Embedded Controller(EC) Firmware Development on Upcoming Platform

Major Project Report

Submitted in fulfillment of the requirements for the degree of

Master of Technology
in
Electronics & Communication Engineering
(Embedded Systems)

By

Falak Mehta

(16MECE10)

Electronics & Communication Engineering Department
Institute of Technology
Nirma University
Ahmedabad-382 481
May 2018

Embedded Controller(EC) Firmware Development on Upcoming Platform

Major Project Report

Submitted in fulfillment of the requirements

for the degree of

Master of Technology

in

Electronics & Communication Engineering

By

Falak Mehta
(16MECE10)

Under the guidance of

External Project Guide:	Internal Project Guide:
Kunal Shah	**Dr. Dhaval Shah**
Firmware Engineer	Assistant Professor, EC Department,
Intel Technology Pvt. Ltd.,	Institute of Technology,
Bangalore.	Nirma University, Ahmedabad.

Electronics & Communication Engineering Department

Institute of Technology-Nirma University

Ahmedabad-382 481

May 2018

Declaration

This is to certify that,

a. The thesis comprises my original work towards the degree of Master of Technology in Embedded Systems at Nirma University and has not been submitted elsewhere for a degree.

b. Due acknowledgment has been made in the text to all other material used.

- Falak Mehta

16MECE10

Disclaimer

“The content of this paper does not represent the technology, opinions, beliefs, or positions of Intel Technology Pvt. Ltd., its employees, vendors, customers, or associates.”

Certificate

This is to certify that the Major Project entitled **"Embedded Controller(EC) Firmware Development on Upcoming Platform "** submitted by **Falak Mehta (16MECE10)**, towards the fulfillment of the requirements for the degree of Master of Technology in Embedded Systems, Nirma University, Ahmedabad is the record of work carried out by her under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination.The results embodied in this major project, to the best of our knowledge, haven't been submitted to any other university or institution for the award of any degree or diploma.

Date: Place: Ahmedabad

Dr. Dhaval Shah
Internal Guide

Dr. N.P. Gajjar
Program Coordinator

Dr. D. K. Kothari
Section Head, EC

Dr. Alka Mahajan
Director, IT

Certificate

This is to certify that the Major Project entitled **"Embedded Controller(EC) Firmware Development on Upcoming Platform"** submitted by **Falak Mehta (16MECE10)**, towards the fulfillment of the requirements for the degree of Master of Technology in Embedded Systems, Nirma University, Ahmedabad is the record of work carried out by her under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination.

Chandra Sekhar
Software Engineering Manager, CSS-EC
Intel Technology India Pvt. Ltd.
Bangalore

Acknowledgements

I would like to articulate my recognition and genuine thanks to **Dr. D. K. Kothari**, Head of Electronics and Communication Engineering Department, and **Dr. N. P. Gajjar**, PG Coordinator of M.Tech Embedded Systems program for permitting me to undertake this thesis work and for his guidelines during the review progress. I take this occasion to express my intense obligation and commendations to **Dr. Dhaval Shah**, guide of my major project for his ideal guidance, monitoring and continuous motivation throughout the course of this thesis.

I would take this moment to deep sense of honor to **Chandra Sekhar**, Software Engineering Manager, Intel Technology India Pvt. Ltd. for his constant supervision as well as for enabling valuable information about the project and advices, which helped me in completing this task through numerous stages. I would also praise **Kunal Shah**, my project mentor for always helping, giving me good suggestions, solving my confusions and guide me to fulfilling the different task of my project in a better way.

- Falak Mehta

16MECE10

Contents

Declaration iii

Disclaimer iv

Certificate v

Acknowledgements vii

Abstract xii

Abbreviation Notation and Nomenclature xiii

1 Introduction 1

1.1 Introduction 1

1.2 Current Scenario 2

1.3 Problem Statement 2

1.4 Objective 2

1.5 Timeline 3

2 Literature Survey 5

2.1 Overview 5

2.2 Embedded Controller Interface Description 6

2.3 Embedded Controller Register Descriptions 7

2.4 Embedded Controller Command Set 9

3 Hardware Design **11**
3.1 System Architecture 11
3.2 Block Diagram of Embedded Controller 12
3.3 Hard-Wired Functionality 13

4 Work done & Design flow **15**
4.1 GPIO(General Purpose I/O) 15
4.2 Timer 19
4.3 Watchdog Timer 21
4.4 Pulse Width Modulation(PWM) 23
4.5 Library File Generation 25
4.6 Analog to Digital Converter(ADC) 26
4.7 Interrupt 27
4.8 Make File 30
4.9 Board Support Package(BSP) 33
4.10 Enhanced Serial Peripheral Interface(eSPI) 33
4.11 SMBus Interface 39

5 Conclusion **47**
5.1 Conclusion 47

6 Future Scope **49**

References **51**

List of Figures

1.1 Timeline 3

2.1 Register detail[1] 8

2.2 EC Command Set[1] 9

3.1 System Architecture 11

3.2 Block Diagram of Embedded Controller 13

4.1 Pull-Up 16

4.2 Pull-Down 16

4.3 GPIO ConfigPin Flowchart 17

4.4 GPIO SetPin Flowchart 18

4.5 GPIO GetPin Flowchart 19

4.6 Timer Flowchart 20

4.7 Watchdog timer[3] 21

4.8 Watchdog timer flowchart 22

4.9 Duty Cycle 24

4.10 PWM Flowchart 25

4.11 PWM Timing diagram 26

4.12 ADC Flowchart 27

4.13 ADC serial logs 28

4.14 Interrupt Flowchart 29

4.15 Make File snapshot 31

4.16 Make File commands 31
4.17 Code Base snapshot 32
4.18 Master-Slave Interface 34
4.19 Peripheral packet format 35
4.20 OOB packet format 36
4.21 Virtual wire packet 37
4.22 Virtual wire command 37
4.23 Virtual wire Flowchart 38
4.24 SMBus 39
4.25 Quick command 41
4.26 Send Byte command 41
4.27 Receive Byte command 42
4.28 Write Byte/Word command 42
4.29 Read Byte/Word command 43
4.30 Block Write/Read command 43
4.31 SMBus Flowchart 44

Abstract

The embedded controller (EC) is a fundamental segment in ultra-mobile, modern mobile and implanted PC frameworks. Embedded controllers are frequently used in I/O systems, low power designs, different management functions. The devices that connected to the platform are USB type c port, serial debug port, CPU fan, PMIC, scan matrix controller and mouse, thermal sensors etc. EC is directly interfacing to host through LPC/eSPI bus. Notifications related to these devices are sent to Operating System (OS) via embedded controller. EC provides ACPI function that defines hardware and software communication between an embedded controller and OS driver. EC can control and manage different I/O and internal features to perform that the Operating System does not handle. Those tasks can be power management, battery management, thermal management, the host interface, user input etc.

Development of this chip by writing firmware for different driver modules like PWM for fan control, GPIO, ADC for hardware monitoring, a timer to get the delay between two tasks, Watchdog timer, Interrupt to handle the events, eSPI, SMBus for battery management etc. Then compiling these drivers by Makefile commands and after that binary file will be generated. This firmware image will be implemented on board through SPINOR and get desired output for each driver modules.

Abbreviation Notation and Nomenclature

BIOS .... Basic Input Output System

eSPI .... Enhanced Serial Peripheral Interface

LPC .... Low Pin Count

ACPI .... Advanced Configuration and Power Interface

OSPM .... Operating System Power Management

PMIC .... Power Management Integrated Circuit

GPIO .... General Purpose Input Output System

ADC .... Analog to Digital Converter

PWM .... Pulse Width Modulation

WDT .... Watchdog Timer

SOC .... System on Chip

USB .... Universal Serial Bus

UART .... Universal Asynchronous Receiver Transmitter

ISR .... Interrupt Service Routine

IVCT .... Interrupt Vector Table

INTC .... Interrupt Controller

IRQ .... Interrupt Request

MAF .... Master Attached Flash Sharing

OOB .... Out Of Band

SMBus .... System Management Bus

Chapter 1

Introduction

1.1 Introduction

The embedded controller (EC) is a fundamental segment in ultra-mobile, modern mobile and implanted PC frameworks. An efficient EC sub-framework empowers energy conscious plans to acquire dramatic power savings. EC can control and manage various features to perform different tasks. Typically the EC performs following tasks:

- Power Plane Management
- Smart Battery Management
- Thermal Management
- Docking
- Host Interface
- User input (Matrix keyboard scanning, PS/2 pointing device, external PS/2 services)
- Peripheral Management (EEPROM, integrated Serial Port, shortcut keys, Lid switch etc.)

1.2 Current Scenario

This new embedded controller (EC) is highly integrated with various system functions. EC is directly connected to host (SOC) through eSPI/LPC. It provides ACPI embedded controller function, PWM for fan control and ADC for hardware monitoring, keyboard controller and scan matrix, a PS/2 interface for an keyboard or mouse devices, and system wake up functions for power management. EC also provides USB type-C and USB PD port control. Reduction in chip size and cost is the main advantage over older EC chip as older EC chip doesn't have these many advantages.

1.3 Problem Statement

The older embedded controller has chip size as well as cost related issues. It doesn't have features as new EC chip have like USB type-C port and USB PD control.

1.4 Objective

Embedded controller manages different features like GPIO, Timer, PWM, ADC, Watchdog timer, SMBus for battery management, Interrupt, eSPI, Chip power management, UART, scan matrix and keyboard controller, a PS/2 interface for external keyboard and mouse devices and many more.

The goal is to write firmware for the different modules of embedded controller and library file generation & verify it on the chip. Port this functionalities to the new embedded controller chip. This new EC chip is used on the next-generation platform.

1.5 Timeline

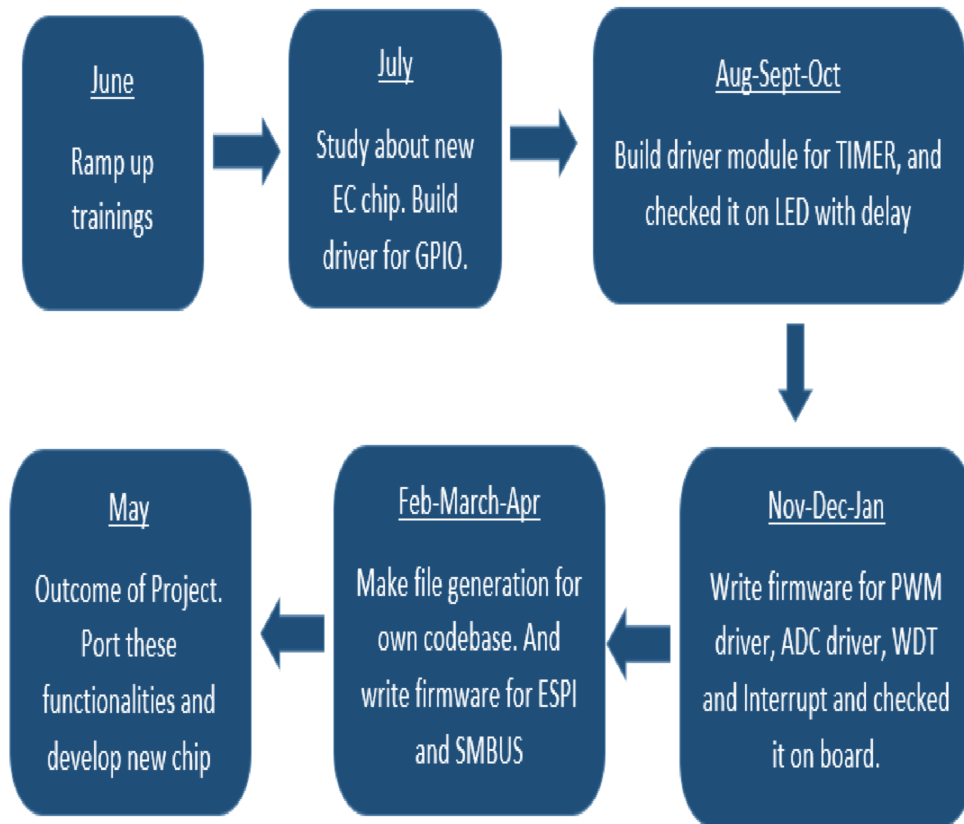

Figure 1.1: Timeline

Chapter 2

Literature Survey

2.1 Overview

ACPI (Advanced Configuration and Power Interface) is widely used for the productive treatment of power utilization in laptops and desktops. Computer's I/O devices, BIOS and operating system communicate with each other about power usage using ACPI. ACPI builds up industry-standard interfaces enabling power management, OS-directed configuration and thermal management of mobile, server platforms and desktop. ACPI is an essential component of INTEL's "INSTANTLY AVAILABLE" technology. OSPM handles device configuration events, thermal status, performance and power of the system. OSPM performs different functions like device management, system power management, processor management, system events, battery management, thermal management, SMBus, EC etc. Thus standard software and hardware communication done by ACPI. Embedded controller and OS driver communicate with each other using ACPI. The standard driver provided by OS can directly communicate with EC in the system.[1]
Multiple embedded controllers are supported by ACPI standard. There is an task query system that allow hardware implemented by EC to take attention of OS driver. Two interfaces are specified:

- Private Interface:- Interface that exclusively owned by the EC driver.
- Shared Interface:-Interface used by the EC driver and some other driver.

2.2 Embedded Controller Interface Description

Embedded controllers are used to support OEM-specific implementations. Embedded Controllers are supported by ACPI in any platform design. It is a unique feature and it can do different functions through connection to the microprocessor(s). The most commonly used EC among the variety of microcontrollers consisits a host interface that interface the EC to the host data bus and will allow bi-directional communication. This mechanism will reduce the latency of host processor in communicating with the EC.
In a shared interface, the EC is shared between system management code and OSMP. In private interface, there is a particular EC decode range for OSPM driver. And in private interface, EC and OSPM communicate without any additional software overhead with using Global Lock.
There are some additional embedded controller interfaces provided by such common system.[1]

a. Non shared embedded controller. This is very much common and system management handler is not required to communicate with the EC when system will be in ACPI mode.

b. Integrated keyboard controller and embedded controller. There are three interface like standard keyboard controller in default component (input-output component and chipset) and EC with two interfaces for different system management activities.

c. Standard keyboard controller and embedded controller. There are three interface which includes keyboard controller and EC with two interfaces for different management activities.

d. Two embedded controllers. There is four host interface with using two embedded controllers. One for keyboard controller functions giving two host interfaces and one controller for managing system activities giving up to two host interfaces.

e. Embedded controller and no keyboard controller. The keyboard functionality with completely different mechanism provided by future platforms that will provide two host interface in the EC for system related activities.

It requires some changes to handle the general embedded controller interface model which is shared between number of tasks processing under the OS control and the SMI handler like. Changes are like

- Firmware changes in EC
- Additional external hardware
- SMI handler firmware changes
- Operating software(OS) changes

2.3 Embedded Controller Register Descriptions

EC has three registers at two address locations: EC_DATA and EC_SC. EC_SC means EC Status-Command register perform like 2 registers in which status register for reading purpose and command register for writing to this port. EC_DATA is used for transferring the data between the embedded controller and host CPU.
EC_SC read-only register indicates only the current status of the embedded controller interface. Register details are shown in below figure.

OBF is set when the EC writes a data byte into the command but the host has not read it. When status bytes have been read by host, it sees that OBF flag is set, then it reads the data port to get data that EC has written. OBF flag is cleared

IGN:	Ignored
SMI_EVT:	1 – Indicates SMI event is pending (requesting SMI query).
	0 – No SMI events are pending.
SCI_EVT:	1 – Indicates SCI event is pending (requesting SCI query).
	0 – No SCI events are pending.
BURST:	1 – Controller is in burst mode for polled command processing.
	0 – Controller is in normal mode for interrupt-driven command processing.
CMD:	1 – Byte in data register is a command byte (only used by controller).
	0 – Byte in data register is a data byte (only used by controller).
IBF:	1 – Input buffer is full (data ready for embedded controller).
	0 – Input buffer is empty.
OBF:	1 – Output buffer is full (data ready for host).
	0 – Output buffer is empty.

Figure 2.1: Register detail[1]

automatically when the host reads the data byte. So it tells EC that data has been read by the host and so embedded controller can write more data to host.
IBF flag is there when the host writes a data byte to the command or data port but EC has not yet read it. IBF flag is cleared automatically when the EC reads the data byte.
The SCI event flag is raise when the internal event has detected by an embedded controller that required operating systems attention. In a status register, EC sets bit and make SCI event to OSPM.

The SMI event flag is raise when the internal event has detected by embedded controller and it requires the SMI handler's attention.
The Burst flag tells that EC has received that burst enable command is received to embedded controller from host. So this allows OSPM to immediately read and write the data bytes without any overwritten between SCI commands.

2.4 Embedded Controller Command Set

Embedded Controller Command	Command Byte Encoding
Read Embedded Controller (RD_EC)	0x80
Write Embedded Controller (WR_EC)	0x81
Burst Enable Embedded Controller (BE_EC)	0x82
Burst Disable Embedded Controller (BD_EC)	0x83
Query Embedded Controller (QR_EC)	0x84

Figure 2.2: EC Command Set[1]

These are several command set that allows communication between OSPM and the embedded controller.[1]

Chapter 3

Hardware Design

3.1 System Architecture

The system architecture of embedded controller is shown as below:

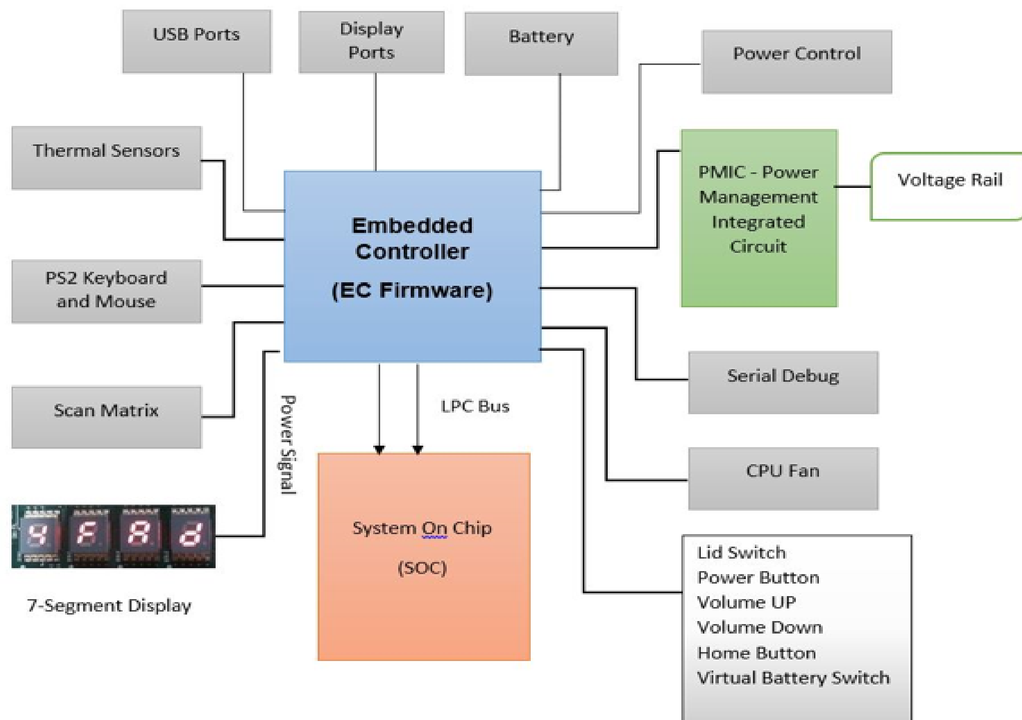

Figure 3.1: System Architecture

The embedded controller is basically a microcontroller with different internal features and I/O components. Figure 3.1 shows the architecture of the embedded controller. Embedded controller can manage control all the devices which are connected to onboard thermal sensors and platform, serial debug ports, USB port, display ports, CPU Fan, PMIC module. The EC firmware on reference platform is organized as a set of tasks that are managed by a multitasking dispatcher kernel. In a round robin fashion the dispatcher calls each task and allows it to run for a specified time slice. If the task has nothing to do it returns control to the dispatcher. Once all tasks are performed, the EC enters an idle low power state. Device notifications related to the temperature reading, battery management, power management are sent to OS via an embedded controller (EC). So for that, an embedded controller has some specific command set which has been described above and those commands are operated in ACPI region which of 256 bytes.

3.2 Block Diagram of Embedded Controller

Figure 3.2 is a typical block diagram of generic EC. We can see there are two main groups:- Programmable and Hard-wired. From the above figure, clock, interrupt controller, Timers, I/O and ADC converter comes from the hard-wired group. And Central Processing Unit, Random Acess Memory and memory blocks are from the programmable group. Functions of these blocks are expressed by the firmware that will be stored in memory. Below are some words used many times instead of the embedded controller.

- KSC: Keyboard and System Controller
- KBC: Keyboard Controller
- SMC: System Management Controller
- H8: H8 embedded controller frequently use in reference design.

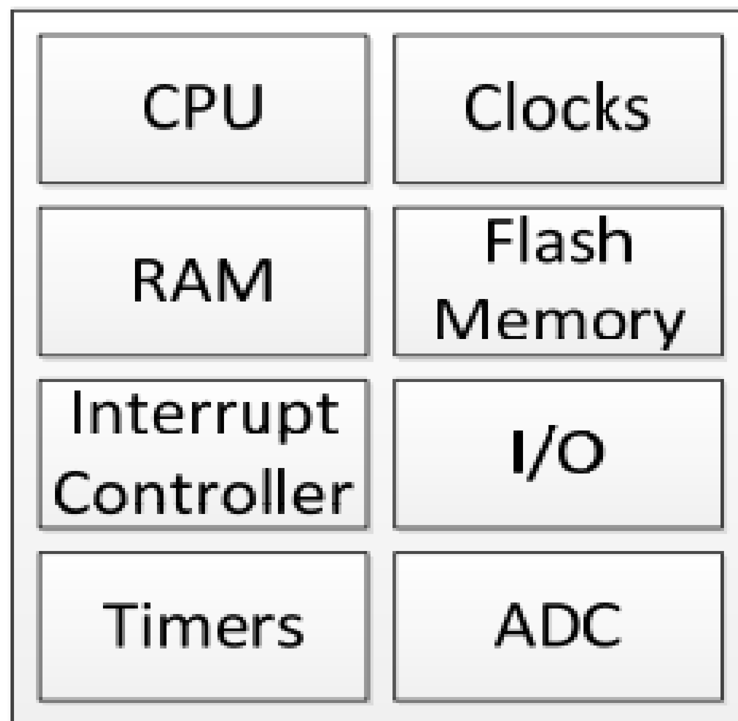

Figure 3.2: Block Diagram of Embedded Controller

3.3 Hard-Wired Functionality

The hard-wired functionality is further bifurcated into 2 parts:-
Keyboard Controller (KBC) and **System Management Controller (SMC)**
Functionalities for Keyboard Controller (KBC),

1. **Keyboard Matrix Scan Support**:- Keyboard keys are sorted into a matrix of rows and columns. These row and columns are represented by the number of signals. And due to this on board controller will translate those addresses to the simple protocol like PS/2, AT or USB.The embedded controller(EC) sends different codes on key press and key release.

2. **PS/2 keyboard and Mouse Interface**:- The embedded controller is connected to both keyboard and mouse. EC has a PS/2 controller inside so whenever we pressed any key on the keyboard, some message will come to EC. Then EC will

take that data and convert it into signaling and send different codes to HOST through eSPI/LPC.

Functionalities for System Management Controller (SMC),

1. **Thermal Management**:- Embedded Controller have pulse width modulation (PWM) interface and PWM is used for controlling CPU fan. EC reads a temperature of a fan and gives that data to host through Espi/LPC.

2. **Power Monitoring**:- Analog to digital converter(ADC) signal is used for controlling voltage in an embedded controller. This information used for monitoring battery charging or inform the administrator and user about power supply conditions.

3. **Battery Management**:- Embedded controller provides ACPI compliant operating system (OS) with notifications and status related to power management activities. It also generates wakeup events to take out the system from the low power states.

4. **ACPI host Interface**:- The embedded controller is used for controlling charging of the battery and also switching between AC adapter and the battery that used for monitoring various battry level like charging , discharging etc.

Chapter 4

Work done & Design flow

This section describes programming flow for the design of the different modules of the embedded controller. it basically gives the information about design flow and uses of each module like GPIO, Timer, Watchdog timer, PWM(Pulse Width Modulation) etc. The information collected about each feature is helpful to develop the modules.

4.1 GPIO(General Purpose I/O)

GPIO is known as General Purpose Input Output. GPIO is an independent input-output pins that are managed by registers. These input-output pins can be taken as input, output or alternate function. A GPIO port is a group of GPIO pins arranged in a group and controlled as a group. GPIO pins used as inputs to detect button press, to receive an interrupt from external devices. And also used as outputs to toggle the LED, sound in a buzzer and for controling power in devices. Both input and output modes have some states. Input modes have Pull-up, Pull-down and high impedance(z) state and output modes have Push-pull and open drain state.

a. **Pull-up:** Pull up circuit is shown in figure 4.1. Pull up consist a register which is placed between the input signal and supply voltage. So when the

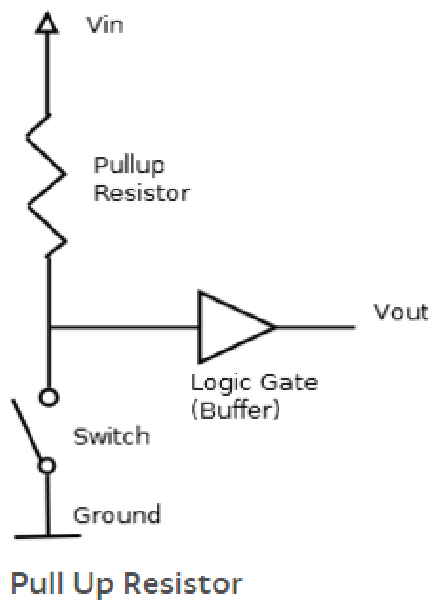

Figure 4.1: Pull-Up

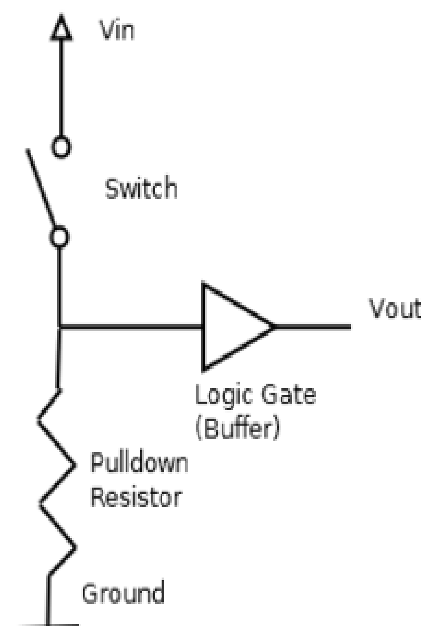

Figure 4.2: Pull-Down

switch is open, the status of the input pin will be high.

b. **Pull-down:** Pull-down circuit is shown in figure 4.2. Pull down consist a register which is connected to input signal and ground (GND). So when the switch is open, the status of the input pin will be low.

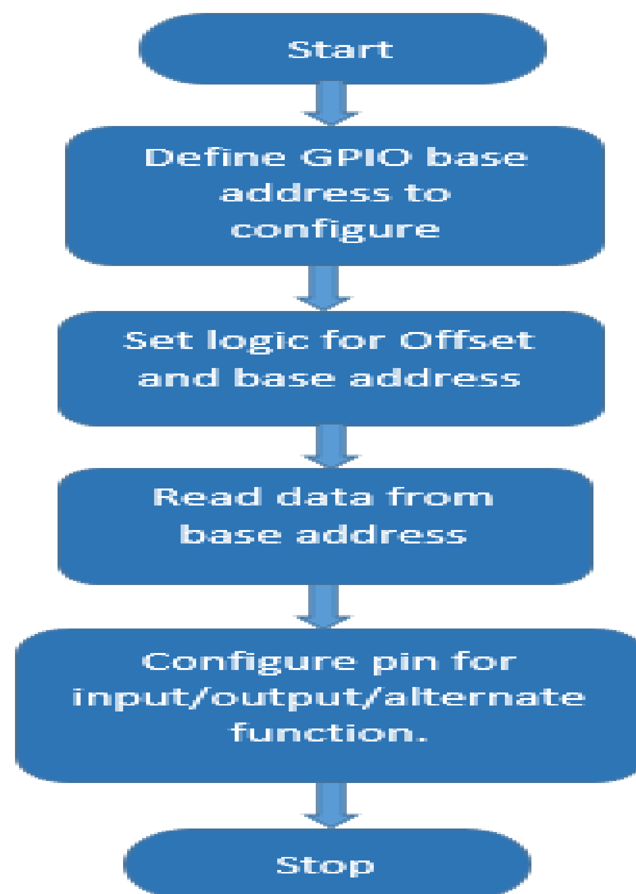

Figure 4.3: GPIO ConfigPin Flowchart

Figure 4.3, 4.4. and 4.5 is my program flow for GPIO driver.

GPIO API consists three functions like, IO_config, IO_get and IO_set pin state.

- Configured different GPIO pin as pull up, pull down, I/O and alternate function using a particular register set.
- Checked input, output, push-pull, open-drain on GPIO pins when EC changes different states like sleep, hibernate etc
- Set any GPIO pin as either high(1) or low(0).
- Get state GPIO to get the output data on the pin.

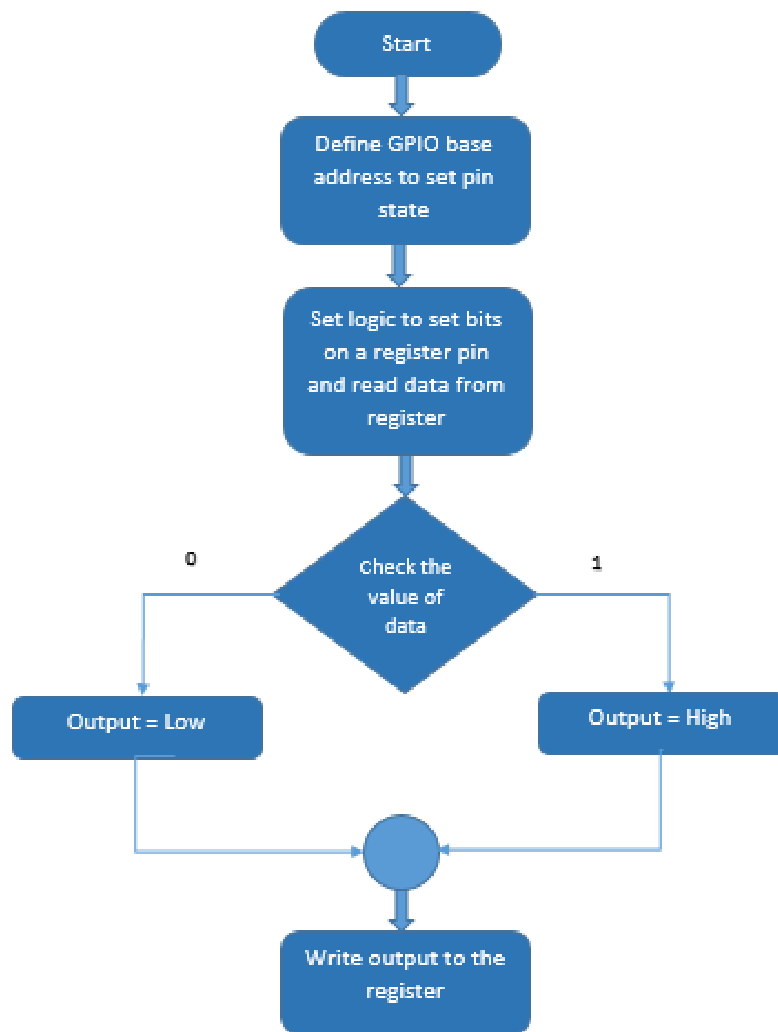

Figure 4.4: GPIO SetPin Flowchart

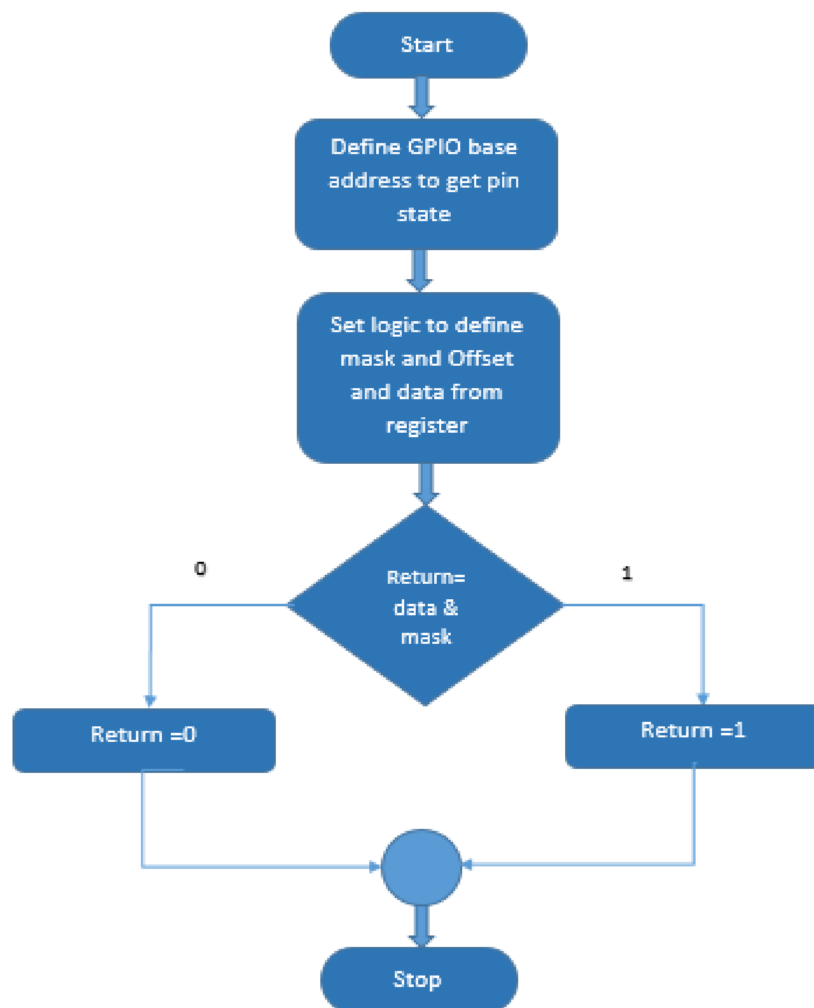

Figure 4.5: GPIO GetPin Flowchart

4.2 Timer

A clock that manage the order of a task while counting the fixed interval of time. A timer is basically used for generating time delay between particular events. A timer can also be used as counters to count an event or action. Counter value increases by one every time when a particular event occurs. We can also repeat an action or task after a known period of time using timers. Setting up an alarm clock that triggers for a certain period of time is the best example of timer clock.

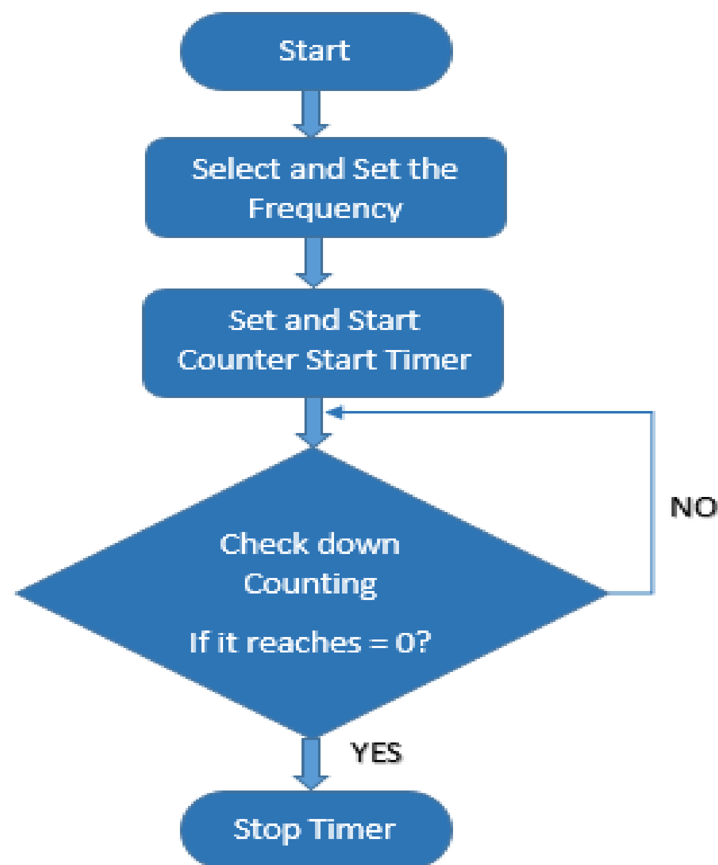

Figure 4.6: Timer Flowchart

Timers are the inbuilt chip in a controller and that is controlled by some special function registers (SFRs). Timers operations are assigned to that SFRs. Timers are configured in different modes of operations using these registers. Two different way to generate a time delay. First is by using infinite loops in c program, but the delays generated by the program are not that much accurate. So the option is to use Timers. And there is much more accuracy in time delay generated by timers.

For new EC chip, Timer driver consists timer initialization and delay timer functions.

- Initialized a timer to blink a particular LED.
- Setting a delay function to get the delays like 1 msec, 10 msec on LED and verified it on CRO.

4.3 Watchdog Timer

A watchdog timer is used for those embedded systems that cant be watched constantly by a human. Some embedded system designs have problems that are not accessible to human. Systems are permanently disabled if system software hangs. Speed would be too slow if human operator reset the system so it is not always possible to wait for someone to reboot the system.[3]

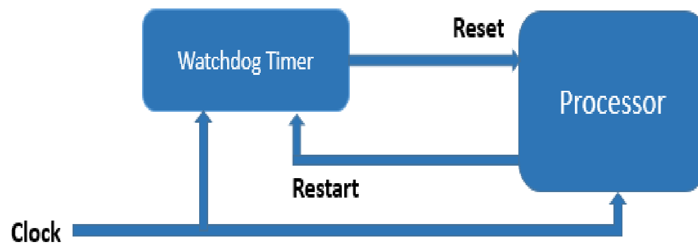

Figure 4.7: Watchdog timer[3]

A watchdog timer is nothing but a hardware that detects software anomalies automatically and reset the processor. A counter is defined and software selects the counters initial values. Counter numbers are defined in registers. The counter counts down to zero from that initial value. When a counter reaches zero watchdog timer will reset the EC domain.

Watchdog timer API consists Timer and watchdog timer initialization. WDT starts only when the timer is started which has been used.

- Timer initialization defines clock and counter number in a particular register.

- WDT initialization defines the same clock used for timer and a counter number that continuously toggles the LED on board.

Below is the program flow for Watchdog timer API.

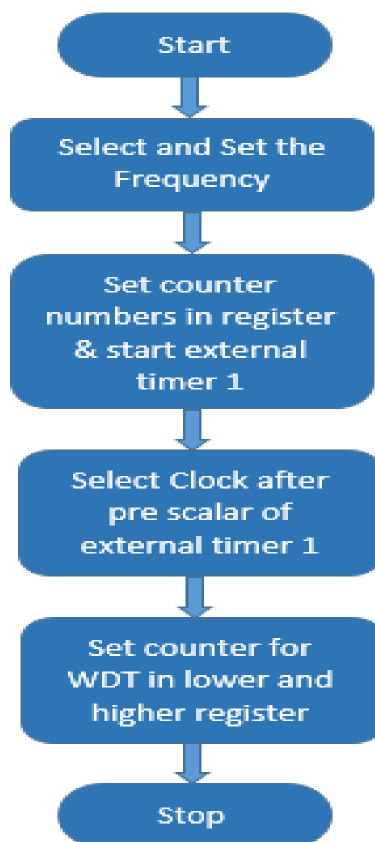

Figure 4.8: Watchdog timer flowchart

4.4 Pulse Width Modulation(PWM)

PWM stands for Pulse Width Modulation and is the strategy to deliver analog voltages in a digital manner. Basically, different value of voltages originate from analog circuits, and digital circuits create two level like ground(0v) and maximum voltage(5V, 3V). So if a user wants a voltage level between high and low then PWM is used for generating different pulse width.

PWM signals are generated by different built-in timers in an embedded controller. PWM signal is at a particular frequency on a device which receives that signal from the embedded controller. Duration of time, when the signal is high known as ON time and duration of time when the signal is low known as OFF time.

As shown in below figure 4.5, pulses have different duration of ON time as well OFF time. A period is the sum of ON time signal and OFF time signal. This period is inversely proportional to frequency. Within that period how many times a signal is high, known as **Duty Cycle**. The duty cycle is measured in percentage.[4]
Duty Cycle = ON time signal/ total period of time
figure 4.9 shows different duty cycle pulses and their ON time as well OFF time.
The embedded controller uses clock source and different built-in timers to generate PWM. Using these timers you can initialize the timer and counter. You can set a counter number so that at a specific count, a pulse can go high (ON) and when a counter reaches to zero, a pulse goes low (OFF). Thats how a user can control the pulses.

For this new EC chip, PWM driver consists PWM initialization and PWM duty cycle functions.

- Developed PWM and see different duty cycle on CRO.
- Set a condition like if the duty cycle is more than 200 then pulses will stop.

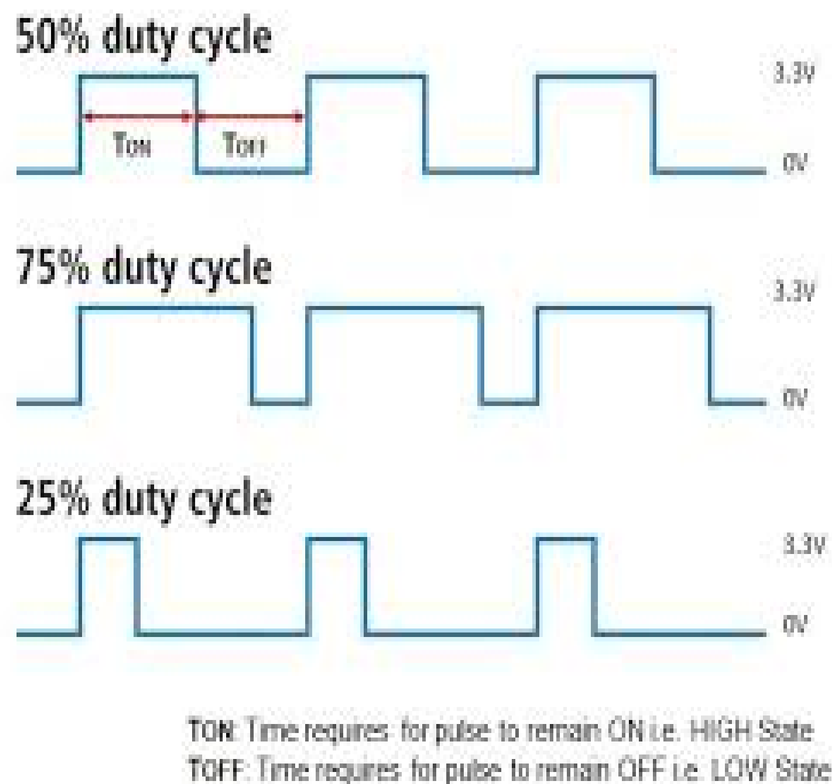

Figure 4.9: Duty Cycle

- Set duty cycle on for some time, then off for some time and then set it with different value and checked on CRO.

PWM flowchart is shown in below figure 4.10. It describes the process for register settings and steps to develop PWM pulses.

PWM Applications:-

- To control fan speed
- Switching regulators
- In LED dimming light
- To control DC motor speed varying from zero to any maximum speed

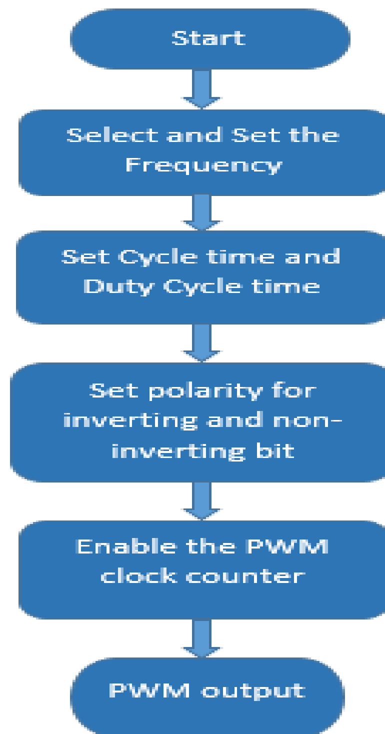

Figure 4.10: PWM Flowchart

PWM voltage regulation:- Averaging the value of PWM signal will give voltage regulation. You can see the relationship between DC output voltage and Duty cycle in figure 4.7.
Average DC voltage output = Voltage to represent High State x Duty Cycle, for example, the input voltage is 5 volt with 70% of duty cycle will give DC output = 3.75 volt.

4.5 Library File Generation

Two library files have been generated,

- One is for all the registers of all programmed modules.
- And another header file with a structure of different registers of programmed modules.

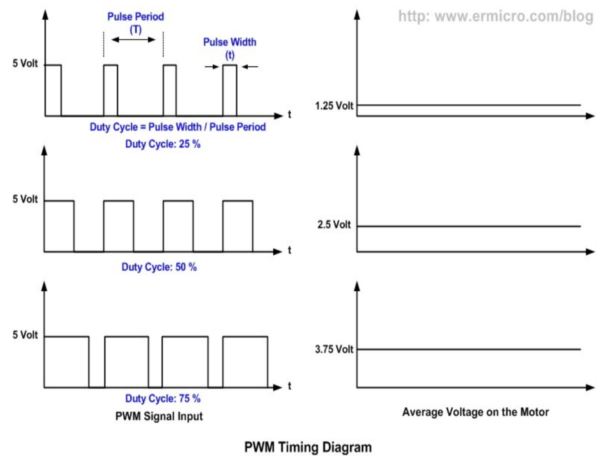

Figure 4.11: PWM Timing diagram

4.6 Analog to Digital Converter(ADC)

ADC is analog to digital converter used to convert the analog signal in to digital manner. Analog voltage values are not constant values. There are ground and maximum voltage value. For ex, we consider voltage range is from 0 to 3V so 0 V means it is binary 0 and maximum value 3 V means it is binary 1. Now, what is the value for 1.5 V. So, ADC will give all the variable voltage values in digital form. For slow changing voltage , ADC is the accurate method.[5]

AS per this embedded controller chip, ADC has 15 inputs and some of them are external voltages for DC voltage sources and others are internal supply voltage like VCC, VSTBY etc. There are 12 voltage buffers. ADC converts 0 to 3 v signal voltage channel first into 10 bit unsigned int and this 10 bit is then stored in data buffer registers.

ADC flowchart is shown in below figure 4.12.

- Got different raw data on serial log based on its ADC channel. There is 6 ADC channel and each of them has different raw data. Based on this data user can see its temperature value in the table.

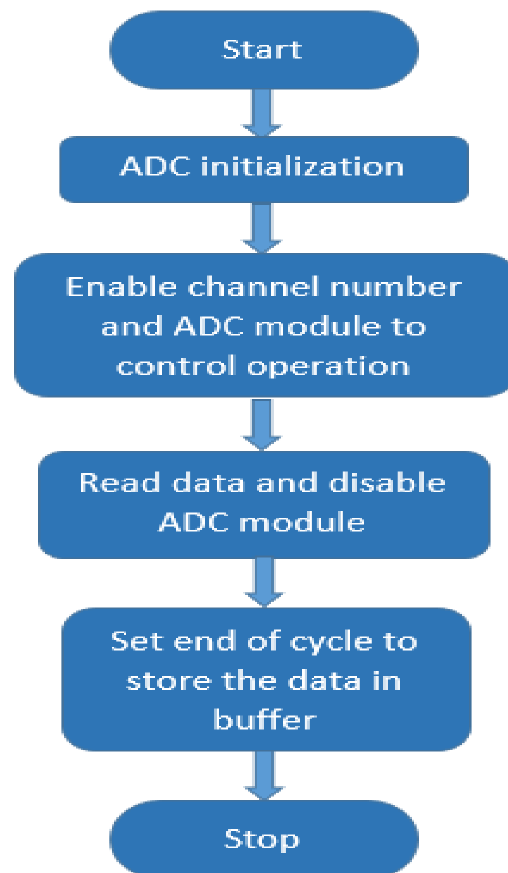

Figure 4.12: ADC Flowchart

Here is a snapshot of this serial logs in below figure. It shows channel numbers and different raw data in hex value.

4.7 Interrupt

The interrupt is a signal from a device that has some input for the processor. So it is an event of a particular device that requires the attention of microcontroller. The microcontroller will pause its current task whenever any interrupt occurs, and then it will execute the interrupt event. The code that gets executed on raising interrupt is called ISR of the corresponding interrupt. Then at the end of the ISR,

```
COM6 - PuTTY
ChNO 3
RawVal 02D0
ChNO 4
RawVal 02D6
ChNO 5
RawVal 02A2
ChNO 6
RawVal 0292
ChNO 3
RawVal 02D0
ChNO 4
RawVal 02D6
ChNO 5
RawVal 02A2
ChNO 6
RawVal 0292
ChNO 3
RawVal 02D0
ChNO 4
RawVal 02D6
ChNO 5
RawVal 02A2
ChNO 6
RawVal 0292
```

Figure 4.13: ADC serial logs

the microcontroller will return to its task that had pause and continue with its normal operation.[6]

There are two types of interrupt which are called software and hardware interrupt. If the microcontroller is interrupted by external device or hardware , then it is called hardware interrupt and if an interrupt is raised by any software instructions then it is called software interrupt. [6] If multiple devices need the attention of microcontroller, then interrupt controller is there to decide which device will raise interrupt and it sets the priority between them.[6]

A device or interrupt controller asserts the interrupt pin of the microprocessor and waits for the interrupt acknowledgement. On receiving this , the device place 8 or 16 bit data in the data bus of the processor. Each device has a unique number. This array/vector of the interrupt handler is known as the interrupt vector table(IVCT).[6]

Interrupt controller flowchart is shown in below figure.

EC chip has Interrupt controller(INTC) which collects several interrupts from modules.

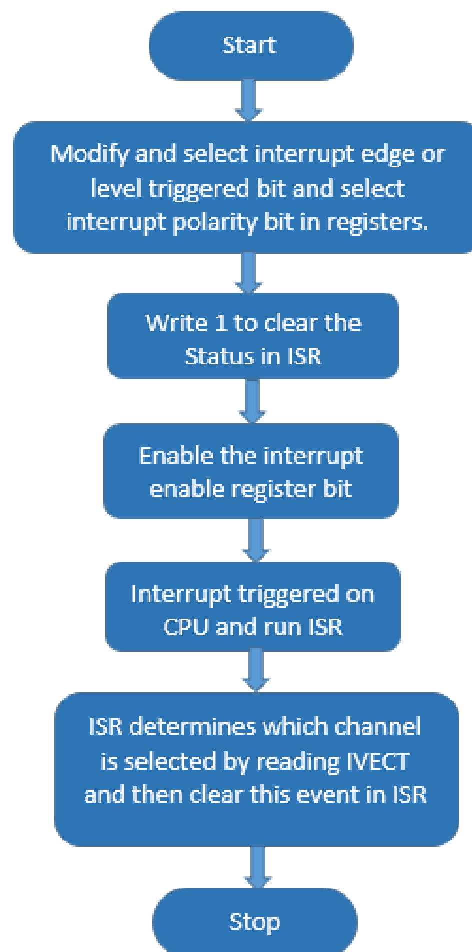

Figure 4.14: Interrupt Flowchart

INTC features:-

- Edge triggered and edge triggered mode
- Interrupt polarity or triggered mode
- Each interrupts source able to enabled/masked individually
- It has individual vector index number from IVCT0-IVCT15 for each interrupt output from INT0-INT15

Each interrupt groups have total 20 registers. Interrupt groups are for interrupt enable register, interrupt edge/level triggered registers, polarity register, status register etc. A user can see group number based on which interrupt is going to use from interrupt vector table. For example, a timer interrupt is from group 3 and on INT30 signal. So the user will see group 3 for each register and set particular bits.

- So on first button press LED will glow and the user will receive interrupt. And on second press LED should glow off. Likewise LED is toggling on each press.

4.8 Make File

A special file which has different shell commands and different paths for all the source file. While user type "make", all the commands in the make file will be executed. There is a list of shell commands and these commands are written for the shell which will process the makefile.
By this "make" command, a user can compile all the source file and from .c files, a user will have object(.o) files as an output.

- Made a "Make File" for code base generation.
- Defined own code base which has different folders for each type of files. for example, object folder will have all .o files. Driver folder has all my driver module's source file. Different paths for all the source file and code base is shown as below figures.

```
98  # all Driver obj files
99  #
100 DriversFiles=\
101         $(OBJ_PATH)\ivect.o\
102         $(OBJ_PATH)\timer.o\
103         $(OBJ_PATH)\wdt.o\
104         $(OBJ_PATH)\adc.o\
105         $(OBJ_PATH)\pwm.o\
106         $(OBJ_PATH)\int.o\
107         $(OBJ_PATH)\gpio.o\
108
109 #
110 # all APP obj files
111 #
112 AppFiles=\
113       $(OBJ_PATH)\main.o
114 #
115 # all OS obj files
116 #
117 OSFiles=\
118       $(OBJ_PATH)\kernel.o
119 #
120 # all CPU obj files
121 #
122 CPUFiles=\
123       $(OBJ_PATH)\cpu.o
124 #
125 # all nds obj files
126 #
127 NDSFiles=\
128         $(OBJ_PATH)\crt0.o
129
```

Figure 4.15: Make File snapshot

```
224
225 #-----------------------------------------------------------------------
226 # Compile App file
227 #-----------------------------------------------------------------------
228 {App\}.c{$(OBJ_PATH)\}.o:
229     $(CC) $(CDirectives) $(ITE_DEFS) -o $*.o $<
230
231 #-----------------------------------------------------------------------
232 # Compile cpu file
233 #-----------------------------------------------------------------------
234 {CPU\}.c{$(OBJ_PATH)\}.o:
235     $(CC) $(CDirectives) $(ITE_DEFS) -o $*.o $<
236
237 #-----------------------------------------------------------------------
238 # Compile OS file
239 #-----------------------------------------------------------------------
240 {OS\}.c{$(OBJ_PATH)\}.o:
241     $(CC) $(CDirectives) $(ITE_DEFS) -o $*.o $<
242
243 #-----------------------------------------------------------------------
244 # Compile board file
245 #-----------------------------------------------------------------------
246 {board\}.c{$(OBJ_PATH)\}.o:
247     $(CC) $(CDirectives) $(ITE_DEFS) -o $*.o $<
248
249 #-----------------------------------------------------------------------
250 # Compile Drivers file
251 #-----------------------------------------------------------------------
252 {Drivers\}.c{$(OBJ_PATH)\}.o:
253     $(CC) $(CDirectives) $(ITE_DEFS) -o $*.o $<
254
```

Figure 4.16: Make File commands

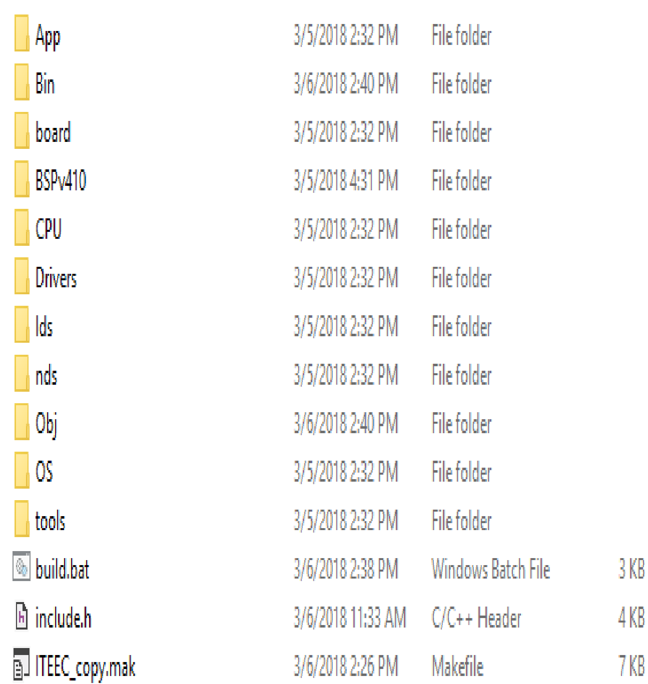

Figure 4.17: Code Base snapshot

4.9 Board Support Package(BSP)

When the user executes a programm or source file, there is some startup code that gets executed before the execution reaches to the main function. Compiler vendors usually provide appropriate startup code with their compiler toolset for the particular platform. This startup code gets executed before the main function is reached. I have initialized all the GPIO signals from the schematic based on their direction(I/O), types like pull up-pull down-open drain and made a table for that.

4.10 Enhanced Serial Peripheral Interface(eSPI)

Enhanced Serial Peripheral Interface(espi) is used by the system host to configure the chip and communicates with the logical devices implemented in the design through the series of read/write registers. Espi is a serial bus and it is based on SPI. ESPI is basically LPC(low pin count) replacement as it has more benefits such as low voltage, low power, higher bandwidth, pin count saving etc. ESPI is having 20MHz to 66MHz clock speed.

There are only one master and one or more eSPI slaves. For EC, it is most of the time slave and all other devices are master. Espi master and slaves are communicated through various signals like "Clock" that provides timing for all the serial input and output operations, "Chip select" to select a particular espi slave for the transaction, "I/O" used to transfer data between master and slaves , "Alert" used by eSPI slave to request services from eSPI master and "Reset" pin to reset the espi interface for both master and slave. [7]

ESPI operates in master/slave mode where commands and data flow between espi master and slave by controlling Chip select pins for each of espi slave.

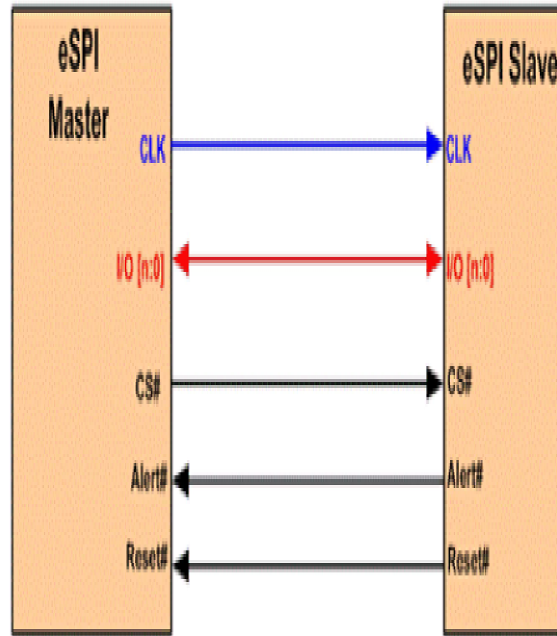

Figure 4.18: Master-Slave Interface

ESPI features:-

- Flash access channel/Master attached Flash sharing(MAF)
- Peripheral channel
- OOB meassage channel
- Virtual wires channel

a. **Flash Access Channel:-**

Flash components are shared run time between a chipset and the eSPI slaves that requires flash access such as EC using flash access channel. The flash access channel is enabled on the eSPI slave side, once the flash controller in the chipset has completed the flash initialization. For the MAF, flash components are attached to the eSPI master such as chip set. EC is allowed to access the

shared flash components through the flash access channel.[7]

b. **Peripheral channel:-**

ESPI channel is used for communication between eSPI endpoints located on the slave side and eSPI host bridge located on the master side. Example for this espi host bridge and endpoints are LPC Host and LPC peripherals. Another example is ACPI devices connected to the Espi bus which talk to a host controller residing on espi master side. Peripheral memory or I/O packet format consisits information about cycle type, Tag, length, address and data bytes. The packet format is shown in below figure.

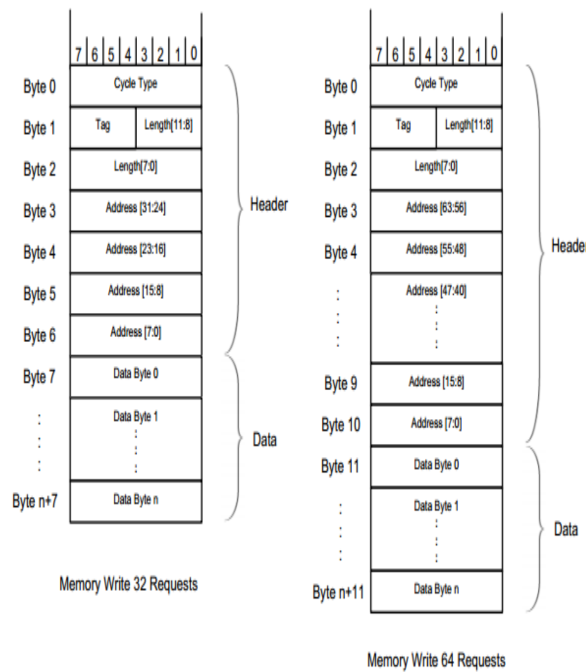

Figure 4.19: Peripheral packet format

c. **Out-Of-Band channel:-**

The OOB channel is used to handle transaction between the OOB processor and EC. EC is able to initiate an upstream OOB message transaction for

reading hardware information, including temperature and time/date, using messages with predefined slave address and command codes. EC is able to initiate and receive OOB message transaction over eSPI bus. OOB packet format also consisits cycle type, length, Tag and data bytes to send and receive using several commands like PUT_OOB and GET_OOB.

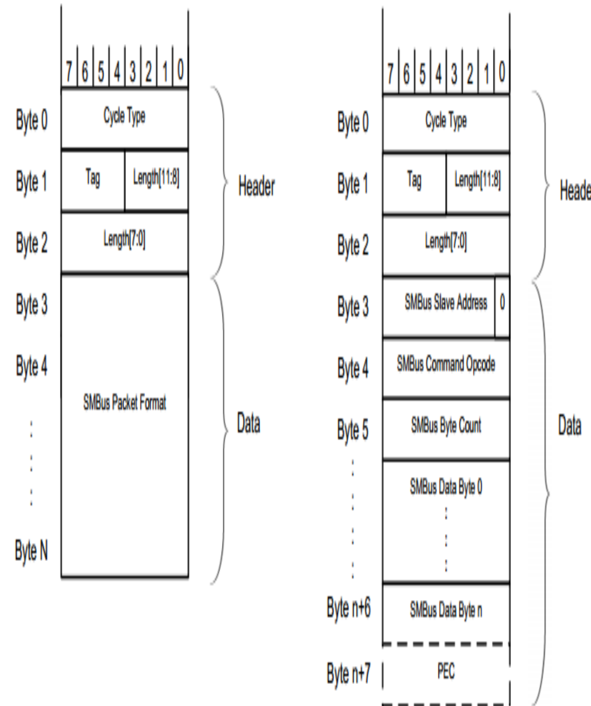

Figure 4.20: OOB packet format

d. **Virtual Wire channel:-**

Virtual wires are used to communicate between Embedded Controller and Platform Control Hub. Serial IRQ interrupts are communicated through this channel as in band message.In band message pass the control data on the same connection of the main data like FTP, HTTP. It is different from out of band message protocol. Virtual wire packet format is shown as given below.

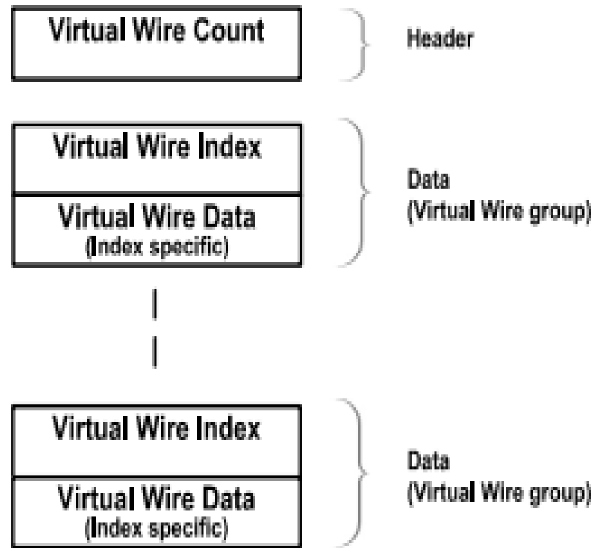

Figure 4.21: Virtual wire packet

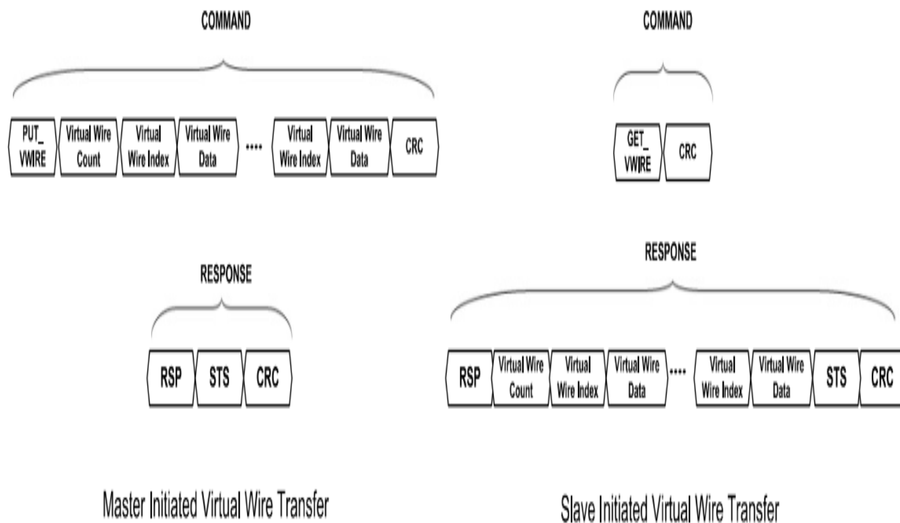

Figure 4.22: Virtual wire command

The Command phase consists of a command opcode ,virtual wire packet and CRC. The packet format for command and response is as shown as above figure 4.21. It has commands to send the data like PUT_VWIRE followed by virtual wire count and groups and ended with CRC bit. This is command phase for master initiated virtual wire transfer for slave it is vise versa.

Virtual wire packet consists "Header" and "Data" block whereas header has virtual wire count and data block has both index and data. [7]

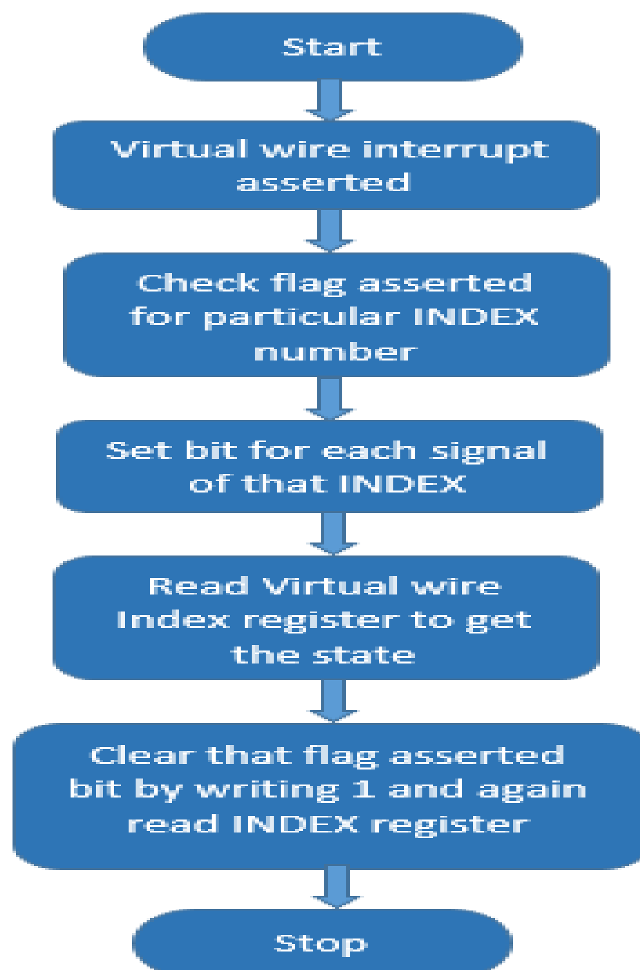

Figure 4.23: Virtual wire Flowchart

Virtual wire consists maximum 7 count numbers. It supports interrupt events with 16 IRQ interrupt and also system event. System event has index from index 2 to index 7.

The index gives information about the direction of transfer whether it is master to slave or slave to master, index number and different signal's bit position.

Each index has their own signal's bit set like sleep s3 for suspend state, sleep S4 for hibernate state, sleep S5 for shutdown state, wake up events, platform reset signal, power button, keyboard controller bit, system management, system controller interrupt etc. Based on tht requirement user can set a particular signal's bit set.

4.11 SMBus Interface

SMBus stands for System Management Bus. SMBus is a bi-directional communication protocol which requires communication between master and multiple slave devices or also with more than one master devices. It is based on the principle operation of I2C. [8]

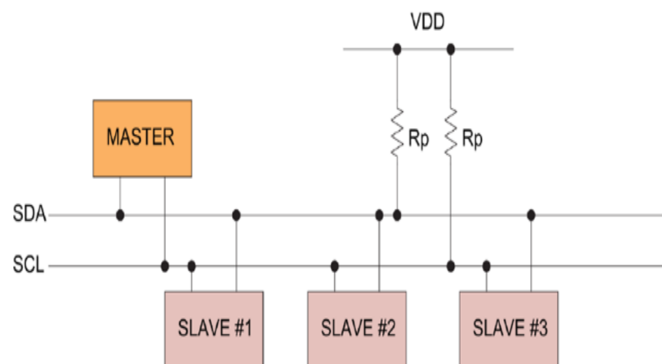

Figure 4.24: SMBus

Each slave device is having unique IDs so the master can select a device to communicate.
The two signals are called SCL(serial clock) and SDA(serial data) . SCL line is generated by the master and it transfers data between multiple devices on the I2C bus and SDA line brings the data.

Control bus provided by SMbus for system and power management task. Instead of using individual control lines a system may use SMBus to pass to and from devices. Removing control lines will reduce the pin count.

A device can give information to the system about device number, save its status for suspend event, send errors and return the status by using SMBus interface.[8]

This EC chip has total four SMBus channel and it can also perform SMBus messages with PEC either enabled or disabled. SMBus master supports two 32 bit FIFO read/write mode. Embedded Controller(EC) is most of the time master in SMBUS protocol and all other devices are slave.

a. **Quick Command:-**
   In this command, slave address register with R/W bit is sent. It is used to enable/ disable a device function. No data is sent or received.[8]

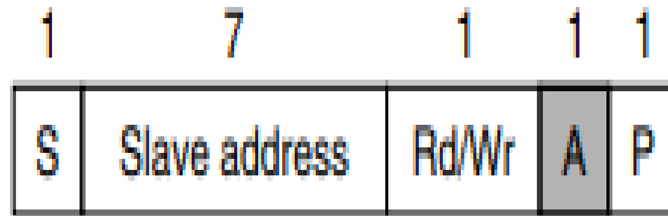

Figure 4.25: Quick command

b. **Send Byte/Receive byte:-**

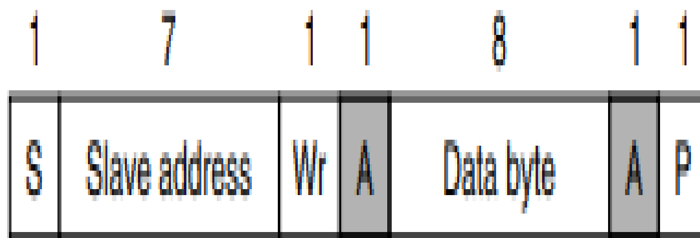

Figure 4.26: Send Byte command

Here, slave address and encoded commands are sent. Both send and receive byte commands are same as shown in figure 4.26 and 4.27, the difference is the direction of the data transfer. A host needs a device information like if it is a battery, then master(EC) needs to know about its battery percentage, capacity etc. In both the figures, a user can see send and receive byte command bit set position.

c. **Write byte/word:-**

Here, slave address is sent by master followed by write bit. Then A is for acknowledgment and master then send command code. The slave again ac-

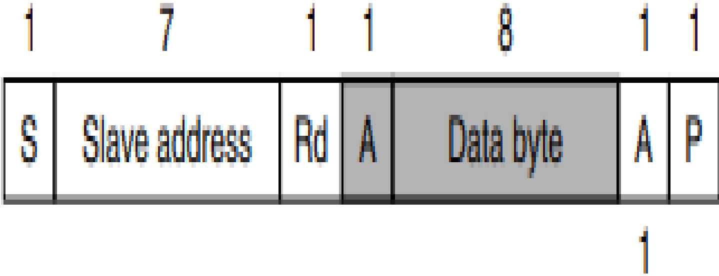

Figure 4.27: Receive Byte command

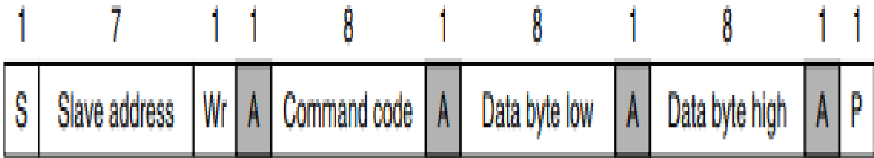

Figure 4.28: Write Byte/Word command

knowledges before master sends data byte or word as shown in below figure. And the transaction is terminated by the stop bit.

d. **Read byte/word:-**

Here, the host writes command to the slave address and then there is a repeated start condition that reads from device. Then two bytes of data will be returned by slave device. Before repeated START condition, there is no STOP condition.

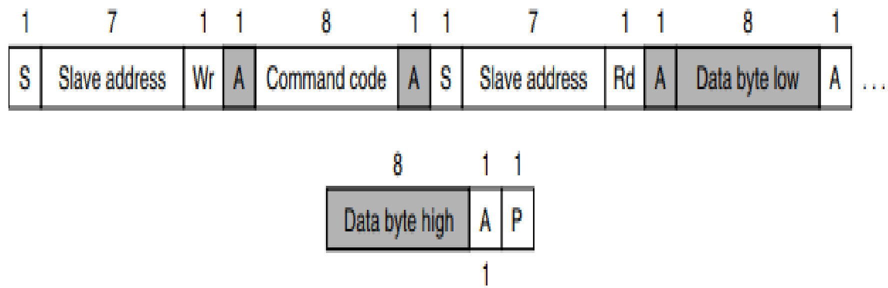

Figure 4.29: Read Byte/Word command

e. **Process call:-**

Here command code sends a data bytes and it will wait for that value which is dependent on that data and return by the slave. There is write word block and after that Read word block.

f. **Block write/read:-**

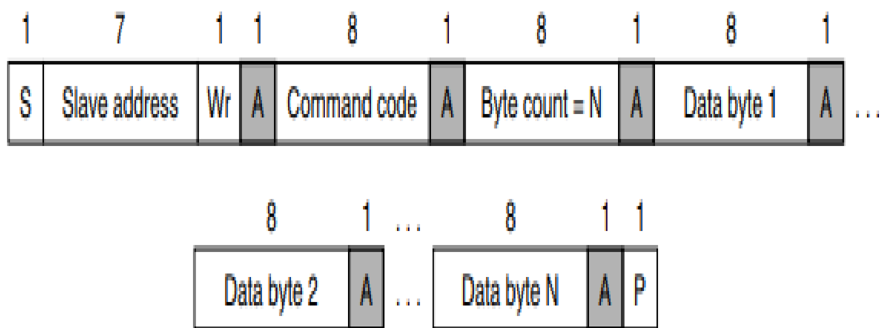

Figure 4.30: Block Write/Read command

Here, host sends byte count that tells number of bytes to be followed by the message. For example, If the slave has 25 bytes to send, then byte count will have 25 value(19h).

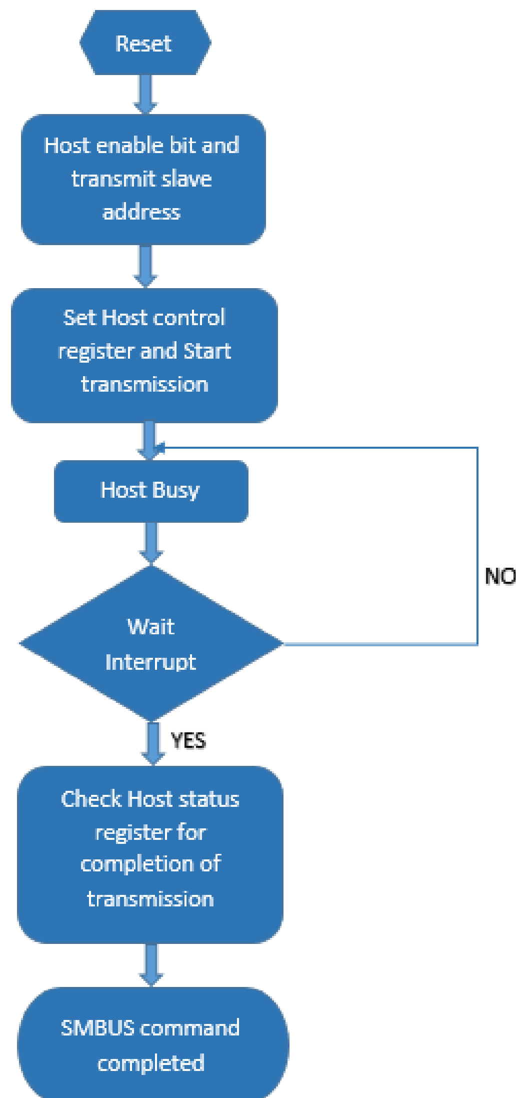

Figure 4.31: SMBus Flowchart

This is the software sequence of SMBus protocol. SMbus is used in he Smart Battery System (SBS).
SBS system consists a Host, a Smart Charger, and a Smart Battery. The Smart Battery and Smart Charger can communicate with each other and with the rest of the system.

Fuel guage is the internal chip inside the battery pack. It will read the analog

data from the battery and convert it into digital form. Fuel gauge can read the information from battery pack like charging, discharging, capacity etc. When Embedded Controller(EC) wants to communicate to the fuel gauge, it will read through SMBus and gets the digital data.

Chapter 5

Conclusion

5.1 Conclusion

The new embedded controller (EC) chip is developed by writing firmware of all the modules and verify it on a chip. And then porting these functionalities to new EC chip. So the following application driver has been developed.

- GPIO:- Configure different GPIO pin and checked GPIO status with LED high or low.
- Timer:- Blink LED on board and checked different delay timer on LED and verify it with CRO.
- Watchdog timer:- Set Watchdog timer on the LED.
- PWM:- Check different PWM duty cycles of PWM on CRO.
- ADC:-Develop Analog to digital converter and get serial logs of temperature values and raw data.
- Interrupt:-Develop Interrupt for LED blinking events on board.
- Make File:-Make file for code base

- ESPI :- Develop ESPI bus protocol, one for communication between EC and SOC.
- SMBUS:- Develop SMBus interface for battery management.

So I have developed this new Embedded Controller(EC) by writing firmware for these many modules to port it on next generation platform.

Chapter 6

Future Scope

Future Scope includes the development of the power management feature, CPU thermal management, Keyboard scan matrix and PS/2 mouse. These features can be developed and implemented on board. And moreover, all other functionalities will be ported on to the next generation board.

References

[1] http://www.uefi.org/sites/default/files/resources/ACPI_6_1.pdf

[2] https://www.thailand.intel.com/content/dam/www/public/us/en/documents/white-papers/controller-usage-low-power-designs-paper.pdf

[3] https://www.embedded.com/electronics-blogs/beginner-s-corner/4023849/Introduction-to-Watchdog-Timers

[4] https://www.newbiehack.com/MicrocontrollerIntroToPWM.aspx

[5] https://learn.sparkfun.com/tutorials/analog-to-digital-conversion

[6] Embedded Realtime System Programming by Sriram V Iyer and Pankaj Gupta

[7] https://www.intel.com/content/dam/support/us/en/documents/software/chipset-software/327432-004_espi_base_specification_rev1.0_cb.pdf

[8] https://www.nxp.com/docs/en/application-note/AN4471.pdf