## Contents

1 **Microcontrollers** .................................................. 1
   1.1 Introduction to Microcontrollers ................................. 1
   1.2 General Computer Architectures .................................. 2
   1.3 Basic Block Diagram of a Microprocessor based design ....... 3
   1.4 Basic Block Diagram of a Microcontroller ....................... 4
   1.5 Microcontrollers vs. Microprocessors ............................. 5
   1.6 Choosing a microcontroller ..................................... 6

2 **System Development Concept** ..................................... 7
   2.1 Host and Target .................................................. 7
   2.2 Using a High Level Programming Language ...................... 7
   2.3 Some important Definitions .................................... 9
   2.4 Understanding HEX Code ......................................... 10
      2.4.1 End-of-File (EOF) Records ................................... 11
      2.4.2 Example Intel HEX File ..................................... 12
   2.5 Tools .......................................................... 12
   2.6 Target Understanding ............................................ 12

3 **Target Understanding** ........................................... 13
   3.1 EmxPICM02 Development Board .................................... 13
   3.2 EmxPICM02 Architecture .......................................... 14
   3.3 EmxPICM02 Board Features ....................................... 14
   3.4 EmxPICM02 - Controller: PIC16F887 ............................. 15
      3.4.1 Architecture of PIC16F887 .................................. 17
      3.4.2 Memory Organization ........................................ 18
      3.4.3 Pin Diagram of 16F887 ...................................... 20
      3.4.4 PIC Microcontroller Clock .................................. 21
      3.4.5 I/O Port Configuration ..................................... 22

4 **Work Environment** .................................................. 23
   4.1 Work Space ...................................................... 23
   4.2 Code Structure ................................................... 24
   4.3 Cross Compilation and Hex Dump Creation ....................... 25
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>EmxPICM02 Interfaces</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5.1 LED</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>5.2 Digital Keypad</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5.2.1 Possible Interfaces</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>5.2.2 Methods of Reading Tactile keys</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>5.2.3 Bouncing Effect</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>5.3 External Interrupts</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>5.4 Timers and Counters</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>5.5 Clock I/O</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5.5.1 External Clocking on pins</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>5.5.2 Pulse Width Modulation - PWM</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>5.6 Analog Inputs</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>5.7 SSD</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>5.8 CLCD</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>5.9 Matrix Keypad</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>5.10 Analog Keypad</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>5.11 Data Storage</td>
<td>57</td>
</tr>
<tr>
<td>6</td>
<td>Communication Protocols</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>6.1 Modes of Communication</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>6.2 Types of Communication</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>6.3 UART</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>6.4 SPI</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>6.5 I²C</td>
<td>65</td>
</tr>
<tr>
<td>7</td>
<td>Assignments</td>
<td>71</td>
</tr>
<tr>
<td>A</td>
<td>Assignment Guidelines</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>A.1 Quality of the Source Code</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>A.1.1 Variable Names</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>A.1.2 Indentation and Format</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>A.1.3 Internal Comments</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>A.1.4 Modularity in Design</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>A.2 Program Performance</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td>A.2.1 Correctness of Output</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td>A.2.2 Ease of Use</td>
<td>86</td>
</tr>
</tbody>
</table>

B Grading of Programming Assignments
Chapter 1

Microcontrollers

1.1 Introduction to Microcontrollers

A microcontroller (or MCU) is a computer-on-a-chip used to control electronic devices. It is a component emphasizing self-sufficiency and cost-effectiveness, in contrast to a general-purpose microprocessor (the kind used in a PC). A typical microcontroller contains all the memory and interfaces needed for a simple application, whereas a general purpose microprocessor requires additional chips to provide these functions. A microcontroller is a single integrated circuit, commonly with the following features:

- Central Processing Unit - ranging from small and simple 4-bit processors to sophisticated 32- or 64-bit processors
- Input/Output interfaces such as serial ports
- Peripherals such as timers and watchdog circuits and signal conversion circuits.
- RAM for data storage
- ROM, EPROM, EEPROM or Flash memory for program storage
- Clock generator

This integration drastically reduces the number of chips and the amount of wiring and PCB space that would be needed to produce equivalent systems using separate chips and have proved to be highly popular in embedded systems since their introduction in the 1970s.
1.2 General Computer Architectures

**Harvard Architecture** The Harvard architecture uses physically separate memories for their instructions and data, requiring dedicated buses for each of them. Instructions and operands can therefore be fetched simultaneously.

**Von Neumann’s Architecture** One shared memory for instructions (program) and data with one data bus and one address bus between processor and memory. Instructions and data have to be fetched in sequential order (known as the Von Neumann Bottleneck), limiting the operation bandwidth. Its design is simpler than that of the Harvard architecture. It is mostly used to interface to external memory.
1.3 Basic Block Diagram of a Microprocessor based design

General Purpose Microprocessors based design contains

- CPU for Computers
- No RAM, ROM, I/O on CPU chip itself
- Example: Intels x86, Motorolas 680x0
1.4 Basic Block Diagram of a Microcontroller

The Microcontroller normally contain the basic interfaces like

- On-chip RAM, ROM, I/O ports, Timers
- One or more special functions peripheral(s) like ADC, DAC, I2C, SPI etc.,
- Example: Motorolas 6811, Intels 8051, ARM based, Renesas R8C/22 group and PIC 16X

Notes:
## 1.5 Microcontrollers vs. Microprocessors

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Microcontroller</th>
</tr>
</thead>
<tbody>
<tr>
<td>All separate components</td>
<td>All basic components in a single chip</td>
</tr>
<tr>
<td>More Flexible in case of extension</td>
<td>Less flexible</td>
</tr>
<tr>
<td>More Design Complexity</td>
<td>Less Design Complexity</td>
</tr>
</tbody>
</table>

**Notes:**
1.6 Choosing a microcontroller

As embedded software designers, we are often asked, “Which microcontroller are you going to use?” If you look at all the companies that design microcontrollers, there seems to be an unending sea of choices.

So how do you choose?

Notes:
Chapter 2

System Development Concept

Any system which is being developed should be discussed in detail from its requirement to maintenance stages. All these topics are generally covered in System Development Life Cycle (SDLC). Now since we are concentrating on the microcontroller programming, there are some basic low level information required to be known by a embedded software programmer.

2.1 Host and Target

**Host:** Generally Host is a system using which we develop end user application which is mostly application specific. So the host should have all the advanced capabilities like speed, storage, network, dev applications (like Hardware and software) etc. Again the sophistication of the Host could be based on the target requirement. So we can say that the host is the system which is used to develop the Target.

**Target:** An application which is designed for the specific requirement can be called Target. The term Target is used when the system is under development. As already mentioned, the Target systems are intended for specific requirement, it could have certain rigidity in its design such as, memories footprints, speed, size etc.,

2.2 Using a High Level Programming Language

Most microcontrollers today can be programmed using the C programming language.

Using the C language allows a microcontroller development engineer to be able to program the microcontroller at a higher level of abstraction, rather than using the assembly language of the particular microcontroller.

Some of the main advantages in using the C programming language are

- Using the C programming language means that you do not have to know too much about the register set or assembly language instruction of the target microcontroller.

- The C language has a standard (as in ANSI Standard C) and most C compilers follow this standard. This means that a C program written to execute on one microcontroller
should be able to execute on another microcontroller, even though the assembly instruction sets on each microcontroller might be very different (and incompatible).

- The C language statements are easy to learn, and once learned can be applied to many projects involving different computing platforms. Using C to program allows the development engineer to make use of C functions (sub programs) from different applications. This promotes function reuse and over a period of many development projects (using different microcontrollers), can result in considerable savings in development time. Also, by making use of the same basic functions in a number of projects, a general similarity in program interface can be achieved (for example, a common user interface could be employed).

- C programs can be written in a way that makes them very readable. This leads to easier understanding of complex programs and can reduce the time spent debugging programs.

Not all projects involving microcontrollers will be suitable for the support of C programming. So what situations would you be likely to use C, and what conditions would you be unlikely to use C?

**Hardware features that support the use of a High Level Language (such as C) are:**

- A large enough program memory with a linear memory space. Programs developed in C tend to be larger that those developed in assembly. This is always going to be the case since a C program must first be compiled into an object file and then linked to library routines to form the executable binary program that will run on the microcontroller.

- Adequate stack or temporary variable storage space. C programs tend to make a lot of use of functions, and these need to store return addresses, as well as passed variable parameters onto the microcontroller stack.

- A lot of C programs use pointers, and these involve indexed memory access. Some microcontrollers have very good indexed addressing modes, others don’t.

- Support for extended arithmetic. Not always needed, but C does support a wide range of variable types. Some C compilers (i.e., the PIC C compiler) do not support such a wide range of variable types.

**Reasons for NOT using HLLs such as C are:**

- Efficiency of resulting op-code. The C compiler is designed to compile C statements into the required instruction set of the microcontroller. This is always going to be a compromise since the C Compiler designer needs to generate general purpose executable code from the C statements. This is never going to be as efficient as a hand
crafted assembly language program developed by an experienced microcontroller development engineer. The result is that a C program will usually require more program memory and data memory, and execute slower than an equivalent program written by a good assembler programmer.

- The architecture of many small (typically 8 bit) microcontrollers are not always well matched to the requirements of HLLs. This is because small microcontrollers usually have small amounts of program and data memory, and a limited instruction set. To some extent, some recent C compilers (I am thinking about the MicoChip PIC C compiler here) are designed to better fit the medium to larger architecture of the PIC, and be less efficient when used with the base line PICs.

- Availability of debugging at the C level. If you develop your program in C you want to debug it in C. Some C compilers do not support this, resulting in a program that has to be debugged in assembly. This is difficult since the programmer may not recognise the assembly language instructions produced by the C compiler. Further, the C compiler optimisation goes beyond statement boundaries making it hard to follow the assembler language version of the code. It is therefore important that a C compiler is available that supports “source level debugging”. This means that the development engineer can single step, and use breakpoints in the C source code he has developed. Most C compiler now support “source level debugging” but you need to look out for those that don’t.

- Controlling software timing loops. Programmer to embed assembly language instructions into a C program. This would overcome the above timing problems but would make the program non portable since the embedded assembly language

### 2.3 Some important Definitions

**Cross Compiler:**
An application capable of creating executable code for a platform other than the one on which it is being run. Such a tool is handy when you want to compile code for a platform that you don’t have access to, or because it is inconvenient or impossible (at least in the low end embedded systems) to compile on that platform.

**Simulator:**
Suppose you have written assembly program in a file and corresponding executable file is ready. The simulator is the pc software which reads the instructions from the executable and ‘mimics’ the operation of the processor.

**Emulator:**
Emulator is a device which will have a controller and some Host application. This device can be plugged into the TARGET BOARD when you want to test the developed software in real time to check run time bugs. When not in use it can be unplugged. The Emulator will have a parallel or JTAG interface with the PC for downloading the executable file for execution.
Hence, whereas the Simulator is slow in execution, Emulator will be able to give real time verification of the developed code. Generally you will test your developed code on simulator first and then go for checking on emulator.

2.4 Understanding HEX Code

The Intel HEX file is an ASCII text file with lines of text that follow the Intel HEX file format. Each line in an Intel HEX file contains one HEX record. These records are made up of hexadecimal numbers that represent machine language code and/or constant data. Intel HEX files are often used to transfer the program and data that would be stored in a ROM or EPROM. Most EPROM programmers or emulators can use Intel HEX files.

Record Format

An Intel HEX file is composed of any number of HEX records. Each record is made up of five fields that are arranged in the following format:

:llaaaatt[dd . . . ]cc

Each group of letters corresponds to a different field, and each letter represents a single hexadecimal digit. Each field is composed of at least two hexadecimal digits-which make up a byte-as described below:

- **:** is the colon that starts every Intel HEX record.
- **ll** is the record-length field that represents the number of data bytes (dd) in the record.
- **aaaa** is the address field that represents the starting address for subsequent data in the record.
- **tt** is the field that represents the HEX record type, which may be one of the following:
  - **00** - data record
  - **01** - end-of-file record
  - **02** - extended segment address record
  - **04** - extended linear address record
- **dd** is a data field that represents one byte of data. A record may have multiple data bytes. The number of data bytes in the record must match the number specified by the ll field.
- **cc** is the checksum field that represents the checksum of the record. The checksum is calculated by summing the values of all hexadecimal digit pairs in the record modulo 256 and taking the two’s complement.
Data Records

The Intel HEX file is made up of any number of data records that are terminated with a carriage return and a linefeed. Data records appear as follows:

```
:10246200464C5549442050524F46494C4500464C33
```

This record is decoded as follows:

```

||| CC->Checksum
|||DD->Data
|||TT->Record Type
|||AAAA->Address
|LL->Record Length
:->Colon
```

where:

- **10** is the number of data bytes in the record
- **2462** is the address where the data are to be located in memory
- **00** is the record type 00 (a data record)
- **464C ... 464C** is the data
- **33** is the checksum of the record.

2.4.1 End-of-File (EOF) Records

An Intel HEX file must end with an end-of-file (EOF) record. This record must have the value **01** in the record type field. An EOF record always appears as follows:

```
:00000001FF
```

where:

- **00** is the number of data bytes in the record.
- **0000** is the address where the data are to be located in memory. The address in end-of-file records is meaningless and is ignored. An address of 0000h is typical.
- **01** is the record type 01 (an end-of-file record).
- **FF** is the checksum of the record and is calculated as **01h + NOT(00h + 00h + 00h + 01h).**
2.4.2 Example Intel HEX File

Following is an example of a complete Intel HEX

:10001300AC12AD13AE10AF1112002F8E0E8F0F2244
:10000300E50B250DF509E50A350CF5081200132259
:03000000020023D8
:0C002300787FE4F6D8FD7581130200031D
:10002F00EFF8DF0A4FFEDC5F0CEA42EFECC88F016
:04003F00A42EFE22CB
:00000001FF

2.5 Tools

An embedded software engineer must be familiar with the tools used in system development. In order to have kick start you should be knowing about the Cross Compiler, Target Code Flasher (Programmer both Software and Hardware application).

Notes:

2.6 Target Understanding

It is obvious that it would very difficult or say impossible to give life to the target without its understanding. As embedded engineers it becomes necessary to get the target level information for the documentation or from the system hardware designers. So some the detail which could help you in bringing up the target could be like,

- Target Architecture - All block level informations
- Target Board - Component placement level informations
- Target Schematic - Interfacing level informations
- Target Manual - Informations on how to handle the target like powering up, host target interconnectivity and importantly erratas if any
- Target Controller - Pin multiplexing level informations
Chapter 3

Target Understanding

As we mentioned in the last section of the previous chapter, it is recommended to an embedded engineers to know certain informations of the target on which they are working on. This chapter briefs about some important aspect of the Target.

3.1 EmxPICM02 Development Board

EmxPICM02 Development board is one of the best board you can find in market which might help you realize your ideas. It is obvious that every target board will be based on a specific controller, EmxPICM02 Development board’s controllers selection is based on the certain points like ease of availability, cost and online support, so that the users can carry forward the work with less hindrance.
3.2 EmxPICM02 Architecture

3.3 EmxPICM02 Board Features

- Based on microchip PIC16F887 microcontroller
- LED array
- SSD Display
- 16x2 character LCD
- 128x64 graphic LCD
- 3x3 matrix keypad
- Digital Keypad
- Analog Keypad
- RS232 and RS458 serial communication
• I2C enabled DS1338 RTC with battery backup
• SPI
• LIN
• LM35 Temperature sensor
• Dedicated Headers for connecting Potentiometers.
• Dedicated Header for PWM interfacing
• Dedicated Header Comparator Module
• Dedicated Header CCP Module
• Compatible with any 40-pin variant of PIC microcontroller.
• Option of exploring Internal Oscillator Feature
• USB interface (With supported Chips)
• ICSP Programmable
• Port Extensions

For more information on the Hardware refer the User Manual

3.4 EmxPICM02 - Controller: PIC16F887

PIC stands for Peripheral Interface Controller given by Microchip Technology to identify its single-chip microcontrollers. These devices have been very successful in 8-bit microcontrollers. The main reason is that Microchip Technology has continuously upgraded the device architecture and added needed peripherals to the microcontroller to suit customers’ requirements.

The architectures of various PIC microcontrollers can be divided as follows.

**Low end PIC Architectures:**

Microchip PIC microcontrollers are available in various types. When PIC microcontroller MCU was first available from General Instruments in early 1980’s, the microcontroller consisted of a simple processor executing 12-bit wide instructions with basic I/O functions. These devices are known as low-end architectures. They have limited program memory and are meant for applications requiring simple interface functions and small program & data memories.

Some of the low-end device numbers are,

• 12C5XX
• 16C5X
Mid range PIC Architectures:
Mid range PIC architectures are built by upgrading low-end architectures with more number of peripherals, more number of registers and more data/program memory. Some of the mid-range devices are,

- 16C6X
- 16C7X
- 16F87X

Program memory type is indicated by an alphabet.
C = EPROM
F = Flash
RC = Mask ROM

Popularity of the PIC microcontrollers is due to the following factors.

- Speed: Harvard Architecture, RISC architecture, 1 instruction cycle = 4 clock cycles.
- Instruction set simplicity: The instruction set consists of just 35 instructions (as opposed to 111 instructions for 8051).
- Power-on-reset and brown-out reset. Brown-out-reset means when the power supply goes below a specified voltage (say 4V), it causes PIC to reset; hence malfunction is avoided. A watch dog timer (user programmable) resets the processor if the software/program ever malfunctions and deviates from its normal operation.
- PIC microcontroller has four optional clock sources.
  - Low power crystal
  - Mid range crystal
  - High rang crystal
  - RC oscillator (low cost)
- Programmable timers and on-chip ADC.
- Up to 12 independent interrupt sources.
- Powerful output pin control (25 mA (max.) current sourcing capability per pin.)
- EPROM/OTP/ROM/Flash memory option.
- I/O port expansion capability.
3.4.1 Architecture of PIC16F887

The basic architecture remains same for almost all pic microcontrollers. The variation comes while dealing with its

- General purpose I/O
- Program memory capacity
- Data memory capacity
- Periperal modules

Notes:
3.4.2 Memory Organization

3.4.2.1 Program Memory

Notes:
### 3.4.2.2 Data Memory and SFRs

<table>
<thead>
<tr>
<th>Address</th>
<th>SFR Name</th>
<th>Address</th>
<th>SFR Name</th>
<th>Address</th>
<th>SFR Name</th>
<th>Address</th>
<th>SFR Name</th>
<th>Address</th>
<th>SFR Name</th>
<th>Address</th>
<th>SFR Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>TMR0</td>
<td>0x01</td>
<td>OPTION_REG</td>
<td>0x02</td>
<td>PCL</td>
<td>0x03</td>
<td>STATUS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x04</td>
<td>FSR</td>
<td>0x05</td>
<td>PORTA</td>
<td>0x06</td>
<td>PORTB</td>
<td>0x07</td>
<td>PORTC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td>PORTD</td>
<td>0x09</td>
<td>PORTE</td>
<td>0x0A</td>
<td>PCLATH</td>
<td>0x0B</td>
<td>INTCN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0C</td>
<td>PIR1</td>
<td>0x0D</td>
<td>PIR2</td>
<td>0x0E</td>
<td>TMRI1</td>
<td>0x0F</td>
<td>TMRI1H</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td>T1CON</td>
<td>0x11</td>
<td>TMRI2</td>
<td>0x12</td>
<td>T2CON</td>
<td>0x13</td>
<td>SSPBUF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x14</td>
<td>SSPCON</td>
<td>0x15</td>
<td>CCP1L</td>
<td>0x16</td>
<td>CCP1H</td>
<td>0x17</td>
<td>CCP1CON</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x18</td>
<td>RCSTA</td>
<td>0x19</td>
<td>TXREG</td>
<td>0x1A</td>
<td>RXREG</td>
<td>0x1B</td>
<td>CCP2L</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1C</td>
<td>CCP2H</td>
<td>0x1D</td>
<td>CCP2CON</td>
<td>0x1E</td>
<td>ADRESH</td>
<td>0x1F</td>
<td>ADCON0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**

- Indirect address values are shown for each SFR.
- Each SFR is listed with its corresponding address.
- The table includes a range of addresses for each SFR, indicating its memory allocation.

*Emertxe Information Technologies Pvt Ltd. Copyright 2018*
3.4.3 Pin Diagram of 16F887

Notes:
3.4.4 PIC Microcontroller Clock

Most of the PIC microcontrollers can operate up to 20MHz. One instructions cycle (machine cycle) consists of four clock cycles. Instructions that do not require modification of program counter content get executed in one instruction cycle.

Notes:
3.4.5 I/O Port Configuration

PIC16F887 has five I/O ports. PortA, Port-B, Port-C and Port-D have 8 pins each. Port-E have 4 pins respectively. Each port has bidirectional digital I/O capability. In addition, these I/O ports are multiplexed with alternate functions for the peripheral devices on the microcontroller. In general, when a peripheral is enabled, that pin may not be used as a general purpose I/O pin. Each port latch has a corresponding TRIS (Tri-state Enable) register for configuring the port either as an input or as an output. The port pins are designated by the alphabet R, followed by the respective port (viz. A, B, C, D or E) and the pin number. For example, Port-A pins are named as RA0, RA1, etc.

Notes:
Chapter 4

Work Environment

4.1 Work Space

Notes:
4.2 Code Structure

Notes:
4.3 Cross Compilation and Hex Dump Creation

Notes:
Chapter 5

EmxPICM02 Interfaces

5.1 LED

Light Emitting Diodes are one of the oldest and simplest form feedback device. Using these components designers can represent different state of the system.

Controlling the LED on the target via software is one of the easiest. If you have know the concept of forward biasing of diode it becomes simple to decide the logic level to be put on the controller pin. Refer the below images for better understanding.

Diode Forward Biasing

LED Forward Biasing

Sink Control

Source Control
Notes:

Code Template:
Fill the below code template to glow the LED array.

```c
#define LED_PORT

static void init_config(void)
{
}

void main(void)
{
    init_config();

    while (1)
    {
        LED_PORT =
    }
}
```

Code Template:
Fill the below code template to glow LED1.

```c
#define LED1

static void init_config(void)
{
}

void main(void)
{
    init_config();

    while (1)
    {
        LED1 =
    }
}
```
5.2 Digital Keypad

The name Digital keypad is since the nature of signal the controller would detect is either 0 or 1. These keypads mostly based on tactile switches. The property of the tactile switches are contact on press by design. So the interface lines should have a default value on the input line which could be either 1 or 0 while the switch is open, based on the design.

Notes:

5.2.1 Possible Interfaces

Pullup Configuration:

![Pullup Configuration Diagram]

Notes:

Pulldown Configuration:

![Pulldown Configuration Diagram]

Notes:
5.2.2 Methods of Reading Tactile keys

Notes:

5.2.3 Bouncing Effect

Notes:
Code Template:
Fill the below code template to glow the LED on press of Switch 1.

```c
#define SWITCH1
#define LED1
#define ON
#define OFF

void glow_on_press(void)
{
    if (SWITCH1)
    {
        LED1 = ON;
    }
    else
    {
        LED1 = OFF;
    }
}

void init_digital_keypad(void)
{
}

static void init_config(void)
{
    init_digital_keypad();
}

void main(void)
{
    init_config();

    while (1)
    {
        glow_on_press();
    }
}
5.3 External Interrupts

Typically a microcontroller will have at least one external pin in its architecture. These pins can be used to handle some higher priority signals which generally should not be missed. For example we can use this line as ZCD input for power failure detection.

Notes:

Code Template:
Fill the below code template to capture the falling edge on the external interrupt line. Toggle a LED on every interrupt generated.

```c
#define LED1

void interrupt isr(void)
{

}

void init_external_interrupt(void)
{

}
```
static void init_config(void)
{
    init_external_interrupt();
}

void main(void)
{
    init_config();
    while (1)
    {
    
    }
}
5.4 Timers and Counters

Timers are of basic block of a microcontroller, which can be used to perform timed synchronous activities. The beauty of the timer peripheral is that its clocking is not shared with instruction clock which makes it an independent block. When we deal with timers there are certain things to be kept in mind like

- Resolution - Depends on the register width (like 8, 16, 32 bits)
- Tick - A count increment due to clocking. This can be Up count or Down count depending on the timer architecture
- Quantum - The time taken to produce one tick. This might also depends on the timer settings apart from normal clock.
- Scaling - A measure to control the processing of ISR. There can be two types scaling called pre scaling and post scaling
- Mode - To choose the type of operation, Again this depends on the controller architecture. Some possible modes are
  - Counters
  - PWM
  - Pulse Measurements etc.

The timer can be configured as counter. The timer register will be incremented on the input pulse received at timer input pin.

Notes:
Code Template:
Fill the below code template to toggle a LED every one second.

```c
#define LED1

void interrupt isr(void)
{
}

void init_timer0(void)
{
}

static void init_config(void)
{
    init_timer0();
}

void main(void)
{
    init_config();
    while (1)
    {
    ;
    }
}
```
5.5 Clock I/O

An Embedded design some time have to accepts clocking from or provide clock to external devices. As an example the controller can be configured to accept the clock from the Real time Watch Crystals to calculate real timing. On the same way the controller can generate some clock pulses to drive some external chips like shift registers, Audio ICs, Memories etc.

Notes:

5.5.1 External Clocking on pins

Code Template:
Fill the below code template to generate clock output using for loop delay. Feed the same clock output to the external interrupt as clock input to Toggle the LED on every raising edge of clock.

#define CLOCK_OUT

void interrupt isr(void)
{

}

void init_external_interrupt(void) {

}

static void init_config(void) {

    init_external_interrupt();

}

void main(void) {
    init_config();

    while (1)
    {

    }
}
5.5.2 Pulse Width Modulation - PWM

The PWM is a clocking method wherein the width of the clock levels can be varied for a fixed period. The level control can be called as duty cycle, in which active signal is the percentage of period. A period is nothing but the total of ON time and OFF time. This can be represented using a formula

$$Duty\ Cycle = \frac{ON\ Time}{Period} \times 100\% \quad (5.1)$$

Some example waveforms are given below

Notes:
5.6 Analog Inputs

Most of the real time signals are linear in nature. If the amplitude these signal has to be detected in the controller, ADC are used. Most commonly used detection method is SAR (Successive Approximation Method).

In SAR the linear signal is converted to discrete digital representation. Once the conversion is done the value would be placed in a n bit register. The value n is nothing but the resolution of ADC. A 10 bit ADC will have 10 bit register carrying the converted value. The ADC output register starts incrementing every step change in the input voltage. Thus the step size can be calculated using input power and resolution of the ADC register. For example a system with 5V and 10 bit ADC would have the following step size

$$stepsize = \frac{VDD}{ADC \text{Resolution}} = \frac{5V}{1024} = 0.0048mV$$  \hspace{1cm} (5.2)

The ADC input bit is normally called as channel. The number of channels available depends on the chosen controller.

Notes:
Code Template:
Fill the below below code template to glow the one LED once the ADC channel value exceeds its half limit.

```c
#include <stdio.h>

#define CHANNEL
#define LED1
#define ON
#define OFF

void glow_led(unsigned short adc_reg_val)
{
    if (adc_reg_val > 512)
    {
        LED1 = ON;
    }
    else
    {
        LED1 = OFF;
    }
}

unsigned short read_adc(unsigned char channe)
{
}

void init_adc(void)
{
}
```
static void init_config(void)
{
    init_adc();
}

void main(void)
{
    unsigned short adc_reg_val;
    init_config();
    while (1)
    {
        adc_reg_val = read_adc(CHANNEL);
        glow_led(adc_reg_val);
    }
}
5.7 SSD

A seven-segment display, or seven-segment indicator, is a form of electronic display device for displaying decimal numerals that is an alternative to the more complex dot-matrix displays. Seven-segment displays are widely used in digital clocks, electronic meters, and other electronic devices for displaying numerical information.

SSD Segment Map:

Common Cathode design:

Common anode design:

Controlling the SSDs is as simple as the LED arrays. The designer has to create the digit map as required. Sometimes the Target design has to be understood to write a SSD driver. In case of EmxPICM02 Board the SSD are common cathode and are connected to
the shift register and the control line are connected to the Controller pins. So lets have the
some idea about the glue logic in SSD design

Serial-in, parallel-out shift register
A SIPO shift register has a data pin which is used to store the data fed. The max data you
can store without over writing depends on the IC make. EmxPICM02 uses CD4094, 8 bit
shift register. The data would be pushed in the internal register with the help of clock pin.
The enable pin would expose output register contents to output pins of the shift register.
So after completion of 8 clocks the user has to control the enable pin to get the output.
Please refer the CD4094 datasheet to refer its internal block diagram.

Notes:

SSD Circuit on EmxPICM02:

Note that the SSD control is a Transistor driver circuit.
Code Template:
Fill the Macro values for common cathode configuration. Use the below table for creation.

<table>
<thead>
<tr>
<th>Macros</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>dp</th>
</tr>
</thead>
<tbody>
<tr>
<td>ONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TWO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>THREE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FOUR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEVEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EIGHT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NINE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MINUS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLANK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MINUS_ONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#define ONE
#define TWO
#define THREE
#define FOUR
#define FIVE
#define SIX
#define SEVEN
#define EIGHT
#define NINE
#define DOT
#define MINUS
#define BLANK
#define MINUS_ONE
Code Template:
Fill the below code template to Display 0 on SSD1.

```c
#define SR_PORT
#define SSD_CONTROL_PORT

void display(unsigned char value)
{

}

void init_ssd_control(void)
{

}

static void init_config(void)
{
    init_ssd_control();
}

void main(void)
{
    init_config();

    while (1)
    {
        display(0);
    }
}
```
5.8 CLCD

If your design requires a simple alpha numeric display then the best choice would be Character LCD. These devices are very commonly used in low end embedded systems like telephones, cash counters, calculators etc. The most commonly used CLCDs are based on Hitachi’s HD44780 controller or other, which are compatible with HD44580. We can get easily get 1 Line, 2 Line or 4 Line CLCDs in market.

Below are some basic and important informations regarding the CLCD.

Pin Information:

<table>
<thead>
<tr>
<th>Pin No.</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>VSS</td>
<td>Power supply (GND)</td>
</tr>
<tr>
<td>2</td>
<td>VCC</td>
<td>Power supply (+5V)</td>
</tr>
<tr>
<td>3</td>
<td>VEE</td>
<td>Contrast adjust</td>
</tr>
</tbody>
</table>
| 4       | RS   | 0 = Command Mode  
          |      | 1 = Data Mode    |
| 5       | R/W  | 0 = Write to CLCD module  
          |      | 1 = Read from CLCD module |
| 6       | EN   | Enable signal |
| 7       | D0   | Data bus line 0 (LSB) |
| 8       | D1   | Data bus line 1 |
| 9       | D2   | Data bus line 2 |
| 10      | D3   | Data bus line 3 |
| 11      | D4   | Data bus line 4 |
| 12      | D5   | Data bus line 5 |
| 13      | D6   | Data bus line 6 |
| 14      | D7   | Data bus line 7 (MSB) |
| 15      | LED+ | Backlight Positive |
| 16      | LED- | Backlight Negative |

Frequently used Commands:

<table>
<thead>
<tr>
<th>No.</th>
<th>Instruction</th>
<th>Hex</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Function Set: 8-bit, 1 Line, 5x7 Dots</td>
<td>0x30</td>
<td>48</td>
</tr>
<tr>
<td>2</td>
<td>Function Set: 8-bit, 2 Line, 5x7 Dots</td>
<td>0x38</td>
<td>56</td>
</tr>
<tr>
<td>3</td>
<td>Function Set: 4-bit, 1 Line, 5x7 Dots</td>
<td>0x20</td>
<td>32</td>
</tr>
<tr>
<td>4</td>
<td>Function Set: 4-bit, 2 Line, 5x7 Dots</td>
<td>0x28</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>Entry Mode</td>
<td>0x06</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>Display off Cursor off (clearing display without clearing DDRAM content)</td>
<td>0x08</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>Display on Cursor on</td>
<td>0x0E</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>Display on Cursor off</td>
<td>0x0C</td>
<td>12</td>
</tr>
</tbody>
</table>
9 Display on Cursor blinking
10 Shift entire display left
11 Shift entire display right
12 Move cursor left by one character
13 Move cursor right by one character
14 Clear Display (also clear DDRAM content)
15 Set DDRAM address or cursor position on display
16 Set CGRAM address or set pointer to CGRAM location

CLCD Circuit on EmxPICM02:

Notes:
Code Template:
Fill the below Macros

#define CLCD_PORT
#define CLCD_EN
#define CLCD_RS
#define CLCD_RW

#define LINE1(x)
#define LINE2(x)

#define STABILIZE_LCD
#define CURSOR_HOME
#define DISP_ON_AND_CURSOR_OFF
#define CLEAR_DISP_SCREEN

#define INSTRUCTION_COMMAND
#define DATA_COMMAND

Code Template:
Fill the below code template to
1) print H on first line first position
2) print Hello on second line fifth Position

void write(
{
}

void clcd_print(
{
}
void clcd_putch(const unsigned char data, unsigned char addr)
{
    write(addr, INSTRUCTION_COMMAND);
    write(data, DATA_COMMAND);
}

void init_clcd(void)
{

    RW = LO;

    STABILIZE_LCD;
    CURSOR_HOME;
    DISP_ON_AND_CURSOR_OFF;
    CLEAR_DISP_SCREEN;
}

static void init_config(void)
{
    init_clcd();
}

void main(void)
{
    init_config();

    while (1)
    {
        clcd_putch('H', LINE1(0));
        clcd_print("Hello", LINE2(4));
    }
}
5.9 Matrix Keypad

The matrix keypad as the name implies is based on the concept of rows and columns. The designer can interface more number of inputs with less inputs lines. The below image gives an example of 2x2 matrix keypad

Notes:
Code Template:
Fill the below code template to glow the one dedicated LED on press of every key press of the 2x2 matrix keypad.

```c
#define MATRIX_KEYPAD_PORT
#define NO_OF_ROWS
#define NO_OF_COLS
#define LED1
#define LED2
#define LED3
#define LED4
#define SWITCH1
#define SWITCH2
#define SWITCH3
#define SWITCH4

void glow_on_press(unsigned char key) {
    switch (key) {
    case SWITCH1:
        LED1 = ON;
        break;
    case SWITCH2:
        LED2 = ON;
        break;
    case SWITCH3:
        LED3 = ON;
        break;
    case SWITCH4:
        LED4 = ON;
        break;
    default:
        LED1 = OFF;
        LED2 = OFF;
        LED3 = OFF;
        LED4 = OFF;
    }
}
```
unsigned char scan_keypad(void)
{
}

t void init_matrix_keypad(void)
{
}

t static void init_config(void)
{
    init_matrix_keypad();
}

t void main(void)
{
    unsigned char key;
    init_config();
    
    while (1)
    {
        key = scan_keypad();
        glow_on_press(key);
    
    }
}
5.10 Analog Keypad

The analog keypad could be used if have tight constraint on the no of I/O on the selected controller. The basic idea is have different signal level on every key press. A single analog port would be used to interface more than one tactile switch.

![Analog Keypad Diagram]

Notes:

Code Template:
Fill the below code template to glow the one LED once the ADC channel value exceeds its half limit.

```c
#define CHANNEL
#define LED1
#define LED2
#define LED3
#define LED4
#define ON
#define OFF
#define SWITCH1
#define SWITCH2
#define SWITCH3
#define SWITCH4
```
void glow_on_press(unsigned char key)
{
    switch (key)
    {
    case SWITCH1:
        LED1 = ON;
        break;
    case SWITCH2:
        LED2 = ON;
        break;
    case SWITCH3:
        LED3 = ON;
        break;
    case SWITCH4:
        LED4 = ON;
        break;
    default:
        LED1 = OFF;
        LED2 = OFF;
        LED3 = OFF;
        LED4 = OFF;
    }
}

unsigned char detect_key(unsigned short adc_reg_val)
{
}

void init_adc(void)
{
}

static void init_config(void)
{
    init_adc();
}

void main(void)
{
    unsigned short adc_reg_val;
    unsigned char key;
    init_config();
    while (1)
    {
        adc_reg_val = read_adc(CHANNEL);
        key = detect_key(adc_reg_val);
        glow_led(key);
    }
}
5.11 Data Storage

This topic deal with the persistence of the data on the system reset. Some application have the dependency of remembering the previous state of the system upon reset, like mode of operation, operation count, some control informations etc. leading to have design some memory interface. We have many choice of memories like EEPROMs, Flash. The decision would be based on the amount of the data to be preserved. For smaller amount of data say some hundreds of Kilobytes, there are many controllers which support internal data storage. It is just the matter of exploring these function in our selected architecture.

Notes:

Code Template:
Fill the below code template remember the state of LED on power failure. The LED would be controlled via a Switch.

```c
#define LED1
#define ON
#define OFF
#define SWITCH1

void control_led(unsigned char key)
{
}
```
static void init_config(void)
{

}

void main(void)
{
    init_config();

    while (1)
    {
        control_led(key);
    }
}
Chapter 6

Communication Protocols

There is no doubt that the communication play the major role in our life, like say, we are compiling this booklet and you read it to gain some understanding about this subject. So in simple terms communication can be defined in multiple ways like

- Exchange of information between one or more entities
- Expression of your requirements in order to proceed further on your task etc.,

And the way how we convey these messages could be by various methods like action, verbal, written, sign and many more. Today we have developed the technology to such an extent that, the system we designed started having inter dependencies within each other to work efficiently. When we say this, there exist an importance of common understanding between the devices we develop, leading to follow certain protocols in order to have effective and seamless communication.

There are ample of protocols available today in market, which makes the designers task of selection bit difficult. The selection could be based in the application requirements, like effective transmission, speed, length, reliability etc.

6.1 Modes of Communication

Notes:
6.2 Types of Communication

Notes:

6.3 UART

The UART is one of the oldest communication protocol we can now. Unfortunately this protocol is obsolete in the user space application, but still widely used in the industrial applications. Its one of the simplest protocol with the following possible frame format.

<table>
<thead>
<tr>
<th>FIELD NAME</th>
<th>BIT POSITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>1</td>
</tr>
<tr>
<td>DATA : (5 - 8)</td>
<td>2, 3, 4, 5, 6, 7, 8</td>
</tr>
<tr>
<td>PARITY</td>
<td>10</td>
</tr>
<tr>
<td>STOP</td>
<td>11, 12</td>
</tr>
</tbody>
</table>

Bit Details:

- The START bit is used to invoke an interrupt on the receiver side
- There can be 5 to 8 bits of DATA depending on the requirement
- The PARITY bit is used to validate the data transmitted. If the data stream has even number of 1’s then it will be even parity else its treated as odd parity
- The stop bit is used to denote the end of transmission. There can be 2 two stop bits if required.

Notes:
6.4 SPI

SPI is abbreviated as Serial Peripheral Interface, which is synchronous, full duplex protocol. The design intention of the this protocol is to have intra board single master and multiple slave communication. The below diagram show a typical master slave interface in SPI.

The interconnection between two devices happen in a circular ring connection. When there is a data transmission the data between both the devices gets exchanged. The result would be as shown in the below diagram:

Notes:
Device Interfacing Options:
Single Master and Single Slave:

Notes:

Single Master and Three Slave:

Notes:
Single Master and Three Daisy Chained Slave:

Notes:
6.5 I²C

I²C stands for Inter-Integrated circuit. I²C communication is a two wire bi-directional interface for connecting one or more master controller with one or more slave devices, such as an EEPROM, ADC, RAM, LCD display, DAC, etc. I²C interface requires two open drain I/O pins, viz. SDA (Serial Data) and SCL (Serial Clock).

The reason for open drain connection is that the data transfer is bi-directional and any of the devices connected to the I²C bus can drive the data line (SDA). The serial clock line (SCL) is usually driven by the master. Since SDA and SCL pins are open drain pins, external pull-up resistances are required for operation of I²C bus.

A typical I²C bus showing the connection of multi-master and multi-slave configuration is shown in the following figure.

From the above example let us say that the Microcontroller A (let’s call it as Master) wants to communicate with RTC (Slave), the following points would help you to understand the communication in simple terms:

- The Master would wait for bus idle condition. The bus idle condition is that no device is communicating with other at a given instance of time. So since the I²C bus is pulled up, the bus idle condition will be High.
- The Master issues the START condition so that every slave on the bus becomes active for reception. This action by the master make the bus non idle.
- The Master would issue the Slave address on the bus, 1 bit per clock pulse. Every slave on the I²C bus should have a unique address.
- All slaves will receive the broadcasted address and compare with itself
- If any slave matches the address would acknowledge the Master by pulling the SDA line low. This would happen on the 9th clock pulse.
- The communication can proceed further after Master receives the acknowledgement.
Once the communication is over between two devices the master issues a STOP condition making the bus idle again.

Notes:

I²C Timing Diagram

- SDA line transmits / receives data bits. MSB is sent first

- Data in SDA line is stable during clock (SCL) high. A new bit is initiated at the negative clock transition after a specified hold time.

- Serial clock (SCL) is driven by the master.

- Data is can be changed/written on the low level of the SCL

- Data is can be read on the high level of the SCL

- An acknowledgement bit (0) is driven by the receiver after the end of reception. If the receiver does not acknowledge, SDA line remains high (1).
I²C bus transfer consists of a number of byte transfers within a START condition and either another START condition or a STOP condition.

During the idle state when no data transfer is taking place, both SDA and SCL lines are released by all the devices and remains high. When a master wants to initiate a data transfer, it pulls SDA low followed by SCL being pulled low. This is called START condition. Similarly, when the processor wants to terminate the data transfer it first releases SCL (SCL becomes high) and then SDA. This is called a STOP condition. START (S) and STOP (P) conditions are shown in the above diagram.

START and STOP conditions are unique and they never happen within a data transfer.

Notes:

Data Communication Protocol:

In I²C communication both 7-bit and 10-bit slave addressing are possible. In 7-bit addressing mode 128 slaves (Theoretically) can be interfaced with a single master. Similarly, in 10-bit addressing mode, 1024 slaves (Theoretically) can be interfaced with the master. We will discuss here 7-bit addressing mode only. 10-bit addressing mode is similar to 7-bit addressing except from the fact that the number of address bits is more.

Following a START condition, the master sends a 7-bit address of the slave on SDA line. The MSB is sent first. After sending 7-bit address of the slave peripheral, a R/W (8th bit) bit is sent by the master. If R/W bit is '0', the following byte (after the acknowledgement bit) is written by the master to the addressed slave peripheral. If R/W is 1, the following byte (after the acknowledgement bit) has to be read from the slave by the master. This is represented in the below figure.
After sending the 7-bit address of the slave, the master sends the address (usually 8 bit) of the internal register of the slave where from the data has to be read or written to. The subsequent access is automatically directed to the next address of the internal register. The following diagrams give the general format to write and read from several peripheral internal registers.

R/W (Read/Write) bit indicates whether the data is to be written by the master or read by the master. If R/W is 1, the subsequent data are to be read by the master. If R/W = 0, the subsequent data are to be written by the master to the addressed slave. It has to be noted that the slave address is sent first, following a START condition.

The addressed slave responds by acknowledging and gets ready for data transfer. If data has to be read from a specific address of the slave device, the master sends the 7-bit address of the slave first following a START condition. R/W bit is sent as 'low'. The addressed slave acknowledges by pulling the ACK line low. The master then sends the 8-bit internal address of the slave from which data has to be read. The slave acknowledges. Since R/W was initially 0, the master is in the write mode.

To change this to read mode, the START condition is again generated followed by 7-bit address of the slave with R/W = 1. The slave acknowledges. The slave then sends
data from previously specified internal address to the master. The master acknowledges by pulling ACK bit low. The data transfer stops when the master does not acknowledge the data reception and a 'stop' condition is generated.

Notes:
Chapter 7

Assignments

Assignment 1:

| Implement a pattern generator on LEDs |
| Complexity | Easy |
| Prerequisites | Knowledge of LEDs, I/O Port Configurations |
| Requirement | Write a program to glow the LEDs as if a Train is coming out of a tunnel and entering again into the tunnel, in both directions i.e. left to right and right to left alternately |
| Inputs | Software Control |
| Outputs | LEDs |

Assignment 2:

| Implement a pattern generator on LEDs with User control |
| Complexity | Easy |
| Prerequisites | Knowledge of LEDs, Tactile Switches, I/O Port Configurations |
| Requirement | Write a program to glow the LEDs as if a Train is coming out of a tunnel and entering again into the tunnel. The direction of the flow should be changed by a key press |
| Inputs | Switch 1 for direction control |
| Outputs | LEDs |
### Assignment 3:
Implement a multiple pattern generator on LEDs

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Easy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of LEDs, Tactile Switches, I/O Port Configurations, PWM</td>
</tr>
<tr>
<td>Requirement</td>
<td>For every switch delicate a LEDs glow pattern. The pattern should change on key press</td>
</tr>
<tr>
<td>Inputs</td>
<td>Digital Keypad</td>
</tr>
<tr>
<td>Outputs</td>
<td>LEDs</td>
</tr>
</tbody>
</table>

### Assignment 4:
Implement the time separator with Timer 0, Timer 1 and Timer 2

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Easy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of LEDs, I/O Port Configurations, Timers</td>
</tr>
<tr>
<td>Requirement</td>
<td>The system should generate the time separator of the digital clock with multiple timers. Every timer should have dedicated indicator (say 1 LED per timer)</td>
</tr>
<tr>
<td>Inputs</td>
<td>Timers</td>
</tr>
<tr>
<td>Outputs</td>
<td>LEDs</td>
</tr>
</tbody>
</table>

### Assignment 5:
Implement a LED dimmer (software based)

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Easy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of LEDs, Tactile Switches, I/O Port Configurations, PWM</td>
</tr>
<tr>
<td>Requirement</td>
<td>Using software PWM, implement a LED dimmer application. The brightness of the LED illumination should be based on the user input using the switches. You need to consider the program cycle as Period for this implementation.</td>
</tr>
</tbody>
</table>
| Inputs | Switch 1 for Increment  
           Switch 2 for Decrement |
| Outputs | LED4 |

### Assignment 6:
Timeout based LED dimmer (Timer Based)

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Easy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of LEDs, Tactile Switches, I/O Port Configurations, PWM, Timers, External Interrupt Configuration</td>
</tr>
<tr>
<td>Requirement</td>
<td>The LED should dim to a fixed brightness (say 25% duty cycle). Once the user presses a switch the LED should glow at 100% duty for 5 seconds and the time out to 25% duty back</td>
</tr>
</tbody>
</table>
| Inputs | Switch 1 as Interrupt  
           Timer for Dimming |
| Outputs | LED1 |
### Assignment 7:

**Implement a LED dimmer (Timer based)**

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Easy</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Prerequisites</strong></td>
<td>Knowledge of LEDs, Tactile Switches, I/O Port Configurations, PWM, Timers</td>
</tr>
<tr>
<td><strong>Requirement</strong></td>
<td>Using Timer based PWM implement a LED dimmer application. The brightness of the LED illumination should be based on the user input using the switches. You need to consider the program cycle as Period for this implementation.</td>
</tr>
</tbody>
</table>
| **Inputs** | Switch 1 for Increment  
Switch 2 for Decrement  
Timer for Dimming |
| **Outputs** | LED1 |

### Assignment 8:

**Implement a LED dimmer (Software Based)**

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Easy</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Prerequisites</strong></td>
<td>Knowledge of LEDs, ADC, Switches, I/O Port Configurations, PWM</td>
</tr>
<tr>
<td><strong>Requirement</strong></td>
<td>Using software PWM implement a LED dimmer application. The brightness of the LED illumination should be based on the user input tuning the potentiometer. You need to consider the program cycle as period for this implementation</td>
</tr>
<tr>
<td><strong>Inputs</strong></td>
<td>Potentiometer 1 as Knob</td>
</tr>
<tr>
<td><strong>Outputs</strong></td>
<td>LED1</td>
</tr>
</tbody>
</table>

### Assignment 9:

**Implement a LED dimmer (Timer Based)**

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Easy</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Prerequisites</strong></td>
<td>Knowledge of LEDs, ADC, Switches, I/O Port Configurations, PWM, Timers</td>
</tr>
<tr>
<td><strong>Requirement</strong></td>
<td>Using Timer based PWM implement a LED dimmer application. The brightness of the LED illumination should be based on the user input tuning the potentiometer. You need to use the system timer to generate the PWM</td>
</tr>
</tbody>
</table>
| **Inputs** | Potentiometer 1 as Knob  
Timer for Dimming |
| **Outputs** | LED1 |
### Assignment 10:
Implement a static clock with internal timer (Hours and Minutes)

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Easy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of SSDs, Shift Registers, I/O Port Configurations, Timers</td>
</tr>
<tr>
<td>Requirement</td>
<td>The clock should display Hours and Minutes fields. The decimal point of the hours field should blink every half second</td>
</tr>
<tr>
<td>Inputs</td>
<td>Timers</td>
</tr>
<tr>
<td>Outputs</td>
<td>SSDs</td>
</tr>
</tbody>
</table>

### Assignment 11:
Implement a 4 digit down counter with variable frequency

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Medium</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of SSDs, Shift Registers, I/O Port Configurations, Timers, ADC</td>
</tr>
<tr>
<td>Requirement</td>
<td>The system should count down from 9999 to 0 and then should stop. The count source would be the program cycle. The frequency of the counting would be varied using a potentiometer. If the potentiometer is tuned to max position the counting should be 10 times faster than the default. The result would be displayed on SSDs</td>
</tr>
<tr>
<td>Inputs</td>
<td>Program Cycles</td>
</tr>
<tr>
<td></td>
<td>Potentiometer 1 for frequency control</td>
</tr>
<tr>
<td>Outputs</td>
<td>SSDs</td>
</tr>
</tbody>
</table>

### Assignment 12:
Implement a dynamic clock with internal timer (Hours and Minutes)

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Medium</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of SSDs, Shift Registers, I/O Port Configurations, Timers, ADC</td>
</tr>
<tr>
<td>Requirement</td>
<td>The clock should display Hours and Minutes fields. The decimal point of the hours field should blink every half second. The user should be able to set the clock if required. Once the clock is set mode time fields should start blinking a the rate of 500 msecs indicating the edit mode. The user should be able to select the field using a switch</td>
</tr>
<tr>
<td>Inputs</td>
<td>Switch 1 Increment</td>
</tr>
<tr>
<td></td>
<td>Switch 2 Decrement</td>
</tr>
<tr>
<td></td>
<td>Switch 3 Choose Field</td>
</tr>
<tr>
<td></td>
<td>Switch 4 Set Clock</td>
</tr>
<tr>
<td></td>
<td>Timer</td>
</tr>
<tr>
<td>Outputs</td>
<td>SSDs</td>
</tr>
</tbody>
</table>
## Assignment 13:
Implement a Timer (Down Count) using matrix keypad

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Medium</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of SSDs, Shift Registers, I/O Port Configurations, Timers, ADC, Matrix Keypad Concepts</td>
</tr>
<tr>
<td>Requirement</td>
<td>The user should be able to preset the timer value from 99 min (Max) to 1 sec (Min) and the Timer should start down counting from set Time to 0. The user should be able set the preset with the help of switches</td>
</tr>
<tr>
<td>Inputs</td>
<td>Switch 1 Increment, Switch 2 Decrement, Switch 3 Choose Field, Switch 4 Start / Set Timer</td>
</tr>
<tr>
<td>Outputs</td>
<td>SSDs</td>
</tr>
</tbody>
</table>

## Assignment 14:
Implement a 4 digit key press counter

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Easy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of SSDs, Shift Registers, I/O Port Configurations, Timers, ADC, Matrix Keypad Concepts</td>
</tr>
<tr>
<td>Requirement</td>
<td>The no of times the key is pressed the system should keep a count of it. The user should be able to reset if require</td>
</tr>
<tr>
<td>Inputs</td>
<td>Switch 1 as count input, Switch 1 Long press to reset the count</td>
</tr>
<tr>
<td>Outputs</td>
<td>SSDs</td>
</tr>
</tbody>
</table>

## Assignment 15:
Implement a temperature sensor node

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Easy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of SSDs, Shift Registers, I/O Port Configurations, ADC, LM35 Temperature Sensor</td>
</tr>
<tr>
<td>Requirement</td>
<td>The SSDs should print the current room temperature</td>
</tr>
<tr>
<td>Inputs</td>
<td>LM35 Temperature Sensor</td>
</tr>
<tr>
<td>Outputs</td>
<td>SSDs</td>
</tr>
</tbody>
</table>
## Assignment 16:
**Implement a Voltage comparator**

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Easy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of I/O Port Configurations, ADC, CLCD</td>
</tr>
<tr>
<td>Requirement</td>
<td>Compare the voltages on channels RA2 &amp; RA3 and display on LCD whether the voltage on RA2 is less than/greater than/equal to the voltage on RA3</td>
</tr>
<tr>
<td>Inputs</td>
<td>Potentiometer 1 and Potentiometer 2</td>
</tr>
<tr>
<td>Outputs</td>
<td>CLCD</td>
</tr>
</tbody>
</table>

## Assignment 17:
**Implement a left scrolling number marquee**

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Easy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of SSDs, Shift Registers, I/O Port Configurations</td>
</tr>
<tr>
<td>Requirement</td>
<td>The system should scroll the static message embedded in code in left direction</td>
</tr>
<tr>
<td>Inputs</td>
<td>Static number message in code</td>
</tr>
<tr>
<td>Outputs</td>
<td>SSDs</td>
</tr>
</tbody>
</table>

## Assignment 18:
**Implement a scrolling number marquee with direction control**

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Easy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of SSDs, Shift Registers, I/O Port Configurations</td>
</tr>
<tr>
<td>Requirement</td>
<td>The system should scroll the static message embedded in code. The user should be able to control the display scroll direction</td>
</tr>
</tbody>
</table>
| Inputs | Static numbers in code  
Switch 1 to left rotation  
Switch 2 to right rotation  
Switch 3 to stop |
| Outputs | SSDs |

## Assignment 19:
**Implement a checksum validation**

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Easy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of I/O Port Configurations, Matrix Keypad</td>
</tr>
<tr>
<td>Requirement</td>
<td>Treat a switch on the matrix keypad as bit of byte. Every key should be indicated with a LED on LED array. On the press of the 9th switch check for odd and even no of switch(es) pressed. Result should be indicated on eight LED (Clear the previous outputs). ON for even and OFF for odd key(s) detections</td>
</tr>
<tr>
<td>Inputs</td>
<td>Matrix Keypad</td>
</tr>
<tr>
<td>Outputs</td>
<td>LEDs</td>
</tr>
</tbody>
</table>
## Assignment 20:
Implement a ZCD output with 1 msec ON time and 9 msecs OFF time

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Easy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of I/O Port Configurations, Timers, ZCD</td>
</tr>
<tr>
<td>Requirement</td>
<td>Your target should generate a typical ZCD signal with 1 msec ON pulse and 9 msecs of OFF pulse</td>
</tr>
<tr>
<td>Inputs</td>
<td>Timers</td>
</tr>
<tr>
<td>Outputs</td>
<td>RD0 Pin</td>
</tr>
</tbody>
</table>

## Assignment 21:
Implement a 10 digit up counter

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Easy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of I/O Port Configurations, CLCD</td>
</tr>
<tr>
<td>Requirement</td>
<td>The system should count up to 9999999999 and then should stop. The count source would be the program cycle. The result would be displayed on CLCD.</td>
</tr>
<tr>
<td>Inputs</td>
<td>Program Cycles</td>
</tr>
<tr>
<td>Outputs</td>
<td>CLCD</td>
</tr>
</tbody>
</table>

## Assignment 22:
Implement a 10 digit down counter

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Medium</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of I/O Port Configurations, CLCD</td>
</tr>
<tr>
<td>Requirement</td>
<td>The system should count down from 9999999999 to 0 and then should stop. The count source would be the program cycle. The result would be displayed on CLCD.</td>
</tr>
<tr>
<td>Inputs</td>
<td>Program Cycles</td>
</tr>
<tr>
<td>Outputs</td>
<td>CLCD</td>
</tr>
</tbody>
</table>

## Assignment 23:
Implement a 10 digit down counter with preset

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Medium</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of I/O Port Configurations, CLCD, Tactile Switch</td>
</tr>
<tr>
<td>Requirement</td>
<td>The user should be able to preset the count value from 9999999999 (Max) to 1 (Min) and the counter should start down counting to 0. The user should be able set the preset with the help of switches</td>
</tr>
<tr>
<td>Inputs</td>
<td>Program Cycles as event, Switch 1 Increment (roll back once max is reached), Switch 2 Choose Field, Switch 3 Start / Set counter</td>
</tr>
<tr>
<td>Outputs</td>
<td>CLCD</td>
</tr>
</tbody>
</table>
Assignment 24:

| Implement a Stop Watch with 5 laps |
| Complexity | Medium |
| Prerequisites | Knowledge of I/O Port Configurations, CLCD, Tactile Switch, Timers |
| Requirement | On press of the start key the timer should start incrementing. By using the lap key the user should be able to capture the lap timing. There should be an option of 5 lap timing which can be viewed using scroll keys. By pressing the start key again the timer should stop. The previous reading would get cleared (or reset) if the user presses the lap key while the timer is not running. |
| Inputs | Switch 1 to Start / Stop |
| | Switch 2 Lap |
| | Switch 3 Scroll Up |
| | Switch 4 Scroll Down |
| | Switch 5 Reset |
| | Timer |
| Outputs | CLCD |

Assignment 25:

| Implement a right scrolling message marquee |
| Complexity | Easy |
| Prerequisites | Knowledge of I/O Port Configurations, CLCD |
| Requirement | The system should scroll the static message embedded in code in right direction |
| Inputs | Static Message in code |
| Outputs | CLCD |

Assignment 26:

| Implement a dynamic scrolling number marquee with rotation control |
| Complexity | Hard |
| Prerequisites | Knowledge of I/O Port Configurations, CLCD, Tactile Switch |
| Requirement | The system should scroll the message entered by the user. There should be a provision for the user to enter the message using the digital keypad. The user should be able to control the display rotation direction |
| Inputs | User entered number |
| | Switch 1 up and left rotation |
| | Switch 2 down and right rotation |
| | Switch 3 to stop and start |
| | Switch 4 to enter message |
| | Switch 5 to accept the selected character |
| Outputs | CLCD |
### Assignment 27:

**Implement a temperature sensor node**

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Easy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of I/O Port Configurations, CLCD, ADC, LM35 Temperature Sensor</td>
</tr>
<tr>
<td>Requirement</td>
<td>The CLCD should print the current room temperature</td>
</tr>
<tr>
<td>Inputs</td>
<td>LM35 Temperature Sensor</td>
</tr>
<tr>
<td>Outputs</td>
<td>CLCD</td>
</tr>
</tbody>
</table>

### Assignment 28:

**Implement a 8 field password access to screen**

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Medium</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of I/O Port Configurations, LEDs, CLCD, Tactile Switch</td>
</tr>
<tr>
<td>Requirement</td>
<td>The user has to enter a key press pattern to unlock the ester egg. If the user succeeds in unlocking the screen, he will be able to read the message. The password should be displayed as * while entry. On failure attempt the LED3 will blink faster for every attempt. The max attempt would be 5 time on which the LED 3 will glow constantly</td>
</tr>
<tr>
<td>Inputs</td>
<td>Switch 1 and Switch 2 as access keys</td>
</tr>
<tr>
<td>Outputs</td>
<td>CLCD, LED3</td>
</tr>
</tbody>
</table>

### Assignment 29:

**Implement a Timer (Up Count) using Analog Keypad**

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Medium</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of SSDs, Shift Registers, I/O Port Configurations, Timers, ADC, Analog Keypad Concepts</td>
</tr>
<tr>
<td>Requirement</td>
<td>The user should be able to preset the timer value from 99 min (Max) to 1 sec (Min) and the Timer should start up counting from 0 to the set Time. The user should be able set the preset with the help of switches</td>
</tr>
<tr>
<td>Inputs</td>
<td>Switch 1 Increment, Switch 2 Decrement, Switch 3 Choose Field, Switch 4 Start / Set Timer, Timer</td>
</tr>
<tr>
<td>Outputs</td>
<td>SSDs</td>
</tr>
</tbody>
</table>
**Assignment 30:**
Implement a 4 field password access to screen

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Medium</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of I/O Port Configurations, CLCD, ADC, Analog Keypad Concepts</td>
</tr>
<tr>
<td>Requirement</td>
<td>The user has to enter to 4 key password to access the screen</td>
</tr>
<tr>
<td>Inputs</td>
<td>Analog Keypad</td>
</tr>
<tr>
<td>Outputs</td>
<td>CLCD</td>
</tr>
</tbody>
</table>

**Assignment 31:**
Implement storage on key press

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Easy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of I/O Port Configurations, SSDs, Shift Registers, EEPROM Concepts, Tactile Switch</td>
</tr>
<tr>
<td>Requirement</td>
<td>Save the count value on SSDs counter using a s/w. Begin the count from the value saved after reset or power ON</td>
</tr>
<tr>
<td>Inputs</td>
<td>Switch 2</td>
</tr>
<tr>
<td>Outputs</td>
<td>SSDs</td>
</tr>
</tbody>
</table>

**Assignment 32:**
Implement a 4 digit key press counter with persistence

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Easy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of I/O Port Configurations, SSDs, Shift Registers, EEPROM Concepts, Tactile Switch</td>
</tr>
<tr>
<td>Requirement</td>
<td>The no of times the key is pressed the system should keep a count of it. The user should be able to reset if required. The count should be saved on the internal EEPROM by a key press</td>
</tr>
</tbody>
</table>
| Inputs | Switch 1 as Count input  
Switch 1 Long press to reset the count  
Switch 2 to save the count |
| Outputs | SSDs |

**Assignment 33:**
Implement data storage on power failure using ZCD input

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Medium</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of I/O Port Configurations, CLCD, EEPROM Concepts, Tactile Switch, ZCD</td>
</tr>
<tr>
<td>Requirement</td>
<td>The output of the Assignment 10 would be used as the input to this requirement. You should save the data if you fail to detect the ON pulse. Assignment 18 can be extended for this Assignment.</td>
</tr>
<tr>
<td>Inputs</td>
<td>Switch 1 as Count input</td>
</tr>
<tr>
<td>Outputs</td>
<td>Data recovery on power restoration</td>
</tr>
</tbody>
</table>
### Assignment 34:

**Implement a 4 digit key press counter with persistence using external EEPROM**

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Hard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of I/O Port Configurations, SSDs, Shift Registers, EEPROM Concepts, SPI, SPI EEPROM, Tactile Switch</td>
</tr>
<tr>
<td>Requirement</td>
<td>The no of times the key is pressed the system should keep a count of it. The user should be able to reset if required. The count should be saved on the internal EEPROM by a key press</td>
</tr>
</tbody>
</table>
| Inputs: | Switch 1 as Count input  
Switch 1 Long press to reset the count  
Switch 2 to save the count |
| Outputs | SSDs |

### Assignment 35:

**Implement a LED dimmer (Timer based)**

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Hard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of LEDs, Tactile Switches, I/O Port Configurations, PWM, CLCD Concepts, Timer</td>
</tr>
<tr>
<td>Requirement</td>
<td>Using Timer based PWM implement a LED dimmer application. The brightness of the LED illumination should be based on the user input using the switches. The brightness level should be indicated on the CLCD</td>
</tr>
</tbody>
</table>
| Inputs | Switch 1 for Increment  
Switch 2 for Decrement  
Timer for dimming |
| Outputs | LED4  
CLCD |

### Assignment 36:

**Implement system sleep and wake up while Interrupt on change**

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Easy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of SSDs, Shift Registers, Tactile Switches, I/O Port Configurations, Interrupts, Sleep function</td>
</tr>
<tr>
<td>Requirement</td>
<td>The SSD would print a message (say 1234) as the system is active. If the system is inactive for 10 secs then the system should go to sleep mode. The system will wake up as soon as a key press is detected. Toggle LED8 every second</td>
</tr>
<tr>
<td>Inputs</td>
<td>Switch 1 as interrupt to wake</td>
</tr>
<tr>
<td>Outputs</td>
<td>SSDs</td>
</tr>
</tbody>
</table>
### Assignment 37:
<table>
<thead>
<tr>
<th>Complexity</th>
<th>Easy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of Tactile Switches, I/O Port Configurations, UART</td>
</tr>
<tr>
<td>Requirement</td>
<td>The pressed key should be informed on the serial application</td>
</tr>
<tr>
<td>Inputs</td>
<td>Digital Keypad</td>
</tr>
<tr>
<td>Outputs</td>
<td>UART (PC Minicom)</td>
</tr>
</tbody>
</table>

### Assignment 38:
<table>
<thead>
<tr>
<th>Complexity</th>
<th>Easy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of I/O Port Configurations, UART, ADC, Analog Keypad Concepts</td>
</tr>
<tr>
<td>Requirement</td>
<td>The pressed key should be informed on the serial application</td>
</tr>
<tr>
<td>Inputs</td>
<td>Analog Keypad</td>
</tr>
<tr>
<td>Outputs</td>
<td>UART (PC Minicom)</td>
</tr>
</tbody>
</table>

### Assignment 39:
<table>
<thead>
<tr>
<th>Complexity</th>
<th>Easy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of Matrix Keypad Concepts, I/O Port Configurations, UART</td>
</tr>
<tr>
<td>Requirement</td>
<td>The pressed key should be informed on the serial application</td>
</tr>
<tr>
<td>Inputs</td>
<td>Matrix Keypad</td>
</tr>
<tr>
<td>Outputs</td>
<td>UART (PC Minicom)</td>
</tr>
</tbody>
</table>

### Assignment 40:
<table>
<thead>
<tr>
<th>Complexity</th>
<th>Easy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of I/O Port Configurations, CLCD, UART</td>
</tr>
<tr>
<td>Requirement</td>
<td>The message typed on the serial application should be displayed on the CLCD</td>
</tr>
<tr>
<td>Inputs</td>
<td>UART</td>
</tr>
<tr>
<td>Outputs</td>
<td>CLCD</td>
</tr>
</tbody>
</table>
### Assignment 41:

**Implement a dynamic scrolling message marquee with rotation control**

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Medium</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of I/O Port Configurations, CLCD, UART, Digital Keypad Concepts</td>
</tr>
<tr>
<td>Requirement</td>
<td>The system should scroll the message entered by the user. There should be a provision for the user to enter the message via UART. The user should be able to control the display rotation direction</td>
</tr>
</tbody>
</table>
| Inputs: | User entered Message via UART  
Switch 1 to left rotation  
Switch 2 to right rotation  
Switch 3 to stop |
| Outputs: | CLCD |

### Assignment 42:

**Implement event counter with timer as counter**

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Hard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of I/O Port Configurations, UART, Timer as Counter, DS1338 RTC</td>
</tr>
<tr>
<td>Requirement</td>
<td>Using timer1 as counter, count the number of events occurred. The count should be transmitted via UART to PC</td>
</tr>
<tr>
<td>Inputs:</td>
<td>Events at RC0 from SQW/OUT pin from RTC</td>
</tr>
<tr>
<td>Outputs</td>
<td>UART</td>
</tr>
</tbody>
</table>

### Assignment 43:

**Print an image GLCD**

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Hard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of I/O Port Configurations, UART, GLCD concepts</td>
</tr>
<tr>
<td>Requirement</td>
<td>Print a image on GLCD</td>
</tr>
<tr>
<td>Inputs:</td>
<td>Image Embedded in Code</td>
</tr>
<tr>
<td>Outputs</td>
<td>GLCD</td>
</tr>
</tbody>
</table>

### Assignment 44:

**Implement a real time graph between voltage vs time**

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Hard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites</td>
<td>Knowledge of I/O Port Configurations, UART, GLCD concepts, Timer, ADC</td>
</tr>
<tr>
<td>Requirement</td>
<td>The real time voltage graph should be plotted with respect to time</td>
</tr>
</tbody>
</table>
| Inputs: | Potentiometer 1 as linear voltage  
Timer for timing |
<p>| Outputs | GLCD |</p>
<table>
<thead>
<tr>
<th>Assignment 45:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Implement a baud rate detector</td>
<td></td>
</tr>
<tr>
<td>Complexity</td>
<td>Hard</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>Knowledge of I/O Port Configurations, UART, CLCD concepts</td>
</tr>
<tr>
<td>Requirement</td>
<td>The system should be able to detect the baud rate set on the serial application and indicate on the CLCD</td>
</tr>
<tr>
<td>Inputs:</td>
<td>Serial Transmission</td>
</tr>
<tr>
<td>Outputs</td>
<td>CLCD</td>
</tr>
</tbody>
</table>
Appendix A

Assignment Guidelines

The following highlights common deficiencies which lead to loss of marks in Programming assignments. Review this sheet before turning in each assignment to make sure that it is complete in all respects.

A.1 Quality of the Source Code

A.1.1 Variable Names

- Use variable names with a clear meaning in the context of the program whenever possible.

A.1.2 Indentation and Format

- Include adequate white-space in the program to improve readability. Insert blank lines to group sections of code. Use indentation to improve readability of control flow. Avoid confusing use of opening/closing braces.

A.1.3 Internal Comments

- Main program comments should describe overall purpose of the program. You should have a comment at the beginning of each source file describing what that file contains/does. Function comments should describe their purpose and other pertinent information, if any.

- Compound statements (control flow) should be commented. Finally, see that commenting is not overdone and redundant.

A.1.4 Modularity in Design

- Avoid accomplishing too many tasks in one function; use a separate module (Split your code into multiple logical functions). Also, avoid too many lines of code in a single module; create more modules. Design should facilitate individual module testing.
Use automatic/local variables instead of external variables whenever possible. Use separate header files and implementation files for unrelated functions.

A.2 Program Performance

A.2.1 Correctness of Output

- Ensure that all outputs are correct. Incorrect outputs can lead to substantial loss in grade.

A.2.2 Ease of Use

- The program should facilitate repeated use when used interactively and should allow easy exit. Requests for interactive input from the user should be clear. Incorrect user inputs should be captured and explained. Outputs should be well-formatted.
Appendix B

Grading of Programming Assignments

- Total points per assignment = 10
- Points for timely/early submission = 1
- The source code is out of 3 points. The distribution of points is as follows:
  - (a) The existence of the code itself (1 pts)
  - (b) Proper indentation of the code and comments (1 pts)
  - (c) Proper naming of the functions, variables + Modularity + (1 pts)
- You get 4 points if the program does exactly what it is supposed to do.
- Two (2) points are reserved for the ease of use, the type of user interface, the ability to handle various user input errors, or any extra features that your program might have.