENR |= (1 << 0) ;           // Enable GPIOA clock
    RCC->AHB1ENR |= (1 << 1) ;           // Enable GPIOB clock
    RCC->AHB1ENR |= (1 << 2) ;           // Enable GPIOC clock
    RCC->AHB1ENR |= (1 << 3) ;           // Enable GPIOD clock
    RCC->AHB1ENR |= (1 << 4) ;           // Enable GPIOE clock
    GPIOA->MODER = 0xA8000000;
    GPIOA->OTYPER = 0xA8000000;
    GPIOA->OSPEEDR = 0xA800AAAA;
    GPIOA->PUPDR = 0xA8000000;
    GPIOB->MODER = 0x00000005;
    GPIOB->OTYPER = 0x00000003;
    GPIOB->OSPEEDR = 0xAAAAAAAA;
    GPIOB->PUPDR = 0x00000000;
    GPIOD->MODER = 0x00000000;
    GPIOD->OTYPER = 0x00000000;
    GPIOD->OSPEEDR = 0xAAAAAAAA;
    GPIOD->PUPDR = 0x00000000;
    GPIOE->MODER = 0x55555555;
    GPIOE->OTYPER = 0x0000FF00;
    GPIOE->OSPEEDR = 0xAAAAAAAA;
    GPIOE->PUPDR = 0x00000000;
    ADC3_CH1_Config();
    ADC_SoftwareStartConv(ADC3);
    GPIOB->BSRRH = ((1 << 0) );         // LCD RW -> 0
    lcd_ini();
    lcd_cmd(0x80); // line 1
    lcd_data('M'); // M
    lcd_data('S'); // S
    lcd_data('I'); // I
    lcd_data(' '); // SPACE
    lcd_data('L'); // L
    lcd_data('A'); // A
    lcd_data('B'); // B
    lcd_cmd(0xC0); // line 2
    lcd_data(' '); // SP
    lcd_data(' '); // SP
    lcd_data(' '); // SP
    lcd_data('S'); // S
    lcd_data('T'); // T
    lcd_data('M'); // M
    lcd_data('3'); // 3
    lcd_data('2'); // 2
    lcd_data('F'); // F
}
```

```

lcd_data('4'); // 4
lcd_data('0'); // 0
lcd_data('7'); // 7
        lcd_cmd(0x01);
        lcd_cmd(0x80); // line 1
lcd_data('L');
lcd_data('D');
lcd_data('R');
lcd_data('=');
while (1)
{
/*-----
Add your code that gets value from ADC and convert into voltage
-----*/

ADC3ConvertedValue = ADC_GetConversionValue(ADC3);
}
}
void ADC3_CH1_Config(void)
{
    ADC_InitTypeDef      ADC_InitStructure;
    ADC_CommonInitTypeDef ADC_CommonInitStructure;
    GPIO_InitTypeDef      GPIO_InitStructure;
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA2 | RCC_AHB1Periph_GPIOA, ENABLE);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC3, ENABLE);
    /* Configure ADC3 Channel 1 pin as analog input *****/
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL ;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    ADC_CommonInitStructure.ADC_Mode = ADC_Mode_Independent;
    ADC_CommonInitStructure.ADC_Prescaler = ADC_Prescaler_Div2;
    ADC_CommonInitStructure.ADC_DMAAccessMode = ADC_DMAAccessMode_Disabled;
    ADC_CommonInitStructure.ADC_TwoSamplingDelay = ADC_TwoSamplingDelay_10Cycles;
    ADC_CommonInit(&ADC_CommonInitStructure);
    ADC_InitStructure.ADC_Resolution = ADC_Resolution_12b;
    ADC_InitStructure.ADC_ScanConvMode = DISABLE;
    ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
    ADC_InitStructure.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_None;
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
    ADC_InitStructure.ADC_NbrOfConversion = 1;
    ADC_Init(ADC3, &ADC_InitStructure);
    /* ADC3 regular channel 1 configuration *****/
    ADC_RegularChannelConfig(ADC3, ADC_Channel_1, 1, ADC_SampleTime_3Cycles);
    /* Enable ADC3 */
    ADC_Cmd(ADC3, ENABLE);
}
void lcd_ini(void)
{
    Delay(10);
    lcd_cmd(0x38);
    lcd_cmd(0x0C);
    lcd_cmd(0x01);
    Delay(10);
}
void lcd_cmd(char i)
{
    unsigned long r=0;
    char loop=0;
    r |= i;
    for(loop=0;loop<=7;loop++)
    {
        r = r << 1;
    }
    /*-----
Add your code that load the data on LCD to display
-----*/
}

```

```

void lcd_data(char j)
{
    unsigned long r=0;
    char loop=0;
    r |= j;
    for(loop=0;loop<=7;loop++)
    {
        r = r << 1;
    }
    GPIOB->BSRRL = ((1 << 1) );
        LCDDATA = r;
    GPIOE->ODR &= 0x000000FF;
    GPIOE->ODR |= LCDDATA;
        GPIOE->BSRRL = ((1 << 7) );
        Delay(100);
        GPIOE->BSRRH = ((1 << 7) );
}

void display_value(unsigned int value){
    unsigned int seg1=0,seg2=0,seg3=0,seg=0;
    seg3=value%10;
    seg2=(value%100)/10;
    seg1= value/100;
    lcd_cmd(0x85); // line 1
    lcd_data(seg1 + 0x30 ); // S
    lcd_data(seg2 + 0x30); // S
    lcd_data(seg3 + 0x30); // S
} // End void display_value(unsigned int value)

```

Task 3: Write a program to interface the temperature sensor STLM20

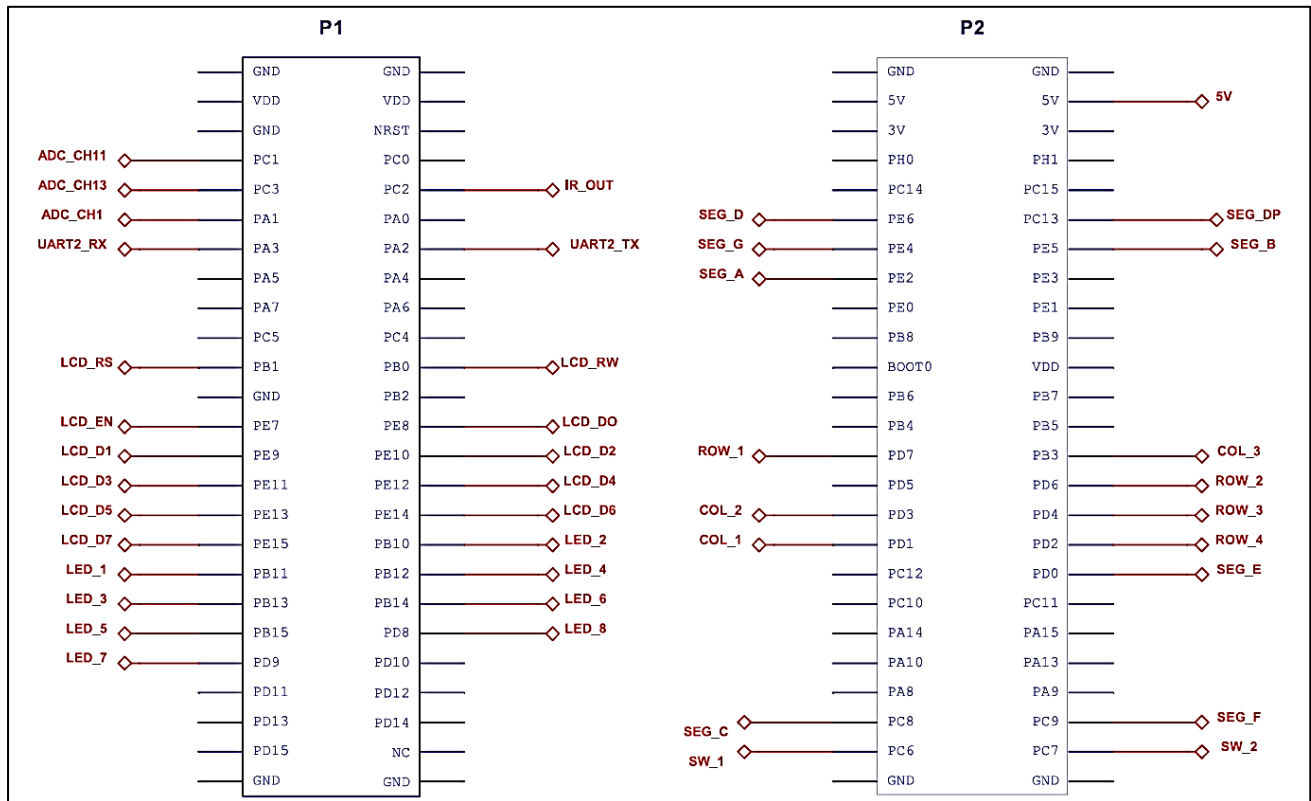
Write a program to interface the temperature sensor STLM20 with STM32F4 microcontroller and LCD. When temperature exceeds 30° C, buzzer beeps indicating that temperature rises. Also implement it on STM32F4 Discovery training board.

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____ **Date:** _____

STM32 Trainer Board Pinout



Hardware Lab Evaluation Rubric

Criteria	Exceeds Expectations (4)	Meets Expectation (3)	Developing (2)	Unsatisfactory (1)
Setup of experiment and implementation (hardware/simulation)	Can identify new ways to set up and implement the experiment without assistance and with detailed understanding of each step	Can fully set up the experiment with successful implementation without assistance	Can setup most of the experiment with some implementation without assistance	Can't set up the experiment without assistance
Follow the procedure/design process	Follows the procedure/design process completely and able to simply or develop alternate procedure/design	Follow the procedure/design process completely	Follows most of the procedures/design process with some errors or omissions	Doesn't follow the procedure/design process
Experimental results	Able to achieve all the desired results with new ways to improve measurements/synthesis	Able to achieve all the desired results	Unable to achieve all the desired results in implementation	Unable to get the results
Safety	Extremely conscious about safety aspects	Observes good laboratory safety procedures	Unsafe lab procedures observed infrequently	Practice unsafe, risky behaviors in lab
Viva	Able to explain design, simulation, implementation and fundamental concepts correctly and provide alternative solutions	Able to explain design, simulation, implementation and fundamental concepts correctly	Able to explain some design and relevant fundamental concepts	Unable to explain design and answer relevant fundamental Concepts

Software Lab Evaluation Rubric

Criteria	Exceeds Expectations (4)	Meets Expectation (3)	Developing (2)	Unsatisfactory (1)
Ability to use software	Student was familiar with the software and was able to use additional features of the software that were not available in instruction set.	Student was familiar with the software and required minimal help from the instructor to perform the experiment	Student demonstrated an ability to use the software but required assistance from the instructor	Student demonstrated little or no ability to perform experiment and required unreasonable amount of assistance from instructor
Ability to follow procedure and/or design a procedure for experiment	<p>Student followed the instructions with no assistance</p> <p>student performed additional experiments or tests beyond those required in instructions</p> <p>if procedure to accomplish an objective in not provided, the student developed a systematic set of tests to accomplish objective</p>	<p>Student followed instructions in the procedure with little or no assistance</p> <p>if procedure was provided, the student was able to determine an appropriate set of experiments to run to produce usable data and satisfy the lab objectives</p>	<p>Student had difficulty with some of the instructions in the procedure and needed clarification from the instructor</p> <p>if procedure was not provided, the student needed some direction in deciding what set of experiments to perform to satisfy the lab objective</p>	<p>Student had difficulty reading the procedure and following directions</p> <p>if procedure was not provided, student was incapable of designing a set of experiments to satisfy given lab objective</p> <p>the data taken was essentially useless</p>
Ability to troubleshoot software	<p>Student developed a good systematic procedure for testing software code that allowed for quick identification of problems</p> <p>student good at analyzing the data</p>	Student demonstrated the ability to test software code in order to identify technical problems, and was able to solve any problems with little or no assistance	Student was able to identify the problems in software code but required some assistance in fixing some of the problems	Student demonstrated little or no ability to troubleshoot software code for the lab.
Q & A	Able to explain program design and fundamental concepts correctly	able to explain most of the program design and relevant fundamental concepts	able to explain some program design and relevant fundamental concepts	unable to explain program design or answer relevant fundamental concepts

Lab Report Evaluation Rubric

Criteria	Exceeds Expectations (4)	Meets Expectation (3)	Developing (2)	Unsatisfactory (1)
Data Presentation	Student demonstrates diligence in creating a set of visually appealing tables and/or graphs that effectively present the experimental data	Experimental data is presented in appropriate format with only a few minor errors or omissions	Experimental data is presented in appropriate format but some significant errors are still evident. Tables could be better organized or some titles, labels or units of measure are missing.	Experimental data is poorly presented. Graphs or tables are poorly constructed with several of the following errors: data is missing or incorrect, units are not included, axis not labeled, or titles missing.
Data Analysis	Student provides a very focused and accurate analysis of the data. All observations are started well and clearly supported by the data	Student has analyzed the data, observed trends, and compared experimental results with theoretical results Any discrepancies are adequately addressed All expected observations are made.	Student has analyzed the data, observed trends, and compared experimental results with theoretical results Any discrepancies are not adequately addressed Some observations that should have been made are missing or poorly supported	Student has simply restated what type of data was taken with no attempt to interrupt trends, explain discrepancies or evaluate the validity of the data in terms of relevant theory Student lacks understanding of the importance of the results
Writing Style	Lab report has no grammatical and/or spelling errors. All sections of the report are very well written	Lab report has very few grammatical and/or spelling errors the sentence flow is smooth	Lab report has some grammatical and/or spelling errors and is fairly readable Student makes effective use of technical terms	Lab report has several grammatical and/or spelling errors and sentence construction is poor