Master's Thesis Project

Josef Israelsson
di98jin@cs.umu.se

February 2005
20 credits

An integration of Java APIs for Bluetooth with Teleca's software suite Obigo

UMEÅ UNIVERSITY
DEPARTMENT OF COMPUTING SCIENCE
SE-901 87 UMEÅ
SWEDEN

Abstract

The Bluetooth technology for wireless communication is used worldwide in wearable devices today. The Java programming language and its platform for wearable devices is also a well used and still improving technology when talking about the wearable market. JSR-82 is a Java application programming interface for the Bluetooth wireless technology. Teleca Software Solution AB in Lund wants an implementation of JSR-82 integrated with their software framework Obigo Q-Line. Obigo is a world leading application system found in hundreds of millions of mobile phones worldwide. Besides this JSR-82 integration, Teleca also want a deeper investigation about three already existing related integrations, and in the end a concrete Teleca JSR-82 solution analysed, designed and presented.

The purpose with this thesis project was to make deeper studies in those technologies and about the Obigo Q-Line architecture. In the end a JSR-82 library should be integrated and a JSR-82 solution should be presented.

The result is an integration of Profix optional API JSR-82 implementation with Obigo, and an extensive JSR-82 and related technology investigation that resulted in three different JSR-82 solutions revealed and presented.

Preface

This Masters Thesis is the result of the JSR-82 project work by Josef Israelsson at Teleca Software Solutions in Lund. The work was carried out between the 6th of September 2004 and the 21st of February 2005, with a break during October 2004.

I would like to thank Jan Olof Svensson my supervisor at Teleca for many useful ideas and guidance along the road. I would also thank Pär Spjuth at Teleca for his great technical expertise during this project. Thanks to my internal supervisor Thomas Nilsson for helpful comments and suggestions regarding this report. Finally I would like to thank the department manager Knut Mårtensson at Teleca who brought me the opportunity to perform this Master's Thesis project at Teleca.

Josef Israelsson

Lund, February 2005

Contents

List of figures

List of tables

1 Notes

1.1 Focus change

The main task with this Master's Thesis project was at first to integrate a JSR-82 implementation with Obigo Q-Line. But the consequences that followed large internal delivery delays were that the focus on the thesis project was changed. When the delays occurred no one knew how long I had to wait before I could get the necessary source code for integrating the JSR-82 implementation with Obigo. Therefore the focus was changed into answering the main questions stated in the "Goal" sub-chapter, and to present a complete JSR-82 solution that could possibly be a future Teleca product.

After a while the required source code was finally delivered, and an investigation about if there was time enough left to integrate the JSR-82 implementation was started. The outcome from that investigation was that the JSR-82 implementation was going to be integrated, but that there was no time for integrating a Bluetooth stack. What limitations this brought is considered later in the report.

1.2 Internal information

Because of the usage of internal confidential information during this Master's Thesis project, some of the company names in this document are fictive and does not exist in reality.

Profix, PEmb, Textone and Btone in this document are all fictive names.

2 Introduction

2.1 Background

Java is one of the most used programming languages for creating software today. With its possibilities and independence of the underlying platform Java is used worldwide by both professional developers as well as the lay public for almost any kind of application development.

JSR-82 (Java Specification Request) is the optional package including the APIs (Application Programming Interface) for programming and controlling Bluetooth devices with the Java programming language. The APIs are used in the J2ME (Java 2 Platform Micro Edition) architecture, and have many advantages compared to native languages like C++. To make use of the classes in the JSR-82 package, and the Bluetooth functionality, Java applications called MIDlets are implemented. MIDlets are placed on top of the J2ME stack on a wearable device, and makes calls to the different Bluetooth methods in the JSR-82 library. When running a Bluetooth MIDlet the KVM (Kilo virtual Machine) in the CLDC (Connected Limited Device Configuration) acts as a virtual computer and executes the byte code output from the compiler. The Bluetooth stack then responds to this execution and communicates further with the Bluetooth hardware.

2.2 Thesis description

What is the cost and what are the requirements for developing a JSR-82 solution within the Teleca Company? What has to be done to integrate a JSR-82 implementation with Obigo Q-Line? Which are the possible JSR-82 solutions? What does the already existing integrations look like, and how can they be used in Teleca's JSR-82 development? These are some of the questions this thesis will answer.

2.3 Goal

There are two principal goals with this master thesis project. One is to answer the two main questions:

- What different JSR-82 solutions is it possible to develop within the Teleca Company?

- How much will it cost in terms of licences, compatibility tests, time consumed, and manpower for Teleca to deliver those JSR-82 software solutions to an end customer?

The other goal with this Master's Thesis project is to integrate a JSR-82 implementation with Obigo Q-Line.

2.4 Scope

There are a lot of different technologies, systems, and integrations involved in this thesis project, so one of the first tasks was to define what should be considered and what should be left outside this project. Despite these restrictions, there are a lot of investigations that had to be done. There was a need to be familiar with the J2ME platform and how to develop MIDlets, the Bluetooth technology had to be investigated, the already existing integrations i.e. the Java / Obigo integration, the Bluetooth / Obigo integration, and the JSR-82 / Bluetooth integration had to be investigated and understood. There was also a must to be familiar with Obigo Q-Line; how to implement functionality and how to integrate external components. At last there was of course important to gain knowledge and perform an investigation about the JSR-82 API.

In this project a JSR-82 implementation is integrated with Obigo, and the investigations of the subjects above are gathered in a couple of internal Teleca reports. The outcome of those investigations is presented in the internal Teleca document "Integration solutions".

2.5 Method

At first the JSR-82 API and the J2ME platform had to be investigated. Then the existing integrations had to be analysed and decided in which way those could be used henceforth. Then the design of the Teleca JSR-82 solutions had to be made and settled, where after those solutions could be elucidated and the circumstances for developing those had to be investigated. After that the final solutions could be presented. After or in parallel with this, the actual JSR-82 integration could be completed.

2.6 Thesis Outline

Chapter 5 is an investigation that elucidates the J2ME technology, MIDlets and the JSR-82 API. Chapter 6 is an extensive investigation of the Bluetooth / JSR-82 integration, Bluetooth / Obigo integration, and the Java / Obigo integration. This analysis was important to the following integration design in chapter 7 that is a proposal of how the

actual JSR-82 integration with Obigo is going to look like. Chapter 8 elucidates the obscurities that can affect such JSR-82 integration by answering the main questions of this Master's Thesis project. Chapter 9 presents the result from this project, the Teleca JSR-82 solutions. Chapter 10 is about the JSR-82 integration with Obigo, followed by chapter 11 that presents what future work that is possible or even necessary. Chapter 12 concludes the Master's Thesis.

3 Background

3.1 Java 2 Micro Edition

The J2SE (Java 2 Standard Edition) platform, provides the Java components for the stationary market. But on the Java 2 conference in June1999, Sun released a new platform which they call J2ME, Java 2 Micro Edition.

The J2ME platform itself is not any piece of software; in fact it is a set of rules and statements about how the Java programming environment should be configured to work on wearable devices with limited resources. It is the primary platform for the wearable market and consists of a JVM (Java Virtual Machine) specification and an API specification [2].

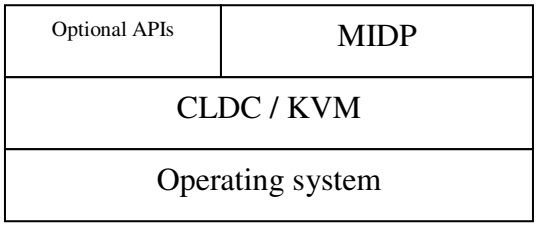

Figure 1: J2ME platform

The J2ME platform is based on configurations, profiles, and optional packages. A configuration combined with a profile and possibly some optional packages in a wearable device is called a J2ME stack. The configuration is the lowest level and specifies the minimal APIs that must be implemented to run Java on a low memory device. In figure 1 the CLDC/KVM represents the configuration layer. The CLDC also specifies a stripped version of the JVM, named KVM [3]. Compared to the JVM the KVM only provides reduced functionality, but still enough to be implemented and used in small devices. The C programming language is the most used language for implementing a KVM. The KVM acts as the lowest layer in the J2ME platform and provides abstraction over the native environment. This will simplify the porting of an application to another environment; only the KVM needs to be re-implemented because the applications uses the same abstract interface independent of the KVM implementation.

Besides the KVM the CLDC configuration also specifies a set of Java packages and classes; the minimal number that is required having a Java application execute on a wearable device. The CLDC does not specify any GUI (Graphical User Interface) or another user friendly mechanism. It is also required that the device has some kind of

underlying operative system that handles the execution of the KVM. This means that the CLDC configuration supplies with the substructure for Java applications, but not enough for satisfy the demands of most Java programmers but only what they expect always is included in a J2ME stack. Therefore profiles are used. Profiles are the layer above the configuration layer, and define a collection of APIs that are available in a certain set of devices, for examples mobile phones. A J2ME profile is a set of Java classes that extends the functionality of a J2ME configuration [2]. An application that implements a certain profile can be used on any device that supports that profile. The most used profile in mobile phones is the MIDP (Mobile Information Device Profile) which is the one that is going to be described in this document. MIDP consists of a collection of Java APIs that makes it possible to download and run Java applications on a mobile device, for example user interface and networking APIs. The CLDC/MIDP combination is a very common assembly because of the ease and developing- friendly APIs. Applications that are developed for the CLDC/MIDP combination are called MIDlets.

The optional APIs extends the functionality even further; they define classes for devices with special demands on the functionality, for example database connection or wireless communication with the Bluetooth technology.

3.2 MIDlets

As told before, MIDlets are Java applications that are written to be put above the CLDC configuration and next to/ above the MIDP profile, and can make use of APIs from both of those specifications. A MIDlet itself is an instance of the javax.microedition.midlet.MIDlet class that is defined in the MIDP profile. When running a MIDlet on a wearable device, you must first download and install it on for example a cellular phone. The class files of a created and compiled MIDlet is packaged in a Java archive file, which one you have to download and install on the device. There is also a descriptor file that comes with the developing of a MIDlet. The descriptor file describes the content of the MIDlet, so that you do not have to download the whole archive file just to examine the functionality of the current MIDlet. Once the MIDlet is downloaded and installed, it has to go through a set of states; the life cycle of a MIDlet.

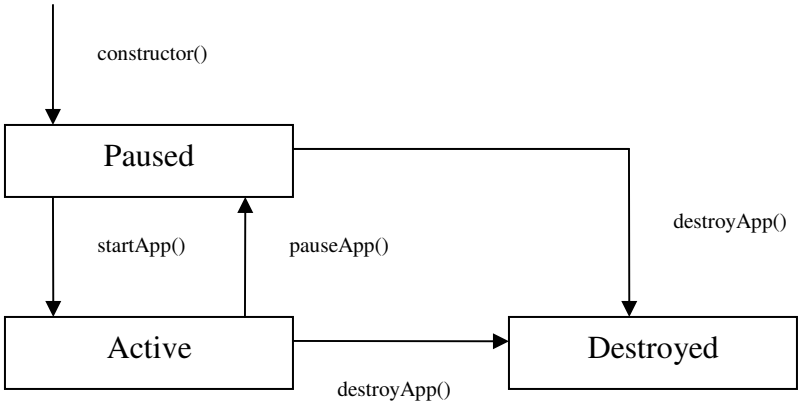

Figure 2: MIDlet life cycle

At first, when the MIDlet is about to run, an instance is created and the application is paused: it is in the Paused state. The instance is created by calling the constructor of the application. The next state is the Active state. The MIDlet will reach this state when the startApp() method is called. StartApp() is automatically called by the application manager during the execution of the MIDlet [3].

From here on the MIDlet can reach two more states: Paused and Destroyed. When the MIDlet does not have any current tasks to execute it may be paused and the pauseApp() method is called. When the MIDlet has finished executing, or if the MIDlet itself has any reasons to terminate, the destroyApp() or notifyDestroyed() methods are called. This will terminate the run of the Java application on the wearable device. When the MIDlet is in the Destroyed state it has no longer any memory references, and therefore waits for the Garbage Collector to remove the instance and free some memory [3]. This is illustrated in figure 2.

This is how the structure of a typical MIDlet looks:

```
Import javax.microedition.midlet.MIDlet;
        Public class Application extends MIDlet {
        Public Application() { }
        Public void startApp() { }
        Public void pauseApp() { }
        Public void destroyApp(Boolean b) { }
        }
```

This was a brief explanation of how a MIDlet works. As told before, a MIDlet makes use of APIs from the CLDC configuration and the MIDP profile. But sometimes the purpose with the MIDlet is more than those APIs can handle, and therefore it exists something called optional APIs. Those APIs are classes that add even more functionality to the Java stack, more specialized functionality to applications that have very specific requirements. Examples on such requirements would be database connectivity, wireless messaging and multimedia. All Java packages have names that start with JSR and are requests in the Java Community Process. What interests us here, is the JSR-82 package; the optional package for wireless developing with the Bluetooth technology.

3.3 JSR-82

3.3.1 Introduction

JSR-82 is the formal name of the APIs for the Bluetooth Wireless Technology (JABWT). The JSR-82 specification defines the architecture of a J2ME Bluetooth system as well as all the classes and methods that must be implemented having the system JSR-82 compatible. The JSR-82 API is based on the CLDC configuration and operates on top of the CLDC layer in the J2ME platform, but in fact it is used to extend the functionality of the profile layer. The most usual J2ME profile is the MIDP profile when developing for the cell phones market, which adds functionality such as multimedia and end-to-end security to the J2ME system [1].

The JSR-82 API has become a fast growing standard for Bluetooth applications on mobile phones, and the goal is to define a standard set of APIs that will enable an open, third party application development environment for Bluetooth wireless technology on devices with limited resources.

3.3.2 Architecture

The general architecture of the JSR-82 API is based on three functional categories: Discovery, Communication and Device Management [16]. Their relation is illustrated in figure 3.

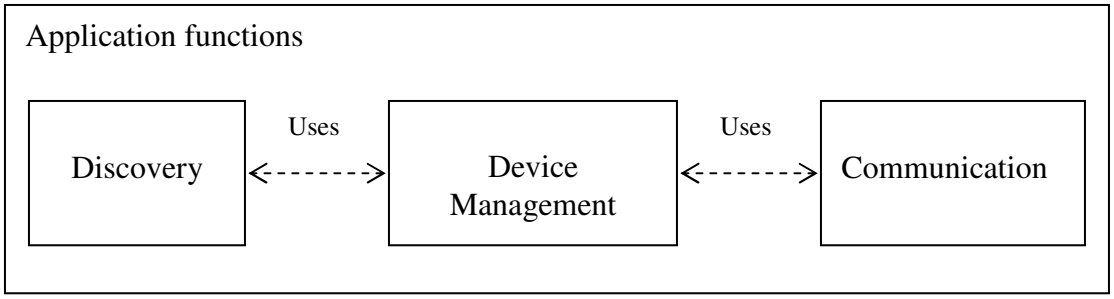

Figure 3: General JSR-82 architecture

Discovery is the process of search for and discovers other Bluetooth devices. This includes APIs for device discovery, service discovery and service registration. The Communication category embraces the actual Bluetooth communication between applications on wearable devices; establish and using the Bluetooth technology with the JSR-82 API. Device Management is about managing this functionality of the Discovery and Communication categories.

JSR-82 has support for the most common used Bluetooth profiles. By having support for only the Bluetooth base profiles, the JSR-82 could quickly be introduced at the market, and because that the higher-level profiles in the Bluetooth profile architecture are based on the lower ones and extends their functionality, developers also have the possibility to extend the API and implement the higher-level profiles [1]. Below is a list of the supported Bluetooth profiles and protocols:

Protocols:

- Logical Link Controller and Adaptation Protocol (L2CAP)
- RFCOMM
- Service Discovery Protocol (SDP)
- Object Exchange Protocol (OBEX)

Profiles:

- Generic Access Profile (GAP)
- Service Discovery Application Profile (SDAP)

- Serial Port Profile (SPP)
- Generic Object Exchange Profile (GOEP)

The three functional categories contain the classes and interfaces for implementing the Bluetooth functionality. A complete list of the JSR-82 methods can be found as an Appendix. The JSR-82 classes and interfaces are listed below:

- javax.bluetooth.DiscoveryAgent

  The methods in this class are implemented for handling the service and device discovery.

- javax.bluetooth.DataElement

  The Bluetooth service attribute can take on different data types. This class contains those data types, for example String and Boolean.

- javax.bluetooth.UUID

  UUID is a unique identifier for the service attribute. This class represents those encapsulated integers.

- javax.bluetooth.DiscoveryListener

  A MIDlet needs to have listeners that respond to device and service discovery events. This interface makes this possible.

- javax.bluetooth.ServiceRecord

  This interface enables Bluetooth services being described to Bluetooth clients.

- javax.bluetooth.LocalDevice

  This class enables the user collecting information about the local Bluetooth device.

- javax.bluetooth.ServiceRegistration

  This class is used to register the internal services so that other devices can locate them.

- javax.bluetooth.L2CAPConnection

The methods for sending and receiving data through a L2CAP connection are defined in this interface. Also the methods for obtaining the MTUs are defined.

- javax.bluetooth.L2CAPConnectionNotifier

  This class contains only one method. This is used by servers to listen for connections from L2CAP clients.

- javax.bluetooth.DeviceClass

  The values for the device type and the types of services on a device are defined in this class.

The Discovery category is split into three sub-sections of classes and interfaces (device and service discovery and service registration), which all maps to the SDAP profile and uses the SDP protocol. These are the DiscoveryListener interface, the DiscoveryAgent class, the UUID class, the DataElement class, and the ServiceRecord interface. The JSR-82 specification supports the following SDAP functionality [16]:

1. Searching for services of a particular class

2. Retrieving service attributes of a service

3. Simultaneously searching for services and retrieving their attributes and terminating a service search transaction in progress.

The Device Management category contains classes and interfaces that are used for configuring the local Bluetooth device, security and how it responds to other Bluetooth devices. Those classes maps to and are part of the Bluetooth GAP profile. Also the methods for requesting secure Bluetooth communication are located in this category. This category contains the LocalDevice class and the RemoteDevice class.

The Communication category provides classes for connections to services that have RFCOMM, L2CAP, or OBEX as the highest-level protocols. This last category maps to the SPP profile, its methods lets devices offer SPP based services or initiate SPP based requests. L2CAPConnection and L2CAPConnectionNotifier are interfaces of this category.

There are two independent packages defined within the JSR-82 API: the javax.bluetooth package and the javax.obex package. Both are depending of the javax.microedition.io package, but can exist independent of each other on top of a CLDC configuration [16].

The MIDP profile can be extended with the JSR-82 package. The JSR-82 package is still not depending of the MIDP layer, but the JSR-82 / MIDP combination should doubtless be the most usable way of using the JSR-82 API [1]. Figure 4 states an example of a J2ME architecture where the JSR-82 API coexists with the MIDP profile on top of a CLDC layer.

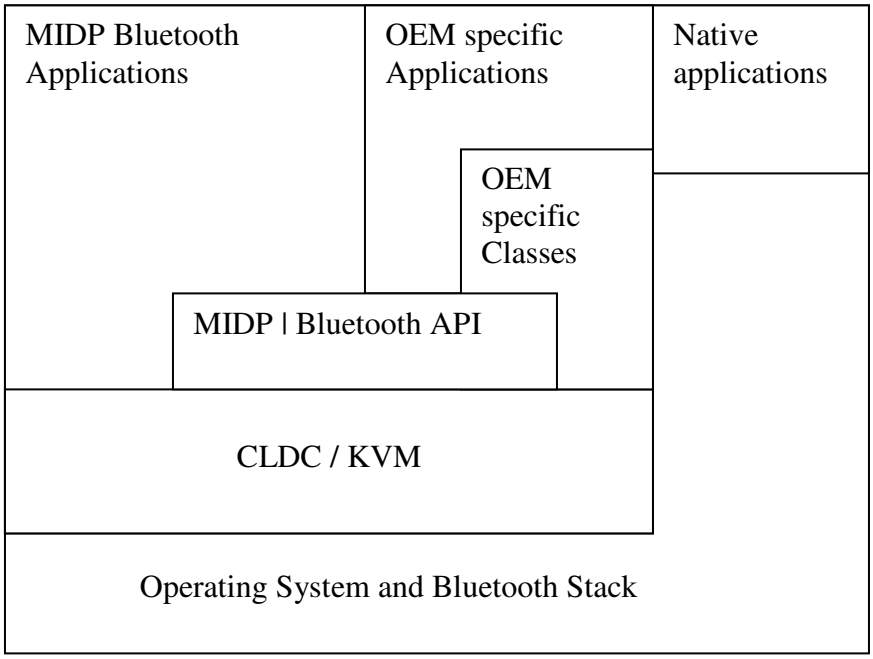

Figure 4: J2ME architecture

The MIDP Bluetooth applications (MIDlets) can either be aimed at the MIDP / JSR-82 layer or directly towards the CLDC configuration.

The BCC (Bluetooth Control Centre) is a piece of software required by the JSR-82 API to exist in a Java Bluetooth system [1]. The JSR-82 does not specify how the BCC should be implemented; the implementation of the BCC is a vendor issue. It may be implemented as a set of Java classes or as a native application on the Bluetooth host. The BCC is the central authority for local Bluetooth device settings and defines security settings for your Bluetooth device, i.e. it is an integral part of the security architecture. The BCC requirements stated in the JSR-82 specification are the following:

- Include base security settings of the device.
- Provide a list of Bluetooth devices that are already known, no matter if the devices are in range or not.
- Provide a list of Bluetooth devices that are already trusted, no matter if the devices are in range or not.
- Provide a mechanism to pair two devices trying to connect for the first time.
- Provide a mechanism to provide for authorization of connection requests.
- Information contained in the BCC must not be modified or altered other than by the BCC itself.

3.3.3 Benefits

So why use Java and the JSR-82 API to make use of the Bluetooth technology? It is a well known issue that Java does not execute as fast as other languages like C or C++, so why bother? The main reason for using Java generally speaking is its low cost for development and its independence of underlying platform [4]. The Java programming language is easy to learn, and the applications created with Java are easy to make accessible to people who wants to make use of the software. Further on there are also two another advantages with Java that makes it a good option when programming Bluetooth applications. In the first place, the JSR-82 API is independent of the stack and the radio [1]. You do not need to have any knowledge about the underlying Bluetooth stack or the radio. As a programmer you can focus on the Java programming, not about other challenging tasks like integration with the stack and such problems; that is a JSR-82 API Vendor issue. Follow the Java and JSR-82 standard and your application will work with basically any hardware and OS that has a CLDC/MIDP J2ME architecture integrated.

The other great advantage by using Java to develop Bluetooth applications is that the JSR-82 is the only standardized Bluetooth API that exists [1]. When using a C or C++ based Bluetooth SDK (Software Development Kit), it is completely a SDK vendor issue naming the functions and classes and which profiles that should be supported. To be able to write JSR-82 applications, the Bluetooth stack must be JSR-82 compatible. This means that the HCI, L2CAP, SDP and RFCOMM layers must be included. The required profiles are GAP, SDAP, SPP and GOEP. If the Bluetooth stack fulfils those requirements, a Java MIDlet that uses the JSR-82 API should be able to make use of this Bluetooth stack.

3.3.4 Functionality

In a Bluetooth application, there are some basic components that you should be aware of and which the application must be able handle. Those are:

- Stack initialization
- Device management
- Device discovery
- Service discovery
- Service registration
- Communication

As described above there is also something called the Bluetooth Control Centre that must exist in a JSR-82 compliant environment according to the JSR-82 specification. The Bluetooth Control Centre is very important because sometimes the initializing of the Bluetooth stack must go through it, why it supports with different security settings for the device. Because the control centre is vendor specific, the package name will be something like com.vendor.bluetooth.bcc, if it is not written in a native language.

The first thing that has to be done is to initialize the Bluetooth stack. Sometimes this initialization is done automatically, but sometimes it requires a small amount of code. The com port name and the baud rate may have to be set.

The next thing to examine is the device management. This means that you should manage your own device; query some statically information about your own Bluetooth device. javax.bluetooth.LocalDevice is a class that gives you all the information about the own device that is needed. But a part of the device management is also about getting some brief information about other Bluetooth devises in the area. javax.bluetooth.RemoteDevice gives the opportunity to access a single remote Bluetooth device in the area and receive its twelve character address or its friendly name.

The device discovery part is about getting deeper information and knowledge about the devices in the area. A client in a client-server based model should for example use the device discovery to discover the server to expose its services. The two classes needed to search for other devices are the javax.bluetooth.DiscoveryAgent and javax.bluetooth.DiscoveryListener classes. For example, to start an inquiry in a Bluetooth environment, the startInquiry(…) method in the DiscoveryAgent must be used. To be

aware of when other devices are found, the DiscoveryListener is used. When the DiscoveryAgent find another devise, the listener is called by the KVM whereupon the deviceDiscovered() method is called.

When other Bluetooth devices are discovered, the next step is to investigate which services they provide. This is about the same procedure as when searching for other devices. The task is now to search for the services in the already located devices, which is done by using the classes DiscoveryAgent, DiscoveryListener, ServiceRecord, DataElement and UUID. The javax.bluetooth.DiscoveryAgent has for example the methods searchServices() and selectService() that are used to search and select any service in another device.

When you initialize your Bluetooth device, you make it ready to access or to be accessed. To have other devices make use of your own services, you must register those so that other devices can find them.

Then it is time to communicate with other devices, as what the Bluetooth technology generally is about. RFCOMM connections are stream oriented and works as a virtual serial port protocol. This type of connections should be used in situations when you would replace a serial cable. The connection sequence can for example start with creating a URL with a UUID on the server side, and then open a connection with this URL. This thread should be blocked until a client makes a response to the connection, and then the data streams could be opened. When the streams are open, the data can be sent and received.

The L2CAP connections are in opposite to RFCOMM connections packet oriented. It is about the same procedure as with stream oriented connections, but here the maximum transmission unit must be set. Otherwise one device may send larger packets than the receiver can handle which will cause problems.

4 Integration investigation

Obigo Q-Line is Teleca's world leading software for mobile devices. Obigo consists of a framework, a browser, a messenger, a content manager, an imager, Digital Rights, security, and a studio. Today there exist an integration of Java with Obigo, and an integration of the Bluetooth stack with Obigo. What is missing is a version of Obigo where both of those technologies are integrated and connected to each other as described in the Java APIs for Bluetooth wireless technology specification (JSR-82). Teleca wants a third party JSR-82 implementation and an arbitrary KVM to be integrated with Obigo

Q-Line. This will make it possible to download and run Bluetooth applications written in Java to a wearable device where the Obigo Q-Line is embedded. For example multiplayer MIDlet games over a Bluetooth network, or a MIDlet that lets the user print documents to a Bluetooth enabled printer server.

This chapter is the result of an investigation of the already existing integrations Java/Obigo, Bluetooth/Obigo, and also the software company Textone's integration of the JSR-82 API with the Bluetooth stack. There is shed light upon the different integrations, and in the end conclusions are given.

The main purpose with this integration investigation is to elucidate the obscurities with the different integration that exists, how they are done, their similarities, and things that couple them together. This investigation should also be very helpful in the next phase of the project where the design of the JSR-82/Obigo integration is made.

4.1 Java / Obigo

4.1.1 Obigo Q-Line

Obigo Q-line is Teleca's software product for mobile platforms. It is a framework that is used on top of existing operative systems of native targets to introduce an environment with pre-defined and usable functionality, such as browser, messaging, content handling etc. The Framework consists generally of two main parts; the Mobile Suite Framework (MSF), and the modules that mostly independently of each other handle the specific tasks that the Obigo Q-Line supports (figure 5). The principal task of the MSF is to provide a common framework for Obigo applications and services to execute within. An advantageous feature with this framework is that it hides its platform dependence by disposing a higher level application support for developers that uses the MSF. The MSF consist not only of API specifications for those different task modules, but also actual implementation of the Obigo functionality.

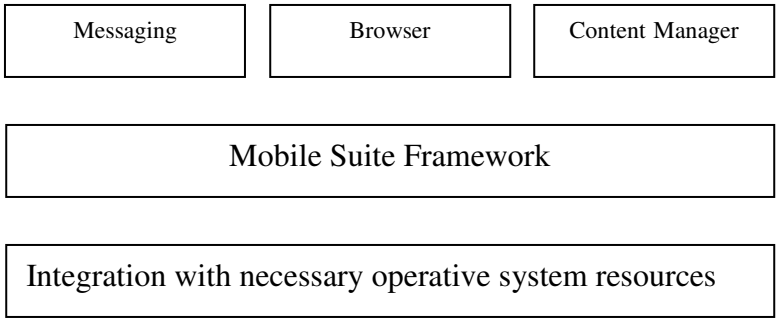

Figure 5: The Obigo Modules and MSF

When integrating the Obigo framework on a native platform, some important interfaces must be implemented to make communication between the module and the operative system possible, and also to provide communication between different modules. Those interfaces are those who abstract the platform dependence to the developer of Obigo modules. There are two kinds of interfaces; Service APIs and Integration APIs. Service APIs are interfaces that are used by different modules to communicate with each other. Integration APIs are the interfaces that are used when entities in the host device wants to communicate with the modules. Those interfaces are either an adaptor or a connector. Adaptors are used when the software module wants to communicate with the surrounding entities in the operative system. Connectors are used in the opposite direction.

When implementing an Obigo module, certain steps must be considered. At first the functional scope of the Module must be set to be able to determine what kind of interface that will suite. Then the module will be implemented, adapted to the MSF, followed by the implementation of the interfaces and the packaging functions. A user manual for the module must also be proposed and written.

One of the products in Obigo, and the one which is most relevant for this investigation, is the Content Manager Service (CMS) module. This service handles the downloading of MIDlets, and is providing an interface for communicating with a KVM. Through this interface, the CMS informs the KVM about installed MIDlets, and about starting and stopping those. The CMS handles the J2ME functionally Teleca today delivers to their customers.

4.1.2 PEmb overview

PEmb is Profix Java platform for embedded devices. PEmb Micro is a variant adapted for the handheld market and for devices with limited resources; it is a J2ME stack intended for integration with existing operative systems in handheld devices.

PEmb is because of the portability with Java easy to integrate with a native platform and OS, and enables use of the Java technology in almost every kind of wearable device. Profix product is a Java Runtime Environment supporting the CLDC configuration, including their implementation of a KVM. It also supports the MIDP profile for downloading and running Java MIDlets. Further optional packages are also supported, as for example the JSR-82 API for Bluetooth wireless communication.

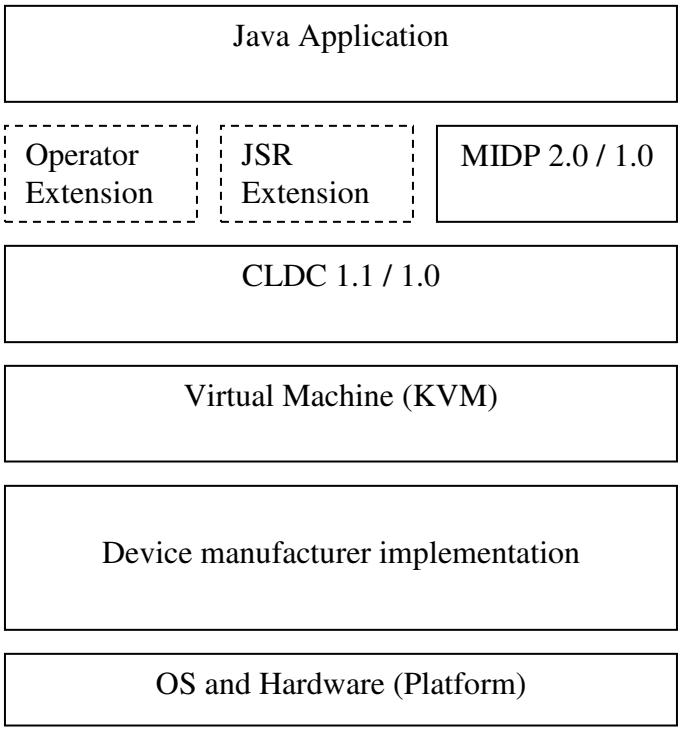

Figure 6: PEmb Architecture

In figure 6 all layers between the Java application and the Device manufacturer implementation is PEmb layers. Those altogether are called the PEmb library, and are provided as precompiled, re-linkable libraries. The device manufacturer implementation is the consumers own implemented interfaces adopted for their own platform, for integration with the PEmb specifications. The Java application on top of the PEmb libraries is a MIDlet that contains the source code that uses the underlying layers.

4.1.3 The integration

4.1.3.1 Overview

The integration of Profix PEmb with the Obigo Q-Line is an attempt to have Teleca's software suite download and execute Java Applications. Because of the expanding use of the Java technology on the handheld market, a KVM is now more of a requirement than a feature in almost all mobile phones on the market today.

The general PEmb software configuration for integration is illustrated in figure 7. The Java application is a downloaded MIDlet that is going to be executed inside the KVM.

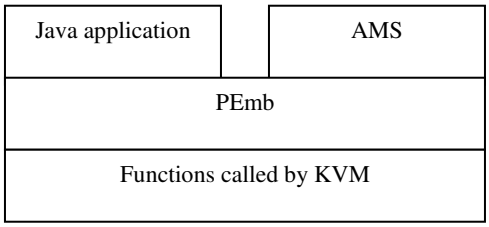

Figure 7: PEmb integration software configuration

The AMS is the Application Management Software, software that manages Java applications, for example the downloading of MIDlets and installing and deletion of those. Also execution management, i.e. the processing of starting a MIDlet is handled by the AMS. The KVM inside the PEmb stack is a Java Virtual Machine with reduced functionality. The KVM main task is to load the class files and execute the byte code they contain. The task can be divided into three general parts:

- Starting and running a MIDlet using a JAD and/or JAR file
- Stopping a MIDlet using interface call or user intervention
- Extracting a file from ZIP

The interpreter inside the KVM interprets the byte code into machine code that can be understood by the operative system.

The purpose of the integration of Profix KVM with Obigo Q-Line was to pass all Sun's TCK tests for MIDP 2.0 on a Windows platform. The Obigo modules that are built and

tested with the integration are CMA, CMS, CPS, DRS, PHS, SES and STK. This means that the integration may not work properly if other modules are added.

4.1.3.2 PEmb Thread

Since the PEmb library has to run in a separate thread in a native environment the Hybrid Wrapper Module has been created. The PEmb library has to access Obigo services in an asynchronous manner, and therefore the calls from the PEmb are transformed into asynchronous calls by suspending the PEmb library while the request is processed. The result from Obigo is then retuned to the PEmb when the thread is resumed, which make this synchronous – asynchronous communication including the message passing style in Obigo work properly. Hence the PEmb and the KVM will share the same memory context, which also is a requirement for using the PEmb library in any environment. The Wrapper Module is a re-implementation and replacement of the earlier KIS implementation (Figure 8). KIS is a KVM interface service that worked as a loop-back for KVM calls in Obigo.

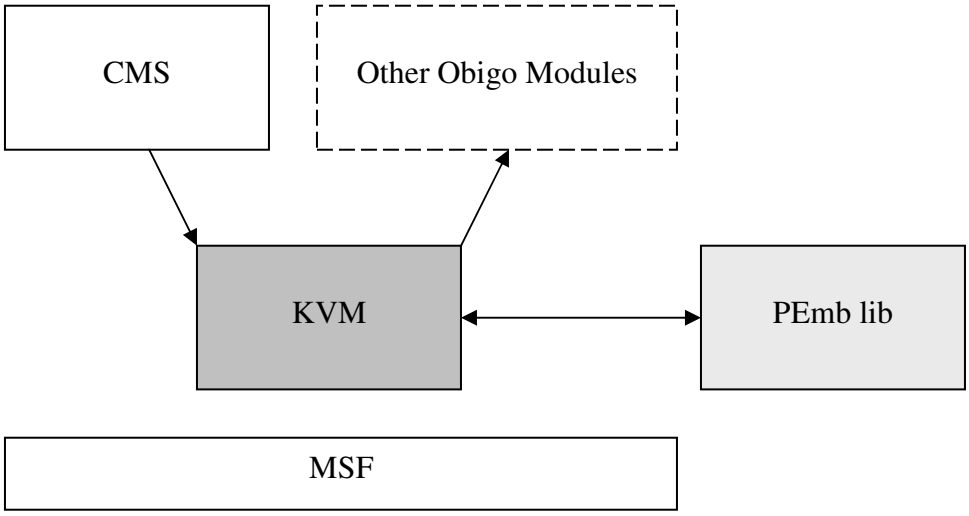

Figure 8: Environment modules communicating with KVM

4.1.3.3 PEmb layers

The layers in the PEmb part of the integration consist of the PEmb library, PSI interfaces, PSC components, and some lower layer interfaces and components. This is illustrated in figure 9. Since the PEmb library is delivered as pre-compiled source code, a binary library, there must exist some interfaces that are wrapping the target platform. Those

interfaces are called PEmb Service Interface (PSI), and are defined at a very high level. To implement those interfaces, components called PEmb Service Component are created. Those are implementations of the definitions in the PSI interfaces, and provide the actual functionality between the PEmb library and the target platform. There also consists interfaces at a lower level but those are not particularly interesting for this integration because main parts are done with PSI interfaces.

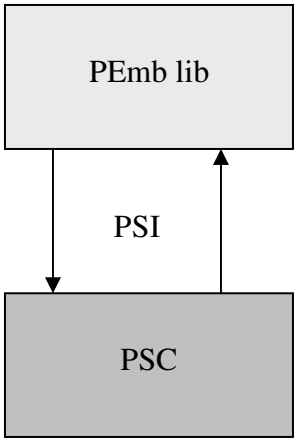

Figure 9: Layers of the PEmb integration model

4.1.3.4 PEmb integration

The design model of the complete integration consists of many parts and looks rather complex (Figure 10). The main request was to integrate the PEmb library on top of Obigo, but because of the limited requirements, some parts was not integrated with Obigo but instead directly to Windows. This was completed by using the Profix Windows RI (Reference Implementation) code as a lower layer interface. The affected part was the MIDP 2.0 Media Library, which fell out of the frames for the estimated time.

The Obigo PSC (PEmb Service Component) in the picture below is the Obigo implementation of the PSI. It contains a module all dedicated to Obigo, to handle asynchronous and long lived operations that needs to be performed by the PSI. The Profix PSC is an implementation that holds the PEmb Graphic library (JBGL), and is implemented towards the MSF as a lower layer interface. Because the MSF does not provide all features stated in MIDP 2.0, some new functions had to be defined towards the KVM. Those functions are called KVM HDI, and are handling the execution of this MIDP functionality.

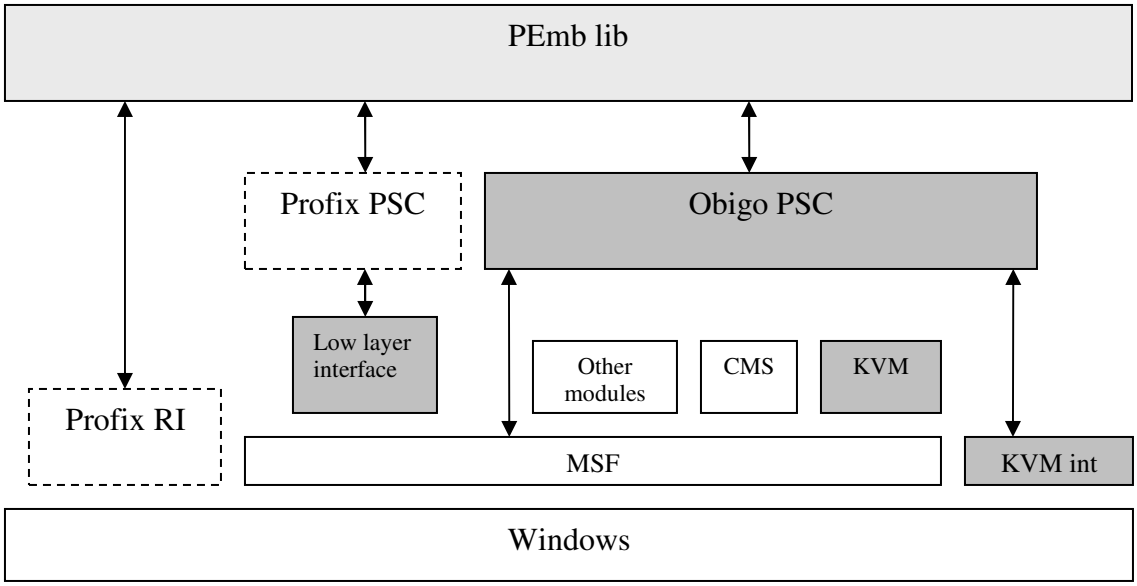

Figure 10: Integration layers, PEmb and Obigo

4.1.3.5 Incomplete Porting

Because the integration is running on a windows based platform, all porting was not done to Obigo but directly towards Windows. Some of the porting was also omitted because of its complexity and the time it have required to integrate fully. The file handling is now running in a synchronous manner, where some work is required to make it be asynchronous. The Media integration uses the PEmb Windows Media RI instead of Obigo PSC. Also the graphics, the permissions, the unzipping, the certificates, and the DRM, are not fully Obigo integrated, and requires a lot of work to be completed.

4.2 Bluetooth / Obigo

4.2.1 Bluetooth

Bluetooth is a technology that simplifies the wireless communication between protocols just as any other communication protocol; it tells how to connect two or more Bluetooth enabled devices and have them communicate with each other. The purpose with Bluetooth is not being the technique with the highest data transfer speed, not either the most wide ranged type of wireless communication. Bluetooth uses the radio spectrum for communication, which means that there is no need for line of sight between the devices that wants to connect to each other. The Bluetooth represents simplicity and user-friendly usage of wearable devices that want to communicate.

The Bluetooth protocol stack is the controlling part and the one implementing the Bluetooth protocol (Figure 11). The protocol stack lets you control your own device, and communicate with other devices. The stack is divided into layers:

- **HCI**: The Host Controller Interface is a software layer that handles the data transfer to and from the Bluetooth device. If a Bluetooth device for example is connected to the computer with a USB cable, then someone must understand the USB data signals. This is the task of the HCI layer.

- **L2CAP**: The Logical Link and Adaptation Protocol process all traffic except the audio signals that comes from the HCI layer. The functions of the L2CAP layer are packet segmentation and re-assembling of data. It also deals with multiplexing if it receives much data at the same time that is addressed to different layers at a higher level.

- **SDP**: The Service Discover Protocol does exactly what the name says; it discovers services that other Bluetooth devices provide.

- **RFCOMM**: This layer emulates a RS232 connection, which means it works as a cable replacement protocol. A wearable device would for example synchronize data with a computer thorough the RFCOMM layer just as if it were transferred with a serial cable.

- **TCS-BIN**: The Telephony Control Protocol Specification handles all signals that have to do with telephony, answer calls, hang up calls etc.

- **WAP**: This is a protocol that has been adopted to fit Bluetooth's needs with internet accessing wearable devices.

- **OBEX**: This is an object exchange protocol and is also adapted to Bluetooth. As the name implies it is used when transferring files between Bluetooth devices.

- **BNEP**: The Bluetooth Network Encapsulation Protocol allows other network protocols to be transmitted over Bluetooth. BNEP encapsulates TCP/IP packets in L2CAP packets before sending them over the network.

- **HID**: The Human Interface Device Protocol is adopted to control human interfaces like keyboards and video game controllers. HID lists the rules and guidelines for transmitting data to and from those.

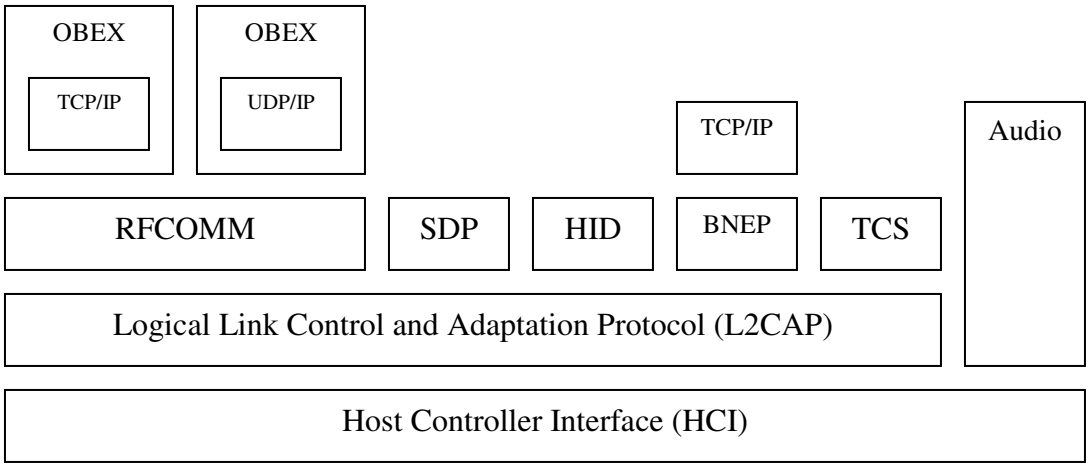

Figure 11: The Bluetooth protocol stack

Besides the protocol stack, the other main part in Bluetooth is the profiles. The Bluetooth profiles define different sets of functionality for Bluetooth devices. If one Bluetooth device for example has a certain kind of functionality, what determines if that device can use its functionality together with another phone? It is the profiles that decide such things. Cordless Bluetooth headphones must of course support the headset profile, but also the Bluetooth device on the computer that the headphones are connected to must support this profile. Therefore, to have two Bluetooth devices communicating with each other, they must both have the protocol stack and implement the required profiles for the action.

The profiles are all in some way connected and dependent on each other (Figure 12). The General Access Profile (GAP) is the base profile that all the other profiles are dependent upon. The profiles are constructed in a hierarchy where the higher ones depend on the lower ones; the functionality of a higher profile needs the functionality of the lower one. The most used profiles are:

- **General Access Profile**: This is the base profile, and the one that controls the basic connection establishment. All the other profiles are dependent upon this one.

- **Service Discovery Application Profile**: A profile that issues exactly what the name implies: it has direct contact with the Service Discovery Protocol and search for services on other Bluetooth devices in the area.

- **Serial Port Profile**: Communicates with the RFCOMM protocol, and creates a virtual serial connection to another Bluetooth device.

- **Generic Object Exchange Profile**: A profile that controls the use of the OBEX protocol in the Bluetooth stack.

- **Human Interface Device Profile**: A set of functionality defined for use with the HID protocol that sets the guidelines for human interfaces like mice or keyboards.

- **Personal Area Networking Profile**: A profile that supports wireless networking in Ad-Hoc form. Requires the BNEP protocol in the Bluetooth stack.

- **FAX Profile**: Just as the name implies: makes it possible to send fax wireless between for example a computer and a fax machine.

- **LAN Profile**: Same functionality as defined in the PANP, but this profile makes LAN networking possible.

- **File Transfer Profile**: A profile that supports file transfer from one Bluetooth device to another.

- **Object Push Profile**: Defines functionality for pushing and pulling a limited type of files, for example vCards.

This is a general description of the Bluetooth technology, and what is needed to understand the integration of Ericsson's Bluetooth stack with Obigo.

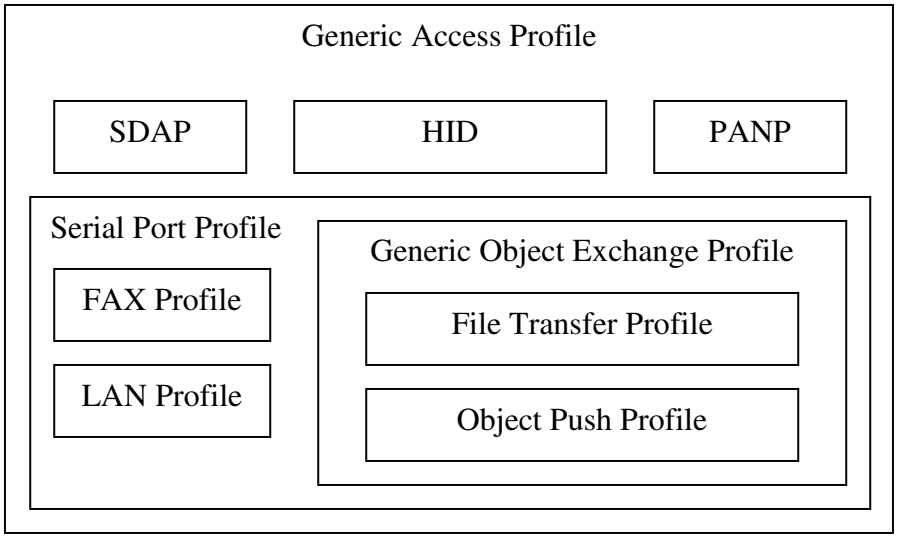

Figure 12: Bluetooth profiles

4.2.2 Stack integration

The integration of the Bluetooth stack with Obigo is made internal at Teleca in Lund. The stack is controlled by a Bluetooth Manager Service inside Obigo, which controls the message passing to and from the stack. This is discussed on the next page.

Figure 13 shows the integration of the Bluetooth stack with Obigo.

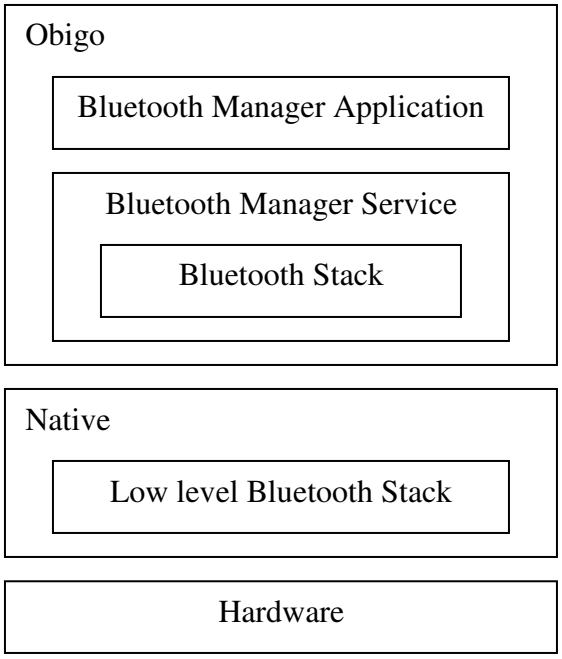

Figure 13: Bluetooth stack integrated with Obigo

When integrating the Bluetooth stack, a couple of issues had to be concerned. Should the stack be fully integrated into the Obigo framework, or should the stack be in the native environment? The result is that most part of the stack is running inside Obigo, with only the most critical parts left outside. Those critical parts are the timer and the read and write process that must run continuously.

Since Obigo and the Bluetooth stack use different message passing systems, an adaptor-callback interface was created. When sending a message from Obigo to the stack, a call to a suitable adaptor function is made. This function transforms the call into a message and passes it to the appointed receiver, the process receiving queue. If a response is possible to the original call, then a callback function is specified in the passed message. Therefore the receiving process knows which function in the interface is the correct for the returning message.

4.3 JSR-82 / Bluetooth

4.3.1 JSR-82 integration issues

JSR-82 has support for the most common used Bluetooth profiles. The scope of the JABWT was defined this way because of the diversity with the many different devices

and user scenarios Bluetooth is involved in. By having this support for the Bluetooth base profiles, the JSR-82 could quickly be introduced at the market. Because the higher profiles in the Bluetooth technology are based on the lower ones, and extend their functionality, developers are free to implement the higher profiles too. The profiles that are supported are the Generic Access Profile, the Service discovery Profile, the Serial Port Profile, and the Generic Object Exchange Profile. The Bluetooth Control Centre and the Service Discovery Database are also abstracted from the Java API.

JSR-82 is stack and radio independent which make it very interesting to companies that will add Bluetooth functionality to their platform. It is also very easy to develop Bluetooth applications with Java compared to C or C++. Therefore Java and MIDlets is a very up-and-coming technology when talking about wearable devices like mobile phones, and Bluetooth applications are today widely used. JSR-82 will for sure be a very used package in most Java Runtime Environments on the handheld market.

4.3.2 Textone

Textone is a leading mobility solutions provider in the European marketplace. The company is a member of the JSR-82 expert group and of the Bluetooth SIG, and among their products a Java/Bluetooth solution can be found. This implementation uses standard Java APIs and the optional package JSR-82 for communication with the Bluetooth stack. This solution is called the Bluetooth Technology Licensing Kit (TLK), and adds Java/Bluetooth capability to the customers' platform. This product is adapted to platforms that already have a Bluetooth stack integrated, and who wants to make Java MIDlet applications make use of it. Ericsson has a co-operation with Textone where the companies have integrated their products; Textone's Java/Bluetooth solution with Ericsson's Bluetooth stack.

This part of this investigation chapter, will focus on the TLK from Textone, and how to integrate it with a Bluetooth stack. The integration of Textone's implementation of the Java APIs for Bluetooth Wireless Technology with the Bluetooth stack from Ericsson will also be investigated.

4.3.3 TLK

TLK stands for Technology Licensing Kit, and is a product that adds Textone's implementation of the JSR-82 APIs to the target platform. The TLK also consists of some porting and development tools.

The TLK is divided into three different parts: JABWT, the asynchronous event manager, and the RACS (Figure 14).

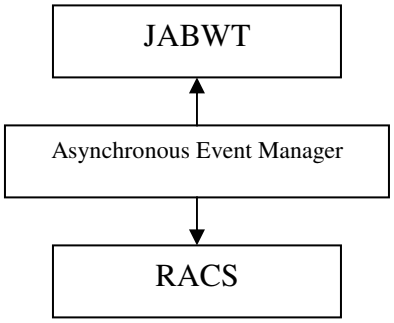

Figure 14: TLK architecture

The JABWT in the picture above is Textone's implementation of the Java APIs for Bluetooth Wireless Technology, and follows the JSR-82 standard defined by the expert group of the Java Community Process. The RACS, Textone Abstract C Stack, is an integration layer written in C that implements the Java methods declared in the JABWT. The RACS works as an interface between the JVM of the target platform and the Bluetooth stack, and is an event based model that uses asynchrony communication to connect the Java Bluetooth abstractions to the Bluetooth API. Its structure and design is made as generic as possible, so that the porting to different stacks on different platforms should be as easy as possible. To have this RACS layer integrated with the JVM (KVM) of the target platform, either the JNI Java API or the JVM native integration mechanism is used.

The Asynchronous Event Manager is a layer between the JABWT and the RACS, which dispatches events asynchronously from the layer below to the layer above. Because many Bluetooth stack APIs provides an asynchronous event based mechanism, the RACS uses the KVM asynchronous native methods for integration with the stack.

A general structure and design of a JSR-82 compliant architecture could contain the following parts (Figure 15):

- **MIDlet:** a Java application that uses the Bluetooth technology. It could be for example wireless multiplayer gaming, or connection with a printer server.

- **Local device manager:** This component is responsible for implementing the logical HCI connection with the Bluetooth device, and the remote device discovery and the remote service discovery.

- **Remote device and service discovery engine:** responsible for the remote devise and remote service discovery. Accesses through the agent in the JSR-82 stack.

- **Bluetooth connection:** A manager that communicates with the two connection profiles GAP and SPP, depending on the discovered service address.

- **Physical connection:** The connection between the physical device, and the controller that controls it.

- **GAP connection:** The connection that manages the L2CAP communication protocol. Accesses the JSR-82 stack through a separate thread.

- **SPP connection:** The connection that manages the RFCOMM communication protocol. Also accessing the JSR-82 stack by creating a thread.

- **Messaging routing engine:** The synchronization engine for the GAP and SPP threads makes the data transfer reliable and stabile.

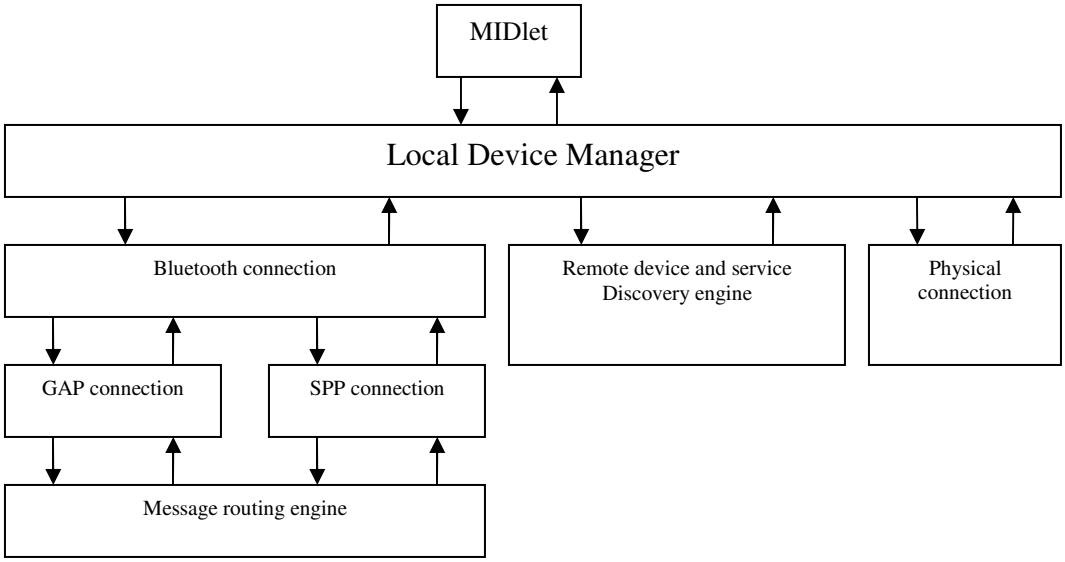

Figure 15: TLK solution architecture (1)

Next step is to add the TLK from Textone into this architecture (Figure 16). The Message routing engine and the remote device and service discovery engine should communicate with the JABWT implementation of the Java stack. And the connection between the module that controls the device and the device itself should be viewed in this model:

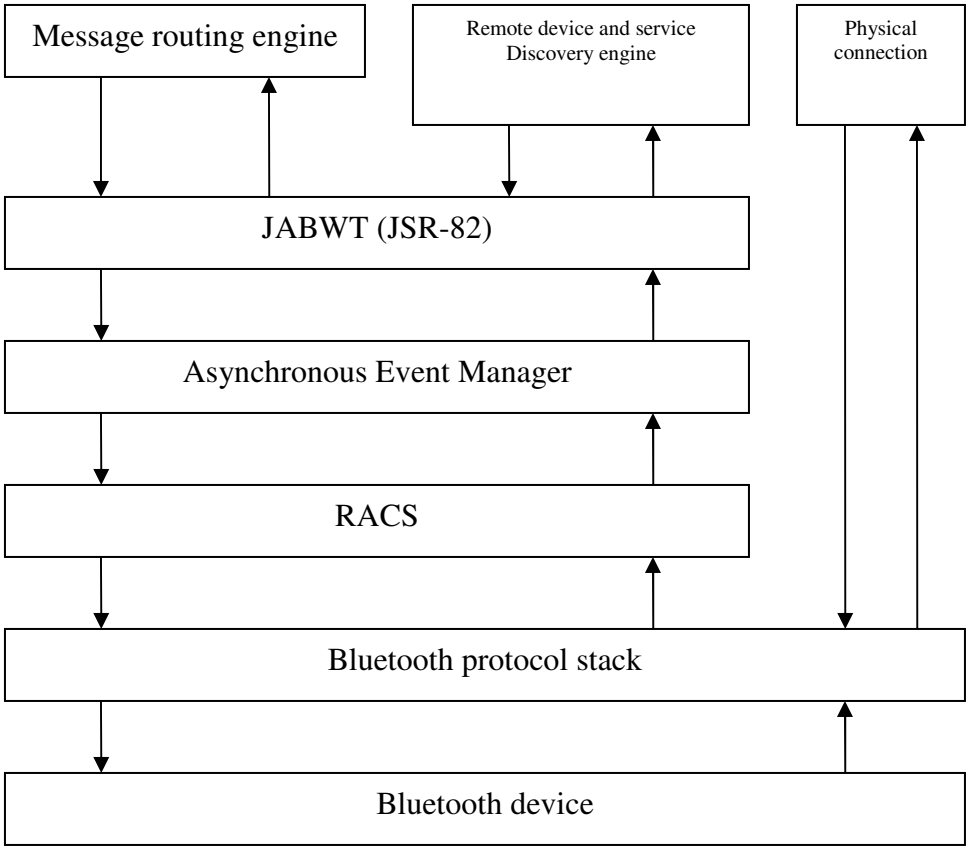

Figure 16: TLK solution architecture (2)

The Bluetooth protocol stack and the Bluetooth device are also included in this model. This is a general picture of the JSR-82 architecture that could be applied as a fundamental image about how the integration of the JSR-82 API could be done.

4.3.4 Using the TLK

To implement applications those make use of the TLK functions, and the Bluetooth technology, standard Bluetooth functionality calls are used. The procedure can be divided into several steps: device discovery, creating services, accessing services, using RFCOMM, using L2CAP, and Bluetooth security.

The device discovery process consists of two main parts: the device inquiry where the device searches for other Bluetooth devices in the area, and the device retrieval where the device queries a pre-configured list of devices or uses the results of a previously device inquiry.

The device inquiry can be divided into four different steps. At first a Discovery Agent must be obtained whereupon the discovery listener interface must be implemented. Then the Discovery Agent must be used to initiate the search, which also will supply the listener with notification of the events. The last step is to verify that the detected devices are of the appropriate class.

A device retrieval is an option to the sometimes very time consuming device inquiry. The device retrieval makes access to other devices without a device inquiry. This can be done by either using the results of a previous device inquiry, or to query a pre-configured list of devices. The first step is to obtain a Discovery Agent, in which a retrieve-devices method can be found and used. To have this method return a list of devices from a previous inquiry, a cache over devices is passed as argument. To use a pre-configured list of commonly used available devices, a list of pre-known devices is passed as argument to the retrieve-devices method.

To make the services of a Bluetooth available to other devices, they must be registered. This is called service registration, and stores the services of a device into the Service Discovery Database (SDDB). If the services of a device is changed or removed, the service must be updated to provide other Bluetooth devices with the latest information. At first a service record must be created. The service record describes the service and what are required to access it. This is done with the Service Record interface, where the attribute ID and attribute value type are set. When the service is ready to be published on the network, an accept-and-open method is called. The record will then be stored in the Service Discovery Database.

When a Bluetooth device has gone through the process of detecting other devices, it is time to investigate which services are available on the found devices. This is called service discovery. The device that has been requested for its services sends a list back as a response to the service discovery. Because a server can update its services, other Bluetooth devices can poll the server for updated information. To discover a service in a Java MIDlet you must at first obtain a discovery agent, and then implement the discovery listener interface. Then a search-services method is called to retrieve the accessible services on the device. When a service is found, a connection to it must be created. This is done by first use the service record returned by the server to get a connection URL, whereupon this URL is used to connect to the service.

RFCOMM emulates serial communication of a RS232 port between two Bluetooth devices. There are restrictions of how many connections that can be multiplexed over a RFCOMM session, which makes some demands that the programmer must show consideration for. To create and accept a server side RFCOMM, the server must first

create a special URL (Uniform Resource Locator) that describes the service and create a service record. Then this service record must be made available to the clients, and the server must also accept client connections. Then it is time to send and receive data to and from the client. To create a client side RFCOMM, the client must first perform a service discovery to found available services and to get the service record. Then the client must construct a URL with information from the service record, and open a connection to the server. Then data can be sent to and from the server.

The layer below RFCOMM is L2CAP. L2CAP makes it possible for Bluetooth devices to communicate with applications and higher layers of the stack. The JSR-82 API supports connection oriented L2CAP connections only, there is connectionless L2CAP communication too, but those are in order not supported in JABWT. To establish a server side L2CAP connection, the server device must create a service record and construct a URL that describes the service. Then the server must make the service available, and accept connections from clients. Then a communication between a server and a client is possible and data can be sent between them. If the client wants to create a L2CAP connection, the device must at first perform a service discovery. When the service record for the service is received, the client must construct a URL by using this service record. Then the client can open a connection to the server with this constructed URL. Now data can be sent between the client and the server.

Every application that runs on a Bluetooth device has it own security and configuration settings. The Bluetooth Control Centre is the software that controls those settings on different applications. Two Bluetooth devices can be associated together with a key. This process is called bonding, and is a requirement for further security functions like authentication, encryption, and authorization. For a device to be authenticated, it must send the key to the other device, to prove that it is the one he claims to be. Encryption is the process of distortion upon a collection of data so it differs from the original data, and so that no one without the decryption key can decrypt it. The last security challenge, authorization, is the process to determine whether a client should get permissions to access the data he wants to access. Only a service can request authorization. When adding security to Bluetooth communication using JABWT, optional security parameters are added to the connection URL. For example, a client connection to a RFCOMM server can be encryption protected by adding the encryption parameter to the connection URL on the server side.

4.3.5 Textone TLK and Ericsson Bluetooth stack

To provide a complete Java/Bluetooth solution, Textone and Ericsson have combined their solutions; Ericsson's Bluetooth stack integrated with Textone's JSR-82/Bluetooth

solution. The solution is produced to work in wearable devices, such as mobile phones, that do not contain any Bluetooth stack.

Platform independent applications (MIDlets) are placed above the JSR-82 optional package in the J2ME stack. This integration of Textone's JSR-82 and Ericsson's Bluetooth stack requires that the target platform has a JRE already integrated. The RACS needs to be integrated with a JVM to communicate with the Bluetooth stack, so a CLDC configuration with a KVM, and the MIDP profile that enables downloading of MIDlets is a requirement on the target platform.

This integration provides a solution that has no dependencies in the target operative system. Ericsson's VOS layer abstracts such dependencies and handles issues such as memory management and message passing. The only aspect that must be well considered is the multithread support in the target operative system. An asynchronous model is used by the TLK for communication with the Bluetooth stack and for integration with the KVM. The Bluetooth stack in turn provides an asynchronous API that models the Bluetooth protocol event mechanisms. To make this fit, the RACS uses the asynchronous native interface methods in the KVM as the most effective approach to stack integration. The TLK is based on a multithread approach, where the threads block waiting for an event, and where each thread has a single message queue. If the target platform for the TLK is non-multithread based, another ways of event processing must be considered. The TLK uses macros to deal with this issue. VOS_SEND and VOS_RECV are two macros defined by TLK, that map to the Ericsson Bluetooth stack functions Vos_Send() and Vos_Receive() in a multithread environment. If the target platform does not support multitasking, those macros must refer to other functions that handle the alternative approach for multi-tasking.

4.4 Conclusions

This chapter is the result of the investigation about the integrations of PEmb with Obigo, the Bluetooth stack with Obigo, and JSR-82 with the Bluetooth stack. Those integrations are all made independently of each other, and the purpose with this investigation was to have more knowledge about those, and how to carry on with the integration of the JSR-82 API with Obigo.

4.4.1 Java / Obigo

The integration of a Java Runtime Environment with Obigo (figure 17) is the most interesting for the future of this thesis work.

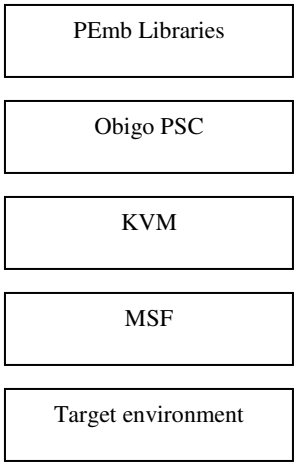

Figure 17: Java / Obigo integration

The PEmb Library is supporting the JSR-82 API, but since this PEmb library is delivered as pre-compiled source code, a new build including the JSR-82 implementation must probably be inquired from Profix. This is though a design issue discussed in the design chapter.

4.4.2 Bluetooth / Obigo

The integration of the Bluetooth stack with the Obigo is also interesting for this thesis project. In a final solution version of Obigo where there is possible to download a Bluetooth MIDlet and communicate to another Obigo device, there must besides the J2ME stack, also be a Bluetooth stack integrated in the same device.

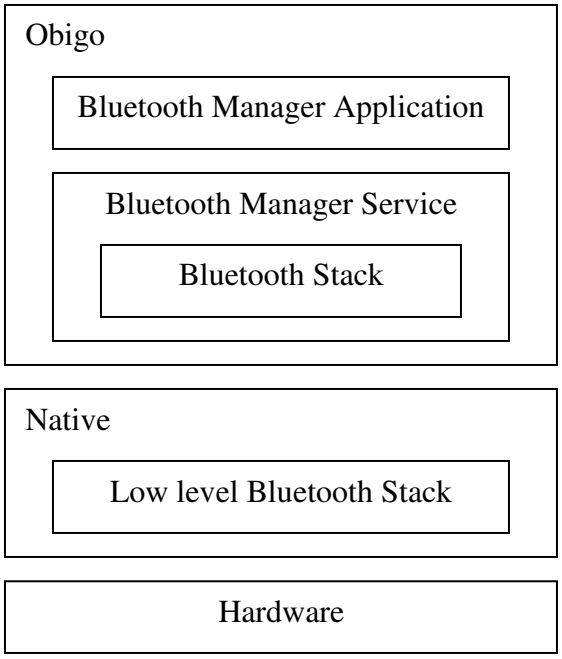

Figure 18: Bluetooth / Obigo integration

This integration of the Bluetooth stack with Obigo (Figure 18) is made placing the main part of the Bluetooth stack inside Obigo, and the low level parts in the native environment. This solution was chosen because it was less time consuming than the other solutions, but still easy to port to another target environments.

It is undecided whether or not this kind of Bluetooth integration will be used in a final Java / Obigo / BT integration. This because of restrictions Ericsson recently has done according to the usage of their Bluetooth stack. To integrate another Bluetooth stack is outside the scope of this project, it would be too time consuming. Instead a Bluetooth simulator will be used for testing and demonstration of the JSR-82 integration with Obigo.

4.4.3 JSR-82 / Bluetooth

The integration of the Java API for Bluetooth wireless communication, the JSR-82, with the Bluetooth stack is made by the software company Textone, using Ericsson's Bluetooth stack. This solution is constructed with the purpose to integrate with an existing J2RE in a wearable device, for example Profix PEmb investigated in this document. The relationship, if there are any, between Profix PEmb and Textone's TLK / Bluetooth integration, will be discussed in the next chapter.

4.4.4 Integration design and specification

The design chapter will focus on how to integrate the JSR-82 with Obigo Q-line. The integrations of this investigation will be used to the most extent possible, but new angles of approach must also be done.

5 Design

This chapter describes how an integration of the JSR-82 API with the Obigo framework can be done. This integration design will be based on the previous investigation that has been made, gathered in the Integration Investigation chapter, and on the conclusions from that investigation. A brief explanation of the integration is that the JSR-82 library from Profix will be integrated with the Java / Obigo integration, i.e. the PEmb that is integrated with Obigo. This is followed by a possibly integration of the Bluetooth stack with this assembly, in any case a deliberation of how such an integration could be done.

Different approaches will be discussed, and one of them will be chosen for the actual integration. Questions as how the already existing integrations can be used and possibly reused will be answered, illustrations of the integration will be made, the PSI interfaces and the PSC components will be explained and discussed. Data flows of the interfaces are also important points that will be considered. The Java KNI will also be discussed as a detached section, as a completely different approach of what can be done to create a product where Profix PEmb is disregarded.

In the Java version of Obigo there is no Bluetooth stack integrated. How can this be solved? And in the end, if there is no Bluetooth stack integrated, how can an integration of the JSR-82 with the Java / Obigo version be demonstrated and presented? Those questions will be answered within this integration design.

Also conclusions about the further work in this project, i.e. if an actual integration within the scope of this thesis work is possible, and if so, how this integration will look like and what is going to be made to have a working demo in the end of this project.

The purpose with this integration design is to come to certain conclusions that will simplify the actual integration of the JSR-82 API. One purpose is also to elucidate different approaches, and other ways and ideas of how the JSR-82 can be used within the Teleca Company. Output from this design will be greater knowledge about the integration, and the approach that will be used for a possibly integration.

5.1 Approach

The output from the Integration Investigation is very interesting for the design of the integration of JSR-82 with Obigo. From a higher point of view there is many different ways to go, different solutions that all could lead to a complete JSR-82 solution. Profix PEmb is today integrated with the Obigo Q-Line. The PEmb library is delivered as pre-compiled source code that is connected to the KVM module inside Obigo. One approach of integration is to add the JSR-82 library into the existing PEmb library, and have it connected via interfaces to the Obigo component that implements the Java interfaces (Figure 19).

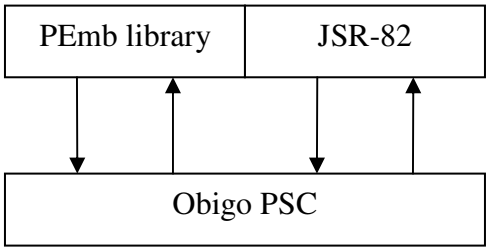

Figure 19: JSR-82 lib integrated with the PEmb lib

Another approach would be to keep the existing PEmb library intact, and let the JSR-82 be a standalone library (Figure 20). This would require some kind of interfaces against the PEmb library that handles the JSR-82 calls.

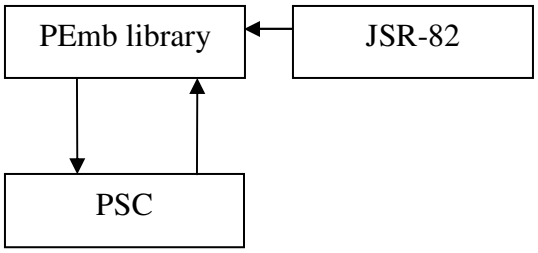

Figure 20: JSR-82 as a standalone library

There are different advantages with those two examples of integration. The first example will require a completely new PEmb build delivered from Profix, containing also the JSR-82 library. Whether this is possible or not is being discussed outside the scope of this thesis work. But this solution will set the fact that the existing PEmb architecture can be used. The interfaces declared from Profix must be implemented in Obigo PSC to have the JSR-82 API communicating with a future Bluetooth stack. The other integration example would not require any new PEmb build from Profix, but only their implementation of the

JSR-82 library. In this case some interfaces towards the PEmb library must be constructed to have the JSR-82 API work together with the KVM and the CLDC/MIDP. There is also unknown if Profix have any standalone version of the JSR-82 implementation and this type of interfaces, or if the only possibility is to have it delivered inside the PEmb library. In any case, another way to do this is to leave Profix outside this integration, and have another JSR-82 implementation integrated with Obigo, for example one that is being developed at Teleca (Figure 21). In this case Teleca also has the possibility to deliver a JSR-82 solution independent of a JRE, in this case PEmb, to a customer that already has a KVM integrated in their environment but wants support for the Bluetooth technology. This can be solved by constructing interfaces with the Java KNI methods instead of using the PSI interfaces from Profix.

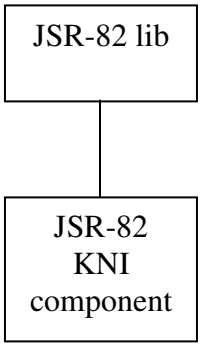

Figure 21: JSR-82 integration with the Java KNI

For this thesis work the only solution that is possible within the scope of the project time and manpower, is the first one where a new build is delivered from Profix. The interfaces towards the JSR-82 library inside the Obigo PSC would not be very time consuming to implement. If a new PEmb build containing the JSR-82 library is delivered from Profix, this PEmb can without too much effort be integrated with Obigo. Then the first step is to construct some dummy interfaces that make it possible to build this new solution. Then those JSR-82 PSI interfaces can be implemented to have the Java Bluetooth methods inside the lib work correctly, and have them ready for being used inside Bluetooth MIDlets.

5.2 Btone Simulator

One problem with this integration is that there is no Bluetooth stack integrated within this version of Obigo. An integration of a Bluetooth stack with this Java/Obigo version can be done, but there is an imminent risk that such integration would be too time consuming for this thesis work. In that case, the communication between the JSR-82 library, the KVM and the Bluetooth stack would not exist. The purpose with this thesis work was not to

integrate a Bluetooth stack, but there may be a problem to demonstrate the functionality of the JSR-82 integration without a Bluetooth stack. The solution to this problem is the Btone simulator from Textone. The purpose with the Btone simulator is to simulate Bluetooth communication between emulated devices; which is well suited for this thesis project.

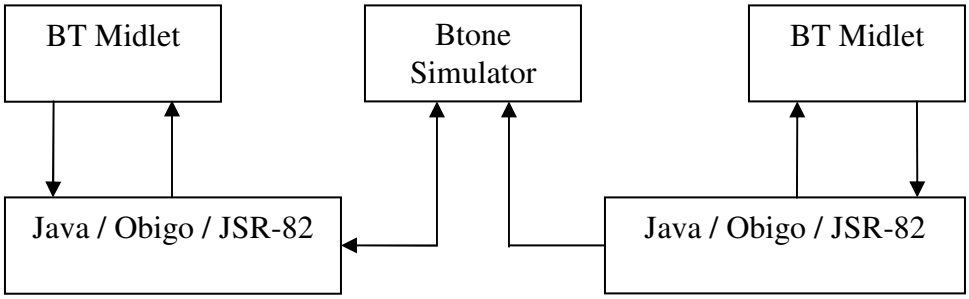

Figure 22: Software configuration using Btone Simulator

This simulator will make it possible to integrate the JSR-82 library with Obigo, implement the necessary parts of the PSI interfaces, and finally have two Obigo devices communicating with each other without having a Bluetooth stack integrated, i.e. simulate JABWT connections without the actual Bluetooth hardware and software. This is illustrated in figure 22. Figure 23 shows two emulators communicating with each other with Btone Simulator.

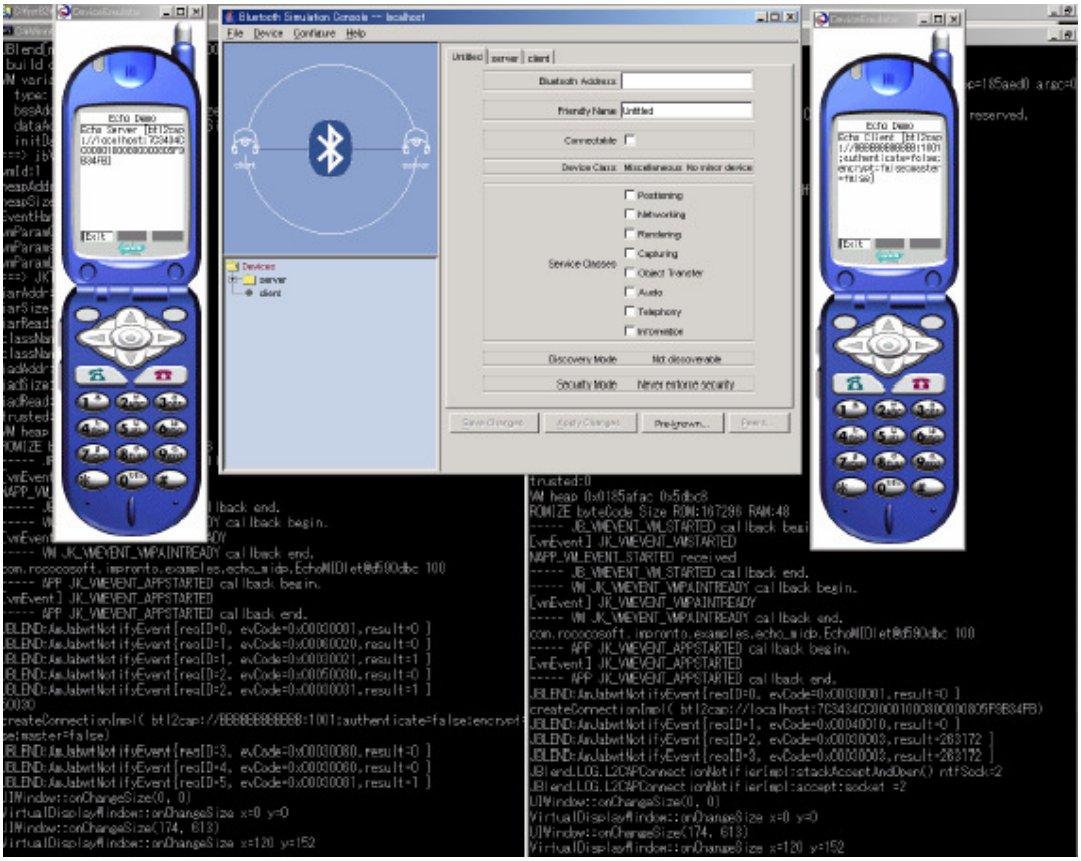

Figure 23: Two client's communication via Bluetooth through the Btone simulator

5.3 Interfaces

PSI is an abbreviation of PEmb Service Interfaces, which are delivered together with the PEmb from Profix. Those are interfaces specifying the features that must be implemented in the PEmb libraries on the target platform. The components that are implementing the functionality according to those interfaces are called PSC (Figure 24).

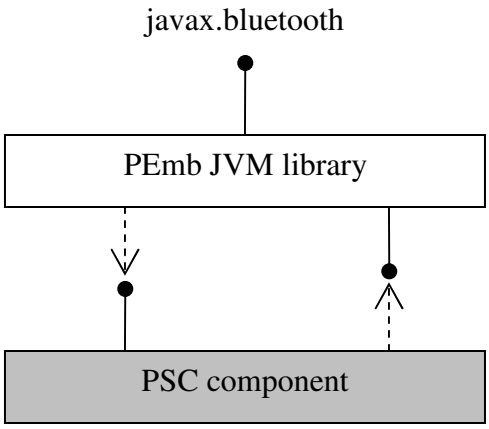

Figure 24: The PSI interfaces and the PSC

The shaded part of figure 24 is the component to be implemented by the manufacturer. The JBI is interfaces already implemented in PEmb for use by the native system in returning to PEmb the result of different processes. In the integration of Java with Obigo, the JVM Support Component is named Obigo PSC. There the PSI interfaces for Java support are implemented. Within this thesis project, the JABWT PSIs are the interesting interfaces. When the JSR-82 library is eventually integrated with Obigo, the necessary parts of that PSI should be implemented inside the Obigo PSC. PSIs are interfaces that PEmb assumes are implemented in the native system; otherwise there will be errors when compiling the integration source code. Therefore the compiling of the JSR-82 / Java / Obigo integration will generate errors in the first step. Some loop back interfaces will be implemented to have this integration compile without errors, and then the actual functionality must be implemented, according to what is needed from the Btone simulator. Unfortunately the lack of a Bluetooth stack in this version of Obigo requires some kind of virtual Bluetooth functionality being implemented in the PSC to have Obigo work properly with the Btone Simulator.

5.4 KVM

The KVM is a reduced version of the Java Virtual Machine, targeted at devices with limited resources. The KVM is located in the lowest layer of the J2ME architecture and acts as an abstract computing machine, and that is why it is named the cornerstone of the Java platform. The KVM is the contributory cause of the platform independence, and is the only component in the J2ME platform that must be re-implemented when porting a JRE from one platform to another; the other layers can be written once and used everywhere.

The KVM consists of a class loader, runtime data areas, an execution engine, and a native method interface.

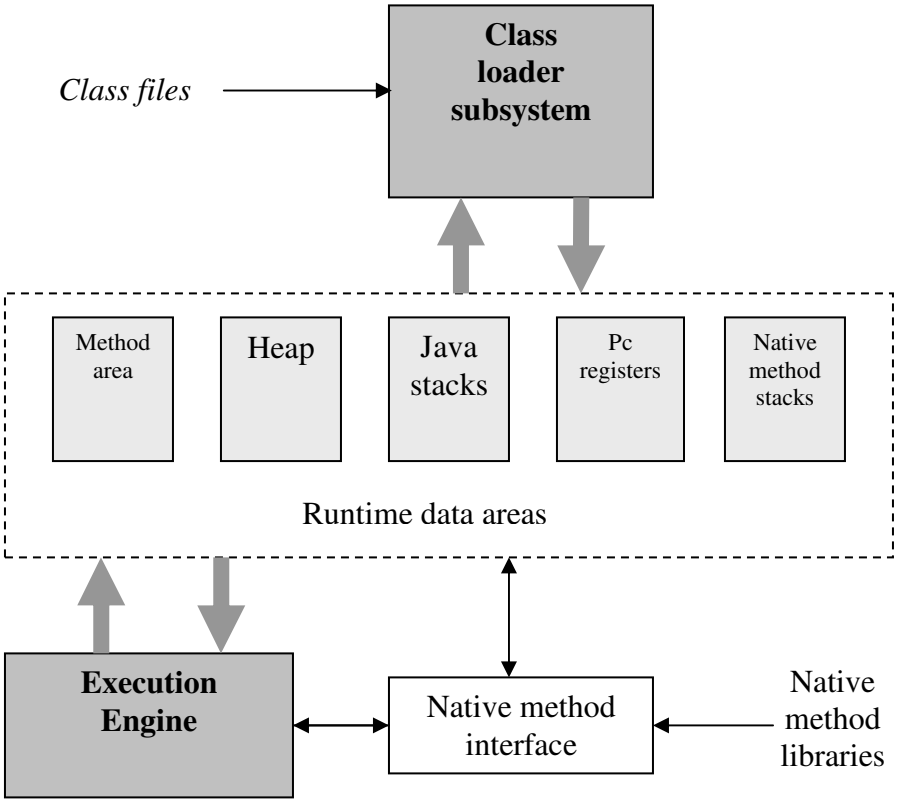

Figure 25: KVM architecture

The class loader loads the class files from the compiled API implementation and the actual application implementation. The runtime data areas are needed by the KVM to store byte code, temporary variables, class information, object instances and parameters during execution time. The execution engine, also called interpreter, executes the byte code that the class loader has loaded, i.e. translates it to machine code that the underlying operating system can understand. The native method interface is a component that enables Java methods to make system calls to native functions. This process is illustrated in figure 25.

5.5 KNI

As discussed earlier, the Java platform provides the KNI (K Native Interface) API to ease the integration of programs written in the Java language with existing non-Java language services. The KNI is incorporated in the KVM, and defines a standard naming and calling convention so the Java virtual machine can locate and invoke native methods.

Suppose a JSR-82 implementation is done at Teleca. This Implementation would use methods declared native, for example Native Method Start_Inquiry(...). This Native function will work as a bridge to be able to run the native function on the target platform. An interface written in C could be implemented for handling data between the KVM and the Bluetooth stack. This C integration layer must include a header file created by the Java Native function, to map the Java functions to the C functions. This is illustrated in figure 26.

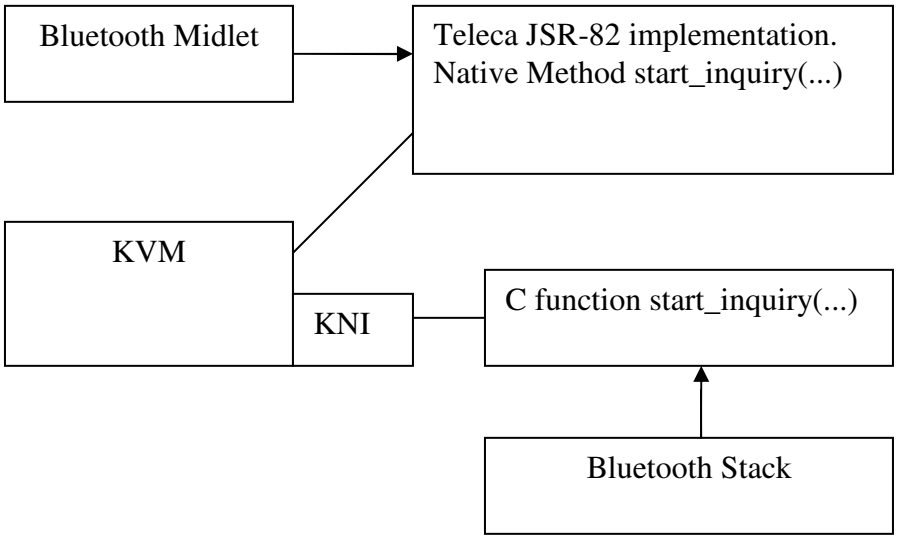

Figure 26: Communication with the KNI interface

In this case Teleca can offer a solution to a customer that already has a Java Runtime Environment integrated in their platform. The C integration layer has to be integrated with the KVM in the target platform, after which the Java KNI will be used for mapping the Java methods declared in the JSR-82 implementation to the actual C implementation of the methods. The C integration layer should be as generic as possible so the porting to different stacks will be as easy as possible.

5.6 Conclusions

The approach of the integration of JSR-82 with the Java / Obigo integration has been decided. A new library build will be requested from Profix that can be integrated with the Obigo Q-line. Loop-back interfaces will be constructed to have the integration compile without errors. The necessary parts of the PSI interfaces will be implemented, and the final integration will be demonstrated with the Btone Simulator from Textone. Bluetooth MIDlets will be downloaded into two Obigo devices, where after those will be connected

to each other through the Btone simulator. Depending on how well the Btone simulator handles the Obigo devices, and how much of the PSI interfaces that will be implemented, the level of the Bluetooth MIDlets will be set.

In best case a Bluetooth stack will also be integrated within this thesis project. This integration is not planned, but depending on how well things turn out, there might be time left for this stack integration.

6 Integration elucidation

The main task with this thesis work was from the beginning according to the project specification to integrate the Java API for the Bluetooth Wireless Technology with the Obigo Q-Line, together with an arbitrary KVM. But during this project, besides the integration work, as an outcome from the many investigations, things that seemed much as interesting as an actual integration have emerged. Those "things" has been investigated and collected into two main questions:

- What different JSR-82 solutions is it possible to develop within the Teleca Company?
- How much will it cost in terms of licenses, compatibility tests, and manpower for Teleca to deliver those JSR-82 software solutions to an end customer?

This chapter is the result from an investigation about those main questions. How many solutions is it possible to develop with the preconditions that exist today? A JSR-82 implementation may be integrated with Obigo within this thesis work, but what is needed to finalize such integration and make it a product that can be delivered to a customer? This JSR-82 implementation is still something that another company has developed; is there any way to produce such an implementation without any help from outside? The needs from customers are slightly different, so what can be done to satisfy them all; many different products, or one solution that suites everyone?

The other question is mostly a money issue. Which license agreements must be signed? Which TCK tests must be carried out? But there are also costs in terms of manpower and time that must be unraveled; how long will it take to develop those solutions, and is it a small one man work or a huge team work?

The first part of this chapter deals with the work that has to be done to integrate JSR-82 with Obigo, and to develop the different solutions. Then the questions about the time estimation will be discussed, followed by which licenses and tests that must be taken.

The purpose with this chapter is to answer the main questions above. The output can hopefully be used by the Teleca Company as a help and guideline in their decision about whether this JSR-82 API should exist as a Teleca product.

6.1 The work

There exist two different versions of Obigo that are interesting to this project, the Bluetooth / Obigo integration and the Java / Obigo integration. The second of those two are the result of cooperation with the Profix Company and the incorporation of their PEmb into Obigo. The PEmb is as declared in the Integration Investigation chapter, delivered as pre-compiled source code. The existing integration with Obigo (Figure 27) does not have any support for the JSR-82 API, but such a library is still a part of the PEmb.

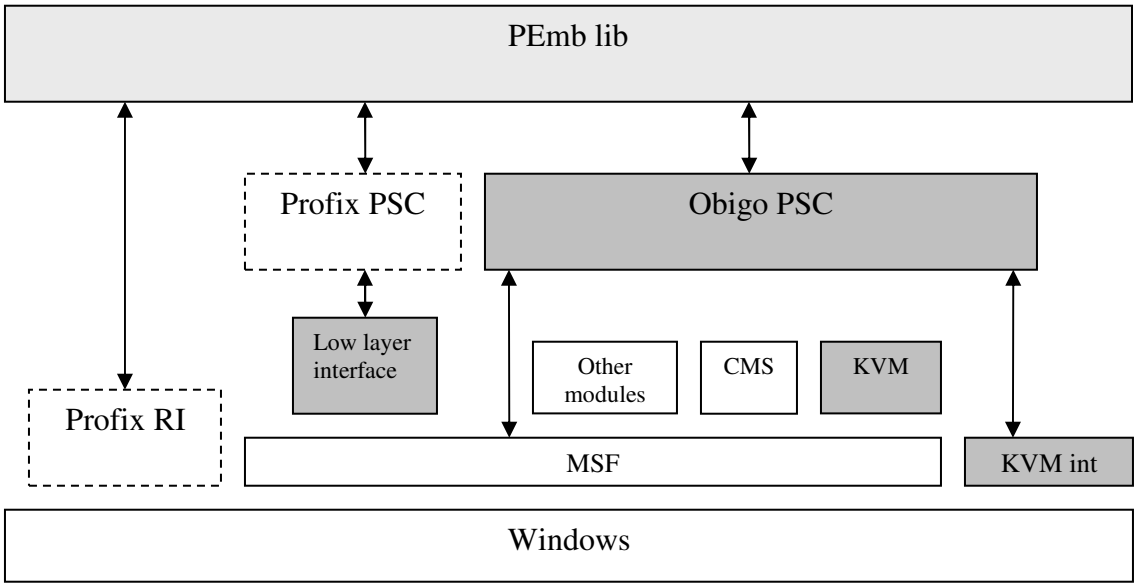

Figure 27: Integration layers, PEmb and Obigo

The layers of the integration of PEmb with Obigo are presented above. According to the integration Design chapter, a new build of PEmb will be used. This means that a new version of the PEmb library in figure 27 will be re-integrated with the rest of the layers. This version will thus also contain a JSR-82 implementation. The existing PSI interfaces will be kept intact, but new JSR-82 PSIs must be implemented to enable communication with a Bluetooth stack. Because there is no Bluetooth stack integrated with this version of

Obigo, only parts of those PSI interfaces can be implemented, the ones that communicate with the PEmb library. To have a fully working JSR-82 Bluetooth Obigo version, a Bluetooth stack must also be integrated with this solution. That is yet not within the scope of this thesis project and will probably not be accomplished. All this integration work will be carried out with Visual Studio 6.0.

From Teleca's point of view, there are besides the PEmb / Obigo integration also other interesting solutions that are not a physical part of this thesis work, but that has emerged through the many investigations and therefore have become a theoretical part of this project. There are two different JSR-82 solutions that seem interesting as future products. At first, there is a solution including an implementation of the JSR-82 API together with a C implementation and a Bluetooth stack (Figure 28).

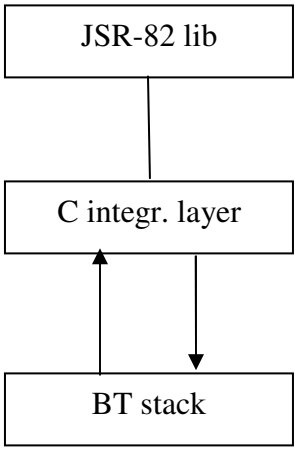

Figure 28: Solution for platform with JRE but without a Bluetooth stack

This solution is adapted to target platforms where a JRE but no Bluetooth stack is integrated. What has to be done here is an implementation of the JSR-82 API, a C integration layer that will be integrated with the KVM of the target platform, and the connection to a Bluetooth stack. The whole packet can be integrated with the KVM of the target platform via Java Native Interfaces (JNI). The JSR-82 implementation may be a third party product, or an implementation developed within Teleca. The C integration layer mustn't be generic but must be aimed to the integrated Bluetooth stack. The things that must be considered during integration is the message passing system, handling of threads, processing calls etc. in the native environment. The Bluetooth stack may also be a third party product or a Teleca implementation.

The other interesting solution is quite the same as the first one, but this time with no Bluetooth stack involved (Figure 29). A stack is expected to already exist in the target platform.

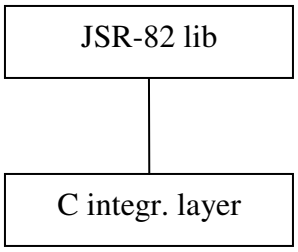

Figure 29: Solution for platform with JRE and with a Bluetooth stack

The JSR-82 task is the same as in the first solution, i.e. it may be a third party product or implemented within Teleca. But because the integration with the Bluetooth stack is going to take place inside the target environment, the C integration layer must be as generic as possible. It will be more an interface that must be implemented inside the target platform to adapt to its Bluetooth stack. The integration may still be handled with JNI, but an interface will declare the functions, or the C layer will only contain function shells, depending on the architecture of the target environment.

This two JSR-82 solutions enables two different customers have interest in the future products; those with an existing JRE but no Bluetooth stack, and those with both a JRE and a Bluetooth stack. This both solutions may also be detached from involvement from other companies.

Those solutions together with the outcome of the thesis exam work (Figure 30), where the customer may be one with a platform without any JRE or stack, seems to cover all demands from the JSR-82 customers on the Java Bluetooth market.

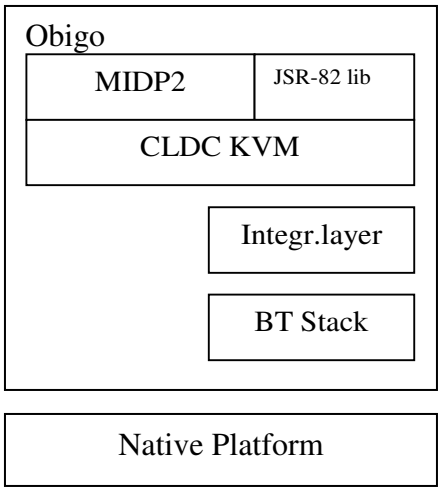

Figure 30: Solution for platform with no JRE or any Bluetooth stack

6.2 Time Estimation

To integrate the PEmb JSR-82 library with Obigo would not be too time consuming, a couple of days to have an error-free build that work properly with the loop back PSI implementation. The work with having the integration interact with the Btone Simulator and simulate Bluetooth communication will probably be more time consuming. Hopefully that can be done in two weeks. If that integration and interaction with the Btone simulator is working as expected, there may be some Bluetooth MIDlets implemented. That is about one week of work.

The time developing the two JSR-82 solutions may differ depending on what exactly will be done. If the JSR-82 implementation used is a third party product, then that part will only be a cost issue. An internal implementation of the JSR-82 API together with a C integration layer could be a rather time consuming task and falls into the category of a small project. To develop an intern Bluetooth stack is a very complex task and requires great knowledge about the Bluetooth technology. A third party solution does not require any implementation work, but is a cost issue. The best solutions for Teleca will be discussed and presented in the Solutions chapter.

6.3 Licenses and TCK

To understand those licenses and test issues, one must have some knowledge about the formal procedure from having an idea from a simple concept incorporated into the Java standard. This formal procedure is named JCP which is an abbreviation of the Java

Community Process. JCP allows anyone, from individuals to industry experts to have their own idea transformed into a set of Java classes that can be used by developers worldwide.

All Java functionality has to pass through the JCP where after that functionality in shape of an API is assigned a formal name. When an idea of new Java functionality is proposed, a formal JSR name is assigned to that functionality, followed by some numbers. JSR-82 is for example the formal name for Java Bluetooth functionality. The person or company that proposed this JSR is called the Specification Lead. Each JSR has a Specification Lead, and that company (or individual) is responsible for ensuring the delivery of the specification, reference implementation, and TCK, which is explained below. That does not mean that they must do the work or license it, but it does mean that they must ensure that someone does.

The specification lead then has to form an expert group for this JSR, which purpose is to create the official Java APIs for this JSR. In the JSR-82 expert group companies like IBM, Mitsubishi, Textone Software, Sony Ericsson and Sun Microsystems are participating.

According to the Java Community Process, the specification lead is responsible for the existence of a Reference Implementation and a Technology Compatibility Kit. The specification lead can create those of their own or let someone else do it; it is their choice and an economical issue. The Reference Implementation is a proof that the JSR also is possible to implement and works in a physical form and not only as a theoretical proposal. The TCK contains documentation of the JSR, and a set of tests that other companies must run to make sure that their own implementation of the JSR is compliant to the JSR standard.

Each platform requires licensing and certification for each JSR the platform wishes to certify. So TCK results for each individual JSR must be obtained and verified with the TCK owner. No matter where the JSR implementation originates, it is required that TCKs pass on the specific platform that will be claiming compliance. This is true for all J2ME certifications, not just JSR-82. A software company may provide a Reference Implementation for a specific JSR and claim it is TCK compliant, but the tests ultimately must still all be run on the specific platform/device that the code will be deployed on.

As described before, all JSRs for which a company wishes to include and claim compatibility with must pass the individual, specific TCK tests. However, a specification does exist (JSR 185, also known as Java Technology for the Wireless Industry, or

JTWI) which aims to provide an architectural description as well as an integrated Reference Implementation and TCK to coordinate selected JSRs specific to wireless devices. This type of specification is colloquially known as an 'umbrella JSR'. In practice, the JTWI TCK essentially is comprised of a number of other JSRs, and its TCK gathers all of these into a single suite of tests. This Umbrella JSR does not include the JSR-82 API; therefore separate tests must be accomplished to have the Bluetooth technology certified into the targeted platform. Still it is a possibility that the JSR-82 API will be incorporated in this Umbrella JSR in the future.

It is interesting to consider whether it is any differences by implementing an own JSR-82 solution, or buying and integrate another company's solution. If the third party implementation that is going to be integrated already is TCK tested, must this solution run through the TCK tests again for every new platform the company integrates with? The answer is yes. Even if a device manufacturer licenses an implementation of JSR 82 from a 3rd party, and that implementation is delivered and integrated in binary form into a final product, that specific product must still certify on the specific platform it ships on. The reason why is that even binary compatible libraries can behave in unpredictable ways when combined with specific other code on a final product. From one JSR to the next this may differ, but in general this is what has been proposed by Sun and is generally followed in the industry. Interesting scenarios have developed which point out the validity of this approach. For example, consider if Rev A of a product ships a certified version of a JSR for a couple of years, and when Rev B of the device is released, bug fixes or newer revs of other code cause the originally certified TCK to fail, even though none of it is code (or the code it directly links with) has been modified.

This should be interesting from Teleca's point of view, and is discussed in the Solutions chapter.

If a company wants to develop a J2ME platform including MIDP2 and JSR-82 functionality (according to the integration of JSR-82 in this thesis project), there is a certain procedure that must be followed to claim Java certification. A device manufacturer must first take a license from Sun for J2ME. Only J2ME licensees can claim compatibility with J2ME specifications and TCKs. For each individual configuration, profile, and JSR, device manufacturers have the option of licensing 3rd party implementations or developing/contracting them on their own. In either case, to be able to claim compatibility with any J2ME API, a license must first be in place from Sun, and then the specific TCKs must be licensed from the owner and demonstrated as passing. Further additional licenses may be necessary if the code is not internally developed. In the case of MIDP2, Motorola is the specification lead; however, the reference implementation and TCK were Sun's responsibility. MIDP2 licensees may

obtain the TCK from Sun in order to achieve certification, after having licensed it from Motorola. Additional licensing agreements are necessary if a licensee wishes to use the MIDP2 Reference Implementation in a product. When it comes to JSR-82, Motorola is again the specification lead, but have also constructed the Reference Implementation and the TCK tests.

A device manufacturer that wishes to claim J2ME compliance must have the following:

- A license agreement from Sun for J2ME.
- A license agreement with a Virtual Machine vendor such as Sun, Profix, Esmertec etc. or an internally developed and certified virtual machine.
- A license agreement for each JSR to be included in the product. Licensing terms are specific to each JSR and must be obtained from each individual JSR Spec Lead company, and generally may include the TCK alone or both the Reference Implementation and TCK. Compliance with any JSR cannot be claimed unless a valid license for the TCK is in place and all TCK tests pass.

6.4 Conclusions

This chapter has described different possibilities of developing JSR-82 solutions within Teleca. What has to be done, and what can be done? How much time will it take, and what different licenses and tests must be bought?

There are three different solutions that should be interesting to Teleca: the solution for a platform with JRE but without a Bluetooth stack, the solution for a platform with a JRE and with a Bluetooth stack, and the solution for a platform with no JRE or any Bluetooth stack. The time required for developing those solutions, and the licenses that must be bought and the tests that must be run differs between those solutions.

In the Solution chapter some complete solutions are presented, suited to the Teleca Company. Rather than the question “What can be done”, the question “What should be done” is answered in that chapter.

7 Solutions

The result from this Master’s Thesis project is presented in this Solutions chapter. It will set an end to the previous chapters, and present the concrete JSR-82 Teleca solutions as an outcome from all the other investigations.

There will be three different JSR-82 solutions presented, designed, and investigated in form of time estimation and licenses that must be considered. The first solution is called the BT stack solution, and is aimed for platforms that have a Java Runtime Environment but no Bluetooth stack integrated. The second solution is called the Standalone solution, and consists only of a JSR-82 implementation and an integration layer. This is the solution with the least number of components, and is the one that has to be most generally constructed; it assumes a large number of components already integrated in the target environment and must therefore be able to interact with those. The third solution is called the Full Scale solution and is a complete J2ME platform including the JSR-82 optional package, ready to be integrated with the customers' platform together with the Obigo Q-line. This one differs a little from the two first solutions, because here Teleca's Obigo is also involved.

For each solution a design will be presented and explained. Also the final product presentation will complete the looks and features issue. After that there is a sub chapter considering the work that has to be done for completing every solution. The licenses and TCKs are very important issues in this thesis project, and are explained well for every solution. Time estimation will set an end for each solution, and that sub chapter explains how much time it will take to accomplish the solutions and the integration work.

7.1 Solutions outline

Today Teleca delivers MIDP2 functionality to their customers through the CMS module in form of clean source code. This CMS module does not cover all of the MIDP2 functionality because Teleca has no Java Runtime Environment in their Software framework Obigo Q-line. The functionality delivered has only to do with the CMS module in Obigo, and therefore only covers user actions like downloading, installing, support for push and for trusted MIDlets, and deleting of MIDP2 applications. To be allowed to deliver this functionality, Teleca has to pass their MIDP2 functionality through a number of compatibility tests from Sun Microsystems, because Sun is responsible for the MIDP2 reference implementation and the MIDP2 TCK. To be able to run and pass that TCK, the MIDP2 functionality that Teleca delivers has to be extended; there must be the whole kit including a KVM and the other required libraries for the J2ME platform. Those tests must run on the customers' platform, which means that the Obigo Q-line must be integrated with that target environment, and run through all the MIDP2 compatibility tests.

Teleca does not own a KVM and that is why the Profix Company is involved in this integration. Teleca has therefore concluded an agreement with Profix, which implies that

Profix product PEmb is integrated with Obigo Q-line in demonstrational purpose and to have Obigo run the TCK on the customers' platform. This will ensure that Teleca's MIDP2 functionality works correct in the native environment.

The rest of this chapter will be based on those preconditions, and will present the best solutions concerning the design, the work, the licenses / TCKs, and the time estimation.

7.2 BT stack solution

7.2.1 Design, the product, scenarios

This is the JSR-82 solution that is aimed at customers who does not have any Bluetooth stack integrated with their platform (Figure 31). It is supposed that they have a Java Runtime Environment which means that there is a KVM existing in their system, but now the company also wants their platform to be able to make use of the Java Bluetooth technology.

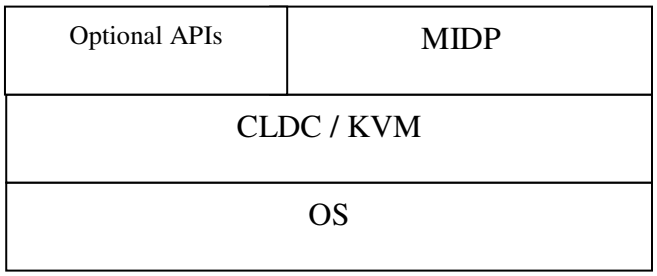

Figure 31: BT stack - Customer platform

What is needed here is a JSR-82 implementation, an implementation of the JSR-82 specification for Java Bluetooth Wireless Technology, i.e. the optional package placed on top of the CLDC / KVM layer in figure 31. That would make the KVM able to load Bluetooth class files imported in MIDlet applications. This is however only the communication between the JSR-82 lib and the KVM. To make use of the Bluetooth functionality, there must also be a Bluetooth stack integrated with the platform, so that the KVM can forward Java Bluetooth calls to the stack and to the physical device. This BT stack solution from Teleca would as the name implies also contain a Bluetooth stack. The JSR-82 implementation together with the Bluetooth stack is yet not a complete solution, because the KVM must have some kind of native-style communication with the Bluetooth stack. This Teleca product should therefore have an integration layer between the JSR-82 implementation and the Bluetooth stack (Figure 32). This integration layer can be used by the KVM to have the Java Bluetooth calls from the MIDlet communicate

with the integration layer and with the Bluetooth stack. The Integration layer is also used for the reverse communication, i.e. to reply the signals from the Bluetooth stack to the KVM and the MIDlet. This integration layer should be written as generic as possible to make the porting to different KVM on different platforms less complicated. However, the integration with the Bluetooth stack is already done between the integration layer and the JSR-82 implementation, so the integration that is left in this solution is the integration of the integration layer with the KVM in the target environment. This should be done using the native function interface KNI. The KNI is a native function interface that provides high performance and low memory overhead, and is developed as a logical subset of the JNI interface because there is no support for JNI in a KVM. The implementation of the JSR-82 API should then declare some native methods, and when the classes are loaded into the KVM, it uses the KNI interface to invoke the corresponding functions in the native system. Those functions shall then communicate with the Bluetooth stack and send the returning data back to the KVM, again using the KNI interface.

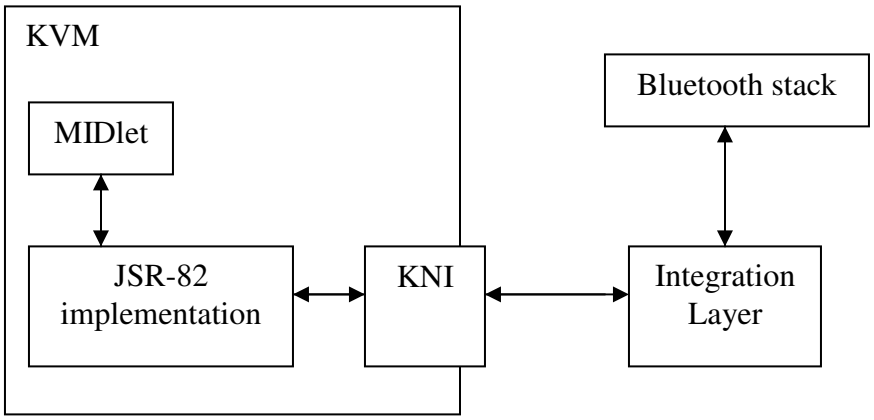

Figure 32: BT stack - Integration of the KVM and the integration layer

The Java KNI implementation in the JSR-82 and the KNI implementation in the Integration layer may look something like this:

```
Package jsr82Func_package;
            Public class BT_operation {
             Public native int BT_op_one();
             Public static void main(String[] args) {
                   New BT_operation().BT_op_one();
             }
            }
```

This BT_operation class defines two method declarations, the main method and the native method. When the MIDlet calls this Bluetooth function implemented in the JSR-82 implementation, it creates a new instance of the BT_operation class and invokes the native Bluetooth function. The native Bluetooth function may for example be implemented in the C programming language located in a source file inside the native system. The invocation of the jsr82Func_package.BT_operation generates a header file which specifies a certain function prototype definition. The native source file must follow this definition and should have the following appearance:

```
#include <kni.h>
#include <stdio.h>
KNIEXPORT    KNI_RETURNTYPE_INT    Java_    jsr82Func_package_
BT_operation_BT_op_one {
 …Source code for Bluetooth communication…
 KNI_ReturnInt(…returned value from BT stack…);
}
```

This code will have the JSR-82 implementation declare native methods, and the KVM will use the KNI interface to invoke the corresponding function in the native environment. This example function may be the getBluetoothAddress() function, where the returned Integer from the C function is the Bluetooth address received from the local Bluetooth device.

The final product from Teleca should therefore consist of three components; the JSR-82 implementation, the integration layer, and the Bluetooth stack (Figure 33).

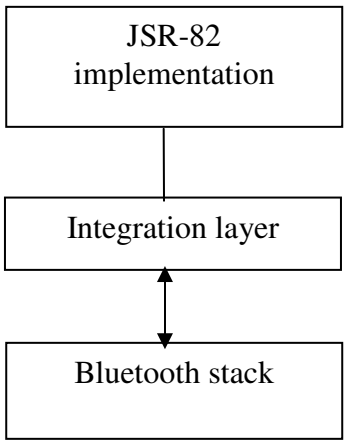

Figure 33: The BT stack solution as a product

The customers interested in buying this product are those who have a JRE existing in their platform, and now want to make use of the Java Bluetooth technology. Figure 34 illustrates this product integrated with the customer's platform.

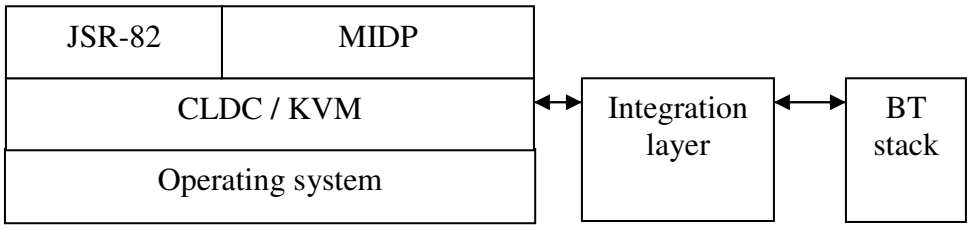

Figure 34: The BT stack solution integrated with customer platform

7.2.2 The work

Teleca must have the rights to a JSR-82 implementation, an integration layer, and a Bluetooth stack. Profix JSR-82 implementation can't be used, because this is delivered as pre-compiled source code which means that there is no possibilities to KNI modify that implementation. It is true that Profix JSR-82 implementation already supplies with interfaces, but these are connected to the PEmb library and the KVM inside.

The options are either to buy a third party source code JSR-82 implementation that can be modified so that it conforms to Teleca's integration layer, or to develop an implementation within the Teleca company. In fact, the JSR-82 is only a specification that defines the methods that must exist in an implementation of the JABWT, i.e. Bluetooth for Java. The JSR-82 implementation component itself does not actually do no more than receive the calls from the MIDlet and invoke the appropriate method inside the integration layer through the KNI interface. An actual JSR-82 implementation does not provide any large amount of functionality; it is the whole package including the integration layer and the Bluetooth stack that represents the JABWT functionality in a system. Buying a third party JSR-82 implementation and modify this so that it adapts to the integration layer seems like a quite labored process considering that the JSR-82 implementation itself shouldn't be that much of work effort. Making an internal JSR-82 implementation would give Teleca the opportunity to construct the library the way that suites the integration layer, and therefore this is the best option in this JSR-82 solution.

The integration layer is the component that implements the methods that JSR-82 specifies, and is the link between the JSR-82 implementation and the Bluetooth stack. This layer is a Teleca construction that suites both their JSR-82 implementation and the Bluetooth stack. Because most KVM are written in the C programming langue, the

integration layer should also be written in C. This will simplify both the communication with the KVM, and with the Bluetooth stack (The Bluetooth stack could also be written in Java, but that is not very likely). When speaking about the Bluetooth stack; the Bluetooth specification is a very technical document that describes all the functions that must be implemented to construct a product that follows the Bluetooth specification.

The Bluetooth protocol stack is a software component that implements the different communication protocols that are defined in the specification. The great technical expertise and the extensive needs for resources that is required for developing a Bluetooth stack, makes it almost impossible for companies to develop a stack on their own. The stack must pass the official Bluetooth test suites that are defined by the Bluetooth SIG. The predominant majority of all companies will therefore include the Bluetooth technology in their products by purchasing a commercial Bluetooth protocol stack from a third party developer. Internal development can't in most cases be justified. The most businesslike should therefore be to buy a third party Bluetooth stack implementation. The integration layer should then be adapted to how this Bluetooth stack communicates. The implementation of the JSR-82 interfaces can then be targeted at this Bluetooth stack and back again, and then the solution is complete.

7.2.3 Licenses

It does not matter if the JSR-82 implementation is an internal implementation or a third party product, the JSR-82 implementation must still certify on the platform it ships on. Therefore the TCK must be bought from the specification lead, which is Motorola, and be run on the customer platform at integration time. Also a JSR-82 license must be in place from Sun to have the permission to deliver JSR-82 functionality. The TCK must be demonstrated as passed to certify JSR-82 compliance.

7.2.4 Time estimation

Moment	**Min**	**Prob.**	**Max**	**Comment**
JSR-82 implementation	30 h	60 h	120 h	N.C
Dev. integration Layer	80 h	100 h	160 h	N.C
Native integration	-	-	-	Read below

Table 1: Time estimation BT solution

The probable time spent on implementing the JSR-82 lib should be one and a half week for a person that have good knowledge about how the integration with the Integration layer should be accomplished. This person should also be experienced in KNI interface implementation. Further on this person must be familiar with the JSR-82 specification, the J2ME platform, and with the Java and the C programming language. Otherwise this must be investigated and analyzed before the actual implementation, which is a time consuming process. The maximum time spent on implementing the JSR-82 library may therefore be three weeks.

The development of the integration layer should be done after or in parallel with the development of the JSR-82 implementation. This because the functions in the integration layer are those who implement the methods defined in the JSR-82 library. This is the actual communication with the Bluetooth device, and is a little more time consuming than the JSR-82 implementation. The JSR-82 specification consists of a great amount of defined functions that should be implemented, but all of those mustn't be implemented to support most Bluetooth functionality. This fact together with the programming skills and knowledge about JSR-82 will depend how long the development of the integration layer will take. A skilled C and Java programmer that is familiar with the Bluetooth technology, the JSR-82 specification, and the JSR-82 implementation may complete this task in two and a half week. A person that is not familiar with the JSR-82 specification or the Bluetooth technology may complete this in four weeks. Good knowledge in the programming languages, particularly the C programming langue is a requirement. To get knowledge about the third party Bluetooth stack and how to communicate with it is also a part of the procedure; a person that has experience from the Bluetooth stack that is going to be used, and also the other skills mentioned above, may complete this integration layer implementation in two weeks.

The time estimation for integrating this product with the native environment depends on the customer platform and can't be estimated. This must be investigated further when the target platform is settled.

7.3 Standalone solution

7.3.1 Design, the product, scenarios

This standalone solution is a reduced version of the BT stack solution, and is aimed at customers who already have a Bluetooth stack implementation integrated with their platform (Figure 35). This customer may for example make use of the C programming language Bluetooth technology, but their J2ME platform is not Bluetooth compatible, i.e.

the JSR-82 API is not integrated and there is no connection between the KVM and the Bluetooth stack. This customer wants to have the users of their platform being able to make use of the JABWT, the executing of Bluetooth MIDlets.

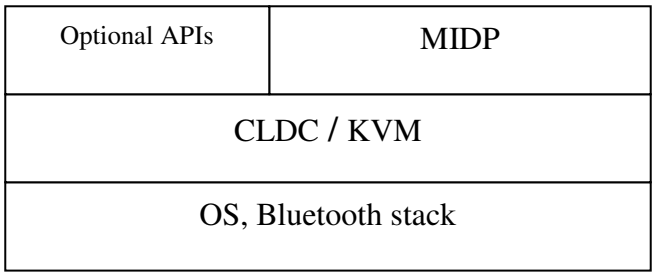

Figure 35: Standalone solution - customer platform

The components needed in this solution are almost the same as in the BT stack solution, except for the Bluetooth stack (Figure 36). This will though make quite different demands on the integration layer. Because the integration with the Bluetooth stack must be accomplished inside the customer platform, the integration layer must be as generic as possible to have it adapted to most Bluetooth stacks on the market and to ease the porting of the solution.

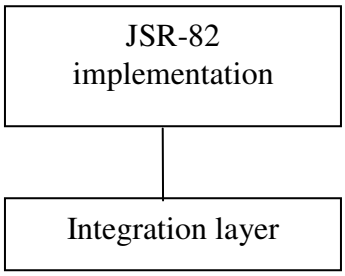

Figure 36: The Standalone solution as a product

The JSR-82 implementation should be placed on top of the MIDP layer on the customer platform, and the KNI interface should be used for communication with the integration layer. But this time, there can't be any further communication inside the actual product. The only thing the integration layer hardly does is to define the functions that are going to communicate with the stack, some kind of loop-back functions or dummy functions. The Bluetooth stack integration itself has to be done when the integration layer is in place in the native environment; by the customers themselves with their own Bluetooth stack. The integration layer is in fact more like a loop back implementation of the interfaces specified by the JSR-82 implementation, than a functional integration layer. It is also a possibility to leave the integration layer outside this solution and leave all of the Bluetooth stack integration implementation to the customer. The JSR-82 interfaces that

must be implemented may for example be specified in an enclosed document. The difference would be that the physical integration of the JSR-82 implementation with the native environment will generate compile errors until the integration layer is constructed, because the methods defined in the JSR-82 implementation requires that those methods exists in the native system. All this is illustrated in figure 37.

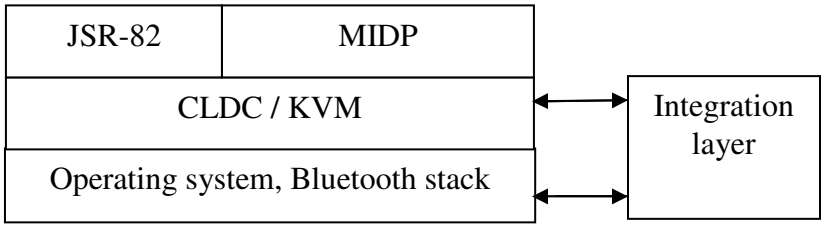

Figure 37: The solution integrated with customer platform

7.3.2 The work

Compared to the BT stack solution, this solution does not require as much work effort. In fact, the only time consuming and manpower demanding process is the development of the JSR-82 implementation. The integration layer only defines the functions that are going to be implemented in the customer environment. The integration layer may also not exist. That is to say this solution requires less work effort, but must be a less expensive product for the end customer.

7.3.3 Licenses

Because the JSR-82 implementation is the only implementation that has to be done, the licenses and TCK are the same as in the BT stack solution.

7.3.4 Time estimation

Moment	**Min**	**Prob.**	**Max**	**Comment**
JSR-82 implementation	30 h	60 h	120 h	N.C
Dev. integration layer	5 h	8 h	16 h	N.C
Native integration	4 h	6 h	10 h	N.C

Table 2: Time estimation Standalone solution

This is the same JSR-82 development as in the BT Stack solution. Read the BT Stack solution time estimation section for more information about the JSR-82 implementation.

The integration layer in this solution is a very small component. It is only some kind of loop-back functions of the methods that the JSR-82 defines. Therefore it shouldn't take more than one working day for a person that is familiar with the JSR-82 implementation to carry out this implementation. A person that is not familiar with the JSR-82 implementation must at first get that knowledge, which should be a one day work.

Also the integration of the integration layer with the native environment should be a rather easy process. It all depends on the architecture of the target system which is unknown until the target system is settled, but this is still only a procedure of having the KNI in the KVM communicating with this interface. Probably there is some kind of KVM module in the native system that executes all Java functionality and the corresponding interfaces, where after this integration layer should be placed inside that module.

7.4 Full scale solution

7.4.1 Design, the product, scenarios

The Full scale solution includes a J2ME platform, a JSR-82 implementation, an integration layer and a Bluetooth stack, and differ a little compared to the other solutions. While those are more like standalone JSR-82 products, this product is a complete JABWT solution that does not require involvement of any other components to have the target platform make use of the Java Bluetooth technology. This solution is mostly located inside Teleca's Obigo Q-line, and is the one focused on within this thesis project, which physically means that the integration of the JSR-82 implementation from Profix within this thesis project is a part of this full scale solution.

This solution is aimed at customers who wants to make use of Teleca's software framework Obigo Q-line, and also wants Java Bluetooth included (Figure 38). This solution should be placed on top of the customers existing operative system within their environment.

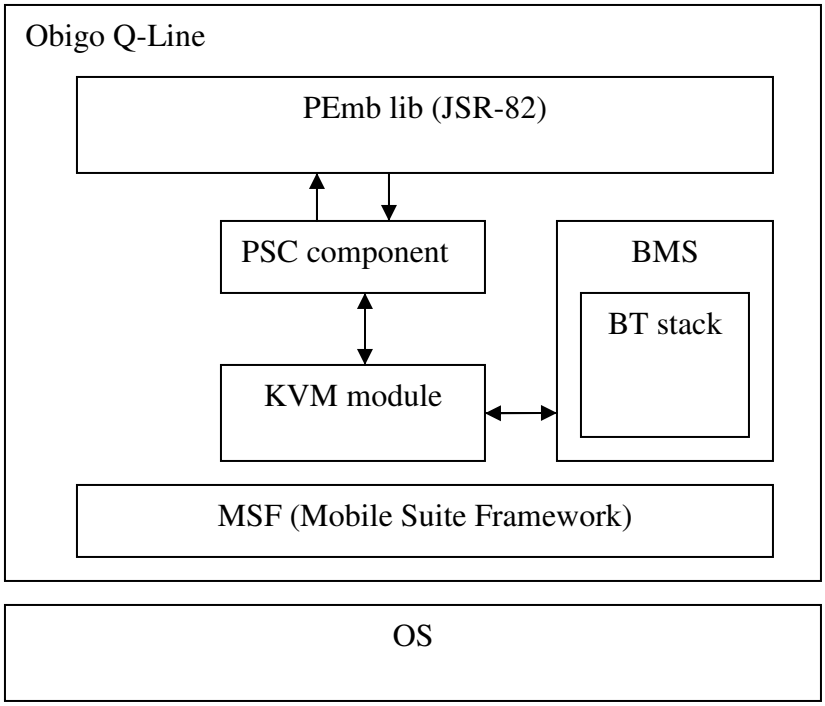

Figure 38: The Full Scale solution product placed on top of the customer platform

The PEmb lib running in a separate thread inside Obigo executes Bluetooth MIDlets and loads the JSR-82 class files. The KVM invokes the appropriate methods implemented in the PSC component through the PSI interfaces. Because of its synchronous communication the PEmb lib goes into a suspended state (That is why the PEmb runs in a separate thread, and that is why the KVM hybrid wrapper module has been created), the KVM module handles the calls, i.e. passes it in a Obigo asynchronous message passing style to the BMS (Bluetooth Manager Service), which handles the signaling to and from the Bluetooth stack. Possibly returning messages are passed back to the PSC through the proxy (transformed into synchronous calls again) and back to the PEmb lib and the KVM MIDlet execution. Figure 38 is a perspicuous picture of the PEmb integration. In fact, the PEmb and the PSC are invisible to the other Obigo components, located inside the KVM module that acts as the actual KVM.

In this solution, the KNI interface won't be used because the KVM and the interfaces to the native system are encapsulated and hidden inside the pre-compiled PEmb lib. The functions required by the PSI interfaces are implemented in the PSC component. Those PSI interfaces are probably built upon the KNI interface, but the PSI interfaces are fairly simplified and adapted to fit the implementation in the PSC components. It is still possible to construct an own JSR-82 implementation and use the KNI interface, but that should be a rather more complicated and time consuming process than using the PSI

interfaces. It is also unknown if the PEmb KVM supports the KNI interface. The drawback with the PSI interfaces is that you must use JSR implementations from Profix because it is not possible to access those interfaces in the KVM; this is a Profix product whose usage is hidden from other developers.

7.4.2 The work

Profix JSR-82 library has been integrated with the Java / Obigo version during this thesis project. Teleca has still only licensed their PEmb for demo and TCK test purpose. Developing an entire new J2ME platform including a KVM is a very time consuming procedure, and considering that the PEmb integration process already is done, the best option would be to license the end product rights from Profix.

The full scale solution is yet not completed as described in the previous sub chapter; a Bluetooth stack is missing in this version of Obigo, and also the BMS module that handles the Bluetooth stack (Figure 39). The BMS module was constructed in another thesis project at Teleca. Note that the PEmb integration was done in a subset of the Obigo suite, which means that the integration may not work as intended if other modules are added. It is therefore unknown what kind of bugs (if any) that will appear when the BMS module is introduced.

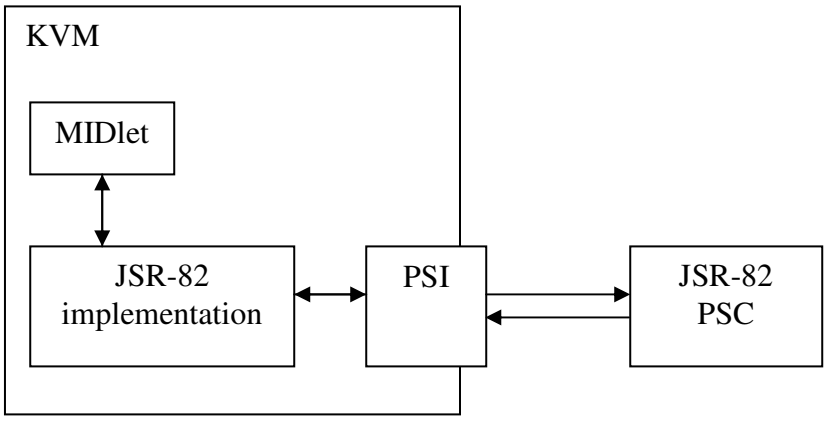

Figure 39: The Java / Obigo / JSR-82 integration today, with a missing BT stack and BMS

The implementation part of this thesis project has therefore been aimed towards a Bluetooth software simulator instead of an actual Bluetooth stack. This means that the JSR-82 PSC component that has been constructed, only implements the communication between the PEmb lib and the PSC. A Bluetooth stack must be integrated with this version of Obigo together with the BMS. Then the JSR-82 PSC implementation can be completed and the Obigo Q-Line will have full Java Bluetooth support.

7.4.3 Licenses

This JSR-82 solution brings the most complicated case when speaking about licenses and TCKs. For every new platform this complete J2ME solution shall be integrated with, new MIDP2 compatibility tests must run in the native environment to have it Java MIDP2 certified on that platform. But also a specific TCK must be obtained from Motorola for the JSR-82 optional API. Those must also be presented as passed on every new platform this solution integrates with. Also a JSR-82 license must be in place from Sun to claim JSR-82 compliance.

Also the missing Bluetooth stack problem must be considered. What third party stack is best suited for this integration? Earlier the Ericsson Bluetooth stack has been used, so an integration of that one would simplify this procedure. But as it seems right now Ericsson has introduced new restrictions about the usage of their Bluetooth stack, so maybe Teleca won't be able to use it in a final product. If Ericsson's stack is going to be used, the stack integration process won't be a problematic or time consuming process, otherwise it will. This lies however as a higher level decision in the Teleca Company structure and is outside the scope of this thesis project.

7.4.4 Time estimation

Moment	Min	Prob.	Max	Comment
BT stack integration (Ericsson)	16 h	24 h	40 h	N.C
BT stack integration (other)	40 h	80 h	160 h	N.C
PSC implementation	24 h	40 h	80 h	N.C
Native integration	-	-	-	Read below

Table 3: Time estimation Full Scale solution

The integration of Eriksson's Bluetooth stack with Obigo has been done once before at Teleca, which took about three days to complete. That person who is familiar with this integration may complete this again in three days, but in best case he knows about the problems that came up last time, so the time spent on this integration can be reduced down to two days. Also in the integration of the Bluetooth stack the BMS integration is included. This may cause trouble, because the PEmb is integrated on a stripped version of

Obigo which may not work correctly if other modules are introduced. If this problem occurs, this procedure of having Java / Obigo work with the BMS module may be a one week work.

If another Bluetooth stack is going to be integrated instead of Eriksson's stack, the integration will be more time consuming. At first knowledge about the stack must be acquired, and then the integration must be designed. The existing BMS is constructed to be stack-independent, but there is still unknown if it is going to work with another stack; this hasn't been tested at Teleca. However, if the stack has the same structure as Eriksson's stack, and the BMS module works fine with this stack, the integration may only take one week. The most probable is still that the stack differs from Eriksson's stack, and therefore the integration will be a little different. This will set the probable integration time to two weeks. But if the BMA module won't integrate with this new Bluetooth stack, the BMA must be modified or a new must be developed. Experience about this kind of development exists at Teleca, but this will still be a time consuming work. If the BMA must be completely rewritten the time for the integration will be about four weeks.

The PSC implementation is the procedure of implementing the communication to and from the Bluetooth stack. This must be done after the stack is integrated, and shouldn't be too time consuming. The PSC should communicate with the Bluetooth stack through the BMS module, where the stack calls already are written. The PSC functions should call the appropriate functions inside the BMS module to have the communication with the Bluetooth stack work correctly. In best case this may take only three days for a person that is familiar with Obigo, Bluetooth and the JSR-82 implementation, to carry out, but the probable time for this implementation is one week. The worst case scenario is the one where the PSC component won't be able to access the BMS functions, and there must be another kind of layer between the PSC and the BMS.

The time estimation for integrating this product with the native environment depends on the customer platform and can't be estimated. This must be investigated further when the target platform is settled.

8 JSR-82 integration

Besides the design and JSR-82 solutions work, the integration of Profix JSR-82 implementation has also been a part of this project. Profix JSR-82 library was delivered as pre-compiled source code, together with the PEmb library. The actual JSR-82 integration was performed at Visual studio source code / structure level, where the JSR-

82 implementation together with the PEmb library was integrated inside the hybrid wrapper module inside Obigo. This entailed Visual studio generates new errors at compilation time; as a result of the JSR-82 implementation introduction with the PEmb library inside the hybrid wrapper module; PEmb now required those JSR-82 PSI interfaces implemented inside the PSC component. To have the JSR-82 integration build successfully, some loop-back interfaces was implemented.

8.1 Hardware and software

The software used for the implementation of the PSI interfaces is Visual Studio 6.0 and the Obigo SDK. The MIDlet applications used has been developed using JEdit and Suns wireless toolkit WTK2.2. Btone Simulator is used for testing the Bluetooth functionality of the MIDlets. Other software used during this project is Cygwin, GnuMake, Viso 2003, Nokia Developer suite and the PEmb Emulator.

8.2 Implementation

The actual implementation is a small part of this project and has been related to the integration the JSR-82 library with Obigo. At first some loop-back interfaces had to be implemented to have the integration builds without any errors. The next step was to implement the actual PSI interfaces. Because no Bluetooth stack was integrated in this Obigo version, the PSI interfaces couldn't be fully developed; only the communication between the PEmb library and the PSC could be implemented.

The PSI interfaces were implemented inside the PSC component inside the Obigo KVM module.

9 Future work

Because no Bluetooth stack is integrated with the Java / Obigo version, such integration has to be done to have a complete Obigo JABWT solution. This integration has been done once at Teleca, but with another Obigo version. See the time estimation part in the "Full scale solution" section for more information of the Bluetooth integration. The Bluetooth stack integration in this project has always been aimed at the BMS module. But because the BMS high level of abstraction, such integration may have the PSC component seem unnecessary; it may only pass the calls from the PEmb forward to the BMS. Another way, and maybe a better one, would be to have the PSC component communicate directly with the Bluetooth stack.

10 Summary and conclusion

When speaking in terms of working this project was divided into two main parts independent of each other, but still very connected as an end product; the theory part is the description and presentation of what the integration part can look like in the end. The description of the Teleca JSR-82 solutions has been the biggest part of the project because of different delay issues during the work. The integration and implementation part is actually a small sub-section of this project, but such an interesting one.

As the result of this project, three possible Teleca JSR-82 solutions has been designed, investigated, and presented. Those are three different products that can co-exist in some kind of product family, where the customers have the opportunity to choose the product that fits their own needs.

The solutions differs in both functionality, size, and developing cost, but covers most of today customer needs when speaking of Java Bluetooth functionality on a wearable platform.

An integration of the JSR-82 library with Obigo has also been completed. The integration contains a half-way implementation of some of the PSI interfaces, and extends the functionality of the Obigo even more. This is still just prototype integration, but it is a great beginning of what could be a complete Teleca JSR-82 solution as presented in the other part of this project.

11 References

1. Hopkins Bruce; Ranjith Anthony, *Bluetooth for Java,* 2003, Springer-Verlag, ISBN 1-59059-078-3

2. Knudsen Jonathan, *Wireless Java: Developing with J2ME, second edition*, 2003, Springer-Verlag, ISBN 1-59059-077-5

3. Kroll Michael; Haustein Stefan, *J2ME Application Development*, 2004, Sams Publishing, ISBN 0-672-32095- 9

4. Mahmoud Qusay, *Learning Wireless Java,* 2001, O'Reilly, ISBN 0-596-00243-2

5. Magnus Söder , *Integrating Profix KVM with Obigo SDK, 2003*

6. Profix, *Micro Developers Guide: Using Device Emulator for Windows,* 2004

7. Burström David; Ohlson Mikael, *Integration of a Kilobyte Virtual Machine with the Teleca Obigo Framework*, 2004

8. Hagberg Jonas, Hast Marcus, *Bluetooth for Mobile Phones*, 2004

9. Teleca, *Obigo Developer's Manual*[1], 2003

10. Teleca, *Obigo Framework Development*[1], 2003

11. Teleca, *Obigo SDK User's manual*[1], 2003

12. Teleca, *Q04C1 Design Specification PEmb KVM Integration*[1], 2004

13. Profix, *Interface Reference Java APIs for Bluetooth Wireless Technology*[1], 2004

14. Profix, *Interface Reference CLDC 1.0/1.1—MIDP 2.0*[1], 2004

15. Textone, *Btone Technology Licensing Kit User Guide*[1], 2003

[1] Internal document at Teleca Software Solutions, Lund

16. Textone, *Btone Technology Licensing Kit – Ericsson Stack Mapping*[1], 2003

17. Textone, *Btone Technology Licensing Kit – Implementation Specific Components*[1], 2003

18. Textone, *Btone Technology Licensing Kit – NEC Porting Guide*[1], 2003

19. Textone, *Btone Simulator User Guide*[1], 2002

20. Motorola Wireless Software, *Java APIs for Bluetooth Wireless Technology (JSR-82),* 2002

21. Sony Ericsson, *Developing Applications with the Java APIs for Bluetooth (JSR-82),* 2004

22. Textone, *Btone Bluetooth Technology Licensing Kit Technical Overwiev*[1]

23. Textone, *Btone White Paper*[1], 2001

24. Karl McCabe, *Btone Technical Overwiev*[1]*, Business Model, Porting Analysis*

25. http://java.sun.com/j2me/overview.html , 2004-09-20, Sun Microsystems

26. http://developers.sun.com/techtopics/mobility/midp/ttips/BTintro/index.html , 2004-09-21, Sun Microsystems

27. http://www.microjava.com/developer/faq/compile_run_debug , 2004-11-14, Microdevnet

28. http://java.com/en/dukeszone/index.jsp , 2004-11- 18. Sun Microsystems

29. http://today.java.net/pub/a/today/2004/07/27/bluetooth.html , 2004-11-18, Sun Microsystems

30. http://www.jcp.org/en/press/success/bluetooth , 2005-01-18, Java Community Process

Appendix A – Abbreviations

(JSR) Java specification Request
(JCP) Java Community Process
(J2ME) Java 2 Micro Edition
(API) Application Programming Interface
(KVM) Kilo Virtual Machine
(KNI) K Native Interface
(JNI) Java Native Interface
(BT) Bluetooth
(CLDC) Connected Limited Device Configuration
(MIDP) MIDlet Information Device Profile
(BMS) Bluetooth Manager Service
(CMS) Content Manager Service
(GUI) Graphical User Interface
(BCC) Bluetooth Control Centre
(SDK) Software Development Kit
(HCI) Host Controller Interface
(L2CAP) Logical Link and Adaptation Protocol
(SDP) Service Discovery Protocol
(SPP) Serial Port Profile
(GOEP) Generic Object Exchange Profile
(UUID) Universal Unique Identifier
(URL) Uniform Resource Locators
(MSF) Mobile Suite Framework
(OS) Operating System
(AMS) Application Management Software
(CMA) Content Manager Application
(RI) Reference Implementation
(HDI) Host Device Integration
(SMF) State Machine Framework
(RACS) Textone Abstract C Stack
(SIG) Special Interest Group
(PSI) PEmb Service Interface
(PSC) PEmb Service Component

Appendix B – JSR-82 Methods

javax.bluetooth.DiscoveryListener

- public void deviceDiscovered(RemoteDevice, DeviceClass)
- public void inquiryCompleted(int)
- public void servicesDiscovered(int, ServiceRecord[])
- public void serviceSearchCompleted(int, int)

javax.bluetooth.L2CAPConnection

- public int getReceiveMTU()
- public int getTransmitMTU()
- public boolean ready()
- public int receive(byte[])
- public void send(byte[])

javax.bluetooth.L2CAPConnectionNotifier

- public L2CAPConnection acceptAndOpen()
  throws IOException

javax.bluetooth.ServiceRecord

- public int getAttributeIDs()
- public DataElement getAttributeValue(int)
- public String getConnectionURL(int, boolean)
- public RemoteDevice getHostDevice()
- public boolean populateRecord(int[])
- public boolean setAttributeValue(int, DataElement)

javax.bluetooth.DataElement

- public void addElement(DataElement)
- public boolean getBoolean()
- public int getDataType()
- public long getLong()

javax.bluetooth.DeviceClass

- public int getMajorDeviceClass()
- public int getMinorDeviceClass()
- public int getServiceClasses()

javax.bluetooth.DiscoveryAgent

- public boolean cancelInquiry(DiscoveryListener)

- public boolean cancelServiceSearch(int)
- public RemoteDevice retrieveDevices(int)
- public int searchServices(int[], UUID[], RemoteDevice, DiscoveryListener)
- public String selectService(UUID, int, boolean)
- public boolean startInquiry(int, DiscoveryListener)

javax.bluetooth.LocalDevice

- public String getBluetoothAddress()
- public DeviceClass getDeviceClass()
- public int getDiscoverable()
- public DiscoveryAgent getDiscoveryAgent()
- public String getFriendlyName()
- public static LocalDevice getLocalDevice()
- public static String getProperty(String)
- public ServiceRecord getRecord(Connection)
- public boolean setDiscoverable(int)
- public void updateRecord(ServiceRecord)

javax.bluetooth.RemoteDevice

- public boolean authenticate()
- public boolean authorize(Connection)
- public boolean encrypt(Connection, boolean)
- public boolean equals(Object)
- public final String getBluetoothAddress()
- public String getFriendlyName(boolean)
- public static RemoteDevice getRemoteDevice(Connection)
- public int hashCode()
- public boolean isAuthenticated()
- public boolean isAuthorized(Connection)
- public boolean isEncrypted()
- public boolean isTrustedDevice()

javax.bluetooth.UUID

- public boolean equals(Object)
- public int hashCode()
- public String toString()