**CHAPTER 1**

**INTRODUCTION**

* 1. **ABOUT OUR PROJECT**

In our project we incorporate the properties and protocols of Bluetooth using J2ME (JSR-82 Library) and run the application logic on a mobile phone using MIDlets. The protocols of Bluetooth we need are implemented in Java, a popular language used on Mobile platforms. This code which successfully implements the design of our proposed system is then run as an application on a mobile phone using the MIDlets of the J2ME group. The following subsections provide the details necessary to implement the project and the chapters ahead will provide in detail the execution of our project.

* 1. **BLUETOOTH**

Bluetooth is an [open](http://en.wikipedia.org/wiki/Open_standard) [wireless](http://en.wikipedia.org/wiki/Wireless) technology standard for exchanging data over short distances (using short length radio waves) from fixed and mobile devices, creating [personal area networks](http://en.wikipedia.org/wiki/Personal_area_network) (PANs) with high levels of security. It uses a radio on chip in conjunction with software to transfer data. Invented by telecoms vendor [Ericsson](http://en.wikipedia.org/wiki/Ericsson) in 1994,it was originally conceived as a wireless alternative to [RS-232](http://en.wikipedia.org/wiki/RS-232) data cables. It can connect several devices, overcoming problems of synchronization.

Conceived initially by Ericsson, before being adopted by a lot of other companies, Bluetooth is a standard for a small, cheap radio chip to be plugged into computers, printers, mobile phones, etc. The name Bluetooth originally is named after a Danish Viking and King, Harald Blåtand (translated as Bluetooth in English), who lived in the latter part of the 10th century. Harald Blåtand united and controlled Denmark and Norway.

Bluetooth is an IEEE standard 802.15.1 protocol. The physical layer radio communicates at carrier frequencies in 2.4GHz band with FHSS (Frequency Hopping Spread Spectrum). Hopping interval is 625µs and number of hopped frequencies are 79. Data transfer can take place between 2 devices or a device with multiple devices.

Frequency-hopping spread spectrum (FHSS) is a method of transmitting radio signals by rapidly switching a carrier among many frequency channels, using a pseudorandom sequence known to both transmitter and receiver.



Fig 1.1 Layers of Bluetooth

Bluetooth consists of the following layers:

1. Radio Layer: It is responsible for transmitting the data as radio waves and deals with all aspects of physical data transfer using radio waves.
2. Baseband Layer: It is equivalent to the MAC layer in LAN. It controls the synchronization activities and FHSS operations.
3. L2CAP (Logical Link Control and Adaptation Protocol): It acts as an interface between the lower level hardware layer and the higher level software. It is responsible for managing the connections.
4. Profiles: Bluetooth specification comprises communications protocols and applications. Specifications for building interoperable applications are called profiles.

Bluetooth uses Master-Slave architecture for communication. A master Bluetooth device can communicate with up to seven devices in a [Wireless User Group](http://en.wikipedia.org/wiki/Wireless_User_Group). This network group of up to eight devices is called a [piconet](http://en.wikipedia.org/wiki/Piconet). The devices can switch roles, by agreement, and the slave can become the master at any time in the existence of many piconets.



Fig 1.2 Piconet

Bluetooth is a standard communications protocol primarily designed for low power consumption, with a short range. Because the devices use a radio (broadcast) communications system, they do not have to be in line of sight of each other. There are 3 classes of Bluetooth Radios depending on the range they offer.

|  |  |
| --- | --- |
| **Class** | **Range (approximate)** |
| **Class 1** | ~100 meters |
| **Class 2** | ~10 meters |
| **Class 3** | ~1 meters |

Table 1.1 classes of bluetooth

Bluetooth is a packet-based protocol since it uses a master-slave structure.

* 1. **J2ME (JAVA 2 MICRO EDITION)**

Java Platform, Micro Edition, or Java ME, is a Java platform designed for mobile devices and embedded systems. Target devices range from industrial controls to mobile phones (especially feature phones) and set-top boxes. Java ME was formerly known as Java 2 Platform, Micro Edition (J2ME). Java ME includes flexible user interfaces, robust security, built-in network protocols, and support for networked and offline applications that can be downloaded dynamically. Applications based on Java ME are portable across many devices, yet leverage each device's native capabilities.

Java ME devices implement a profile. The most common of these are the Mobile Information Device Profile aimed at mobile devices, such as cell phones, and the Personal Profile aimed at consumer products and embedded devices like set-top boxes and PDAs. Profiles are subsets of configurations, of which there are currently two: the Connected Limited Device Configuration (CLDC) and the Connected Device Configuration (CDC).

To date, Sun has released the following profiles:

* The Foundation Profile -- A profile for next generation consumer electronic devices
* The Mobile Information Device Profile (MIDP) -- A profile for mobile information devices, such as cellular phones and two-way pagers, and PDAs

A profile in itself does not do anything; it just defines the specification. Profiles are implemented with a configuration. You can think of a configuration as an implementation of a J2ME profile for a particular type of device such as a PDA. Some of the configurations currently available are

|  |  |
| --- | --- |
| Connected Device Configuration (CDC) | An implementation of the Foundation Profile for next-generation, consumer electronic and embedded devices |
| Connected Limited Device Configuration (CLDC) | An implementation of MIDP for small, resource-constrained devices such as Palm OS devices. |

* 1. **MIDlets**

A **MIDlet** is an application that uses the Mobile Information Device Profile (MIDP) of the Connected Limited Device Configuration (CLDC) for the Java ME environment. Typical applications include games running on mobile devices and cell phones which have small graphical displays, simple numeric keypad interfaces and limited network access over HTTP. The MIDP is a set of Java APIs designed for mobile devices such as phones and PDAs.  The APIs are included in the J2ME platform.

**CHAPTER 2**

**LITERATURE SURVEY**

**2.1 JAVA**

Java is a [programming language](http://en.wikipedia.org/wiki/Programming_language) originally developed by [James Gosling](http://en.wikipedia.org/wiki/James_Gosling) at [Sun Microsystems](http://en.wikipedia.org/wiki/Sun_Microsystems) and released in 1995 as a core component of Sun Microsystems's [Java platform](http://en.wikipedia.org/wiki/Java_(software_platform)). The language derives much of its [syntax](http://en.wikipedia.org/wiki/Syntax_(programming_languages)) from [C](http://en.wikipedia.org/wiki/C_(programming_language)) and [C++](http://en.wikipedia.org/wiki/C%2B%2B).

Java applications are typically [compiled](http://en.wikipedia.org/wiki/Compiler) to [bytecode](http://en.wikipedia.org/wiki/Java_bytecode) ([class file](http://en.wikipedia.org/wiki/Class_(file_format))) that can run on any [Java Virtual Machine](http://en.wikipedia.org/wiki/Java_Virtual_Machine) (JVM) regardless of [computer architecture](http://en.wikipedia.org/wiki/Computer_architecture). Java is general-purpose, concurrent, class-based, and object-oriented, and is specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere". Java is considered by many as one of the most influential programming languages of the 20th century, and is widely used from application software to web applications.

**2.1.1 LIFECYCLE OF JAVA**

The figure below explains the lifecycle of a Java Program. In words, the figure can be explained as:

* A Java program is written using either a Text Editor like [Textpad](http://textpad.com)or an IDE like [Eclipse](http://www.eclipse.org/) and is saved as a .java file. (Program.java)
* The .java file is then compiled using Java compiler and a .class file is obtained from it. (Program.class)
* The .class file is now portable and can be used to run this Java program in any platform.
* Class file (Program.class) is interpreted by the JVM installed on a particular platform. JVM is part of the JRE software.

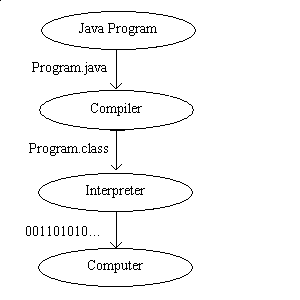
[](http://technofriends.files.wordpress.com/2007/09/java-lifecycle2.gif)

Fig 2.1 lifecycle of java

**2.2 MIDP**

**Mobile Information Device Profile** (**MIDP**) is a specification published for the use of [Java](http://en.wikipedia.org/wiki/Java_(programming_language)) on [embedded devices](http://en.wikipedia.org/wiki/Embedded_device) such as [mobile phones](http://en.wikipedia.org/wiki/Mobile_phone) and [PDAs](http://en.wikipedia.org/wiki/Personal_digital_assistant). MIDP is part of the [Java Platform, Micro Edition](http://en.wikipedia.org/wiki/Java_Platform,_Micro_Edition) (Java ME) [framework](http://en.wikipedia.org/wiki/Framework) and sits on top of [Connected Limited Device Configuration](http://en.wikipedia.org/wiki/Connected_Limited_Device_Configuration) (CLDC), a set of lower level programming interfaces. MIDP was developed under the [Java Community Process](http://en.wikipedia.org/wiki/Java_Community_Process). The first MIDP devices were launched in April 2001.

CLDC and MIDP provide the core application functionality required by mobile applications, in the form of a standardized Java runtime environment and a rich set of Java APIs. Developers using MIDP can write applications once, then deploy them quickly to a wide variety of mobile information devices. MIDP has been widely adopted as the platform of choice for mobile applications. It is deployed globally on millions of phones and PDAs, and is supported by leading integrated development environments (IDEs). Companies around the world have already taken advantage of MIDP to write a broad range of consumer and enterprise mobile applications.

MIDP stands for Mobile Information Device Profile. MIDP, combined with the Connected Limited Device Configuration (CLDC), is the Java runtime environment for today's mobile information devices (MIDs) such as phones and entry level PDAs. What MIDP provides is the core application functionality required by mobile applications - including the user interface, network connectivity, local data storage, and application lifecycle management - packaged as a standardized Java runtime environment and set of Java technology APIs.

CLDC and MIDP provide the core application functionality required by mobile applications, in the form of a standardized Java runtime environment and a rich set of Java APIs. Developers using MIDP can write applications once, then deploy them quickly to a wide variety of mobile information devices. MIDP has been widely adopted as the platform of choice for mobile applications. It is deployed globally on millions of phones and PDAs, and is supported by leading integrated development environments (IDEs). Companies around the world have already taken advantage of MIDP to write a broad range of consumer and enterprise mobile applications.

**Benefits**

* **Rich User Interface Capabilities:** MIDP applications provide the foundation for highly graphical and intuitive applications. The graphical user interface is optimized for the small display size, varied input methods, and other native features of modern mobile devices. MIDP provides intuitive navigation and data entry by taking full advantage of phone keypads, extra buttons such as arrow keys, touch screens, and small QWERTY keyboards. MIDP applications are installed and run locally, can operate in both networked and unconnected modes, and can store and manage persistent local data securely.
* **Extensive Connectivity:** MIDP enables developers to exploit the native data network and messaging capabilities of mobile information devices. It supports leading connectivity standards, including HTTP, HTTPS, datagrams, sockets, server sockets, and serial port. MIDP also supports the Short Message Service and Cell Broadcast Service capabilities of GSM and CDMA networks, through the Wireless Messaging API (WMA) optional package.
* **Multimedia and Game Functionality:** MIDP is ideal for building portable games and multimedia applications. A low-level user-interface API complements the high-level UI API, giving developers greater control of graphics and input when they need it. A game API adds game-specific functionality, such as sprites and tiled layers, which take advantage of devices' native graphics capabilities. Built-in audio provides support for tones, tone sequences, and WAV files. In addition, developers can use the Mobile Media API (MMAPI) optional package to add video and other rich multimedia content to MIDP applications.
* **Over-the-Air-Provisioning:** A major benefit of MIDP is its capability to deploy and update applications dynamically and securely, over the air.
* **End-to-End Security:** MIDP provides a robust security model that complies with open standards and protects the network, applications, and mobile information devices. HTTPS support enables applications to use existing standards such as SSL and WTLS to send and receive encrypted data.

**2.3 MIDlet**

All applications for the MID Profile must be derived from a special class, MIDlet. The MIDlet class manages the life cycle of the application. It is located in the package

**javax.microedition.midlet**

* MIDlets can be compared to J2SE applets, except that their state is more independent from the display state. A MIDlet can exist in four different states: **loaded, active, paused,** and **destroyed**. Figure 3.1 gives an overview of the MIDlet lifecycle. When a MIDlet is loaded into the device and the constructor is called, it is in the loaded state. This can happen at any time before the program manager starts the application by calling the startApp() method. After startApp() is called, the MIDlet is in the active state until the program manager calls pauseApp() or destroyApp(), pauseApp() pauses the MIDlet, and desroyApp() terminates the MIDlet. All state change callback methods should terminate quickly, because the state is not changed completely before the method returns.

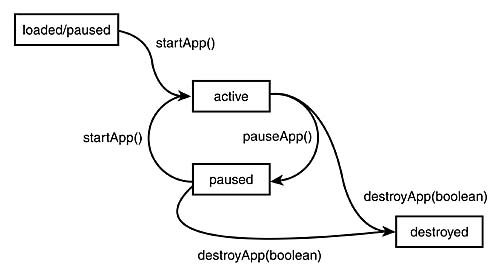


Fig 2.2 The life cycle of a MIDlet.

* In the pauseApp() method, applications should stop animations and release resources that are not needed while the application is paused. This behavior avoids resource conflicts with the application running in the foreground and unnecessary battery consumption. The destroyApp() method provides an unconditional parameter; if it is set to false, the MIDlet is allowed to refuse its termination by throwing a MIDletStateChangeException. MIDlets can request to resume activity by callingresumeRequest(). If a MIDlet decides to go to the paused state, it should notify the application manager by calling notifyPaused(). In order to terminate, a MIDlet can call notifyDestroyed(). Note that System.exit() is not supported in MIDP and will throw an exception instead of terminating the application.

**2.3.1 Display and Displayable**

* MIDlets can be pure background applications or applications interacting with the user. Interactive applications can get access to the display by obtaining an instance of the Display class. A MIDlet can get its Display instance by calling Display.getDisplay (MIDlet midlet), where the MIDlet itself is given as parameter.
* The Display class and all other user interface classes of MIDP are located in the package javax.microedition.lcdui. The Display class provides a setCurrent() method that sets the current display content of the MIDlet. The actual device screen is not required to reflect the MIDlet display immediately—the setCurrent() method just influences the internal state of the MIDlet display and notifies the application manager that the MIDlet would like to have the given Displayableobject displayed. The difference between Display and Displayable is that the Display class represents the display hardware, whereas Displayable is something that can be shown on the display. The MIDlet can call the isShown() method of Displayable in order to determine whether the content is really shown on the screen.

**2.4 JSR-82**

The overall goal of this specification is to define a standard set of APIs that will enable an open, third party application development environment for Bluetooth wireless technology. The API is targeted mainly at devices that are limited in processing power and memory, and are primarily battery-operated.

The following two packages are defined:

1. javax.bluetooth

2. javax.obex

Based on that scope, the functionality addressed by this specification can be classified into three major categories:

1. Discovery

2. Communication

3. Device Management



Fiq 2.3 Categories of JSR-82 specification

**2.4.1 Device Discovery**

An application may obtain a list of devices using either startInquiry() (nonblocking) or retrieveDevices() (blocking). The startInquiry() method requires the application to specify a listener; this listener is notified when new devices are found from a real inquiry. If an application does not wish to wait for an inquiry to begin, the API provides the retrieveDevices() method that returns the list of devices that were already found via a previous inquiry or devices that are classified as pre-known.

**Device Discovery Classes**

* *interface javax.bluetooth.DiscoveryListener*

This interface allows an application to specify an event listener that will respond to inquiry-related events. This interface is also used for service searching. The method *deviceDiscovered()* is called each time a device is found during an inquiry. When the inquiry is completed or canceled, the *inquiryCompleted()* method will be called. This method receives as an argument either the INQUIRY\_COMPLETED, INQUIRY\_ERROR or INQUIRY\_TERMINATED constant to differentiate between completed, error or canceled inquiries.

* *class javax.bluetooth.DiscoveryAgent*

This class provides methods for service and device discovery. For device discovery, this class provides the startInquiry () method to place the local device in inquiry mode and the retrieveDevices() method to return information about devices that were found via previous inquiries performed by the local device. It also provides a way to cancel an inquiry via the cancelInquiry() method.

* *class javax.bluetooth.UUID*

The class UUID encapsulates unsigned integers that are 16 bits, 32 bits or 128 bits long. The class is usedto represent a universally unique identifier used widely as the value for a service attribute. Only service attributes represented by UUIDs are searchable in Bluetooth SDP. The Bluetooth specification defines a

few “short” (16-bit or 32-bit) UUIDs and describes how a 16-bit or 32-bit UUID is converted to a 128-bit UUID. This promotion is required for matching; normally only 128-bit UUIDs are compared.

* *class javax.bluetooth.DataElement*

This class contains the various data types that a Bluetooth service attribute value can take on.

Valid service attribute data types include:

• signed and unsigned integers that are one, two, four, eight or sixteen bytes long,

• String,

• boolean,

• UUID, and

• sequences of any one of these scalar types.

The class also presents an interface to construct and retrieve the value of a service attribute. Note that all Bluetooth Service Discovery Protocol related values accessible through the Bluetooth API MUST be presented to applications in a form that requires no conversion with respect to endianness/byte order. If, in the underlying Bluetooth stack, these values have an endianness/byte order that is different from the one required in Java by the Java Virtual Machine Specification (big endian), then the Bluetooth API implementation MUST perform all necessary conversions.

* *interface javax.bluetooth.ServiceRecord*

This interface defines the Bluetooth Service Record, which contains attribute *ID, value* pairs. A Bluetooth attribute ID is a 16-bit unsigned integer and an attribute value is a DataElement. A DataElement is a self-describing value of one of the types listed in Section 5.3.2. In addition to providing the remote Bluetooth server device from which a ServiceRecord was obtained, this interface has a method populateRecord() to retrieve desired service attributes.

* *class javax.bluetooth.DiscoveryAgent*

The class DiscoveryAgent provides methods for service and device discovery. It supports service discovery in non-blocking mode and provides a way to cancel a service search transaction in progress.

* *interface javax.bluetooth.DiscoveryListener*

This interface allows an application to specify an event listener that responds to device and service discovery events. The method servicesDiscovered() is called whenever services are discovered. When a service search transaction is completed or canceled, the serviceSearchCompleted() method is called.

**2.4.2 DEVICE MANAGEMENT**

* *class javax.bluetooth.LocalDevice*

This class provides access to and control of the local Bluetooth device. It is designed to fulfill the requirements of the GAP as defined in the Bluetooth specification.

* *class javax.bluetooth.RemoteDevice*

This class represents a remote Bluetooth device. It provides basic information about a remote device, including the device’s Bluetooth address and its friendly name (Bluetooth device name).

* *class javax.bluetooth.BluetoothStateException extendsjava.io.IOException*

This exception is thrown when a device cannot honor a request that it normally supports because of the radio’s state. For example, some devices do not allow inquiry when the device is connected to another device.

* *class javax.bluetooth.DeviceClass*

This class defines values for the device type and the types of services on a device.

**2.4.3 COMMUNICATION**

To use a service on a remote Bluetooth device, the local Bluetooth device must communicate using the same protocol(s) as the remote service. So that applications can conveniently access a wide variety of Bluetooth services, APIs are provided to allow connections to services that have RFCOMM, L2CAP or OBEX as their highest-level protocol

**CHAPTER 3**

**SOFTWARE REQUIREMENT SPECIFICATION**

**3.1 FUNCTIONAL REQUIREMENTS**

**Bluetooth System Requirements**

The requirements of the underlying Bluetooth system upon which this API will be built are:

* The underlying system shall be “Qualified” in accordance with the Bluetooth Qualification Program for at least the Generic Access Profile, Service Discovery Application Profile and Serial Port Profile.
* The following layers are supported as defined in the Bluetooth specification version 1.1, and the implementation of this API has access to them.
* Service Discovery Protocol (SDP)
* RFCOMM (type 1 device support)
* Logical Link Control and Adaptation Protocol (L2CAP)
* A Bluetooth stack implementing the desired layers so that the protocol we implement uses the underlying Bluetooth Stack (WIDCOMM/BROADCOMM stack is preferable and is widely used).
* JSR-82 JAR file that contains all the APIs related to the Bluetooth implementation in Java.
* An optional Bluecove JAR file if the underlying stack cannot provide service due to missing layers.

**3.2 Non-Functional Requirements**

**Hardware:**

PROCESSOR : PENTIUM IV 2.6 GHz

RAM : 512 MB DD RAM

MONITOR : 15” COLOR

HARD DISK : 20 GB

FLOPPY DRIVE : 1.44 MB

CDDRIVE : LG 52X

KEYBOARD : STANDARD 102 KEYS

MOUSE : 3 BUTTONS

Mobile Device : Series 60.

**Software:**

Front End : MIDlets (Forms).

Tools Used : Eclipse IDE, Sun Java Wireless Toolkit 2.5.2

Operating System : Windows-XP, Linux.