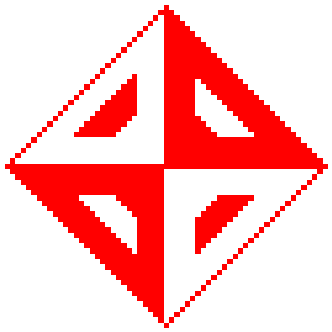
MIDDLE EAST TECHNICAL UNIVERSITY

Department of Computer Engineering



CENG 491

Initial Design Report

ComFuture Technology

Orhan Tuncer 1250851

Ugur Turan 1348028

Guven Orkun Tanik 1347947

Sebnem Sonmezler 1298231

Hakan Okten 1250562

INDEX:

1. Introduction

1.1 Purpose of This Document

This document is prepared to summarize the efforts spent on the early design phases of our project, ComFuture Bluetooth Controller. It basically covers the initial design activities and establishes a basis for the detailed design phase. In this document design concepts will be undertaken in a general sense. The detailed design will ascertain the concepts in this report and so this report will form a basis for the final design report.

1.2 Definition and Scope of the Project

Our project is to design and implement a general-purpose controller device which will allow any suitably configured Bluetooth device to control the intended device which contains our controller module. Thus our controller will be modular, easily installed, compliant with the current standards in the area. We will design and implement developer libraries to use the controller and thus the end product will depend on the specific implementation. We are going to produce a detailed and easily understandable API and a broad spectrum of library functions.

The main speciality of the product will be its generic modules and compliance with other hardware units. By this way Bluetooth won't be a close-packed technology for other developers since by this product they can easily integrate Bluetooth property to their products with ease and a very little background.

1.3 Overview of the Project Properties

Since the project has some predefined concerns it is useful to explain these concerns in order to give a more precise understanding on the overall

design. They are fundamental to understand the design goals which are explained in the next chapter (1.4).

1.3.1 Portable

Since we are designing a general purpose Bluetooth controller, the product has to be portable for a variety of devices.

If this functional requirement would not be satisfied, it will contradict with the main idea of a “general” purpose Bluetooth controller. In order to reduce the probability of the occurrence of this type of contradiction, we will consider a wide variety of devices that can be controlled by our product and try to meet their standards during the design phase.

1.3.2 Secure

Although there would be some applications for the Bluetooth controller which does not require security, we assume that our controller system should provide a level of security which can be defined by four fundamental elements: Availability, access, integrity, and confidentiality. If we assure that the connection between the controller and the device is secure under these conditions, no other third party device can interfere with the connection or interrupt, while the device and the controller is communicating without losing integrity and confidentiality.

As it is known by most of the cellular phone users interested in Bluetooth, any Bluetooth device can be realized by a phone when searched for devices. Since Bluetooth technology can communicate out of sight, this can be a critical problem in security issues. In today's cellular phones there is an authentication protocol for giving permission to connect to a Bluetooth device. This must be implemented in our product since the proof of concept can be used in critical places for security.

Also we want to add portability and generic issues to our system. As a result we must implement all the same protocols for the Bluetooth connection about security and in our libraries there must be additional security issues for the hardware which are wanted to be kept more private that a cellular phone.

1.3.3 Reusable

Our hardware will be easily pluggable and unpluggable. As long as the number of pins are enough it can be reprogrammed and used in an other context.

1.3.4 Easily Programmable

The generic component on the board will be the client application which is located on the PIC. It can be easily reprogrammed to accommodate any user needs. So it can be easily reprogrammed to be used in any context or situation. Our client side library will offer the necessary tools to develop an client application fastly and easily.

On the server side our Application Server will offer the necessary container to keep and process the Java Classes. Our server side library will offer the necessary tools to develop a Java Class fastly and easily.

1.4 Design Goals

Our design goals are driven by the facts of simplification necessities of any hardware by the means of design costs and production costs, simplification necessities of software systems by the means of design costs and maintainability as well as the concerns about the satisfaction and demands of the end users.

1.4.1 Hardware Design

We divided our hardware design process into two parts consisting of selecting the suitable PIC, which has a high importance level in our designs

including the software design which is highly connected to the PIC selection, and intermediate circuit design which is necessary to include the PIC to the current hardware, the Bluetooth development environment.

1.4.1.1 PIC

We will be working with a PIC16F877A model PIC. Its relatively high internal capacity and embedded analog to digital converter will be used to simplify the hardware part considerably.

1.4.1.2 Intermediate Circuit

We will use an extremely simple intermediate circuit which will consist of noise dampening bypass capacitors and voltage adjustment subcircuit to connect PIC and serial port of the Bluetooth controller.

1.4.2 Library Design

There will be two different libraries located at two sides of the architecture; server side and client side. This division is required because the Java Classes and the client side applications will be working on two different systems which also separate them by the means of programming languages. The server side library, as well as the application server itself, will be constructed on the Java Technology, on the other hand client side library will be developed with C language for PIC programming, which also defines the client side programming language. The details of client side programming and server side programming will be explained in chapter 4 “Library Design”.

1.4.2.1 Client side Library

It will be used at the compile time of the client side application. It will contain the necessary tools to communicate with the Bluetooth controller. We tried to develop a compact but powerful library to be used at the client side that will perform the hardware related operations in order to give the application programmer a more logical view of the system.

1.4.2.2 Server side Library

It will be used to communicate with the client application. We tried to develop a compact but powerful library by not incorporating any redundant or interlaced functions. It will give the application programmer the ability of controlling system events and develop a better business context.

1.4.3 Application Server

Application Server is the main improvement we have added to the design. It will be the main container of the Java Classes with ability to offer a better control to the system designer using our Bluetooth controller. It will be constructed on the Java Technology. This improvement will give a more flexible working environment for both the application and system designers as well as the client application itself.

1.4.4 Server Side API

It will be used to coordinate the interaction with the application server. To improve the consistency and the security of the system, an application programmer can define some rules to be forced on the application by the server. It is essential if the Java Class has various components and if a well defined business logic is required. Our server side API will give the designer these abilities by offering the complementary tools effectively using the abilities that the Application Server offers.

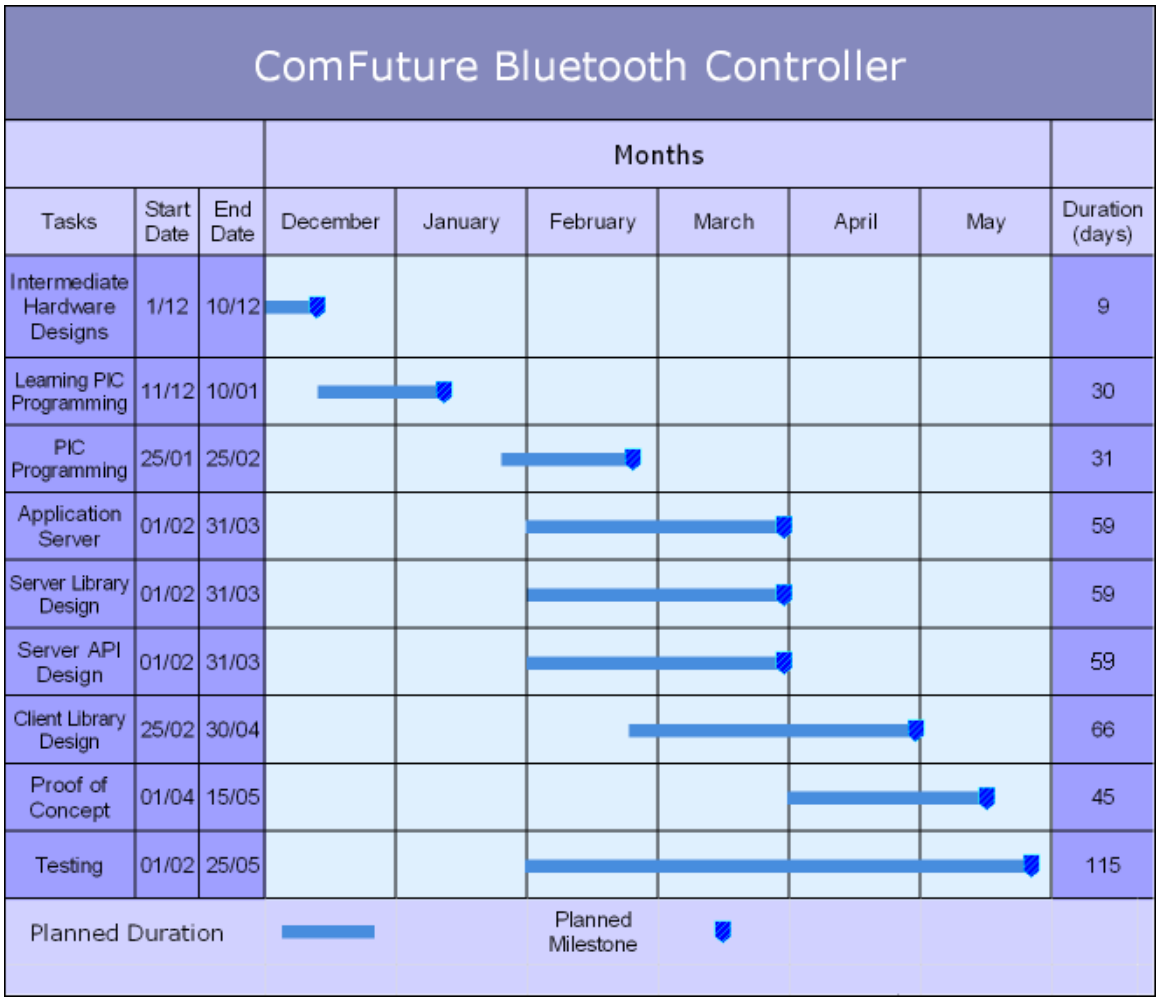
1.4.5 Proof of Concept

Our proof of concept will be a modified remote-controlled vehicle, which we will buy as a toy, but going to modify to meet our project needs. Since it will also be used as a testing and evolution tool of our design, hardware and software it needs to be simple but improvable. Some possibilities have already been started to evaluated and they will be explained later.

2. Project Schedule

We will start with intermediate hardware designs which will enable us to implement and try our PICs as we program.

We supplied our PIC programming software and hardware and also we purchased necessary bread boards and circuit components. This will allow us to concentrate on building our project rather than trying to decide on which part to use. As such we will proceed as learning and programming in order. We will do the peripheral programming first than concurrently start application server and Client programming. Also library design can not be separated from any of these. As we approach to the final phase of development we will concentrate on the proof of concept. Testing will be held during all of the development stages.



3. Architectural Overview

Our architecture has two main aspects. First one is Application Server and the second one is ComFuture Bluetooth Controller.

The Server Application in our previous model is replaced by an application server in our current model. The reason is to provide users a better intermediate tool between client application and server application. In this approach user can define multiple classes that can work together and also the client application can choose the class to work with.

In this scheme such a layout will be formed:

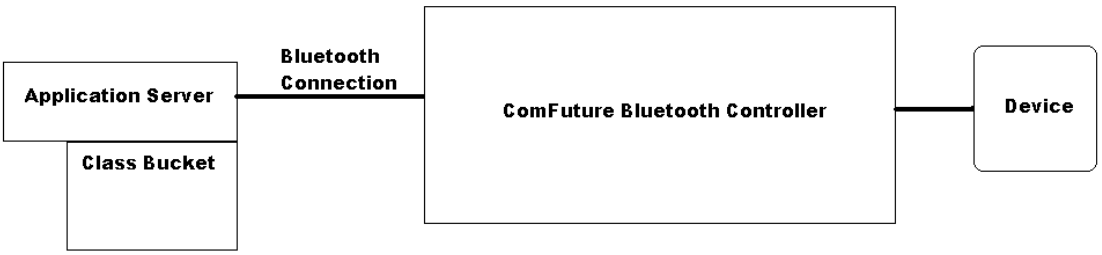


Figure - 3.1 Architectural Overview

In our Controller besides to Bluetooth communication circuit given to us we will implement the following circuits:

- Intermediate hardware to connect the serial port of the Bluetooth communication circuit. This circuit will generally condition the input to the PIC in our desired format.
- PIC16F877A as a container for the client application and intermediate processes.
- Intermediate hardware to distribute and format the output according to our specifications.
- Output Pins.

Figure 3.2 depicts this layout:

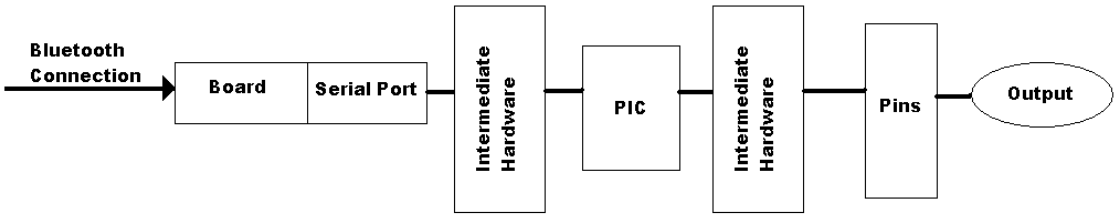


Figure - 3.2 ComFuture Bluetooth Controller Details

4. Library Design

There are two libraries in our design separated by the means of location, usage and programming language: client side and server side. The motivation of separating them is explained before so here we will be focusing on details of their designs.

4.1 Client-Side Library

Client Side Library is designed to fulfill the needs of application programmer. It consists of necessary tools which will hide the hardware connection details from the programmer. This library will be implemented in C language for PIC and the client side applications are restricted to be implemented in C language.

Client application will have the ability of choosing the server that it wants to work with and also the Java Class to handle the data that it sends. These powerful abilities will give the client application a more flexible working environment and the system designer more easily designable and maintainable environment.

Functions of this library and their explanations are given below:

FUNCTIONS	EXPLANATION
bcnt blu_Connect (char* serverID, char* ClientID)	Opens connection to Blucon server. Returns Blucon connection object which defines the path of Java Class which is decided by the server.
int blu_classPref(bcnt* con, string class_name)	Sends the request of working with a specific class located at the server.
int blu_Disconnect (bcnt* con)	Closes connection to Blucon server.
int blu_sendLayout (bcnt* con)	Sends the pin layout to the server in Blucon pins object format.
bpin blu_readLayout (void)	Reads the values of the pins and returns Blucon pins object.
int blu_writeLayout (bpin pin)	Sets the values of the pins.
void blu_onLayout (bfn* fnc)	The function to be called when new layout arrives. The function must be in bfn format.
void blu_onPinUpdate (bfn* fnc)	The function to be called when the values of the pins changes. The function must be in bfn format.
void blu_clientLoop (bmain* main)	Client side main processing loop. The function must be in bmain format.

Defined Function Formats

void bfn (bcnt* cnt, bpin pin)
int bmain (void)
void bcon (bcnt* cnt)

4.2 Server-Side Library

Server Side Library is designed to fulfill the needs of application programmer by the means of defining events and give the programmer control over these events. It is main tool that enables the programmer to communicate with the client application. Since a server must only serve we did not provided any tools which will interfere the client application.

Server side applications will be java classes resting in the container section of the application server which uses our library functions to interact with the client.

Functions of this library and their explanations are given below:

FUNCTIONS	**EXPLANATION**
int blu_sendLayout (bcnt* cnt, bpin pin)	Sends the pin layout to the client.
void blu_onConnect (bcon* con)	The function to be called when a new connection is requested. The function must be in bcon format.
void blu_onDisconnect (bcon* con)	The function to be called when a connection is closed. The function must be in bcon format.
void blu_onLayoutServe (bfn* fun)	The function to be called when new layout arrives. The function must be in bfn format.
void blu_serverLoop (bmain* main)	Server side main processing loop. The function must be in bmain format.

Defined Function Formats

void bfn (bcnt* cnt, bpin pin)
int bmain (void)
void bcon (bcnt* cnt)

5. Application Server

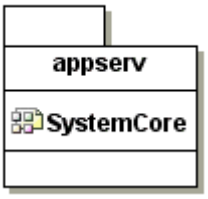
Application Server is the most powerful aspect of our design. It will be constructed on the top of the Java Technology and it will offer the necessary tools to develop a well defined system using our Bluetooth controller.

The backbone of our application server will be the Java's dynamic class loading ability. The Java Classes will be resting in the container section of our application server. After a client is connected to the server it will choose the class that it wants to work with. At this moment application server dynamically loads the requested java class and maintains the necessary linkage with the client application and Java Class. After a connection dies the linkage is halted and the Java Class is killed.

It is also possible for different clients to work with the same class. In this case each client works with a different copy of the java class and handled independently.

More detailed information about application server is below.

Appserv:

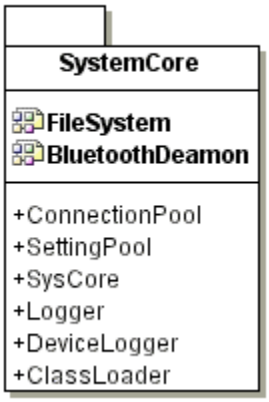


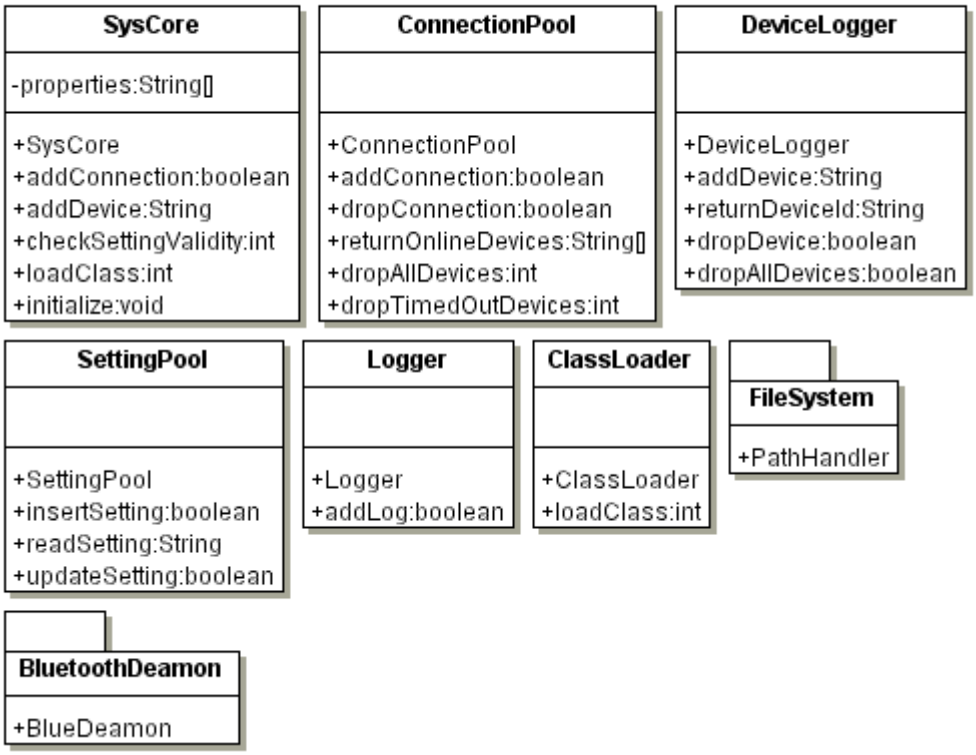
Packages	
appserv.SystemCore	
appserv.SystemCore.BluetoothDeamon	
appserv.SystemCore.FileSystem	

Class Hierarchy

- class java.lang.Object
  - class appserv.SystemCore.BluetoothDeamon.**BlueDeamon**
  - class appserv.SystemCore.**ClassLoader**
  - class appserv.SystemCore.**ConnectionPool**
  - class appserv.SystemCore.**DeviceLogger**
  - class appserv.SystemCore.**Logger**
  - class appserv.SystemCore.FileSystem.**PathHandler**
  - class appserv.SystemCore.**SettingPool**
  - class appserv.SystemCore.**SysCore**

Package appserv.SystemCore





Class Summary	
ClassLoader	Copyright: Copyright (c) 2006 Company: ComFuture Technology
ConnectionPool	Copyright: Copyright (c) 2006 Company: ComFuture Technology
DeviceLogger	Copyright: Copyright (c) 2006 Company: ComFuture Technology
Logger	Copyright: Copyright (c) 2006 Company: ComFuture Technology
SettingPool	Copyright: Copyright (c) 2006 Company: ComFuture Technology
SysCore	Copyright: Copyright (c) 2006 Company: ComFuture Technology

appserv.SystemCore

Class ClassLoader

```
java.lang.Object

      appserv.SystemCore.ClassLoader
```

Method Detail

loadClass

```
public int loadClass(java.lang.String device_id,
                     java.lang.String class_name,
                     java.lang.String path)
```

Loads a class with given name and path and marks it as "loaded" in local databese. If the class is loaded it just skips the dynamic loading process. It generates a new copy of the loaded class and writes the necassery path to reach to the 'copy' into local XML Database and gives a unique class_id to the 'copy'.

Parameters:

`device_id` - String

`class_name` - String

path - String

Returns:

int

appserv.SystemCore

Class ConnectionPool

java.lang.Object

appserv.SystemCore.ConnectionPool

Method Detail

addConnection

```
public boolean addConnection(java.lang.String device_id,
                             java.lang.String class_id)
```

it adds a new entry to connection table of the local XML database. It identifies which device uses which copy of which class. The incomming data from the device will be directed to selected class copy.

Parameters:

device_id - String

class_id - String

Returns:

boolean

---

dropConnection

```
public boolean dropConnection(java.lang.String device_id)
```

it drops the entry belongs to the device_id from the connection table.

Parameters:

`device_id` - String

Returns:

boolean

---

returnOnlineDevices

```
public java.lang.String[] returnOnlineDevices()
```

it returns the device_id's of all online devices.

Returns:

String[]

---

dropAllDevices

```
public int dropAllDevices()
```

it clears the connection table and drops all entries.

Returns:

int

---

dropTimedOutDevices

```
public int dropTimedOutDevices()
```

it scans all the connection entries and if it finds a connection is timed out, it drops that connection from the table.

Returns:

int

appserv.SystemCore

Class DeviceLogger

`java.lang.Object`

`appserv.SystemCore.DeviceLogger`

Method Detail

addDevice

`public java.lang.String` **`addDevice`**`(java.lang.String device_name)`

It adds the device_name to the local database and gives a unique id to it. This means that the device is connected to server.

Parameters:

`device_name` - String

Returns:

String

---

returnDeviceId

`public java.lang.String` **`returnDeviceId`**`(java.lang.String device_name)`

Returns the unique DeviceId reserved for that device name.

Parameters:

`device_name` - String

Returns:

String

---

dropDevice

`public boolean` **`dropDevice`**`(java.lang.String device_name)`

It deletes the device entry from database to implement the disconnection.

Parameters:

`device_name` - String

Returns:

boolean

---

dropAllDevices

```
public boolean dropAllDevices()
```

it clears the connected device table from the local database and drops all entries.

Returns:

boolean

appserv.SystemCore

Class Logger

```
java.lang.Object
      appserv.SystemCore.Logger
```

Method Detail

addLog

```
public boolean addLog(java.lang.String device_id,
                      java.lang.String class_id,
                      java.lang.String value)
```

It adds an entry to the log table in local XML database with given parameters.

This table can later be investigated to retrieve desired information.

Parameters:

`device_id` - String

`class_id` - String

`value` - String

Returns:

boolean

appserv.SystemCore

Class SettingPool

`java.lang.Object`

`appserv.SystemCore.SettingPool`

Method Detail

insertSetting

```
public boolean insertSetting(java.lang.String device_id,
                             int setting_id,
                             java.lang.String value)
```

It inserts a new entry to setting table in local XML database. These entries will be used for validity checking on settings of the class file before serving to a request from a device.

Parameters:

`device_id` - String

`setting_id` - int

`value` - String

Returns:

boolean

---

readSetting

```
public java.lang.String readSetting(java.lang.String device_id,
                                    int setting_id)
```

returns the value of a specific setting of a specific device.

Parameters:

`device_id` - String

`setting_id` - int

Returns:

String

---

updateSetting

```
public boolean updateSetting(java.lang.String device_id,
                             int setting_id,
                             java.lang.String value)
```

it changes the specific setting of a specific device.

Parameters:

`device_id` - String

`setting_id` - int

`value` - String

Returns:

boolean

appserv.SystemCore

Class SysCore

```
java.lang.Object

      appserv.SystemCore.SysCore
```

Method Detail

addConnection

```
public boolean addConnection(java.lang.String device_name)
```

it adds a connection entry to the connected device table by using ConnectionPool class. It is called after the LoadClass class and class_id parameter comes from there.

Parameters:

`device_name` - String

Returns:

boolean

---

addDevice

```
public java.lang.String addDevice(java.lang.String device_name)
```

when a new device sends connection infirmations to the server the device is added to the device list by using DeviceLogger class.

Parameters:

`device_name` - String

Returns:

String

---

checkSettingValidity

```
public int checkSettingValidity(java.lang.String device_id,
                                int setting_id,
                                java.lang.String value)
```

it checks for validity of a setting for a given class by using SettingPool class.

Parameters:

`device_id` - String

`setting_id` - String

`value` - String

Returns:

int

---

loadClass

```
public int loadClass(java.lang.String class_name)
```

it dynamically loads a class from file system. The path is stored in properties file.

Parameters:

`class_name` - String

Returns:

int

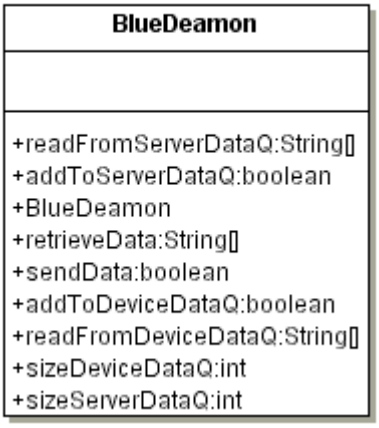
---

initialize

```
public void initialize()
```

it initializes the server at start up.

Package appserv.SystemCore.BluetoothDeamon



Class Summary

BlueDeamon	Copyright: Copyright (c) 2006 Company: ComFuture Technology

appserv.SystemCore.BluetoothDeamon

Class BlueDeamon

```
java.lang.Object
```

`appserv.SystemCore.BluetoothDeamon.BlueDeamon`

Method Detail

readFromServerDataQ

```
public java.lang.String[] readFromServerDataQ()
```

returns the oldest data in the server data to queue.

Returns:

String[]

---

addToServerDataQ

```
public boolean addToServerDataQ(java.lang.String device_name,
                                java.lang.String value)
```

when server finishes to process a data it returns it to the deamon. it is added to a queue and when the deamon is available it sends is to the device.

Parameters:

`device_name` - String

`value` - String

Returns:

boolean

---

retrieveData

```
public java.lang.String[] retrieveData()
```

server uses this function to accept data from connected devices.

Returns:

String[][][]

---

sendData

```
public boolean sendData(java.lang.String device_name,
                        java.lang.String data)
```

server uses this function to send data to connected devices.

Parameters:

`device_name` - String

`data` - String

Returns:

boolean

---

addToDeviceDataQ

```
public boolean addToDeviceDataQ(java.lang.String device_name,
                                java.lang.String value)
```

when a new info comes from a device it is added to a queue. when the server is available it asks for the oldest data in the queue and process it.

Parameters:

`device_name` - String

`value` - String

Returns:

boolean

---

readFromDeviceDataQ

```
public java.lang.String[] readFromDeviceDataQ()
```

returns the oldest entry in the device data queue

Returns:

String[]

---

sizeDeviceDataQ

```
public int sizeDeviceDataQ()
```

returns the size of device data queue

Returns:

boolean

---

sizeServerDataQ

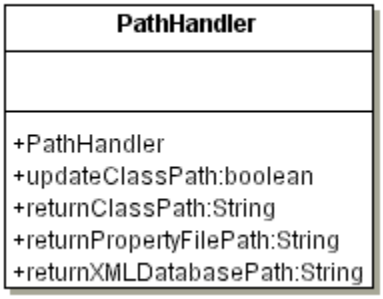
`public int` **`sizeServerDataQ`**`()`

returns the size of server data queue.

Returns:

boolean

Package appserv.SystemCore.FileSystem



Class Summary

PathHandler	Copyright: Copyright (c) 2006 Company: ComFuture Technology

appserv.SystemCore.FileSystem

Class PathHandler

`java.lang.Object`

`appserv.SystemCore.FileSystem.PathHandler`

Method Detail

updateClassPath

`public boolean **updateClassPath**(java.lang.String path)`

updates the path of stored class files in properties file.

Parameters:

`path` - String

Returns:

boolean

---

returnClassPath

`public java.lang.String **returnClassPath**()`

it returns the path of class files that are stored. When a device requests to use a class, the class is assumed to be here.

Returns:

String

---

returnPropertyFilePath

`public java.lang.String **returnPropertyFilePath**()`

at start up properties are loaded from this file. which defines the necassery paths.

Returns:

String

---

returnXMLDatabasePath

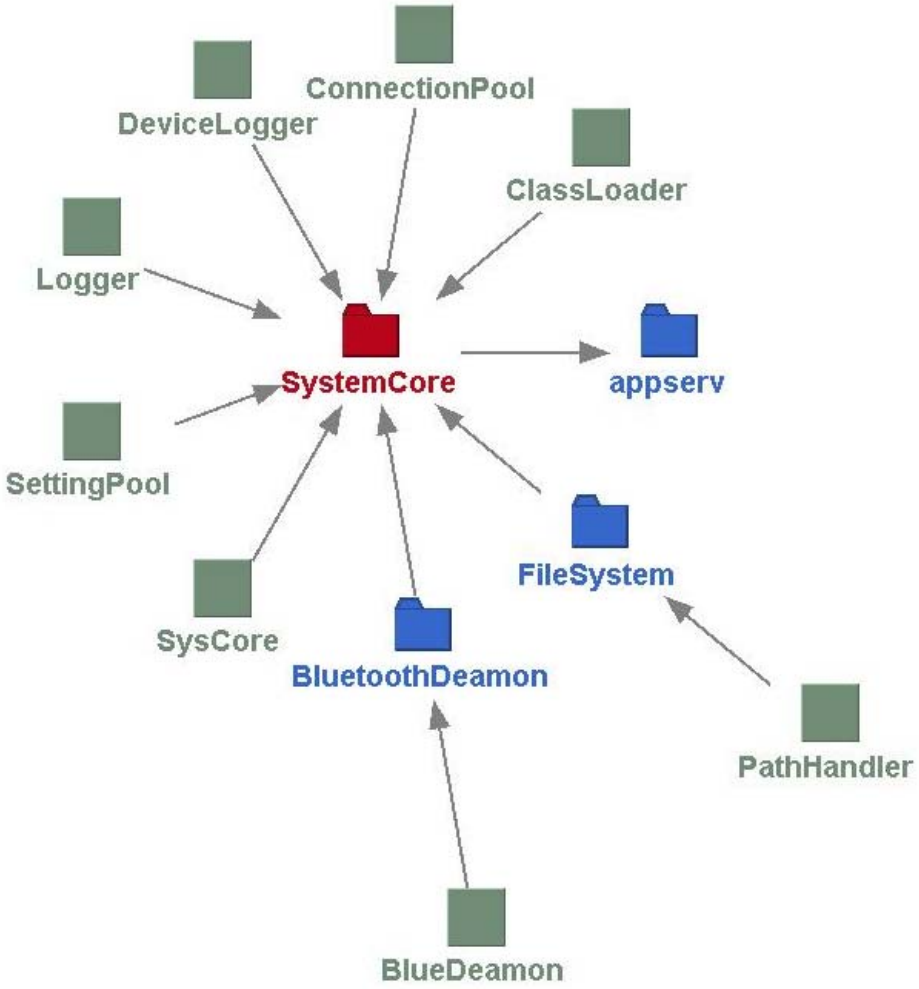
`public java.lang.String **returnXMLDatabasePath**()`

returns the path of local XML Database where the system keeps its valuable information.

Returns:

String

Structural View



Summary

The overall stability of the system is 87% . Highly stable systems are typically above 90%.

There are 12 objects, forming a total of 11 relationships. The typical object in this system immediately depends on 0.92 objects. On average, the modification of one object potentially affects 1.5 other objects.

Statistics

Property	Value
Number of Objects	12
Number of Packages	4
Number of Relationships	11
Maximum Dependencies	8
Minimum Dependencies	0
Average Dependencies	0.92
Maximum Dependents	1
Minimum Dependents	0
Average Dependents	0.92
Relationship To Object Ratio	0.92
Affects on Average	1.5

INDEX

A

B

BlueDeamon - class appserv.SystemCore.BluetoothDeamon.BlueDeamon.

Copyright: Copyright (c) 2006 Company: ComFuture Technology

BlueDeamon() - Constructor for class appserv.SystemCore.BluetoothDeamon.BlueDeamon

C

ClassLoader - class appserv.SystemCore.ClassLoader.

Copyright: Copyright (c) 2006 Company: ComFuture Technology

ClassLoader() - Constructor for class appserv.SystemCore.ClassLoader

ConnectionPool - class appserv.SystemCore.ConnectionPool.

Copyright: Copyright (c) 2006 Company: ComFuture Technology

ConnectionPool() - Constructor for class appserv.SystemCore.ConnectionPool

checkSettingValidity(String, int, String) - Method in class appserv.SystemCore.SysCore

it checks for validity of a setting for a given class by using SettingPool class.

D

DeviceLogger - class appserv.SystemCore.DeviceLogger.

Copyright: Copyright (c) 2006 Company: ComFuture Technology

DeviceLogger() - Constructor for class appserv.SystemCore.DeviceLogger

dropAllDevices() - Method in class appserv.SystemCore.ConnectionPool

it clears the connection table and drops all entries.

dropAllDevices() - Method in class appserv.SystemCore.DeviceLogger

it clears the connected device table from the local database and drops all entries.

dropConnection(String) - Method in class appserv.SystemCore.ConnectionPool

it drops the entry belongs to the device_id from the connection table.

dropDevice(String) - Method in class appserv.SystemCore.DeviceLogger

It deletes the device entry from database to implement the disconnection.

dropTimedOutDevices() - Method in class appserv.SystemCore.ConnectionPool

it scans all the connection entries and if it finds a connection is timed out, it drops that connection from the table.

I

initialize() - Method in class appserv.SystemCore.SysCore

it initializes the server at start up.

insertSetting(String, int, String) - Method in class appserv.SystemCore.SettingPool

It inserts a new entry to setting table in local XML database.

L

Logger - class appserv.SystemCore.Logger.

Copyright: Copyright (c) 2006 Company: ComFuture Technology

Logger() - Constructor for class appserv.SystemCore.Logger

loadClass(String, String, String) - Method in class appserv.SystemCore.ClassLoader

Loads a class with given name and path and marks it as "loaded" in local databese.

loadClass(String) - Method in class appserv.SystemCore.SysCore

it dynamically loads a class from file system.

P

PathHandler - class appserv.SystemCore.FileSystem.PathHandler.

Copyright: Copyright (c) 2006 Company: ComFuture Technology

PathHandler() - Constructor for class appserv.SystemCore.FileSystem.PathHandler

R

readFromDeviceDataQ() - Method in class appserv.SystemCore.BluetoothDeamon.BlueDeamon

returns the oldest entry in the device data queue

readFromServerDataQ() - Method in class appserv.SystemCore.BluetoothDeamon.BlueDeamon

returns the oldest data in the server data to queue.

readSetting(String, int) - Method in class appserv.SystemCore.SettingPool

returns the value of a specific setting of a specific device.

retrieveData() - Method in class appserv.SystemCore.BluetoothDeamon.BlueDeamon

server uses this function to accept data from connected devices.

returnClassPath() - Method in class appserv.SystemCore.FileSystem.PathHandler

it returns the path of class files that are stored.

returnDeviceId(String) - Method in class appserv.SystemCore.DeviceLogger

Returns the unique DeviceId reserved for that device name.

returnOnlineDevices() - Method in class appserv.SystemCore.ConnectionPool

it returns the device_id's of all online devices.

returnPropertyFilePath() - Method in class appserv.SystemCore.FileSystem.PathHandler

at start up properties are loaded from this file. which defines the necassery paths.

returnXMLDatabasePath() - Method in class appserv.SystemCore.FileSystem.PathHandler

returns the path of local XML Database where the system keeps its valuable information.

S

SettingPool - class appserv.SystemCore.SettingPool.

Copyright: Copyright (c) 2006 Company: ComFuture Technology

SettingPool() - Constructor for class appserv.SystemCore.SettingPool

SysCore - class appserv.SystemCore.SysCore.

Copyright: Copyright (c) 2006 Company: ComFuture Technology

SysCore() - Constructor for class appserv.SystemCore.SysCore

sendData(String, String) - Method in class
appserv.SystemCore.BluetoothDeamon.BlueDeamon

server uses this function to send data to connected devices.

sizeDeviceDataQ() - Method in class
appserv.SystemCore.BluetoothDeamon.BlueDeamon

returns the size of device data queue

sizeServerDataQ() - Method in class
appserv.SystemCore.BluetoothDeamon.BlueDeamon

returns the size of server data queue.

U

updateClassPath(String) - Method in class
appserv.SystemCore.FileSystem.PathHandler

updates the path of stored class files in properties file.

updateSetting(String, int, String) - Method in class appserv.SystemCore.SettingPool

it changes the specific setting of a specific device.

6. Server API

Server API is designed to be a middleware between Java Class and the application server. It is used to define the working conditions of the Java Classes and tell the application server how to run the Java Class. It offers the necessary tools to maintain the consistency and the security of the designed system.

Functions of the Server API and their explanations are given below:

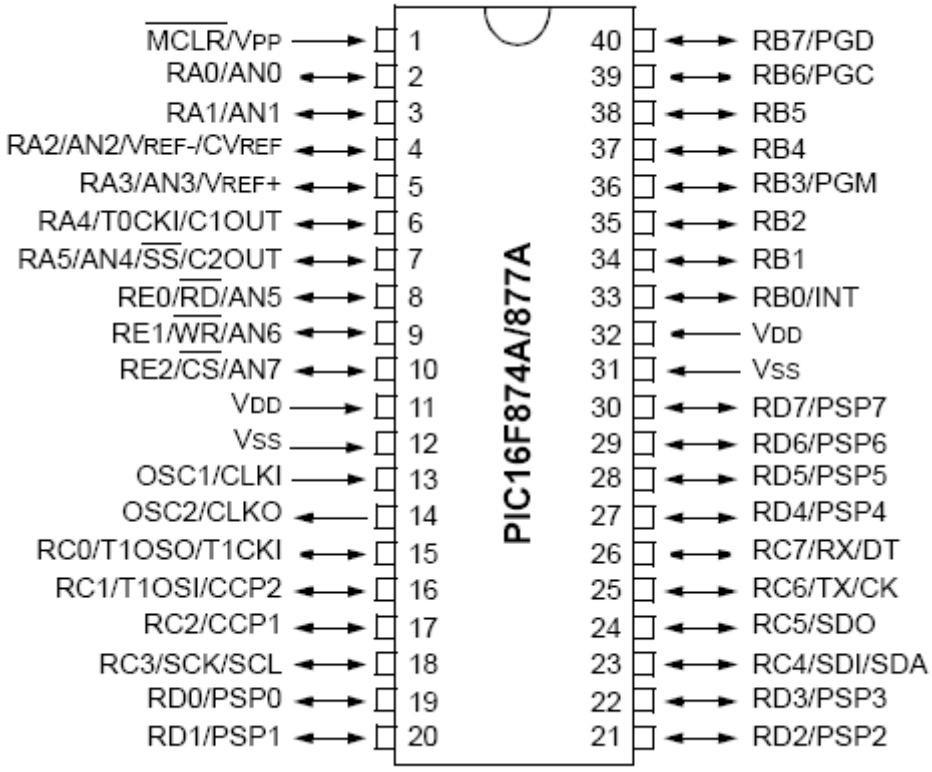
FUNCTIONS	EXPLANATION
int setMaxIdleTime(int time)	Sets the max allowed idle time for the Java Class. If no interrupt comes from client for the defined amount of time, the application is killed.
int activateLogging (string file_name)	Keep the log of transferred pin values between server and client. Data will be written to a file with the given name.
int deactivateLogging (void)	Deactivate logging.
int setMaxClient(int number)	Set the number of maximum clients that can use that class at a time.
int activateSecurity(void)	Activates the restrictions on the class to increase security.
int deactivateSecurity(void)	Deactivate security mode.
Int setAllowedDevices(string* deviceList)	Sets the allowed device identities that can use the class. No other client is allowed to work with that class.

7. Hardware Design

7.1 PIC

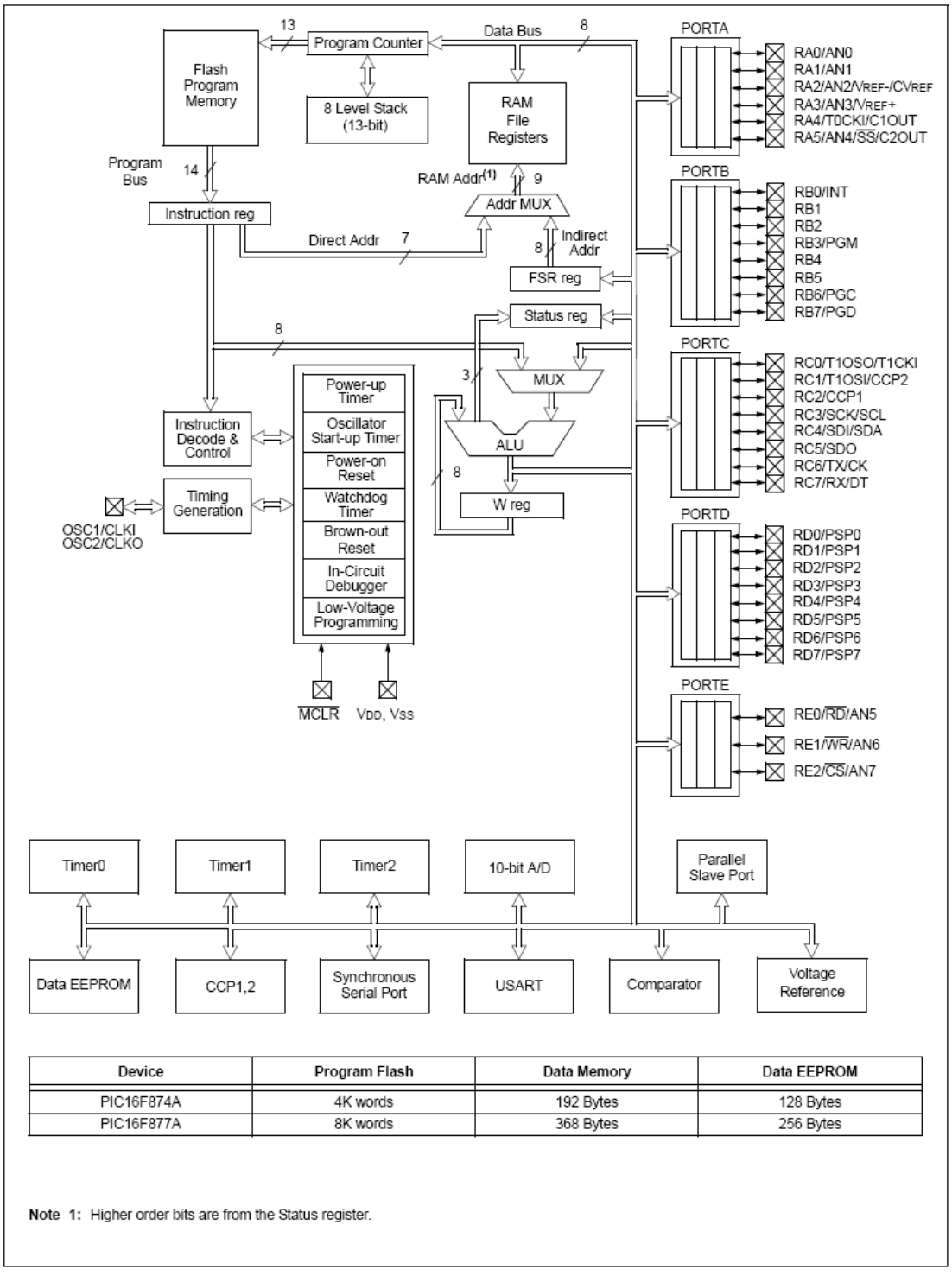
Since we need a client application on the client side and since the client will not be a computer, the application should be embedded on the client. We decided that the most suitable solution for an embedded microcontroller was a PIC. A long survey led us to choose PIC16F877A PDIP, which satisfies our needs with its 40 pins, 8K x 14 words of Flash Program Memory, 368 x 8 bytes of Data Memory (RAM), 256 x 8 bytes of EEPROM Data Memory and embedded analog-to-digital converter.

40-Pin PDIP



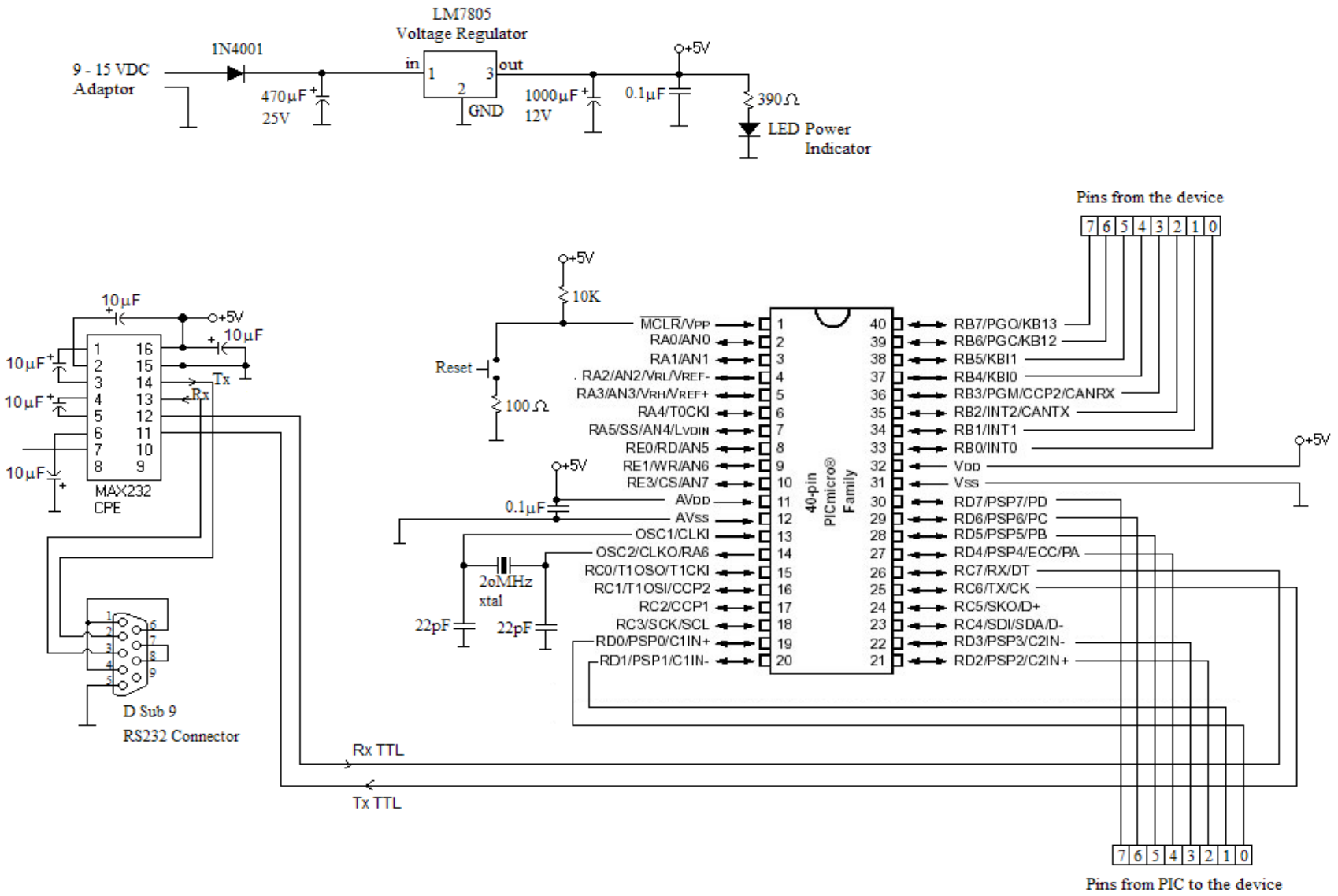
Key Features	PIC16F877A
Operating Frequency	DC – 20 MHz
Resets (and Delays)	POR, BOR (PWRT, OST)
Flash Program Memory (14-bit words)	8K
Data Memory (bytes)	368
EEPROM Data Memory (bytes)	256
Interrupts	15
I/O Ports	Ports A, B, C, D, E
Timers	3
Capture/Compare/PWM modules	2
Serial Communications	MSSP, USART
Parallel Communications	PSP
10-bit Analog-to-Digital Module	8 input channels
Analog Comparators	2
Instruction Set	35 Instructions
Packages	40-pin PDIP 44-pin PLCC 44-pin TQFP 44-pin QFN

PIC16F874A/877A BLOCK DIAGRAM



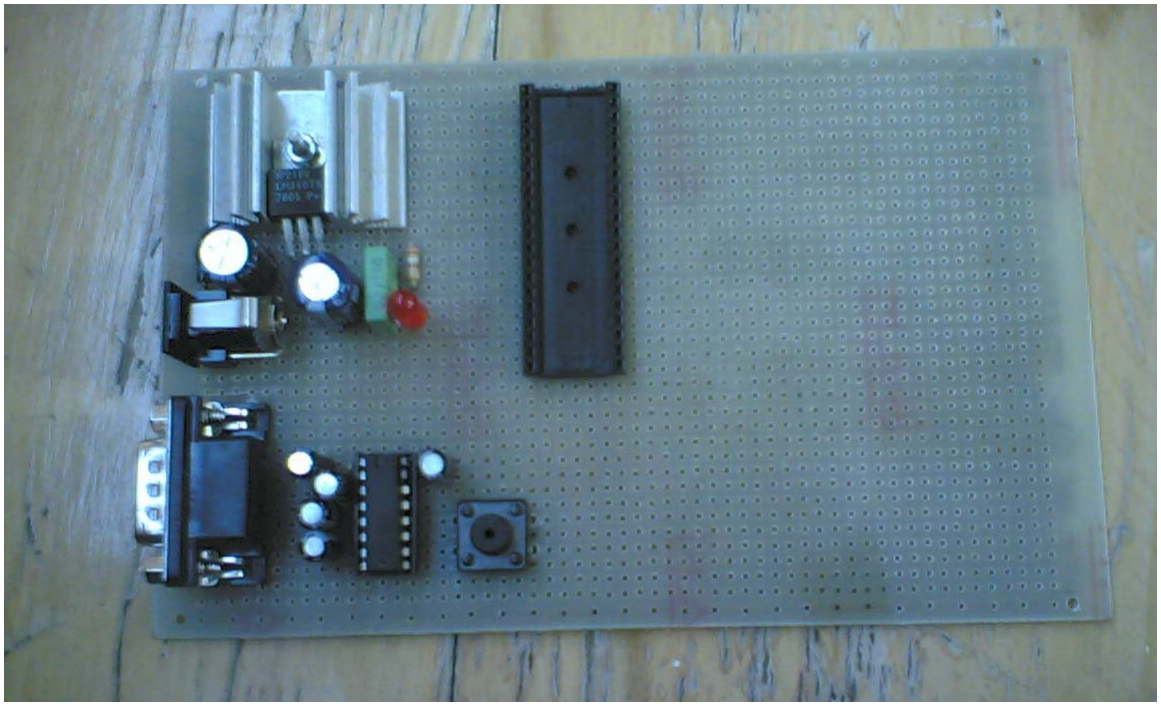
Device	Program Flash	Data Memory	Data EEPROM
PIC16F874A	4K words	192 Bytes	128 Bytes
PIC16F877A	8K words	368 Bytes	256 Bytes

Note 1: Higher order bits are from the Status register.


LM7805
Voltage Regulator
1N4001
9 - 15 VDC
Adaptor
470µF
25V
in
out
GND
1000µF
12V
0.1µF
+5V
390Ω
LED Power
Indicator
Pins from the device
10µF
MAX232
CPE
Tx
Rx
D Sub 9
RS232 Connector
Rx TTL
Tx TTL
10K
Reset
100Ω
20MHz
xtal
22pF
MCLR/VPP
RA0/AN0
RA1/AN1
RA2/AN2/VRL/VREF-
RA3/AN3/VRH/VREF+
RA4/T0CKI
RA5/SS/AN4/LVDIN
RE0/RD/AN5
RE1/WR/AN6
RE3/CS/AN7
AVDD
AVSS
OSC1/CLKI
OSC2/CLKO/RA6
RC0/T1OSO/T1CKI
RC1/T1OSI/CCP2
RC2/CCP1
RC3/SCK/SCL
RD0/PSP0/C1IN+
RD1/PSP1/C1IN-
40-pin
PICmicro®
Family
RB7/PGD/KB13
RB6/PGC/KB12
RB5/KBI1
RB4/KBI0
RB3/PGM/CCP2/CANRX
RB2/INT2/CANTX
RB1/INT1
RB0/INT0
VDD
VSS
RD7/PSP7/PD
RD6/PSP6/PC
RD5/PSP5/PB
RD4/PSP4/ECC/PA
RC7/RX/DT
RC6/TX/CK
RC5/SKO/D+
RC4/SDI/SDA/D-
RD3/PSP3/C2IN-
RD2/PSP2/C2IN+
Pins from PIC to the device

7.2 Intermediate Hardware

We designed our hardware, which will connect the BlueRadios board and the device being controlled also in which the PIC is embedded as in the figure on the previous page. Below is the photograph of the actual hardware:



7.3 Programming Software

As a result of our researches and discussions after considering various software we decided on using MPLAB v6.42 as our IDE. As it is a comprehensive editor, project manager and design desktop for application development of embedded designs using Microchip PICmicro microcontrollers. In addition, a few of the many MPLAB IDE system features are provided to help finish applications quickly.

The software that will be loaded on PIC is demonstrated below. Please nothe that the program is in a pseudocode-like C programming language and not ready to be compiled, which is left to the implementation phase of the project development.

```
/*Global variables for status of connection and connection constants*/


      char status=0; /*Connection status, 0 disconnected, 1 connected*/
      char devoutpins[8];     /*The state of the device output pins(pic
                                    to device)*/
      char devinppins[8];     /*The state of the device input
                              pins(device to pic)*/
      char rcreg1;      /*Holds the value sent by the blu_controller*/
      void (*pt2func)(void)=NULL;

      /*There will be constants corresponding to different AT commands
      defined in the pic so that this function builds up the constants
      or AT commands and connects to the server
      */

      int blu_Connect()
      {
            char in;
        /*Sequence of input and output commands (AT commands) we need*/
        Example:
            /* Get Input*/
            rcreg1=receiveChar();
            /*Write Output*/
            sendChar(deger);
            /*Finish*/
            if( "Success"  )
            { status=1; return 1;}
            status=0;
            return 0;
      }

      /* Disconnection */
      int blu_Disconnect()
      {
         /* Has the same routine as the above function*/
      }

      /* Informs the server of the classes this device intends to use*/
      int blu_ClassPref(char *classname)
      {
            char *pt=classname;
            send_char(*pt);
            return 0;
      }

      /* Updates the Global pin object.*/
      void blu_readLayout()
      {
            devinppins[0]=input(PIN_B0);
            devinppins[1]=input(PIN_B1);
            devinppins[2]=input(PIN_B2);
            devinppins[3]=input(PIN_B3);
            devinppins[4]=input(PIN_B4);
```

```
	devinppins[5]=input(PIN_B5);
	devinppins[6]=input(PIN_B6);
	devinppins[7]=input(PIN_B7);
}

/*Sends the Global pin object to the device output*/
void blu_writeLayout()
{
	char out=0;
	char i=0;
	int k=1;
	for(; i < 8 ; k=k*2,i++)
	{ out += k * devoutpins[i]; }

	sendChar(out);
}

void blu_sendLayout()
{
	char out=0;
	char i=0;
	int k=1;
	blu_readLayout();
	for(; i < 8 ; k=k*2,i++)
	{ out += k * devinppins[i]; }

	output_b(out); /*to the device*/

}


void blu_onLayout()
{
	char command=0;
	char i=0;
	int k=1;
	rcreg1=receiveChar();
	devoutpins[0]=rcreg1(0);
	devoutpins[1]=rcreg1(1);
	devoutpins[2]=rcreg1(2);
	devoutpins[3]=rcreg1(3);
	devoutpins[4]=rcreg1(4);
	devoutpins[5]=rcreg1(5);
	devoutpins[6]=rcreg1(6);
	devoutpins[7]=rcreg1(7);

	for(; i < 8 ; k=k*2,i++)
	{ command += k * devoutpins[i]; }

	/* Interpret the command, get the input for the second time
	for parameters etc., call the necessary functions*/
	/*    0 -> disconnect
		1 -> Send Layout to server
		2 -> set the internal device out pins with next input
		3 -> Read Layout
		4 -> Write Layout to device
	*/
```

```
      if(command==0) blu_Disconnect();
      if(command==1) blu_sendLayout();
      if(command==2) {while(blu_onpinupdate());
                        devoutpins[0]=rcreg1(0);
                        devoutpins[1]=rcreg1(1);
                        devoutpins[2]=rcreg1(2);
                        devoutpins[3]=rcreg1(3);
                        devoutpins[4]=rcreg1(4);
                        devoutpins[5]=rcreg1(5);
                        devoutpins[6]=rcreg1(6);
                        devoutpins[7]=rcreg1(7);
                     }
      if(command==3) blu_readLayout();
      if(command==4) blu_writeLayout();
      if(command==5) ;
      if(command==6) ;
      if(command==7) ;
      if(command==8) ;
       /* Up to 256 different commands can be implemented using
      any combination of internal commands
      or writing other code*/
}

/*  busy wait until one of the Device or controller pins are
updated, return which one is updated*/
int blu_onPinUpdate()
{
      /*Which input; 0 btcontroller, 1 device */
 start:
      if(devoutpins[0]!=rcreg1(0)) return 0;
      if(devoutpins[1]!=rcreg1(1)) return 0;
      if(devoutpins[2]!=rcreg1(2)) return 0;
      if(devoutpins[3]!=rcreg1(3)) return 0;
      if(devoutpins[4]!=rcreg1(4)) return 0;
      if(devoutpins[5]!=rcreg1(5)) return 0;
      if(devoutpins[6]!=rcreg1(6)) return 0;
      if(devoutpins[7]!=rcreg1(7)) return 0;

      if(devinppins[0]!=input(PIN_B0)) return 1;
      if(devinppins[1]!=input(PIN_B1)) return 1;
      if(devinppins[2]!=input(PIN_B2)) return 1;
      if(devinppins[3]!=input(PIN_B3)) return 1;
      if(devinppins[4]!=input(PIN_B4)) return 1;
      if(devinppins[5]!=input(PIN_B5)) return 1;
      if(devinppins[6]!=input(PIN_B6)) return 1;
      if(devinppins[7]!=input(PIN_B7)) return 1;
 go to start;
}

void sendChar(char value)
{
    while (!(txBufferIsReady()));
    txreg = value;  // Load TXREG
}
```

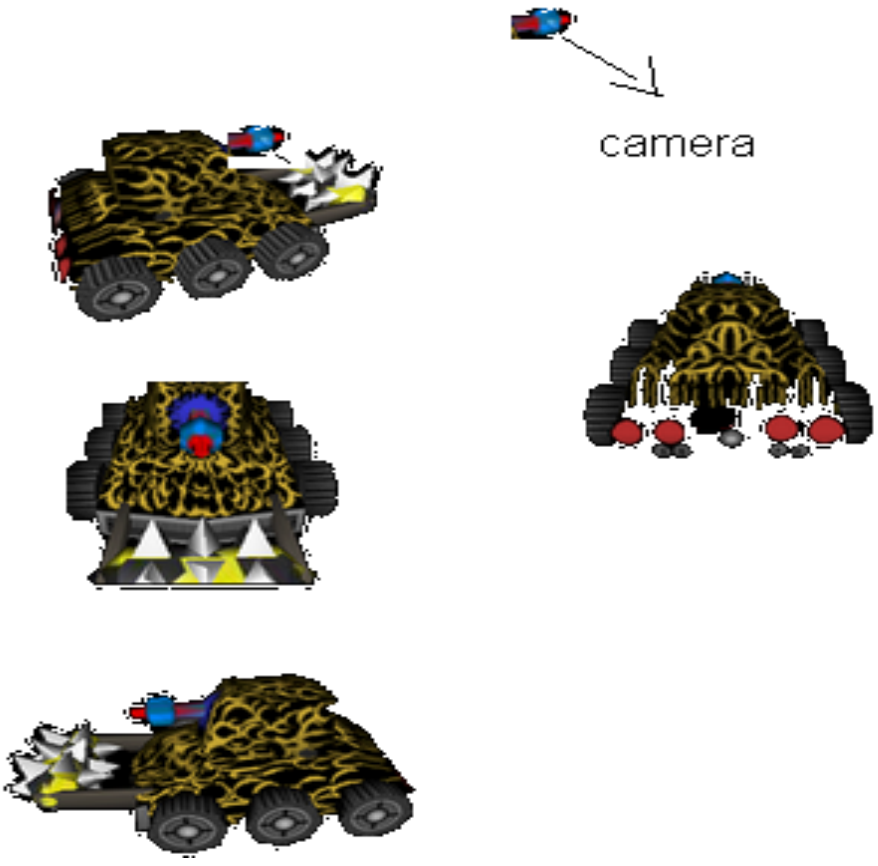
```
char receiveChar()
{
    while (!(rxBufferIsReady()));
    return rcreg;
}

void blu_ClientLoop()
{
      int changed=0;
      while(1)
      { changed=blu_onPinUpdate();
            if(changed) /*device input changed*/
                        /*Function to be called*/
                  { pt2func();}
            else blu_onLayout();
      }
}


int main(void)
{
      Char class1[20]="MyClass.class";

      /*
      Port A --> Device to picanalog input)
      Port B --> Device to pic
      Port C --> BtController to pic & Pic to BtController
                  (serial)
      Port D --> Pic to device (Parallel)
      */
      set_tris_a(255);
      set_tris_b(255);
      set_tris_c(11000000b);
      set_tris_d(0);


      txsta=0;    // Transmit Status Register-init everything to 0
      rcsta=0;    // Receive Status Register-init everything to 0

      // Now set up baud rate
      set_bit(txsta,BRGH);    // High Baud Rate Select
      spbrg=129;  /* 20MHz: 129=9600, 64=19200  (if BRGH=0 then
                        129=2400)*/

      // Set SYNC to 0 for Async mode
      clear_bit(txsta,SYNC);  // SYNC=0;  // Async Mode

      set_bit(rcsta,SPEN);    // SPEN=1;  // Serial Port Enable

      set_bit(txsta,TXEN);    // Transmit Enable

      pt2func=userdefineddevicestatechangefunction;
      blu_Connect();
      blu_classPref(class1);

      blu_ClientLoop();
}
```

8. Proof of Concept

There are many areas where we can use our controller design. We have gathered some of possible applications of our bluetooth controller. These application areas are also can be used as testing environments where we can develop our proof of concept. These documented are presented below.

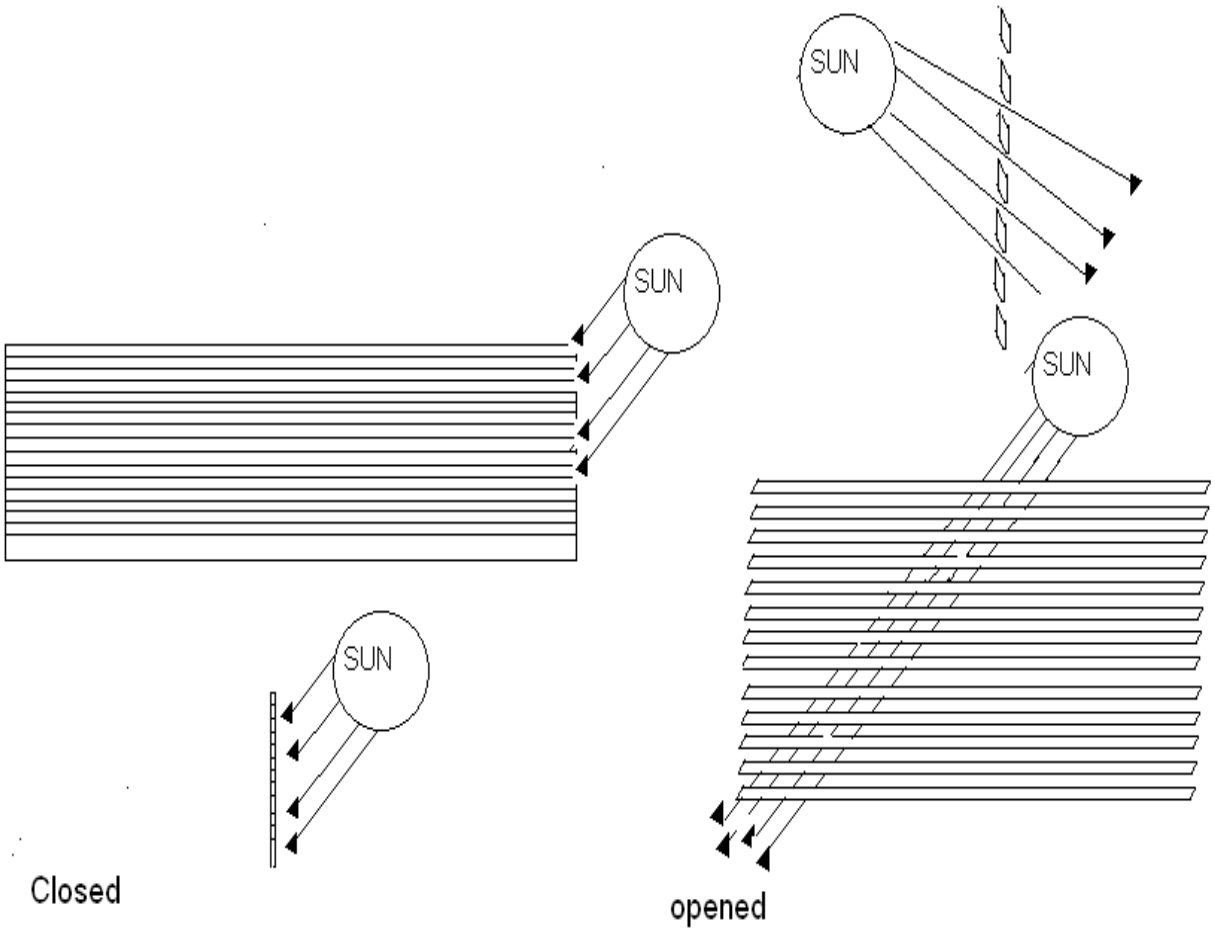
Camera controller using bluetooth

Assuming that every room in a house includes wireless camera connected to a robot car controlled by bluetooth signals. Then, we can control the cameras using bluetooth, that is, we can watch each part of the rooms by changing the position of the robot car using bluetooth. This means that, if we have a small child and we have to work in study room for log time , we can watch the child and his actions from your computer. By using such an instrument, there is no need to go to child's room and check him frequently. Because the car has ability to turn right ,turn left go directly ,turn back as we can see in the figure.



Controlling Sunblind Using Bluetooth

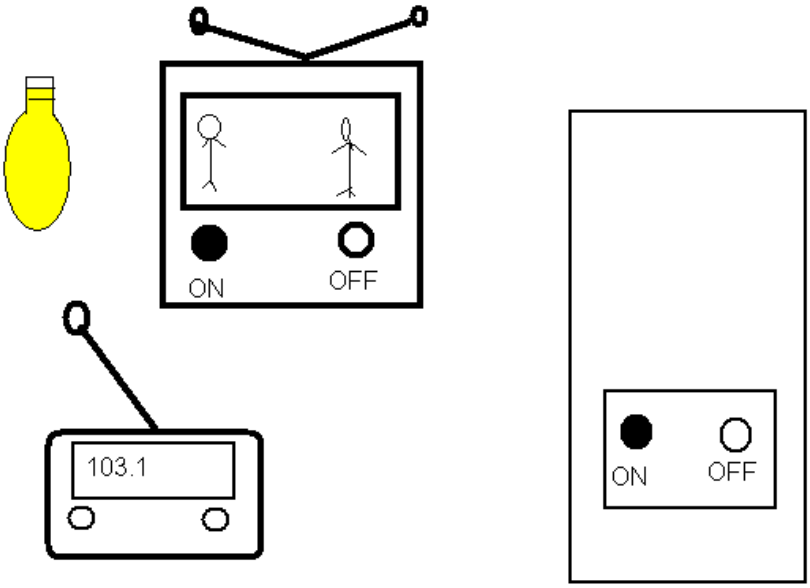
Assuming there are no curtains in the house and there are sunblinds on the windows. We know that the lights of the sun comes with different angles during the day. Because of the rays coming in different angles and ability to change the angle of sunblind in the house, we can not make use of sun light very efficiently. As we can see in the figure.



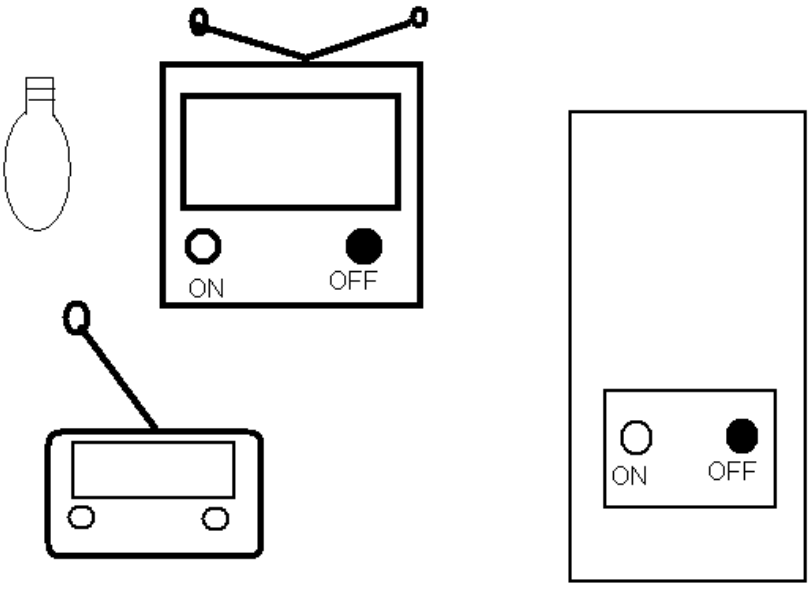
We can connect a timer and bluetooth connected to the sunblinds, we can make some arrangement with the angles of sunblind according to sun rays. With the use of timer bluetooth system, we can change the angles of sunblind in some periods automatically. In addition to this, this system can also be connected to computer and we can change the angles manually. For example, if we want to watch a movie on computer in a dark room, we can close the sunblinds manually.

A Bluetooth Controller For The Machinery At Home

We can design a box, connected to all machinery via bluetooth, next to the door inside the house. This box includes a controller for all machinery, lights and kombi. When leaving the home, we can switch of all of them or some of them by using this box. We can also control the radio and TV via this box which uses bluetooth.



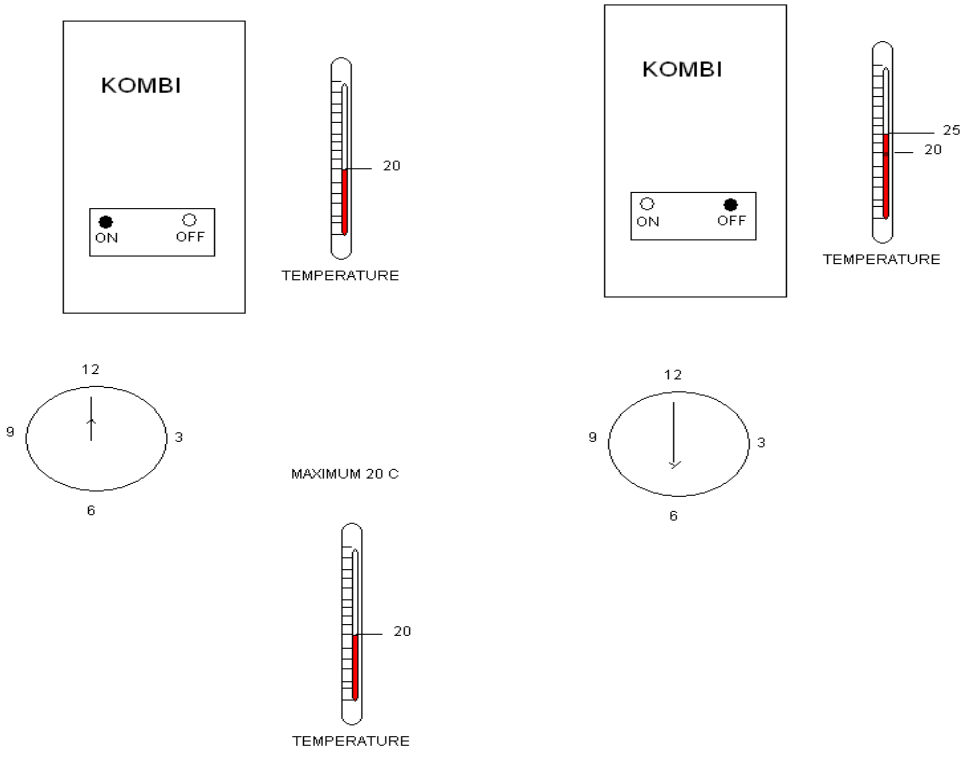
when you are at home



when you left home

Controlling Kombi via Bluetooth

We are planning to control kombi via bluetooth. There are thermometers in each room and we want to keep room temperatures between some ranges. The thermometers and the kombi are connected to each other via bluetooth. And the thermometer tells about the room temperature in each room via bluetooth. For example, we are planning to keep room temperature between 20 ^{o}C and 25 ^{o}C. The kombi starts working at 20^{o}C and goes on working till 25 ^{o}C and at 25 ^{o}C it stops working till the room temperature becomes 20 ^{o}C. When it reaches this value, it gets started to work again. By connecting a timer to this system, we can arrange the the room temperature as for example between 15 ^{o}C and 20 ^{o}C at nights. This means we can make kombi work at different temperatures at different day times. In addition to this, that we can save money. By using this system, we can arrange different temperatures for different rooms. For example, by using thermometer we can make some arrangement for the child room like 25-28 ^{o}C while it is 20 -25 ^{o}C for the sitting room. Also, we can make these arrangements by giving commands automatically, or by using computer manually. As we can see in the figure



9. Conclusion

We choosed our intended tools and built our guidelines in this report. We made throughout research about our hardware and project requirements and chose our hardware which was a great experience in our project.

This initial design report is prepared to establish a connection between our design and implementation. The information given here such as diagrams and other design products are produced in order to guide us through our way in the implementation of our project. Despite being an initial design, this document is a milestone that will
help us make our prototype and real design report. We believe that this report will contribute to our project in a quite useful way.

10. References

(1) Dynamic Class Loading in java
http://www-h.eng.cam.ac.uk/help/tpl/languages/java/javaplugins.html

(2) Dynamically Extend Java Applications
http://www.javaworld.com/javaworld/jw-08-2001/jw-0810-extend.html

(3) PIC16F877A PDIP modeled PIC documentation.

(4) MPLAB IDE documentation.

(5) Java OBEX API documentation.

(6) Sun's home for Java.
http://java.sun.com/

(7) Hands-on, how-to features and columns by Java experts; news; Java applets; sample code; tips

http://www.javaworld.com/

(8) Java[TM] technology collaboration center.

http://java.net/

(9) JavaBT:Bluetooth API for Personal Java running under Symbian OS.

http://www.sics.se/humle/projects/mobitip/javabt/tutorial.php