4. Implementation

The modified architecture in Section 3.4 was designed and implemented based on the specifications and requirements raised in Chapters 2 and 3. The architecture was implemented and produced, both as a form of validation for the design, and as a basis for practical experimentation. The implementation chapter is split into two sections: Section 4.1 explains the back-end components and Section 4.2 describes the hardware requirements. Each section provides a detailed description of the implementation. Section 4.1 includes excerpts of programming code to explain certain functionalities of the application. A series of graphical user interfaces in Section 4.3 illustrate how the back-end and hardware components are merged to demonstrate how the application functions. Additionally, the application prototype implemented is defined.

4.1 Back-end Components

The back-end components required to fulfil the proposed architecture are programmed specifically for this implementation. The core element requires J2ME (Java 2 Micro Edition) as a software application programming interface (API) to implement all the components. As a basis of the implementation, an overview of J2ME is provided and explains the basic structure of the programming language. The architecture implementation is then discussed, detailing how the context discovery and reception layer, the context management layer and the constructive memory query layer are configured.

4.1.1 Overview of the J2ME Architecture

J2ME is a highly optimised, robust and flexible Java runtime environment aimed at the mobile device market. These devices include mobile phones, PDAs and other embedded devices. The modular design of the J2ME architecture enables an application to be scaled based on the constraints of a small computing device (Mahmoud, 2002). The J2ME architecture does not replace the operating system of a small computing device, but instead consists of layers located above the native operating system, collectively referred to as the Connected Limited Device Configuration (CLDC). The CLDC installed on top of the operating system forms the runtime environment for applications on small computing devices. It should be noted that the operating system will vary on different mobile computing devices. Applications based on the J2ME architecture are advantageous to users, as they are portable across many devices.

The J2ME architecture comprises three software layers, illustrated in Figure 4.1. The first layer is the configuration layer that includes the Java Virtual Machine (JVM), which directly interacts with the native operating system. The configuration layer also handles interactions between the profile and JVM. The second layer is the profile layer, which consists of the minimum set of APIs for the small computing device. The third layer is the Mobile Information Device Profile (MIDP), containing Java APIs for user network connections, persistent storage and user interfaces. Additionally, it has access to CLDC libraries and MIDP libraries. A mobile device has two components supplied by the original equipment manufacturer (OEM), which are classes and applications. OEM classes are used by the MIDP to access device-specific features such as sending and receiving messages and accessing device-specific persistent data.

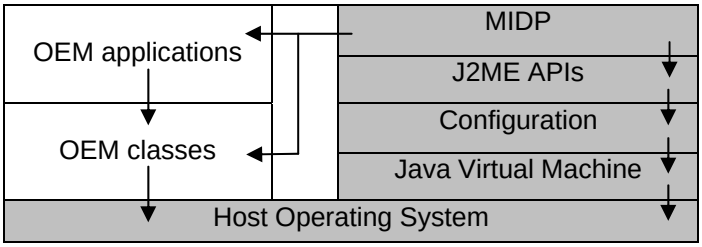

Figure 4.1. J2ME architecture layers (Keogh, 2003).

The main goal of the CLDC specification is to standardise a highly portable Java application development platform for resource-constrained, connected devices. CLDC is core technology designed to be the basis for one or more profiles. The CLDC does not define device-specific functionality in any way, but instead defines the basic Java libraries and functionality available. Table 4.1 lists the CLDC java packages.

Package	Description
`java.io`	Provides classes for input and output through data streams
`java.lang`	Provides classes that are fundamental to the programming language
`java.util`	Contains the collection classes, the date and time facilities
`java.microedition.io`	Classes for the Generic Connection Framework (GCF)

Table 4.1. CLDC 1.1 packages.

Two versions of the MIDP specification exist: MIDP 1.0 and MIDP 2.0. MIDP 2.0 is a revised version of the MIDP 1.0 specification, and includes new features such as an enhanced user interface and greater connectivity. However, MIDP 2.0 is backward compatible with MIDP 1.0, and continues to target mobile devices such as mobile phones and PDAs. The MIDP 2.0

standard defines a device with a set requirement for display, input, networking, sound and memory (see Table 4.2). The main difference between MIDP 1.0 to MIDP 2.0 is that the memory capacity is increased. In the MIDP 2.0 specification, there must be 256 KB of non-volatile memory for MIDP implementation beyond what is required for CLDC, and 128 KB of volatile memory for the Java runtime (Knudsen and Li, 2005).

MIDP 2.0 Requirements	
Display:	Screen size of at least 96 x 54 pixels, 1-bit display depth
Pixel Shape:	Approximately 1:1 aspect ratio
Input	One-handed or two-handed keyboard, touch screen
Memory:	256 KB of non-volatile memory for the MIDP components 8 KB of non-volatile memory for application-created persistent data 128 KB of volatile memory for the Java runtime environment
Networking	A two-way intermittent connection, usually wireless with limited bandwidth
Sound	The ability to play tones either via dedicated hardware or software algorithm

Table 4.2. MIDP 2.0 specification requirements (Knudsen and Li, 2005).

The MIDP is built on top of the CLDC and addresses the application lifecycle, user interface and data storage on devices which are areas omitted by the CLDC. Table 4.3 lists some of the MIDP 2.0 packages. The first package contains interfaces and classes used to build the graphical interface on the limited displays of CLDC devices. The second package adds only one class, serving as the base class for all of the MIDlets. The third package provides four interfaces, one class and five exceptions for managing persistent storage on MIDP devices. The four interfaces are important to this application to allow customisation of how the record store compares, enumerates, filters and handles the events that occur with data records.

Package	**Description**
`javax.microedition.lcdui`	Provides classes for enhanced user interfaces
`javax.microedition.midlet`	Provides classes and methods for starting, pausing and destroying applications
`javax.microedition.rms`	Provides a form of persistent storage (Record Management System)

Table 4.3. MIDP 2.0 packages.

A MIDlet is a J2ME application designed to operate on any MIDP small computing device (Keogh, 2003). A MIDlet is defined with at least a single class derived from the `javax.microedition.midlet.MIDlet` abstract class. MIDlets are usually distributed in MIDlet suites, which are contained within the same package and implemented simultaneously on a mobile computing device. A MIDlet suite is installed, executed and removed by the application manager running on the device. Once installed, each member of the MIDlet suite is given access to classes of the CLDC by the application manager. Similarly, a MIDlet can access classes defined in the MIDP to interact with the user interface and persistent storage. Additionally, the application manager makes two files available to the MIDlet suite; `.jad` and `.jar`.

All the files necessary to implement a MIDlet suite must be contained within a production package called a Java Archive (JAR) file. The JAR file contains compiled MIDlet classes, graphic images (if required by the MIDlet) and the manifest file. The manifest file contains a list of attributes and related definitions that are used by the application manager to install the files contained in the JAR file onto the mobile device. Several MIDlets may be included in a MIDlet suite. Hence, the JAR file will contain all these MIDlet classes. This enables multiple MIDlets to share resources, like common libraries included in the MIDlet suite or data stored on the device. Because of security constraints, a MIDlet may only access the resources associated with its own MIDlet suite. This applies to all resources, such as libraries it may depend on or data stored on the MID. The Java Application Descriptor (JAD) file is a plain text file containing information about a MIDlet suite (Knudsen and Li, 2005). All MIDlets must be named in this file and the size of the JAR file must be included. In addition, the MIDlet suite version number is included here. This is essential information for a MID always downloading the JAD file first to inspect its contents. If the MIDlet suite is already installed, it will know if a newer version is available. The size of the JAR file is important, as the MID can determine whether enough memory is available to install the MIDlet suite. Figure 4.2 displays the JAD file of this application.

JAD File Contents
MIDlet-1: BT_RMS_CM, BT_RMS_CM.png, BluetoothDiscovererMidlet
MIDlet-Jar-Size: 11369
MIDlet-Jar-URL: BT_RMS_CM.jar
MIDlet-Name: BT_RMS_CM
MIDlet-Vendor: Unknown
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.1
MicroEdition-Profile: MIDP-2.0

Figure 4.2. The file contents of `BT_RMS_CM.jad`.

A MIDlet is a class that extends the MIDlet class and is the interface between the application statements and the run-time environment controlled by the application manager. A MIDlet class must contain three abstract methods called by the application manager to manage the lifecycle of the MIDlet on the mobile device. These abstract methods are `startApp()`, `pauseApp()` and `destroyApp()`. When a MIDlet is first initiated, it is placed in the pause state, and only when it is ready does the controlling software place the MIDlet in the active state (Figure 4.3).

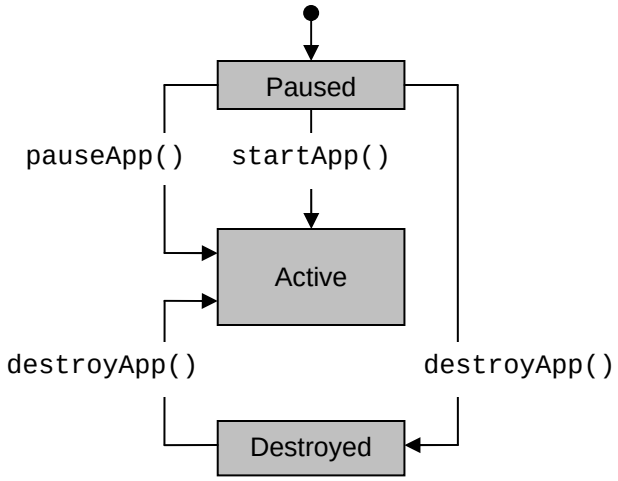

Figure 4.3. The application manager calls the methods of a MIDlet (Keogh, 2003).

The `startApp()` method is called by the application manager when the MIDlet is started and contains the statements that are executed each time the application begins. The MIDlet will typically initialise any objects required while the MIDlet is active and then set the current display. The `pauseApp()` method is called before the application manager temporarily stops the MIDlet. This means it will pause any thread currently active as well as optionally setting the next display to be shown when the MIDlet restarts. The application manager restarts the MIDlet by recalling the `startApp()` method. The `destroyApp()` method is called prior to the termination of a MIDlet by the application manager. The method will free or close all resources that have been acquired during the life of the MIDlet. Additionally the method will persist with any data that the user may wish to be saved for future use. Both the `startApp()` and `pauseApp()` methods are public and have no return value nor parameter list. The `destroyApp()` method is also a public method without a return value. However, the `destroyApp()` method has a Boolean parameter set to `true` if the termination of our MIDlet is unconditional and false if the MIDlet can throw a `MIDletStateChangeException`, telling the application manager the MIDlet does not want to be destroyed just yet (Keogh, 2003). Figure 4.4 shows an excerpt of code calling the methods in this application.

Code Excerpt

```
public void startApp()
{
Display.getDisplay(this).setCurrent(mainList);
}

public Display getDisplay()
{
return Display.getDisplay(this);
}
public void pauseApp() {}

public void destroyApp(boolean unconditional) {}
```

Figure 4.4. The methods called by the application manager from the `BluetoothDiscovererMidlet`.

The class diagram, illustrated in Figure 4.5, depicts the static relation among the classes in the overall implementation. In this object oriented application, classes have attributes (member variables), operations (member functions) and relationships with other classes. The fundamental element of the class diagram, is an icon that represents a class. A class icon, is simply a rectangle divided into three compartments. The topmost compartment contains the name of the class. The middle compartment contains a list of attributes (member variables) of the class. The bottom compartment, contains a list of the methods of the class or simply operations (member functions) done by the class. In this diagram (see Figure 4.5), the attributes are shown as `attribute_name : attribute_type` and the methods are shown as `method_name(parameter_name : parameter_type, )`.

The main MIDlet class is `BluetoothDiscovererMidlet`. It does the jobs for discovery phase, the RMS phase and also on the memory query layer, with the help of the other three classes. There is a bi-directional association between `BluetoothDiscovererMidlet` and `DeviceDiscoverer`, which means `BluetoothDiscovererMidlet` has a `DeviceDiscoverer` and vice versa. In other words, both of them can send messages to the other. There is also similar association between `BluetoothDiscovererMidlet` and `ServiceDiscoverer`. There is a directional association from `ServiceDiscoverer` to `BlueToothDevice`, which means `ServiceDiscoverer` has a `BlueToothDevice`, but `BlueToothDevice` does not have any `ServiceDiscoverer`.

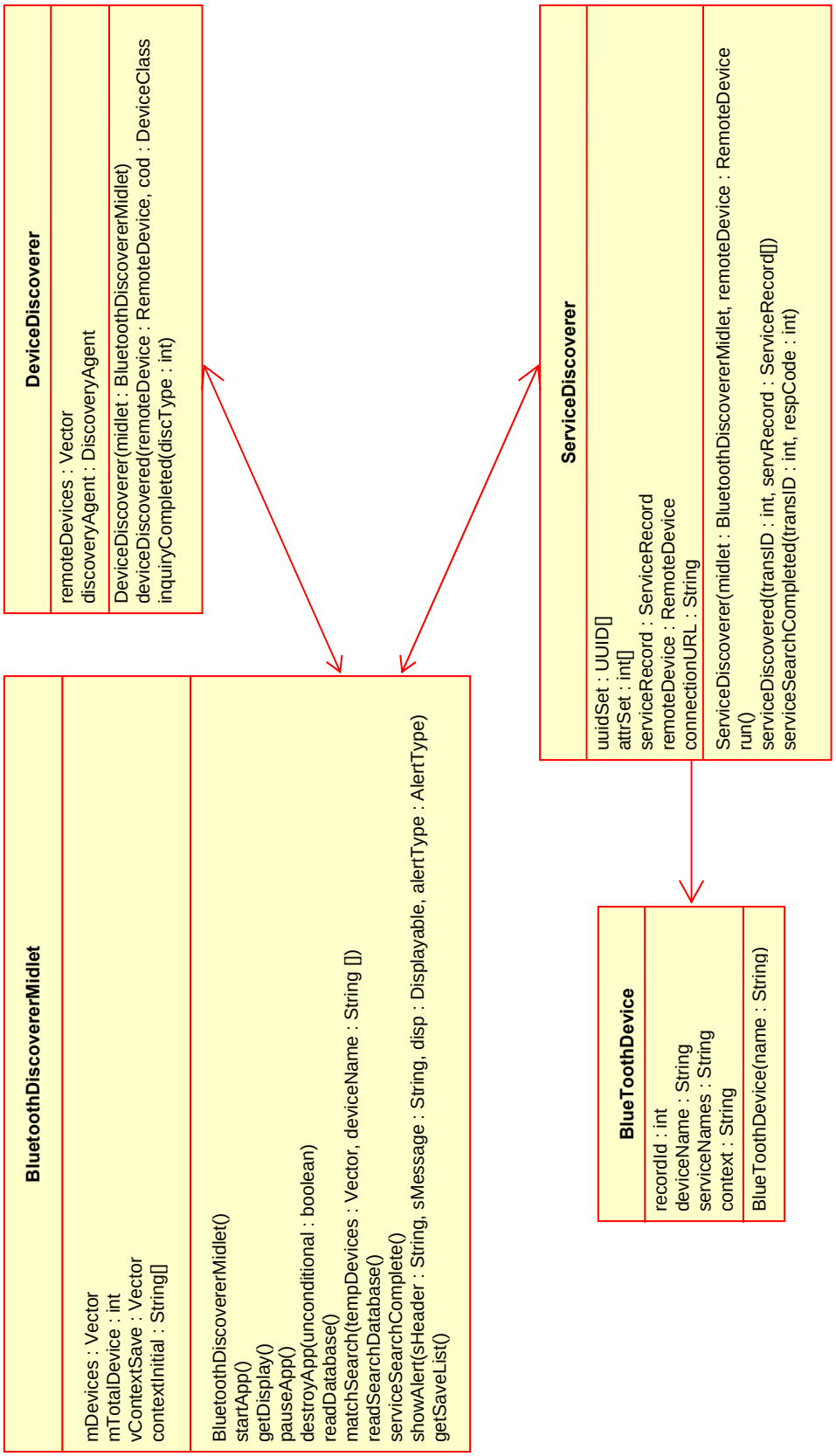

Figure 4.5. A class diagram illustrating the class structure of the application.

Table 4.4 lists the J2ME technology components and their relation to a specific layer in the architecture. After the MIDlet is initialised, the discovery of percepts (devices) begins. Section 4.1.2 details the J2ME implementation of the context discovery and reception layers in the modified architecture.

Architecture Layer	J2ME Component
Context Discovery and Reception Layer	Bluetooth API - Device Discovery - Service Discovery
Context Management Layer	Record Management System
Constructive Memory Query Layer	Constructive Memory Algorithm

Table 4.4. Modified architecture layer and its correlating J2ME component.

The next three sub-sections of Section 4.1 present the specific implementation for each architecture layer described in Section 3.4.

4.1.2 Context Discovery and Reception Layers

One of the key components of the implementation is the context discovery and reception layers. The functionalities of the two layers are merged for the purposes of the prototype application. The device discovery and sensing technology chosen for implementation is Bluetooth. Bluetooth is a radio standard and communications protocol and allows devices to communicate when in range. Since Bluetooth uses a radio communication system, devices do not need to be in the line of sight of each other, unlike infrared systems explored in the literature. Bluetooth is chosen as the sensing technology for the discovery of devices in contexts as well as for its ability to function on mobile devices. An overview of the Bluetooth infrastructure is provided in relation to J2ME, explaining the Bluetooth protocol stack. The implementations of the context discovery and reception layers are then presented using the Bluetooth technology. In this application, a percept is defined as a *device* and *service* discovered in a context. This definition is provided to address any concerns when these terms – percept or device – are referred to throughout this thesis.

The Java APIs for Bluetooth Wireless Technology (JABWT) define an optional J2ME package for Bluetooth wireless technology. It is also known as JSR-82, the Java specification request (JSR) for Bluetooth APIs. Figure 4.6, illustrates how the JABWT operates on top of the CLDC and is intended to extend the capabilities of profiles like the MIDP. JABWT uses the GCF, defined in the CLDC specification for Bluetooth communication. JABWT consists of two packages listed in Table 4.5. These packages are separate optional packages, so the CLDC implementation may include either of the packages or both of them. The `javax.bluetooth` package provides an API for both device discovery and service discovery (Hopkins and Antony, 2003). Additionally, it

provides functionality for setting up services and customisation of local service records. The `javax.obex` package provides an API for the Object Exchange (OBEX) protocol (Hopkins and Antony, 2003). This package is not tied to the Bluetooth API alone but is intended to be of more general use.

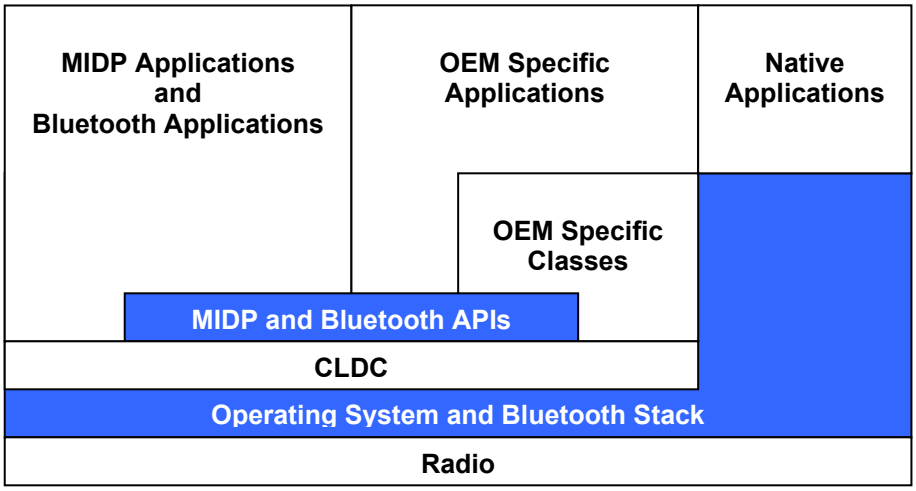

Figure 4.6. High level architecture of J2ME CLDC/MIDP and Bluetooth (Kumar et al., 2003).

Package	Description
`javax.bluetooth`	The core Bluetooth API
`javax.obex`	The Object Exchange (OBEX) API

Table 4.5. The Java APIs for Bluetooth Wireless Technology (JABWT).

The Bluetooth specification aims to allow Bluetooth devices from different manufacturers to communicate with one another, so it is not sufficient to specify just a radio system. Because of this, the Bluetooth specification does not only outline a radio system but a complete protocol stack to ensure that Bluetooth devices can discover each other, explore each other's services, and make use of these services. The Bluetooth protocol stack is the software or firmware component that has direct access to the Bluetooth device. It has control over device settings, communication parameters and power levels for the Bluetooth device. The Bluetooth stack consists of many layers, as shown in Figure 4.7, and each layer of the stack has a specific task in the overall functionality of the Bluetooth device. The Host Controller Interface (HCI) is usually the layer separating hardware from software and is implemented partially in software and hardware/firmware. The layers below the HCI are usually implemented in hardware and the layers above the HCI are usually implemented in software (Hopkins and Antony, 2003). It should be noted that Bluetooth device manufacturers are not required to use all of the layers in the stack.

However, the main layers are implemented in almost every Bluetooth device and their responsibilities are detailed below.

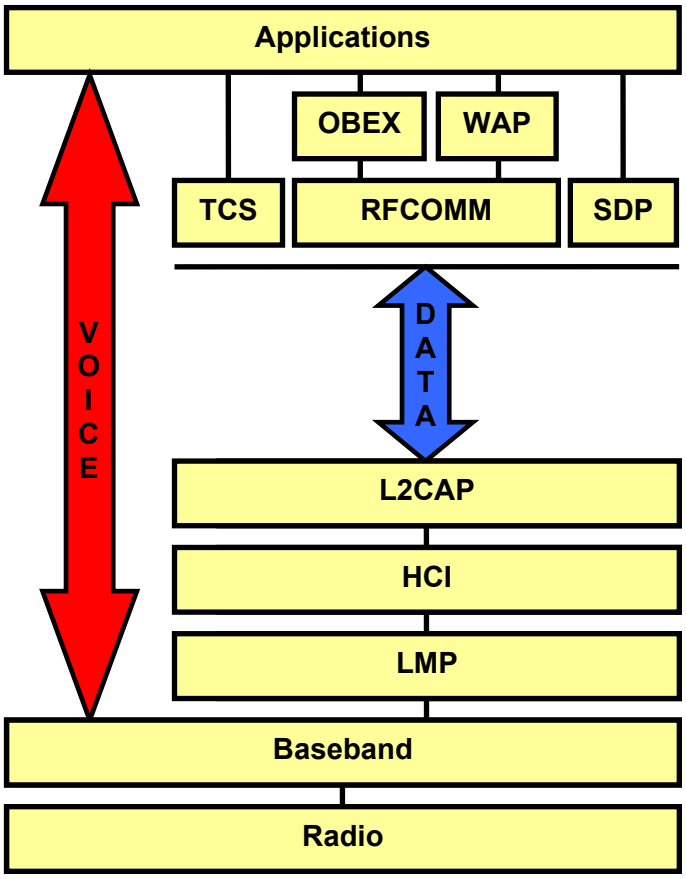

Figure 4.7. Bluetooth Protocol Stack (Bray and Sturman, 2002).

The Radio layer is the physical wireless connection. To avoid interference with other devices that communicate in the ISM band, the modulation is based on fast frequency hopping. Bluetooth divides the 2.4 GHz frequency band into 79 channels 1 MHz apart (from 2.402 to 2.480 GHz), and uses this spread spectrum to hop from one channel to another, up to 1,600 times a second (Kumar et al., 2003). The standard average wavelength is 10 centimetres to 10 metres and can be extended to 100 metres by increasing transmission power.

The Baseband is the physical layer of the Bluetooth responsible for controlling and sending data packets over the radio link. It manages physical channels and links apart from other services like error correction, data-whitening, hop selection and Bluetooth security. The Baseband layer lies on top of the Bluetooth radio layer in the stack. The baseband also manages asynchronous and

synchronous links, handles packets and does paging and inquiry to access and inquire Bluetooth devices in the area.

The Link Manager Protocol (LMP) uses the links set up by the baseband to establish connections and manage piconets. Responsibilities of the LMP also include authentication and security services, and monitoring of service quality.

The Host Controller Interface (HCI) is the dividing line between software and hardware. The L2CAP and layers above it are currently implemented in software and LMP and lower layers are in the hardware. The HCI is the driver interface for the physical bus that connects these two components. The HCI may not be required and L2CAP can be accessed directly by the application or through certain support protocols provided to ease the burden on application programmers.

The Logical Link Control and Adaptation Layer Protocol (L2CAP) is layered over the Baseband protocol and exists in the data link layer. L2CAP receives application data and adapts it to the Bluetooth format. L2CAP provides connection-oriented and connectionless data services to upper layer protocols with protocol multiplexing capability, segmentation and reassembly operation, and group abstractions (Hopkins and Antony, 2003). L2CAP permits higher level protocols and applications to transmit and receive L2CAP data packets up to 64 kilobytes in length.

The Radio Frequency Communication (RFCOMM) presents a virtual serial port that is designed to make replacement of cable technologies as transparent as possible. RFCOMM is the cable replacement protocol included in the Bluetooth specification. Serial ports are one of the most common types of communications interfaces used with computing and communications devices. Therefore, the RFCOMM enables the replacement of serial port cables with the minimum of modification of existing devices.

The Service Discovery Protocol (SDP) provides a means for applications to discover which services are available and to determine the characteristics of those available services. A specific SDP is needed in the Bluetooth environment, as the set of services that are available changes dynamically, based on the RF proximity of devices in motion. Device information, services, and the characteristics of the services can be queried to enable the establishment of a connection between two or more Bluetooth devices. The SDP defined in the Bluetooth specification is intended to address the unique characteristics of the Bluetooth environment.

Layer	Description
Applications	Bluetooth profiles guide developers on applications that should use the protocol
Telephony Control System (TCS)	Provides telephony services
Service Discovery Protocol (SDP)	Used for service discovery on remote Bluetooth devices
WAP and OBEX	Provides interfaces to higher layer parts of other communication protocols
RFCOMM	Provides an RS-232 like serial interface
L2CAP	Multiplexes data from higher layers and converts between different packet sizes
Host Controller Interface (HCI)	Handles communications between the host and the Bluetooth module
Link Manager Protocol (LMP)	Controls and configures links to other devices
Baseband	Controls physical links, frequency hopping and assembling packets
Radio	Modulates and demodulates data for transmission and reception on air

Table 4.6. Summary of the Bluetooth stack protocol layers (Hopkins and Antony, 2003).

For a local device to use a service on a remote device, the two devices must share a common communications protocol. So that applications can access a wide variety of Bluetooth services, the Java APIs for Bluetooth provide mechanisms which allow connections to any service using RFCOMM, L2CAP OR OBEX as its protocol. The RFCOMM, layered over the L2CAP protocol (see Figure 4.7), is the protocol layer the Serial Port Profile (SPP) uses in order to communicate between Bluetooth devices by providing a stream-based interface.

Bluetooth profiles provide a well-defined set of higher layer procedures and uniform ways of using the lower layer of Bluetooth (Kumar et al., 2003). They are intended to ensure interoperability among Bluetooth-enabled devices and applications. A profile defines the roles and capabilities for specific types of applications. The devices cannot interact unless they conform to a particular profile – having the bare minimum Bluetooth stack is not enough. Two of many profiles used in the middleware application are the Generic Access Profile and the Service Discovery Application and Profile. Firstly, the Generic Access Profile defines the connection procedures, device discovery and link management. Additionally, it defines procedures related to different security models and common format requirements for parameters accessible on the user interface level (Kumar et al., 2003). All Bluetooth devices have to support this profile requirement at a minimum. Secondly, the Service Discovery Application and Profile defines the features and procedures of an application in a Bluetooth device to discover services registered in other Bluetooth devices (Kumar et al., 2003). Additionally, it retrieves information related to the services as well. Table 4.7 presents an overview and description of other Bluetooth foundation profiles.

Profile	Description
Generic Access Profile	The basis for all profiles in the Bluetooth system Defines basic Bluetooth functionality like setting up L2CAP links, handling security modes and discoverable modes
Serial Port Profile	Provides serial port (RS-232) emulation based on the RFCOMM part of the Bluetooth stack
Dial Up Networking Profile	Defines functionality for using a Bluetooth device as a Dial Up Networking gateway
FAX Profile	Defines functionality for using a Bluetooth device as a FAX gateway
Headset Profile	Defines the functionality required to do audio transfer with, e.g. a wireless Bluetooth headset
LAN Access Point Profile	Defines functionality for using a Bluetooth device as a LAN access point
Generic Object Exchange Profile	Provides support for the OBject EXchange (OBEX) protocol over Bluetooth links
Object Push Profile	Defines functionality for exchanging vCard and vCalendar objects, based on the GOEP
File Transfer Profile	Defines functionality for navigating through folders and copying/deleting/creating a file or folder on a Bluetooth device, based on the GOEP
Synchronisation Profile	Defines functionality for synchronising Object Stores containing IrMC objects (vCard, vCalendar, vMessaging and vNotes objects) between Bluetooth devices, based on the GOEP

Table 4.7. Bluetooth foundation profiles (Kumar et al., 2003).

Due to the ad-hoc nature of Bluetooth networks, remote Bluetooth devices will move in and out of range frequently. Bluetooth devices must therefore have the ability to discover nearby Bluetooth devices. When a new Bluetooth device is discovered, a service discovery may be initiated in order to determine which services the device is offering. The Bluetooth Specification refers to the device discovery operation as an inquiry. During this inquiry process, the inquiring Bluetooth device will receive the Bluetooth address and clock from nearby discoverable devices (Knudsen and Li, 2005). The inquiring device then identifies the other devices by their Bluetooth address and is also able to synchronise the frequency hopping with discovered devices by using their Bluetooth address and clock. Overall, the basic concept of any Bluetooth application consists of five components:

- Stack initialisation
- Device management
- Device discovery
- Service discovery
- Communication

The stack is a piece of software that controls the Bluetooth device and, before anything can begin, needs to be initialised. The main purpose of stack initialisation is to get the Bluetooth

device ready to start the wireless communication. The Bluetooth specification leaves the implementation of the stack initialisation up to the device manufacturers, as each handles it differently.

`LocalDevice` and `RemoteDevice` are the two main classes in the Java Bluetooth specification which perform device management. These classes are able to query statistical information about the Bluetooth device (`LocalDevice`) and information on other devices (`RemoteDevice`). The static method `LocalDevice.getLocalDevice()` returns a instantiated `LocalDevice` object to use. The `setDiscoverable()` method is called in the `LocalDevice` object to find other Bluetooth devices in the area. Devices make themselves discoverable by entering the inquiry scan mode. In this mode, frequency hopping will be slower than usual, meaning the device will spend a longer period of time on each channel. This increases the possibility of detecting inquiring devices. Also, discoverable devices make use of an Inquiry Access Code (IAC). Two IACs exist, the General Inquiry Access Code (GIAC) and the Limited Inquiry Access Code (LIAC). The GIAC is used when a device is generally discoverable, meaning it will be discoverable for an undefined period of time. The LIAC is used when a device will be discoverable for only a limited period of time.

Device discovery is the first step required when sensing and discovering nearby Bluetooth devices such as mobile phones, printers, computers and PDAs in the area. Device discovery is the basic *percept* requirement for context discovery. When nearby devices are discovered, the services they offer can be made known. To use any Bluetooth related methods, a reference to the `LocalDevice` object is obtained by calling the `LocalDevice.getLocalDevice()` method. When the `LocalDevice` object is obtained, it provides access to Bluetooth properties for the device such as the Bluetooth address, friendly name and discovery mode. The `LocalDevice` object is used to obtain a `DiscoveryAgent` object. The `DiscoveryAgent` object is used for both device discovery and service discovery. Device discovery begins using the `LocalDevice` and `DiscoveryAgent` objects. The `doDeviceDiscovery()` method is called in the `startApp()` method. Searching with the GIAC (general inquiry access code) parameter will discover devices which are generally discoverable. The `DiscoveryListener` parameter is `this`, meaning the MIDlet.

Code Excerpt

```
public DeviceDiscoverer(BluetoothDiscovererMidlet midlet)
{
this.midlet = midlet;
try
{
LocalDevice localDevice = LocalDevice.getLocalDevice();
discoveryAgent = localDevice.getDiscoveryAgent();

midlet.updateStatus("Searching for Bluetooth devices in the area");
discoveryAgent.startInquiry(DiscoveryAgent.GIAC, this);
}
catch(Exception e)
{
e.printStackTrace();
}
}

public void deviceDiscovered
                 (RemoteDevice remoteDevice, DeviceClass cod) {

try{
remoteDevices.addElement(remoteDevice);
midlet.updateStatus("Device Discovered: " +
                    remoteDevice.getFriendlyName(true));
}

catch(Exception e){
e.printStackTrace();
}
}
```

Figure 4.8. A simple constructor starts the Bluetooth device discovery.

When devices are discovered or the search is complete, events will be delivered to the MIDlet. The `startInquiry()` method returns immediately, returning true if the device discovery is initiated or false if the device discovery process has not started. An event will be delivered to the MIDlet when the device discovery is completed. The `deviceDiscovered()` and `inquiryCompleted()` methods are used to catch events related to the device discovery process. When a device is discovered, the `deviceDiscovered()` method of the object `this` is called (Figure 4.8). The parameter `remoteDevice` is the discovered device. A vector is an appropriate data structure for percept reception and to save discovered devices, as it is not known how many devices will be discovered. The `inquiryCompleted()` method is called when the inquiry ends. Discovered devices are kept in the `DevicesFound` vector by adding them as they are discovered.

Code Excerpt

```
public void inquiryCompleted(int discType)
{
String inqStatus = null;

if (discType == DiscoveryListener.INQUIRY_COMPLETED)
{
inqStatus = "Inquiry completed. " + remoteDevices.size() + "
            Devices found";
}

else if (discType == DiscoveryListener.INQUIRY_TERMINATED)
{
inqStatus = "Inquiry terminated";
}

else if (discType == DiscoveryListener.INQUIRY_ERROR)
{
inqStatus = "Inquiry error";
}

midlet.updateStatus(inqStatus);
midlet.mainForm.deleteAll();
midlet.mainForm.addCommand(midlet.serviceSearchCommand);
midlet.updateStatus("Click on the Search button to search for all
                     the services available");
}
```

Figure 4.9. This method is called when the Bluetooth device discovery process has ended.

After the device discovery is completed, the application begins to search for available services offered by the discovered devices. Service searches can be of a general nature, by polling a device for all available services, but can also be narrowed down to find just a single service. The service discovery process uses the Service Discovery Protocol (SDP). The `servicesDiscovered()` and `serviceSearchCompleted()` methods must be implemented. They handle the events occurring when services are found or the service discovery completes. The `doServiceSearch()` method is added to show how a service discovery is initiated. This method will start a service discovery on the `RemoteDevice` supplied as a parameter, and is called in the `inquiryCompleted()` method. The `searchServices()` method will return immediately, returning a transaction ID for the service discovery. The transaction ID is used to identify the particular service discovery. When the service discovery completes, an event will be delivered to the MIDlet.

Code Excerpt

```
public void servicesDiscovered(int transID, ServiceRecord[]
                                            servRecord)
{
for(int i = 0; i < servRecord.length; i++)
{
DataElement
serviceNameElement=servRecord[i].getAttributeValue(0x0100);
String temp_serviceName = (String)serviceNameElement.getValue();
String serviceName = temp_serviceName.trim();
midlet.updateStatus("-" + serviceName );
blueToothDevice.serviceNames.addElement(serviceName);
}
}
```

Figure 4.10. This method is called when a new service is discovered on a Bluetooth device.

A sequence diagram, illustrated in Figure 4.11, is a good diagram to use to summarise the flow of the context discovery and reception layers. The reason the sequence diagram is so useful is because it visually shows the interaction logic between the objects in the system in the time order that the interactions take place. The diagram conveys this information along the horizontal and vertical dimensions: the vertical dimension shows, top down, the time sequence of messages/calls as they occur, and the horizontal dimension shows, left to right, the object instances that the messages are sent to.

In explaining the sequence diagram, discovering a device starts by creating instance of `DeviceDiscoverer`. This instance obtains `LocalDevice` and `DiscoveryAgent` instance. Searching is started by calling the `startInquiry` method of `discoveryAgent`. When a device is found, an event is sent to the MIDlet and `deviceDiscovered` of `DeviceDiscoverer` is called. `DeviceDiscoverer` adds the found device in a vector. Similarly `inquiryCompleted` is called when searching is over. After finding the devices, services offered by each device are found by `ServiceDiscoverer`. The MIDlet calls for the start of the service discovery. `ServiceDiscoverer` gets `LocalDevice` and `DiscoveryAgent` instances and starts searching by calling `searchServices` of `discoveryAgent`. When a service is found, an event is found and `servicesDiscovered` is called in response of the event. Similarly when search is over `serviceSearchCompleted` is called. It notifies the MIDlet that search is over by calling `serviceSearchComplete` of `BluetoothDiscovererMidlet`.

The next requirement is to manage and store this information in a repository. Section 4.1.3 explains the implementation of the context management layer in the architecture.

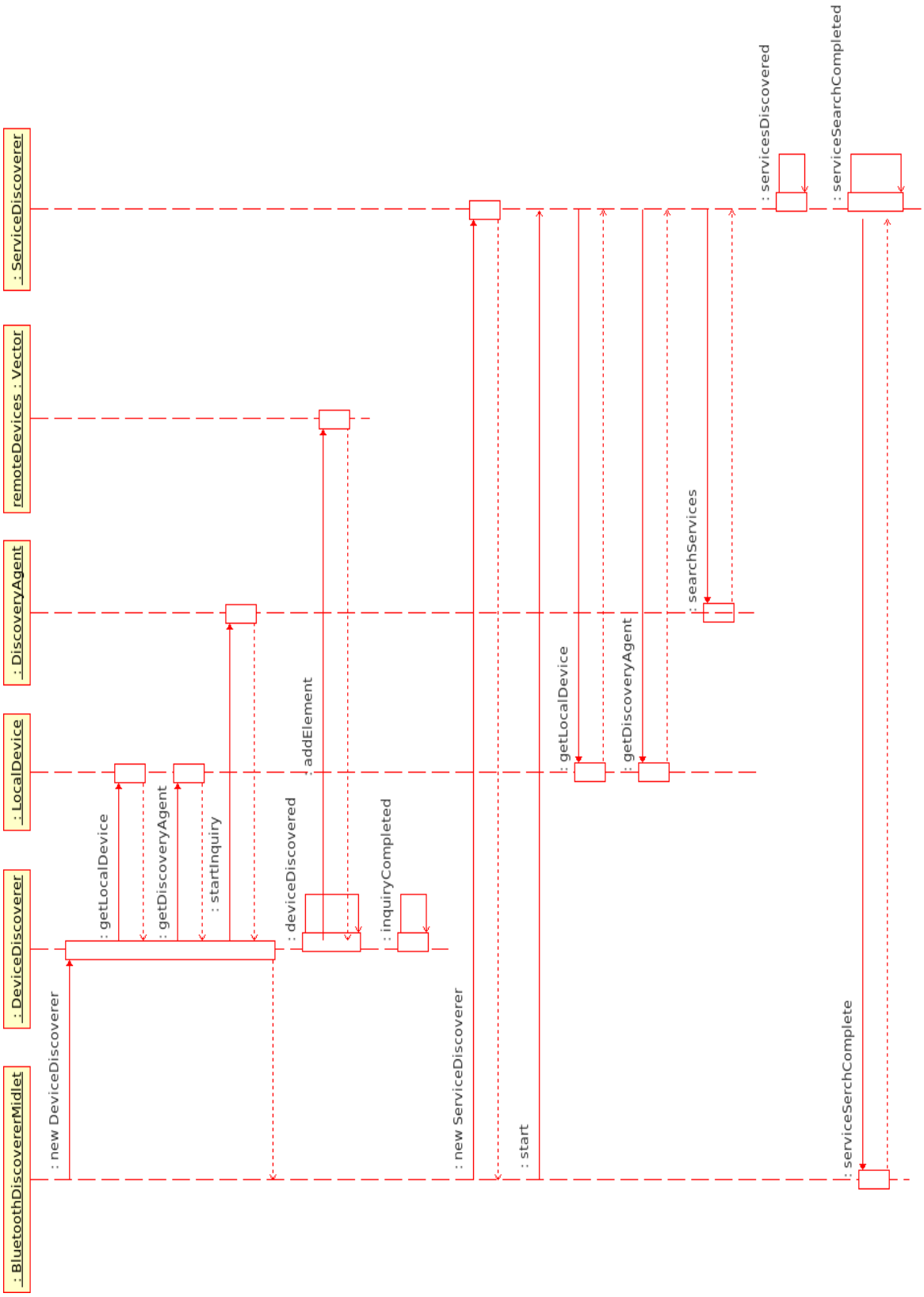

Figure 4.11. This sequence diagram illustrates the discovery and reception phases of the application implementation.

4.1.3 Context Management Layer

An important component of the implementation is the context management layer, which acts as a repository of past and current context data. The functionality of the context management layer is implemented as a Record Management System (RMS) in J2ME. An RMS provides a form of persistent storage for the context discovery and reception layers. An overview of the RMS architecture is presented below and the implementation of the context management layer for the application is explained.

This context-awareness application requires persistence to retain and construct memory. Persistence refers to the retention of information during the operation of the MIDlet and when it is not running. The nature of the information is application dependent but typically stretches the breadth of data storage from application settings to information common to a database. The manner in which persistence is maintained in this J2ME application differs from traditional Java methods of persistence, due to the limited resources available on mobile computing devices. Many mobile computing devices lack disk drives and access to network database servers. Therefore, J2ME applications must store information in non-volatile memory, using the RMS.

RMS is a mechanism to maintain a database in storage memory on small computing devices. RMS is a combination file system and database management system enabling the storage of data in columns and rows similar to the organisation of data in a database table. RMS provides the same functionality to J2ME applications that a JDBC (Java Database Connectivity) provides to conventional Java applications. The RMS can perform similar functionalities of database management software, allowing reading, inserting, deleting, searching and sorting of records. Although RMS provides the functionality of a database, it is not a relational database and therefore does not use SQL to interact with the data.

The RMS stores information in a record store and in the proposed architecture it is referred to as the context repository. In this application the record store is called `btDB`. The record store is a collection of records organised as rows (records) and columns (fields). MIDP provides the set of classes and interfaces in the package `javax.microedition.rms` to work with the `RecordStore`. Some of the important interfaces, classes and exceptions are summarised in Table 4.8 with the prefix `javax.microedition.rms`.

RMS Classes	
`.RecordStore`	A class representing a record store, provides the various methods for addition, modification, deletion for records from `RecordStore`
RMS Interfaces	
`.RecordComparator`	Defines a comparator which compares two records (in an implementation-defined manner) to see if they match or their relative sort order
`.RecordEnumeration`	Represents a bi-directional record store
`.RecordFilter`	Defines a filter which examines a record to see if it matches (based on application-defined criteria)
`.RecordListener`	Receives record changed/added/deleted events from a record store
RMS Exception	
`.RecordStoreException`	Thrown to indicate a general exception has occurred in a record store operation
`.InvalidRecordIDException`	Thrown to indicate an operation could not be completed because the record ID was invalid
`.RecordStoreFullException`	Thrown to indicate an operation could not be completed because the record store storage is full
`.RecordStoreNotFoundException`	Thrown to indicate an operation could not be completed because the record store could not be found
`.RecordStoreNotOpenException`	Thrown to indicate that an operation was attempted on a closed record store

Table 4.8. Classes, interfaces and exceptions for `javax.microedition.rms`.

The `btDB` record store is seen as set of records. A record is an array of bytes, and the RMS automatically assigns to each row a unique integer which identifies the row in the record store that is called the record ID. The record ID is a column on its own within the record store. The record ID is considered the primary key of the record store and serves the same purpose as it would in a traditional database, to uniquely identify each record in the table. The record ID value for the first record created is 1, for the next it is 2 and so on, with each record ID monotonically increasing. It is important to note that the record ID is never reused. As an example, after inserting five records, the fourth record is then deleted from the record store. A new record is then inserted in the record store. The resultant record store does not have any record with record ID 4 any more and the logical state of the record store is the same as that shown in Figure 4.12. Although conceptually a record store is a set of rows and columns, technically there are only two columns. The first column is the record ID and the other column is an array of bytes that contain the persistent data. In this application, a record is defined as an *experience* which includes all of the devices and services discovered for a selected context, and stored in the record store. This definition is provided to address any concerns when these terms – record or experience – are referred to throughout this thesis.

Record ID	Data
1	Data 1……
2	Data 2……
3	Data 3……
5	Data 5……
n	Data n……

Figure 4.12. The storage structure of a record.

The `openRecordStore()`method shown in Figure 4.13 is called to create a new record store and to open the `btDB` record store. This method creates or opens a record store depending on whether the record store already exists within the MIDlet suite. The `openRecordStore()` method requires two parameters. The first parameter is a string containing the name of the record store. The second parameter is a Boolean value indicating whether the record store should be created if the record store does not exist. A true value causes the records store to be created if the record store is not in the MIDlet suite and also opens the `btDB` record store. A false value does not create the record store if it is not located. When a MIDlet has finished with the `btDB` record store, it is closed by calling the `closeRecordStore()` method, which does not require any parameters. Additionally, the method is called to release resources used to maintain an open record store.

Code Excerpt

```
rs = RecordStore.openRecordStore(btDB, true);
```

Figure 4.13. This method opens the `btDB` `record store`.

The `btDB` record store remains in non-volatile memory even after the mobile device is powered down. Non-volatile memory is a scarce resource to ensure that sufficient memory is available when required to store information by a MIDlet. Removing the record store which is no longer being used on the device can help provide sufficient memory resources. The entire `btDB` record store is deleted and reset by calling the static `deleteRecordStore()` method (Figure 4.14). This requires only one parameter and is the string containing the name of the record store (`btDB`) to be removed from the device.

Code Excerpt

```
RecordStore.deleteRecordStore(btDB);
```

Figure 4.14. This method deletes the entire `btDB` `record store`.

After a MIDlet opens the `btDB` record store, records can be written and saved to the record store. Additionally, it can read information already stored in the record store. A record is simply an array of bytes and the `addRecord()` method is used to write a record to the record store. The `addRecord()` method requires three parameters. The first parameter is a byte array containing the byte value of the string being written to the record store. The second is an integer representing the index of the first byte of the byte array written to the record store. The third is the total number of bytes that is written to the record store. The first step in writing a string to a record store is to create an instance of a `String` and assign text to the instance. The string must be converted to a byte array by calling the `getBytes()` method, which returns a byte array. The second parameter of the `addRecord()` method is usually zero and the third parameter is the length of the byte array, indicating the entire byte array to be written to the record store. Figure 4.15 shows an excerpt of the code of the three parameters required in this application.

Code Excerpt

```
rs.addRecord(baos.toByteArray(), 0, baos.size());
```

Figure 4.15. This method is used to write a record to the record store.

Typically information is read from a record store, one record at a time and stored in a byte array. The byte array is then converted to a string, which is then displayed on the screen or processed further, based on the needs of the application. The record is saved in the RMS in the format: `context|deviceCount|deviceName1|serviceCountA|service1|service2| ... | deviceName2|serviceCountB|service1|service2|` and so on. Figure 4.16 shows an excerpt of code where this process occurs in the application.

Code Excerpt

```
{
int recordId = getInt("RecordCount");
String saveString =
vContextSave.elementAt(saveList.getSelectedIndex())+ "|" +
                 mDevices.size();

for(int j=0;j<mDevices.size();j++)
        {
        BlueToothDevice currentDevice =
        (BlueToothDevice)mDevices.elementAt(j);
        saveString += "|" + currentDevice.deviceName + "|" +
        currentDevice.serviceNames.size();

        for(int i=0; i<currentDevice.serviceNames.size() ;i++)
        {
        saveString += "|" +

        (String)currentDevice.serviceNames.elementAt(i);
        }
        }

saveString(saveString, ""+recordId);
recordId++;
saveInt(recordId, "RecordCount");
showAlert("Notification", "Devices are saved into Database",
          deviceList, AlertType.INFO);
}
```

Figure 4.16. Saving the record in the RMS after the context discovery layer is completed.

In order to read all the records from the record store, the MIDlet requires the total number of records in a record store. The `numRecords()` method of the `RecordStore` class returns an integer representing the total number of records in the record store. This value is then used as the maximum value for a `for` loop used to transverse through each record in the record store. Calling the `getRecord()` method of the `RecordStore` class cycles each iteration of the `for` loop. The `getRecord()` method returns bytes from the `RecordStore` which are stored in a byte array that is created. The `getRecord()` method requires three parameters. The first parameter is the record ID, the second is the byte array created for storing the record and the third parameter is the integer representing the position in the record from which to begin copying into the byte array.

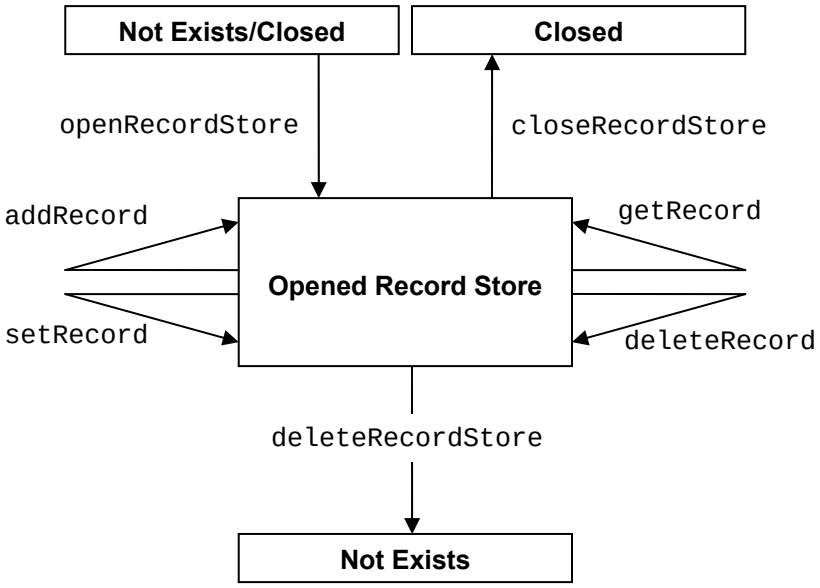

Figure 4.17. The different record store states in our BT_RMS_CM application.

A `RecordEnumeration` provides a way to traverse data elements and the enumeration object manages how data is retrieved from a record store. Changes made by the user are reflected when the record store's content is iterated. A record enumeration is obtained by calling the `enumerateRecords()` method, which requires three parameters. The first is the record filter used to exclude records returned from the record store. The second is a reference to the record comparator, which is a method used to compare records returned from the record store. The last parameter is a Boolean value indicating whether or not the enumeration is automatically updated when changes are made to the underlying record store. The `enumerateRecords()` method returns a `RecordEnumeration`, as shown in Figure 4.18. There is no filter or comparator method and the record enumeration is not automatically updated when a change is made to the record store. The `RecordEnumeration` method is used to interact with records in the record enumeration.

Code Excerpt
`RecordEnumeration re = rs.enumerateRecords(null, null, false);`

Figure 4.18. This method is used to transverse data elements in the record store.

Browsing through each record is a common interaction that occurs with record enumeration. The `hasNextElement()` method is called to evaluate whether or not there is another record in the RecordEnumeration. If this condition is satisfied, records can transverse forwards and backwards within the `RecordEnumeration`. The `nextRecord()` method is called to move to the next record, or the `previousRecord()` method is called to move back one record. Both methods

return a byte array containing a copy of the record. When the `RecordEnumeration` is created, you are positioned at the top of the record enumeration. However, the top is not the first record; the `nextRecord()` method needs to be called to move to the first record. Moving to the last record is called by the `previousRecord()` method while at the top of the record enumeration. The `hasNextElement()` method is called to determine whether there is a next record, as shown in Figure 4.19. Calling the `hasPreviousElement()` method determines whether there is a previous record. Both methods return a Boolean value indicating whether or not there is another record. Tracking progress through the `RecordEnumeration` is achieved by retrieving the record IDs of records in the enumeration. As an example, if there are ten records in the enumeration, the ID of the first records is assigned zero and the ID of the last record is assigned nine. The record ID of the next record is determined by calling the `nextRecordId()` method. The `nextRecordId()` method returns an integer representing the ID of the next record (Figure 4.19). Similarly, the `previousRecordId()` method is called to retrieve the ID of the previous record.

Code Excerpt

```
if (re.hasNextElement())
{
int iRecordID = re.nextRecordId();
rs.setRecord(iRecordID, baos.toByteArray(), 0, baos.size());
}
```

Figure 4.19. The methods used for browsing through each record of our application.

The overall requirements of the RMS are to store and browse all discovered devices in their selected contexts. The RMS aids in the construction of contexts based on the previous experiences stored. Section 4.1.4 details the implementation of the constructive memory query layer in the architecture and how the RMS is used to help in this process.

4.1.4 Constructive Memory Query Layer

The most important part of implementing the constructive memory query layer is to learn new experiences as a result of interaction with the application. The functionality of this layer is to select the most likely context for the devices currently discovered. Additionally, it is used in future memory-matching computations in determining a context by recalling experiences from the RMS. A component of the constructive memory process is the memory-matching algorithm and this is only part of the overall process. It is based on the dot product, also known as the scalar product, from the field of mathematics. The dot product is an operation that takes two vectors and calculates the angle between the vectors. The smaller the angle is, the closer the relationship is

between the two. As an example of the notation, given $a = [a_1, a_2 \ldots a_n]$ and $b = [b_1, b_2 \ldots b_n]$, the dot product by definition is:

$$a \cdot b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \ldots + a_n b_n$$

Equation 1. The dot product of two vectors where Σ denotes the summation notation.

This can be written very succinctly using the Einstein summation notation where the convention that repeats indices is implicitly summed over; $A \cdot B = a_i b_i$. In the Euclidean distance space, there is a relationship between the dot product and the lengths and angles. For a vector a, $a \cdot a$ is the square of its length, and more generally b is another vector $a \cdot b = \|a\|\|b\|\cos\theta$ where $\|a\|$ and $\|b\|$ denote the length (magnitude) of a and b, θ is the angle between them. Therefore, given two vectors, the angle between them can be found by rearranging the above formula using:

$$\cos\theta = \left(\frac{a \cdot b}{\|a\|\|b\|} \right)$$

Equation 2. Finding the angle between two given vectors.

Considering the definition of the dot product, consider a in the notation to represent the current percepts or devices being discovered and b in the notation to represent previous percepts or devices already discovered and stored as a record or experience. Using this notation, the vector is a record and computes the dot product between the current discovered percepts and previous percepts, assigning a point value on the match. The maximum value will be the most likely predicted context. The method by which this is implemented in J2ME is explained below.

Once the current devices are discovered, the matching algorithm searches the `btDB` record store and recalls the records (experiences) of where it has discovered the devices before. The constructive memory-matching algorithm begins by counting the combination of devices, using these counts to produce a metric, using the metric to find the dissimilarity between the current context and every item in memory, and retrieve the best matches. The process commences by first initialising the match count to zero and then passing through each record in the record store. A point is assigned – 1 for a match or 0 for a non-match – for each device match in our current discovery with devices stored in records previously. All services offered by previous devices discovered are ignored in the matching algorithm. The focus and concentration are on matching

the devices in previous records with the current devices discovered and predicting the most likely context based on what it has sensed before.

The `divider` function computes the total number of devices in the current discovery that match devices which have been previously stored as records (experiences) in the context repository. This number is then divided by the total number of devices in that specific previous record that match devices in the current discovery. As an example, record 1 has a total of six devices previously stored as context A, record 2 has a total of eight devices previously stored as context B, and record 3 has a total of ten devices previously stored as context C. In the current discovery, a total of eight devices are discovered and a context needs to be determined. Using the memory-matching algorithm, the total number of devices matching with previously stored records are the following:

- record 1 matches three of six devices stored;
- record 2 matches six of eight devices stored; and
- record 3 matches six of ten devices stored.

The constructive memory-matching algorithm computes the `divider` function as a percentage for the current devices discovered in the record that match the previously saved devices in their respective records. Referring back to the example, the following percentages are computed for each record to predict a context for the current device discovery:

- record 1 has a 50% prediction on the context being A (three of six devices match);
- record 2 has a 75% prediction on the context being B (six of eight devices match); and
- record 3 has a 60% prediction on the context being C (six of ten devices match).

Although the current device discovery matches the same total of devices in both records 2 and 3, there is a rule built into the algorithm that will take the maximum or highest percentage match as the most likely predicted context. Therefore using this rule, record 2 has a higher percentage than record 3 and context B is selected as the most likely context for the devices currently discovered. This current discovery is saved and added as a new record in the RMS and will be used again in future memory-matching computations in determining a context. Figure 4.20 shows an excerpt of code demonstrating the complete process of the constructive memory-matching algorithm implemented in J2ME. It has commenting (bold type) throughout the code to explain each specific component of the algorithm.

Code Excerpt

```
public List getSaveList()
{
//Holds the maximum match for each context
int []matchCount = new int[vContextSave.size()];
//Checks against multiple devices with same friendly name
boolean []flag = new boolean [mDevices.size()];
for(int i=0;i<vContextSave.size();i++)
{
//Initialised to zero
matchCount[i] = 0;
}
//Reads number of records
int count = getInt("RecordCount");
for(int i=0;i<count;i++)
{
//Reads a record
String []sArray = split(geString("" + i), "|");
int deviceCount = Integer.parseInt(sArray[1]);
int index = 2;

int currentMatch = 0;
int divider = getMax(deviceCount, mDevices.size());
for(int j=0;j<mDevices.size();j++)
{
flag[j] = false;
}
//Walks through a record
for(; deviceCount != 0; deviceCount--)
{
String deviceName = sArray[index++];
//Skips all the services per device
int serviceCount = Integer.parseInt(sArray[index++]);
index += serviceCount;

for(int j=0;j<mDevices.size();j++)
{
BlueToothDevice currentDevice =
        BlueToothDevice)mDevices.elementAt(j);
if(currentDevice.deviceName.equalsIgnoreCase(deviceName)
        && flag[j] == false)
{
//Will force a device with same friendly name to match twice
currentMatch++;
flag[j] = true;
break;
}
}
}
//Converting to a percentage
currentMatch = (currentMatch*100)/divider;
for(int j=0;j<vContextSave.size();j++)
{
String context = (String)vContextSave.elementAt(j);
```

Code Excerpt (Continued)

```
//Matching the saved context
if(context.equalsIgnoreCase(sArray[0]))
{
//Storing the maximum match
if(matchCount[j] < currentMatch)
{
matchCount[j] = currentMatch;
}
break;
}
}
}
//Building the saved list
for(int i=0;i<vContextSave.size();i++)
{
String context = (String)vContextSave.elementAt(i);
saveList.set(i, context + " (" + matchCount[i] +"%)", null);
}
return saveList;
}
```

Figure 4.20. The memory-matching process in this application.

All source code written for this application is presented in Appendix A. Please see Appendix B for instructions to download all of the source code and run the application on a mobile device. A total of four individual java files make up this application:

- BluetoothDiscovererMidlet.java – executes the overall functional flow of this application;
- DeviceDiscoverer.java – performs the Bluetooth device discovery;
- ServiceDiscoverer.java – performs the Bluetooth service discovery; and
- BlueToothDevice.java – holds all the information of a device together in this application.

4.2 Hardware Components

The hardware components provide the physical means by which the prototype implementation can be put into practice. The first component of the prototype is the mobile device itself. The mobile device provides the means on which to demonstrate this application in a sensate environment. An important consideration in selecting an appropriate mobile device is that it does support a J2ME run-time environment. Unfortunately, this is not very specific and there is a lot of variance between mobile devices. Additionally, an important part of J2ME development for mobile devices concerns insuring the target device selected supports the profile and APIs used in this application. Otherwise this application simply will not work. Communities of hardware developers typically propose extensions to the capabilities beyond the basic CLDC and MIDP. These extensions are contained in Java Specifications Requests (JSR). Specifically in this application, the JSR-82 enables J2ME applications to access Bluetooth networking services on the mobile

device. The mobile device is required to support this JSR and Bluetooth radio connectivity. An overview of J2ME and its configurations, as well as Bluetooth support are discussed in detail in Section 4.1. An internal memory component on the mobile device or support for expandable memory – Memory stick, Compact Flash or SD cards – is a hardware requirement to support the RMS mechanism of the application.

The Nokia 6131 (Figure 4.21, left) and the Nokia N80 (Figure 4.21, right) are used for deploying this application and are used in the experiments conducted in Chapter 5. These devices are selected as they supported the requirements of the J2ME infrastructure, as well as the Bluetooth and RMS APIs outlined in the Section 4.1. Additionally, the devices have Bluetooth radio connectivity and minimum 12 MB internal memory capacity, with the option of using expandable memory to support the application as the context repository increases in size.

Figure 4.21. The Nokia 6131 (left) and the Nokia N80 (right).

The second hardware component is best described as a series of minor components that make up the overall prototype implementation. These components are Bluetooth-enabled devices that can be discovered by the application. More specifically, these devices are made discoverable in

the context discovery and reception layers of the architecture. Between six to eight devices at a time are made discoverable when the application is running. These Bluetooth-enabled devices include printers, mobile phones, PDAs, laptops, USB dongles, mobile phone stereo headphones and Bluetooth LAN access points. The purpose of these minor hardware components is to aid in the implementation and demonstrate the functionality of this application.

4.3 Synthesis of Components and Application Interaction

The overall implementation merges the back-end and hardware components together to provide a prototype application. The mobile device displays the graphical user interface of back-end components where application interaction takes place. It should be noted that the mapping of the graphical user interface differs between all mobile device manufacturers and models. The user is able to navigate the application interface through the mobile device to discover and sense Bluetooth devices that form the percepts in the modified architecture. Moreover, the user will be able to construct and assign a context from memory, based on the devices currently discovered.

The prototype application is a proof-of-concept and designed to demonstrate how constructive memory can be used in a context-aware application. The prototype application is called BT RMS CM; Bluetooth (BT), Record Management System (RMS), Constructive Memory (CM). The application is based on students in a university context. As stated in Section 1.1, this thesis defines context as *place* – a unit of context which users may wish to be notified about. The range of places are based on a university building where there are various spaces and rooms, as well as the devices a student might discover in a university setting. This has all been reflected in the overall application development and implementation process.

Displayed below are visual walkthroughs of each graphical user interface implemented and the application flow and interaction of the prototype. Additionally, other application functionalities are illustrated and extend on the implementation of back-end components.

4.3.1 Discovering Devices in the BT RMS CM Application

This visual walkthrough is a demonstration of discovering Bluetooth devices and services in the application. Each graphical user interface presents a phase the application goes through in discovering and capturing devices and services for a current experience. In each figure below an image specifically details and explains what is currently occurring in each phase of the *Discover BT Devices* interface. Additionally, it visually illustrates some of the programming code written for that particular interface which has been described and explained extensively in Section 4.1.

Figure 4.22. The user selects *Discover BT Devices* from the *BT Main Menu* to discover Bluetooth devices and sensors in the area.

Figure 4.23. The application begins searching for Bluetooth devices in the area, calling the `DeviceDiscoverer` method.

Figure 4.24. The user selects the *Search* button to begin searching for all available Bluetooth services in the area called by the `ServiceDiscoverer` method. In this application, the service discovery results are not directly used in helping the constructive memory process. However, they may be used in future applications to predict a context more accurately based on the services offered by discovered devices.

Figure 4.25. After the application finds a new service on a Bluetooth device, the `servicesDiscovered` method is called to search for services on the discovered device.

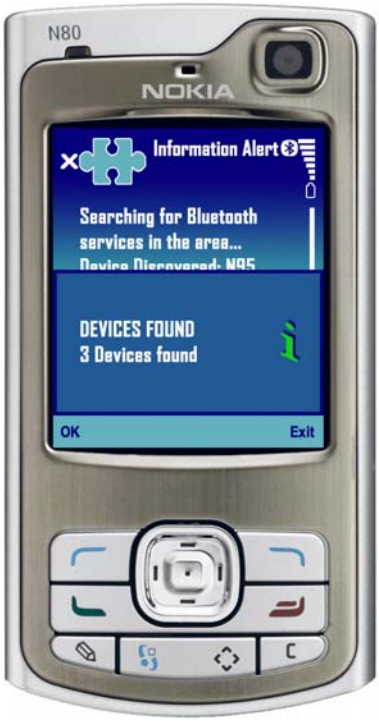

Figure 4.26. When the Bluetooth device and service discovery processes have ended, the `inquiryCompleted` method is called. An *Information Alert* is displayed, showing the total number of *Devices Found*. The user selects *OK* to accept this notification.

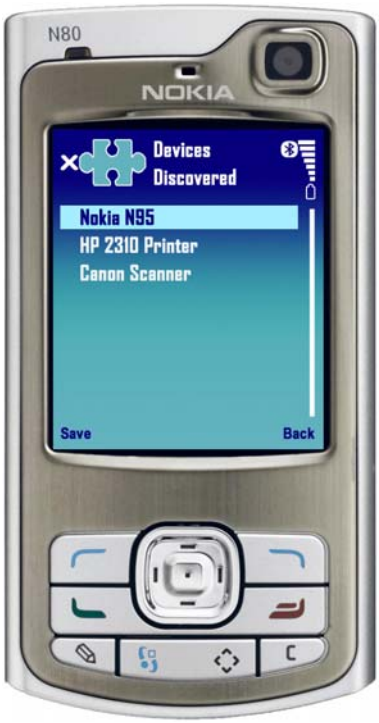

Figure 4.27. The *Devices Discovered* interface displays all of the devices found. The user will need to select *Save*, to save the devices into the RMS.

Figure 4.28. A context will need to be selected and assigned to save the devices discovered. The *Save Context* interface demonstrates the constructive memory component, expressing the devices that match in a context as a percentage from a previously saved experience in the RMS.

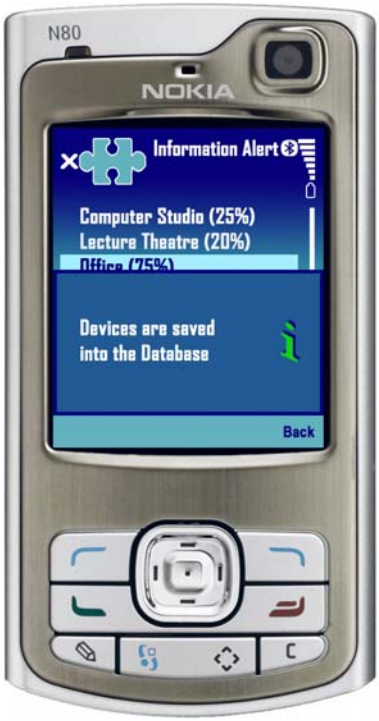

Figure 4.29. When the user selects a context based on the highest percentage match, *Devices are saved into the Database* under the selected context. The user selects *Back* to accept the *Information Alert*, exit the application and return to the *BT Main Menu*.

4.3.2 Browsing the BT RMS CM Application

The browse interface is useful for viewing the total number of devices discovered and the context it is assigned from previous application experiences. Specifically, a device can be selected from the RMS to display the services that particular device offers in more detail. This visual walkthrough is a demonstration of browsing the BT RMS CM application of previously saved context experiences. Each graphical user interface presents a phase the application goes through in browsing the RMS. In each figure below an image specifically details and explains what is currently occurring in each phase of the *Browse BT Database* interface.

Figure 4.30. The user selects *Browse BT Database* from the *BT Main Menu* to browse all context experiences saved previously in the BT RMS CM application.

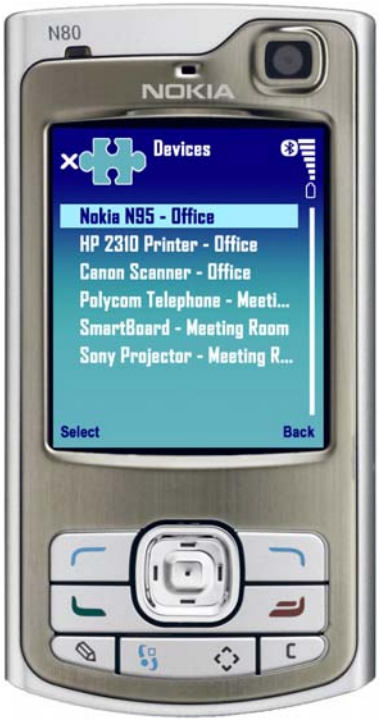

Figure 4.31. The device-friendly names and the context assigned are displayed in the *Devices* interface. The use can choose *Select* to display the individual details of the device or choose *Back* to return to the *BT Main Menu*.

Figure 4.32. When the user selects the device, a *Device Details* interface is displayed. The interface displays the device-friendly name, the context assigned and the services the device offers. The user can choose to *Delete* this device if they wish from this context. Additionally, this will also delete the device from the RMS.

Figure 4.33. If the user does wish to delete a device from the context, a confirmation interface is displayed. The user can select *OK* to proceed with the deletion or *Back* to cancel and return to the *Devices* interface to browse more devices and contexts.

Figure 4.34. After the user proceeds with deleting the device, an *Information Alert* is displayed notifying the user the *Device is Deleted*. The user can decide to *Dismiss* the notification and return to the *Devices* interface to browse more devices and contexts.

4.3.3 Searching the BT RMS CM Application

The search interface is useful for understanding in what contexts a specific device may appear over many experiences. Alternatively, it is useful to discover what devices may appear in a specific context. This visual walkthrough demonstrates searching the BT RMS CM application of previously saved experiences. Each graphical user interface presents a phase the application goes through to search the RMS. Overall, the search features will be useful for further analysis beyond the scope of this thesis, in determining devices for a specific context or a context for specific devices. The search results are displayed in a similar interface format that is illustrated in Figure 4.31, depending on the search criteria. In each figure below an image specifically details and explains what is currently occurring in each phase of the *Search BT Database* interface.

Figure 4.35. The user can select *Search BT Database* from the *BT Main Menu* to search for device-friendly names and/or a specific context from the BT RMS CM application.

Figure 4.36. The user has the option to enter the device-friendly name for which they are searching manually, using their mobile device's keypad and/or selecting to search a specific context.

Figure 4.37. When the user selects to search by context, a new interface is shown displaying all the contexts. The user can scroll to select an appropriate *Context*, in which they would like the search to perform and select *OK* to confirm their selection.

Figure 4.38. Once the user is satisfied with their search criteria, the user must select *Search* to perform the search and display the search results. Alternatively, the user can select *Back* which will cancel the search and return to the *BT Main Menu* interface.

4.3.4 Resetting the BT RMS CM Application

Resetting the BT RMS CM application is only useful if the RMS is corrupted or errors occur in the application and it is not functioning correctly. Errors may include that the device discovery is not working or the constructive memory-matching algorithm is not computing correctly. This visual walkthrough is a demonstration of deleting and resetting the BT RMS CM application. Each graphical user interface presents a phase the application goes through to reset and delete the RMS from the application. Although this is an important feature to have in this application, it should only be used cautiously, as the entire database of previously stored experiences will be deleted. In each figure below an image specifically details and explains what is currently occurring in each step of the *Reset BT Database* interface.

Figure 4.39. The user can delete the entire RMS by selecting *Reset BT Database* from the *BT Main Menu*.

Figure 4.40. If the user does wish to delete all records, a confirmation interface is displayed in the event the user, by accident, selected the *Reset BT Database.* The user can select *OK* to proceed with the deletion or select *Cancel* to return to the *BT Main Menu.*

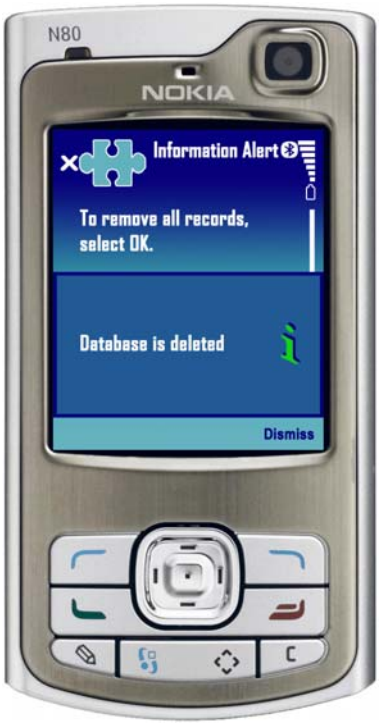

Figure 4.41. After the user decides to proceed with deleting the RMS, an *Information Alert* is displayed, notifying the user the *Database is deleted*. The user can decide to *Dismiss* the notification and return to the *BT Main Menu* interface.

4.3.5 Adding a New Context to the BT RMS CM Application

The add interface is useful when the application enters a context it has never encountered before and the user is aware of this situation. Alternatively, based on the constructive memory-matching algorithm computed for the current discovery, the maximum percentage in all contexts is below a certain level. As an example, for the devices currently discovered, the memory-matching algorithm computes a maximum percentage of 37% for a specific context. Depending on the interpretation by the user and the environment they may be in, they will not feel confident with this result and possibly assign a new context for this current experience. This visual walkthrough is a demonstration of adding a new context to the BT RMS CM application. Each graphical user interface presents a phase the application goes through to add a new context to the RMS. Each figure below an image specifically details and explains what is currently occurring in each phase of the *Add Context* interface.

Figure 4.42. The user can add a new context in the BT RMS CM application by selecting *Add Context* from the *BT Main Menu*.

Figure 4.43. The user will have to use their mobile device's keypad to manually enter a new context. The context field is restricted to 50 characters, including spacing between the text.

Figure 4.44. After a new context is entered in the context field, the user will need to select *OK* to save the new context into the BT RMS CM application.

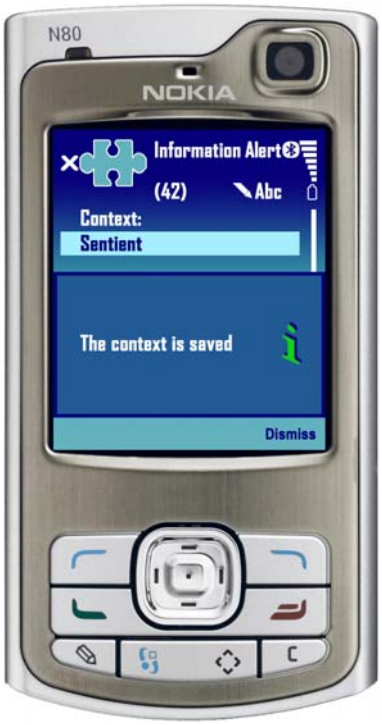

Figure 4.45. If the user proceeds to save the new context, an *Information Alert* is displayed that *The context is saved*. The user can decide to *Dismiss* the notification and return to the *BT Main Menu* interface.