GSM based Configurable Remote Power Management System with Intrusion Alert

A project report submitted in the fulfillment of the requirements for the award of the degree of B.Tech Computer Science & Engineering

By

Supratim Das

Under the Guidance of

Asst. Prof. Abhijit Mitra

Department of Computer Science & Engineering

Calcutta Institute of Engineering & Management

A project report prepared and submitted as per regulations for the degree of Bachelor of Technology in Computer Science and Engineering

BY

SUPRATIM DAS

REGISTRATION NO:- 081650110057 OF 2008-2009

ROLL NO:- 08165001044

GUIDED BY

ASST. PROF. ABHIJIT MITRA

DEPT. OF COMPUTER SCIENCE AND ENGINEERING

CALCUTTA INSTITUTE OF ENGINEERING AND MANAGEMENT

(A GOVT. AIDED, AICTE APPROVED, WBUT AFFILIATED, SELF FINANCED ENGINEERING COLLEGE)

24/1A CHANDI GHOSH ROAD, KOLKATA – 700040, WEST BENGAL, INDIA

2011 – 2012

DECLARATION

I hereby declare that this project report is based on my original work except for citations and quotations which have been duly acknowledged. All hardware and software developed as a part of this project are part of my original work and research. Third party libraries/APIs (open source or proprietary) and supporting hardware used is also duly acknowledged. I also declare that it has not been previously and concurrently submitted for any other degree or award at CIEM or other institutions.

Signature:- ……………………….

Name:- Supratim Das

Reg. No:- 081650110057

Roll No:- 08165001044

Date:- ……………………….

CERTIFICATE

This is to certify that the project on ***GSM based Configurable Remote Power Management System with Intrusion Alert*** *submitted by* ***Supratim Das*** *has been prepared according to the regulations of the B.Tech degree in Computer Science and Engineering as per suggested by the West Bengal University of Technology and they have fulfilled the requirements for the submission of the project. All hardware and software developed in the project are based on original work and research. Third party library/API (proprietary or open source) and supporting hardware used if any are duly acknowledged.*

…………………………..

Supervisor

Computer Science and Engineering

Calcutta Institute of Engineering and Management

…………………………..

Head of Department

Computer Science and Engineering

Calcutta Institute of Engineering and Management

…………………………..

Principal

Calcutta Institute of Engineering and Management

ACKNOWLEDGEMENTS

I would like to thank everyone who had contributed to the successful completion of this project. I would like to express my gratitude to my project supervisor, Asst. Prof. Abhijit Mitra for his invaluable advice, guidance, tolerance, acceptance and his enormous patience throughout the development of the project. I would also like to thank the entire open source community for making available the open source libraries/APIs used in the development of the project.

Table of Contents

ABSTRACT

The initial objective of this project was to use a cell phone as a remote controlling device to operate and monitor a wide range of devices. The idea to use a cell phone as a remote control has many obvious reasons. The most significant of these can be cited as firstly, the GSM network is a standardized network and very reliable and secondly, it is possible to achieve large coverage without having bulky and expensive setup or having to face legal hassles in terms of radio spectrum usage. The idea has its root in another project which was a mobile controlled rover which the author observed at a science and technology expo at Birla Industrial and Technical Museum in the year 2011. The project used DTMF (Dual Tone Modulated Frequency) as a means of control for the rover. The simplicity of the project was attractive and the idea about a HACS (Home Automatic Control System) using the same principle was conceived. After some initial research it became evident that using DTMF as a mode of control has some flaws when implementing something more complicated as a HACS. To state a few of the problems, are glitches due to noise, problem in communicating response messages to the user and poor configurability. The next approach which is finally implemented and hereby discussed in this report is using SMS (Short Messaging Service) as a mode of control and response. The System which is discussed here uses simple text based commands in the form of SMS for controlling devices attached to it. Although many references to similar kind of work can be found ranging from health monitoring systems to HACS, an attempt has been made to add easy configurability to the system. The system supports an USB (Universal Serial Bus) interface and through a configuration software written in JAVA, the system can be easily reconfigured by the user to alter/add/remove devices attached to it, and the phone number to which the device responds when it receives a SMS command. The system also features an intrusion alert, which notifies the user about an intrusion in the form of a SMS to the registered number. The system in effect makes an average home more accessible, with every control of almost any device that can be imagined right at our pockets anywhere we go. This report also discusses extensively hardware and software design of embedded systems and various modes of interfacing a custom device with a PC, and specifically the USB protocol and framework.

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND

In today's busy schedule it is a common observation that people are often forgetful regarding matters like switching off household electrical appliances. Many people faced situations where they leave in a hurry and later they can't remember whether they have turned off the AC, lights or fans. This is an act of wastage of energy. Also leaving electrical appliances such as heaters, microwaves, geysers on for extended hours without check may prove dangerous. Also people are concerned about burglary and theft when they are away from their home. A HACS (Home Automatic Control System) is an effective solution to the above mentioned problems. In this report the implementation of an economic and user friendly GSM based Configurable Power Management System with Intrusion Alert is discussed.

The major challenge that is faced when implementing a HACS is the mode of control. The obvious answer to this problem is a dedicated remote control, but again this solution would prove to be bulky, expensive and will have a limited range. Using cell phone as a remote control device effectively counters all the above mentioned problems, as cell phones are easily available, cheap and handy as well as the GSM network has a wide coverage and as per international standards hence quite reliable. Communication between a cell phone and the HACS can be done by three different means which are

1. DTMF (Dual Tone Modulated Frequency)
2. SMS (Short Messaging Service)
3. GPRS (General Packet Radio Service)

Initially DTMF was the choice of implementation, but after some initial research it became evident that using DTMF will have a number of limitations and will not be appropriate for implementation of a HACS. DTMF is the tone that is generated when a number pad key is pressed in a cell phone. The numeric information i.e. the key pressed, is encoded in the tone which is composed of two distinct frequencies, hence the term Dual Tone. When DTMF tone is fed to a DTMF decoder IC (MT8870) it outputs the BCD (Binary Coded Decimal) equivalent code on its output lines, which can be read by a microcontroller. Although this approach is simple, it is problematic due to noise glitches and conveying message from the HACS to the user.

SMS is the next elegant solution in our list and does not suffer from problems as mentioned in case of DTMF. SMS is a form of short text message that can be exchanged between cell phones or modems over a GSM/CDMA network. The system discussed in this report is based on SMS commands and SMS responses. The user types simple commands in their cell phone and sends it to the HACS. The HACS on the other hand is equipped with a GSM modem, which intercepts the SMS and sends it to the microcontroller over a serial line for further processing. The system on successful execution of the command sends back a response in the form of a SMS to the user's cell phone.

Alternatively GPRS may also be used for the same purpose and it would let the user monitor and control the HACS over the Internet. GPRS is the packet data service that is offered by almost every telecom operator nowadays. GPRS is the technology that lends Internet connectivity to cell phones and would have done the same in the case of HACS. But imparting internet connectivity would lead to a more complicated design, which would include implementing the TCP/IP stack for the microcontroller and also developing some kind of custom protocol or implementing the HTTP protocol for proper communication.

1.2 OBJECTIVES, AIMS AND MOTIVATION

The objective of this project is to build a GSM based Configurable Power Management System with Intrusion Alert, that would impart remote controlling of household electrical appliances and security system that would notify the user in case of an intrusion over a GSM network in the form of SMS.

The aim of this assessment is to provide controlling of home appliances remotely and will also enable home security against intrusion in the absence of home owner.

The motivation is to provide users with home automation solution that is economic, sophisticated yet easy to develop, highly portable, mobile and easy to use. The system would impart remote access to household electrical appliances including status reports, on/off control and timers. For ease of use and flexibility the system can be configured as per user requirements. This system can be considered as the primitive stage of development to the realization of a totally online smart home.

1.3 PROJECT OUTCOME

- To develop embedded hardware design skills
- To develop embedded software design and writing skills
- To develop GUI based software design and writing skills in JAVA SWING
- To develop device driver writing skills
- To develop and induce creative thinking and problem solving skills

CHAPTER 2

LITERATURE REVIEW

2.1 GLOBAL SYSTEM FOR MOBILE COMMUNICATION (GSM)

In 1982, the European Conference of Postal and Telecommunications Administrations (CEPT) created the Groupe Spécial Mobile (GSM) to develop a standard for a mobile telephone system that could be used across Europe. GSM (Global System for Mobile Communications: originally from Groupe Spécial Mobile) is the most popular standard for mobile telephony systems in the world.

2.1.1 Structure of GSM network

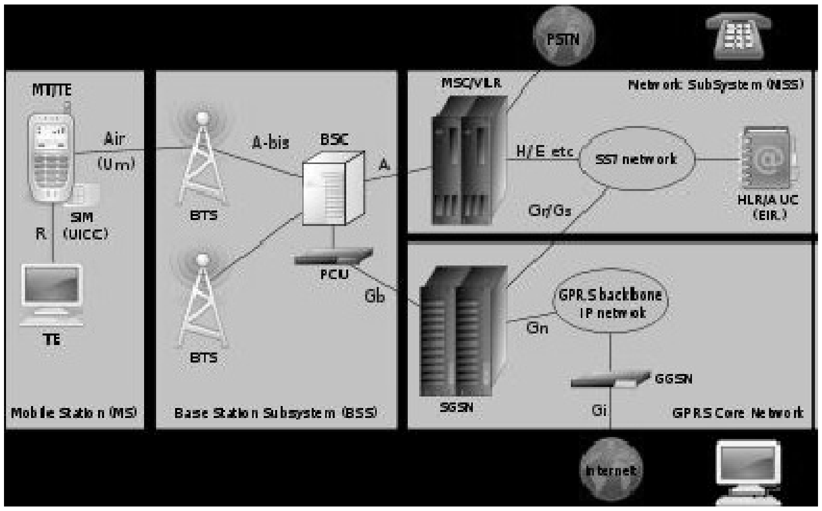


Figure 2.1: The structure of GSM network

The network is structured into a number of discrete sections:

- The Base Station Subsystem (the base stations and their controllers).

  It handles handling traffic and signalling between a mobile phone and the network switching subsystem. The BSS carries out trans-coding of speech channels, allocation of radio channels to mobile phones, paging, transmission and reception over the air interface and tasks that are related to the radio network.

- The Network and Switching Subsystem (the part of the network most similar to a fixed network).

  It is also referred as core network. It provides functions such as out call switching and mobility management for mobile phones roaming on the network of base stations. It allows mobiles devices to communicate with each other and as well to telephones through wider Public Switched Telephone Network (PSTN).

- The General Packet Radio Service (GPRS) Core Network (the optional part which allows packet based Internet connections).

  GSM mobile phones use GPRS system for transmitting IP packets. The GPRS core network is the centralized part of the GPRS system.

- The Operations Support System (OSS) for maintenance of the network.

  It is used by telecommunications service providers by supporting processes such as maintaining network inventory, provisioning services, configuring network components, and managing faults.

  Apart from that, the complementary term Business Support Systems or BSS is a newer term and typically refers to "business systems" dealing with customers, supporting processes such as taking orders, processing bills, and collecting payments. The two systems together are often abbreviated as BSS/OSS.

  Nevertheless in general, an OSS covers at least the application areas:

  - Network Management Systems
  - Service Delivery
  - Service Fulfillment, including the Network Inventory, Activation and Provisioning
  - Service Assurance
  - Customer Care
  - Billing

2.1.2 GSM Service – Short Messaging System (SMS)

In this assessment, only the main GSM service which is the Short Message Service (SMS) will be discussed.

Short Messages is the most used data application on mobile phones. User can use mobile phones or other GSM related devices to SMS text message to other user. The messages are usually sent from mobile devices via the Short Message Service Centre (SMSC) using the MAP protocol.
Basically, SMSC is a central routing hub for Short Messages. Many mobile service operators use their SMSCs as gateways to external systems, including the Internet, incoming SMS news feeds, and other mobile operators.

SMSC functioned by practicing "store and forward" mechanism. Whenever a message is sent but the recipient is not achievable the SMSC queues the message for later retry. However, there are some SMSCs provide a "forward and forget" option where transmission is tried only once.

2.1.3 Cellular Radio Network

Mobile phones can connect to GSM (cellular network) via cells in the immediate vicinity which each cells vary in coverage area by according to the implantation environment. Basically, there are five different cell sizes in the GSM network.

- Macrocells are cells where the base station antenna is installed on a mast or a building above average roof top level
- Micro cells are cells whose antenna height is usually under average roof top level and are typically used in urban areas.
- Picocells are small cells whose coverage diameter is a few dozen meters and mainly used in indoors.
- Femtocells are cells designed for use in residential or small business environments. It can connect to the service provider's network via a broadband internet connection.
- Umbrella cells are used to cover shadowed regions of smaller cells and fill in gaps in coverage between those cells.

2.1.4 Hayes Command Set (AT Commands)

The Hayes command set is a specific command-language originally developed for the Hayes Smartmodem 300 baud modem in 1977. The command set consists of a series of short text strings which combine together to produce complete commands for operations such as dialing, hanging up, and changing the parameters of the connection. Most dialup modems follow the specifications of the Hayes command set. Nowadays, it is also called as AT command sets.

2.2 Transistor – Transistor Logic (TTL)

TTL is digital circuits that built from bipolar junction transistors and resistors. It is called TTL because transistors perform function both functions as the logic gating and the amplifying. It is widely used in many applications such as computers, controls equipment, test equipment and instrumentation, electronics, synthesizers, and many more.

2.3 MAX232

The MAX232 is a type of integrated circuit that converts signals from an RS-232 serial port to signals suitable for use in TTL compatible digital logic circuits. It has dual driver and receiver that typically convert the RX, TX, CTS and RTS signals. It is helpful to understand what occurs to the voltage levels of the MAX232. Figure 2.2 illustrates the pinout of MAX232 IC and table 2.1 specifies the RS232 line types and logic levels and the corresponding TTL logic levels.

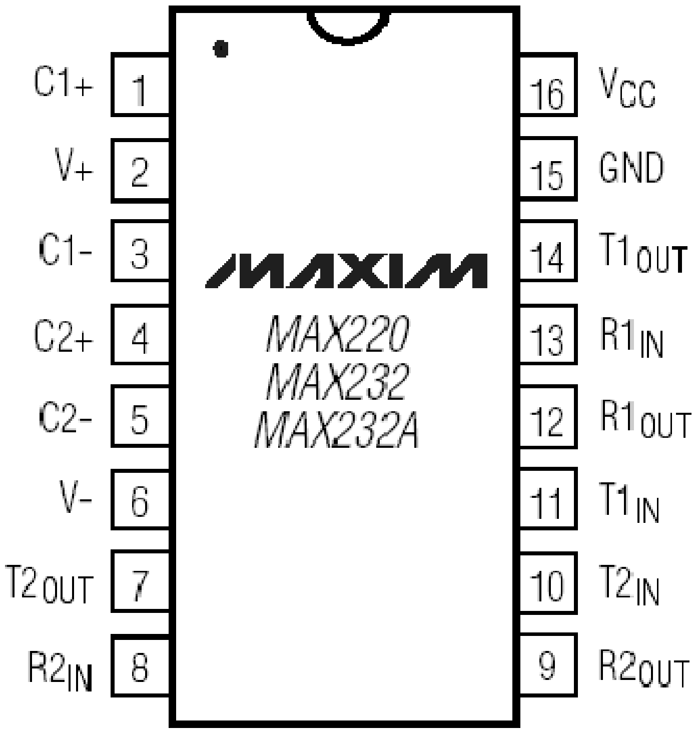


Figure 2.2: MAX232 IC

RS232 Line & Logic Level	**RS232 Voltage level**	**TTL Voltage to/from MAX232**
Data transmission (RX/TX) logic 0	+3V to +15V	0V
Data transmission (RX/TX) logic 1	-3V to -15V	5V
Control signals (RTS,CTS,DTR.DSR) logic 0	-3V to -15V	5V
Control signals (RTS,CTS,DTR.DSR) logic 1	+3V to +15V	0V

Table 2.1: RS232 Line types and Logic levels

2.4 Universal Asynchronous Receiver/ Transmitter (UART)

Western Digital made the first single-chip UART WD1402A around 1971. A universal asynchronous receiver/transmitter is a type of asynchronous receiver/transmitter that translates data between parallel and serial forms. UARTs are commonly used in conjunction with other communication standards such as RS-232.
Usually, UART is an individual integrated circuit used for serial communications throughout a computer or peripheral device serial port and it is commonly included in microcontrollers. As for dual UART (DUART), two UARTs will be combined into a single chip.

2.4.1 Transmitting and Receiving Serial Data

UART controller is the key component of the serial communications subsystem of a computer. The UART takes bytes of data and transmits the individual bits in a sequential trend. At the end of destination, a second

UART reassembles the bits into complete bytes. Serial transmission of digital information (bits) through a single wire or other medium is much more cost effective than parallel transmission through multiple wires. A UART is used to convert the transmitted information between its sequential and parallel form at each end of the link. Every single UART contains a shift register which is the fundamental method of conversion between serial and parallel forms.

Typically, UART usually does not directly generate or receive the external signals used between different items of equipment. Hence, separate interface devices are used to convert the logic level signals of the UART to and from the external signaling levels. External signals may be of many different forms and examples of standards for voltage signaling are RS-232, RS-422 and RS-485. For embedded systems communications, UARTs are commonly used with RS-232. Many chips provide UART functionality in silicon, and low-cost chips exist to convert logic level signals (such as TTL voltages) to RS-232 level signals (for example, Maxim's MAX232).

2.5 Recommended Standard 232 (RS232)

Basically, RS-232 is a standard in telecommunication especially in computer serial ports. It allows serial binary single ended data and control signal that connect in between data terminal equipment and a data circuit terminating equipment. Figure 2.3 illustrates the DB9 pinout and figure 2.4 shows the MAX232 interfacing circuitry.

2.5.1 Connector

Data Terminal Equipment (DTE) or Data Communication Equipment (DCE) define at each device which wires will be sending and receiving each signal. The standard recommended male connectors with DTE pin functions, and modems have female connectors with DCE pin functions.

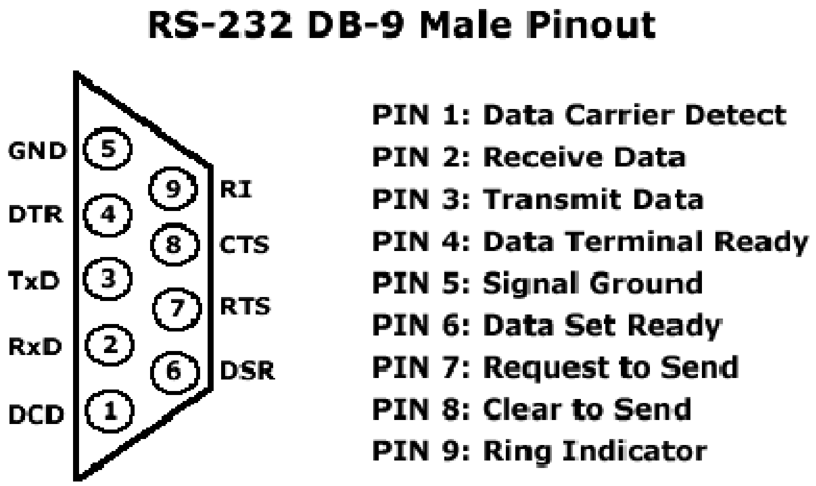


Figure 2.3: DB9 male connector pinout

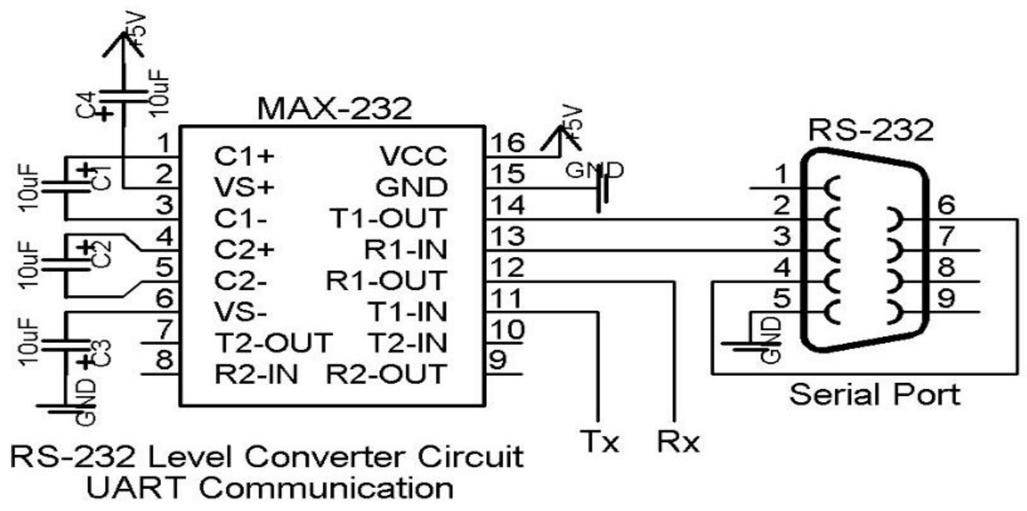


Figure 2.4 MAX232 connections with RS232 port and TTL RX-TX

2.5.2 Limitation of the Standard

- The power consumption increases due to large voltage swings and requirement for positive and negative supplies increases and hence leads to complicated power supply design.
- Due to the single-ended signaling referred to a common signal ground, it limits the noise immunity and transmission distance.
- Usually, multi-drop connection is not a normal practice that is predefined. Even with multi-drop is defined; there are limitation in speed and compatibility.
- The use of handshake lines for flow control is not really reliable in many other devices because it is initiated for setup and takedown of a dial-up communication circuit purposes.
- There is no method for power sending which is specified. There will be only a small amount of current will be extracted from DTR and RTS lines. Hence, it is only preferable for low power devices
- Nowadays laptops and computers are rarely found with an in built RS232 port and USB-to-RS232 converters fail to work in many scenarios.

2.6 Hyperterminal

HyperACCESS is the name for a number of successive computer communications software, made by Hilgraeve. In 1995, Hilgraeve licensed a low-end version of HyperACCESS, known as HyperTerminal to Microsoft for use in their set of communications utilities. Nowadays, Microsoft HyperTerminal is a small program that comes with Microsoft Windows. It can be used to send AT commands to your mobile phone or GSM/GPRS modem. By using some AT commands, some desired data from the GSM phone can be retrieved and analyzed. Some screenshots of hyperterminal are shown in figures 2.5 and 2.6.

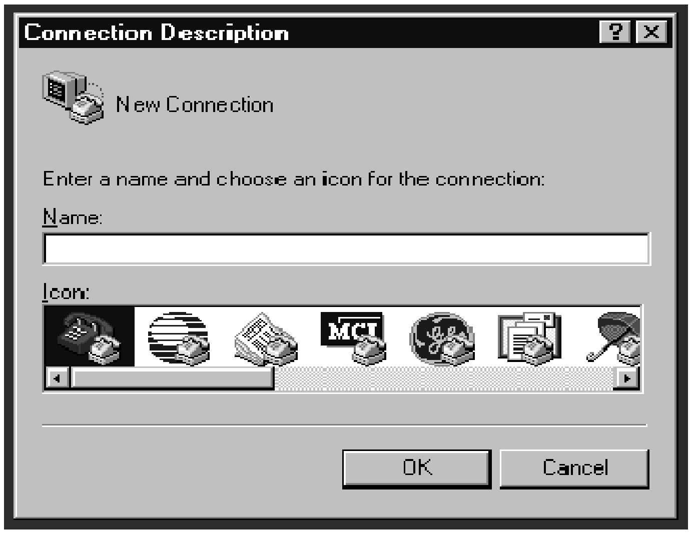
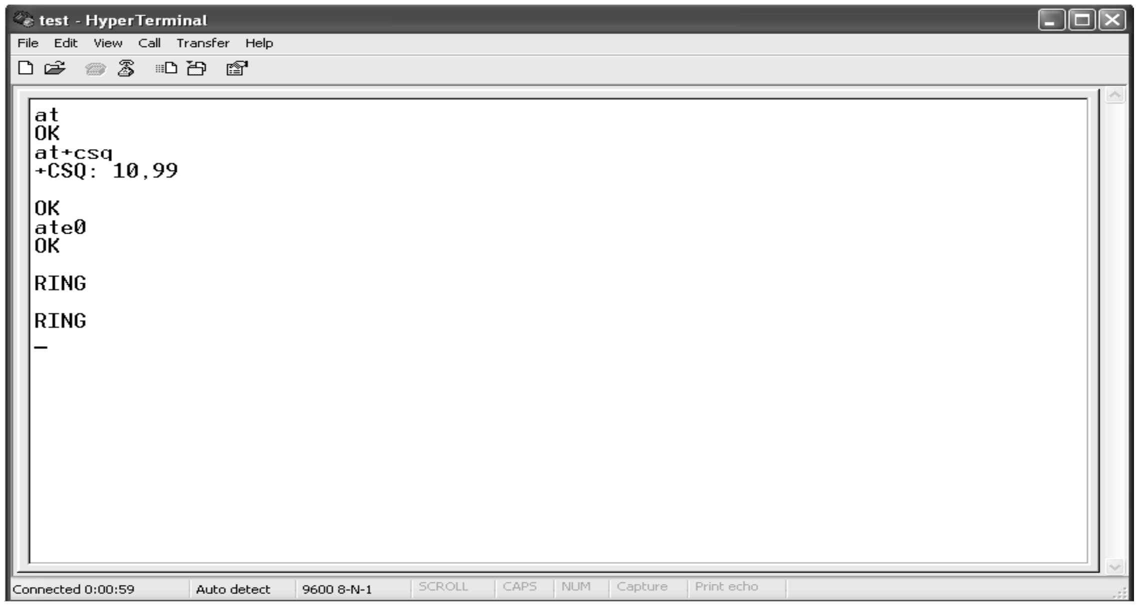


Figure 2.5: Screen Shot of Hyperterminal



```
at
OK
at+csq
+CSQ: 10,99

OK
ate0
OK

RING

RING
_
```

Connected 0:00:59 | Auto detect | 9600 8-N-1 | SCROLL | CAPS | NUM | Capture | Print echo

Figure 2.6: Screen shot of hyperterminal communicating with a GSM Modem using AT Commands

2.7 Universal Serial Bus (USB)

Universal Serial Bus (USB) is an industry standard developed in the mid-1990s that defines the cables, connectors and communications protocols used in a bus for connection, communication and power supply between computers and electronic devices.
USB was designed to standardize the connection of computer peripherals, such as keyboards, pointing devices, digital cameras, printers, portable media players, disk drives and network adapters to personal computers, both to communicate and to supply electric power. It has become commonplace on other devices, such as smartphones, PDAs and video game consoles.

2.7.1 USB System architecture

The design architecture of USB is asymmetrical in its topology, consisting of a host, a multitude of downstream USB ports, and multiple peripheral devices connected in a tiered-star topology. Additional USB hubs may be included in the tiers, allowing branching into a tree structure with up to five tier levels. A USB host may implement multiple host controllers and each host controller may provide one or more USB ports. Up to 127 devices, including hub devices if present may be connected to a single host controller.

USB devices are linked in series through hubs. One hub is known as the root hub which is built into the host controller.

A physical USB device may consist of several logical sub-devices that are referred to as device functions. A single device may provide several functions, for example, a webcam (video device function) with a built-in microphone (audio device function). This kind of device is called composite device. An alternative for this is compound device in which each logical device is assigned a distinctive address by the host and all logical devices are connected to a built-in hub to which the physical USB wire is connected.

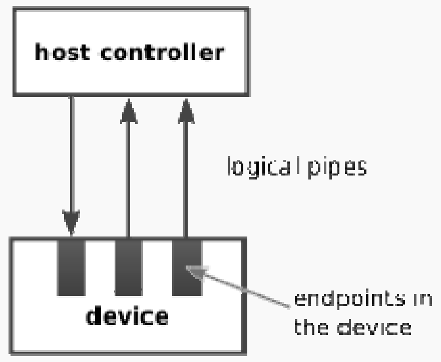


Figure 2.7: USB Device endpoints and logical pipes

USB device communication is based on pipes (logical channels). A pipe is a connection from the host controller to a logical entity, found on a device, and named an endpoint (shown in figure 2.7). Because pipes correspond 1-to-1 to endpoints, the terms are sometimes used interchangeably. A USB device can have up to 32 endpoints: 16 into the host controller and 16 out of the host controller. The USB standard reserves one endpoint of each type, leaving a theoretical maximum of 30 for normal use. USB devices seldom have this many endpoints.

There are two types of pipes: stream and message pipes depending on the type of data transfer.

- *isochronous transfers:* at some guaranteed data rate (often, but not necessarily, as fast as possible) but with possible data loss (e.g., realtime audio or video).
- *interrupt transfers:* devices that need guaranteed quick responses (bounded latency) (e.g., pointing devices and keyboards).
- *bulk transfers:* large sporadic transfers using all remaining available bandwidth, but with no guarantees on bandwidth or latency (e.g., file transfers).
- *control transfers:* typically used for short, simple commands to the device, and a status response, used, for example, by the bus control pipe number 0.

When a USB device is first connected to a USB host, the USB device enumeration process is started. The enumeration starts by sending a reset signal to the USB device. The data rate of the USB device is determined during the reset signaling. After reset, the USB device's information is read by the host and the device is assigned a unique 7-bit address. If the device is supported by the host, the device drivers needed for communicating with the device are loaded and the device is set to a configured state. If the USB host is restarted, the enumeration process is repeated for all connected devices.

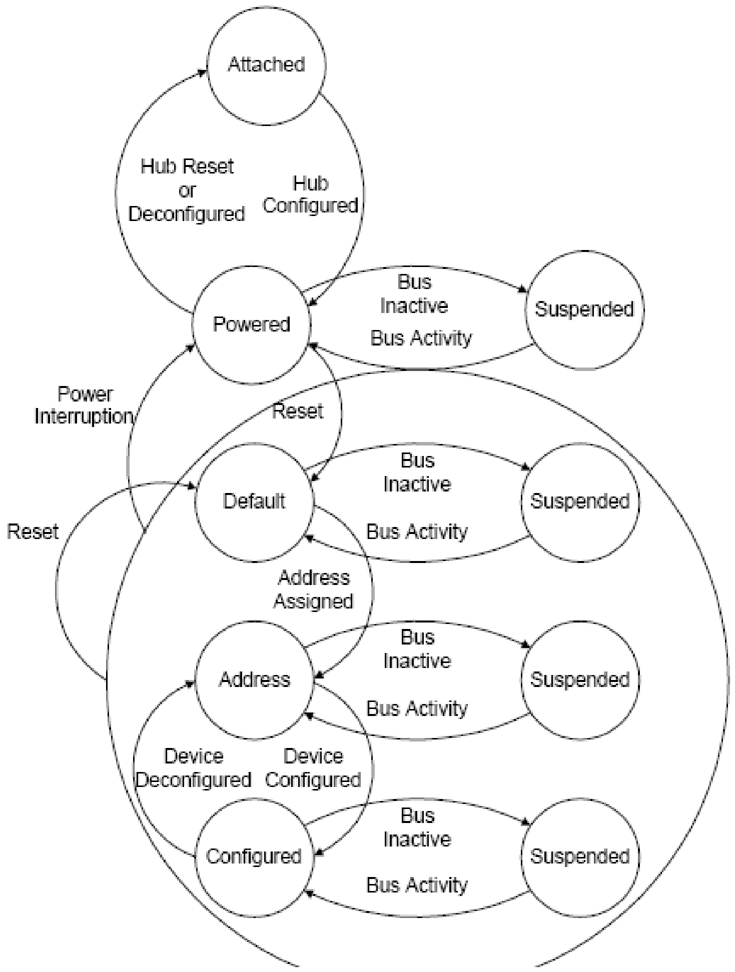


Figure 2.8: USB Device state diagram

2.7.2 USB Device classes

The functionality of USB devices is defined by class codes, communicated to the USB host to effect the loading of suitable software driver modules for each connected device. This provides for adaptability and device independence of the host to support new devices from different manufacturers.
Table 2.2 lays out the various USB device classes.

Class	Usage	Description	Examples, or exception
00h	Device	Unspecified	Device class is unspecified, interface descriptors are used to determine needed drivers
01h	Interface	Audio	Speaker, microphone, sound card, MIDI
02h	Both	Communications and CDC Control	Modem, Ethernet adapter, Wi-Fi adapter
03h	Interface	Human interface device (HID)	Keyboard, mouse, joystick

Class	Usage	Description	Examples, or exception
05h	Interface	Physical Interface Device (PID)	Force feedback joystick
06h	Interface	Image	Webcam, scanner
07h	Interface	Printer	Laser printer, inkjet printer, CNC machine
08h	Interface	Mass storage	USB flash drive, memory card reader, digital audio player, digital camera, external drive
09h	Device	USB hub	Full bandwidth hub
0Ah	Interface	CDC-Data	Used together with class 02h: communications and CDC control
0Bh	Interface	Smart Card	USB smart card reader
0Dh	Interface	Content security	Fingerprint reader
0Eh	Interface	Video	Webcam
0Fh	Interface	Personal Healthcare	Pulse monitor (watch)
DCh	Both	Diagnostic Device	USB compliance testing device
E0h	Interface	Wireless Controller	Bluetooth adapter, Microsoft RNDIS
EFh	Both	Miscellaneous	ActiveSync device
FEh	Interface	Application-specific	IrDA Bridge, Test & Measurement Class (USBTMC), USB DFU (Direct Firmware update)
FFh	Both	Vendor-specific	Indicates that a device needs vendor specific drivers

Table 2.2: USB Device classes

2.7.3 USB Device framework

USB Device framework is one of the most important sections from an USB Device implementer’s point of view. Although an extensive discussion about the framework is beyond the scope of this report, USB Device descriptors and USB device requests are discussed here.

A USB device may be divided into three layers:
• The bottom layer is a bus interface that transmits and receives packets.

• The middle layer handles routing data between the bus interface and various endpoints on the device. An endpoint is the ultimate consumer or provider of data. It may be thought of as a source or sink for data.

• The top layer is the functionality provided by the serial bus device, for instance, a mouse or ISDN interface.

This section describes the common attributes and operations of the middle layer of a USB device. These attributes and operations are used by the function-specific portions of the device to communicate through the bus interface and ultimately with the host.

2.7.3.1 USB Device requests

All USB devices respond to requests from the host on the device's Default Control Pipe. These requests are made using control transfers. The request and the request's parameters are sent to the device in the Setup packet. The host is responsible for establishing the values passed in the fields listed in Table 2.3. Every Setup packet has eight bytes.

Offset	Field	Size	Value	Description
0	bmRequestType	1	bitmap	Characteristics of request: D7: Data transfer direction 0 = Host-to-device 1 = Device-to-host D6...5: Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved D4...0: Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4...31 = Reserved
1	bRequest	1	Value	Specific request (refer table 2.4)
2	wValue	2	Value	Word-sized field that varies according to Request
4	wIndex	2	Index or offset	Word-sized field that varies according to request; typically used to pass an index or offset
6	wLength	2	Count	Number of bytes to transfer if there is a Data stage

Table 2.3: USB Device request fields

2.7.3.2 USB Standard requests

Table 2.4 describes the standard device requests defined for all USB devices. Table 2.5 and Table 2.6 give the standard request codes and descriptor types, respectively.

USB devices must respond to standard device requests, even if the device has not yet been assigned an address or has not been configured.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B 00000001B 00000010B	CLEAR_FEATURE	Feature Selector	Zero Interface Endpoint	Zero	None
10000000B	GET_CONFIGURATION	Zero	Zero	One	Configuration Value
10000000B	GET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID	Descriptor Length	Descriptor
10000001B	GET_INTERFACE	Zero	Interface	One	Alternate Interface
10000000B 10000001B 10000010B	GET_STATUS	Zero	Zero Interface Endpoint	Two	Device, Interface, or Endpoint Status
00000000B	SET_ADDRESS	Device Address	Zero	Zero	None
00000000B	SET_CONFIGURATION	Configuration Value	Zero	Zero	None
00000000B	SET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID	Descriptor Length	Descriptor
00000000B 00000001B 00000010B	SET_FEATURE	Feature Selector	Zero Interface Endpoint	Zero	None
00000001B	SET_INTERFACE	Alternate Setting	Interface	Zero	None
10000010B	SYNCH_FRAME	Zero	Endpoint	Two	Frame Number

Table 2.4: Standard device requests

bRequest	Value
GET_STATUS	0
CLEAR_FEATURE	1
Reserved for future use	2
SET_FEATURE	3
Reserved for future use	4
SET_ADDRESS	5
GET_DESCRIPTOR	6
SET_DESCRIPTOR	7
GET_CONFIGURATION	8
SET_CONFIGURATION	9
GET_INTERFACE	10
SET_INTERFACE	11
SYNCH_FRAME	12

Table 2.5: Standard request codes

Descriptor Types	Value
DEVICE	1
CONFIGURATION	2
STRING	3
INTERFACE	4
ENDPOINT	5
DEVICE_QUALIFIER	6
OTHER_SPEED_CONFIGURATION	7
INTERFACE_POWER[1]	8

Table 2.6: Descriptor types

2.7.4 USB Connectors

The connectors specified by the USB committee were designed to support a number of USB's underlying goals, and to reflect lessons learned from the menagerie of connectors which have been used in the computer industry. The connector mounted on the host or device is called the receptacle, and the connector attached to the cable is called the plug.

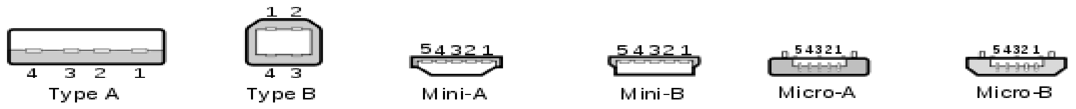


Figure 2.7: Various USB Connectors

Pin	Name	Cable color	Description
1	VBUS	Red	+5 V
2	D–	White	Data –
3	D+	Green	Data +
4	GND	Black	Ground

Table 2.8: USB 1.x/2.x standard pinout

Pin	Name	Cable color	Description
1	VBUS	Red	+5 V
2	D–	White	Data –
3	D+	Green	Data +
4	ID	None	Permits distinction of host connection from slave connection * host: connected to Signal ground * slave: not connected
5	GND	Black	Signal ground

Table 2.9: USB 1.x/2.x mini/micro pinout

Figure 2.7 shows the various types of USB connectors while tables 2.8 and 2.9 tabulate the pinout description for USB standard and mini/micro connectors respectively.

CHAPTER 3

METHODOLOGY

3.1 HARDWARE DESIGN AND ARCHITECTURE

3.1.1 Block Diagram of the system

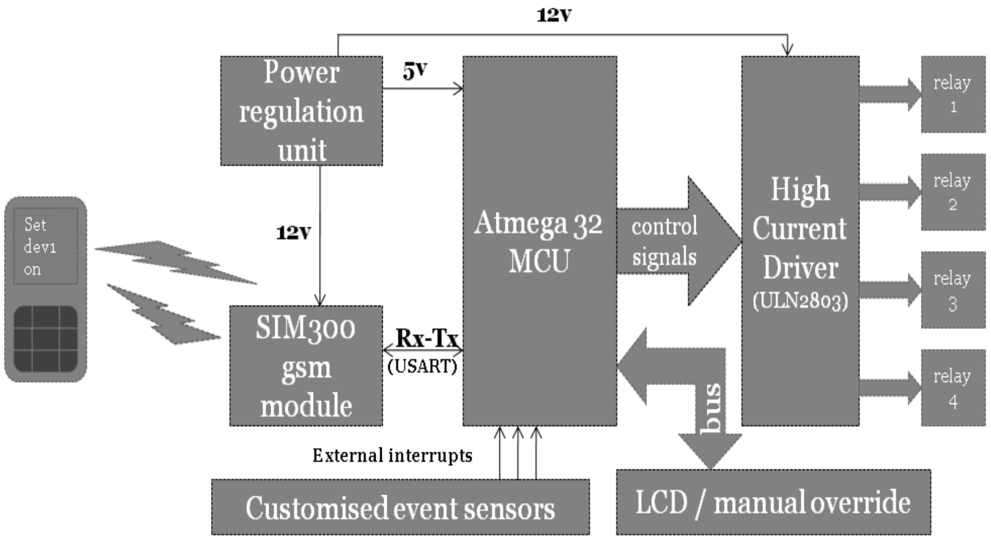


Figure 3.1: Hardware block diagram

For the hardware schematic and the PCB layout please refer diagrams x and y in Appendix C.

3.1.2 Component Description

3.1.2.1 Sim300 GSM Modem

Designed for global market, SIM300 is a Tri-band GSM/GPRS engine that works on frequencies EGSM 900 MHz, DCS 1800 MHz and PCS1900 MHz. Sim300 is the heart of the GSM communication subsystem in this system. The version of modem used in this system supports both RS232 DB9 serial port as well as TTL rx-tx port. This saves us from the trouble of TTL<->RS232 conversion using MAX232 and hence reduces the chip count of the system. Apart from this, the board also features a regulated 5v power supply which is subsequently used to power the MCU in the system. The modem needs an external 12v power supply. Figure 3.2 shows the Sim300 GSM module.

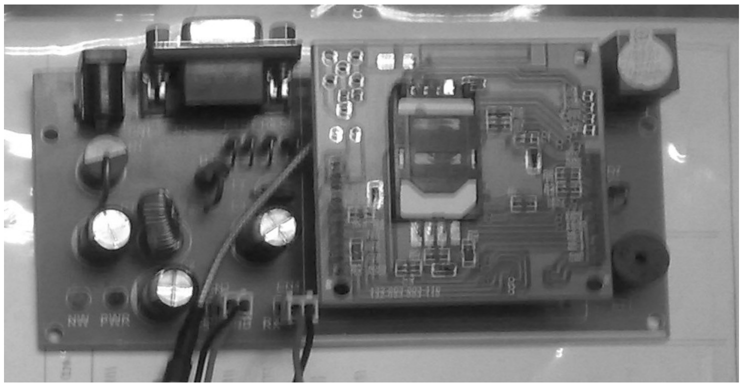


Figure 3.2: Sim300 GSM modem board

3.1.2.2 Atmega32

The ATmega32 is a low-power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture from Atmel Corporation. By executing powerful instructions in a single clock cycle, the ATmega32 achieves throughputs approaching 1 MIPS per MHz allowing the system designer to optimize power consumption versus processing speed. The ATmega32 is responsible for almost all processing done by the system from parsing incoming commands to sending back proper response. The Atmega32 features the following in a single chip package:

- 32KB programmable flash
- 1KB EEPROM
- 1KB SRAM
- 32 General purpose IO ports many which can be used to perform special functions
- Supports In-System programmability (ISP)
- 2 8bit Timers/counters, 1 16bit timer/counter
- Internal 8 channel 10 bit ADC, comparators and many advanced features

Besides the above mentioned features, AVR based MCUs from Atmel has got a wide support for development tools including IDEs, compilers, burners and libraries. To mention a few added advantages that is availed by using an AVR are:

- Used Obdev's open source USB library for AVR V-USB
- Used open source WinAVR C compiler
- Used Fischl's open hard usbASP programmer
- Used Atmel's free AVR STUDIO 4.0 IDE

So it is quite evident that using an AVR brings down the development cost of an embedded project.

The pinout of 40pin DIP of ATmega32 is given in figure 3.3.

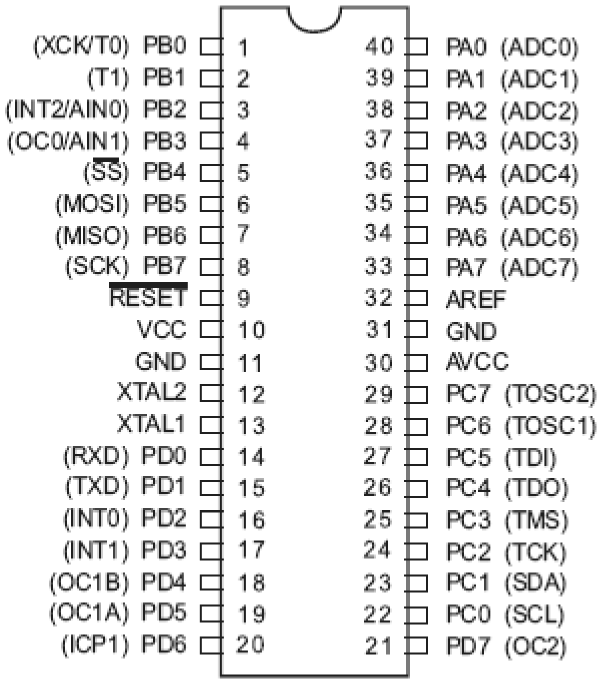


Figure 3.3: ATmega32 DIP Pinout

3.1.2.3 ULN2003A/ULN2803

TTL signals from a microcontroller or other digital IC are strong enough to drive LEDs but cannot drive higher loads such as motors or relays. To drive higher loads from a TTL signal, a driver circuit is needed. The current system under discussion contains four relays which can be interfaced with any other electrical appliances. For these relays to work in response to the TTL signals from the microcontroller we need ULN2003A or ULN2803 darlington pair high current driver. These ICs internally consists of a number of darlington pairs (a configuration of two transistors with increased amplification power) which is 7 in case of ULN2003A and 8 in case of ULN2803. These ICs receive TTL signals as input and drive the corresponding relays accordingly but requires an external 12v supply for the relays.

3.1.2.4 16x2 LCD

LCDs are common display devices on almost every electronics products. LCDs are sleek cheap and are used to display crisp text and graphics and are used ubiquitously. A variety of LCDs are available from black and white text LCDs to color graphics LCDs. The LCD that is used here, a 16x2 LCD is a standard text LCD with two rows and 16 columns each row. It displays black text on a green background and utilizes the extremely common HD44780 parallel interface chipset. Interfacing these LCDs in small electronics projects is extremely cost effective and easy. The LCD is used as a monitoring tool during development and testing phase, later during operational phase it displays current status information of the system. The controller HD44780 has a specific command set for configuring the LCD and displaying text, which is given in table 3.1.

Sno	Instruction	Hex	Decimal
1	Function Set: 8-bit, 1 Line, 5x7 Dots	0x30	48
2	Function Set: 8-bit, 2 Line, 5x7 Dots	0x38	56
3	Function Set: 4-bit, 1 Line, 5x7 Dots	0x20	32
4	Function Set: 4-bit, 2 Line, 5x7 Dots	0x28	40
5	Entry Mode	0x06	6
6	Display off Cursor off (clearing display without clearing DDRAM content)	0x08	8
7	Display on Cursor on	0x0E	14
8	Display on Cursor off	0x0C	12
9	Display on Cursor blinking	0x0F	15
10	Shift entire display left	0x18	24
11	Shift entire display right	0x1C	30
12	Move cursor left by one character	0x10	16
13	Move cursor right by one character	0x14	20
14	Clear Display (also clear DDRAM content)	0x01	1

Table 3.1: 16x2 LCD Command subset

3.1.2.5 Miscellaneous

- Electrolytic Capacitors: 100uf, 1000uf
- Ceramic capacitors: 22pf, .1ufd
- Zener diode: 3.6v
- Resistors: 1k ohm, 68ohm
- Diodes: ln4007
- LEDs
- Transformer: 15-0-15 V, 2A

3.2 SOFTWARE DESIGN AND ARCHITECTURE

The software is the most intricate and complex part of the project, hence to ensure simplicity and reliable operation, the software is designed using a very modular manner. Roughly speaking the system consists of two major software components, which are

- *Firmware:* The program or software that is running onboard the microcontroller and is responsible for managing and proper functioning of every piece of hardware it is attached to. The firmware is there to initialize the GSM modem, LCD during system startup, parsing the SMS commands as they arrive, executing the command and sending back proper response.
- *PC based configuration software:* PC based configuration software is used by the user to re-configure the device so that it can be attached to new devices or some previously assigned devices can be unplugged or changed. This software also enables the user to change the cell phone number, which the system uses as an authentication mechanism when it receives a command.

3.2.1 Firmware Architectural model

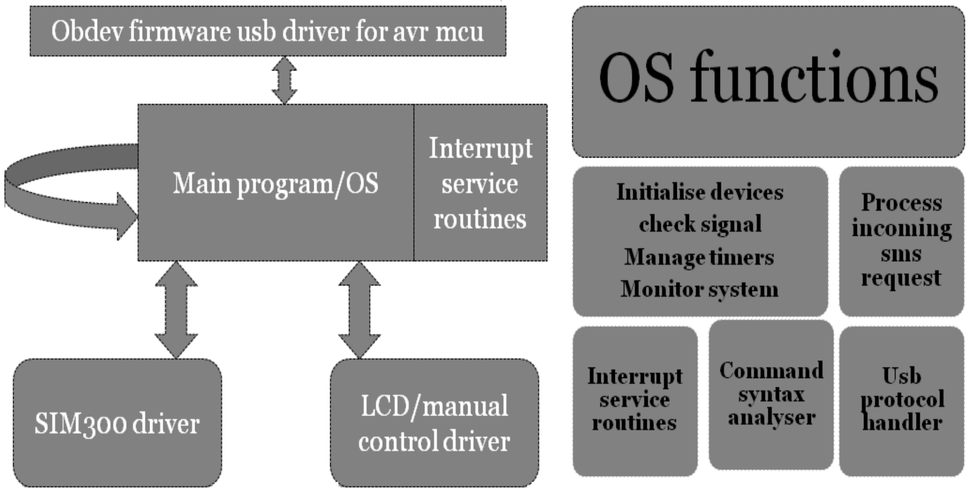

Obdev firmware usb driver for avr mcu
Main program/OS
Interrupt service routines
SIM300 driver
LCD/manual control driver
OS functions
Initialise devices
check signal
Manage timers
Monitor system
Process incoming sms request
Interrupt service routines
Command syntax analyser
Usb protocol handler

Figure 3.4: Firmware architectural model

Due to complex nature of the firmware, an attempt has been made to implement it in the lines of a simple Real Time Operating System (RTOS). The entire firmware has been highly modularized into independent modules as seen in figure 3.4.

By breaking the entire firmware into smaller modules, has led to a framework. The usefulness of this framework based approach is that, if in future another developer tries to implement another system using similar hardware but with a different objective, he will be exempted from the knitty gritty low level details about how to establish a communication with the GSM module, what kind of data structures to use to hold incoming message and memory management. He instead will focus on his application specific logic.

A discussion about the various modules and how are they glued together, will make things a little bit clearer.

3.2.1.1 Sim300 Driver

One of the most complex parts of this project is the communication between the system and the Sim300 GSM module. Although the underlying communication uses a standard UART, a trivial matter, the complexity lies with how to store incoming message in a way such that it can be later retrieved and processed with ease. As when an SMS may arrive cannot be predicted, maintaining specific message borders are very important. Also the same UART line may be used for other purposes such as checking signal strength; sending messages which will have network generated responses, it is very important to identify valid messages and dispatch them to the proper processing function and isolate other data. All these intricacies are abstracted in the sim300 driver. The driver is implemented as a common C header file (sim300.h) which consists of a bunch of function, data structures and proper logic to dispatch valid messages to an extern function which will be implemented by the application programmer using the driver in his project. The developer just expects the message data in a predefined buffer.

Refer the actual code of sim300.h in Appendix C.

3.2.1.2 LCD Driver

The LCD driver is also implemented as a common C header file (lcd.h), and consists of initialization codes, functions and data structure to operate the LCD such as to clear the screen, writing a character, writing a string, scrolling etc. the syntax of using the lcd is very familiar to using the fprintf() function in C. e.g. To write a string all the programmer needs to do is, lcdWriteString(lcdBuff,"Hello World"); where lcdBuff is the lcd buffer which needs to be declared before using any lcd function.

Refer the actual code of lcd.h in Appendix C.

3.2.1.3 Obdev's V-USB firmware only USB driver

V-USB is Obdev's open source firmware only USB driver for AVR microcontroller family. V-USB also follows a framework based architecture, and the previous two discussed driver modules draws its inspiration from this. Prior using V-USB in any project it is required to modify the usbdrv/usbconfig-prototype.h in the project folder as per requirement and rename it to usbdrv/usbconfig.h. This file contains relevant information regarding hardware configuration as well as the device descriptor for the device being implemented with USB functionality.

Refer the usbconfig.h in Appendix C.

3.2.1.4 Coding structure

This section will discuss briefly the coding structure, and how to integrate the above mentioned modules into a single coherent program. The Entire code is listed in Appendix C as gsm_control_with_usb.c

```
/*include all the module header files in this section*/
/*make sure all the mentioned headers are in the project directory*/

#include "sim300.h"   //sim300 driver module header
#include "lcd.h"                 //lcd driver module header
#include "usbdrv/usbdrv.h"       //V-USB header
#include <util/delay.h>

void        usb();               //usb related code

void        controller();        //controller related code

lcdline     lcdBuff[2];//declare the lcd buffer here if lcd is intended to be used in all functions


int main()
{
 /*check necessary condition to operate in usb mode or controller mode*/

            If(CONDITION_IS_USB)
```

```
          {
                    usb();
          }
          Else
          {
                    Controller();
          }
}

void controller()
{
          /*implementation of controller specific code here*/
          Int sigStrength = sim300Init(); //initialize sim300
          lcdInit();                //initialize lcd

          lcdWriteString(lcdBuff,"System Ready");

          for(;;)        //infinite loop
          {
                    delayMs(100);        //some delay
                    sigStrength=checkSignalStrength();
                    if(sigStrength>0)
                    {
                              lcdWriteString(lcdBuff,"\n\nSignal OK");
                    }
                    Else
                    {
                              lcdWriteString(lcdBuff,"\n\nNO SIGNAL");
                    }
          }
}

void processRequest()
{
 /*The extern function declared in sim300.h which the programmer needs to implement*/
 /*write application specific logic here. And expect required data in the array buffer*/
 /*This function is called by sim300.h upon successful reception of a message*/
}

void usb()
{
          uchar   i;
          lcdInit();   //initialize lcd
          usbInit();                //initialize V-USB driver
          usbDeviceDisconnect();  /* enforce re-enumeration, do this while interrupts are disabled! */
          i = 0;
          while(--i)
          {          /* fake USB disconnect for > 250 ms */
                    _delay_ms(1);
          }
          usbDeviceConnect();
          sei();
          for(;;)
          {             /* main event loop */
                    usbPoll();
                    if(!i)
                    {
                              lcdClearScreen(lcdBuff);
                              lcdWriteString(lcdBuff,"USB ready...");
                    }
                    i--;
                    _delay_ms(50);
          }
          return 0;
 }

uchar usbFunctionWrite(uchar *data, uchar len)
{
 /*this function is called in chunks of 8 bytes to transmit data from host--->device   (device write function)*/
}

uchar usbFunctionRead(uchar *data, uchar len)
{
 /*this function is called in chunks of 8 bytes to transmit data from device----->host (device read            function)*/
}
```

```
USB_PUBLIC usbMsgLen_t  usbFunctionSetup(uchar data[8])
{
  /*this function is called at the start of each control transfer, appropriate flag variables are set in this      function*/
}

/*a detailed study of the headers and usb specification is recommended for a in-depth understanding*/
```

3.2.2 PC Based Configuration software

The PC based configuration software is an extension to the original project idea, enabling users to reconfigure the device with the help of a PC through the USB interface. Before discussing about the software, we need to have at the USB specification yet again!

USB based devices are plug n' play enabled, which means all we need to do is just plug in the device to the USB port and the computer does the rest of the configuration for us automatically. Now if we look what goes on under the hood, we will see that once the USB device is plugged in it goes through a number of states and the USB enumeration process which is already discussed previously. At this point of time it is a little helpful to know a little bit more about the USB enumeration process.

During the USB enumeration process, the USB host (PC) probes the attached USB device and identifies which port it is connected to, powers it up and requests it device descriptor on its default control endpoint. The device descriptor of an USB device is a data structure containing many important information regarding the device class, power consumption, product name, vendor name, device endpoints and last but not the least the 16bit Vendor ID (VID) and Product ID (PID) of the device. The OS uses this VID-PID pair to locate and load the appropriate device driver for the device. VID-PID pairs are unique and they are required to be registered which takes a lot of money. Fortunately V-USB provides some VID-PID pairs for custom use and we could be sure that our devices will clash with no other devices except the ones that use V-USB firmware USB driver.

3.2.2.1 Libusb and LibusbJava

libusb is a suite of user-mode routines for controlling data transfer to and from USB devices on Unix-like systems without the need for kernel-mode drivers. libusb-win32 is a port of the USB library libusb 0.1 to the Microsoft Windows operating systems (Windows 2000, Windows XP, Windows Vista and Windows 7; Windows 98 SE and Windows ME for versions up to 0.1.12.2). The library allows user space applications to access many USB device on Windows in a generic way without writing any line of kernel driver code.

libusbJava is a Java wrapper for the libusb (0.1) and libusb-win32 USB library. For installation/usage instruction refer to libusbJava documentation.

3.2.2.2 Getting the device drivers

Under a Windows machine explicit device drivers (inf files) are required before proper functioning. Libusb-Win32 which is a port of the Libusb 0.1 for Microsoft windows operating system, comes with an inf-wizard. The exe can be found under libusb-win32-bin-1.2.5.0\bin\inf-wizard. After connecting the device the steps for building the inf files using the wizard is illustrated in the figures 3.6 to 3.8.

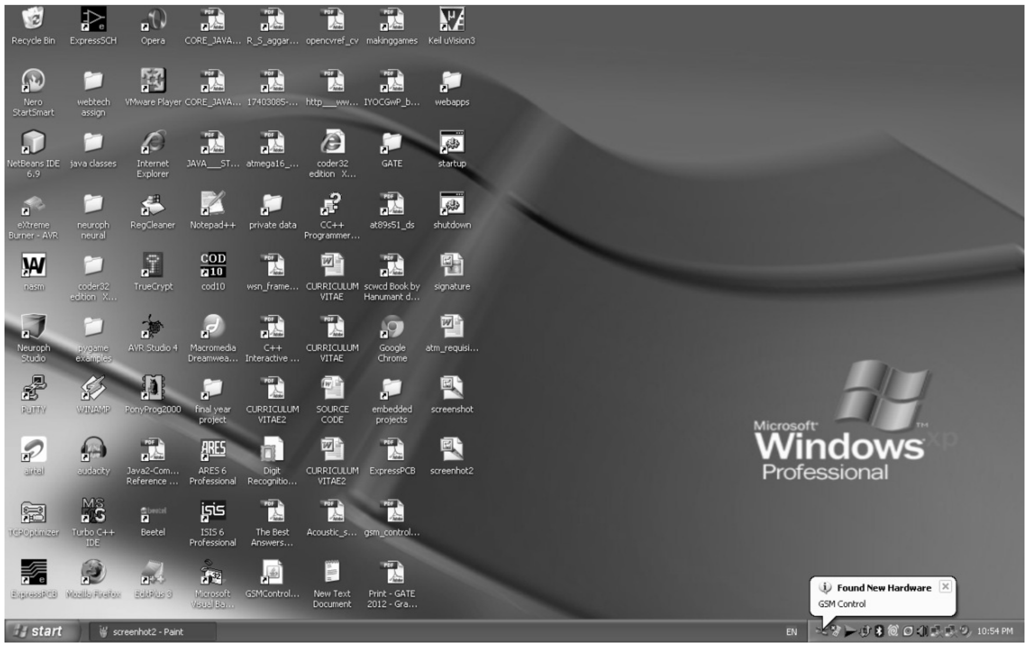
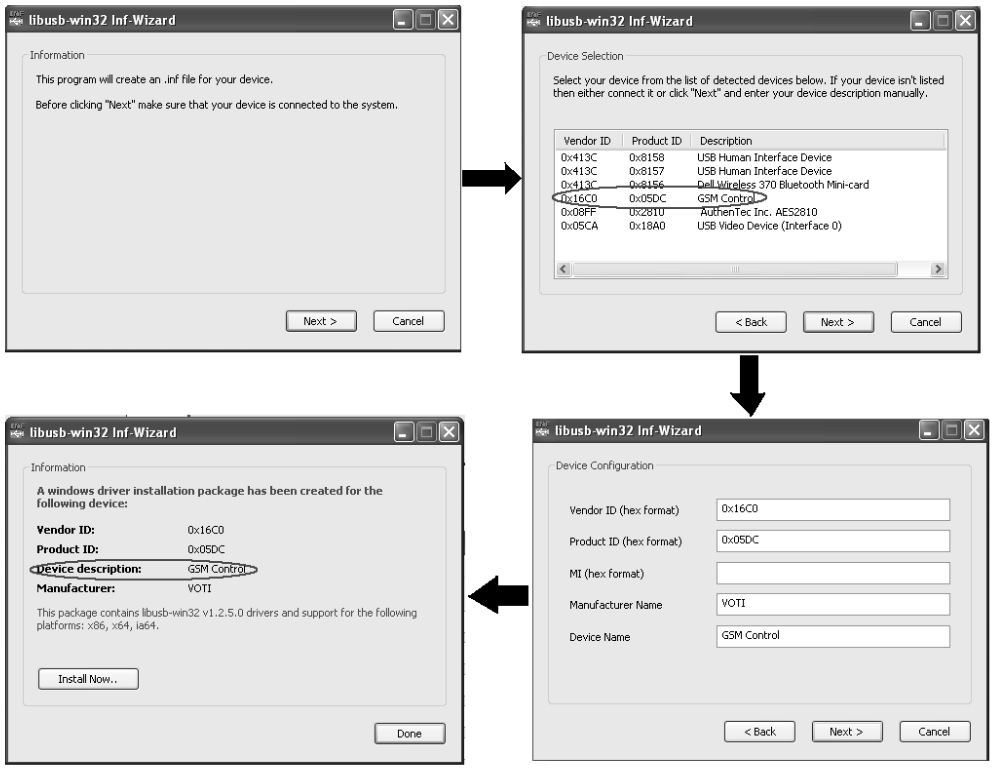


Figure 3.6: Found new hardware dialog after connecting the device



libusb-win32 Inf-Wizard

Information

This program will create an .inf file for your device.

Before clicking "Next" make sure that your device is connected to the system.

Next > | Cancel

libusb-win32 Inf-Wizard

Device Selection

Select your device from the list of detected devices below. If your device isn't listed then either connect it or click "Next" and enter your device description manually.

Vendor ID	Product ID	Description
0x413C	0x8158	USB Human Interface Device
0x413C	0x8157	USB Human Interface Device
0x413C	0x8156	Dell Wireless 370 Bluetooth Mini-card
0x16C0	0x05DC	GSM Control
0x08FF	0x2810	AuthenTec Inc. AES2810
0x05CA	0x18A0	USB Video Device (Interface 0)

< Back | Next > | Cancel

libusb-win32 Inf-Wizard

Device Configuration

Vendor ID (hex format)	0x16C0
Product ID (hex format)	0x05DC
MI (hex format)	
Manufacturer Name	VOTI
Device Name	GSM Control

< Back | Next > | Cancel

libusb-win32 Inf-Wizard

Information

A windows driver installation package has been created for the following device:

Vendor ID:	0x16C0
Product ID:	0x05DC
Device description:	GSM Control
Manufacturer:	VOTI

This package contains libusb-win32 v1.2.5.0 drivers and support for the following platforms: x86, x64, ia64.

Install Now.. | Done

Figure 3.7: Steps for creating the device driver using libusb-win32 inf-wizard

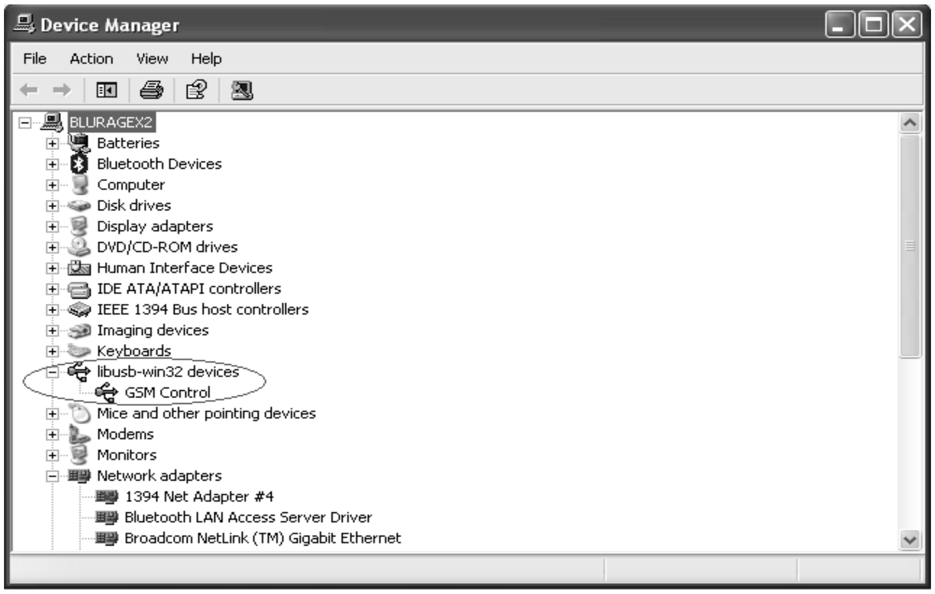


Figure 3.8: Device listed under device manager after proper driver installation

3.2.2.3 Command line configuration application in C

A few screenshots of the command line application written in C.

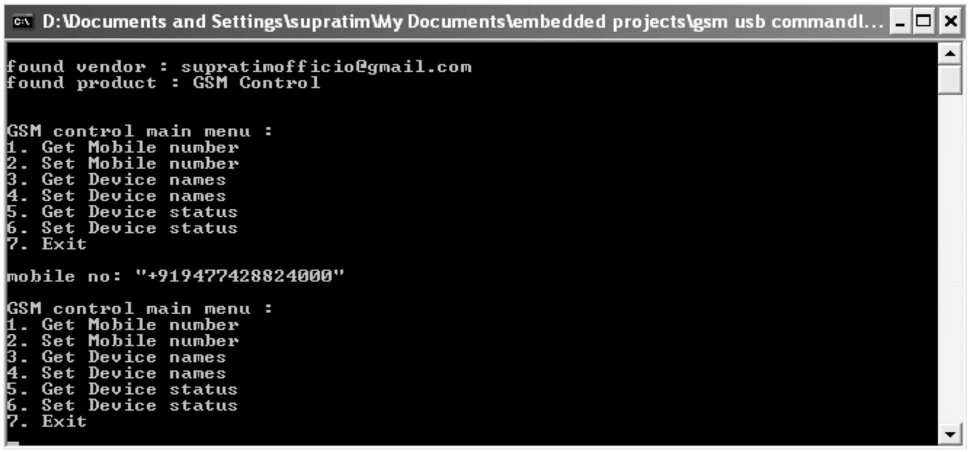

```
D:\Documents and Settings\supratim\My Documents\embedded projects\gsm usb commandl...
found vendor : supratimofficio@gmail.com
found product : GSM Control


GSM control main menu :
1. Get Mobile number
2. Set Mobile number
3. Get Device names
4. Set Device names
5. Get Device status
6. Set Device status
7. Exit

mobile no: "+919477428824000"

GSM control main menu :
1. Get Mobile number
2. Set Mobile number
3. Get Device names
4. Set Device names
5. Get Device status
6. Set Device status
7. Exit
```

Figure 3.9: C application screenshot1

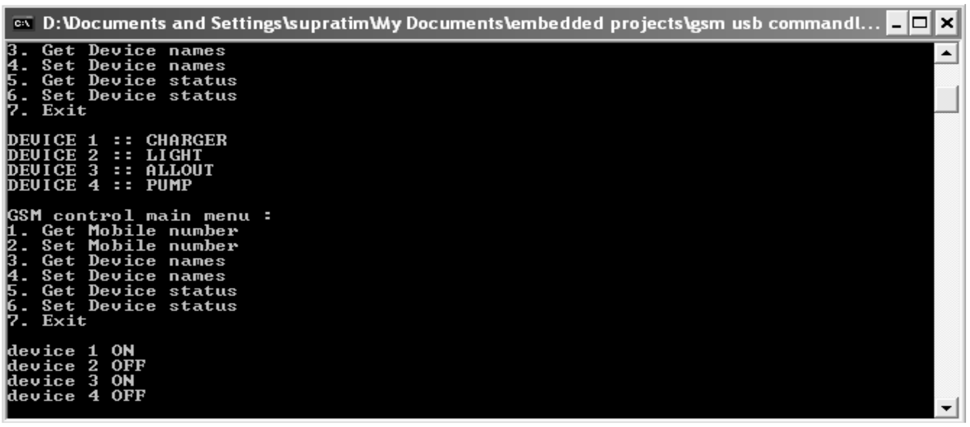
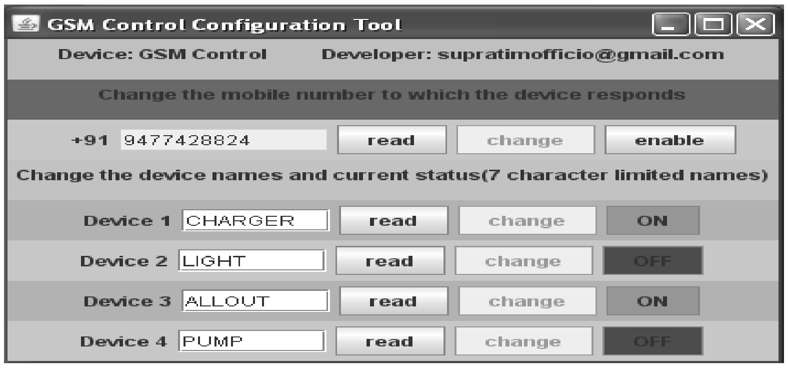

```
D:\Documents and Settings\supratim\My Documents\embedded projects\gsm usb commandl...
3. Get Device names
4. Set Device names
5. Get Device status
6. Set Device status
7. Exit

DEVICE 1 :: CHARGER
DEVICE 2 :: LIGHT
DEVICE 3 :: ALLOUT
DEVICE 4 :: PUMP

GSM control main menu :
1. Get Mobile number
2. Set Mobile number
3. Get Device names
4. Set Device names
5. Get Device status
6. Set Device status
7. Exit

device 1 ON
device 2 OFF
device 3 ON
device 4 OFF
```

Figure 3.10: C application screenshot2

Refer the code for the command line application in Appendix C, in commandline application section.

3.2.2.4 GUI based configuration application in JAVA

At a later stage, the configuration application was ported from command line based C to GUI based java swing application. LibusbJava was used for the USB communication API. The GUI screenshot is given in figure 3.11.



Device: GSM Control Developer: supratimofficio@gmail.com

Change the mobile number to which the device responds

+91 9477428824

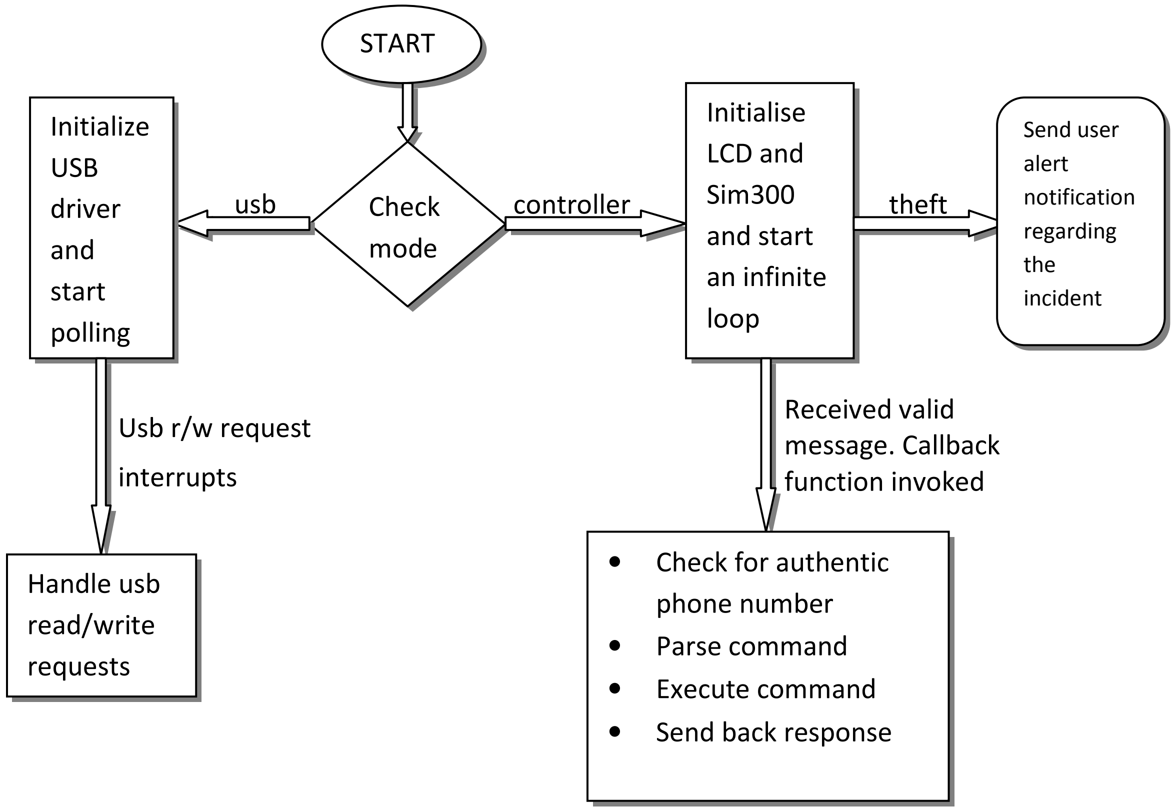
Change the device names and current status(7 character limited names)

Device 1	CHARGER	ON
Device 2	LIGHT	OFF
Device 3	ALLOUT	ON
Device 4	PUMP	OFF

Figure 3.11: Java swing based configuration application

Refer the code for the GUI application in Appendix C, in the GUI java application section

3.3 FLOWCHART


START
Check mode
usb
controller
Initialize USB driver and start polling
Usb r/w request interrupts
Handle usb read/write requests
Initialise LCD and Sim300 and start an infinite loop
theft
Send user alert notification regarding the incident
Received valid message. Callback function invoked
• Check for authentic phone number
• Parse command
• Execute command
• Send back response

CHAPTER 4

RESULTS AND DISCUSSIONS

4.1 Communication between the microcontroller and GSM module

The GSM module has an UART port besides the RS232 port, and the same is used for the communication with the UART of the microcontroller. The recommended baud rate for the GSM module was mentioned at 9600bps, but there were many errors in transmission at this rate. When the baud rate was set at 4800bps the transmission was stable. The two most significant reasons for the problem could be

- Oscillator characteristics
- Noise due to the radio transmissions from the modem, as the microcontroller and supporting electronics were housed very closely without any proper shielding

The next challenging task was to device a mechanism to retrieve messages, isolate message boundaries and call the proper function where the programmer is supposed to implement the application logic. To implement this framework, the modem is first initialized in the configuration as listed in table 4.1.

SL no	Mode/Operation	AT Command
1	Disable echo from modem	ATE0
2	Set Modem to Text Mode	AT+CMGF=1
3	Enable the automatic reception of SMS	AT+CNMI=1,2,0,0,0

Table 4.1: Modem configuration

The Receive complete interrupt is enabled on the UART of the microcontroller. When the modem is configured as stated above, whenever a SMS arrive it automatically redirects the SMS in text form to the microcontroller. The following steps outline the rest of the logic.

1. The Timer0 is set to generate overflow interrupt, initialized to 0 and not started yet
2. Upon the first receive complete interrupt, the timer is started and upon consequent receive complete interrupts the timer0 is reset to 0. Every character read is put in a predefined buffer.
3. When the timer0 overflows (happens when the entire message has been sent to microcontroller from the GSM module),
    - The status code at the beginning of the message buffer is checked to determine whether it is a SMS.
    - If the message buffer contains a SMS, the timer0 is reset and stopped, global interrupts are disabled and the function processRequest() is called which is defined as extern and implemented by the programmer in the main c file. When the function returns the buffer index is reset and global interrupts are again re-enabled
    - If the message buffer does not contain a SMS, the timer0 is reset and stopped, buffer index is reset and the ISR is exited

4.2 Authenticating mechanism and recovering from power failure

The system is designed with two useful features, which are

1. The system authenticates SMS commands from the mobile number it is being sent
2. In case of a power failure, the system remembers its last state and once the power is back, it resumes its previous state

The above two features use the internal EEPROM of the ATmega32. EEPROM unlike SRAM can hold data even after a power failure. So every time the system boots up, it reads the registered phone number and information regarding its last state from its EEPROM and loads in its working memory.

4.3 Power related problems

The system comprises of three distinct components that requires power. These are as follows

1. The GSM modem 12v regulated power
2. The Microcontroller and supporting circuitry 5v regulated power
3. The Relays 12v unregulated power

The GSM module board has a terminal for 12v regulated power, and the same is used to power the microcontroller. Alternatively the microcontroller can also be separately powered, and in that case a 7805 voltage regulator IC must be used to obtain a 5v regulated power and is highly recommended.

Many times in case of high loads, the microcontroller undergoes a brownout reset, and in many cases the microcontroller resets repeatedly. To prevent this from happening it is recommended to use a separate power supply for the microcontroller and to use a moderate capacitance (around 10uf) very near to the microcontroller power pins.

Initially a major problem was faced with the relays. It was observed that during system startup, when all the relays were in on state, there was a lot of chatter as the relays turned on and off rapidly in response to the sudden load. The problem could be easily bypassed by putting 100uf capacitance very close to the relay, to decrease the ripple effect, as shown in figure 4.1.

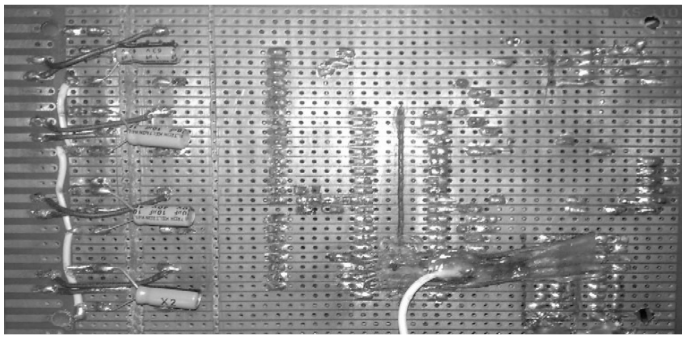
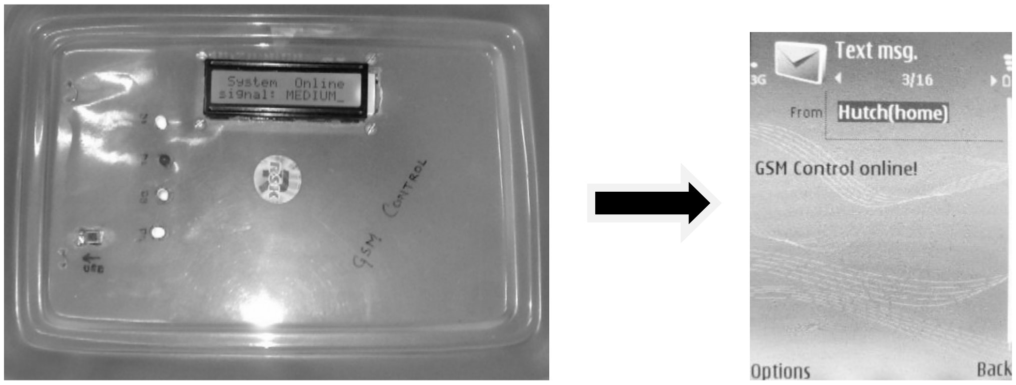


Figure 4.1: Capacitors placed very close to the relays

4.4 Results and Testing

The system was tested rigorously, and test results proved to be satisfactory. The system correctly responded to SMS commands and only processed commands sent from the registered phone number. Proper response SMS was also sent by the system. At times due to network problems there was a delay in response, as the sms got delivered late. Also if some SMS is undelivered and the same command is resent multiple times, unexpected responses occurred when some undelivered SMS got delivered. A few sequential screenshots of the system under operation is provided through figure 4.2 to figure 4.4 in the next page.

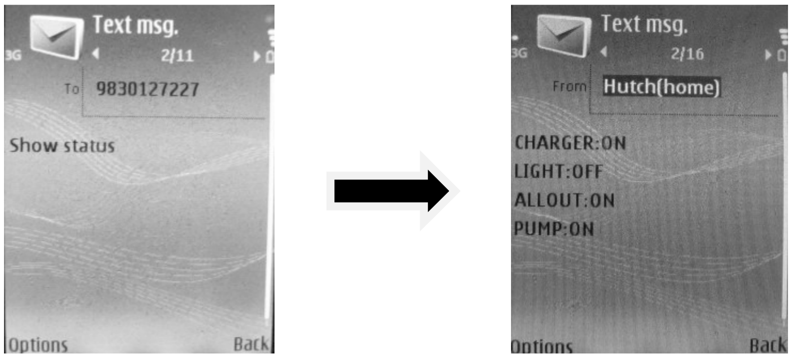


System Online
signal: MEDIUM_

From Hutch(home)

GSM Control online!

Figure 4.2: System turned on and response SMS from the system



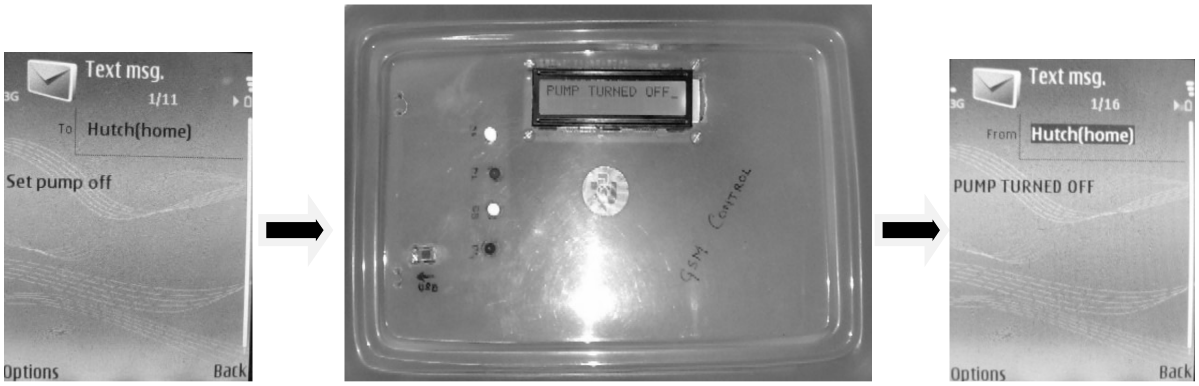
To 9830127227

Show status

From Hutch(home)

CHARGER:ON
LIGHT:OFF
ALLOUT:ON
PUMP:ON

Figure 4.3: Show status command and response



To Hutch(home)

Set pump off

PUMP TURNED OFF_

From Hutch(home)

PUMP TURNED OFF

Figure 4.4: Set pump off command and response

CHAPTER 5

CONCLUSION AND RECOMMENDATION

5.1 Conclusion

In conclusion, the Final Year Project is a success and was completed within the stipulated period of time. The system has been tested with real electrical appliances with performance as expected. Student, Supratim Das is able to achieve the objective whereby able to design a hardware by integrating several available electronic devices. The software framework based approach proved very effective enabling a smooth development, integration and testing of the project. The project also gave insights about USB devices, device drivers and writing codes for communicating with such devices

5.2 Potential

Nowadays, people are rushing for daily tasks and being busy, sometimes, they being forgetful and careless. These might be a dangerous scenario where electrical appliances such as microwave, heater and so on might not be turned off. This product allows user to turn off the house appliances through SMS. Apart from that it also helps to save energy where energy wasting can be prevented. Moreover, security issues are of concerned in current society. House break-in seems to terrify most of the house residents. This system can notify security guards and police officer whenever break-in occurs but without signify the burglar.

5.3 Limitation

The system is limited by the reliability of the GSM network. Sometimes due to network congestion problems, SMS may not get delivered properly either way. In such situation if the system is connected to critical electrical appliances like heaters, microwaves etc. may cause confusion and anxiety.

One other limitation is that, the security system is disabled in case of power failures, although this can be averted by connecting the device with some kind of UPS.

Lastly, integrating the system to a wide array of electrical appliances distributed at various points of a house would require major re-wiring.

5.4 Future recommendation

The system proposed in this report is based only on SMS. A much novel approach would be using GPRS/Internet connectivity, allowing other extended functionality like remote live video feeds, notifications in the form of email besides SMS, and a more user friendly and flexible access. Also with Internet connectivity a person can do much more than just turning things on and off, and would lead to online homes.

REFERENCES

Steven F. Barrett and Daniel J. Pack (2008), “Atmel AVR Microcontroller Primer: Programming and Interfacing”, Morgan & Claypool publishers.

Atmel Corporation, “Atmel atmega32/atmega32l datasheet”.

SIMCOM limited (2005), “SIM300 hardware specification”.

A. Alheraish, W. Alomar, and M. Abu-Al-Ela (2006), “Remote PLC system using GSM network with application to home security system”, 18th National Computer Conference.

Teo Che Shen (May 2011), “Cell phone controlled electrical appliances (GSM Controller)”

Hewlett Packard, Intel, Compaq, Lucent, Microsoft, NEC, Philips (April 27, 2000) “Universal Serial Bus specification”, Revision 2.0.

GSM Modems, Retrieved July 18, 2011, from http://www.nowsms.com/GSM%20Modems.htm

APPENDICES

APPENDIX A: COMPUTER PROGRAMME LISTING

WINAVR-20100110 AVR C Compiler for Win32 platform

AVR Studio 4.0

Proteus 6.0

eXtreme Burner-AVR 1.1

ExpressPCB 7.0.2

Netbeans 6.9

Microsoft Visual C 6.0

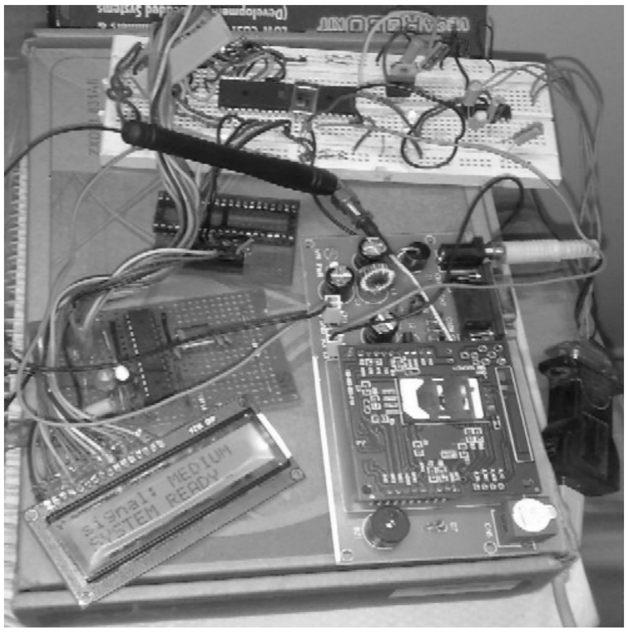
APPENDIX B: THIRD PARTY LIBRARY LISTING

V-USB 20100715

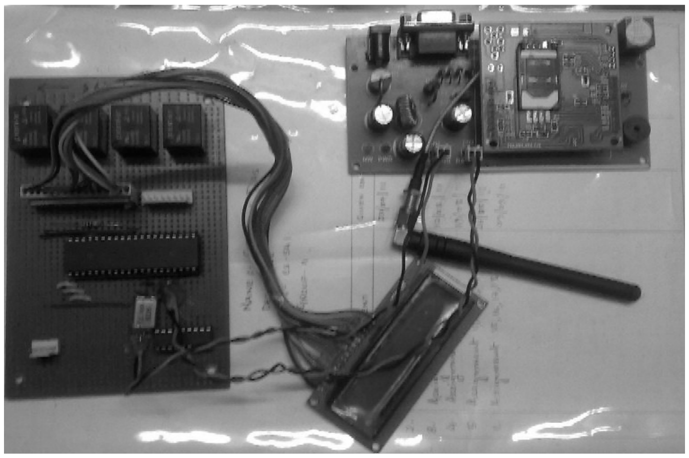
Libusb-win32 0.1

libusbJava

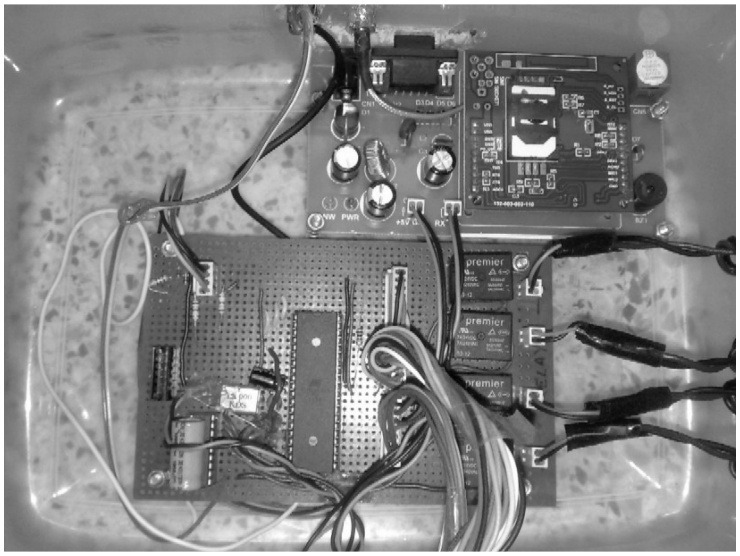
APPENDIX C: PROJECT PLANNING



Picture1: Circuit in Breadboard



Picture2: Partially completed circuit


Picture3: Fully completed circuit housed in a cabinet

Picture4: The system ready to be operational

DEVICE SCHEMATIC

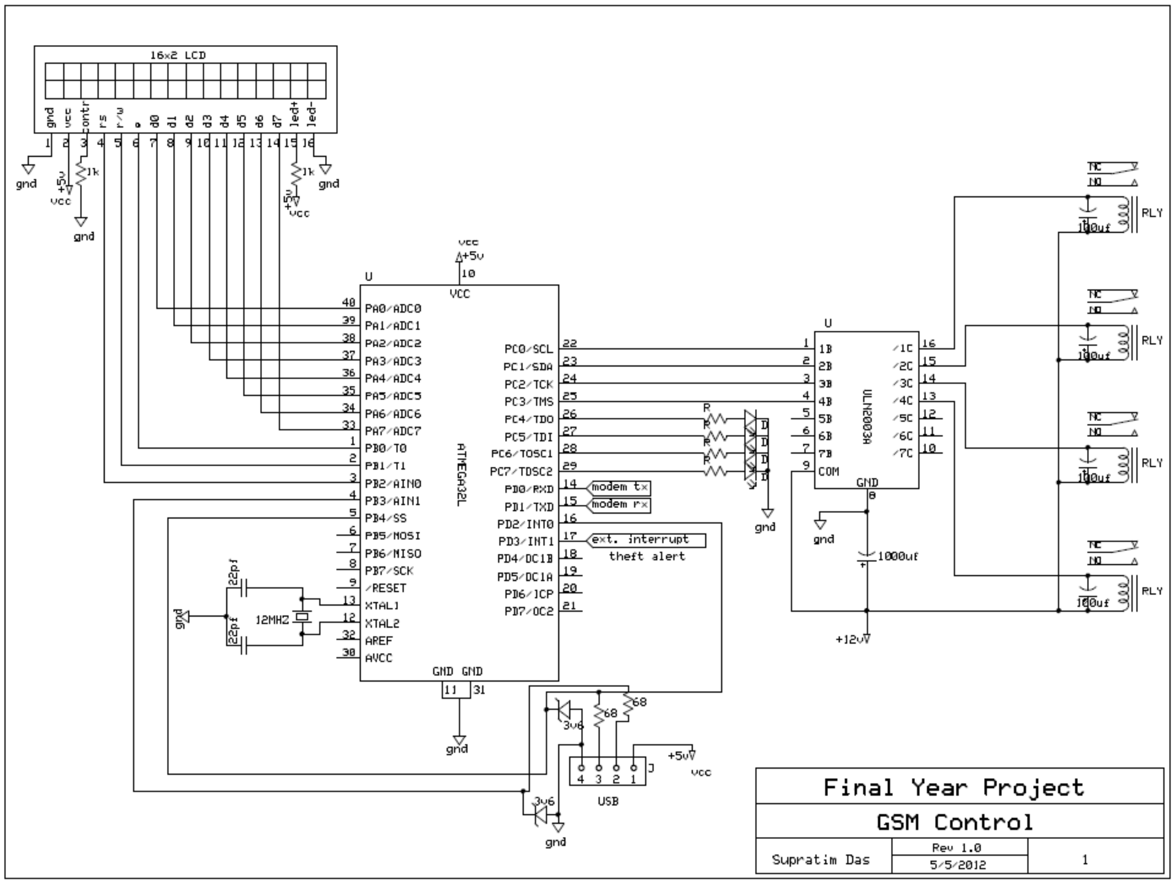


Diagram1: GSM Control Schematic

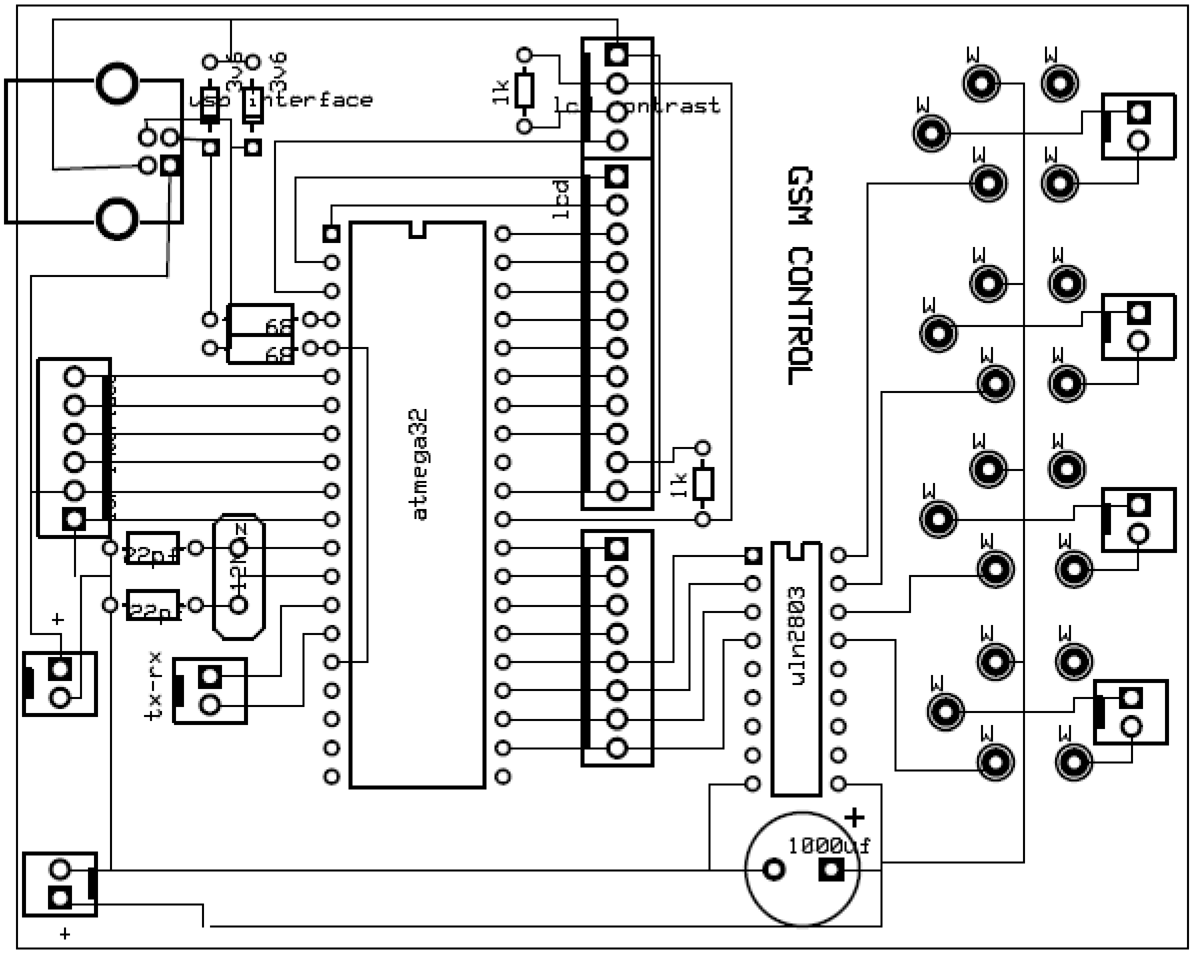


Diagram2: GSM Control PCB Design

FIRMWARE CODE LISTINGS

sim300.h

```
/*Author: Supratim Das
*Institute: Calcutta Institute of Engineering and Management
*version: 20110826
*description: sim300 driver header
*/

/*This driver api is a part of Remote GSM based power management and intrusion detection system. Final year B.tech project by the
*author
*/

/*this header file contains definition of various constants, buffers and constants for the sim300 gsm module. This header can be considered
a basic driver
*module for the sim300 hardware. USART communication and and other interrupt service routines for the message reception from the
sim300 has been incorporated
*this file. the implementer must put the definition of the function processRequest() in the main application code, as per his/her needs.
Whenever sim300
*sends a message, upon reception, the message is entirely stored in a buffer, and processRequest() function is invoked. Apart from this the
implementer must
*keep in his/her mind that, he should call from the main application code the initialisation code for the sim300 initSim300(), as well as
sim300Poll(). This
*driver is largely based on USART_RXC interrupt and TIMER0_OVF interrupt. Global interrupts are automatically enabled in the
initialisation code, so one should
*keep in mind not to disable global interrupts from the application code after initSim300(), or else the driver will not work.*/
```

```
#ifndef _SIM_300_H_
#define _SIM_300_H_

#ifndef _AVR_IO_H_
#include<avr/io.h>
#endif

#ifndef F_CPU
#define F_CPU 12000000l
#endif

#ifndef _UTIL_DELAY_H_
#include<util/delay.h>
#endif

#ifndef _AVR_INTERRUPT_H
#include<avr/interrupt.h>
#endif

#ifndef _STDLIB_H_
#include<stdlib.h>
#endif

#ifndef _AVR_WDT_H
#include<avr/wdt.h>
#endif

#ifndef _STRING_H_
#include<string.h>
#endif

#ifndef DEBUG
#define DEBUG 0
#endif

#if DEBUG
#include "lcd.h"
#endif

#ifndef USE_WATCHDOG
#define USE_WATCHDOG 0          //define this to 1 if watchdog timer is used! set the timer to WDTO_2S in the program
#endif              //also in the program in the infinite loop, call wdt_reset() in short times ~ 250ms

#define SET_BIT(x)        (1<<(x))   //sets bit at position x
#define RESET_BIT(x)    (~SET_BIT(x))        //resets bit at position x

#define TX_ON              UCSRB |=SET_BIT(TXEN)
#define TX_OFF             UCSRB &=RESET_BIT(TXEN)
#define RX_ON              UCSRB |=SET_BIT(RXEN)
#define RX_OFF             UCSRB &=RESET_BIT(RXEN)
#define RX_INTR_DISABLE UCSRB &= RESET_BIT(RXCIE)
#define RX_INTR_ENABLE UCSRB |= SET_BIT(RXCIE)

#define STOP_TIMER0 TCCR0=0x00
#define START_TIMER0 TCCR0=(1<<CS02)|(1<<CS00)             //clock prescaler --> 1024
#define RESET_TIMER0 TCNT0=0x00
#define BUFF_MAX 80
#define TRUE 1
#define FALSE 0

#define STANDARD_MS_DELAY 250
#define STANDARD_US_DELAY 500

char buffer[BUFF_MAX]; //recieve buffer max 128 characters
unsigned char buffLength=0;          //length of message recieved
char incomingMsg=FALSE;            //status flag to show inbound message

#if DEBUG
lcdline buff[2];
lcdInit();
lcdClearScreen(buff);
#endif

//if you don't intend to send sms in any part of the program then assign FALSE value to it
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/*implement this function in the main application as per need. when this function is called expect a string in the buffer[] array*/
extern void processRequest();        //one must reset buffLength back to 0, for proper reception of next messages
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void delayMs(int ms)        //watchdog safe delay routine
{
        ms/=2;
        while(ms--)
        {
                #if USE_WATCHDOG
                        wdt_reset();
                #endif
                _delay_ms(2);
        }
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/*this is the timer initialisaton code*/
void initTimer0()               //ok
{
        STOP_TIMER0;
        RESET_TIMER0;
        TIFR|=1<<TOV0;      //clear interrupt
        TIMSK|=1<<TOIE0; //enable overflow interrupt
        incomingMsg=FALSE;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void timer0Interrupt()        //actual implementaion of the TIMER)_OVF_INTERRUPT service routine
{
        initTimer0();
        buffer[buffLength++]='\0';
        delayMs(STANDARD_MS_DELAY);
        if(!strncmp(buffer,"\n+CMT",5))
                processRequest();   //call process request to take required actions
        delayMs(STANDARD_MS_DELAY);
        buffLength=0;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/*interrupt service routine for timer0 overflow*/
ISR(TIMER0_OVF_vect) //ok
{
        timer0Interrupt();
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/*USART initialisation code. this function accepts baud rate as the only parameter. Recommended to set baud rate as 4800bps @12Mhz
crystal. The baud
*is set using U2X bit*/
void usartInit(unsigned int baud_rate)              //ok
{
        baud_rate=baud_rate/2;
        int ubrr=(int)(F_CPU/(16*baud_rate))-1;
        UCSRA = 0x62;       //set RXC bit, UDRE bit,U2X bit
        UCSRB = 0x98;       //enable receiver,transmitter,interrupt on receive,
        UCSRC = 0x86;       //8 bit data width, no parity
        UCSRC &= 0x7f;
        UBRRH=ubrr>>8 ;
        UBRRL=ubrr;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void rxcInterrupt()             //actual implementation of the USART_RXC_INTERRUPT service routine
{
        if(buffLength>=(BUFF_MAX-2))//boundary checking
                buffLength=0;
        buffer[buffLength++]=UDR;
        if(buffer[buffLength-1]=='\r')      //ignore carriage return responses
                buffLength--;
        RESET_TIMER0;    //reset timer to indicate message reception not complete
        if(incomingMsg==FALSE)        //indicates message transfer initiation
        {
                incomingMsg=TRUE;               //change status of flag variable to indicate incoming message
                START_TIMER0;    //start the timer, timer overflow condition indicates message reception complete
```

```
	}
}
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/*Interrupt service routine for USART_RXC interrupt.*/
ISR(USART_RXC_vect)  //ok
{
	rxcInterrupt();
}
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/*this function writes a byte of data to the usart*/
void usartWrite(char ch)  //ok
{
	loop_until_bit_is_set(UCSRA,UDRE);	//wait for previous transmission request to complete
	UDR=ch; //copy data to Usart Data Register to initiate transfer of data
	buffLength=0;
	return;
}
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/*this function sends a string to the usart for sending*/
void sendResponse(char* msg)	//ok
{
	unsigned char i=0;
	while(msg[i])		//send data untill null character
	{
		usartWrite(msg[i++]);
	}
}
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/*this is the sim300 initialisation code. this must be called from the application program. this initialises the USART @ 4800bps baud and also performs
*a synchronisation with the sim300. finally it disables the echo mode*/
int sim300Init()
{
	unsigned char i;
	int sigStrength=0;
	char sig[3];

#if USE_WATCHDOG
	wdt_disable();
#endif

#if DEBUG
	lcdWriteString("\nsim300Init()");
#endif

	usartInit(4800);	//initialise usart @ 4800 baud
	sei();	//enable global interrupts
	for(i=0;i<64;i++)
	{
		usartWrite('U');	//synchronisation sequence
	}
	usartWrite('\r');
	_delay_ms(500);
	sendResponse("at\r");		//dummy at command
	_delay_ms(STANDARD_MS_DELAY);

#if DEBUG
	buffer[buffLength++]='\0';
	lcdClearScreen(buff);
	lcdWriteString(buff,buffer);
#endif

	sendResponse("ate0\r");	//disable echo from sim300
	_delay_ms(STANDARD_MS_DELAY);

#if DEBUG
	buffer[buffLength++]='\0';
	lcdClearScreen(buff);
	lcdWriteString(buff,buffer);
```

```
#endif

	sendResponse("at+cmgf=1\r");		//enable text mode
	_delay_ms(STANDARD_MS_DELAY);

#if DEBUG
	buffer[buffLength++]='\0';
	lcdClearScreen(buff);
	lcdWriteString(buff,buffer);
#endif

	sendResponse("at+cnmi=1,2,0,0,0\r");	//enable auto recieve sms mode
	_delay_ms(STANDARD_MS_DELAY);

#if DEBUG
	buffer[buffLength++]='\0';
	lcdClearScreen(buff);
	lcdWriteString(buff,buffer);
#endif

	buffLength=0;

#if USE_WATCHDOG
	wdt_enable(WDTO_2S);
#endif
	while(!sigStrength)	//wait for network access and retrieve signal strength
	{
			sendResponse("at+csq\r");
			_delay_ms(250);
			sig[0]=buffer[7];
			sig[1]=buffer[8];
			sig[2]='\0';
			sigStrength=atoi(sig);
			if(!(sigStrength>0 && sigStrength<32))
					sigStrength=0;
			else
					sigStrength+=10;
			sigStrength=sigStrength/10;
			buffLength=0;
	}
	delayMs(500);
	initTimer0();		//initialise timer0, basically enable timer0 overflow interrupt
	return sigStrength;
}
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/*this function takes as input two string parameters, first is the mobile number and second is the message and sends the sms to the given number
*NOTE: mobile number must be of the format "+XXXXXXXXXXXX" not including the qoutes*/
void sendSms(char* mobNo,char* message)  //ok
{
	char end=26;
	char* command="at+cmgs=";
	sendResponse(command);
	sendResponse("\"");
	sendResponse(mobNo);
	sendResponse("\"");
	usartWrite('\r');
	_delay_us(100);
	sendResponse(message);
	usartWrite(end);
	usartWrite('\r');
	delayMs(2000);
}
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/*this function checks the signal strength*/
int checkSignalStrength()
{
	char sig[3];
	int sigStrength;
	sendResponse("at+csq\r");
	sig[0]=buffer[7];
	sig[1]=buffer[8];
	sig[2]='\0';
	sigStrength=atoi(sig);
	if(!(sigStrength>0 && sigStrength<32))
		sigStrength=0;
	else
		sigStrength+=10;
	sigStrength=sigStrength/10;
	return sigStrength;
}
#endif
```

lcd.h

```
/*Author: Supratim Das
*Institute: Calcutta Institute of Engineering and Management
*version: 20110823
*description: 16x2 lcd driver header
*/

/*This driver api is a part of Remote GSM based power management and intrusion detection system. Final year B.tech project by the
*author
*/

/*this is header file contains the lcd API. this header contains the definition of the data and control port to which lcd is connected. This
header should
*only be included after the inclusion of <avr/io.h> and <util/delay.h> in the main program. This header also contains the definition of various
lcd cmds
*and functions necessary for updating the lcd display. Please go through the comment section of each functions for more information about
that function
*/
//check the inclusion of necessary headers for proper operation
#ifndef _LCD_H_
#define _LCD_H_

#ifndef _AVR_IO_H_
#include <avr/io.h>
#endif

#ifndef F_CPU
#define F_CPU 12000000l
#endif

#ifndef _UTIL_DELAY_H_
#include <util/delay.h>
#endif

#ifndef USE_WATCHDOG
#define USE_WATCHDOG 0
#endif

#ifndef _AVR_WDT_H_
#include<avr/wdt.h>
#endif

//port and pin definition of the lcd connections

#define RS	PB0	//register select
#define		RW	PB1	//read/write
#define		EN	PB2	//enable
#define LCD_DATA_PORT		PORTA
#define LCD_DATA_DDR		DDRA
#define		LCD_CONTROL_PORT	PORTB
#define		LCD_CONTROL_DDR		DDRB
#define		LCD_ENABLE	LCD_CONTROL_PORT|=(1<<EN)
#define		LCD_DISABLE	LCD_CONTROL_PORT&=~(1<<EN)

//LCD command codes

#define		CLRSCR	0x01
#define		HOME	0x02
```

```
#define         DEC_CUR             0x04
#define         INC_CUR0x06
#define         SHIFT_DISP_RIGHT            0x05
#define         SHIFT_DISP_LEFT             0x07
#define         DISP_OFF_CUR_OFF            0x08
#define         DISP_OFF_CUR_ON                         0x0a
#define         DISP_ON_CUR_OFF                         0x0c
#define         DISP_ON_CUR_BLINK           0x0e
#define         SHIFT_CUR_LEFT              0x10
#define         SHIFT_CUR_RIGHT                         0x14
#define         SHIFT_ENT_DISP_LEFT         0x18
#define         SHIFT_ENT_DISP_RIGHT        0x1c
#define         MOV_CUR_1_LINE_BEG          0x80
#define         MOV_CUR_2_LINE_BEG          0xc0
#define         LINEX2    0x38


//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/*this function takes as input the specific lcd command and executes them. for the various lcd commands please refer to the LCD command
code section at
*the beginning of this document
*/
void lcdCmd(char cmd)
{
        LCD_CONTROL_PORT&=~(1<<RS);     //select command register
        LCD_DATA_PORT=cmd;        //output command code on lcd port
        LCD_ENABLE;        //enable lcd
        _delay_us(80);        //delay to latch inputs
        LCD_DISABLE;       //disable lcd
}
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/*this function takes as input a character data and writes the character to the lcd display
*/
void lcdData(char data)
{
        LCD_CONTROL_PORT|=(1<<RS);        //select data register
        LCD_DATA_PORT=data;
        LCD_ENABLE;
        _delay_us(80);        //delay to latch inputs
        LCD_DISABLE;
}
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/*this function contains the lcd initialisation code and must be called before any operation can be performed on the lcd.
*/
void lcdInit()     //lcd initialisation
{
#if USE_WATCHDOG
        wdt_reset();
        wdt_disable();
#endif
        LCD_DATA_DDR=0xff;          //initialise lcd output port
        LCD_DATA_PORT=0x00;
        LCD_CONTROL_DDR=0x0f;  //initialise lcd control port
        LCD_CONTROL_PORT=0x00;
        LCD_CONTROL_PORT&=~(1<<RW);
        _delay_ms(100);
        lcdCmd(LINEX2); // 8 data lines
        lcdCmd(INC_CUR); // cursor setting
        lcdCmd(DISP_ON_CUR_BLINK); // display ON
        lcdCmd(CLRSCR); // clear LCD memory
        _delay_ms(500);
        lcdData(' ');            //write a blank
        lcdCmd(MOV_CUR_1_LINE_BEG);       //move back to first line first character
        _delay_ms(1000);
#if USE_WATCHDOG
        wdt_enable(WDTO_2S);
#endif
}
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
/*this is a datastructure that can be associated with one line of a 16x2 lcd. In the  main program this structure should be used as a buffer for the lcd
*and should be declared as lcdline buff[2]={0};
*/
struct line
{
	char ch[16];		//buffer for the 16 character space for one line of a 16x2 lcd
	unsigned char i;	//stores the offset of the line that is written
};

typedef struct line lcdline;
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/*this is a higher level lcd function that writes into the lcd which is very similar to lcd_data() but additionally takes care of wrapping up the display
*as well as scrolling the diisplay at display boundaries
*/
void lcdWriteChar(lcdline* display,char ch)
{
	unsigned char line=0,i;
	if(display[0].i>=16)	//check if line 1 is full
	{
		line=2;	//if full set line 2 as working line
	}
	else
	{
		line=1;	//else remain in line 1
	}
	if(ch=='\r')
	{
		//display[line-1].i=0;
		return;
	}
	if(ch=='\n')
	{
		while(display[line-1].i<16)
		{
			display[line-1].ch[display[line-1].i]=' ';
			display[line-1].i++;
		}
		return;
	}
	switch(line)
	{
		case 1:	//if the current working line is 1
			if(display[0].i==0)	//check if first character of the line
			{
				lcdCmd(MOV_CUR_1_LINE_BEG);	//move cursor to the beginning of second line
			}
			display[0].ch[display[0].i]=ch;	//assing character data to appropriate location in the buffer
			lcdData(display[0].ch[display[0].i]);	//display the data
			display[0].i++;	//update the line offset
			break;
		case 2:	//if the current working line is 2
			if(display[1].i==0)	//check if first character of the line
			{
				lcdCmd(MOV_CUR_2_LINE_BEG);	//move cursor to the beginning of second line
			}
			if(display[1].i>=16)	//if line 2 is full
			{
				lcdCmd(MOV_CUR_1_LINE_BEG);	//move cursor to the beginning of line 2
				lcdCmd(DISP_OFF_CUR_OFF);
				for(i=0;i<16;i++)	//copy line 2 buffer to line 1 buffer and scroll line 2 to line 1
				{
					display[0].ch[i]=display[1].ch[i];
					lcdData(display[0].ch[i]);
				}
				lcdCmd(MOV_CUR_2_LINE_BEG);	//the following lines of code clears the line 2
				for(i=0;i<16;i++)
				{
					lcdData(' ');
				}
				display[1].i=0;
				lcdCmd(DISP_ON_CUR_BLINK);
```

```
                        lcdCmd(MOV_CUR_2_LINE_BEG);
                    }
                    display[1].ch[display[1].i]=ch;
                    lcdData(display[1].ch[display[1].i]);
                    display[1].i++;
                    break;
    }
}
///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/*this function writes a string to the lcd. Its important to consider the lcd character space. To big a string will be scrolled, so the programmer should
only use this for short strings to display
*/
void lcdWriteString(lcdline* display,char *string)
{
    unsigned char len=0;
    while(string[len]!='\0')
    {
            lcdWriteChar(display,string[len]);
            len++;
    }
}
///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/*lcd clear screen*/
void lcdClearScreen(lcdline buff[])
{
    lcdCmd(CLRSCR);
    buff[0].i=0;
    buff[1].i=0;
#if USE_WATCHDOG
    wdt_reset();
#endif
    _delay_ms(100);
#if USE_WATCHDOG
    wdt_reset();
#endif

}
///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#endif
```

usb_command_codes.h

```
#ifndef _USB_INTERFACE_COMMAND_H_
#define _USB_INTERFACE_COMMAND_H_
/*this is the command code the device is supposed to respond over the USB interface
*NOTE: THIS FILE IS TO BE INCLUDED IN BOTH DEVICE AND HOST SIDE
*C CODE*/

#define GET_MOB_NO   0
#define SET_MOB_NO   1
#define GET_DEV_ID          2
#define SET_DEV_ID          3
#define SYSTEM_CHECK        4
#define GET_LOG             5
#define CLEAR_LOG           6
#define GET_DEV_STATUS 7
#define SET_DEV_STATUS 8
#endif
```

usbconfig.h

```
/* Name: usbconfig.h
 * Project: V-USB, virtual USB port for Atmel's(r) AVR(r) microcontrollers
 * Author: Christian Starkjohann
 * Creation Date: 2005-04-01
 * Tabsize: 4
 * Copyright: (c) 2005 by OBJECTIVE DEVELOPMENT Software GmbH
 * License: GNU GPL v2 (see License.txt), GNU GPL v3 or proprietary (CommercialLicense.txt)
 * This Revision: $Id: usbconfig-prototype.h 785 2010-05-30 17:57:07Z cs $
 */

#ifndef __usbconfig_h_included__
#define __usbconfig_h_included__

/*
General Description:
This file is an example configuration (with inline documentation) for the USB
driver. It configures V-USB for USB D+ connected to Port D bit 2 (which is
also hardware interrupt 0 on many devices) and USB D- to Port D bit 4. You may
wire the lines to any other port, as long as D+ is also wired to INT0 (or any
other hardware interrupt, as long as it is the highest level interrupt, see
section at the end of this file).
+ To create your own usbconfig.h file, copy this file to your project's
+ firmware source directory) and rename it to "usbconfig.h".
+ Then edit it accordingly.
*/

/* ---------------------------- Hardware Config ---------------------------- */

#define USB_CFG_IOPORTNAME      B
/* This is the port where the USB bus is connected. When you configure it to
 * "B", the registers PORTB, PINB and DDRB will be used.
 */
#define USB_CFG_DMINUS_BIT      3
/* This is the bit number in USB_CFG_IOPORT where the USB D- line is connected.
 * This may be any bit in the port.
 */
#define USB_CFG_DPLUS_BIT       4
/* This is the bit number in USB_CFG_IOPORT where the USB D+ line is connected.
 * This may be any bit in the port. Please note that D+ must also be connected
 * to interrupt pin INT0! [You can also use other interrupts, see section
 * "Optional MCU Description" below, or you can connect D- to the interrupt, as
 * it is required if you use the USB_COUNT_SOF feature. If you use D- for the
 * interrupt, the USB interrupt will also be triggered at Start-Of-Frame
 * markers every millisecond.]
 */
#define USB_CFG_CLOCK_KHZ       (F_CPU/1000)
/* Clock rate of the AVR in kHz. Legal values are 12000, 12800, 15000, 16000,
 * 16500, 18000 and 20000. The 12.8 MHz and 16.5 MHz versions of the code
 * require no crystal, they tolerate +/- 1% deviation from the nominal
 * frequency. All other rates require a precision of 2000 ppm and thus a
 * crystal!
 * Since F_CPU should be defined to your actual clock rate anyway, you should
 * not need to modify this setting.
 */
#define USB_CFG_CHECK_CRC       0
/* Define this to 1 if you want that the driver checks integrity of incoming
 * data packets (CRC checks). CRC checks cost quite a bit of code size and are
 * currently only available for 18 MHz crystal clock. You must choose
 * USB_CFG_CLOCK_KHZ = 18000 if you enable this option.
 */

/* ----------------------- Optional Hardware Config ------------------------ */

/* #define USB_CFG_PULLUP_IOPORTNAME   D */
/* If you connect the 1.5k pullup resistor from D- to a port pin instead of
 * V+, you can connect and disconnect the device from firmware by calling
 * the macros usbDeviceConnect() and usbDeviceDisconnect() (see usbdrv.h).
 * This constant defines the port on which the pullup resistor is connected.
 */
/* #define USB_CFG_PULLUP_BIT          4 */
/* This constant defines the bit number in USB_CFG_PULLUP_IOPORT (defined
 * above) where the 1.5k pullup resistor is connected. See description
 * above for details.
 */

/* --------------------------- Functional Range ---------------------------- */
```

```
#define USB_CFG_HAVE_INTRIN_ENDPOINT    0
/* Define this to 1 if you want to compile a version with two endpoints: The
 * default control endpoint 0 and an interrupt-in endpoint (any other endpoint
 * number).
 */
#define USB_CFG_HAVE_INTRIN_ENDPOINT3   0
/* Define this to 1 if you want to compile a version with three endpoints: The
 * default control endpoint 0, an interrupt-in endpoint 3 (or the number
 * configured below) and a catch-all default interrupt-in endpoint as above.
 * You must also define USB_CFG_HAVE_INTRIN_ENDPOINT to 1 for this feature.
 */
#define USB_CFG_EP3_NUMBER              3
/* If the so-called endpoint 3 is used, it can now be configured to any other
 * endpoint number (except 0) with this macro. Default if undefined is 3.
 */
/* #define USB_INITIAL_DATATOKEN           USBPID_DATA1 */
/* The above macro defines the startup condition for data toggling on the
 * interrupt/bulk endpoints 1 and 3. Defaults to USBPID_DATA1.
 * Since the token is toggled BEFORE sending any data, the first packet is
 * sent with the oposite value of this configuration!
 */
#define USB_CFG_IMPLEMENT_HALT          0
/* Define this to 1 if you also want to implement the ENDPOINT_HALT feature
 * for endpoint 1 (interrupt endpoint). Although you may not need this feature,
 * it is required by the standard. We have made it a config option because it
 * bloats the code considerably.
 */
#define USB_CFG_SUPPRESS_INTR_CODE      0
/* Define this to 1 if you want to declare interrupt-in endpoints, but don't
 * want to send any data over them. If this macro is defined to 1, functions
 * usbSetInterrupt() and usbSetInterrupt3() are omitted. This is useful if
 * you need the interrupt-in endpoints in order to comply to an interface
 * (e.g. HID), but never want to send any data. This option saves a couple
 * of bytes in flash memory and the transmit buffers in RAM.
 */
#define USB_CFG_INTR_POLL_INTERVAL      10
/* If you compile a version with endpoint 1 (interrupt-in), this is the poll
 * interval. The value is in milliseconds and must not be less than 10 ms for
 * low speed devices.
 */
#define USB_CFG_IS_SELF_POWERED         0
/* Define this to 1 if the device has its own power supply. Set it to 0 if the
 * device is powered from the USB bus.
 */
#define USB_CFG_MAX_BUS_POWER           200
/* Set this variable to the maximum USB bus power consumption of your device.
 * The value is in milliamperes. [It will be divided by two since USB
 * communicates power requirements in units of 2 mA.]
 */
#define USB_CFG_IMPLEMENT_FN_WRITE      1
/* Set this to 1 if you want usbFunctionWrite() to be called for control-out
 * transfers. Set it to 0 if you don't need it and want to save a couple of
 * bytes.
 */
#define USB_CFG_IMPLEMENT_FN_READ       1
/* Set this to 1 if you need to send control replies which are generated
 * "on the fly" when usbFunctionRead() is called. If you only want to send
 * data from a static buffer, set it to 0 and return the data from
 * usbFunctionSetup(). This saves a couple of bytes.
 */
#define USB_CFG_IMPLEMENT_FN_WRITEOUT   0
/* Define this to 1 if you want to use interrupt-out (or bulk out) endpoints.
 * You must implement the function usbFunctionWriteOut() which receives all
 * interrupt/bulk data sent to any endpoint other than 0. The endpoint number
 * can be found in 'usbRxToken'.
 */
#define USB_CFG_HAVE_FLOWCONTROL        0
/* Define this to 1 if you want flowcontrol over USB data. See the definition
 * of the macros usbDisableAllRequests() and usbEnableAllRequests() in
 * usbdrv.h.
 */
#define USB_CFG_DRIVER_FLASH_PAGE       0
/* If the device has more than 64 kBytes of flash, define this to the 64 k page
 * where the driver's constants (descriptors) are located. Or in other words:
 * Define this to 1 for boot loaders on the ATMega128.
 */
#define USB_CFG_LONG_TRANSFERS          0
/* Define this to 1 if you want to send/receive blocks of more than 254 bytes
 * in a single control-in or control-out transfer. Note that the capability
 * for long transfers increases the driver size.
 */
/* #define USB_RX_USER_HOOK(data, len)     if(usbRxToken == (uchar)USBPID_SETUP) blinkLED(); */
/* This macro is a hook if you want to do unconventional things. If it is
 * defined, it's inserted at the beginning of received message processing.
 * If you eat the received message and don't want default processing to
 * proceed, do a return after doing your things. One possible application
 * (besides debugging) is to flash a status LED on each packet.
 */
/* #define USB_RESET_HOOK(resetStarts)     if(!resetStarts){hadUsbReset();} */
/* This macro is a hook if you need to know when an USB RESET occurs. It has
 * one parameter which distinguishes between the start of RESET state and its
 * end.
 */
/* #define USB_SET_ADDRESS_HOOK()              hadAddressAssigned(); */
/* This macro (if defined) is executed when a USB SET_ADDRESS request was
 * received.
 */
#define USB_COUNT_SOF                   0
/* define this macro to 1 if you need the global variable "usbSofCount" which
 * counts SOF packets. This feature requires that the hardware interrupt is
 * connected to D- instead of D+.
 */
/* #ifdef __ASSEMBLER__
 * macro myAssemblerMacro
 *     in      YL, TCNT0
 *     sts     timer0Snapshot, YL
 *     endm
 * #endif
 * #define USB_SOF_HOOK                    myAssemblerMacro
 * This macro (if defined) is executed in the assembler module when a
 * Start Of Frame condition is detected. It is recommended to define it to
 * the name of an assembler macro which is defined here as well so that more
 * than one assembler instruction can be used. The macro may use the register
 * YL and modify SREG. If it lasts longer than a couple of cycles, USB messages
 * immediately after an SOF pulse may be lost and must be retried by the host.
 * What can you do with this hook? Since the SOF signal occurs exactly every
 * 1 ms (unless the host is in sleep mode), you can use it to tune OSCCAL in
 * designs running on the internal RC oscillator.
 * Please note that Start Of Frame detection works only if D- is wired to the
 * interrupt, not D+. THIS IS DIFFERENT THAN MOST EXAMPLES!
 */
#define USB_CFG_CHECK_DATA_TOGGLING     0
/* define this macro to 1 if you want to filter out duplicate data packets
 * sent by the host. Duplicates occur only as a consequence of communication
 * errors, when the host does not receive an ACK. Please note that you need to
 * implement the filtering yourself in usbFunctionWriteOut() and
 * usbFunctionWrite(). Use the global usbCurrentDataToken and a static variable
 * for each control- and out-endpoint to check for duplicate packets.
 */
#define USB_CFG_HAVE_MEASURE_FRAME_LENGTH   0
/* define this macro to 1 if you want the function usbMeasureFrameLength()
 * compiled in. This function can be used to calibrate the AVR's RC oscillator.
 */
#define USB_USE_FAST_CRC                0
/* The assembler module has two implementations for the CRC algorithm. One is
 * faster, the other is smaller. This CRC routine is only used for transmitted
 * messages where timing is not critical. The faster routine needs 31 cycles
 * per byte while the smaller one needs 61 to 69 cycles. The faster routine
 * may be worth the 32 bytes bigger code size if you transmit lots of data and
 * run the AVR close to its limit.
 */

/* -------------------------- Device Description --------------------------- */

#define  USB_CFG_VENDOR_ID       0xc0, 0x16 /* = 0x16c0 = 5824 = voti.nl */
/* USB vendor ID for the device, low byte first. If you have registered your
 * own Vendor ID, define it here. Otherwise you may use one of obdev's free
 * shared VID/PID pairs. Be sure to read USB-IDs-for-free.txt for rules!
 * *** IMPORTANT NOTE ***
 * This template uses obdev's shared VID/PID pair for Vendor Class devices
 * with libusb: 0x16c0/0x5dc.  Use this VID/PID pair ONLY if you understand
 * the implications!
 */
#define  USB_CFG_DEVICE_ID       0xdc, 0x05 /* = 0x05dc = 1500 */
/* This is the ID of the product, low byte first. It is interpreted in the
 * scope of the vendor ID. If you have registered your own VID with usb.org
 * or if you have licensed a PID from somebody else, define it here. Otherwise
 * you may use one of obdev's free shared VID/PID pairs. See the file
 * USB-IDs-for-free.txt for details!
 * *** IMPORTANT NOTE ***
 * This template uses obdev's shared VID/PID pair for Vendor Class devices
 * with libusb: 0x16c0/0x5dc.  Use this VID/PID pair ONLY if you understand
 * the implications!
 */
#define USB_CFG_DEVICE_VERSION  0x00, 0x01
/* Version number of the device: Minor number first, then major number.
 */
#define USB_CFG_VENDOR_NAME     's', 'u', 'p', 'r', 'a', 't', 'i', 'm','o','f','f','i','c','i','o','@','g','m','a','i','l','.','c','o','m'
#define USB_CFG_VENDOR_NAME_LEN 25
/* These two values define the vendor name returned by the USB device. The name
 * must be given as a list of characters under single quotes. The characters
 * are interpreted as Unicode (UTF-16) entities.
 * If you don't want a vendor name string, undefine these macros.
 * ALWAYS define a vendor name containing your Internet domain name if you use
 * obdev's free shared VID/PID pair. See the file USB-IDs-for-free.txt for
 * details.
 */
#define USB_CFG_DEVICE_NAME     'G', 'S', 'M', ' ', 'C', 'o', 'n', 't','r','o','l'
#define USB_CFG_DEVICE_NAME_LEN 11
/* Same as above for the device name. If you don't want a device name, undefine
 * the macros. See the file USB-IDs-for-free.txt before you assign a name if
 * you use a shared VID/PID.
 */
/*#define USB_CFG_SERIAL_NUMBER   'N', 'o', 'n', 'e' */
/*#define USB_CFG_SERIAL_NUMBER_LEN   0 */
/* Same as above for the serial number. If you don't want a serial number,
 * undefine the macros.
 * It may be useful to provide the serial number through other means than at
 * compile time. See the section about descriptor properties below for how
 * to fine tune control over USB descriptors such as the string descriptor
 * for the serial number.
 */
#define USB_CFG_DEVICE_CLASS        0xff    /* set to 0 if deferred to interface */
#define USB_CFG_DEVICE_SUBCLASS     0
/* See USB specification if you want to conform to an existing device class.
 * Class 0xff is "vendor specific".
 */
#define USB_CFG_INTERFACE_CLASS     0   /* define class here if not at device level */
#define USB_CFG_INTERFACE_SUBCLASS  0
#define USB_CFG_INTERFACE_PROTOCOL  0
/* See USB specification if you want to conform to an existing device class or
 * protocol. The following classes must be set at interface level:
 * HID class is 3, no subclass and protocol required (but may be useful!)
 * CDC class is 2, use subclass 2 and protocol 1 for ACM
 */
/* #define USB_CFG_HID_REPORT_DESCRIPTOR_LENGTH    42 */
/* Define this to the length of the HID report descriptor, if you implement
 * an HID device. Otherwise don't define it or define it to 0.
 * If you use this define, you must add a PROGMEM character array named
 * "usbHidReportDescriptor" to your code which contains the report descriptor.
 * Don't forget to keep the array and this define in sync!
 */

/* #define USB_PUBLIC static */
/* Use the define above if you #include usbdrv.c instead of linking against it.
 * This technique saves a couple of bytes in flash memory.
 */

/* ------------------- Fine Control over USB Descriptors ------------------- */
/* If you don't want to use the driver's default USB descriptors, you can
 * provide our own. These can be provided as (1) fixed length static data in
 * flash memory, (2) fixed length static data in RAM or (3) dynamically at
 * runtime in the function usbFunctionDescriptor(). See usbdrv.h for more
 * information about this function.
 * Descriptor handling is configured through the descriptor's properties. If
 * no properties are defined or if they are 0, the default descriptor is used.
 * Possible properties are:
 *   + USB_PROP_IS_DYNAMIC: The data for the descriptor should be fetched
 *     at runtime via usbFunctionDescriptor(). If the usbMsgPtr mechanism is
 *     used, the data is in FLASH by default. Add property USB_PROP_IS_RAM if
 *     you want RAM pointers.
 *   + USB_PROP_IS_RAM: The data returned by usbFunctionDescriptor() or found
 *     in static memory is in RAM, not in flash memory.
 *   + USB_PROP_LENGTH(len): If the data is in static memory (RAM or flash),
 *     the driver must know the descriptor's length. The descriptor itself is
 *     found at the address of a well known identifier (see below).
 * List of static descriptor names (must be declared PROGMEM if in flash):
 *   char usbDescriptorDevice[];
 *   char usbDescriptorConfiguration[];
 *   char usbDescriptorHidReport[];
 *   char usbDescriptorString0[];
 *   int usbDescriptorStringVendor[];
 *   int usbDescriptorStringDevice[];
 *   int usbDescriptorStringSerialNumber[];
 * Other descriptors can't be provided statically, they must be provided
 * dynamically at runtime.
 *
 * Descriptor properties are or-ed or added together, e.g.:
 * #define USB_CFG_DESCR_PROPS_DEVICE   (USB_PROP_IS_RAM | USB_PROP_LENGTH(18))
 *
 * The following descriptors are defined:
 *   USB_CFG_DESCR_PROPS_DEVICE
 *   USB_CFG_DESCR_PROPS_CONFIGURATION
 *   USB_CFG_DESCR_PROPS_STRINGS
 *   USB_CFG_DESCR_PROPS_STRING_0
 *   USB_CFG_DESCR_PROPS_STRING_VENDOR
 *   USB_CFG_DESCR_PROPS_STRING_PRODUCT
 *   USB_CFG_DESCR_PROPS_STRING_SERIAL_NUMBER
 *   USB_CFG_DESCR_PROPS_HID
 *   USB_CFG_DESCR_PROPS_HID_REPORT
 *   USB_CFG_DESCR_PROPS_UNKNOWN (for all descriptors not handled by the driver)
 *
 * Note about string descriptors: String descriptors are not just strings, they
 * are Unicode strings prefixed with a 2 byte header. Example:
 * int  serialNumberDescriptor[] = {
 *     USB_STRING_DESCRIPTOR_HEADER(6),
 *     'S', 'e', 'r', 'i', 'a', 'l'
 * };
 */

#define USB_CFG_DESCR_PROPS_DEVICE                  0
#define USB_CFG_DESCR_PROPS_CONFIGURATION           0
#define USB_CFG_DESCR_PROPS_STRINGS                 0
#define USB_CFG_DESCR_PROPS_STRING_0                0
#define USB_CFG_DESCR_PROPS_STRING_VENDOR           0
#define USB_CFG_DESCR_PROPS_STRING_PRODUCT          0
#define USB_CFG_DESCR_PROPS_STRING_SERIAL_NUMBER    0
#define USB_CFG_DESCR_PROPS_HID                     0
#define USB_CFG_DESCR_PROPS_HID_REPORT              0
#define USB_CFG_DESCR_PROPS_UNKNOWN                 0

/* ----------------------- Optional MCU Description ------------------------ */

/* The following configurations have working defaults in usbdrv.h. You
 * usually don't need to set them explicitly. Only if you want to run
 * the driver on a device which is not yet supported or with a compiler
 * which is not fully supported (such as IAR C) or if you use a differnt
 * interrupt than INT0, you may have to define some of these.
 */
/* #define USB_INTR_CFG            MCUCR */
/* #define USB_INTR_CFG_SET        ((1 << ISC00) | (1 << ISC01)) */
/* #define USB_INTR_CFG_CLR        0 */
/* #define USB_INTR_ENABLE         GIMSK */
/* #define USB_INTR_ENABLE_BIT     INT0 */
/* #define USB_INTR_PENDING        GIFR */
/* #define USB_INTR_PENDING_BIT    INTF0 */
/* #define USB_INTR_VECTOR         INT0_vect */

#endif /* __usbconfig_h_included__ */
```

gsm_control_with_usb.c

```
#include <avr/io.h>
#include <avr/interrupt.h>  /* for sei() */
#include <avr/pgmspace.h>   /* required by usbdrv.h */
#include <util/delay.h>     /* for _delay_ms() */
#include <avr/eeprom.h>
#define USE_WATCHDOG 1
#include "lcd.h"
#include "sim300.h"
#include <string.h>
#include "usb_command_codes.h"
#include "usbdrv/usbdrv.h"

#define MAX_DEVICE 4
#define EEPROM_DEVICE_START_ADDR   0x000010
#define EEPROM_DEVICE_OFFSET 0x000008
#define EEPROM_MOB_ADDR      0x000000
#define EEPROM_MOB_LEN 14
#define EEPROM_DEVICE_STATUS_ADDR (EEPROM_DEVICE_START_ADDR + (EEPROM_DEVICE_OFFSET*MAX_DEVICE))

lcdline lcdBuff[2];             //lcd output buffer

#define STOP_TIMER1 TCCR1B=0x00
#define START_TIMER1 TCCR1B=0x05
#define RESET_TIMER1 TCNT1=0x0000
#define OFF_TIMER 2
#define ON_TIMER 1
#define NO_TIMER 0
#define ON 1
#define OFF 0


////////////////////////////////////////////////////////////////////////////////////
////////////////////////CONTROLLER INTERFACE/////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////

char device[MAX_DEVICE][EEPROM_DEVICE_OFFSET];
char mobNo[14]="";
unsigned int timerStatus[MAX_DEVICE];
unsigned char timerMask[MAX_DEVICE];
unsigned char outputStatus;
long int pollTime=20000;
unsigned int securityTimer;


void switchPortPin(unsigned char pin,unsigned char action)
{
	if(action) //on
	{
			PORTC|=1<<pin;
	}
	else	//off
	{
			PORTC&=~(1<<pin);
	}
	outputStatus=PORTC;
	eeprom_write_byte((uint8_t*)EEPROM_DEVICE_STATUS_ADDR,0x0f & outputStatus);
}

void initTimer1()		//initialisation of 16bit timer/counter
{
	TIMSK|=1<<OCIE1A;		//enable compare match A interrupt
	TIFR|=1<<OCF1A;  //clear interrupt flag
	OCR1A=0xe4e2;     //compare match every 5 secs --> experimentally determined for accuracy
	TCCR1A=0x00;
	STOP_TIMER1;
}

void timer1Interrupt()
{
	unsigned char i,required=0;
	RESET_TIMER1;
	//lcdWriteString(lcdBuff,"t1 isr-->");
	for(i=0;i<MAX_DEVICE;i++)	//scan through all the timer variables of devices attached
	{
		if(timerStatus[i])	//if variable is non zero
		{
			timerStatus[i]--;
			required++;		//this variable is used to decide whether to keep the timer running
		}
		else	//timer timeout
		{
			switch(timerMask[i]) //check the timer mask of the device in question
			{
				case ON_TIMER:
					switchPortPin(i,ON);
					break;
				case OFF_TIMER:
					switchPortPin(i,OFF);
					break;
				default:
					break;
			}
		}
	}
	if(securityTimer)
	{
		required++;
		securityTimer--;
	}
	else
	{
		GICR|=1<<INT1;	//re-enable int1 ext interrupt!
	}
	if(!required)		//if no device is having a pending timer
		STOP_TIMER1;	//shut down timer
}

ISR(TIMER1_COMPA_vect)		//called every 5 secs
{
	timer1Interrupt();
}

ISR(INT1_vect)
{
	GICR&=~(1<<INT1);		//disable int1 ext interrupt
	securityTimer=12*2;//setting for re-enabling security timer after 2 mins
	lcdClearScreen(lcdBuff);
	lcdWriteString(lcdBuff,"ALERT! INTRUSION");
	sendSms(mobNo,"ALERT! INTRUSION DETECTED!");
	lcdWriteString(lcdBuff,"ALERT SMS SENT");
	START_TIMER1;
}

int controller()
{
	unsigned short int i,sigStrength;
	char sigVal[16];
	lcdWriteString(lcdBuff,"GSM Control 1.0\ninitialising...");
	_delay_ms(2000);
	initTimer1();
	sigStrength=sim300Init();
	/*read data from non volatile eeprom*/
	eeprom_read_block((void*)mobNo,(const void*)EEPROM_MOB_ADDR,EEPROM_MOB_LEN);
	mobNo[EEPROM_MOB_LEN-1]='\0';
	for(i=0;i<MAX_DEVICE;i++)
	{
		eeprom_read_block((void*)device[i],(const
void*)(EEPROM_DEVICE_START_ADDR+(i*EEPROM_DEVICE_OFFSET)),EEPROM_DEVICE_OFFSET);
	}
	outputStatus=eeprom_read_byte((uint8_t*)EEPROM_DEVICE_STATUS_ADDR);

	DDRC=0xff;		//configure output ports
	PORTC=0x0f & outputStatus;

	MCUCR&=~((1<<ISC11)|(1<<ISC10));
```

```
        GICR|=1<<INT1;      //enable int1 external interrupt; level trigerred

        delayMs(2000);
        //sendSms(mobNo,"GSM Control online!");
        for(;;)
        {
        CHECK_AGAIN:
                sigStrength=checkSignalStrength();
                lcdClearScreen(lcdBuff);
                lcdWriteString(lcdBuff," System  Online");
                lcdWriteString(lcdBuff,"\nsignal: ");
                switch(sigStrength)
                {
                        case 1:
                                lcdWriteString(lcdBuff,"WEAK");
                                break;
                        case 2:
                                lcdWriteString(lcdBuff,"MEDIUM");
                                break;
                        case 3:
                                lcdWriteString(lcdBuff,"STRONG");
                                break;
                        default:
                                lcdWriteString(lcdBuff,"UNKNOWN");
                                lcdWriteString(lcdBuff,buffer);
                                delayMs(1000);
                                goto CHECK_AGAIN;
                }
                delayMs(pollTime);
        }
}

int countTokens(char* str)              //this function counts the number of tokens in the command string
{
        unsigned short int i=0;
        unsigned int short count=0;
        unsigned char space=FALSE;
        while(str[i])
        {
                if(!space && str[i]==' ')
                {
                        count++;
                        space=TRUE;
                }
                else if(str[i]!=' ')
                {
                        space=FALSE;
                }
                i++;
        }
        return count+1;
}

void processRequest()     //application specific logic is implemented here, rest is handled by the embedded OS
{
        char *tmp;
        char *token[4];
        unsigned short int count;
        unsigned short int i;
        pollTime+=10000;
        tmp=strtok(buffer,"\"");
        tmp=strtok(NULL,"\"");
        if(!strncmp(mobNo,tmp,13))     //valid number
        {
                tmp=strtok(NULL,"\n");
                tmp=strtok(NULL,"\n");          //now tmp contains the command
                tmp=strupr(tmp);
                count=countTokens(tmp);
                for(i=0;i<count;)       //extract tokens
                {
                        if(i==0)
                                token[i]=strtok(tmp," ");
                        else
                                token[i]=strtok(NULL," ");
                        if(token[i][0]==' ')
```

```
				continue;
			i++;
		}
		buffer[0]='\0';
		switch(count)
		{
			case 1:
				if(!strcmp(token[0],"RESET") || !strcmp(tmp,"RESET"))
				{
					lcdWriteString(lcdBuff,"\nDEVICE RESET");
					_delay_ms(3000);	//reset using watchdog!
				}
				break;
			case 2:
				if(!(strcmp(token[0],"SHOW") || strcmp(token[1],"STATUS")))
				{
					lcdWriteString(lcdBuff,"\nshow status cmd");
					for(i=0;i<MAX_DEVICE;i++)
					{
						strcat(buffer,device[i]);
						if(outputStatus & (1<<i))
						{
							strcat(buffer,":ON\r");
						}
						else
						{
							strcat(buffer,":OFF\r");
						}
					}
					count=0;
				}
				break;
			case 3:
				if(!(strcmp(token[0],"SET") || strcmp(token[2],"ON")))
				{
					lcdWriteString(lcdBuff,"\ndevice on cmd");
					for(i=0;i<MAX_DEVICE;i++)
					{
						if(!(strcmp(token[1],device[i])))
						{
							switchPortPin(i,ON);
							strcat(buffer,device[i]);
							strcat(buffer," TURNED ON");
							count=0;
							break;
						}
					}
				}
				else if(!(strcmp(token[0],"SET") || strcmp(token[2],"OFF")))
				{
					lcdWriteString(lcdBuff,"\ndevice off cmd");
					for(i=0;i<MAX_DEVICE;i++)
					{
						if(!(strcmp(token[1],device[i])))
						{
							switchPortPin(i,OFF);
							strcat(buffer,device[i]);
							strcat(buffer," TURNED OFF");
							count=0;
							break;
						}
					}
				}
				break;
			case 4:
				if(!(strcmp(token[0],"SET") || strcmp(token[2],"ON")))
				{
					lcdWriteString(lcdBuff,"\ndevice on timer cmd");
					for(i=0;i<MAX_DEVICE;i++)
					{
						if(!(strcmp(token[1],device[i])))
						{
							switchPortPin(i,OFF);
							timerMask[i]=ON_TIMER;
							timerStatus[i]=atoi(token[3])*12;
```

```
                            strcat(buffer,device[i]);
                            strcat(buffer," ON TIMER SET FOR ");
                            strcat(buffer,token[3]);
                            strcat(buffer," mins");
                            count=0;
                            break;
                        }
                    }
                }
                else if(!(strcmp(token[0],"SET") || strcmp(token[2],"OFF")))
                {
                    lcdWriteString(lcdBuff,"\ndevice off timer cmd");
                    for(i=0;i<MAX_DEVICE;i++)
                    {
                        if(!(strcmp(token[1],device[i])))
                        {
                            switchPortPin(i,ON);
                            timerMask[i]=OFF_TIMER;
                            timerStatus[i]=atoi(token[3])*12;
                            strcat(buffer,device[i]);
                            strcat(buffer," OFF TIMER SET FOR ");
                            strcat(buffer,token[3]);
                            strcat(buffer," mins");
                            count=0;
                            break;
                        }
                    }
                }
                START_TIMER1;
                break;
            default:
                break;
        }
        if(!count)
        {
            delayMs(250);
            lcdClearScreen(lcdBuff);
            lcdWriteString(lcdBuff,buffer);
            sendSms(mobNo,buffer);
        }
        else
        {
            delayMs(250);
            lcdClearScreen(lcdBuff);
            lcdWriteString(lcdBuff,"error parsing \ncommand");
            sendSms(mobNo,"error parsing command");
        }
    }
}

////////////////////////////////////////////////////////////////////////////////

/* ------------------------------------------------------------------------- */
/* ----------------------------- USB interface ----------------------------- */
/* ------------------------------------------------------------------------- */

unsigned char dataLength,requestType;
void *eepromAddress;
uchar usbFunctionWrite(uchar *data, uchar len)              //this function is called in chunks of 8 bytes to transmit data from host---
>device        (device write function)
{
    lcdWriteString(lcdBuff,"\nusb write...");
    switch(requestType)
    {
        case SET_MOB_NO:
        case SET_DEV_ID:
        case SET_DEV_STATUS:
            eeprom_write_block(data,eepromAddress,len);
            break;
        default:
            return -1;
            break;
    }
    dataLength-=len;
    eepromAddress+=len;
```

```
	if(dataLength)
		return 0;
	else
		return 1;

}


uchar usbFunctionRead(uchar *data, uchar len)			//this function is called in chunks of 8 bytes to transmit data from device--
--->host (device read function)
{
	lcdWriteString(lcdBuff,"\nusb read...");
	switch(requestType)
	{
		case GET_MOB_NO:
		case GET_DEV_ID:
		case GET_DEV_STATUS:
			eeprom_read_block(data,eepromAddress,len);
			break;
		default:
			return -1;
			break;
	}
	dataLength-=len;
	eepromAddress+=len;
	return len;
}


USB_PUBLIC usbMsgLen_t  usbFunctionSetup(uchar data[8])	//this function is called at the start of each control transfer, appropriate
flag variables are set in this function
{
	usbRequest_t    *rq = (void *)data;
	unsigned char valueType;
	dataLength=rq->wLength.bytes[0];
	requestType=rq->bRequest;
	valueType=rq->wValue.bytes[0]-1;
	switch(requestType)
	{
		case GET_MOB_NO:
		case SET_MOB_NO:
			eepromAddress=(void*)EEPROM_MOB_ADDR;
			break;
		case GET_DEV_ID:
		case SET_DEV_ID:
			eepromAddress=(void*)EEPROM_DEVICE_START_ADDR+(valueType*EEPROM_DEVICE_OFFSET);
			break;
		case GET_DEV_STATUS:
		case SET_DEV_STATUS:
			eepromAddress=(void*)EEPROM_DEVICE_STATUS_ADDR;
			break;
		default:
			break;
	}
	return 0xff;
}

/* ------------------------------------------------------------------------- */



int	usb(void)		//the main function which does usb initialisation and polling
{
	uchar   i;
	usbInit();
	usbDeviceDisconnect();  /* enforce re-enumeration, do this while interrupts are disabled! */
	i = 0;
	while(--i)
	{		/* fake USB disconnect for > 250 ms */
	    _delay_ms(1);
	}
	usbDeviceConnect();
	sei();
	for(;;)
	{		/* main event loop */
```

```
                usbPoll();
                if(!i)
                {
                        lcdClearScreen(lcdBuff);
                        lcdWriteString(lcdBuff,"USB ready...");
                }
                i--;
                _delay_ms(50);
        }
        return 0;
}

/* ------------------------------------------------------------------------- */


int main()
{
        DDRC=0xdf;
        PORTC=0x40;
        lcdInit();
        wdt_disable();
        _delay_ms(20);
        if(PINC & 0x20)
                usb();
        controller();
        return 0;
}
```

CONFIGURATION APPLICATION CODE LISTINGS

commandline c application

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include <usb_command_codes.h>
#include <lusb0_usb.h>    /* this is libusb, see http://libusb.sourceforge.net/ */

#define USBDEV_SHARED_VENDOR    0x16C0  /* VOTI */
#define USBDEV_SHARED_PRODUCT   0x05DC  /* Obdev's free shared PID */
/* Use obdev's generic shared VID/PID pair and follow the rules outlined
 * in firmware/usbdrv/USBID-License.txt.
 */

/* These are the vendor specific SETUP commands implemented by our USB device */

/* PowerSwitch uses the free shared default VID/PID. If you want to see an
 * example device lookup where an individually reserved PID is used, see our
 * RemoteSensor reference implementation.
 */

#define USB_ERROR_NOTFOUND  1
#define USB_ERROR_ACCESS    2
#define USB_ERROR_IO        3

char errorDescription[256];

static int usbOpenDevice(usb_dev_handle **device, int vendor, char *vendorName, int product, char *productName)
{
struct usb_bus      *bus;
struct usb_device   *dev;
usb_dev_handle      *handle = NULL;
int                 errorCode = USB_ERROR_NOTFOUND;
static int          didUsbInit = 0;
static char    string[256];

    if(!didUsbInit){
        didUsbInit = 1;
        usb_init();
    }
    usb_find_busses();
    usb_find_devices();
    for(bus=usb_get_busses(); bus; bus=bus->next){
        for(dev=bus->devices; dev; dev=dev->next){
            if(dev->descriptor.idVendor == vendor && dev->descriptor.idProduct == product){
                int     len;
                handle = usb_open(dev); /* we need to open the device in order to query strings */
                if(!handle){
                    errorCode = USB_ERROR_ACCESS;
                                                strcpy(errorDescription,"Warning: cannot open USB device:");
                    strcat(errorDescription, usb_strerror());
                    continue;
                }
                if(vendorName == NULL && productName == NULL){  /* name does not matter */
                    break;
                }
                /* now check whether the names match: */
                len = usb_get_string_simple(handle, dev->descriptor.iManufacturer, string, sizeof(string));
                if(len < 0){
                    errorCode = USB_ERROR_IO;
                                                strcpy(errorDescription,"Warning: cannot query manufacturer for device:");
                     strcat(errorDescription, usb_strerror());
                }else{
                                                strcat(errorDescription,"found vendor : ");
                                                strcat(errorDescription,string);
                    errorCode = USB_ERROR_NOTFOUND;
                    if(strcmp(string, vendorName) == 0){
                        len = usb_get_string_simple(handle, dev->descriptor.iProduct, string, sizeof(string));
                        if(len < 0){
                            errorCode = USB_ERROR_IO;
                                                                strcpy(errorDescription,"Warning: cannot query product for device : ");
                            strcat(errorDescription, usb_strerror());
                        }else{
                            errorCode = USB_ERROR_NOTFOUND;
                            if(strcmp(string, productName) == 0)
                                                                {
                                                                        strcat(errorDescription,"\nfound product : ");
                                                                        strcat(errorDescription,string);
                                break;
                                                                }
                        }
                    }
                }
                usb_close(handle);
                handle = NULL;
            }
        }
        if(handle)
            break;
    }
    if(handle != NULL){
        errorCode = 0;
        *device = handle;
    }
    return errorCode;
}


int main(int argc, char **argv)
{
        usb_dev_handle      *handle = NULL;
        static unsigned char       buffer[255];
        char choice;
        int                nBytes,i;

    if(usbOpenDevice(&handle, USBDEV_SHARED_VENDOR, "supratimofficio@gmail.com", USBDEV_SHARED_PRODUCT, "GSM
Control") != 0){
        fprintf(stderr, "Could not find USB device \"GSM Control\" with vid=0x%x pid=0x%x\n", USBDEV_SHARED_VENDOR,
USBDEV_SHARED_PRODUCT);
                    getch();
        exit(1);
    }
        printf("\n%s\n",errorDescription);
/* We have searched all devices on all busses for our USB device above. Now
 * try to open it and perform the vendor specific control operations for the
 * function requested by the user.
 */
	while(1)
	{
		printf("\n\nGSM control main menu :");
		printf("\n1. Get Mobile number");
		printf("\n2. Set Mobile number");
		printf("\n3. Get Device names");
		printf("\n4. Set Device names");
		printf("\n5. Get Device status");
		printf("\n6. Set Device status");
		printf("\n7. Exit\n");
		choice=getch();
		switch(choice)
		{
			case '1':
				nBytes=16;
				nBytes=usb_control_msg(handle,USB_TYPE_VENDOR | USB_RECIP_DEVICE |
USB_ENDPOINT_IN,GET_MOB_NO,0,1,buffer,nBytes,5000);
				if(nBytes>=16)
				{
					printf("\nmobile no: \"%s\"",buffer);
				}
				else
				{
					printf("\nusb read error: %s", usb_strerror());
					getch();
				}
				break;
			case '2':
				printf("\nenter mobile number in format \"+xxxxxxxxxxxx\" : ");
				gets(buffer);
				nBytes=strlen(buffer)+1;
				if(nBytes==usb_control_msg(handle,USB_TYPE_VENDOR | USB_RECIP_DEVICE |
USB_ENDPOINT_OUT,SET_MOB_NO,0,1,buffer,nBytes,5000))
				{
					printf("\nMobile number set successfully");
				}
				else
				{
					printf("\nUsb write error: %s",usb_strerror());
				}
				break;
			case '3':
				for(i=1;i<=4;i++)
				{
					nBytes=8;
					nBytes=usb_control_msg(handle,USB_TYPE_VENDOR | USB_RECIP_DEVICE |
USB_ENDPOINT_IN,GET_DEV_ID,i,1,buffer,nBytes,5000);
					if(nBytes<8)
					{
						printf("\nusb read error: %s", usb_strerror());
					}
					else
					{
						printf("\nDEVICE %d :: %s",i,buffer);
					}
				}
				break;
			case '4':
				printf("\nplease identify device by number : ");
				scanf("%d",&i);
				printf("\nplease give a name for the device (within 7 characters): ");
				fflush(stdin);
				gets(buffer);
				nBytes=strlen(buffer);
				nBytes++;
				if(nBytes>8)
				{
					printf("\nname is too big");
					break;
				}
				printf("\n%s %d\n",buffer,nBytes);
				switch(i)
				{
					case 1:
```

```
                                        nBytes=usb_control_msg(handle,USB_TYPE_VENDOR | USB_RECIP_DEVICE |
USB_ENDPOINT_OUT,SET_DEV_ID,i,1,buffer,nBytes,5000);
                                        break;
                                case 2:
                                        nBytes=usb_control_msg(handle,USB_TYPE_VENDOR | USB_RECIP_DEVICE |
USB_ENDPOINT_OUT,SET_DEV_ID,i,1,buffer,nBytes,5000);
                                        break;
                                case 3:
                                        nBytes=usb_control_msg(handle,USB_TYPE_VENDOR | USB_RECIP_DEVICE |
USB_ENDPOINT_OUT,SET_DEV_ID,i,1,buffer,nBytes,5000);
                                        break;
                                case 4:
                                        nBytes=usb_control_msg(handle,USB_TYPE_VENDOR | USB_RECIP_DEVICE |
USB_ENDPOINT_OUT,SET_DEV_ID,i,1,buffer,nBytes,5000);
                                        break;
                                default:
                                        printf("\ndevice %d is not in system",i);
                                        break;
                        }
                        if(nBytes<0)
                        {
                                printf("\nUsb write error: %s",usb_strerror());
                        }
                        break;
                case '5':
                        nBytes=usb_control_msg(handle,USB_TYPE_VENDOR | USB_RECIP_DEVICE |
USB_ENDPOINT_IN,GET_DEV_STATUS,0,1,buffer,1,5000);
                        if(nBytes)
                        {
                                for(i=0;i<4;i++)
                                {
                                        if((buffer[0]>>i)&0x01)
                                        {
                                                printf("\ndevice %d ON",i+1);
                                        }
                                        else
                                        {
                                                printf("\ndevice %d OFF",i+1);
                                        }
                                }
                        }
                        else
                        {
                                printf("\nusb read error: %s", usb_strerror());
                        }
                        break;
                case '6':
                        printf("\nplease enter the status byte value: ");
                        scanf("%d",&nBytes);
                        buffer[0]=nBytes;
                        nBytes=usb_control_msg(handle,USB_TYPE_VENDOR | USB_RECIP_DEVICE |
USB_ENDPOINT_OUT,SET_DEV_STATUS,0,1,buffer,1,5000);
                        break;
                case '7':
                        exit(0);
                default:
                        break;
        }
    }
  usb_close(handle);
  return 0;
}
```

GUI java application

Package usbio

USBIO.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package usbio;

import ch.ntb.usb.LibusbJava;
import ch.ntb.usb.USB;
import ch.ntb.usb.USBException;
import ch.ntb.usb.Usb_Bus;
import ch.ntb.usb.Usb_Device;

/**
 * @author supratim
 * This Class is contains initialization code and I/O procedures for the GSM control USB device
 * The class also contains the predefined command set for communicating with the device. In the
 * Main program an object of this class needs to be instantiated and corresponding I/O procedures
 * or informative procedures must be invoked to get the job done!
 */
public class USBIO {
    private static short idVendor= 0x16C0;
    private static short idProduct= 0x05DC;
    private static boolean deviceFound=false;
    private static long handle=0;
//The predefined command set that the GSM control usb interface understands
    public static final int GET_MOB_NO=0;
    public static final int SET_MOB_NO=1;
    public static final int GET_DEV_ID=2;
    public static final int SET_DEV_ID=3;
    public static final int SYSTEM_CHECK=4;
    public static final int GET_LOG=5;
    public static final int CLEAR_LOG=6;
    public static final int GET_DEV_STATUS=7;
    public static final int SET_DEV_STATUS=8;

    private Usb_Bus bus;    //Usb_Bus is a linked list of USB busses
    private Usb_Device dev; //Usb_Device is a linked list of USB devices
    private String vendorName;
    private String productName;


    public void open() throws USBException
    {
        LibusbJava.usb_init();  //Libusb init() method initializes various internal data structures and must be called before any other code
        LibusbJava.usb_find_busses();   //finds all USB busses
        LibusbJava.usb_find_devices();  //finds all USB devices
        bus=LibusbJava.usb_get_busses();    //Gets the USB bus previously found
        dev=bus.getDevices();   //retrieve the first libusb device on the usb bus
        while(dev!=null)    //search the entire bus for the correct device
        {
            handle=LibusbJava.usb_open(dev);
            if(LibusbJava.usb_get_string_simple(handle, 2).equals("GSM Control"))   //match device name from device descriptor
            {
                deviceFound=true;
                break;
            }
            dev=dev.getNext();
        }
        if(!deviceFound)
        {
            throw new USBException("Device not found");
        }
 else
        {
            vendorName=LibusbJava.usb_get_string_simple(handle, 1); //get device vendor name from device descriptor
            productName=LibusbJava.usb_get_string_simple(handle, 2);    //get device product name from device descriptor
 }
    }

    public Usb_Bus getBus() {
        return bus;
    }

    public Usb_Device getDev() {
        return dev;
    }
    public static boolean isDeviceFound() {
        return deviceFound;
    }

    public static long getHandle() {
        return handle;
    }

    public static short getIdProduct() {
        return idProduct;
    }

    public static short getIdVendor() {
        return idVendor;
    }

    public String getProductName() {
        return productName;
    }

    public String getVendorName() {
        return vendorName;
    }
/**
 * The usbRead() method is used for a USB read operation on the GSM Control USB interface. USB control
 * message is used for communication. The command is one of the predefined command set previously
 * identified by this class. index is a number between 1-4 identifying the four devices that GSM Control
 * is attached to. Refer the firmware implementation and USB specification for better understanding.
 */
    public String usbRead(int command,int index) throws Exception
    {
        String readStr=null;
        byte[] data=new byte[16];
        switch(command)
        {
            case GET_MOB_NO:
                LibusbJava.usb_control_msg(handle, USB.REQ_TYPE_TYPE_VENDOR | USB.REQ_TYPE_DIR_DEVICE_TO_HOST,
GET_MOB_NO, 0, 1, data, 13, 5000);
                readStr=new String(data);
                break;
            case GET_DEV_ID:
                LibusbJava.usb_control_msg(handle, USB.REQ_TYPE_TYPE_VENDOR | USB.REQ_TYPE_DIR_DEVICE_TO_HOST,
GET_DEV_ID, index, 1, data, 8, 5000);
                readStr=new String(data);
                break;
            case GET_DEV_STATUS:
                index--;
                LibusbJava.usb_control_msg(handle, USB.REQ_TYPE_TYPE_VENDOR | USB.REQ_TYPE_DIR_DEVICE_TO_HOST,
GET_DEV_STATUS, index, 1, data, 8, 5000);
                if(((data[0]>>index)&0x01)==0x01)
                {
                    readStr="ON";
                }
 else
                {
                    readStr="OFF";
 }
                break;
            default:
                throw new Exception("unknown read command");
        }
        return readStr;
    }

    /**
 * The usbWrite() method is used for a USB write operation on the GSM Control USB interface. USB control
 * message is used for communication. The command is one of the predefined command set previously
 * identified by this class. index is a number between 1-4 identifying the four devices that GSM Control
 * is attached to. Refer the firmware implementation and USB specification for better understanding.
 */
    public int usbWrite(int command,String writeStr,int index) throws USBException
    {
        byte[] data=new byte[16];
        byte[] str=writeStr.getBytes();
        for(int i=0;i<16;i++)
        {
            if(i<writeStr.length())
                data[i]=str[i];
            else
                data[i]=0;
        }
        switch(command)
        {
            case SET_MOB_NO:
                LibusbJava.usb_control_msg(handle, USB.REQ_TYPE_TYPE_VENDOR | USB.REQ_TYPE_DIR_HOST_TO_DEVICE,
SET_MOB_NO, 0, 1, data, 13, 5000);
                break;
            case SET_DEV_ID:
                LibusbJava.usb_control_msg(handle, USB.REQ_TYPE_TYPE_VENDOR | USB.REQ_TYPE_DIR_HOST_TO_DEVICE,
SET_DEV_ID, index, 1, data, 8, 5000);
                break;
            case SET_DEV_STATUS:
                LibusbJava.usb_control_msg(handle, USB.REQ_TYPE_TYPE_VENDOR | USB.REQ_TYPE_DIR_DEVICE_TO_HOST,
GET_DEV_STATUS, 0, 1, data, 8, 5000);
                index--;
                if(writeStr.equals("ON"))
                {
                    data[0]|=(byte) (1 << index);
                }
                else
                {
                    data[0]&=(byte) ~(1 << index);
                }
                LibusbJava.usb_control_msg(handle, USB.REQ_TYPE_TYPE_VENDOR | USB.REQ_TYPE_DIR_HOST_TO_DEVICE,
SET_DEV_STATUS, 0, 1, data, 1, 5000);
                break;
            default:
                throw new USBException("unknown write command");
        }
        return 0;
    }
}
```

Package gsmcontrolusb

<u>Main.java</u>

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package gsmcontrolusb;

import ch.ntb.usb.USBException;
import java.awt.Color;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import usbio.USBIO;

/**
 *
 * @author supratim
 * This is the main GUI implementation and uses the USBIO class for USB communication
 */
public class Main{

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
```

```
final JFrame frame=new JFrame("GSM Control Configuration Tool");//main gui window frame
frame.setLayout(new GridLayout(8,1));
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setLocation(200, 200);

final JPanel deviceInfo=new JPanel();

JPanel header1=new JPanel();
JLabel headerInfo1=new JLabel("Change the mobile number to which the device responds");
header1.add(headerInfo1);
header1.setBackground(Color.MAGENTA);

JPanel header2=new JPanel();
JLabel headerInfo2=new JLabel("Change the device names and current status(7 character limited names)");
header2.add(headerInfo2);
header2.setBackground(Color.ORANGE);

final JPanel panel1=new JPanel();
panel1.setBackground(Color.pink);
final JPanel panel2=new JPanel();
panel2.setBackground(Color.cyan);
final JPanel panel3=new JPanel();
panel3.setBackground(Color.pink);
final JPanel panel4=new JPanel();
panel4.setBackground(Color.cyan);
final JPanel panel5=new JPanel();
panel5.setBackground(Color.pink);

JLabel label1=new JLabel("    +91");
JLabel label2=new JLabel("Device 1");
JLabel label3=new JLabel("Device 2");
JLabel label4=new JLabel("Device 3");
JLabel label5=new JLabel("Device 4");

final JTextField mobileNumber=new JTextField();
mobileNumber.setColumns(10);
mobileNumber.setEditable(false);
final JTextField device1=new JTextField();
device1.setColumns(7);
final JTextField device2=new JTextField();
device2.setColumns(7);
final JTextField device3=new JTextField();
device3.setColumns(7);
final JTextField device4=new JTextField();
device4.setColumns(7);

JButton mobileRead=new JButton("read");
JButton dev1Read=new JButton("read");
JButton dev2Read=new JButton("read");
JButton dev3Read=new JButton("read");
JButton dev4Read=new JButton("read");

mobileRead.setActionCommand("mobileRead");
dev1Read.setActionCommand("dev1Read");
dev2Read.setActionCommand("dev2Read");
dev3Read.setActionCommand("dev3Read");
dev4Read.setActionCommand("dev4Read");

final JButton mobileChange=new JButton("change");
final JButton dev1Change=new JButton("change");
final JButton dev2Change=new JButton("change");
final JButton dev3Change=new JButton("change");
final JButton dev4Change=new JButton("change");

mobileChange.setActionCommand("mobileChange");
dev1Change.setActionCommand("dev1Change");
dev2Change.setActionCommand("dev2Change");
dev3Change.setActionCommand("dev3Change");
dev4Change.setActionCommand("dev4Change");

final USBIO usb=new USBIO();    //create a new USBIO object
try{
    usb.open();
    JLabel info=new JLabel("Device: "+usb.getProductName()+"          Developer: "+usb.getVendorName());
    deviceInfo.setBackground(Color.lightGray);
    deviceInfo.add(info);
    frame.add(deviceInfo);
}
catch(USBException e)
{
    JFrame error=new JFrame("error!");
    JLabel msg=new JLabel(e.toString());
    error.add(msg);
    error.pack();
    error.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    error.setVisible(true);
    frame.dispose();
}
final JButton disableField=new JButton("enable");
disableField.setActionCommand("disableField");
String status;

final JButton dev1Switch=new JButton();
final JButton dev2Switch=new JButton();
final JButton dev3Switch=new JButton();
final JButton dev4Switch=new JButton();

dev1Switch.setActionCommand("dev1Switch");
dev2Switch.setActionCommand("dev2Switch");
dev3Switch.setActionCommand("dev3Switch");
dev4Switch.setActionCommand("dev4Switch");

try{
if((status=usb.usbRead(USBIO.GET_DEV_STATUS, 1)).equals("ON"))
    dev1Switch.setBackground(Color.GREEN);
else
    dev1Switch.setBackground(Color.RED);
dev1Switch.setText(status);

if((status=usb.usbRead(USBIO.GET_DEV_STATUS, 2)).equals("ON"))
    dev2Switch.setBackground(Color.GREEN);
else
    dev2Switch.setBackground(Color.RED);
dev2Switch.setText(status);

if((status=usb.usbRead(USBIO.GET_DEV_STATUS, 3)).equals("ON"))
    dev3Switch.setBackground(Color.GREEN);
else
    dev3Switch.setBackground(Color.RED);
dev3Switch.setText(status);

if((status=usb.usbRead(USBIO.GET_DEV_STATUS, 4)).equals("ON"))
    dev4Switch.setBackground(Color.GREEN);
else
    dev4Switch.setBackground(Color.RED);
dev4Switch.setText(status);
}
catch(Exception e)
{
    JFrame error=new JFrame("error!");
    JLabel msg=new JLabel(e.toString());
    error.add(msg);
    error.pack();
    error.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    error.setVisible(true);
    frame.dispose();
}


mobileChange.setEnabled(false);
dev1Change.setEnabled(false);
dev2Change.setEnabled(false);
dev3Change.setEnabled(false);
dev4Change.setEnabled(false);

panel1.add(label1);
panel1.add(mobileNumber);
panel1.add(mobileRead);
panel1.add(mobileChange);
```

```
panel1.add(disableField);

panel2.add(label2);
panel2.add(device1);
panel2.add(dev1Read);
panel2.add(dev1Change);
panel2.add(dev1Switch);

panel3.add(label3);
panel3.add(device2);
panel3.add(dev2Read);
panel3.add(dev2Change);
panel3.add(dev2Switch);

panel4.add(label4);
panel4.add(device3);
panel4.add(dev3Read);
panel4.add(dev3Change);
panel4.add(dev3Switch);

panel5.add(label5);
panel5.add(device4);
panel5.add(dev4Read);
panel5.add(dev4Change);
panel5.add(dev4Switch);

frame.add(header1);
frame.add(panel1);
frame.add(header2);
frame.add(panel2);
frame.add(panel3);
frame.add(panel4);
frame.add(panel5);
try{
  mobileNumber.setText(usb.usbRead(USBIO.GET_MOB_NO, 0).substring(3));
  device1.setText(usb.usbRead(USBIO.GET_DEV_ID, 1));
  device2.setText(usb.usbRead(USBIO.GET_DEV_ID, 2));
  device3.setText(usb.usbRead(USBIO.GET_DEV_ID, 3));
  device4.setText(usb.usbRead(USBIO.GET_DEV_ID, 4));
}
catch(Exception e)
{
  JFrame error=new JFrame("error!");
  JLabel msg=new JLabel(e.toString());
  error.add(msg);
  error.pack();
  error.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  error.setVisible(true);
  frame.dispose();
}
frame.pack();
frame.setVisible(true);

// GUI code is completed here
//Event handling coed is what follows

ActionListener eventHandler=new ActionListener(){   //implented as an innerclass
  public void actionPerformed(ActionEvent e)
  {
    String actionCommand=e.getActionCommand();
    try{
      if(actionCommand.equals("mobileRead"))  //mobile read button is pressed
      {
        mobileNumber.setText(usb.usbRead(USBIO.GET_MOB_NO, 0).substring(3));
      }
      else if(actionCommand.equals("mobileChange"))   //mobile change button is pressed
      {
        String number=mobileNumber.getText().substring(0, 10);
        if(Long.parseLong(number)<=0 || number.length()!=10)
        {
          mobileNumber.setText("10 digit mob no");
        }
        else
        {
          number="+91"+number;
          usb.usbWrite(USBIO.SET_MOB_NO, number, 0);
        }
      }
      else if(actionCommand.equals("dev1Read"))   //device 1 read button is pressed
      {
        device1.setText(usb.usbRead(USBIO.GET_DEV_ID, 1).trim());
      }
      else if(actionCommand.equals("dev2Read"))   //device 2 read button is pressed
      {
        device2.setText(usb.usbRead(USBIO.GET_DEV_ID, 2).trim());
      }
      else if(actionCommand.equals("dev3Read"))   //device 3 read button is pressed
      {
        device3.setText(usb.usbRead(USBIO.GET_DEV_ID, 3).trim());
      }
      else if(actionCommand.equals("dev4Read"))   //device 4 read button is pressed
      {
        device4.setText(usb.usbRead(USBIO.GET_DEV_ID, 4).trim());
      }
      else if(actionCommand.equals("dev1Change"))   //device 1 change button is pressed
      {
        String name=device1.getText();
        if(name.length()>7)
        {
          name=name.substring(0,7);
          name=name.trim();
        }
        usb.usbWrite(USBIO.SET_DEV_ID, name.toUpperCase(), 1);
      }
      else if(actionCommand.equals("dev2Change")) //device 2 change button is pressed
      {
        String name=device2.getText();
        if(name.length()>7)
        {
          name=name.substring(0,7);
          name=name.trim();
        }
        usb.usbWrite(USBIO.SET_DEV_ID, name.toUpperCase(), 2);
      }
      else if(actionCommand.equals("dev3Change")) //device 3 change button is pressed
      {
        String name=device3.getText();
        if(name.length()>7)
        {
          name=name.substring(0,7);
          name=name.trim();
        }
        usb.usbWrite(USBIO.SET_DEV_ID, name.toUpperCase(), 3);
      }
      else if(actionCommand.equals("dev4Change")) //device 4 change button is pressed
      {
        String name=device4.getText();
        if(name.length()>7)
        {
          name=name.substring(0,7);
          name=name.trim();
        }
        usb.usbWrite(USBIO.SET_DEV_ID, name.toUpperCase(), 4);
      }
      else if(actionCommand.equals("dev1Switch")) //device 1 switch button is pressed
      {
        if(dev1Switch.getText().equals("ON"))
        {
          usb.usbWrite(USBIO.SET_DEV_STATUS, "OFF", 1);
          dev1Switch.setText("OFF");
          dev1Switch.setBackground(Color.red);
        }
        else
        {
          usb.usbWrite(USBIO.SET_DEV_STATUS, "ON", 1);
          dev1Switch.setText("ON");
          dev1Switch.setBackground(Color.green);
        }
      }
      else if(actionCommand.equals("dev2Switch")) //device 2 switch button is pressed
```

```
        {
            if(dev2Switch.getText().equals("ON"))
            {
                usb.usbWrite(USBIO.SET_DEV_STATUS, "OFF", 2);
                dev2Switch.setText("OFF");
                dev2Switch.setBackground(Color.red);
            }
            else
            {
                usb.usbWrite(USBIO.SET_DEV_STATUS, "ON", 2);
                dev2Switch.setText("ON");
                dev2Switch.setBackground(Color.green);
            }
        }
        else if(actionCommand.equals("dev3Switch")) //device 3 switch button is pressed
        {
            if(dev3Switch.getText().equals("ON"))
            {
                usb.usbWrite(USBIO.SET_DEV_STATUS, "OFF", 3);
                dev3Switch.setText("OFF");
                dev3Switch.setBackground(Color.red);
            }
            else
            {
                usb.usbWrite(USBIO.SET_DEV_STATUS, "ON", 3);
                dev3Switch.setText("ON");
                dev3Switch.setBackground(Color.green);
            }
        }
        else if(actionCommand.equals("dev4Switch")) //device 4 switch button is pressed
        {
            if(dev4Switch.getText().equals("ON"))
            {
                usb.usbWrite(USBIO.SET_DEV_STATUS, "OFF", 4);
                dev4Switch.setText("OFF");
                dev4Switch.setBackground(Color.red);
            }
            else
            {
                usb.usbWrite(USBIO.SET_DEV_STATUS, "ON", 4);
                dev4Switch.setText("ON");
                dev4Switch.setBackground(Color.green);
            }
        }
        else if(actionCommand.equals("disableField"))   //the enable/disable button is pressed
        {
            if(disableField.getText().equals("enable"))
            {
                disableField.setText("disable");
                mobileNumber.setEditable(true);
                mobileChange.setEnabled(true);
                dev1Change.setEnabled(true);
                dev2Change.setEnabled(true);
                dev3Change.setEnabled(true);
                dev4Change.setEnabled(true);
            }
            else
            {
                disableField.setText("enable");
                mobileNumber.setEditable(false);
                mobileChange.setEnabled(false);
                dev1Change.setEnabled(false);
                dev2Change.setEnabled(false);
                dev3Change.setEnabled(false);
                dev4Change.setEnabled(false);
            }
        }
    }
    catch(Exception ex)
    {
        JFrame error=new JFrame("error!");
        JLabel msg=new JLabel(ex.toString());
        error.add(msg);
        error.pack();
        error.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
                error.setVisible(true);
                frame.dispose();
            }
         }
       };

       //register the components agains the event handler for responding to the events
     disableField.addActionListener(eventHandler);
     mobileChange.addActionListener(eventHandler);
     dev1Change.addActionListener(eventHandler);
     dev2Change.addActionListener(eventHandler);
     dev3Change.addActionListener(eventHandler);
     dev4Change.addActionListener(eventHandler);
     dev1Switch.addActionListener(eventHandler);
     dev2Switch.addActionListener(eventHandler);
     dev3Switch.addActionListener(eventHandler);
     dev4Switch.addActionListener(eventHandler);
     mobileRead.addActionListener(eventHandler);
     dev1Read.addActionListener(eventHandler);
     dev2Read.addActionListener(eventHandler);
     dev3Read.addActionListener(eventHandler);
     dev4Read.addActionListener(eventHandler);
   }

}
```