Open Broadcasting Software Stack and Applications for Mobile Devices

by

Jean-Michel Bouffard

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of

Master of Applied Science

Ottawa-Carleton Institute for Electrical and Computer Engineering

Faculty of Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada

June 2009

The undersigned hereby recommend to the Faculty of Graduate Studies and Research acceptance of the thesis

Open Broadcasting Software Stack and Applications for Mobile Devices

Submitted by

Jean-Michel Bouffard, B.A.Sc.

in partial fulfillment of the requirements for the degree of M.A.Sc. in Electrical Engineering

---

Thesis Supervisor

Professor Thomas Kunz

Chair, Department of Systems and Computer Engineering

Professor Victor Aitken

Carleton University

June 2009

Abstract

New digital broadcasting technologies, such as DMB and DVB-H were developed recently, but their deployment is happening very slowly. It is believed that a lack of innovation in the available applications is one of the main reasons for this slow acceptance. The goal of the Openmokast research project described in this thesis is the implementation of a mobile broadcasting software stack targeting the emergent open mobile devices.

The open devices ecosystem was surveyed and the missing components were identified. Useful projects from the open source community were integrated, together with custom code, to provide a complete middleware software stack. The integration of the stack in the Openmokast prototype created the first open mobile broadcasting device to date. A key contribution of this work is an API that enables third party development over the broadcasting middleware. This work sets the ground for future innovation in the field of multimedia broadcasting.

Acknowledgments

This research project was funded and executed as part of the Mobile Multimedia Broadcasting project conducted under the Broadcast Technology branch of the Communications Research Centre (CRC) in Ottawa. The CRC provided the context, the equipment and the resources necessary for the success of the project.

I'd like to thank the following people for their contribution, help and support during the course of this project:

François Lefebvre for coming up with the vision of open broadcasting handhelds which became the starting point of this research project. François initiated and leads the MMB Openmokast project at CRC under which this research took place. His leadership and guidance were essential in defining the scope of the project and during the whole R&D process. His interest for broadcasting and his clear view of the technical issues at stake were the origin of many interesting discussions that took place during the course of this effort.

Thomas Kunz for his supervision of the thesis work, during which his knowledge and guidance were much appreciated. His challenging questions about the concepts elaborated in the project helped me clarify my ideas while writing the document.

Pascal Charest for his help in resolving many technical issues encountered. Pascal contributed to the successful implementation of a functional Linux driver for the USB receiver used in the Openmokast device. He was also involved in the development of the CRC-DABRMS software that was used as a core component of the Openmokast framework.

Martin Quenneville and Stephen Montero for their help with the hardware integration of the device. Martin contributed to the connection of the USB receiver inside the prototype. Stephen worked at designing the extension with the Pro/E software.

Valérie Noël for her patience, comprehension and support during the long hours spent on this project.

Contents

List of Tables

List of Figures

List of Acronyms

API	Application Programming Interface
COM	Component Object Model
CRC	Communications Research Centre
DAB	Digital Audio Broadcasting
DCSR	DAB Command Set for Receiver
DLS	Dynamic Label Segment
DMB	Digital Multimedia Broadcasting
DRDI	Digital Radio Data Interface
DRM	Digital Radio Mondial
DVB-H	Digital Video Broadcasting - Handheld
EPG	Electronic Program Guide
FEC	Forward Error Correction
FIC	Fast Information Channel
FPS	Frames Per Second
FSO	Freesmartphone.org
GDB	GNU Debugger

GPL	General Public License
IDE	Integrated Development Environment
IPC	Inter-Process Communication
IRC	Internet Relay Chat
ISPs	Internet Service Providers
JML	Journaline Markup Language
LiMo	Linux Mobile
LiPS	Linux Phone Standards
MCI	Multiplex Configuration Information
MMB	Mobile Multimedia Broadcasting
MOT	Multimedia Object Transfer
MOT-BWS	MOT Broadcast Website
MOT-SS	MOT Slideshow
MPEG-TS	MPEG Transport Stream
MSC	Main Service Channel
OFDM	Orthogonal Frequency-Division Multiplexing
OHA	Open Handset Alliance
PCM	Pulse-Code Modulation
PDAs	Personal Digital Assistants
PIM	Personal Information Management
RSCI	Receiver Status and Control Interface
RTP	Real-time Transport Protocol

SDIO	Secure Digital Input Output
SDK	Software Development Kit
TDC	Transparent Data Channel
UMTS	Universal Mobile Telecommunications System
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
USRP	Universal Software Radio Peripheral
VM	Virtual Machine
XML	Extensible Markup Language

Part I

Preface

Chapter 1

Introduction

"Today's modern handset represents a 'melting pot' of communications and multimedia technologies."

– *R. Wietfeldt, Texas Instruments.*

THE introductory quotation represents very well the state of the current mobile market where devices such as Personal Digital Assistants (PDAs), cell phones and mp3 players are becoming ubiquitous. The mobile nature of these appliances provides a strong incentive to merge as many functionalities as possible into well-integrated devices [1]. However, in today's context, mobile telecommunications organizations have gained control over most integrated mobile platforms that include voice communication capabilities, which easily represents the major part of this market. It becomes thus quite challenging for industries that are not related to telecommunications, such as broadcasting, to include new services on these devices based on their own specific sets of technologies and software [2].

The open source movement is seen as an attractive strategy to provide the huge software development efforts required for highly integrated mobile devices [3, 4]. Following this approach, multiple multifunction open hardware platforms, driven by collaboratively developed open source software frameworks and applications, have made their appearance in the last years. Openmoko [5], Qtopia [6] and Google's Android [7] are the best-known examples. However, no such open and collaborative effort has been seen so far in the broadcasting community.

Considering the growing enthusiasm for digital mobile broadcasting technologies, which include DVB-H [8], DAB/DMB [9], MediaFLO and others [10, 11, 12], broadcasters could benefit greatly if corresponding chipsets would find their way into such open mobile communication platforms. This could unleash an ensemble of new services that would take advantage of the unique capabilities of broadcasting infrastructures. It is believed that the availability of the required broadcasting-enabled software stack for those open platforms could become the enabler that would encourage chipset manufacturers to participate in these initiatives.

This research work focuses on studying the specific requirements for open source mobile multimedia broadcasting (MMB) applications and on identifying, adapting or developing corresponding software building blocks with the goal of integrating them as part of one of the previously mentioned popular open frameworks. To begin with, a survey was conducted about the current ecosystem for openness in the world of mobility, including the related standardization efforts. Next, the main contribution of this work is the first software stack that controls a set of different digital broadcasting receivers, integrates the mechanisms to manage the available channels and multimedia stream, and provides some application decoders for different broadcast services. Moreover, a functional prototype, used together with a compact external receiver module, will be used to demonstrate real-time mobile multimedia broadcasting applications. In addition, the platform is also suitable for validation and performance evaluation of the services.

The resulting broadcasting software stack was also demonstrated with different client applications that access the broadcast data through a newly defined API (Application Programming Interface). This API has the features to become the key component required to enable the framework to be used easily by other software developers interested in building new applications based on a broadcasting technology or to integrate it in an already existing application. Ultimately, this mobile broadcasting platform could become the enabling software that will convince broadcasting hardware manufacturer to enter the market of open mobile devices.

The context for the work described in this document and some respective results were presented in the following publications:

- [13] F. Lefebvre, J.-M. Bouffard, and P. Charest, ”Open source handhelds - a broadcaster-led innovation for BTH services”, published in the EBU Technical Review, edition 2008-Q4.
- [14] F. Lefebvre, J.-M. Bouffard, and P. Charest, ”Open mobile broadcasting phones”, published in the Proceedings of the Broadcast Asia 2008 conference, held June 17-20 2008, Singapore.

The project was also presented at different events[1]:

- ”Mobile Digital Broadcasting - Democratizing Innovation”, presented at the Emerging Communications Conference in San Francisco, March 3-5 2009.
- ”Openmokast: The open broadcasting software stack for mobile devices”, presented at the Free/Open Source Mobile Development Conference in Waterloo, Canada, February 4th 2009.
- ”Open Mobile Broadcasting Phones”, presented at the Broadcast Asia 2008 conference in Singapore, June 17-20 2008.

[1] The slides of the different presentations are available on Slideshare at `http://www.slideshare.net/tag/crcmmb`.

The project described in this document was named Openmokast, which is an acronym for **open mo**bile ”broad**kast**ing”. More information about the project, the documentation and the available source code can be found on the Openmokast website [15].

The thesis is structured in three main parts. The first part contains Chapters 1 to 4. Chapter 2 is introducing the basic background information related to the project. Chapter 3 is a detailed overview of the state-of-the-art in mobile embedded platforms that were considered for the realization of the project. Chapter 4 explains the main advantages of broadcasting technologies and why their deployment would be beneficial for users and service providers. The next section, containing Chapters 5 to 7, is the core technical content of the thesis. Chapter 5 is an analysis of all the open source components that are already available to support the development of the software stack of this project. Chapter 6 describes the development effort that led to the Openmokast software and prototype. Chapter 7 is a case study of a new application based on the Openmokast stack and API. Finally, the last section contains the ideas for future work in Chapters 8 and the discussions and conclusions in Chapter 9.

Chapter 2

Background Information

”Once GNU is written, everyone will be able to obtain
good system software free, just like air.”
– *Richard Stallman.*

2.1 The Open Source Paradigm

THE open source software[1] concept first appeared sometime in the 1980s. It started under the form of the GNU Project [16], a set of tools required to build a functional operating system environment. This project was published under a newly created license, the General Public License (GPL) [17], which promotes complete freedom of usage, modification and redistribution of computer software. It was never really well-known before the beginning of the 1990s, when it was used in collaboration with a newly developed computer kernel known as Linux. This resulted in the creation of the GNU/Linux operating system which is one of the most renowned free software projects to date. If GNU/Linux has already gained a large amount of support in the

[1] Also often referenced as ”free software”. This naming convention may sometime be confusing as it relates to the notion of freedom, not that it is distributed at no cost.

field of server and desktop computing, it is completely the opposite in mobile and embedded computing. As it was stated by Doc Searls in the Linux Journal [18] in 2007:

> Yet the world needs open phones. In fact, I'd hazard a prophesy that open phones are inevitable, because there will be far more money to be made because of open phones than will ever be made with closed ones (and closed services offered only by carriers). We're starting to see vertical cracks in the closed wall of mobile telephony in settings such as universities, where rogue companies like Rave Wireless provide students with custom (based on open) phones that run on familiar networks (such as Cingular and T-Mobile), but that do far more than the closed phones sold at stores by those same networks. Users are even free to do their own programming, create and add their own features and services. With each crack of this kind in a vertical market, the chance improves that open phones will become the norm rather than the exception.

The open source movement is traditionally considered as a software concept.

However, recently, some hardware manufacturer have adopted a similar open approach in designing and implementing their devices. In this case, the documentation and the schematics of a hardware system are published with a license that permits modification and re-publication of the modified design. A free and open development process is also often associated with a community working around the project. This is made possible because all the required elements needed to submit contributions to the project are freely available, no matter if they are related to hardware, software or both.

2.2 Mobile Multimedia Broadcasting

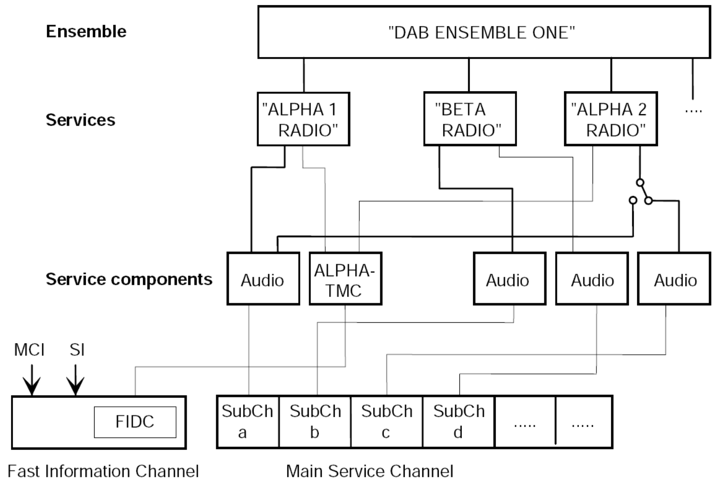
2.2.1 Technologies

Mobile digital broadcasting can be defined as a unidirectional communication channel that is able to transport digital audio, video and data. The main technologies[2] in that field are known as DAB/DMB [19] and DVB-H [20] but new emerging standards are also being defined regularly. The common features provided by these digital networks are their **high capacity downlink channels** and their **capability to transport any types of data**.

Both DAB/DMB and DVD-H are taking advantage of Orthogonal Frequency-Division Multiplexing (OFDM) digital multi-carrier modulation method to efficiently transport the signal to mobile receivers. The capacity of a single transmission, usually called a multiplex or an ensemble, is much higher than a single audio or video service. For this reason, any DAB/DMB or DVB-H multiplex can be split into different subchannels to transport multiple services. The configuration of a multiplex is completely dynamic as it is possible to combine different amounts of low to high quality subchannels until all the available capacity is used. The main differences between both standards are in the used frequency, the channel bandwidth and the transport mechanisms as described in [21].

A sample structure of a DAB ensemble, as defined in [22], is depicted in Fig. 2.1. The low level elements of the structure are the subchannels. Each subchannel represents a part of the physical capacity of the data stream of the ensemble. The next layer contains the components. The components are the logical elements that represent an actual subchannel from the physical layer. The components are then grouped into different services in the upper layer. Each of the services is composed of

[2]Other technologies such as HDRadio and FLO are currently gaining a lot of exposure, however, they are not mentioned in this report because of their proprietary nature.



Source: [22]

Figure 2.1: DAB service structure example

at least a primary component, but it can also be composed of a secondary and even of some extended components. For instance, a radio service could have a primary musical audio component, a secondary news audio component and some extended components with textual information. Finally, all these services are grouped together inside an ensemble that can then be modulated and transmitted over the air. In the transmitted stream, the set of subchannels contained in the ensemble are transported in the Main Service Channel (MSC). The information about the structure of the multiplex, which uses only a fraction of the bandwidth of the channel, is transported into the Fast Information Channel (FIC). The FIC is transmitted in a redundant loop that enables the receivers to rapidly receive the information about the structure of the ensemble. These substreams, the MSC and the FIC, are the two main parts of a transmitted DAB ensemble.

As of the writing of this thesis, DAB is the standard terrestrial digital broadcasting technology accepted in Canada, as it was decided at the end of the 1990s. However, the first transmitters deployed in Canada's largest cities made only a discreet appearance and they never became popular. The available services are composed of a retransmission of the main FM stations of each region. It is believed that the main cause of the failure of the DAB in Canada was mainly the lack of receiver availability and the lack of unique content available only on the digital network.

2.2.2 Services Capabilities

The main services that are targeted by mobile digital broadcasting technologies are digital radio (audio) and mobile TV (audio/video). But the mechanism are included to perform many different types of transports and applications as explained in [23]. For example, the DAB/DMB specification supports the following features:

Dynamic Label Segment (DLS): A way to transport text information associated with the currently playing service. This mechanism can be used to show text information about the content or be unrelated to the content such as local news.

IP datagram tunnelling: Transport mechanism for IP packets. The packets are grabbed at the service encoders and they are encapsulated inside a data subchannel. These data are then decapsulated at the receiver side and retransmitted into an IP stream.

Multimedia Object Transfer (MOT): Transport of multimedia objects such as images, text, HTML pages, sound files and many other. This mechanism is the standard way of transporting MOT SlideShows (MOT-SS) and MOT Broadcast Websites (MOT-BWS). MOT-SS is implemented by sending pictures one after the other over the MOT subchannel. MOT-BWS uses a file caching mechanism to transport a carousel of files, which contains the website structure, to the receiver. The MOT standard has been defined for a broadcast environment where packet loss and transmission errors can occur. For this reason, some redundancy can be used to lower the effects of these errors.

Transparent Data Channel (TDC): This protocol can be used to transparently transmit data to the receiver. The flexible protocol has no idea of the content that is transported, hence it is the ideal mechanism to extend the functionality of DAB with new services.

Electronic Program Guide (EPG): The EPG corresponds to a listing of the content that will play on the audio and video services.

DAB Audio: The first defined standard to transport audio. It is using the commonly known MPEG layer 2 audio codec.

DAB+ Audio: This standard was defined to replace the legacy DAB Audio service with a more robust and bandwidth-efficient mechanism. The AACv2 codec is used in addition to a new layer of FEC (Forward Error Correction) to attain this goal.

DMB Video: This standard was defined for mobile television over DAB. The process uses the AVC video codec with added layer of FEC.

If digital broadcasting was widely deployed, the capabilities described in this section could enable many innovative and useful applications to extend the traditional audio and video services.

2.2.3 Receivers

Some receivers are currently available on the market as shown in the examples from Fig. 2.2. The most common type is the standalone broadcasting receiver, which is only able to receive broadcasting signals. However, in some rare markets around the world, such as in Germany and in Korea, receivers that support both telephony and a broadcasting technology were released. The common characteristics that are shared

iRiver B20
Standalone broadcasting handset

Samsung SGH-P900
Combined
telephony/broadcasting handset
Source: [19]

Figure 2.2: Example of DAB/DMB devices

by all currently available devices are their closed nature, the impossibility to modify them and the fact they usually implement only a limited amount of the services that were described previously.

Chapter 3

Open Embedded Platforms State-of-the-Art

"If you can't open it, you don't own it."

– *Openmoko.*

THIS section presents a review of the latest development in the field of open embedded platforms. A lot of work has already been done to provide the required tools to build and deploy the next mobile technology. This work can be categorized in two main categories: standardization efforts [24, 25, 26, 27] and actual mobile platform projects [5, 6, 7]. The main players in each of these categories are presented in the next subsections, in addition to their main goals and their current status. A survey of these technologies can be found in [28], but the following section focuses on the openness of the solution and its possibilities for extension of the platform.

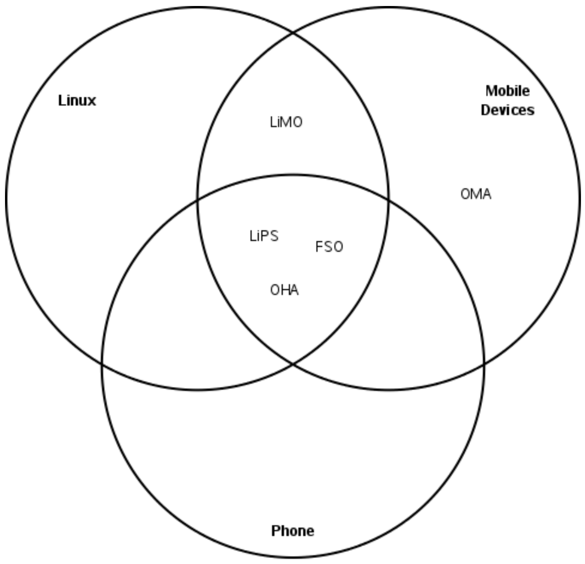


Figure 3.1: Classification of mobile devices ecosystem standards.

Organization	Web address
Linux Phone Standards (LiPS) Forum	`http://www.lipsforum.org/`
LiMo (Linux Mobile) Foundation	`http://www.limofoundation.org/`
Open Handset Alliance (OHA)	`http://www.openhandsetalliance.com/`
Freesmartphone.org (FSO)	`http://www.freesmartphone.org/`

Table 3.1: Main open mobile devices standards organizations

3.1 Standardization Efforts

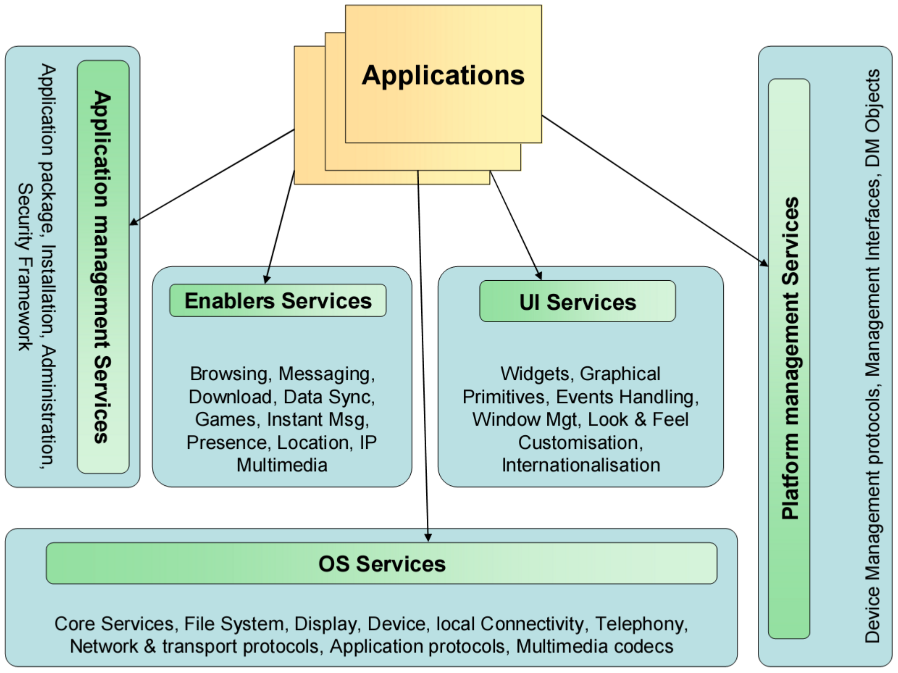
The standardization efforts are driven by consortia of corporations that share an interest in defining a common platform structure that will be used throughout the industry. The main issues that are resolved by these organization are the definition of the services that should be available on the platforms in combination with the APIs that are required to access those services. The services are composed of a list of components that are often available on mobile devices such as telephony, networking, multimedia, etc. Each of the standards has its own specific way of representing these services. The result is that the applications developed for one device that is compliant to one of these standard should normally work on another compliant device. The main groups are listed in Table 3.1 and are classified regarding their main focus in Fig. 3.1.

3.1.1 Linux Phone Standards (LiPS) Forum

The LiPS forum was founded in November 2005 to promote the development and deployment of Linux-based mobile phone devices. The forum is composed of about 24 members from different sectors of the telecommunication industry, for instance, network operators, equipment manufacturers, and software vendors.

The LiPS forum has published the first draft version of their specifications at the end of 2007. The specification is publicly available on their website as an archive of multiple specification files. An overview of the scope of the specification is available in the reference model document [29] and a high level diagram of the service sets is included in Fig. 3.2. The specification describes the interface to support the services, which are divided into five main groups, also called "Service Sets":

Application Management (AM) Services. The AM services manage the applications that are running on the platform. They take care of everything that is


Applications
Application management Services
Application package, Installation, Administration, Security Framework
Enablers Services
Browsing, Messaging, Download, Data Sync, Games, Instant Msg, Presence, Location, IP Multimedia
UI Services
Widgets, Graphical Primitives, Events Handling, Window Mgt, Look & Feel Customisation, Internationalisation
Platform management Services
Device Management protocols, Management Interfaces, DM Objects
OS Services
Core Services, File System, Display, Device, local Connectivity, Telephony, Network & transport protocols, Application protocols, Multimedia codecs

Source: [29]

Figure 3.2: LiPS high-level overview

needed during the life of an application such as downloading, installing, launching, suspending, resuming, terminating, removing and updating. This group is also responsible to implement the access control for the applications.

User Interface (UI) Services. The UI services are used with applications that need to interact with the user. They manage the visual elements on the screen in addition to the input and output events with the user.

Enabler Services. The enabler services consist of a collection of functionalities that are thought to be useful in a mobile communication device. A call manager, a messaging manager and an instant messaging engine are some examples of the included services.

OS Services. OS services are a lower-level API that interacts directly with the OS capabilities. The basic features provided by the Linux kernel can, indeed, be offered by this service category. Some examples of these OS services are multimedia codecs, 2G and 3G telephony, local connectivity protocols such as Bluetooth or IrDA, data-base management, etc. The LiPS forum does not redefine any features that are already covered by other standardization efforts such as POSIX.

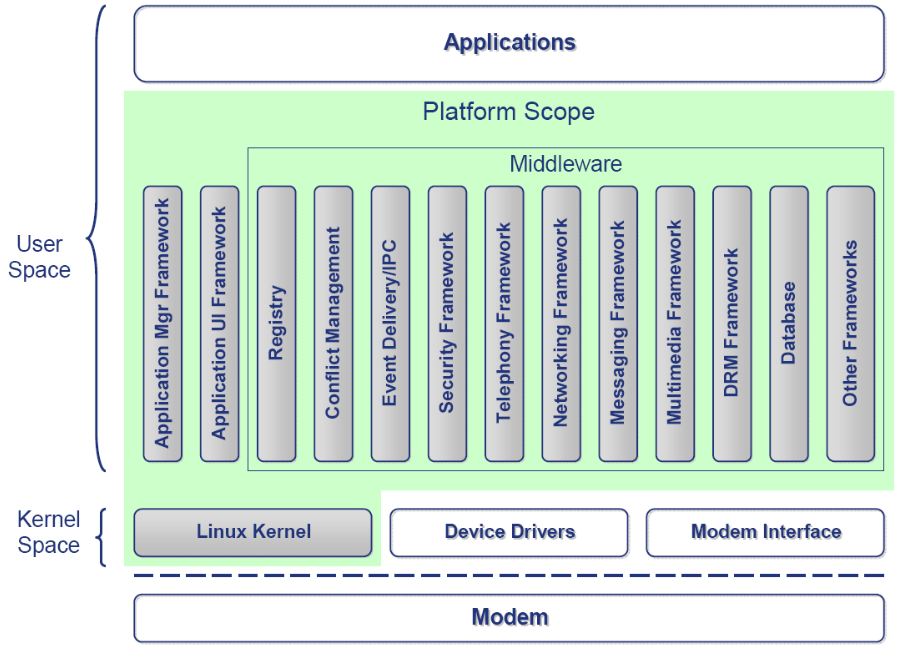
Platform Management Services. The platform management services cover the platform configuration and maintenance.

Each of these sets defines a collection of interfaces that are required to use the specified services. The other aspect of the LiPS specification is in the programming model that must be followed by each interface. In fact, these guidelines assure that the different workgroups that are set up to create the specifications will produce compatible components.

3.1.2 LiMo (Linux Mobile) Foundation

The LiMo Foundation was founded in January 2007 with the mandate to standardize a modular plugin-based platform based on an open operating system for the mobile environment. The goal of this work is to encourage and to accelerate the development and the deployment of such open mobile platforms. The work that will lead to this goal will go through the creation of a rich ecosystem composed of different products, applications and services provided by the different LiMo members.

The main difference between the LiMo Foundation and the LiPS Forum described in Subsection 3.1.1 is in the implementation. In fact, a common Linux implementation will be produced as part of the LiMo work, while only a specification is expected from the LiPS group. The system architecture that was produced as the first version of the platform is shown in Fig. 3.3. Each component is described in the LiMo Platform



Source: [30]

Figure 3.3: LiMo architecture overview

Architecture document [30]. The detailed functionalities are exposed in a set API documents that can be summarized as follows:

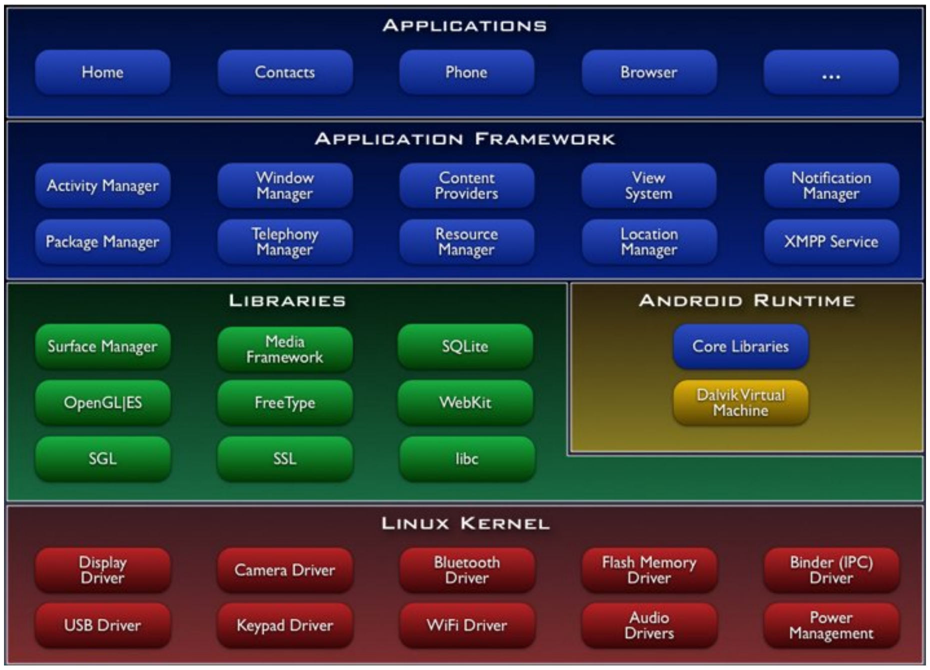
Applications. The applications are the main components that interact with the users on one side and that use the services provided by the framework on the other side. The developers have the freedom to use any available APIs to create new original and innovative applications.

Application Manager Framework and Application UI Framework. The Application Manager Framework and the Application UI Framework are the most important parts of the LiMo platform. The first element is responsible for managing the applications that are installed on the platform, from the installation itself to the execution by the user and during the complete life span of the software. The second element is responsible for controlling the look and feel of the applications on the actual platform on which it is executed. The UI Framework is based on the well-known GTK+ toolkit.

Middleware. The middleware layer is composed of all the components, namely the daemons, services, and other non-UI elements of the system, which are available to the Application Manager Framework, the Application UI Framework, and the application space. The entities that are part of the Middleware are distributed in the following categories:

- Registry.
- Conflict Management.
- Event Delivery/IPC.
- Security Framework.
- Telephony Framework.
- Networking Framework.
- Messaging Framework.
- Multimedia Framework
- DRM Framework.
- Database.
- Other Frameworks.

Linux Kernel. All the previously mentioned layers are running on a common kernel architecture, which is the Linux kernel. This contains the drivers for each hardware component of the device and the advanced process and memory management that Linux is known for.



Source: [31]

Figure 3.4: Android architecture

Modem. The modem layer, which provides the telecommunication network interface, is considered as a separate component because it is often provided with a dedicated controller that has the capability to run by itself.

3.1.3 Open Handset Alliance (OHA)

The Open Handset Alliance was created to promote an open platform for mobile devices named Android. Different from the organizations described in Subsections 3.1.1 and 3.1.2, the Open Handset Alliance is more oriented towards an existing software platform and the associated development kit and less focused on creating a set of well documented APIs between the components. Fig. 3.4 shows the resulting platform with all the included components that are classified in these categories:

Applications. This category was composed, at the platform's launch, of a small set of default core applications. But it is expandable to an unlimited amount of user-created software. The third party applications have access to the same programming API as the core applications to assure that the same capabilities will be accessible. The execution priority will be shared equally between the core applications and the third party applications to assure that the execution time stays fair for everyone [32].

Application Framework. The application framework is a set of managers that offers a collection of services to the applications. These services are the only elements that can be used from the application layer and they can be accessed through the Android Java API.

Libraries. These are a set of C/C++ libraries that can be used by the applications. However, access to these libraries is provided by the Application Framework and is limited to the available features of the framework. The C/C++ libraries are taking advantage of a lot of open source development that took place in the past years.

Android Runtime. The Android runtime is the execution environment of the applications that are running inside the framework. This component is, in fact, a Java virtual machine that acts as a bridge between the applications and application framework, which are in Java, and the libraries and Linux kernel, which are natively running on the platform.

Linux Kernel. The Linux kernel is also the last layer of this architecture and it provides the drivers for the hardware and the advanced low level memory management, process management, file management, and many more.

A particular distinction of the framework defined by the OHA is that it focuses on building phone applications. In fact, the design philosophy that is stated by the

OHA documentation [32] enumerates the main characteristics that an application must have to be used efficiently in the context of an Android mobile device. These characteristics are categorized as performance, responsiveness and seamlessness. The way memory and thread management is achieved by the Android framework is also mainly targeted for mobile embedded devices.

3.1.4 FreeSmartPhone.org

The freesmartphone.org is a work in progress that aims to create collaboration and discussion about free software for the ”smart phone” type of devices. The project is based on specifications and software that was already created by Freedesktop.org [33]. Their work focuses on adapting the specification for smart phones and on adding the components that are specific for this category of devices. The main issues that are covered are related to power conservation and network connection management, while still offering a good level of performance to use a multitude of applications.

Their first work item is to provide a set of middleware APIs for a common set of services. These services will assure that low-level development of core services will be interoperable between different mobile Linux distributions. These core services will then be usable from any type of interfaces, no matter which technology they are based on. The device model is based on work from the Universal Mobile Telecommunications System (UMTS) group.

To provide the APIs, Freesmartphone.org is building over the D-Bus messaging system. This project, hosted at Freedesktop.org [33] is providing a lightweight mechanism for inter-process communication between system services and user applications. When a component is added to the system, it can advertise its available services through D-Bus and any other compatible application will be aware of their availability. On the host system, D-Bus is provided by a set of daemons that are constantly

FSO Framework Version 0.3.5
0. Introduction
- ... to be written ...
1. Device API
- org.freesmartphone.Device.Audio — Audio Device Access
- org.freesmartphone.Device.Display — Display Device Access
- org.freesmartphone.Device.LED — LED Device Access
- org.freesmartphone.Device.Input — Input Device Access
- org.freesmartphone.Device.IdleNotifier — Idle Notification Service
- org.freesmartphone.Device.PowerControl — Device Power Control
- org.freesmartphone.Device.PowerSupply — Power Supply Access
- org.freesmartphone.Device.RealtimeClock — Realtime Clock
2. GSM Telephony API
- org.freesmartphone.GSM.MUX — GSM 07.10 Multiplexing
- org.freesmartphone.GSM.Device — Device Inquiry
- org.freesmartphone.GSM.SIM — SIM Card Access
- org.freesmartphone.GSM.Network — GSM Network Access
- org.freesmartphone.GSM.Call — GSM Voice Calls
- org.freesmartphone.GSM.SMS — Short Message Service
- org.freesmartphone.GSM.PDP — Packet Data Protocol Connections
- org.freesmartphone.GSM.CB — Cell Broadcast Service
- org.freesmartphone.GSM.HZ — O2/Genion HomeZone Service
3. Usage API
- org.freesmartphone.Usage — Resource Manager Service
- org.freesmartphone.Resource — Resource Control Interface
4. Phone API
- org.freesmartphone.Phone
- org.freesmartphone.Phone.Call
5. Preferences API
- org.freesmartphone.Preferences
- org.freesmartphone.Preferences.Service
Under development / Brainstorming
- org.freesmartphone.PIM
- org.freesmartphone.DateTime
- org.freesmartphone.LifeCycle

Source: [27]

Table 3.2: Freesmartphone.org framework D-Bus interface specification

monitoring the bus for new messages to transport between applications. Table 3.2 shows a preliminary version of the D-Bus API for the services defined by Freesmartphone.org.

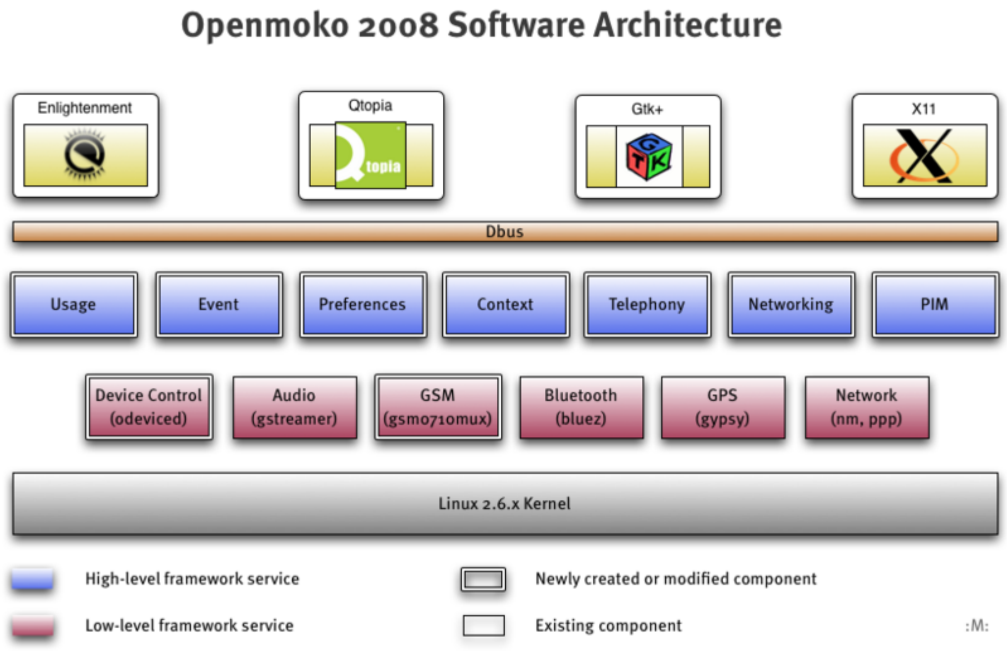
3.2 Current Open Software Platforms

This section presents the main existing software platforms for embedded mobile phones that are in active development at the time of writing this thesis. There exists some relation between two of these platforms, Android and Openmoko, and the previously mentioned standardization efforts. The last one, Qtopia, is a free software stack that was implemented in the context of a free phone project. All of the platforms are based on the Linux kernel and are using free and open source software libraries at the core of their main functionalities.

3.2.1 Android

The Android platform [7] is the implementation that is promoted by the OHA described in Subsection 3.1.3. It is unique when compared with the other projects because all the development is done with a new Java API that was created for the platform. This API offers many mechanisms and libraries to carry on the development of new and forward-looking mobile applications. However, when compared with the two other platforms, Android can be considered as less open since the developers cannot access the system outside the scope of the Java virtual machine. This prevents Android applications to access the standard Linux features and, moreover, the impressive amount of already existing Linux applications will not be able to run over Android.

At the time of writing these lines, development for the Android platform was pos-



Source: [5]

Figure 3.5: Openmoko software architecture

sible with a SDK (Software Development Kit) provided from their website. The SDK needs to be installed as an Eclipse [34] plugin that includes all the necessary tools for development, namely a code editor, a debugger and a phone emulator. This last tool enables applications testing in conditions similar to the real execution environment, even if no actual hardware is available.

3.2.2 Openmoko

The Openmoko [5] software platform was created to become the basis of the open mobile device of the same name. The main objective of the project is to offer the first completely open mobile phone device. This means that all aspect of the device must be open and accessible whenever possible, including the hardware components, the drivers, the operating system and the applications. By offering this level of openness, the users will be able to add and to innovate over this platform without ever being limited by closed and undocumented components. The software architecture of the Openmoko platform is depicted in Fig. 3.5.

The other unique aspect of the Openmoko project is its development process which is also mostly open and public. This enables anybody to look at and to participate in the design decisions about the hardware and software of the project. This was made possible because the team's communication is established using different tools that are publicly accessible on the web, for instance, a public forum, mailing lists, a wiki and an IRC (Internet Relay Chat) channel. With regard to private internal communications inside the Openmoko company, a community update message containing the key information is regularly transmitted to the "community" mailing list.

The FreeSmartPhone organization mentioned in Subsection 3.1.4 is the standardization effort that is behind the Openmoko project. The architecture and the middleware services for Openmoko will follow the FSO framework specification. A reference implementation is currently developed in a test distribution named FSO, but this development will be integrated in the main Openmoko distribution when it will be ready. These high-level services will provide an easier API to access the information that is usually available on a "smart phone" type of device. For example, this information can be categorized as in the following:

- Usage
- Event
- Preferences
- Context
- Telephony
- Networking

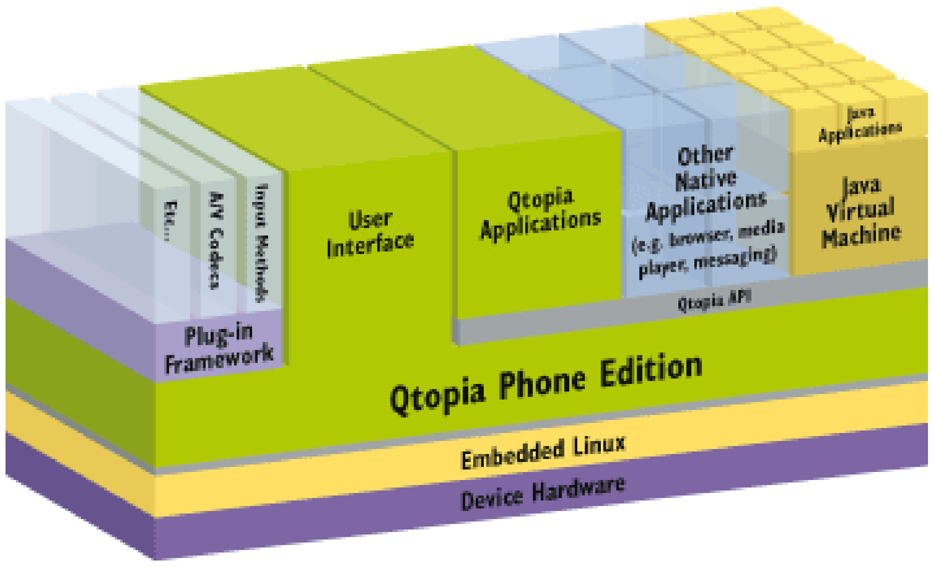
- PIM (Personal Information Management)

The PIM term is used to reference the personal data management capabilities of a device. The data that is considered in this category are the contacts, calendar, todo list, text messages, email messages, etc. The development of some low-level services is also planned to simplify the control of the GSM subsystem and the other optional hardware systems:

- GSM (gsm0710mux)
- Device Control (odeviced)

Currently, the OM2008 official distribution still does not provide any middleware services. Instead, the developers wanted to rapidly implement the Openmoko operating system as the base software that would enable other third party developers to add their own applications. Many already existing open source components are used in this platform to provide a functional core.

The Openmoko Linux distribution is based on the OpenEmbedded [35] environment, which provides all the tools required to generate a fully functional Linux operating system for embedded devices. The development environment consists mostly of the standard Linux development tools such as the GCC compiler and the autotools build management system. Moreover, a QEMU [36] based phone emulator is provided to assure that developer can rapidly test their creation without owning any hardware. Because the Openmoko platform is also an almost complete Linux distribution, software debugging can be achieved on the development computer with typical Linux debugging tools.



Source: [6]

Figure 3.6: Qtopia Phone Edition software architecture

3.2.3 Qtopia Phone Edition

The Qtopia [6] Phone Edition software stack is a complete mobile phone environment that was originally created for the defunct Greenphone [3] device. The targeted distribution is mainly based on the regular Qtopia stack to which was added the elements required for phone operation, for Voice-over-IP operation and for multimedia interaction. In addition, a set of applications commonly seen on phones, to carry on personal information management tasks, have been built into the distribution. The architecture of Qtopia Phone Edition is presented in Fig. 3.6.

Qtopia is based on the QT cross-platform application framework for desktop and embedded development. This framework and the Qtopia distribution were developed by a Linux development company called Trolltech. The Trolltech products were not distributed under Open Source licenses until recently. Currently, the company is offering different licensing options for its operating systems, including the well-known open source license GPLv2.

Development tools that are available for Qtopia Phone Edition are the same that were provided for development with the QT application framework. This set of tools is the most complete when compared to the other platforms described in this document. They are composed of:

qmake. An automatic build system.

QT Designer. A visual forms and dialogs designer.

QT Compilers. A set of compilers for different elements that can be accessed from an application. There are compilers for meta-objects, user interface and data resources.

QT classes. A collection of high-level development libraries used to help and accelerate applications development.

Others. Many other tools, not mentioned here, are provided by the QT application framework.

QT development is conducted in C++ and applications are built with the traditional GCC compiler. QT applications can be debugged with usual Linux code debugger such as the GNU Debugger (GDB).

3.3 Platform Selection

In the scope of this work, a main development platform needed to be chosen in order to base the development on a standard set of tools and to test broadcast reception equipment on actual hardware devices. Some criteria were defined that would help in taking a decision about the first platform on which development would be started. The list of criteria is summarized in Table 3.3.

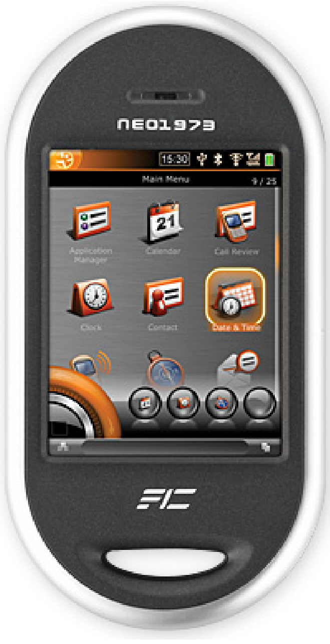
The Google Android platform was rejected first because it is not compatible with most of these requirements. For instance, the API to create new applications is open

Criteria
Openness of the software
Openness of the hardware
License
Availability of the hardware
Quality of development tools
Cross compatibility with other platforms
Support of hardware extension

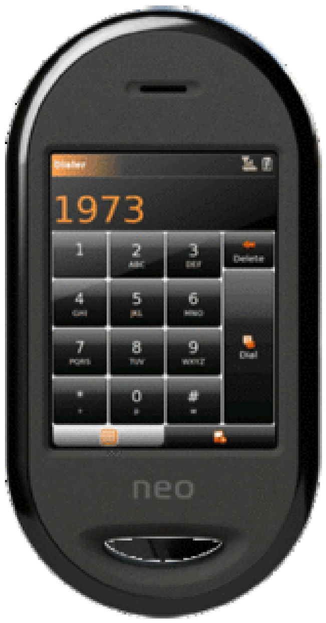
Table 3.3: Criteria for development platform selection

but the platform itself is not. This would be a major drawback when trying to adapt new hardware, which would also need new drivers, to the platform. Moreover, at the time of implementing this work, no Android hardware was available and the first prototypes were then expected for the beginning of 2009. Finally, because Android development is conducted using the Java programming language and is running on a specific runtime environment, compatibility of Android applications to other platforms is not expected.

It was then decided to target one of the two Linux mobile platforms, either Qtopia or Openmoko. Again considering the criteria from Table 3.3, Openmoko proved to be the most compliant for different reasons. First of all, it was designed to become the most open mobile phone platform to date, as it offers every detail of the hardware design, specifications and documentation, in a publicly available format on their website [5]. This would enable any hardware developer to create a component to extend the functionalities of the device. Moreover, the Openmoko software platform is based on a standard Linux distribution and it can be considered as the embedded platform which is the nearest to a standard desktop Linux system. The software architecture, which is described in Subsection 3.2.2, is compatible with a wide array of application development frameworks. Because Openmoko is a standard Linux system, any development made for that platform should be easily portable to any of the other mobile



Neo 1973
codename GTA01
(released July 2007)



Neo FreeRunner
codename GTA02
(released July 2008)

Source: [5]

Figure 3.7: Neo series of mobile devices

and embedded Linux systems. Finally, at the time of writing, Openmoko was the only project that had actual hardware available to developers. In fact, the Neo series of phones are completely functional developer devices that were designed to run the Openmoko software stack. Fig. 3.7 shows the two devices that are released to date. The Neo 1973 is the first developer version that was released with non-final hardware. The Neo FreeRunner is the latest release which has consumer-level hardware.

The hardware available on the FreeRunner is quite complete and it can be compared with any latest-generation device available these days. The detailed specifications are as follow:

- High resolution touch screen 2.84" (43mm x 58mm) 480x640 pixels
- 128MB SDRAM memory
- 256 MB integrated flash memory (expandable with microSD or microSDHC card)
- microSD slot supporting up to 8GB SDHC cards
- Internal GPS module
- Bluetooth
- 802.11 b/g WiFi
- 400Mhz ARM processor
- 2 x 3D accelerometers
- 2 LEDs illuminating the two buttons on the rim of the case (one bicolor [blue / orange] behind the power button, 1 unicolor [red] behind the aux button)
- Triband

- GSM and GPRS
- USB Host function with 500mA power, allowing to power USB devices for short periods

What is interesting to note is that the Neo devices are generic Linux-compatible hardware. Hence it is quite easy to adapt mostly any Linux-based system for the device. Many efforts were started to port popular Linux operating systems for the Neo as shown in Fig. 3.8. The distributions from the figure are classified according to their main target platform and according to their sponsoring organization. Some of these distributions are built specifically for the Openmoko devices and some of them are sponsored by the Openmoko, Inc company. The OM2007.x branch was the first iteration of the Openmoko platform but it is not developed anymore. The OM2008.x distributions, which is also described in Subsection 3.2.2, were developed by Openmoko and they are currently the official distributions for their devices. The FSO distribution is the implementation of the framework from Subsection 3.1.4 and will become the official Openmoko distribution upon completion. The next distributions in the figure, FDOM, SHR and Hackable.1, are developed for the Openmoko devices by communities of developers that did not like the direction that the official distributions were heading. SHR wants to offer a more stable version of FSO. FDOM wants a version of OM2008.12 with more features and Hackable.1 wants a mobile distribution based on the GNOME Mobile [37] standards. Finally, the last three in the list were not even developed for the Openmoko devices but some work was conducted to port them for the FreeRunner. Even a full featured distribution targeting the desktop computer, such as Debian, has proved to be usable on the FreeRunner. Another notable distribution is the Android platform from Subsection 3.2.1 that was successfully used on the Neo FreeRunner. The Openmoko platform was selected to be used in the course of this project and the main official distribution, currently OM2008.12, was

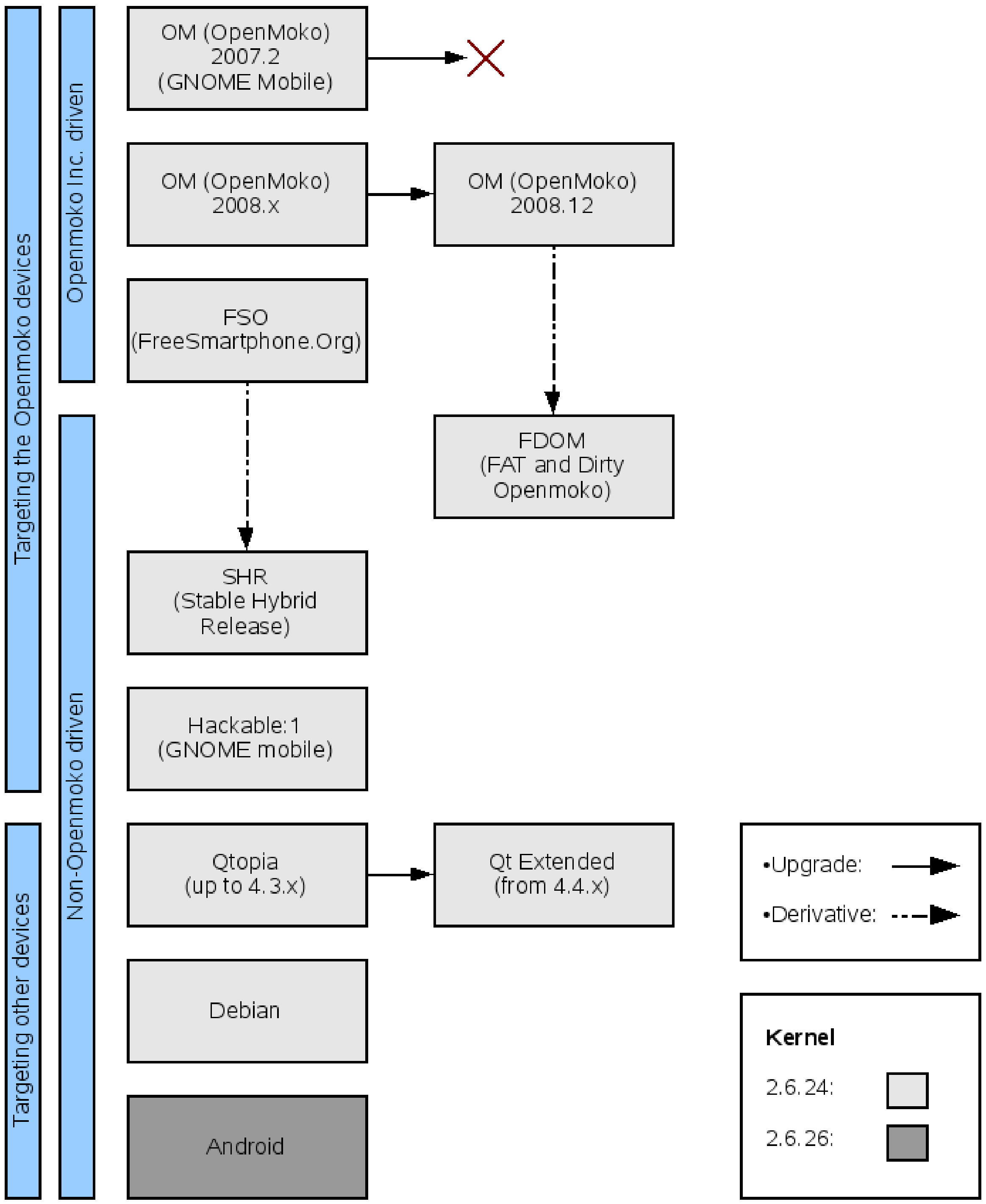


Figure 3.8: Available software distributions for the Neo FreeRunner

selected as the base operating system.

Finally, as mentioned previously, Qtopia was also considered. However, the main drawbacks of this platform, when compared with Openmoko, was its lack of flexibility, as it is compatible only with QT development. Other important issues are the lack of compatible hardware and the fact that this project is a proprietary platform from Trolltech that only offers an optional open source version of its software. This last issue, however, was partly resolved when Nokia announced [38] that the QT development framework will be available under the LGPL license in future releases.

Chapter 4

Advantages of Mobile Digital Broadcasting

”Because network operators control the devices’ design, only the services that operators want to include will be offered.”

– *Skip Pizzi.*

THE problem addressed by this research project is the integration of broadcasting technologies to open mobile devices projects. Broadcasting is referring to the collection of new protocols to transport digital data in a wireless unidirectional channel as described in Section 2.2. The open mobile devices projects are referring to a class of hardware and software development efforts with the goal of creating a new generation of mobile devices based on open source software and open hardware platforms. These projects all have the common goal of permitting the devices to be modified and extended by the users while not being limited by restrictive user license agreements and patents.

As it was briefly mentionned in the introduction, a complete class of broadcasting protocols are nowadays absent from popular telecommunication devices such as mobile

WORLD DMB
Digital Multimedia Broadcasting
Radio • Mobile TV • Multimedia • Traffic Data
DVB H
ATSC-M/H
drm
HD Radio

Figure 4.1: Common Mobile Multimedia Broadcasting (MMB) technologies

phones, the most notable being shown in Fig. 4.1. The telecommunication network operators are currently selling the capacity of their own network technologies based on:

- CDMA2000 (using 1X, EVDO, etc, for data access).
- GSM (using GPRS, EDGE, etc, for data access).
- 3G (using UMTS for data access).

The use of broadcasting in this context would be in direct competition with their network for high quality multimedia content.

As outlined in Chapter 3, the current open device development projects are focusing their efforts on providing a number of APIs to control the most common features of a mobile device such as the phone component, the network connection component, the multimedia capabilities, the PIM components and so on. From these available APIs, the only networking that is available is the traditional IP networking, which is supported by most telecommunication networks such as WiFi, WiMAX, GPRS, 3G/UMTS, EVDO and Bluetooth. On the other hand, broadcasting technologies are completely left aside and no mechanism have been included to offer this possibility.

Because there are many differences when we compare traditional telecommunications networks with unidirectional broadcasting networks, it would be very difficult to convert functionalities from an existing networking API to broadcasting. In fact, where a networking API would use features such as CONNECT, REQUEST, RESPOND, SEND and RECEIVE, the broadcasting counterpart would use SCAN, TUNE, ADD COMPONENT and REMOVE COMPONENT. The communication process for an IP network usually takes place in one of these two types: connection oriented communication (TPC/IP) and connectionless communication (UDP/IP). In the case of TCP/IP, an exchange of request and response data packets is conducted, where each part of the information is transmitted from one end and is acknowledged on the other end to prevent data loss. The UDP/IP communication does not use this acknowledgment mechanism but it sends packets from a source to a unique destination address. A broadcasting receiver is, on the other side, connecting to a stream that is already present in the air. Next, the content of the stream is analyzed to extract the information about the multiple data components available, if any. Finally, the receiver selects which parts of the transmission needs to be extracted and passed to the upper layer decoding application. During the stream extraction, no data is sent back to the transmitter at all and any lost packet of information cannot be retransmitted to the receiver.

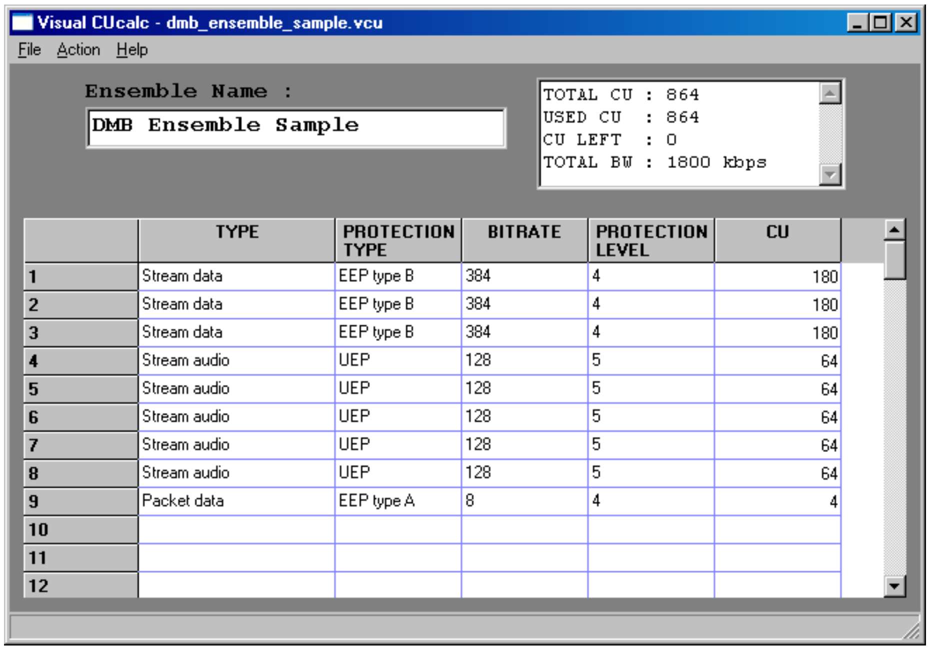
There are many issues that could be addressed by enabling broadcasting technologies on such open devices. To begin with, broadcasting technologies have traditionally been associated with mobile communication because they feature large downstream capabilities for a large amount of users at once. Moreover, this is requiring a fraction of the bandwidth, when compared to one-to-one communication networks. This is, however, only one of the main incentive in providing this technology as shown in the following discussion.

Shifts the control of mobile devices from telecommunication companies to broadcasters. In the current mobile devices ecosystem, the phone-like devices provided by the cellular networks operators have a major market advantage when considering the amount of devices in operation. The users have limited pocket real estate and it is used mostly, of course, by the most popular mobile application, the phone. When it comes to offering new mobile applications and capabilities, broadcasting has always been kept away from these devices because the broadcasting networks are in direct competition with the telecommunication network, which could harm the massive profits generated by the multimedia services provided by the expensive and less efficient cellular network. For example, as pointed out by Skip Pizzi [2], the telecommunication industry is often deactivating the FM receivers usually found in cell phones in order to force the users to consume audio services via their pay-per-use network. From the broadcasters point-of-view, there is then a large incentive to obtain the technology required to create broadcast-enabled devices that would be compatible with the large amount of available broadcasting technologies without restrictions.

Another example is the control that was exercised by Apple and Rogers during the launch of the iPhone 3G in Canada during the summer 2008. One of the interesting feature of that device is the so called ”Apple Store”, with which developers from around the world can develop and distribute their applications for the iPhone. The problem is that Apple is keeping a tight control over the applications that can or cannot be distributed. As expected, some really useful applications have been refused because they were providing services that were supposedly against their policy. The most obvious case comes from the applications that would support VoIP (Voice over IP) over the 3G network, hence bypassing the very profitable traditional voice network. As expected, VoIP applications are strictly forbidden outside of the WiFi network [39]. Moreover, even if the iPhone has great hardware capabilities for audio, video and multimedia playout, no broadcasting component has been included nor offered for the device.

Opens the network to any developers and applications. The Internet, as a free and open platform for development, has seen a tremendous growth since it was commonly introduced in the beginning of the 1990s. The main feature of this networking technology is that any developer is free to create a web page or, as it was popularized during the Web 2.0 evolution, a web application. The developers also have the capability to use the network at its full capacity and they only need to pay for their usage of the network following a pay-per-use model, meaning that popular and bandwidth-consuming services will have to pay more than less popular services. Moreover, the barrier of entry to web development is very low, as anyone with a computer can use some state-of-the-art development tools and participate in the innovation of the web. The ”raison d’être” of ISPs (Internet Service Providers) is, in the case of the Internet, to provide the network and to upgrade as more and more resources are required. They can, of course, as anyone, try to develop and offer web services but they will compete on level ground with any other application developers in the world. The result of this experience has shown that restrictive control, as it is currently the case in mobile telecommunications, can only slow down innovation. To make a parallel with the case of the Internet, the free and open access to broadcasting applications development is believed to be the enabling technology that is missing to trigger a new market of innovative broadcasting services.

Provides a new medium to support innovative applications. By enabling broadcasting protocols on mobile devices, a collection of new and innovative applications could be developed to take advantage of this technology. Some applications that are currently available on mobile telecommunication networks are, in fact, fundamentally broadcasting applications. We can think of live video transmission of popular events, popular data which is accessed by most users such as the weather information, and many more. However, these are currently provided by the unicast network which



Ensemble Name : DMB Ensemble Sample

```
TOTAL CU : 864
USED CU  : 864
CU LEFT  : 0
TOTAL BW : 1800 kbps
```

	TYPE	PROTECTION TYPE	BITRATE	PROTECTION LEVEL	CU
1	Stream data	EEP type B	384	4	180
2	Stream data	EEP type B	384	4	180
3	Stream data	EEP type B	384	4	180
4	Stream audio	UEP	128	5	64
5	Stream audio	UEP	128	5	64
6	Stream audio	UEP	128	5	64
7	Stream audio	UEP	128	5	64
8	Stream audio	UEP	128	5	64
9	Packet data	EEP type A	8	4	4
10					
11					
12					

Figure 4.2: Example of a DMB ensemble capacity and channel usage

is far from being optimal in this situation. The access to a broadcasting technology would enable to efficiently support those fundamentally broadcasting applications, as well as new types of services, some of which we cannot even think of. Some exemple of innovative services were developped in the scope of previous projects which are described in [40] and [41]. As another example, highly popular content found on the Web could be converted into a broadcasting service, like it was demonstrated in [42]. New innovations in that field could even combine multiple networks. For example, a live blogging service could use GPRS for a single upload from a user but it could take advantage of broadcasting to transmit the information back to every subscribers.

Provides large downstream capacity. Most broadcasting technologies provide a large downstream pipe that can be used to stream bandwidth-intensive multimedia applications such as high quality audio and video. The DAB/DMB technology introduced in Section 2.2, for instance, offers up to 1.8 Mbit/s in a single channel and it can be split into multiple subchannels that are each free to transport any type of service. Fig. 4.2 shows an example of a DMB ensemble that transports three 384 Kbit/s video services, five 128 Kbit/s high quality audio services and one 8 Kbit/s low bandwidth data service for text news.

Supports an infinite number of users without affecting the performance of the network. One of the main features of broadcasting networks is that bandwidth can be used by an infinite number of receivers at once with no effects on the transmission. In fact, the transmitter has no idea of the number of equipments that are currently tuned to the transmitted content. This makes the technology very well suited for extremely popular content which a lot of users want to consume at once.

Achieves excellent performance in mobile reception. Some broadcasting technologies, including DAB/DMB, have been designed from the ground up with mobility in mind. The modulation and coding schemes used have proved their good performance in many mobile usage scenarios [43]. One of the scenarios that usually causes problems in transmission systems is the dense urban area, in which the buildings create many sources for signal reflexions and refractions, hence causing heavy multipath at the receiver. These situations are also problematic for systems that need a strong line-of-sight component, such as satellite communications. Terrestrial digital broadcast is then a technology of choice for multimedia applications in these dense urban areas.

Part II

Implemention of the Open Broadcasting Receiver

Chapter 5

Development Components

”80 percent of all commercial software products will include elements of open-source code by 2010.”

– *Gartner (2007).*

THE implementation of the Openmokast software requires the integration of multiple components that will interoperate to enable the different features of the software. An important contribution of the project to the mobile Linux world will be an API for broadcasting services. The goal of this API is to become the starting point used by any developers that would like to use a broadcasting receiver inside their mobile Linux application. This chapter presents an overview of the components involved in the Openmokast project.

5.1 Receiver Management

Openmokast is based on a software platform that was implemented at the CRC earlier to receive digital broadcasting data from a receiver and to dispatch the content to a decoder through an IP network. In the course of this project, the platform was enhanced to support more inputs and outputs and new control options. The software

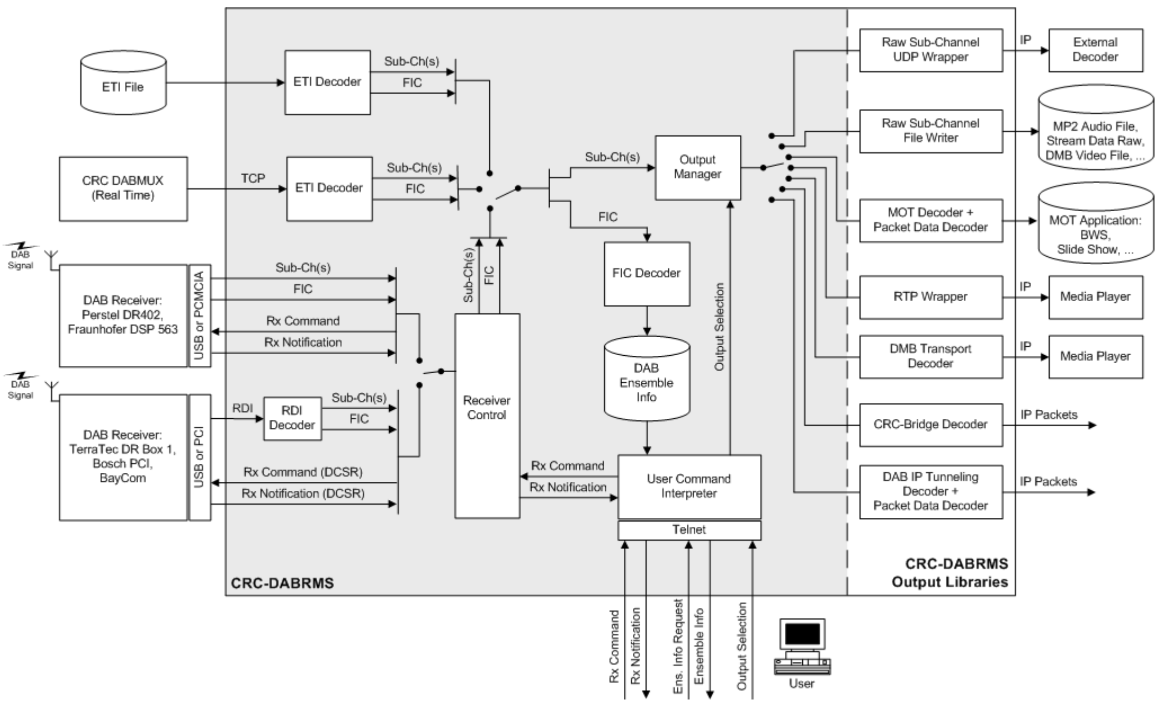


Figure 5.1: Architecture of the CRC-DABRMS broadcasting receiver software

architecture of the platform, named CRC-DABRMS, is depicted in Fig. 5.1. On the left part of the figure, different inputs can be used to grab the data. These inputs exist in two categories, the physical inputs and the simulated inputs. Physical inputs are actual broadcasting receivers such as the Terratec Dr Box, the Bosch PCI receiver and the Perstel DR402. These receivers are capable of receiving DAB services over the air. The simulated inputs, namely a file input and a basic TCP/IP input, are capable to receive raw broadcasting data as if it was processed by the DAB receiver. This last type of input is perfect for a testing environment, where a real wireless signal is not available.

Right after the start of data acquisition by the receiver, the information about the available data services is extracted and inserted in memory for future access. This information is used to present the services to the user. As it was explained in Section 2.2, the structure of a DAB ensemble is composed of two main streams:

Fast Information Channel (FIC): The FIC basically transports the information about the ensemble. This includes the description of the services contained in the MSC and the way they are multiplexed. The FIC also contains the structure of the subchannels, components and services inside the ensemble. This part of the stream is the only one that must be decoded by the receiver after a new ensemble is tuned. The information is constantly re-transmitted in a rapid loop to decrease tuning latency.

Main Service Channel (MSC): The MSC is a much less complex structure when compared to the FIC. The MSC is basically only a multiplex of all the data subchannels that compose the ensemble. Usually a receiver will not access this information unless the user starts the playout of a service. In this case it will extract only the required part of the MSC. Depending on the hardware capabilities of a receiver, it will be able to extract only one, two or multiple subchannels at a time.

In the receiver management software, there is a clear separation between the data from the FIC and the data from the MSC. The FIC is sent to a ”FIC Decoder” block to be interpreted while the MSC is sent to the ”Output Manager”. At the beginning, when a new ensemble is tuned, the ”Output Manager” is not transmitting any streams at the output. Instead, it waits to receive instructions from the ”User Command Interpreter” block.

A control interface is available through a simple console telnet connection. After connecting to the console, the user can issue some pre-defined commands to read the information about the services, to set up an output socket and to start and stop the services. The ”User Command Interpreter” block receives the commands and calls the right block of the receiver software with the associated command.

The last part of the CRC-DABRMS architecture is composed of a collection of output libraries that can be used to transport, convert and process the services. Some of them will only forward the stream, for instance the UDP/IP output will transmit the raw data of the service using a simple UDP/IP socket. Others will include a stage of conversion of the data, such as the RTP (Real-time Transport Protocol) output which will insert an MPEG2 audio service inside an RTP/IP stream and send it to a compatible audio player.

5.2 Service Interface

The service interface is the component that lets application developers use the broadcasting framework for their own needs. For this purpose, an IPC (Inter-Process Communication) system is used to enable the communication between the framework and the application. This section describes the D-Bus system that is used in the Openmokast project.

D-Bus is a mechanism used to transport messages between applications over a simple bus system. An application or a daemon can use D-Bus to offer a collection of services to other applications on the same computer. A client connecting to the bus will then be able to look up which services are available. Moreover, D-Bus can provide the coordination of the process life cycle by launching applications and daemons on demand when their services are needed. D-Bus provides two main types of communication entities: (1) a system daemon, which can be used to implement system events such as hardware detection and (2) a user daemon, that can be used for general inter-process communication among user applications. The D-Bus developers are commenting about the stability of the framework on their project page [33]:

> The message bus is built on top of a general one-to-one message passing framework, which can be used by any two apps to communicate directly

> (without going through the message bus daemon). The D-Bus low-level API reference implementation and protocol have been heavily tested in the real world over several years, and are now "set in stone." Future changes will either be compatible or versioned appropriately.

The D-Bus messaging system is still a project in developement but it is used in more software projects every month. Moreover, the constraints encountered in embedded development are to be taken into consideration when selecting the components that will be used for the Openmokast framework. The D-Bus system offers, however, many features that makes it worth considering for this project:

- The low-level libdbus implementation has no required dependencies and the bus daemon has only one required dependency, an XML (Extensible Markup Language) parser, which can be either libxml or expat. Higher level bindings mentioned below could add some dependencies.

- A large collection of language-specific and framework-specific higher-level bindings are available for D-Bus. These bindings can make more assumptions and are thus much simpler to use, but at the cost of augmenting the complexity of dependencies of the software. These bindings are developed by separate groups, thus they are usually not as mature as the main D-Bus project. There are currently high-level APIs that are usable with Qt, GLib, Java, C# and Python.

- D-Bus is portable to any Linux or UNIX flavor. Also, a port to Windows is expected in the near future.

- D-Bus is the main IPC mechanism that is part of the two most widely used Linux desktop environments, namely KDE[1] and GNOME.

[1]KDE 4 uses D-Bus as a replacement of the DCOP system used in KDE 2 and 3.

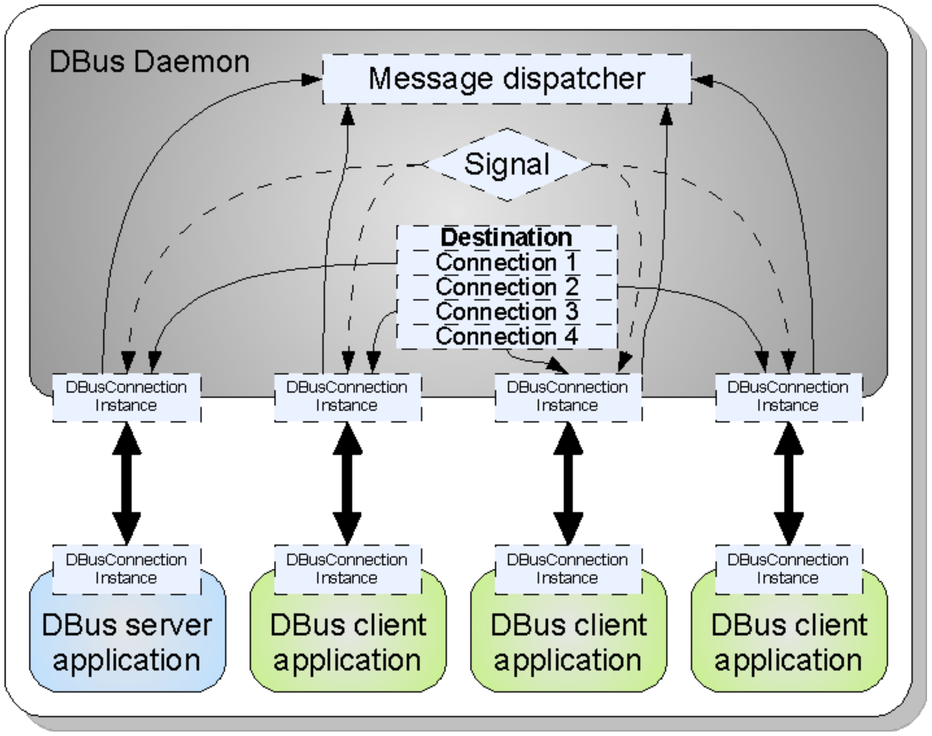


Figure 5.2: D-Bus communication diagram

- The service definition used in D-Bus enables a framework to offer different methods and signals to any other application. This is well suited for a mobile broadcasting receiver where methods can be used to control the tuner and signals can be used to send updates on the status of the receiver and the new available multimedia streams.

The D-Bus system communication is pictured in Fig. 5.2. Each application is connected to the D-Bus daemon through an instance of the *DBusConnection* object. The message daemon acts as a message dispatcher by retransmitting the information to the right location. There are two different types of communication: regular messages and signals. A regular message will be sent to a single destination. A signal will be broadcasted to any application that is listening for this type of signal. Using D-Bus, each application can send messages to any other. The diagram also show two different types of applications: server and client. The server offers a service through

D-Bus that can be used by any client application. Any application can use the model of the server, the client or both at the same time. This communication system, on the other side, is not meant to carry high bandwidth and low latency applications such as multimedia streaming. For this reason, the multimedia data streams will not be transported by it. Instead, they can use the system's IP networking layer which offer simple and efficient mechanisms for that purpose such as UDP/IP.

The D-Bus architecture will most likely obtain good acceptance in the mobile devices world because it is part of the majority of standards described in Section 3.1. More specifically, it is the base of the mobile phone framework defined by the FreeSmartPhone.org organization and it is the standard IPC system for the LiPS and the LiMo groups.

Many other technologies are available to provide IPC inside a Linux application and were considered in the scope of this project, however, none of them have the level of acceptance of D-Bus in the mobile Linux world. Moreover, in some cases, either they were missing some of the features mentioned above or they were providing too much overhead to the system. The other main IPC systems on Linux that were considered are:

- Missing features:
  - Message Bus (MBUS)
  - Lightweight Communications and Marshalling (LCM)
  - ONC RPC
  - XML based XML-RPC or SOAP
  - Anonymous pipes and named pipes
  - Sockets

- Unwanted overhead for embedded devices:
  - Common Object Request Broker Architecture (CORBA)
  - Distributed Computing Environment (DCE/RPC)
- Less commonly accepted in mobile Linux
  - KDE's Desktop Communications Protocol (DCOP)

5.3 Multimedia Applications and Frameworks

Many multimedia application decoder projects are available on the Internet, supporting an enormous amount of different protocols and formats. In the course of the Openmokast project, it was considered a great idea to take advantage of these available components and to integrate them inside the software instead of reimplementing everything from scratch. Because DAB digital broadcasting is mostly a digital pipe that transports data, no matter their format, it was possible to experiment with many non-standard multimedia formats ranging from simple text news to full audio/video streams.

Some of these projects were developed for traditional multimedia formats, so the parts of the stream that are specific to mobile broadcasting, such as the Reed Solomon error correction, need to be removed prior to send them in the decoder application. Other available libraries are implementations of multimedia protocols specific to mobile broadcasting such as the open source MOT decoder. More details about these components are provided in the following sections.

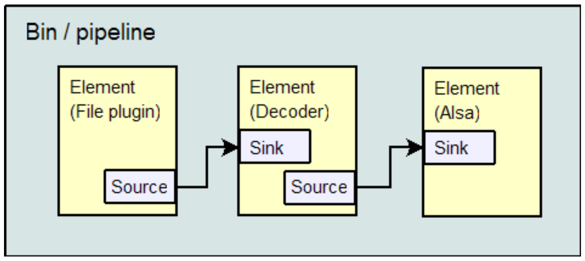


Figure 5.3: GStreamer technical overview

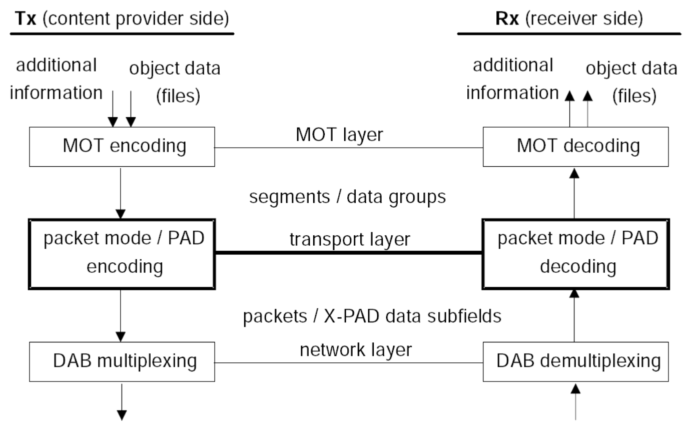
5.3.1 MPlayer

The MPlayer software [44] is the ultimate multimedia player that supports just about any available format. Moreover, MPlayer is compatible with many common multimedia network transport protocols. This makes it real easy to use because it will be able to play a stream only by specifying the URL (Uniform Resource Locator), there is no need to grab it from the network and record it into a file. This software can be launched as an external process with the URL of the multimedia stream as a parameter. MPlayer is excellent as a decoder but because it is an application that must be launched externally from Openmokast, it cannot be controlled easily and errors are not convenient to detect from inside the Openmokast application.

5.3.2 GStreamer

The GStreamer framework [45] is a library for linking together different multimedia processor components in a graph-like decoding chain. The processing components are known as plugins inside the framework. By using GStreamer, it is possible to choose from a large collection of already available plugins but it is also easy to create new components related to our needs.

Three types of plugins are available, namely sources, filters and sinks. Sources are always the first elements of the decoding sequence, as they acquire data from a source external to the framework. Filters can be linked together in different ways to



Source: [47]

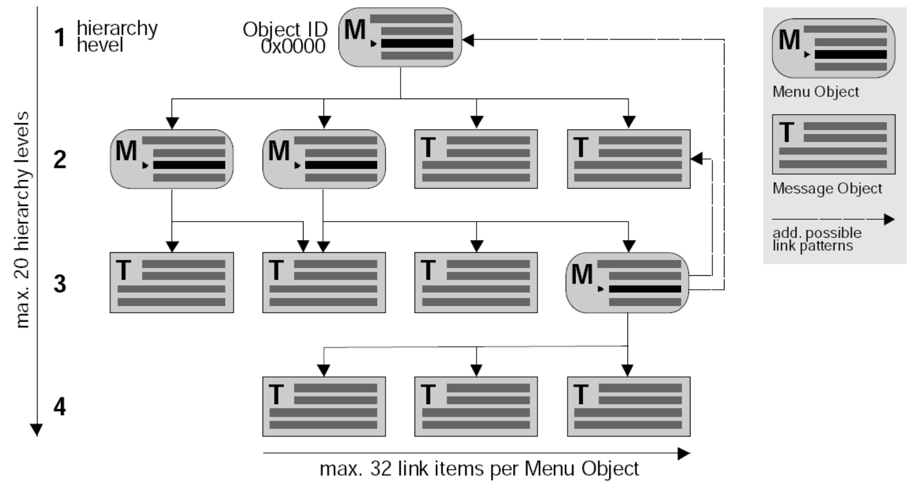
Figure 5.4: MOT transport protocol layers

achieve the decoding results that are expected by the user. For instance, an audio player for movie files would require a component to de-multiplex the audio from the main stream and an audio decoder compatible with the codec used in the stream. The last type, the sink, is used to send the result back to the system, such as in a file or through an audio output mechanism. An example of a GStreamer graph is shown in Fig 5.3.

5.3.3 Dream Project

The Dream [46] project is a reference implementation of a DRM (Digital Radio Mondial) receiver. The DRM technology is different from DAB when considering the transmission channel. However, at the application layer, different multimedia data protocols from DAB were reused in DRM. Hence, the Dream implementation contains decoding libraries for some of these multimedia applications such as MOT [47] and Journaline [48].

The MOT library can be used to transparently extract the multimedia objects from the radio data subchannel. The subchannel must be decoded up to the transport layer, after which some datagroups are extracted and passed to the MOT decoder as



Source: [48]

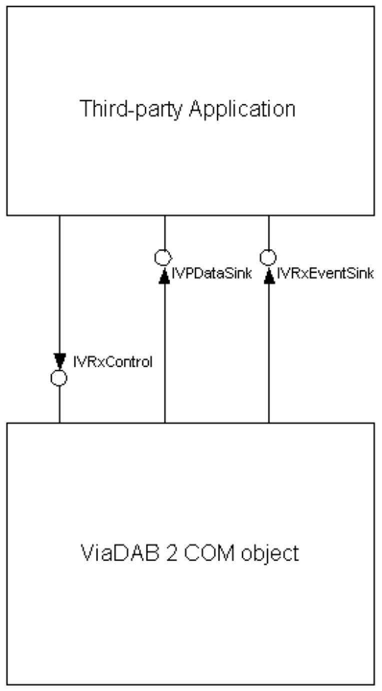
Figure 5.5: Journaline hierarchy example

shown is Fig. 5.4. The MOT library then informs the application when new objects become available and manages the access to the content of the objects.

The Journaline library works at the same level as the MOT library but it extracts a JML (Journaline Markup Language) structure. This structure, demonstrated in Fig. 5.5, can then be browsed by the main application. Each page contains text elements and links to other parts of the news content.

5.4 Broadcasting APIs

To define the Openmokast API, many other already existing interfaces were studied. This section of the thesis is an overview of the considered APIs. Not all of these could be directly applied because they are targeting either the hardware link communication of a device or the communication between an application and its driver. However, many of these APIs had a good conceptual format from which the Openmokast API could get a solid basis.



Source: [49]

Figure 5.6: ViaDAB receiver interfaces

5.4.1 ViaDAB

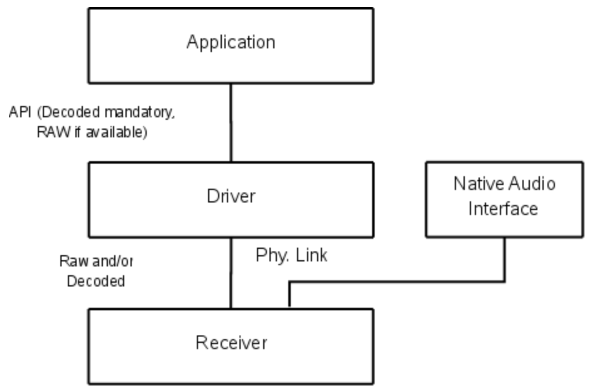
The ViaDAB API [49] was defined by Radioscape with the goal of offering a common way of controlling and extracting the data from a DAB/DMB receiver. This protocol is a high-level set of methods that can be used between an application running on a computer and the driver that is handling the device. Each of the methods have a set of input parameters that are represented with standard data types such as integers, doubles and strings. The call to these methods is done in a synchronous way, which means the call is blocking until the return value is ready. As depicted in Fig. 5.6, the API is split into three different interfaces, namely IVRxControl, IVRxEventSink and IVPDataSink. The IVRxControl interface is used to control the receiver and to access the Multiplex Configuration Information (MCI) data. The IVRxEventSink interface informs the calling application of events that are occurring inside the receiver. Finally, the IVPDataSink interface is used to transfer the raw data from the receiver to the client application.

5.4.2 DCSR

The DAB Command Set for Receiver (DCSR) [50] was also defined with the goal of controlling a DAB receiver. A set of methods are also defined with capabilities similar to ViaDAB. The major difference is that the calls must be created by generating a stream of bytes that contains a command ID followed by a binary map of the parameters. The calls are handled asynchronously. The communication will instantly return a value telling if the command was accepted by the receiver, but the return data stream will be sent back to the caller through a notification call that can be sent at any time. The binary format of the method call makes it easy to serialize the data for their transport over a physical serial bus such as USB and PCI. However, this format is less programmer friendly and is less suited for its usage directly inside the framework.

5.4.3 DAB-Java

The goal of providing the DAB-Java specification [51] was to enable Java applications to be executed on a DAB platform to expand the capabilities of a receiver. The specification defines how to implement a Java VM (Virtual Machine) for broadcasting receivers. A DAB-Java enabled receiver could receive a new application using the broadcasting channel, for example, through a MOT [47] service component. The specification also defines the DAB package, a Java class that enables the applications to access the DAB resources. The package consists of a set of commands where the application sends requests to the package and the package responds with confirmations and notifications. This follows the Event-Listener pattern between the Java



Source: [52]

Figure 5.7: Availability of raw and/or decoded data in a DRDI based receiver

application and the DAB receiver.

5.4.4 DRDI

The Digital Radio Data Interface (DRDI) [52, 53] was never officially completed but the specification documents were made available by the WorldDMB [19] technical committee to its members. The protocol uses a collection of tags, or messages, that are transported between the host and the broadcasting receiver. Moreover, it introduces the notion of transporting raw or decoded services. This means that, depending on the device capabilities, some services will be transported as the original encoded bitstream while others will provide the decoded and ready to play audio PCM[2] (Pulse-Code Modulation) audio format as illustrated in Fig. 5.7. DRDI also proposes an addressing mechanism to identify a unique service inside the digital radio space. An address identifies the frequency, the ensembles, the service and the component to assure that no two streams can be wrongly referenced as the same. For example, a single service could be represented by the string "dab.srv://<receiver module>*<ServiceAddress>" where the receiver module is an identifier for the device and the service address is a combination of DAB specific identifiers in the format

[2]PCM is a non-compressed digital representation of an analog signal.

"<SId>[.<ECC>][:<EnsembleAddress>]". These elements could be the base for service addressing in the Openmokast framework. The tags format, however, is meant to be transported on a low level physical layer, thus it's not very well adapted for a high level programming abstraction layer. DRDI defines tags to control many different types of broadcasting receivers, namely DAB/DMB, DRM, FM and AM. The API does not offer a generic approach to combining all these technologies, instead it includes different sets of methods, one for each of the compatible technologies.

5.4.5 RSCI

The Receiver Status and Control Interface (RSCI) [54, 53] was created for the DRM broadcasting technology. The format of this API is based on tags, or messages, like the previously mentioned DRDI protocol. The control commands are based on the DRM technology and the format is also optimized for lower-level serial communication.

5.4.6 Selection of an API

In conclusion, the ViaDAB API is simpler to use when compared to the other described APIs because it runs at a higher level and uses standard programming data types. ViaDAB was defined using the COM (Component Object Model) architecture that is a popular IPC system for the Microsoft environment. One section of ViaDAB, the IVRxControl interface, contains most of the element required to control a broadcast receiver. The list of control elements is shown in Table 5.1. On the other side, the list offers methods that are mostly for controlling the playout directly over a hardware device and are not well suited to a digital broadcasting framework. For example, the methods to control the volume of the device would be considered as not needed. For this reason, ViaDAB was selected as a good starting base for the Openmokast framework. In addition, some missing features were added to it and

IVRxControl interface method listing	
GetComponentIdArray	GetUserApplicationData
GetComponentLabel	GetVolume
GetDABInfo	IsDecoding
GetDABLocalTimeOffset	IsPlaying
GetDABTime	IsTuned
GetDataServiceComponentType	SetEventSink
GetDLS	SetVolume
GetDynamicProgrammeType	StartDecoding
GetEnsembleId	StartPlaying
GetEnsembleLabel	StopDecoding
GetFICBER	StopPlaying
GetProgrammeType	SubscribeFIC
GetServiceIdArray	SubscribeMSC
GetServiceLabel	Tune
GetTransportMode	Unsubscribe
GetUserApplicationArray	

Table 5.1: List of methods of the ViaDAB IVRxControl interface

some unnecessary features were removed. The details about the development of the Openmokast API are specified in Section 6.3 of the thesis.

5.5 Openmoko Development Environment

5.5.1 OpenEmbedded Framework

The build environment for the Openmoko platform is based on the OpenEmbedded framework [35]. OpenEmbedded was designed to provide all the required components to build a complete Linux-based operating system for embedded devices. The main included tools, the so called cross-compilers, are available for multiple popular processors often found in embedded systems. These special compilers can build a Linux distribution from a full featured desktop computer[3] but that is targeted to run on an

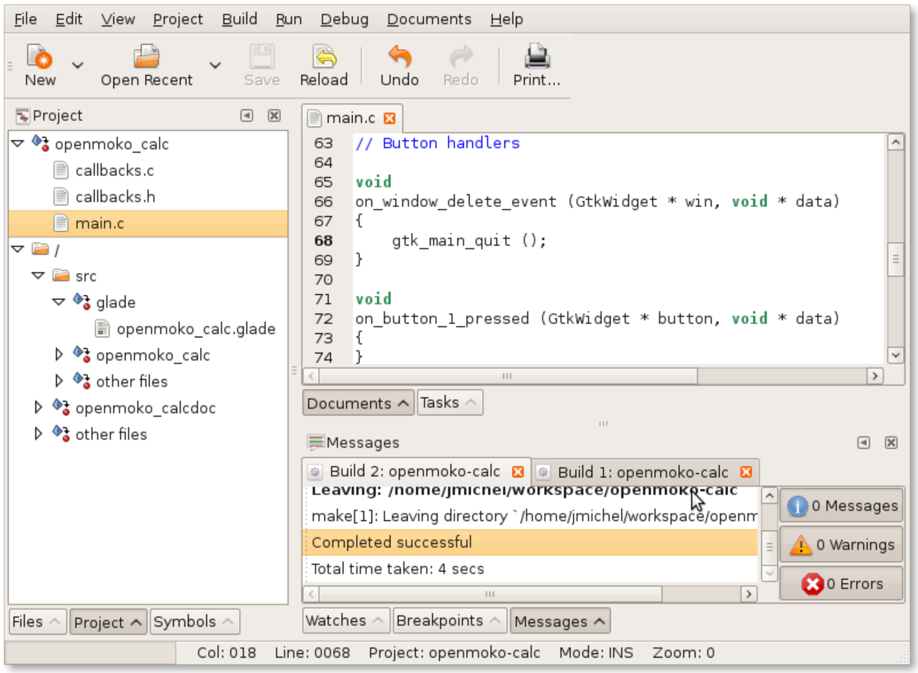
[3]The use of the term "desktop computer" here relates to any computer with the common x86 architecture.

openmokast.bb
DESCRIPTION = "Openmokast Broadcasting Receiver"
AUTHOR = "Jean-Michel Bouffard <jean-michel.bouffard@crc.ca>"
HOMEPAGE = "www.openmokast.org"
SECTION = "console/applications"
PRIORITY = "optional"
LICENSE = "GPL"
PN = "openmokast"
PV = "0.5"
PR = "r0"
EXTRA_OECONF += "–enable-gtk-gui –enable-dbus"
FILES_${PN} += "${datadir}/dbus-1/services/org.openmokast.Receiver.service"
SRC_URI = "file://openmokast-0.5.tar.gz"
inherit autotools

Table 5.2: BitBake definition file for the Openmokast software

embedded architecture. OpenEmbedded works by letting developers define the different specifications of the distribution to build. These specifications include which software must be built and included in the distribution and also how to build the kernel and applications to be compatible with the target platform.

Basic knowledge of the OpenEmbedded framework is required for application developers because it is also used to build the software for the device. A useful feature about OpenEmbedded is that only a single build command is required to configure, compile, build, test and package the software into an .ipk file ready to be installed on the FreeRunner. Log files are collected and saved for each step of the build for reference in case an error is encountered. To build a new application, a configuration file in a format used by OpenEmbedded must be created with the details about the new package. This file, called a BitBake recipe, is in fact read by the BitBake tool which will interpret the data and process the build tasks. The file *openmokast.bb* that defines the Openmokast software is shown in Table 5.2.



```
// Button handlers

void
on_window_delete_event (GtkWidget * win, void * data)
{
      gtk_main_quit ();
}

void
on_button_1_pressed (GtkWidget * button, void * data)
{
}
```

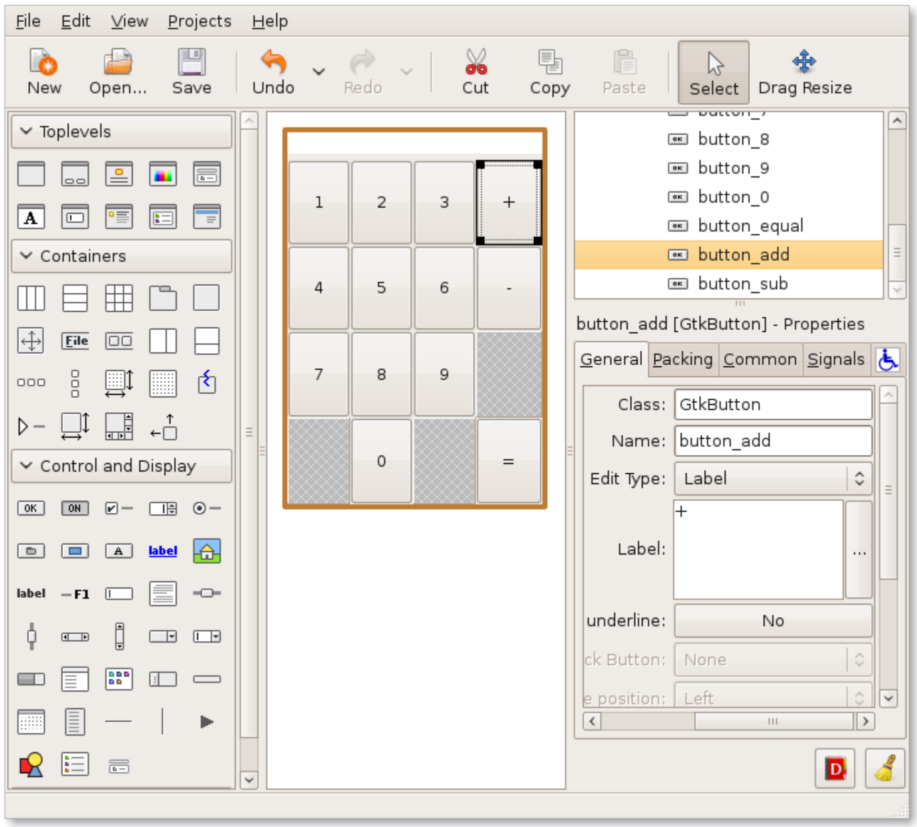
```
Leaving: /home/jmichel/workspace/openmoko-calc
make[1]: Leaving directory `/home/jmichel/workspace/openm
Completed successful
Total time taken: 4 secs
```

Figure 5.8: Anjuta Integrated Development Environment

5.5.2 Development Tools

If it is true that the build system is restricted to the OpenEmbedded framework, it is exactly the opposite for the development tools. In fact, almost any development technology that is available on Linux can be used to develop applications for the Openmoko devices. In the case of the Openmokast broadcasting software, because the user interface and the communication libraries used were part of the GNOME project, it was decided to give a try to the GNOME tools for C/C++ development, namely the Anjuta IDE (Integrated Development Environment) and the Glade Interface Designer. These tools offer convenient templates to start development based on GNOME libraries.

Anjuta, which is shown in Fig. 5.8, is a modern development environment that supports most features common in this type of software such as object browsing, code auto-completion and integrated debugging. A wizard can be used for the creation of



button_8
button_9
button_0
button_equal
button_add
button_sub

button_add [GtkButton] - Properties

Class: GtkButton
Name: button_add
Edit Type: Label
Label: +
underline: No
ck Button: None
e position: Left

Figure 5.9: Glade Interface Designer

a new project and it creates the template and the files required for an *autotools* based project, which is then directly supported by OpenEmbedded. The use of *autotools* can be advertised to OpenEmbedded by the usage of the ”inherit autotools” procedure as shown at the end of the BitBake recipe from Table 5.2. Moreover, because C/C++ development with the same libraries used on Openmoko is also supported on a Standard Linux computer, the application can be built, tested and debugged right from the IDE prior to be built for the Neo FreeRunner.

The Glade Interface Designer is a complement of the Anjuta IDE that lets the developer design and draw the user interface using a graphical point and click design window. The design environment is pictured is Fig. 5.9. Widgets, such as text fields, button, sliders, and so on, can be inserted into different types of layout that will control the interface look depending on the dimension of the window. The characteristics for each widgets can be specified. Moreover, for each possible interaction of the user on a widget, the name of a handling function must be defined. The different types of interaction vary from a widget to another, for instance, a button widget has the "clicked" action that can be connected to a handling function. The same name for the handling function has to be reused in the code to assure the right behavior of the software. The Glade Interface Designer, unlike most alternative solutions, produces XML files that describe the interface. The XML files are then loaded at runtime by the application to draw the interface on screen. The major advantage of this method is that the design of the interface can be modified without re-building the applications, at the condition that the names of the handler functions do not change. On the other side, a simple naming error, that would have been detected by the compiler in a traditional compiled interface, will generate a runtime error in a software that uses a Glade-based interface.

Chapter 6

Development

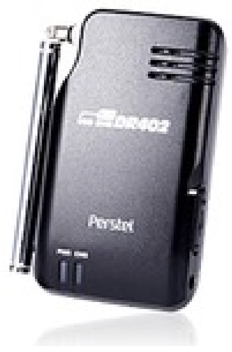
”With bitbake+autotools, you can build a boat on the cross-compiler ocean, and be done with it.”

– *Jay Vaughan.*

WITH the Openmoko hardware and software platform as a starting point, many components are missing to get a functional broadcasting receiver. The missing components that were identified are:

1. Compatible broadcasting receivers
2. Linux drivers for the receivers
3. Broadcasting stack to control the receivers and access the data
4. Applications to decode the multimedia streams

The Openmokast development effort was started to address these four elements and to create the first open source mobile broadcasting handset to date. The components were cleanly integrated into the Neo FreeRunner hardware and this resulted in the Openmokast prototype.


(a) Perstel DR402

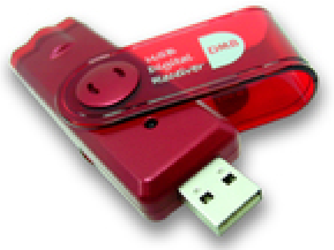

(b) Mtech UDR-A3L

Figure 6.1: DAB compatible USB receivers

6.1 Broadcasting Receiver

The receivers compatible with any mobile broadcasting technologies are not really common on the market and they do not necessarily support the physical connection from the phone. In fact, the only two practical connections to the Openmoko device are the USB port, which is accessible from the outside of the case, and the micro SD port, compatible with SDIO[1] (Secure Digital Input Output), which is located under the battery inside the case. Because SDIO is still a new format, it was not possible to find a receiver available on the market that would suite our needs. On the other side, some USB receiver were available. The Perstel DR402 is an older generation receiver available since the start of DMB service in Korea around 2005. The Mtech UDR-A3L is a newer receiver available since last year. Both are pictured in Fig. 6.1.

Both of these receivers have specifications that are compatible with the Canadian digital broadcasting standard shown in Table 6.1. They are then considered as candidates for the Openmokast receiver prototype.

[1]SDIO is used to connect peripherals to some mobile devices. Compatible peripherals are available today ranging from GPS receivers, Wi-Fi or Bluetooth adapters, modems, Ethernet adapters, barcode readers, IrDA adapters, and many more.

Specification	Value
Protocol	Fully compliant to ETSI EN 300 401 (Eureka147)
RF frequency range	L band: 1452 to 1492MHz
Transmission mode	All modes with auto detection
Channel decoding	Single channel decoding
Video service decoding capacity	Up to 1.5Mbps
Audio service decoding capacity	Up to 384Kbps

Table 6.1: Specifications of USB DAB receivers (related to Canadian DAB)

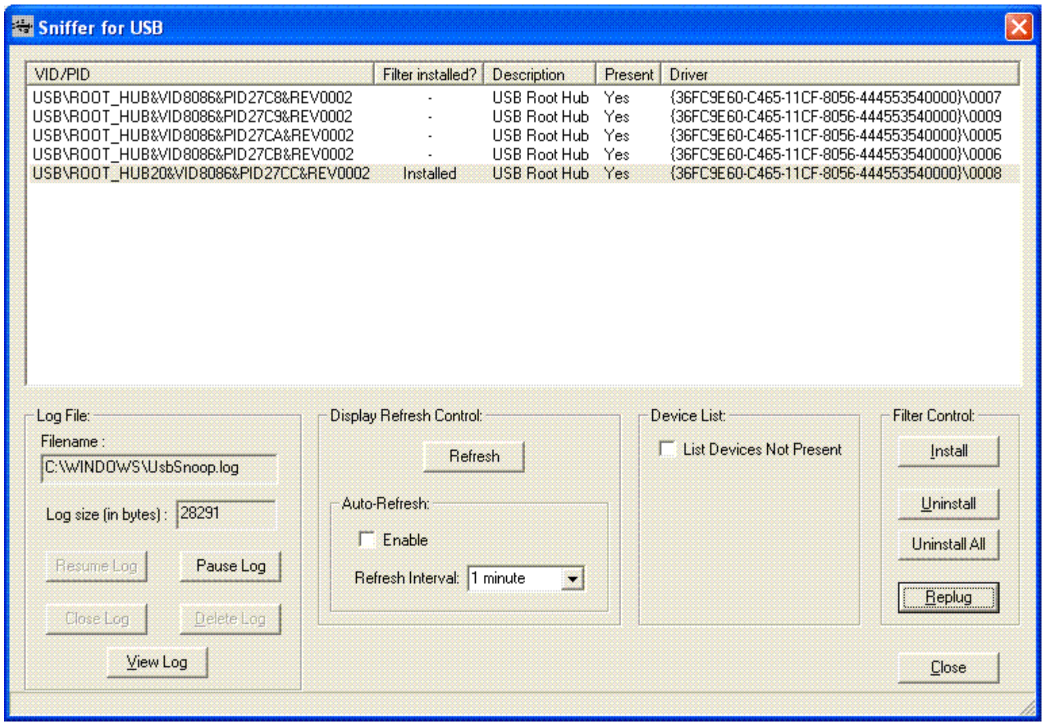
6.2 Linux Drivers

Both of the candidate receivers are available with proprietary drivers and software for Microsoft Windows^{TM} only. This is a major issue when considering that all of the open platforms described in Section 3.2, including Openmoko, are based on Linux. The implementation of a Linux driver was a mandatory component of the platform to build a functional prototype.

6.2.1 Probing of the USB Communication

As it is the case with most devices, the communication format that is used between the USB receivers and the host computer, which is running the proprietary drivers, is unknown and undocumented. To work around this issue, the communication needs to be observed with a tool called a USB sniffer. The tool *SniffUsb 2.0* [55] was used in conjunction with a Windows XP^{TM} computer and the proprietary driver and software of both receivers. Using this tool, log files of the communication between the driver and the device during normal usage of the receiver could be generated.

To generate these log files, a special driver, named as a filter throughout the application, must be linked to the USB driver of the device to monitor. The first thing to do is to launch *SniffUsb 2.0* and to look at the list of USB devices that are



VID/PID	Filter installed?	Description	Present	Driver
USB\ROOT_HUB&VID8086&PID27C8&REV0002	-	USB Root Hub	Yes	{36FC9E60-C465-11CF-8056-444553540000}\0007
USB\ROOT_HUB&VID8086&PID27C9&REV0002	-	USB Root Hub	Yes	{36FC9E60-C465-11CF-8056-444553540000}\0009
USB\ROOT_HUB&VID8086&PID27CA&REV0002	-	USB Root Hub	Yes	{36FC9E60-C465-11CF-8056-444553540000}\0005
USB\ROOT_HUB&VID8086&PID27CB&REV0002	-	USB Root Hub	Yes	{36FC9E60-C465-11CF-8056-444553540000}\0006
USB\ROOT_HUB20&VID8086&PID27CC&REV0002	Installed	USB Root Hub	Yes	{36FC9E60-C465-11CF-8056-444553540000}\0008

Log File:
Filename : C:\WINDOWS\UsbSnoop.log
Log size (in bytes) : 28291
Auto-Refresh: Enable
Refresh Interval: 1 minute

Figure 6.2: *SniffUsb 2.0* user interface

currently connected to the computer. Next, after connecting the USB receiver, a new device will appear in the list which will reveal the item that must be monitored. The *SniffUsb 2.0* interface, shown in Fig. 6.2, has the option to attach the special driver to the USB receiver by clicking "Install" in the right section named "Filter Control" while having the USB receiver line selected in the top list. It can be confirmed that the filter is installed correctly by looking at the "Filter Installed" column in the list of drivers. It is recommended, to simplify the log files reading, that a single log file is created for each action of the receiver. For instance, it is better to create a log file only for the action of starting the application since there are many chances that this will trigger the initialization of the device. A new log file can be created for each new action triggered by the application, such as when a new ensemble is tuned. To be able to recreate functional communication between the receiver and the Linux computer, communication must be logged for every functions used by the software. The list of calls to log includes:

- The initialization of the receiver
- Scanning to find all available frequencies
- Tuning to a specific frequency
- Collecting the information about available services
- Collecting the information about a specific service
- Starting a service or subchannel
- Reading data streams from the receiver
- Stopping a service or subchannel
- The deinitialization of the receiver

The log files contain massive amounts of information for each transfer to the USB port. A tool to filter the files is available and it was used to keep only the important communication elements. The resulting bistream was then inserted into the code of the tuner application. For instance, to command the Perstel DR402 DAB receiver to tune to the channel L18, which corresponds to a frequency of 1482.464 Mhz, the binary sequence "0x050002401600E09EF7DE" must be sent to the receiver on the USB port.

6.2.2 Reimplementation of the USB Communication

Different approaches can be used to implement a USB driver for the Linux operating system. The usual approach is to develop a kernel module based on the Linux USB subsystem. This method is used to create a real kernel-mode Linux driver that can

Characteristics	Kernel-mode driver	User-mode library (Libusb)
Complexity	Kernel development under Linux can be complex and involves many specific requirements.	Libusb can be used as any other user-mode library under Linux and offers a well-documented API.
Reliability	Kernel development is subject to system crash if an error occurs.	Libusb is built over a reliable generic USB driver that is usually not crashing the system when errors occur.
Flexibility	USB communication is not limited in any way.	USB communication can only use the API implemented by Libusb.
Compatibility	Kernel-mode drivers are supported by virtually any Linux based systems.	Libusb is available on most, but not all, Linux embedded systems.

Table 6.2: Comparison of USB development solutions for Linux

be loaded or unloaded at runtime. The other approach is by using a user-mode USB control library such as libusb [56]. The two solutions are compared side by side in Table 6.2. The Libusb library was selected for this part of the project to reduce development complexity and to optimize reliability. Some embedded system may miss Libusb support but, currently, all platforms mentioned in Section 3.2, including Openmoko, feature full support for the library.

By using the Libusb API [56], every necessary commands were implemented in the Openmokast receiver to support DAB receiver control. The communication of both receivers, the Perstel DR402 and the Mtech UDR-A3L, were reimplemented using this method. After testing the resulting performance, it was found that the Perstel was able to tune and to connect to a service but the connection was always suddenly dropped for no apparent reason. On the other side, the Mtech receiver is working reliably, so it was chosen as the receiver for the Openmokast prototype. No more work was done to correct the problem encountered with the Perstel receiver to date.

6.3 Broadcasting Stack

6.3.1 Architecture

Now that the hardware control part is functional, resulting in a functional DAB USB receiver, the broadcasting software stack is the next component in the chain. The components previously mentioned in Chapter 5 were integrated in a new architecture as shown is Fig. 6.3. This enhanced architecture brings many improvements to the framework. The software components were all developed in the C++ programming language.

The available inputs, on the left part of the framework, are available as dynamically loadable libraries. These components can then be compiled, built and distributed separately to offer more flexibility. The main advantage of this feature is that new inputs could be developed and distributed by third party developers. When combined with the framework, these inputs would be loaded automatically and used as any other type of input. Broadcasting receiver manufacturers usually do not want to reveal the APIs of their hardware, thus they are releasing closed-source drivers that obfuscate the low-level communication. Hence, the capability to dynamically load a closed source input was seen as an incentive for manufacturers to participate in the project by releasing compatible drivers.

Technically, dynamic object loading is provided by the *dlopen* C++ API. The API enables to load a Linux shared object library at runtime inside the application instead of being loaded by the operating system when the application starts. The classes and methods can be linked from the application and called as if they were built together. The important requirement is that the dynamically loaded object must support the exact same interface that the application expects to see. Otherwise the load operation fails and the new input is not usable.

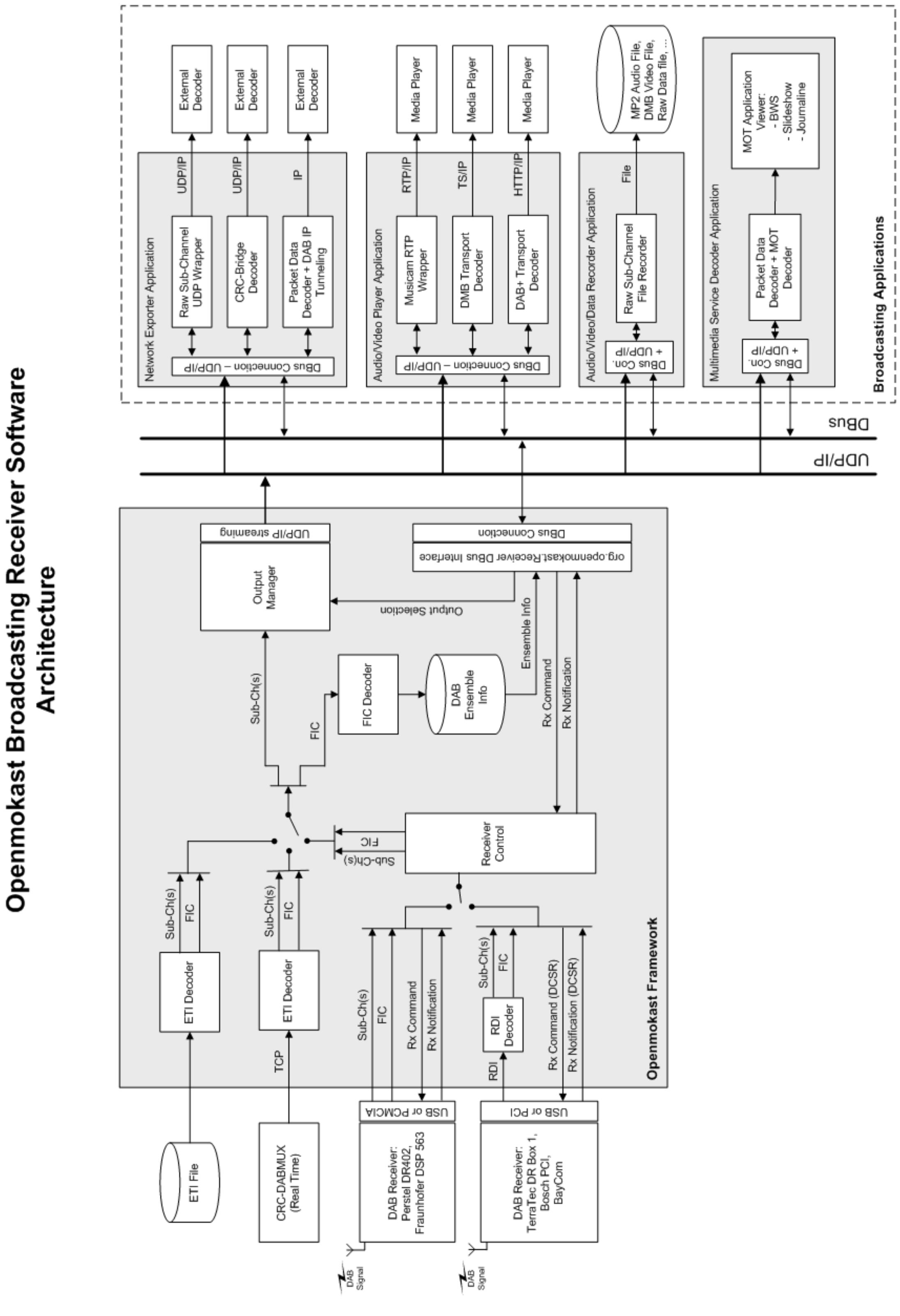

Openmokast Broadcasting Receiver Software Architecture
ETI File
CRC-DABMUX (Real Time)
DAB Receiver: Perstel DR402, Fraunhofer DSP 563
DAB Receiver: TerraTec DR Box 1, Bosch PCI, BayCom
DAB Signal
USB or PCMCIA
USB or PCI
TCP
RDI
ETI Decoder
RDI Decoder
Sub-Ch(s)
FIC
Rx Command
Rx Notification
Rx Command (DCSR)
Rx Notification (DCSR)
Receiver Control
FIC Decoder
DAB Ensemble Info
Ensemble Info
Output Selection
Output Manager
UDP/IP streaming
org.openmokast.Receiver DBus Interface
DBus Connection
Openmokast Framework
UDP/IP
DBus
Network Exporter Application
DBus Connection – UDP/IP
Raw Sub-Channel UDP Wrapper
CRC-Bridge Decoder
Packet Data Decoder + DAB IP Tunneling
IP
External Decoder
Audio/Video Player Application
Musicam RTP Wrapper
DMB Transport Decoder
DAB+ Transport Decoder
RTP/IP
TS/IP
HTTP/IP
Media Player
Audio/Video/Data Recorder Application
DBus Con. + UDP/IP
Raw Sub-Channel File Recorder
File
MP2 Audio File, DMB Video File, Raw Data file, ...
Multimedia Service Decoder Application
Packet Data Decoder + MOT Decoder
MOT Application Viewer: - BWS - Slideshow - Journaline
Broadcasting Applications

Figure 6.3: Architecture of the Openmokast software

Openmokast API method listing	
Tune	GetServiceArray
Scan	GetComponentArray
ScanAll	GetTransportMode
GetFrequency	GetDataComponentType
GetStatus	GetUserApplicationType
IsDecoding	GetReceiverCapabilities
StartDecoding	GetAvailableReceivers
StopDecoding	SelectReceiver
GetEnsemble	SelectOutputType

Table 6.3: List of methods of the Openmokast API

The next part of the architecture is the actual Openmokast framework. The framework is using the core of the CRC-DABRMS software described in Section 5.1. The main code that was reused is related to the DAB protocol, for instance, the ”Receiver Control”, ”FIC decoder” and ”Output Manger” blocks were included in the Openmokast framework. The target use of the framework is a mobile broadcasting stack for mobile Linux distribution. Such a stack is meant to be used as a middleware layer between the receivers and the applications that want to access the broadcast services. For that matter, the interface needs to be usable by other applications.

6.3.2 Application Programming Interface

Chapter 5 describes the available components that were considered for the Openmokast implementation and the factors that had an influence in the choice. As it was mentioned in Sections 5.2 and 5.4, the API included in Openmokast is based on ViaDAB [49] and it is build using the D-Bus [57] inter-process communication mechanism. The set of methods from the ViaDAB control interface, that were shown in Table 5.1, was reduced to keep only the necessary elements.

To produce an API that can be general enough for different broadcasting technologies, the elements related to DAB (*SubscribeFIC, SubscribeMSC, Unsubscribe,*

Openmokast API signal listing
ensemble_update_notify

Table 6.4: List of signals of the Openmokast API

GetDABLocalTimeOffset, GetDABTime and GetDLS) were removed. The functionalities meant to control the external player (*GetVolume, SetVolume, StartPlaying and StopPlaying*) were also remove because the service decoding will only be done in software in the context of Openmokast. Moreover, to assure a flexible usage, some method were added to the Openmokast API to support multiple broadcast receivers at a time. Where, in ViaDAB, multiple calls were required to get the identifier and label for each ensemble, service and component, Openmokast needs only one method. This single method returns multiple data elements, thanks to the flexible return types that are possible with D-Bus. This simplifies the task for third party developers by requiring less programming to achieve the same result. The Openmokast methods are synchronous calls, hence the calling application will block until the reply is received. Another type of D-Bus message, called a signal, is used to provide a way to advertise the modification of the current ensemble to the application. D-Bus signals are sent asynchronously to applications that have previously registered to receive them. The list of Openmokast methods is shown in Table 6.3 and the signal is in Table 6.4. The complete API documentation is available in Appendix A.

The resulting *org.openmokast.Receiver.Control* interface was defined with the goal to support the main uses of broadcasting services from a third-party application. D-Bus offers interesting mechanisms to support the generation of a new D-Bus object. The interface can be defined with an easy-to-use XML-based format and the *dbus-binding-tool* will take care of header files generation. The new files automatically generated from the original XML file are listed in Table 6.5. The server header file is used inside the Openmokast framework software and all the functions it defines must

File	Description
org.openmokast.Receiver.xml.in	Original XML description of the D-Bus object.
org.openmokast.Receiver.server.h	Header file required for the implementation of a server of the D-Bus object.
org.openmokast.Receiver.client.h	Header file required for the implementation of a client of the D-Bus object.
org.openmokast.Receiver.html	HTML documentation (see Appendix A) generated with a tool provided by [27].
org.openmokast.Receiver.xml	Raw XML description of the D-Bus object.
org.openmokast.Receiver.service	User generated file requirement by D-Bus to associate a D-Bus service with the application that is providing the service.

Table 6.5: List of D-Bus interface related files

be implemented. The client header file can then be added to projects that will use the *org.openmokast.Receiver.Control* interface. Then, each call to the Openmokast server is represented by a simple function from the client side.

D-Bus was created mostly for message exchange between applications, hence it was not designed for high bitrate applications. For this reason, D-Bus will be used for controlling the broadcasting software stack but the data streams will use the networking stack of the operating system, which is more efficient for high bitrate transfers. When a client application is requesting the data from a broadcast service, the Openmokast framework will start sending the data to a free UDP/IP port and will return the specification of the port to the client. The format of this API command is shown here in a sample of the Openmokast API documentation:

StartDecoding (uu) -> sb

Description: Starts the decoding of a component.

Parameters

u: service_id

The service identifier.

u: component_id

The component identifier.

Returns

s: uri_ret

The URI of the newly started component.

b: ret

True if the command succeeded. False if the component doesn't exist or if it cannot be decoded.

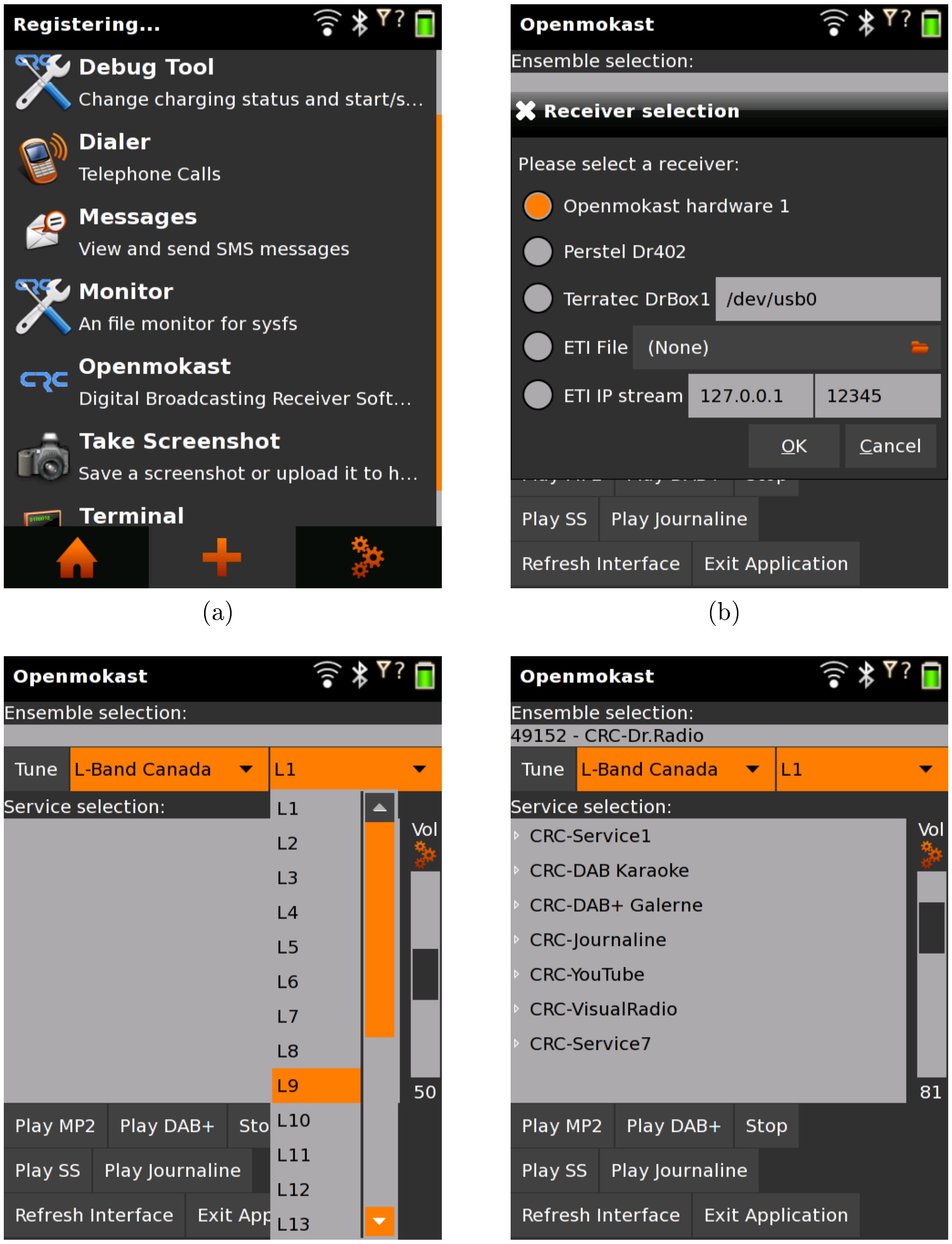
The *StartDecoding* method takes two parameters, the identifiers for the service and for the component to start. After the execution, the method will return a string containing the URI (Uniform Resource Identifier) to the data stream and the indication of the success of the call. D-Bus uses a simple way to identify data types using a single letter. In this case, the letter *u* is used for unsigned integer, *s* for string, and *b* for a boolean value.

6.3.3 User Interface and System Integration

The Openmokast framework is build with external libraries that provide mechanisms for easier development. The libraries that are used to provide the D-Bus API and the graphical user interface, are:

libGTK+ A graphical user interface toolkit.

libDBUS The D-Bus library required for every applications that uses the messaging system.



Registering...

Debug Tool
Change charging status and start/s...

Dialer
Telephone Calls

Messages
View and send SMS messages

Monitor
An file monitor for sysfs

Openmokast
Digital Broadcasting Receiver Soft...

Take Screenshot
Save a screenshot or upload it to h...

Terminal

(a)

Openmokast
Ensemble selection:

Receiver selection

Please select a receiver:

[✓ Openmokast hardware 1]
Perstel Dr402
Terratec DrBox1 /dev/usb0
ETI File (None)
ETI IP stream 127.0.0.1 12345

OK Cancel

Play SS Play Journaline
Refresh Interface Exit Application

(b)

Openmokast
Ensemble selection:
Tune L-Band Canada L1
Service selection:
L1 L2 L3 L4 L5 L6 L7 L8 L9 L10 L11 L12 L13
Vol 50
Play MP2 Play DAB+ Sto
Play SS Play Journaline
Refresh Interface Exit App

(c)

Openmokast
Ensemble selection:
49152 - CRC-Dr.Radio
Tune L-Band Canada L1
Service selection:
CRC-Service1
CRC-DAB Karaoke
CRC-DAB+ Galerne
CRC-Journaline
CRC-YouTube
CRC-VisualRadio
CRC-Service7
Vol 81
Play MP2 Play DAB+ Stop
Play SS Play Journaline
Refresh Interface Exit Application

(d)

Figure 6.4: Openmokast framework screen captures

libglib A library that provides many tools for application development, one of them being an object oriented development framework in C. The framework is used to define the D-Bus interface.

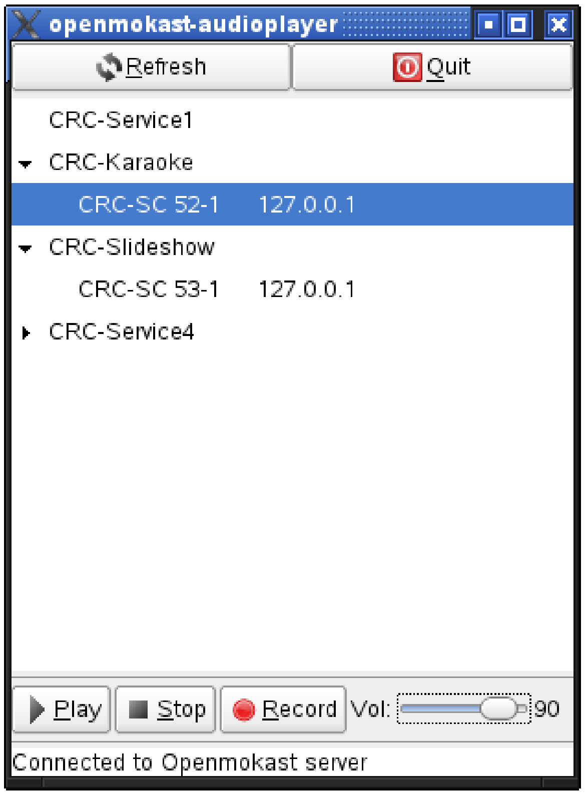
libglib-dbus A set methods compatible with libglib which wraps the D-Bus library. This enables the applications to use libDBUS with the object oriented development framework offered by libglib.

The user interface of the Openmokast framework application, running on an Openmoko FreeRunner, is pictured in Fig. 6.4. The screen (a) is the desktop of the phone. The screen (b) is the welcome dialog where the user is choosing the type of receiver that will be used. Screens (c) and (d) feature the tuning and ensemble information screen.

Another interesting feature of D-Bus is that the system always knows which application provides which service. So at the moment when an application makes a request for the broadcasting interface, the D-Bus system will first check if the provider of the service is available on the system. If not, it will then try to start the application that is registered to provide this service. The link between the service and the provider are in a file that must be installed on the system with the service provider application. For that matter, the file *org.openmokast.Receiver.service* is installed with the Openmokast framework:

```
[D-BUS Service]
Name=org.openmokast.Receiver
Exec=/usr/local/bin/CrcDabRms
```

After these steps are completed, the D-Bus system will be able to respond to the client application if the connection was established successfully or not.



Connected to Openmokast server

Figure 6.5: Openmokast-audioplayer client application

6.4 Multimedia Applications

If we go back to the Openmokast framework architecture from Fig. 6.3, the multimedia applications are located on the right part of the D-Bus interface. These applications are clients to the Openmokast framework that can request access to broadcasting services through the *org.openmokast.Receiver* D-Bus service. The next sections will describe some clients that were built in the course of the project.

6.4.1 Broadcast Radio Player

To begin with, a simple audio player application was developed to validate the functionality of the Openmokast framework. The openmokast-audioplayer, depicted in Fig. 6.5, is reading the available services from the currently tuned ensemble and it filters them to only show the audio services. The user is then able to play any of these audio streams.

The audio playout was implemented with the GStreamer multimedia framework presented in Section 5.3.2. One of the GStreamer plugins called ”playbin” has the features of a complete audio application. Among other things, it can access network streams directly and it will even play it over the sound hardware available on the machine. GStreamer automatically detects the type of media that is streamed to the application and it will select the right combination of plugins to achieve the decoding task. However, the user needs to take care of installing the right plugins for the media format. In the case of standard DAB radio, a regular mpeg2 audio decoder is required.

6.4.2 Template for Client Application Development

The next sub-project that was developed is the openmoko-client template application. This client application is a simple command line tool that connects to the *org.openmokast.Receiver* D-Bus service and requests the available services. It is meant to be used as a template for the development of new clients by third party developers. The demonstrated connection mechanism to the D-Bus API and examples of function calls is all that is required to be able to add broadcasting to a new application or to integrate it inside an already existing one.

The template application code is available in Appendix B. The application includes the header file *org.openmokast.Receiver.client.h* which is required to use the client D-Bus service interface. This auto-generated file defines all the methods that can be called by the client applications. More information about the usage of the template in a new software project are included in the development use case presented in Chapter 7.

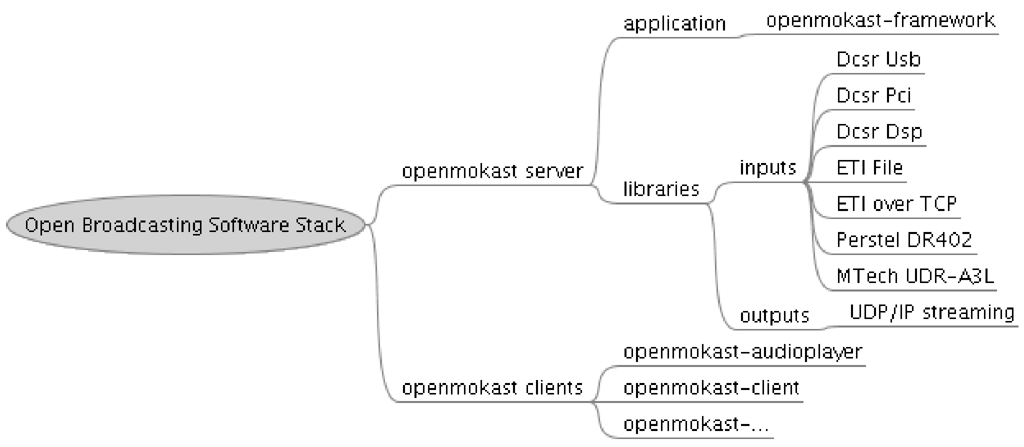


Figure 6.6: Openmokast project structure

Fig. 6.6 shows the structure of the different applications and libraries developed in the course of the project. The Openmokast server contains a subproject for each of its input and output plugins. The client applications are the one described in this section and the template is the starting point for the new clients that will be implemented by third party developers.

6.5 Hardware Integration

For the demonstration of the Openmokast project, the software and hardware components were integrated into a working prototype. The hardware integration is somewhat outside the scope of this thesis but it is interesting to note how the democratization of design techniques is also happening outside the field of software engineering. To produce this prototype, the idea was to insert the USB receiver hardware inside the case of the Openmoko FreeRunner. However, even after removing the plastic casing of the receiver and any other unnecessary parts, for instance the USB connector, it was still much larger than the available space inside the FreeRunner's case.

To address this issue, an extension to the plastic casing of the FreeRunner was designed based on the CAD (Computer-Aided Design) files released on the Openmoko

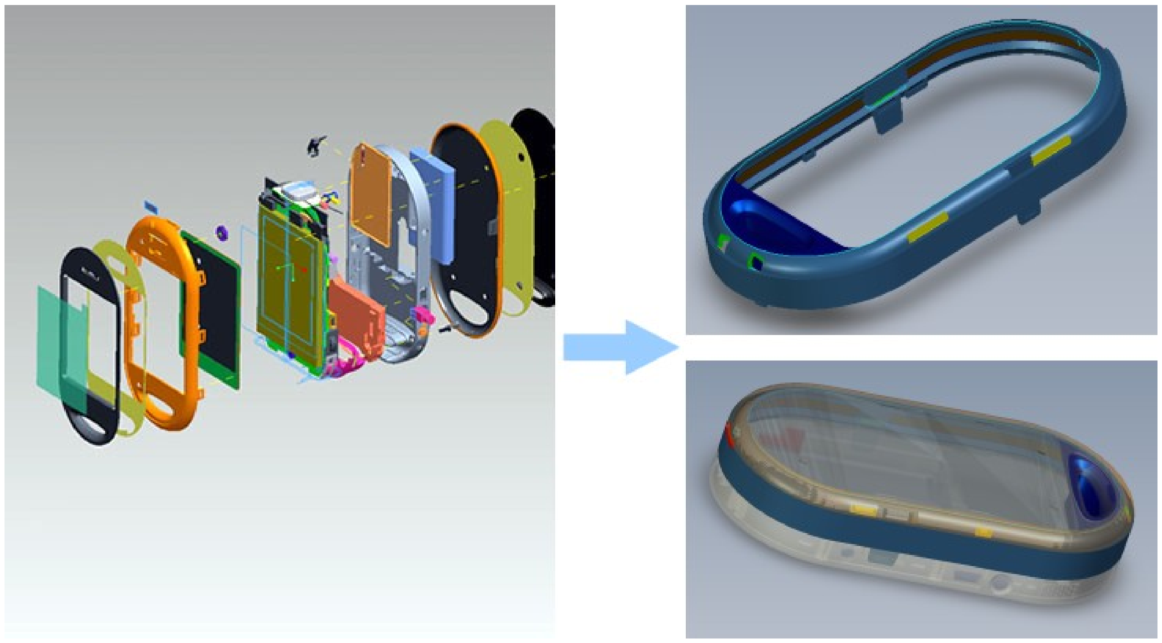


Figure 6.7: Modification of Openmoko CAD files

website [5]. These original CAD files were stored in the Pro/ENGINEER format, a 3D CAD software commonly used in the industry. From these files, it was possible to generate an extension that can be inserted between the phone and its back cover. The extension can be fixed securely to the phone, thanks to the exact same clips that were copied from the original CAD files. Fig. 6.7 shows a representation of the original 3D design and the resulting extension.

From the new 3D CAD representation of the plastic extension, the actual model was ordered from a local company that provides the service of an ABS plastic 3D printer. Pro/E models are a well-known industry standard, hence it was easily usable with the 3D printing equipment. The resulting model was a perfect fit in its first iteration and it was later modified slightly to make the clips less fragile. The resulting Openmokast prototype is shown in Fig. 6.8.

The final step towards achieving a seamless hardware integration was the connection of the Mtech receiver from inside the device. Again, thanks to the openness of

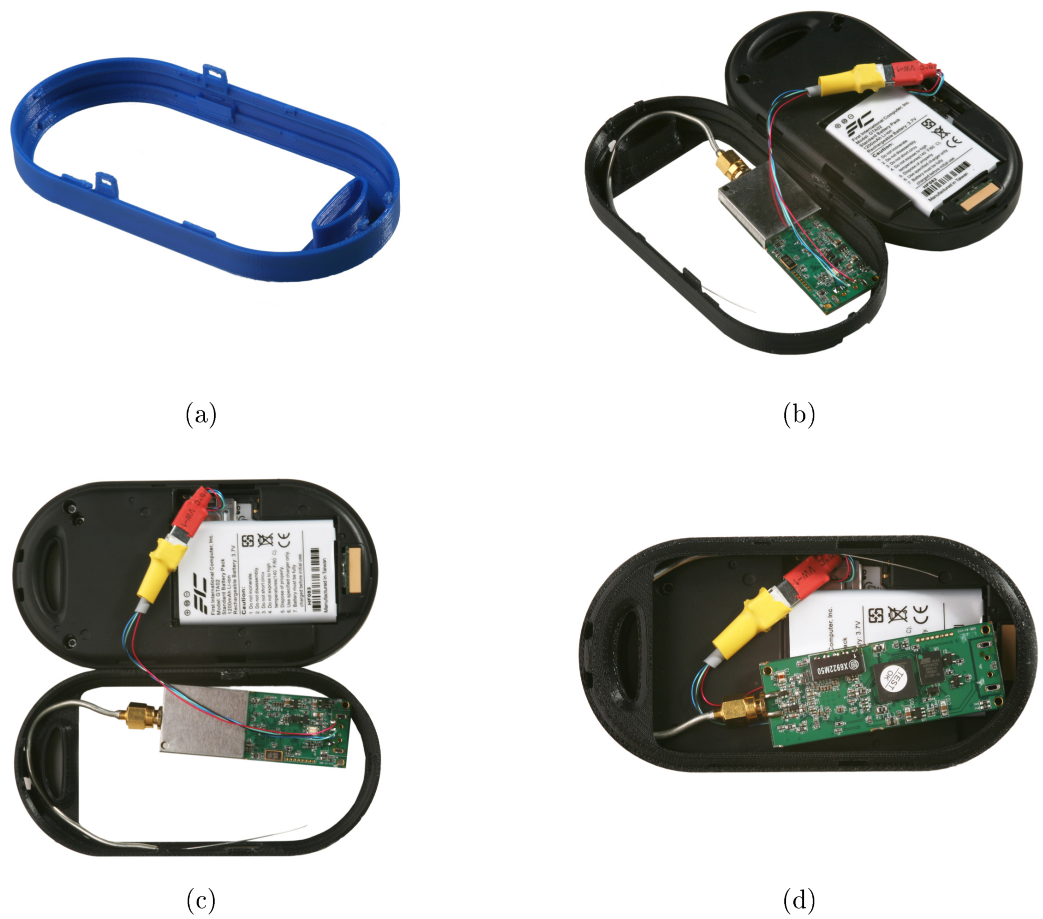
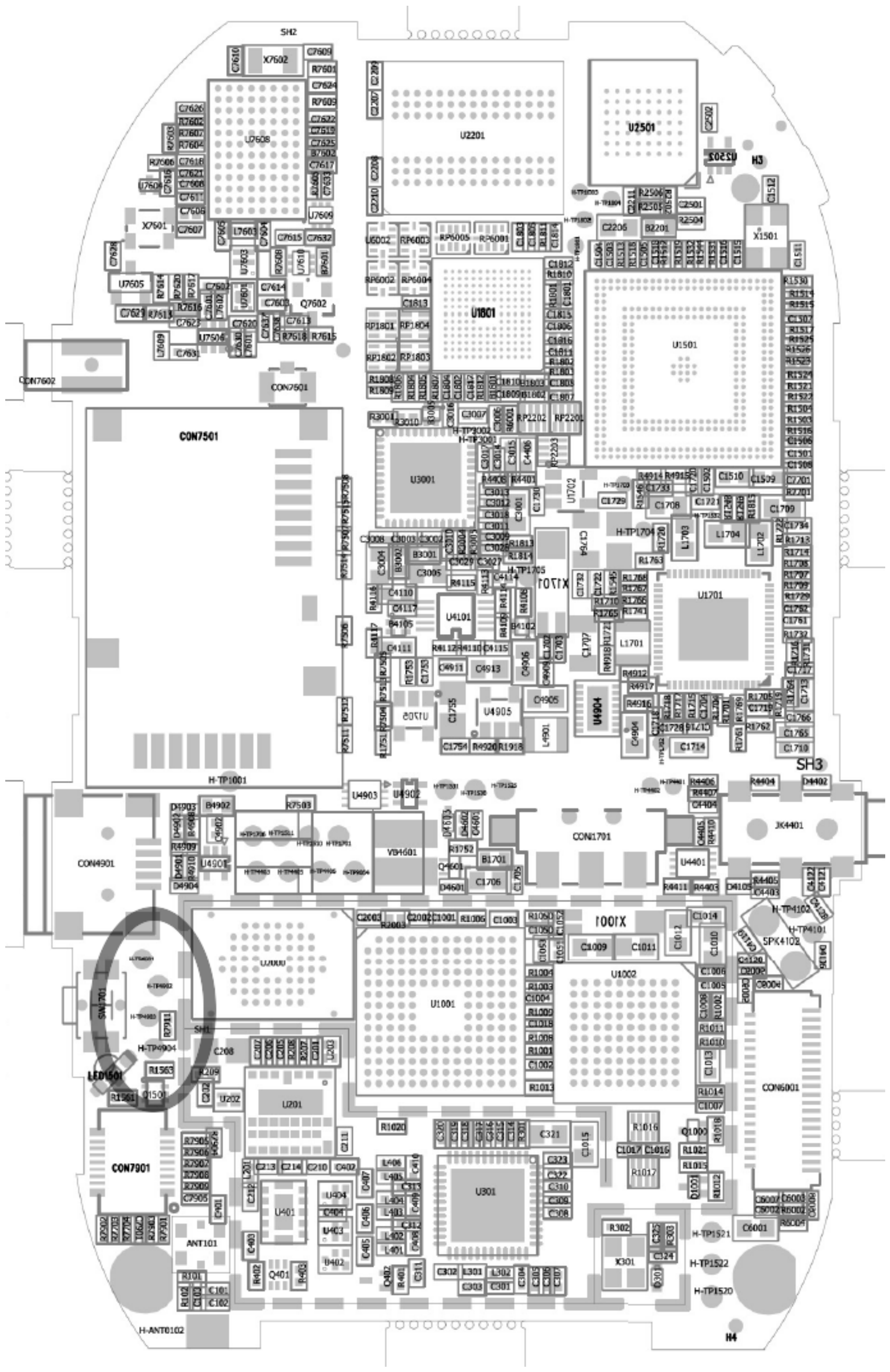


Figure 6.8: Pictures of the Openmokast prototype



Source: [5]

Figure 6.9: FreeRunner's electronic schematics with highlighted USB test points

the Openmoko platform, it was possible to have access to the schematics of the electronics of the Neo FreeRunner. USB test points for the side USB port of the device were identified on the schematic as depicted in Fig. 6.9. These test points were used to solder a mini USB connector from inside the case and complete the integration of the Openmokast prototype.

Chapter 7

Case Study

"For most user organizations the question is not whether they will adopt open source but when and where."

– *Ovum Research (2006).*

THIS chapter will explain how to build an application based on a broadcasting technology by showing a concrete example. The standard DMB video protocol is using a special format of H.264 video that is not used in any other implementation, hence, the tools required to generate the stream and to decode it are not widely available. This may have contributed to the slow adoption of the technology. With the Openmokast framework, it becomes easy to produce broadcasting video with standard tools. A test video service was generated and transmitted over the air. Then, a receiver application was build to connect to the Openmokast server, read the available services and decode the audio/video stream. The next sections explain the details of this case study.

DAB subchannel	Audio stream	Video stream
MPEG-TS	MPEG-1 Audio Layer 3	H.264
544 kbps	96 kbps	384 kbps
-	48 khz	368 x 246
-	Stereo	29,97 fps

Table 7.1: Specifications of the test video service

7.1 Generation of a New Broadcasting Service

The work described in this thesis is mostly related to the receiver side of broadcasting. This section describes how to generate a new broadcast service to be transmitted over the air. The service used in the case study is a video stream comparable to DMB video but generated using available free and open source tools. This has the disadvantage of not being compatible with commercial DMB equipment but, on the other side, it uses generic audio/video encoders that are commonly available in free and open source projects. The stream is composed of H.264 video and MP3 audio which are multiplexed in an MPEG-TS (MPEG Transport Stream). Table 7.1 shows the specifications of the service. It can be seen that it is transported in a 544 kbps subchannel even if the total bitrate of the content is 480 kbps. This security margin is necessary because the encoders, which are not specific to broadcasting, do not produce perfectly constant bitrate streams. On the other side, the production cost of such a broadcast video service would be a fraction of the real DMB because it does not require the expensive DMB-specific equipment.

The diagram of the service is shown in Fig. 7.1. The video content is taken directly from a DVD and it is encoded and multiplexed into a MPEG-TS stream using the VLC [58] video encoder/player. VLC is a very flexible audio/video software that supports a lot of formats and, moreover, it can transcode to different other formats using a collection of encapsulation protocols for streaming over a network or for writing files. The next part in the diagram uses this stream and inserts it into a

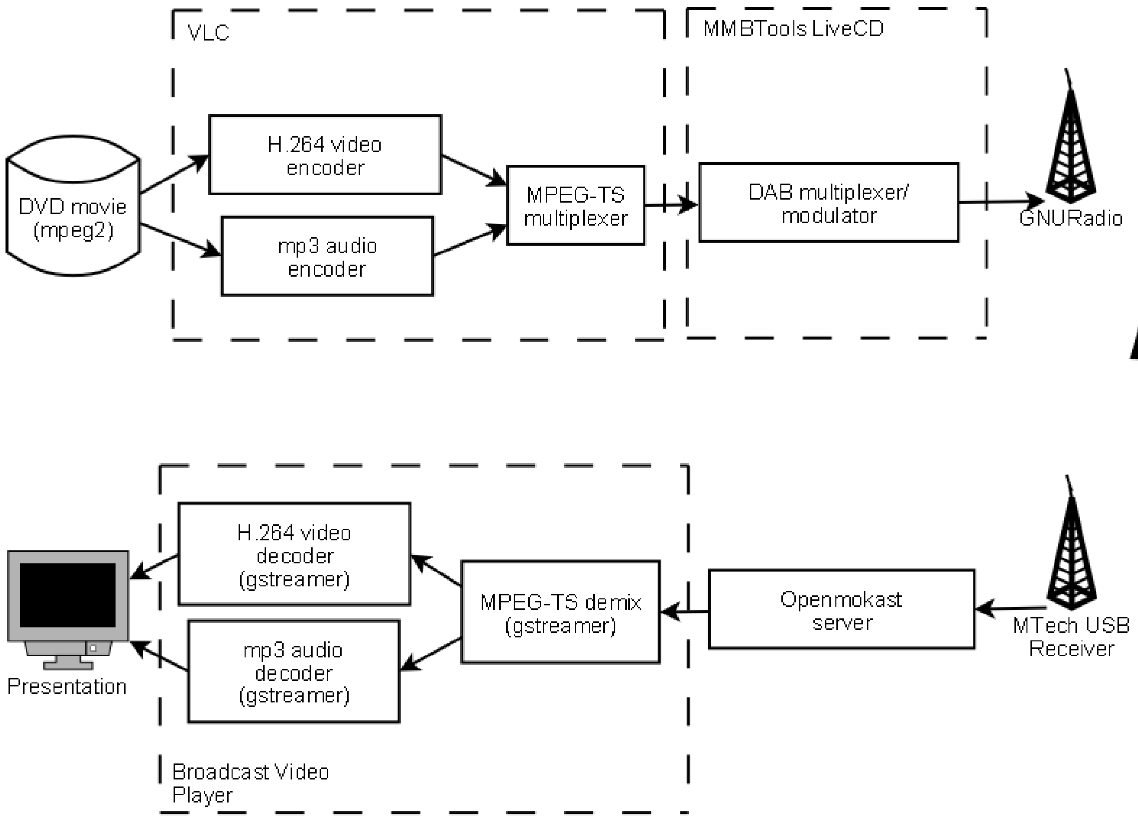


Figure 7.1: Block diagram of the broadcast video service used in the case study

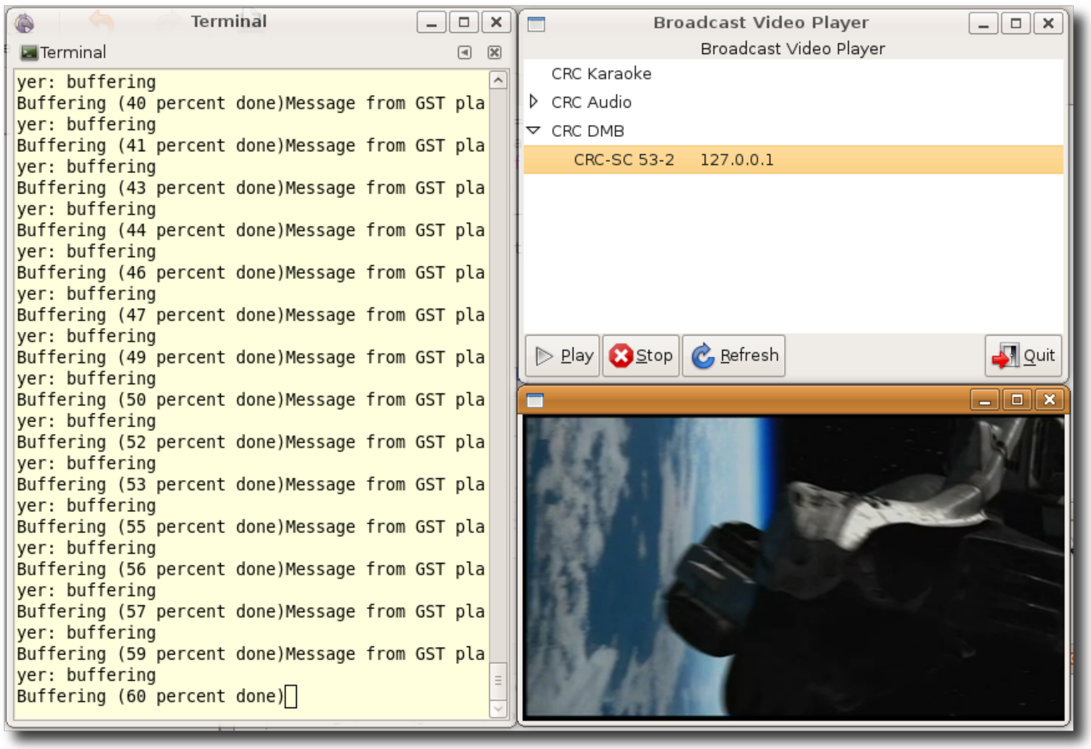
DAB compliant multiplex. This can be achieved with different software available on the MMBTools LiveCD [59] project. This LiveCD, that was developed at the CRC, offers an integrated solution for DAB multiplexing and modulation of about any type of content. Even non-standard services can be broadcast using these tools. The MMBTools LiveCD can use a USRP (Universal Software Radio Peripheral) board from the GNURadio [60] project to transmit the resulting signal over an antenna.

The receiver is depicted in the lower part of Fig. 7.1. The Openmokast server can use the Mtech USB receiver to tune to the broadcast video service. Then, only a client application is missing to control the server and to access the multimedia content. The next section will introduce the development of the client application.

7.2 Implementation of the Receiver Application

To develop a client application that uses the Openmokast server to access the video stream, the best starting point is the application template that was mentioned in Section 6.4.2. The new Broadcast Video Player application was started from this template. Most development environments offer the option to create a new project from existing code, and the Anjuta environment, as mentioned in Section 5.5, is no exception. The template includes the code to connect to the Openmokast server and to read the information about the available services. Some code needed to be added to present the information to the user and to integrate the multimedia capabilities required for the playout of the content. The Openmokast API is used to start and stop the services as the user presses the *Play* and *Stop* buttons. The GStreamer multimedia framework, as described in Section 5.3.2, is integrated in the new client application to handle the audio/video data. The new application was simply named ”Broadcast Video Player”.

The development of the video client application took only a single day, thanks to the help of the Openmokast client template project. A small amount of code, in the order of 340 lines of C, had to be written to handle the buttons of the user interface and to show the available broadcast services on the screen. As it is the case with the development environment, it is more practical to launch and test the applications from the host development computer than from the device itself. For these tests to work, the application must be built with the default compiler from the host system, without any use of the cross-compiler environment. Then, a regular debugger application, such as the GNU Debugger, can be used on the host computer. After the functionality of the application is validated on the host, it can then be built with the OpenEmbedded cross-compiler environment for the FreeRunner device. When the application package is ready, it has to be transferred on the device and installed with the *opkg* package



```
yer: buffering
Buffering (40 percent done)Message from GST pla
yer: buffering
Buffering (41 percent done)Message from GST pla
yer: buffering
Buffering (43 percent done)Message from GST pla
yer: buffering
Buffering (44 percent done)Message from GST pla
yer: buffering
Buffering (46 percent done)Message from GST pla
yer: buffering
Buffering (47 percent done)Message from GST pla
yer: buffering
Buffering (49 percent done)Message from GST pla
yer: buffering
Buffering (50 percent done)Message from GST pla
yer: buffering
Buffering (52 percent done)Message from GST pla
yer: buffering
Buffering (53 percent done)Message from GST pla
yer: buffering
Buffering (55 percent done)Message from GST pla
yer: buffering
Buffering (56 percent done)Message from GST pla
yer: buffering
Buffering (57 percent done)Message from GST pla
yer: buffering
Buffering (59 percent done)Message from GST pla
yer: buffering
Buffering (60 percent done)
```

Figure 7.2: Video player application connected on the Openmokast framework

manager.

Some tests were conducted with the complete transmission and reception chain. Fig. 7.2 shows the interface of the Broadcast Video Player software that is playing the content of the service named ”CRC DMB” on the host development computer. The client application was able to play the audio and video streams with little effort on a the host. The quality of the broadcast is considered relatively high because it uses the full framerate of the DVD but at the lower resolution of 368 pixels by 246 pixels. This lower resolution would, however, be considered as high quality when consumed on a smaller mobile device’s screen. This service could be received by thousands of people at a time without any quality loss because of the broadcasting technology used. When compared with current mobile television offerings on cellular phone networks, where the framerate is usually limited to save on bandwidth, broadcasting has a real

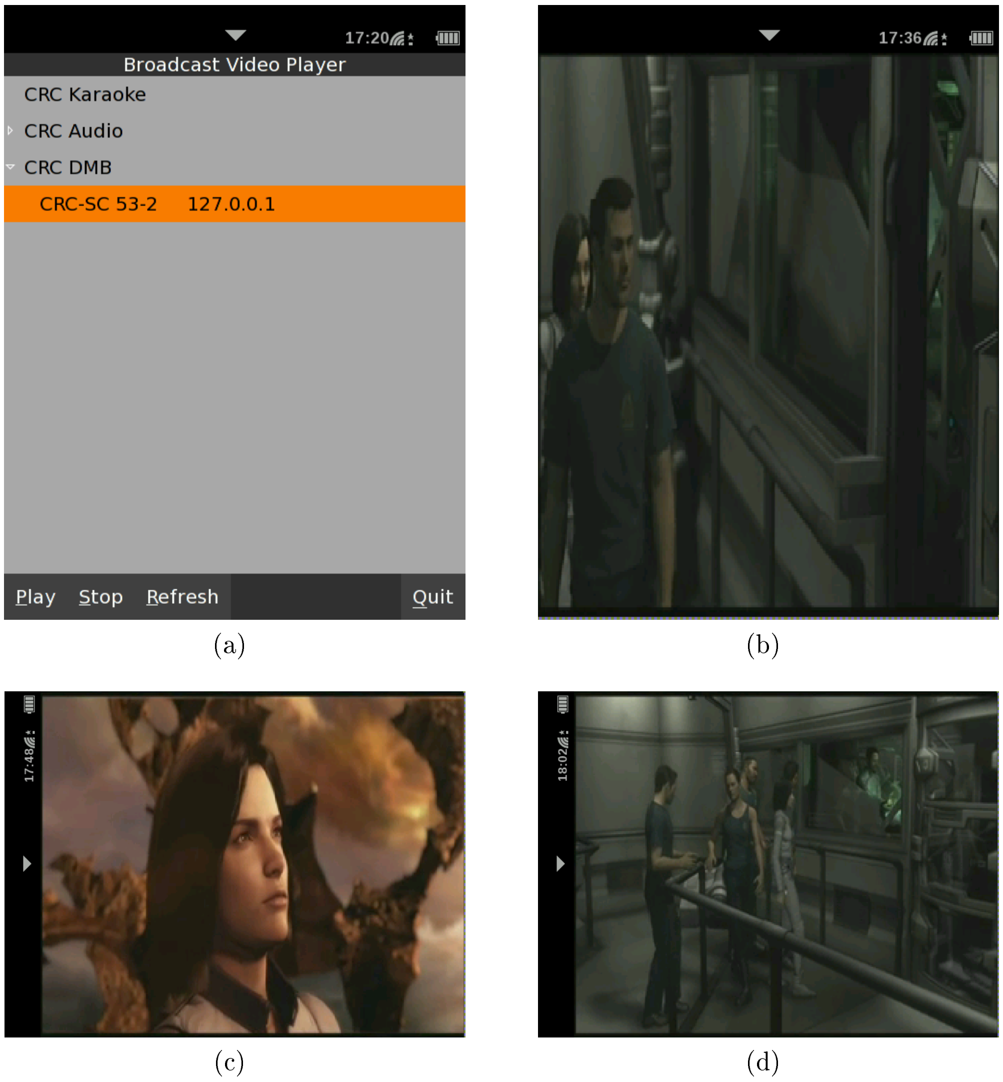


Figure 7.3: Video player application running on the FreeRunner

advantage.

After validation that everything is running as expected on a Linux computer, the application was deployed on the FreeRunner for further testing. The ”Broadcast Video Player” application is shown running on that target platform in Fig. 7.3. The features of the client are all running as expected. The only issue is related to the performance of the video decoder. In fact, the FreeRunner device was only able to achieve a very limited framerate, between 2 and 3 FPS (Frames Per Second), from a source video normally running at 29.97 FPS. To identify the cause of this performance issue, the two execution environments were compared. The possible problematic elements are highlighted and discussed in the following paragraphs.

Hardware USB receiver. The hardware receiver was not considered as a possible issue because the same Mtech device could be used on both the full featured Linux computer and the FreeRunner.

USB port. A second execution test was then conducted, both on the Linux computer and on the FreeRunner, with the broadcast video service accessed locally from a file. In this case, the resulting performance of the video playout was the exact same. Following this test, the USB port could be removed from the list of possible bottleneck.

Internal communication bus. The internal communication bus is known to be a possible performance problem of the Neo FreeRunner. As mentioned in their wiki [5], the device's graphic accelerator and the SD card reader are using a shared bus that is capable of a limited 7 Mb/s. When receiving a live stream, the SD card is mostly unused so the whole bus is dedicated to the video component. We can evaluate the amount of data that would be required to show the video with the following equation. It takes into consideration the resolution, the framerate and the color level of the video, which is of at least 16 bits per pixels:

$$368 \times 246 \times 29.97 \times 16 \approx 43\, Mb/s$$

The result shows that the graphical bus of the FreeRunner is probably overloaded when trying to show a fullscreen video like it is done in this case study. Some hardware video acceleration is available on the device and it could reduce the bandwidth required on the bus. However, this feature was not used by our decoding library.

	Test platforms	
Running process	Full featured Linux laptop	Neo FreeRunner
bcast-video-player	8%	81%
openmokast	<1%	2%
X driver	3%	8%
Idle	92%	0%

Table 7.2: Average resources usage of the video player application

Audio/video software libraries. The same media libraries were used on both the Linux computer and the FreeRunner.

Openmokast server. There were no modifications of the server between the tests on a Linux computer and on the FreeRunner. The same code is running on both platforms.

Efficiency of compiling tools. This element is difficult to evaluate because only one application building suite was available, so it was not possible to find a comparison point. However, since this build environment is used by all Openmoko developers, it can be considered that the building tools are in a mature state.

Processing capabilities of the device. While decoding the video service, there is a large difference in resources usage between the Linux host and the FreeRunner. In fact, on the FreeRunner, the processor usage ratio is at 100% during the whole video playout. The average resources usage for each software components of the video player is shown in Table 7.2.

These results show that the FreeRunner is not yet ready for fullscreen video services. Some new development is required, either to produce more efficient decoding libraries, to make better use of the available accelerator hardware or to improve the hardware capabilities of the device.

Concerning the process of developing the application, the development tools can be considered to be mostly equivalent in performance as many well known commercial solutions. The build system, however, is a lot more complex to use and this may be a problem for potential third party developers that would like to innovate with broadcasting applications. Finally, application distribution may also be problematic because the main Openmoko distributions are still subject to many stability issues. The developers that want to offer an easy access to their applications can use the distribution service available in [61].

Part III

Conclusions

Chapter 8

Future Work

”Open source software represents the most significant all encompassing and long-term trend that the industry has seen since the invention of the fundamental data storage architectures and SQL APIs in the early 1980s.”

– *IDC (2006).*

8.1 Adaptation for the Android Platform

DURING the course of this work, the Android platform, which is described in Section 3.2.1, was released officially as an open source project and also gained some popularity inside the mobile developers community. The Openmokast software can most probably be adapted easily to any Linux-based mobile platform, but the case of Android is a little bit different because of its Java-based API. In fact, the software developed for the Android platform must be implemented in the Java language and it has access only to the Android development API.

On the other side, the Android platform is running on top of a full Linux system and it uses many native non-Java libraries to offer some of its middleware services. In the case of these native Linux libraries, control is provided to the Java applications

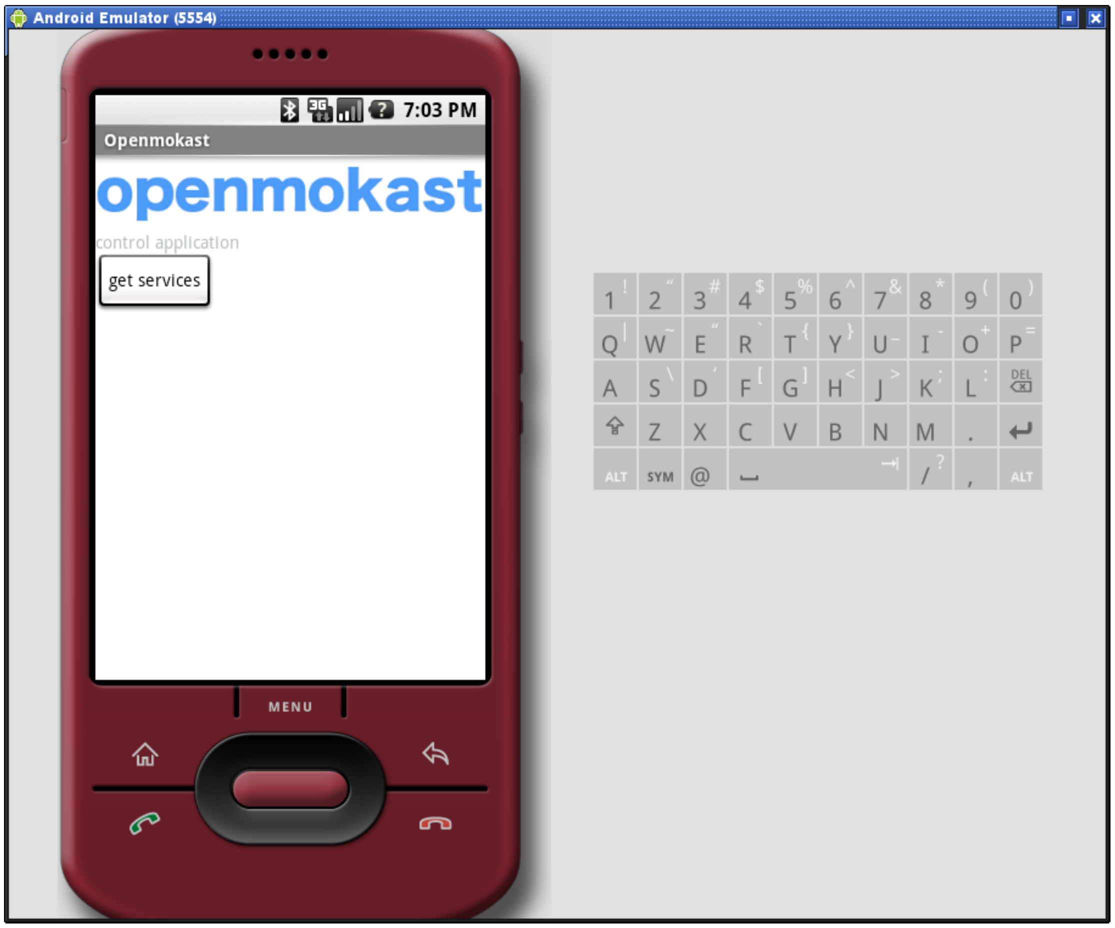


Figure 8.1: Openmokast test application running on the Android emulator

through a conversion layer that translates the communication into the Java API. This means that Java methods were integrated in the plaform to access the services offered by each of these native libraries. Consequently, the Openmokast software could be integrated in the Android project by adding the Openmokast software as an Android library and by implementing a broadcast service API in Java.

A project from a Canadian company, called Koolu, is currently ongoing with the goal to port Android to the Neo FreeRunner. The latest version is available from their website [62] and it was installed on one FreeRunner for trial purpose. Moreover, as a first proof of concept, the Openmokast server software was built for the Linux version that is at the base of Android on the FreeRunner, resulting in a functional

Openmokast server that was running natively on an Android phone. The next step was to find a way to establish the communication between Openmokast and the Android Java environment. Because the first version of the receiver management software presented in Section 5.1 was supporting a simple Telnet communication protocol, an Android application was developed to test the interaction between the native Openmokast server and a Java client application. It was then possible to execute the required Telnet commands to read the information about a tuned DAB service. The test application running on Android is shown in Fig. 8.1. This solution enables to use the Openmokast server software from an Android application by using only the networking layer of the device. The functional access to the broadcasting receiver was simpler and faster to develop with this method. On the other side, this solution bypasses the Android API.

The full functionalities of the receiver were not tested during the trials with Android, but the results show that it would be possible to build some client applications for broadcasting services under the Android platform. The next step would be to build some demonstration applications that can actually play some of these services. The multimedia components available from the Android API could be used to decode the multimedia content extracted from the broadcasting receiver and sent to Android through the networking layer.

8.2 Codec Adaptation

Because multiple digital broadcasting technologies exist nowadays, different codecs are used to carry the services from the server to the receiver. Moreover, there are some research under way [63] with the goal to provide a multi-standard receiver front end that would be able to receive different broadcasting standards on a single device.

The API defined in this document takes care of the control of the devices and the access to the data streams. However, no standardized decoder has been discussed to handle the multimedia streams. Another layer of the Openmokast software could hence be added to handle the codec divergence between the different technologies.

The main advantage of this future development would be a major simplification of the applications that would use broadcasting streams. This means that a software that wants to simply support broadcast video playout, no matter what broadcasting technology is used in its geographic location, would not need to support a vast list of audio/video codecs. On the other side, a standard selection of audio/video codecs would be the opposite of the open paradigm where digital broadcasting applications could rapidly evolve and offers new broadcasting applications. For this reason, this problem will be left open for future research.

8.3 Standardization

The Openmokast API is using D-Bus, a popular interprocess communication protocol, and uses a set of methods to control the broadcasting hardware on a mobile device. These characteristics are the key elements that make it well-suited for its integration into mobile middleware standards such as the FreeSmartPhone.org [27] platform. Some subsequent work to adapt the API to make it consistent with such a middleware standardization effort could position the Openmokast API as the main broadcasting API used in the major mobile devices projects.

Chapter 9

Discussions and Conclusions

"The linux billionaires are those who use linux and save money."

– *Znork, on Slashdot.*

THE Openmokast project described in this work was created to solve the problem of enabling broadcasting technologies on mobile devices as defined in Chapter 4. First, some components available in the open source software world and at CRC were identified in Chapter 5. These components were used in the development of the Openmokast platform and they contributed in the short time span required for the creation of the Openmokast prototype. The details about the development of the software and prototype were described in Chapter 6. The Openmokast software was developed and then integrated into an Openmoko FreeRunner device in addition with the required hardware receiver to produce a functional prototype. A key component of this development is the Openmokast API. This D-Bus based interface can be used by other software developers to easily and rapidly integrate broadcasting services into their own application. The API was used throughout the example of the development of a new application in Chapter 7. This case study was conducted to validate the usability and the performance of the Openmokast software to develop new multimedia broadcasting applications. A broadcast video player could be implemented in a short time with little resources thanks to the Openmokast server API. This case study demonstrates how the Openmokast project can be used to catalyse new and innovative services that take advantage of broadcasting technologies.

Chapter 8 defines the major topics that would be important to develop for the future success of the Openmokast platform. To generate increased interest in the platform and to help bringing more contributors to the development, the Openmokast software was released as an open source project[1]. It is believed that this was the best thing to do to achieve the future goals of the platform in the shortest time possible.

While working on this project, the Openmoko platform and organization went through a lot of changes. The architecture of the official Openmoko Linux distribution was also modified more than once. For instance, GTK+ was chosen as the user interface toolkit for the Openmokast software because, at the time of starting the project, it was the only one that was supported by the Openmoko platform. Currently, the support for many additional toolkits is included in the platform. The Openmoko building tools, based on OpenEmbedded, were also constantly modified. This made it difficult to stay up-to-date with the latest development of the Openmoko software and it often caused many issues at the time of the update. Moreover, the main documentation source for the platform is hosted on their wiki, which is a community tool open to anyone that has completed the registration process. While this idea follows the principal of openness of the project, it contributed to the over-production of documentation that was often outdated and duplicated into different pages. All these problems contributed to the difficulty of the development for the Openmoko platform. To be considered as a good platform for mobile applications development, the Openmoko project still needs some major improvement.

[1]The code releases are available from Sourceforge at `https://sourceforge.net/projects/openmokast`.

On a more positive note, these encountered problems may have been the price to pay to be able to work in a completely free and open environment. We believe that no project, other than the Openmoko platform, would have enabled the level of modifications that were achieved to produce the Openmokast framework and prototype. This innovative way of designing, producing and distributing the platform is an interesting concept that offer many advantages over the competing solutions. The participation of a large community rapidly generated a large amount of contributions of new software and new Linux distributions for the Openmoko devices. Few projects were able to achieve this level of participation in such a short time.

The Openmokast software was tried by numerous users but the large scale acceptance of such a technology will become possible only from the moment the broadcast receiver manufacturers will start building and selling devices compatible with open mobile handsets. We think the Openmokast project is an obvious element to help them selling their equipment but it was a surprise to discover that most of these manufacturers will be ready to provide their receivers only to other device integrators and, moreover, only under restrictive non-disclosure agreements.

Finally, during the course of this project, the Openmokast platform was presented in different events related to broadcasting or to open source software development. So far the platform generated a lot of interest throughout the open source developers community. It is often referenced as one of the only important hardware modification project that was based on the Neo FreeRunner device to date.

References

[1] R. Wietfeldt, "Handset system architectures for mobile dtv," *Consumer Electronics, 2006. ISCE '06. 2006 IEEE Tenth International Symposium on*, pp. 1–6, 2006.

[2] S. Pizzi, "Whose device is it, anyway?." `http://www.radioworld.com/pages/s.0054/t.8712.html`, Sept. 2007. Many Services Can Converge and Connect at the Personal Portable Media Device, But Who Decides Which?

[3] G. Brown, "Linux - a platform for innovation in converged mobile handsets," *BT Technology Journal*, vol. 25, no. 2, pp. 126–132, 2007.

[4] Y. C. Cho and J. W. Jeon, "Current software platforms on mobile phone," *Control, Automation and Systems, 2007. ICCAS '07. International Conference on*, pp. 1862–1867, 17-20 Oct. 2007.

[5] "Openmoko." `http://www.openmoko.org`. Last visited March 2009.

[6] "Qtopia phone edition." `http://trolltech.com/products/qtopia/phone_edition`. Last visited Feb. 2009.

[7] "Android." `http://code.google.com/android`. Last visited March 2009.

[8] M. Kornfeld and G. May, "Dvb-h and ip datacast - broadcast to handheld devices," *Broadcasting, IEEE Transactions on*, vol. 53, no. 1, pp. 161–170, March 2007.

[9] S. Cho, G. Lee, B. Bae, K. Yang, C.-H. Ahn, S.-I. Lee, and C. Ahn, "System and services of terrestrial digital multimedia broadcasting (t-dmb)," *Broadcasting, IEEE Transactions on*, vol. 53, no. 1, pp. 171–178, March 2007.

[10] F. Allamandri, S. Campion, A. Centonza, A. Chernilov, J. P. Cosmas, A. Duffy, D. Garrec, M. Guiraudou, K. Krishnapillai, T. Levesque, B. Mazieres, R. Mies, T. Owens, M. Re, E. Tsekleves, and L. Zheng, "Service platform for converged interactive broadband broadcast and cellular wireless," *Broadcasting, IEEE Transactions on*, vol. 53, no. 1, pp. 200–211, March 2007.

[11] F. Hartung, U. Horn, J. Huschke, M. Kampmann, T. Lohmar, and M. Lundevall, "Delivery of broadcast services in 3g networks," *Broadcasting, IEEE Transactions on*, vol. 53, no. 1, pp. 188–199, March 2007.

[12] M. Luby, T. Gasiba, T. Stockhammer, and M. Watson, "Reliable multimedia download delivery in cellular broadcast networks," *Broadcasting, IEEE Transactions on*, vol. 53, no. 1, pp. 235–246, March 2007.

[13] F. Lefebvre, J.-M. Bouffard, and P. Charest, "Open source handhelds - a broadcaster-led innovation for bth services," *EBU Technical Review*, vol. Q4, p. xx, 2008.

[14] F. Lefebvre, J.-M. Bouffard, and P. Charest, "Open mobile broadcasting phones," in *Broadcast Asia 2008*, 2008. held June 17-20, 2008, Singapore.

[15] "Openmokast - resources for open mobile broadcast devices." `http://openmokast.org/`. Last visited April 2009.

[16] "The gnu project." `http://www.gnu.org/gnu/the-gnu-project.html`. Last visited Sept. 2008.

[17] "Gnu general public license." `http://www.gnu.org/copyleft/gpl.html`. Last visited June 2008.

[18] D. Searls, "Linux for suits: Picking new fights," *Linux J.*, vol. 2007, no. 158, p. 15, 2007.

[19] "Worlddmb - welcome to worlddmb online." `http://www.worlddab.org/`. Last visited Dec. 2008.

[20] "Dvb mobile tv - dvb-h - dvb-sh - dvb-ipdc." `http://www.dvb-h.org/`. Last visited Dec. 2008.

[21] M. Brooks, "Development of broadcast technologies for mobile tv," *Broadcasting Spectrum: The Issues, 2005. The IEE Seminar on (Ref. No. 2005/11049)*, pp. 16 pp.–, June 2005.

[22] "Radio broadcasting systems; digital audio broadcasting (dab) to mobile, portable and fixed receivers." ETSI EN 300 401, 06 2006. V1.4.1.

[23] W. Hoeg and T. Lauterbach, *Digital audio broadcasting: principles and applications of digital radio.* Wiley, 2nd edition ed., October 2003.

[24] "Limo foundation." `http://www.limofoundation.org`. Last visited June 2008.

[25] "Linux phone standards (lips) forum." `http://www.lipsforum.org`. Last visited June 2008.

[26] "Open handset alliance." `http://www.openhandsetalliance.com`. Last visited Sept. 2008.

[27] "Freesmartphone.org." `http://www.freesmartphone.org`. Last visited March 2009.

[28] E. Oliver, "A survey of platforms for mobile networks research," *SIGMOBILE Mobile Computing and Communications Review*, vol. 12, no. 4, pp. 56–63, 2008.

[29] "Lips reference model." Ref No.: LIPS-AWG-Ref_Arch-v1_0_1-20071205-A, 5 December 2007. LiPS Forum.

[30] "Limo foundation platform architecture white paper - v1.0," January 17, 2007.

[31] "What is android?." `http://code.google.com/android/what-is-android.html`. Last visited March 2009.

[32] "Documentation / android - an open handset alliance project." `http://code.google.com/android/documentation.html`. Last visited March 2009.

[33] "Freedesktop.org." `http://www.freedesktop.org`. Last visited March 2009.

[34] "Eclipse - an open development platform." `http://www.eclipse.org/`. Last visited Dec. 2008.

[35] "Openembedded | metadata for building distributions - preferably embedded target platforms." `http://oe.linuxtogo.org/`. Last visited Feb. 2009.

[36] "Qemu - open source processor emulator." `http://bellard.org/qemu/`. Last visited Jan. 2009.

[37] "Gnome mobile." `http://www.gnome.org/mobile/`. Last visited April 2009.

[38] "Lgpl license option added to qt." `http://www.qtsoftware.com/about/news/lgpl-license-option-added-to-qt`, Feb. 2009. Nokia to add LGPL Open Source licensing option for the Qt UI and application framework.

[39] S. Gilbertson, "Apple opens iphone but key restrictions remain." `http://blog.wired.com/monkeybites/2008/03/apple-delivers.html`, March 2008.

[40] J.-M. Bouffard and F. Lefebvre, "An ip based file delivery platform for mobile multimedia broadcasting," in *Proceedings of the IASTED International Conference on Wireless Networking and Emerging Technologies, as part of the Fifth IASTED International Multi-Conference on Wireless and Optical Communications* (A. Fapojuwo, ed.), pp. 105–110, IASTED, ACTA Press, July 2005. held July 19-21, 2005, Banff, Alberta, Canada.

[41] J.-M. Bouffard, F. Lefebvre, and B.-H. Lee, "Multimodal applications for mobile multimedia broadcasting," in *Proceedings of the IASTED International Conference on Wireless Networking and Emerging Technologies, as part of the Seventh IASTED International Multi-Conference on Wireless and Optical Communications* (A. Vukovic, ed.), IASTED, ACTA Press, May 2007. held May 30 - June 1, 2007, Montreal, Quebec, Canada.

[42] J.-M. Bouffard and F. Lefebvre, "Large scale distribution of popular internet user generated content to mobile devices," in *Proceedings of the IASTED International Conference on Wireless Networking and Emerging Technologies, as part of the Eighth IASTED International Multi-Conference on Wireless and Optical Communications* (A. Vukovic, ed.), IASTED, ACTA Press, May 2008. held May 26-28, 2008, Quebec City, Quebec, Canada.

[43] M. Velez, D. de la Vega, P. Angueira, D. Guerra, G. Prieto, and A. Arrinda, "Field measurement based performance analysis of digital audio broadcasting (dab) reception in mobile channels," *Vehicular Technology Conference, 2005. VTC 2005-Spring. 2005 IEEE 61st*, vol. 1, pp. 247–251 Vol. 1, May-1 June 2005.

[44] "Mplayer - the movie player." http://www.mplayerhq.hu/. Last visited Jan. 2009.

[45] "gstreamer: open source multimedia framework." http://www.gstreamer.net/. Last visited Jan. 2009.

[46] "Dream drm receiver." http://drm.sourceforge.net/. Last visited Dec. 2008.

[47] "Digital audio broadcasting (dab); multimedia object transfer (mot) protocol." ETSI EN 301 234, 05 2006. V2.1.1.

[48] "Digital audio broadcasting (dab); journaline; user application specification." ETSI TS 102 979, 06 2008. V1.1.1.

[49] "Viadab 2 - programmer's guide - version 1.4," 2002. RadioScape Ltd. 2 Albany Terrace - Regents Park - LONDON - NW1 4DS Phone: +44 (0)20 7224 1586 Fax: +44 (0)20 7224 1595 Website: www.radioscape.com.

[50] "Digital audio broadcasting system - specification of the dab command set for receivers (dcsr)." EN 50320:2000, 2001.

[51] "Digital audio broadcasting (dab); a virtual machine for dab: Dab java specification." ETSI TS 101 993, 03 2002. V1.1.1.

[52] "Digital audio broadcast (dab); digital radio mondiale (drm); digital radio data interface (drdi)." draftETSI TS XXX XXX, 08 2004. V0.0.0.

[53] "Digital radio mondiale (drm); distribution and communications protocol (dcp)." ETSI TS 102 821, 10 2005. V1.2.1.

[54] "Digital radio mondiale (drm); receiver status and control interface (rsci)." ETSI TS 102 349, 11 2005. V1.2.1.

[55] “"sniffusb 2.0" usb sniffer for windows.” `http://www.pcausa.com/Utilities/UsbSnoop/default.htm`. Last visited March 2009.

[56] “Libusb sourceforge project page.” `http://sourceforge.net/projects/libusb/`. Library to enable user space application programs to communicate with USB devices. Last visited Sept. 2008.

[57] “D-bus project page on freedesktop.org.” `http://www.freedesktop.org/wiki/Software/dbus`. Last visited March 2009.

[58] “Vlc media player - the cross-platform media player and streaming server.” `http://www.videolan.org/`. Last visited April 2009.

[59] “Crc mmb tools - an effort to support, foster and promote the development of new applications and tools in the field of mobile multimedia broadcasting (mmb).” `http://mmbtools.crc.ca/`. Last visited April 2009.

[60] “Gnu radio - the gnu software radio.” `http://www.gnu.org/software/gnuradio/`. Last visited April 2009.

[61] “opkg - a software directory for openmoko phones.” `http://www.opkg.org/`. Last visited April 2009.

[62] “Android on freerunner.” `http://freerunner.android.koolu.com/`. Last visited March 2009.

[63] C. Babla, “A software modem approach to multi-standard radio & tv reception on mobile devices,” *EBU Technical Review*, vol. Q1, p. xx, 2009.

Part IV

Appendices

Appendix A

Openmokast D-Bus Interface Specification

org.openmokast.Receiver.Control

Description

The Receiver Control interface is used to control the broadcasting receiver.

Namespace

org.openmokast.Receiver.Control

Methods

- Tune
- Scan
- ScanAll

- GetFrequency
- GetStatus
- IsDecoding
- StartDecoding
- StopDecoding
- GetEnsemble
- GetServiceArray
- GetComponentArray
- GetTransportMode
- GetDataComponentType
- GetUserApplicationType
- GetReceiverCapabilities
- GetAvailableReceivers
- SelectReceiver
- SelectOutputType

Signals

- ensemble_updated_notify

Errors

None

Methods

Tune (uu) -> b

Description: Tune the given frequency.

Parameters

u: frequency The frequency in KHz.

u: mode The DAB mode.

Returns

b: ret True if command was accepted.

Scan (uu) -> b

Description: Tuning the first available frequency starting at the given frequency and band.

Parameters

u: frequency The frequency in KHz.

u: band The DAB band: 1=Band-III, 2=L-Band, 3=Canada-L-Band(default), 4=Korea-Band-III.

Returns

b: ret True if command was accepted.

ScanAll (u) -> aub

Description: Create a list of the available frequencies at the given band.

Parameters

u: band The DAB band: 1=Band-III, 2=L-Band, 3=Canada-L-Band(default), 4=Korea-Band-III.

Returns

au: frequency_array_ret An array containing the available frequencies.

b: ret True if command was accepted.

GetFrequency ( ) -> ub

Description: Returns the frequency that is currently tuned or 0 if nothing is tuned.

Returns

u: frequency_ret The currently tuned frequency or 0 of not tuned.

b: ret True if command succeeded.

GetStatus ( ) -> sb

Description: Returns the status of the receiver.

Returns

s: status_ret A protocol dependent string containing the current status of the receiver.

b: ret True if command succeeded.

IsDecoding (uu) -> b

Description: Returns the status of a component decoding.

Parameters

u: service_id The service identifier.

u: component_id The component identifier.

Returns

b: ret True if the selected component is currently decoded.

StartDecoding (uu) -> sb

Description: Starts the decoding of a component.

Parameters

u: service_id The service identifier.

u: component_id The component identifier.

Returns

s: uri_ret The URI of the newly started component.

b: ret True if the command succeeded. False if the component doesn't exist or if it cannot be decoded.

StopDecoding (uu) -> b

Description: Stops the decoding of a component.

Parameters

u: service_id The service identifier.

u: component_id The component identifier.

Returns

b: ret True if the command succeeded. False if the component doesn't exist.

GetEnsemble ( ) -> usb

Description: Get information about the tuned ensemble.

Returns

u: id_ret The identifier of the tuned ensemble or 0 if no ensemble found.

s: label_ret The name of the tuned ensemble.

b: ret True if command succeeded.

GetServiceArray ( ) -> auasb

Description: Get information about the available services.

Returns

au: id_array_ret Array containing a list of service identifiers for the current ensemble.

as: label_array_ret Array containing a list of service labels for the current ensemble. The array size is the same than the array of identifiers

b: ret True if command succeeded.

GetComponentArray (u) -> auasb

Description: Get information about the components inside a service.

Parameters

u: service_id The service identifier.

Returns

au: id_array_ret Array containing a list of component identifiers for the current service. The first element in the list is the main component.

as: label_array_ret Array containing a list of component labels for the current service. The array size is the same than the array of identifiers

b: ret True if command succeeded.

GetTransportMode (uu) -> ub

Description: Get information about the transport mode for a component.

Parameters

u: service_id The service identifier.

u: component_id The component identifier.

Returns

u: transport_mode_ret The identifier of the transport mode for the component. The possible values are: 0 for AUDIO STREAM, 2 for DATA STREAM, 3 for FIDC SERVICE and 4 for DATA PACKET SERVICE.

b: ret True if component exists.

GetDataComponentType (uu) -> ub

Description: Get information about the type of a data component.

Parameters

u: service_id The service identifier.

u: component_id The component identifier.

Returns

u: data_component_type_ret The type of data component.

b: ret True if component exists.

GetUserApplicationType (uu) -> ub

Description: Get information about the User Application (UA) type for a component.

Parameters

u: service_id The service identifier.

u: component_id The component identifier.

Returns

u: ua_type_ret The User Application type as defined in Table 16 of ETSI TS 101 756.

b: ret True if component exists.

GetReceiverCapabilities ( )

Description: TO BE DEFINED - Get information about the receiver such as the number of subchannels that can be decoded at the same time.

GetAvailableReceivers ( )

Description: TO BE DEFINED - Get the available receivers from the platform.

SelectReceiver ( )

Description: TO BE DEFINED - Select the active receiver.

SelectOutputType ( )

Description: TO BE DEFINED - Select the output format for the data of a decoded component.

Signals

ensemble_updated_notify (s)

Description: Emitted when the current ensemble is updated with new information.

Parameters

s: update_str A protocol dependent string that identifies what was updated.

| Specified 2008 by Jean-Michel Bouffard for the openmokast.org project. | For more information: jean-michel (dot) bouffard (at) crc (dot) ca

Appendix B

Openmokast-Client Sample Project

```
/* -*- Mode: C; indent-tabs-mode: t; c-basic-offset: 4; tab-width: 4 -*- */
/*************************************************************
 *
 * Copyright (C) Her Majesty the Queen in Right of Canada, 2009
 * Communications Research Centre (CRC)
 *
 *      This program is free software; you can redistribute it and/or modify
 *      it under the terms of the GNU General Public License as published by
 *      the Free Software Foundation; either version 2 of the License, or
 *      (at your option) any later version.
 *
 *      This program is distributed in the hope that it will be useful,
 *      but WITHOUT ANY WARRANTY; without even the implied warranty of
 *      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *      GNU General Public License for more details.
 *
 *      You should have received a copy of the GNU General Public License
 *      along with this program; if not, write to the Free Software
 *      Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 *
 * Contact: Jean-Michel Bouffard
 *
 * Email: <jean-michel.bouffard@crc.ca>
 *
 * Web: http://openmokast.org/
 *      http://mmbtools.crc.ca/
 *      http://www.crc.ca/mmb
 *
 *
 * 2008/12 CRC
 *     Initial release
 *
 ******************************************************************/

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>

#include <config.h>

#include <gtk/gtk.h>
#include <glade/glade.h>

 // DBus related includes
#include <org.openmokast.Receiver.client.h>
#define DBUS_SERVICE_OPENMOKAST    "org.openmokast.Receiver"
#define DBUS_PATH_OPENMOKAST       "/org/openmokast/Receiver"
#define DBUS_INTERFACE_OPENMOKAST "org.openmokast.Receiver.Control"

#define SIGNAL_ENSEMBLE_UPDATED_NOTIFY "ensemble_updated_notify"
 //#define DBUS_INTERFACE_OPENMOKAST_DATA "org.openmokast.Receiver.Data"

 /*
  * Standard gettext macros.
  */
#ifdef ENABLE_NLS
#  include <libintl.h>
#  undef _
#  define _(String) dgettext (PACKAGE, String)
#  ifdef gettext_noop
#    define N_(String) gettext_noop (String)
#  else
#    define N_(String) (String)
#  endif
#else
#  define textdomain(String) (String)
#  define gettext(String) (String)
#  define dgettext(Domain,Message) (Message)
#  define dcgettext(Domain,Message,Type) (Message)
#  define bindtextdomain(Domain,Directory) (Domain)
#  define _(String) (String)
#  define N_(String) (String)
#endif

#include "callbacks.h"

 /* For testing propose use the local (not installed) glade file */
 /* #define GLADE_FILE PACKAGE_DATA_DIR"/openmokast-client/glade/
     openmokast-client.glade" */
#define GLADE_FILE "openmokast-client.glade"

GtkWidget *
create_window (void)
{
  GtkWidget *window;
  GladeXML *gxml;

  gxml = glade_xml_new (GLADE_FILE, NULL, NULL);

  /* This is important */
  glade_xml_signal_autoconnect (gxml);
  window = glade_xml_get_widget (gxml, "window");

  return window;
}

static void
EnsembleUpdatedNotifySignalHandler (DBusGProxy * proxy,
                                    const char *in_string, gpointer
                                       userData)
{
  /* Since method calls over D-Bus can fail, we'll need to check
     for failures. The server might be shut down in the middle of
     things, or might act badly in other ways. */
  GError *error = NULL;

  g_print (":Ensemble updated (%s)\n", in_string);

  // Process any required task here

  /* Free up error object if one was allocated. */
  g_clear_error (&error);
}

int
main (int argc, char *argv[])
{
  GtkWidget *window;

#ifdef ENABLE_NLS
  bindtextdomain (GETTEXT_PACKAGE, PACKAGE_LOCALE_DIR);
  bind_textdomain_codeset (GETTEXT_PACKAGE, "UTF-8");
  textdomain (GETTEXT_PACKAGE);
#endif

  gtk_set_locale ();
  gtk_init (&argc, &argv);

  window = create_window ();
  gtk_widget_show (window);

  // DBus initialisation
  DBusGConnection *connection;
  GError *error;
  DBusGProxy *proxy;
  char **name_list;
  char **name_list_ptr;
  gboolean test_ret;

  g_type_init ();

  error = NULL;
  connection = dbus_g_bus_get (DBUS_BUS_SESSION, &error);
  if (connection == NULL)
    {
      g_printerr ("Failed to open connection to bus: %s\n", error->
          message);
      g_error_free (error);
      exit (1);
    }

  /* Create a proxy object for the "openmokast server" (name "org.
     openmokast.Receiver") */
  proxy = dbus_g_proxy_new_for_name (connection,
                                      DBUS_SERVICE_OPENMOKAST,
                                      DBUS_PATH_OPENMOKAST,
                                      DBUS_INTERFACE_OPENMOKAST);
  if (!proxy)
    {
      g_printerr ("Failed to create proxy...\n");
      //g_error_free (error);
      exit (1);
    }

  // Signal registration
  /* Since the function doesn't return anything, we cannot check
     for errors here. */
  dbus_g_proxy_add_signal (      /* Proxy to use */
                            proxy,
                            /* Signal name */
                            SIGNAL_ENSEMBLE_UPDATED_NOTIFY,
                            /* Will receive one string argument */
                            G_TYPE_STRING,
                            /* Termination of the argument list */
                            G_TYPE_INVALID);

  // Connection between the signals and the callback method
  dbus_g_proxy_connect_signal ( /* Proxy object */
                                proxy,
                                /* Signal name */
                                SIGNAL_ENSEMBLE_UPDATED_NOTIFY,
                                /* Signal handler to use. Note that the
                                   typecast is just to make the compiler
                                   happy about the function, since the
                                   prototype is not compatible with
                                   regular signal handlers. */
                                G_CALLBACK
                                (EnsembleUpdatedNotifySignalHandler),
                                /* User-data (we don't use any). */
                                NULL,
                                /* GClosureNotify function that is
                                   responsible in freeing the passed
                                   user-data (we have no data). */
                                NULL);

  // Reads the available ensemble name
  guint test_id_ret;
  char *test_label_ret;
  if (!org_openmokast_Receiver_Control_get_ensemble (proxy,
                                                    &test_id_ret,
                                                    &test_label_ret,
                                                    &test_ret, &error))
    {
      /* Checks for remote exceptions */
      if (error->domain == DBUS_GERROR
          && error->code == DBUS_GERROR_REMOTE_EXCEPTION)
        g_printerr ("Caught remote method exception %s: %s",
                    dbus_g_error_get_name (error), error->message);
      else
        g_printerr ("Error: %s\n", error->message);
      g_error_free (error);
      exit (1);
    }

  // Reads list of avalable services in ensemble
  GArray *array1;
  char **array2;
  if (!org_openmokast_Receiver_Control_get_service_array (proxy,
                                                         &array1,
                                                         &array2,
                                                         &test_ret, &
                                                             error))
    {
      /* Checks for remote exceptions */
      if (error->domain == DBUS_GERROR
          && error->code == DBUS_GERROR_REMOTE_EXCEPTION)
        g_printerr ("Caught remote method exception %s: %s",
                    dbus_g_error_get_name (error), error->message);
      else
        g_printerr ("Error: %s\n", error->message);
      g_error_free (error);
      exit (1);
    }

  /* Print the results */

  g_print ("Name of the current Ensemble:\n");
  g_print ("id->%d, label->%s\n", test_id_ret, test_label_ret);

  guint val1;
  char *val2;
  guint i = 0;
  if (array1->len > 0)
    {
      while ((i < array1->len) && (val1 = g_array_index (array1, guint,
          i)))
        {
          val2 = array2[i];
          g_print ("Service id: %d label: %s\n", val1, val2);
          i++;
        }
    }
  else
    {
      g_print ("No services...\n");
    }

  g_free (test_label_ret);

  g_object_unref (proxy);

  //gtk_main ();
  return 0;
}
```