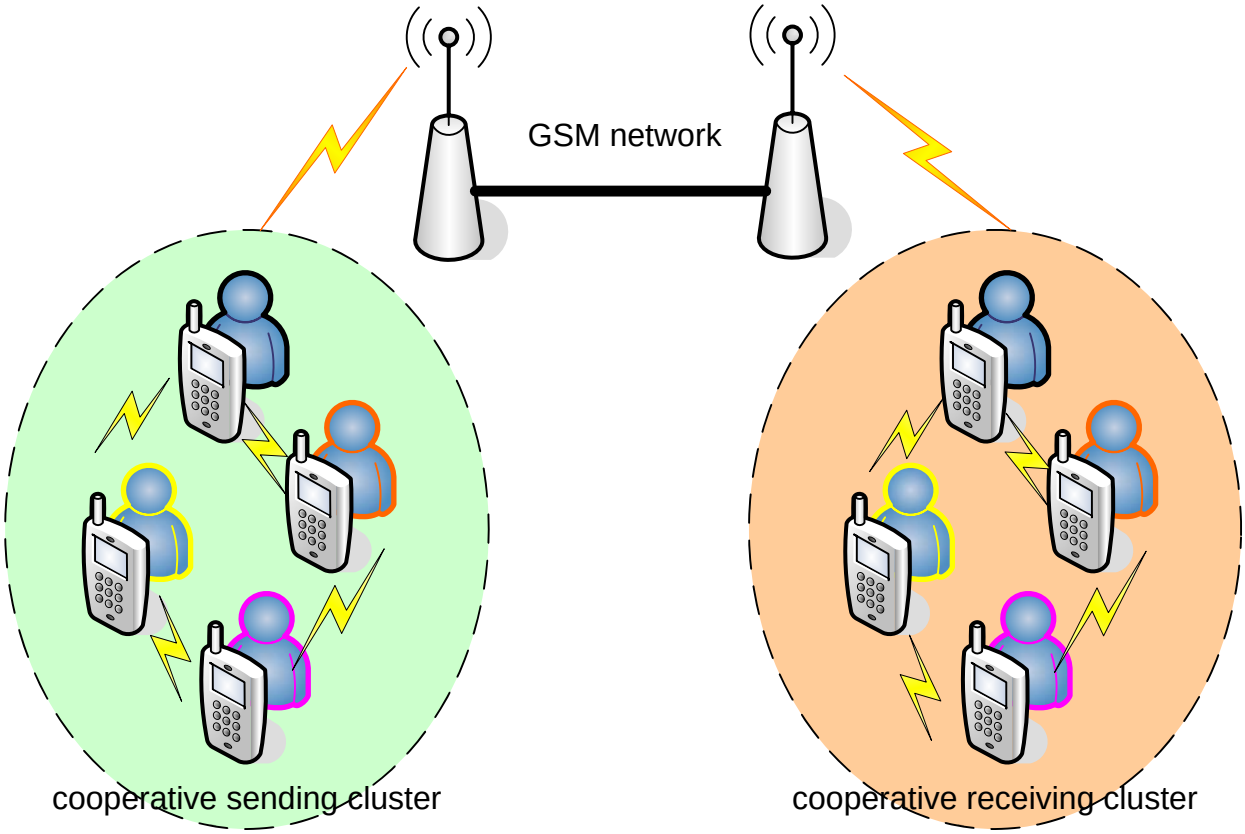
Embedded realtime systems

The Medium Is The Message

Group 451
Anders Grauballe, Mikkel Gade Jensen, Ulrik Wilken Rasmussen, Janne Dahl Rasmussen, Peter Østergaard

Supervisor: Frank H.P. Fitzek



4th Semester
Spring 2006
Institute for electronic system
Computer engineering

AALBORG UNIVERSITY
Institute for electronic systems
Computer engineering

Title: The medium is the message

Semester Theme: Embedded realtime systems

Project period:
4th semester, spring 2006

Project group:
451

Group members:
Anders Grauballe
Mikkel Gade Jensen
Janne Dahl Rasmussen
Ulrik Wilken Rasmussen
Peter Østergaard

Supervisor:
Frank H.P. Fitzek

Circulation: 8

Number of pages: 110

Number of attachment and type:
5 appendix, 1 CD

Finished: 29-05-2006

Synopsis:

The purpose of this project has been to develop a new way to send SMS-messages by making unanswered phone calls which makes it possible to send the messages for free.
Different methods could be used to accomplish this and the most robust method with groups of phones calling each other was selected. By letting the phones from the sender group make calls in a specific pattern it is possible to derive different bit combination.
An application using this method has been developed for Nokia Series 60 phones. The application has been implemented using Python for Series 60 with extension modules in Symbian C++.
The application has been tested on Nokia 6600 and N70 and can successfully send a message of 20 character from one group of phones to another in 65% of the attemts. With the current implementation the avarage time of sending one character is 94 seconds with each group containing 2 members.

Preface

This project has been carried out by project group 451, Computer Engineering on 4. semester at Aalborg University. The focus group for this report is people with an interest in alternative use of networks, mobile development and the idea of cooperation between mobile devices.

In this report pictures, tables etc. are labelled with a number for easy reference like 2.3. References to literature are shown as e.g. [Wik06]. The listings of code and references to methods and variables included in this report is written as `code example` .

In the back of the report a CD containing the following is included:

- The report in PDF form.
- The source code for the program
- An installation file of the application for Nokia S60 FP2 & FP3.
- All literature from web pages that has been cited in PDF form.
- A demo video showing the program running.

Finally we would like to express our gratitude to our supervisor Frank H. P. Fitzek as well as PhD-student Gian Paolo Perrucci and student Morten V. Pedersen for the help and advice we have received while working on this project.

Aalborg University may 29th 2006

Anders Grauballe	Mikkel Gade Jensen
Ulrik Wilken Rasmussen	Janne Dahl Rasmussen

Peter Østergaard

Contents

Chapter 1

Introduction

In Italy a trend has emerged where people utilize their mobile phones to send messages in a new experimental way. Young Italian students have developed a new communication system based on not answering phone calls. They call this system ”Squillo” which is a form of the word ”squillare” that means ”to ring” in Italian.

A squillo is a phone call which causes the receivers phone to ring for just a moment after which the caller hangs up. This can mean anything from a smile, to get a persons attention or to inform a friend that your are running late for a scheduled meeting and a lot of other things depending on the context. For the system to work, it is important that the two parts in the communication are aware of the meaning of a squillo and that an incoming call actually is a squillo. If this is not agreed upon, it could result in a misunderstanding where the squillo receiver calls you back wondering why you hung up before he could answer your call [Fle02].

The idea of squillo system could be used as a code system like morse code, the number of calls determining the message e.g. you call a friend three times, sending the message ”Meet me at three”. The code system could be a formal standard or simply something agreed between two people. Thus the squillo system is based on simple signals, basically carrying no data. It is up to the receiver to derive what the meaning of the squillo is.

Suppose this idea could be combined with existing technology expanding the capacity of a communication channel i.e. taking advantage of a switching of a media from one state to another and interpret this media coding as data. It is also possible to switch a signal from one channel to an other to gain certain benefits. Bluetooth is an example of this shifting, also known as frequency hopping. This is used for noise reduction and security reasons where the two parts in a communication agrees on a specific pattern of the frequency hopping to prevent others from understanding the signal. Figure 1.1 on the facing page shows how such a shifting might look like.

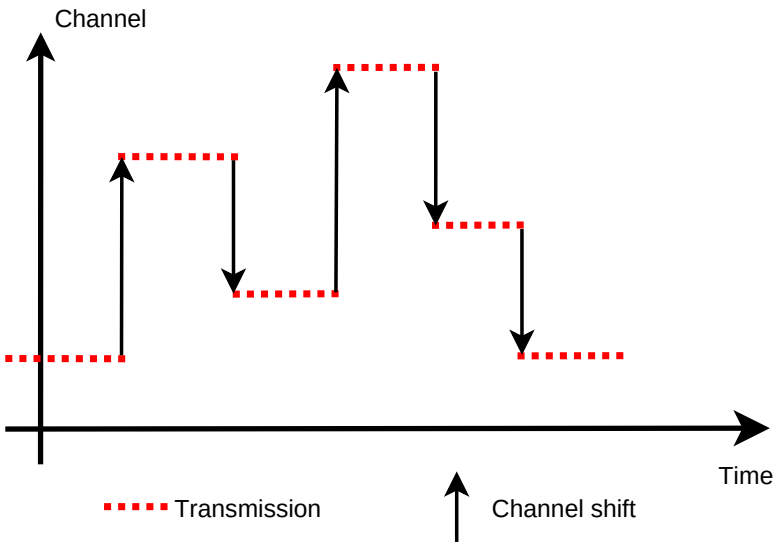


Figure 1.1: Transmission of information on different channels over time, this is also known as frequency hopping

The shifting technique can be used as another way to convey data from one part to an other. Combined with the squillo approach, it should be possible to develop a new application for messaging on mobile phones, mapping a pattern of unanswered calls to a message. In this way it would be possible to send a message from one phone to another free of charge, provided there is no dial-up charge. The transmission is of cause paid for by the GSM network provider and the amount of data transmitted this way is very small relative to the cost of establishing phone calls, thus this may be considered as a trick.

Almost every network provider in Denmark charges 0,25 kr. for dialling which means that this method makes it expensive to send a message. This problem is solved by using sim cards from a foreign network provider who does not charge for dialling [oT06].

For this project the following materials are at the project group's disposal:

- 4 Nokia 6600
- 4 Nokia N70
- 8 German Vodafone sim cards with no dial-up charge
- 8 Danish TDC mobil sim cards with dial-up charge

This project is made as a proof of concept of sending a message this way. The concept proves that it is possible to convey a derived message without actually sending it. The

concept may not have a market potential for sending messages but it gives a good understanding of this technology.

Chapter 2

Pre-analysis

This chapter presents different ways to transmit data over a network and this is followed by the different forms of scenarios which describes this project. Then a contextual scenario that demonstrates the context of the application is presented followed by a use case of the application describing the purpose of this project. This will demonstrate the user's interaction with the application. Furthermore an activity diagram will be made to illustrate the different tasks in the application and last a requirements specification will be found from the scenarios and the diagram.

2.1 Proposal Scheme

This section will describe some of the different ways of transmitting data over a network. It will focus on the speed and robustness of each type of communication and end up with a summary that concludes which way of transmitting is the best to use in this project.

2.1.1 Call / No call

An obvious way to handle such a transmission is to let two mobile phones ring each other in a specified time pattern. Given a time interval of e.g. one minute, the sender can call the receiver 1 or 0 times every minute to indicate whether the next bit is 1 or 0. In this case the frequency of the transmission is 1 bit/minute which is a very slow but also very robust protocol as no jitter occurs when the time interval is long enough. The time interval can be less than one minute but practically no less than 26 seconds as the call test between two German sim card concludes in section 3.4 on page 39. This scenario is

shown in figure 2.1. An application that uses this form of communication is described in appendix D.1 on page 104.

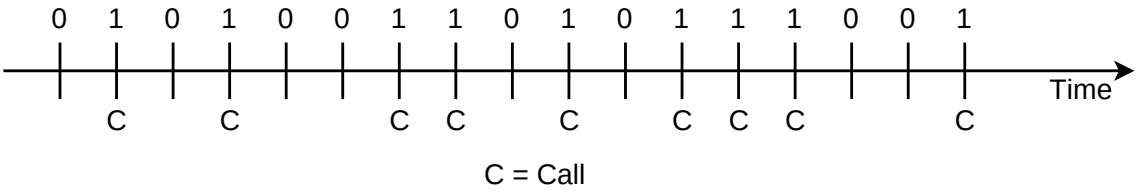


Figure 2.1: The Call / No call protocol. Call = 1, No call = 0

If more than one pair of phones is available, more bits can be sent at the same time. Then each pair of phones must have a priority to specify the order of the bits. To specify this, the phones on the sender side and the receiver side must be able to communicate with each other e.g. via a bluetooth connection. To make the communication on each side as limited as possible each pair of phones could get a specific part of the message to transmit. The number of bits that can be transmitted is equal to the number of phones, that is: $B = N$ where B is the number of bits transmitted per time period and N is the number of phone pairs available.

2.1.2 Time slot

The bit rate in a connection between two phones can be increased by letting the time interval between two calls determine the bit sequence. The sequence is initialized by a call from the sender to the receiver, a timer starts and resets at the next incoming call. Then the timer value is the transmitted bit sequence. E.g. if the purpose is to transmit four bits per time slot, and the timer interval is one second, the sender must call the receiver 0-15 seconds after the previous call. This scenario is shown in figure 2.2.

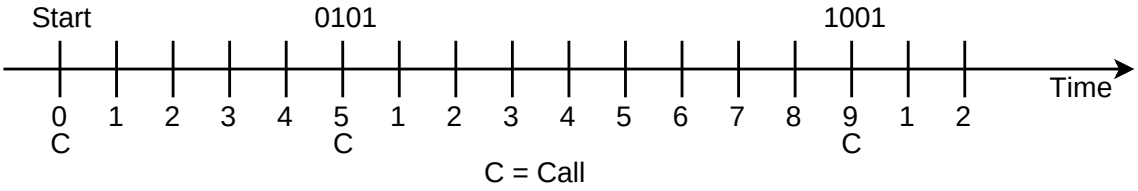


Figure 2.2: The Time slot protocol. The data sent corresponds to the elapsed time between two calls

This protocol is faster but also less robust compared to "Call / No call", described in section 2.1.1 on the previous page, because the calls must be registered at the exact time to avoid bit errors. This problem can be fixed by making the timer interval long enough to

eliminate jitter in the net, but this correction makes the bit rate considerably lower. Also maintaining a short timer interval and ignoring the least significant bits produces an equal result.

2.1.3 Cluster calls

If more than two phones are available for transmitting the message, the method ”Call / No call” has a fast bit rate, but in average it takes one call to send two bits. This can be improved but it is at the expense of the bit rate. An example of using the Cluster call method from four to one phone is described in appendix D.2 on page 106.

Presume that two clusters of phones are linked together in a local network (e.g. by bluetooth). By ordering one of the phones in a cluster to call a specific phone in an other cluster it is possible to derive $2 \times \log_2 N$ bits assuming that both clusters have N members. This scheme is shown in figure 2.3.

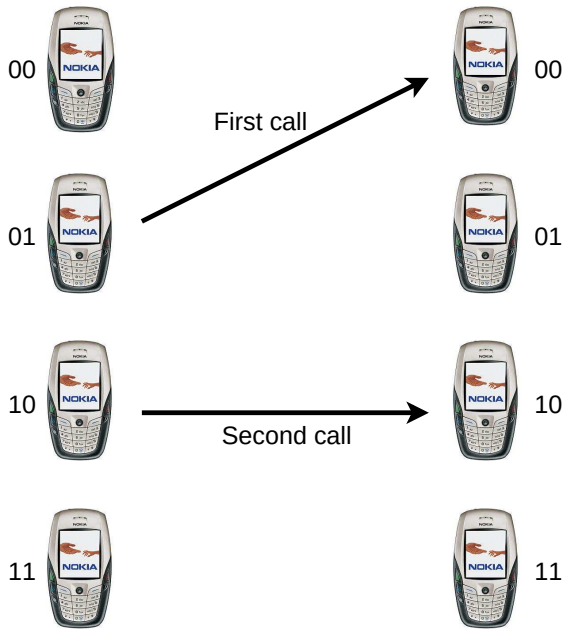


Figure 2.3: The Cluster Calls protocol. This call sequence is equivalent to the bit stream 0100 1010

2.1.4 Summary

The methods described in the previous sections is compared in table 2.1. Here it is clear that if only two phones are available the method Call / No call is preferable, but if there is more than two phones available there is no clear preferable method.

Total phones	2		More than 2	
Method	Time slot	Call / No call	Call/ No call	Cluster calls
Bit rate	Low	Medium	Highest	High
Robustness	Low	Medium	Medium	High

Table 2.1: Comparison of the different methods

Phones/side	Call / No call	Cluster Calls
2	2	2
4	4	4
8	8	6
16	16	8

Table 2.2: Comparison of number of bits transmitted per time period

Table 2.2 shows the difference in bit rates and from the discussion of the two methods it is clear that the method Cluster calls is more robust than Call / No call. For this reason the Cluster calls method will be used in this project to avoid using a lot of time on transmission errors.

2.2 Contextual scenario

After deciding what method to use, a contextual scenario will be set up to illustrate a possible use of the application. The scenario does not describe the use of the exact application developed in this project, but its purpose is to visualize the perspective in the application. A paper published by Hickey, Dean and Nunamaker [Hik99] recommends four steps to improve the completeness of a scenario. The four steps which are described below will be followed in the contextual scenario:

1. Define scenario goals and preconditions
2. Describe normal action sequence
3. Add additional details (e.g. complete actions descriptions or data requirements)

4. Add critical exceptions or error checking

This scenario describes how two people can send messages to each other by using more than one phone like described in section 2.1.3 on page 13.

2.2.1 Goal and preconditions

Preconditions: The program must be widely distributed and the users must carry more than one phone to obtain best use. It must be possible to make calls without having to pay dial-up charge.

Goal: To send a message between two people by making calls that are interpreted as bit-sequences.

2.2.2 Normal actions sequence

Tim is sitting in the bus on his way to school. In the bus is 30 other people of which 10 have the application installed on their phones. Tim wants to send a short message to his friend Alice to ask why she is not on the bus. He has two phones available and one of them has always the application running making it available to other users nearby to. He can see that someone else in the bus is already using it to send a message.

Tim starts the program on the other phones, and chooses to write a message in the menu that pops up. While Tim is writing the phone uses bluetooth to search for other phones in range that has the application started. When Tim is finished writing he can see in the display that 10 other phones are available and the estimated time for the message to be transmitted and he is asked if he wants to send the message now. Tim chooses yes and the phone starts transmitting orders to the other phones which starts calling the phones near Alice in a sequence determined by the message Tim wrote.

While the message is being transmitted a person gets of the bus and his phone is now out of range, Tim gets a new message on the display with the new estimated time for the transmission to complete.

2.2.3 Critical exceptions or error checking

If there is no or only a few other phones around with the program installed and started, the message can not be transmitted or it will take a very long time. Therefore Tim should get a choice if he rather wants to send the message the normal way and pay for it.

2.2.4 Limitations

In this project it will be attempted to make a simple version of the program described above. To make this simple program it is decided that the focus will be on transmitting the message, not on handling the different phones on each side, therefore the phones need to know each other.

2.3 Use case analysis

The use case analysis shows the system functionality of the program where the different actors interact with the system. The analysis shows a scenario where a user (the sender) wants to send a message to an other user (the receiver). Each user must have a group of phones which contains a master which controls the transmission and a slave which performs tasks requested by the master. From now on the terms master and slave will be used about these phones.

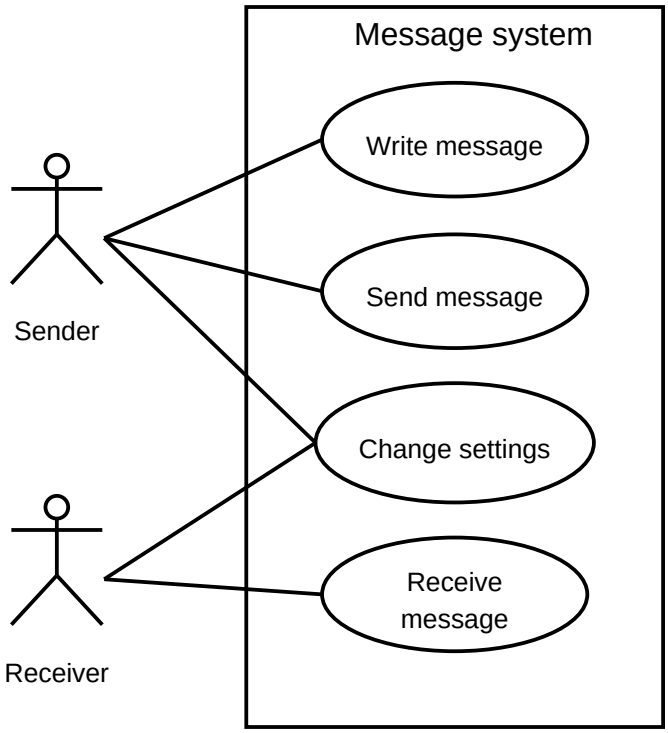


Figure 2.4: Use case diagram

2.3.1 Change settings

Before the user can send or receive a message, it has to be determined who the receiver and sender is, i.e. the two phones in the sender and the receiver group should be connected via bluetooth and the phone numbers for the opposite group needs to be entered.

Goal
The goal of this use case is to make the user choose the phones which should be used for sending or receiving a message.

Main scenario
The user needs to make the phone start searching for available phones and connect to one of them. The user also needs to decide which phone in the sender or the receiver group to send to or read the message from (the master).

1. Both the sender and the receiver search for available phones
2. Then each of them decide which phones to use and connects them
3. The sender decides which phone to write the message from
4. The receiver decides which phone to read the message from
5. The phone numbers of the opposite group is entered by the users

Exceptions
If either the sender or the receiver forgets to put in a phone number or do not make the bluetooth connection between the phones, they must get a warning on the screen of the phone

2.3.2 Write message

When the sender has set up the phone to send a message, the message for sending should be typed into an editor.

Goal
It must be possible for the sender to input a message as a string of 0-160 characters like an ordinary SMS.

Main scenario

1. Sender chooses ”write message” in the application

2. Sender writes the message he wants to send to the receiver
3. Sender chooses ”Send”

Exceptions
If the user is interrupted in this process the user should have the opportunity to save the message for later use. If the user does not write anything in the message and presses ”send” there should be a warning in the display.

2.3.3 Send message

This use case is very simple from the user's point of view, but there are some very critical points. First of all, it is very important that the phones that are chosen for sending this message is connected via bluetooth. Secondly the phone numbers entered under settings must be correct. Finally it is very important that the phones stay in range.

Goal
The messages written by the sender must be compressed and sent to the receiver.

Main scenario

1. The message is loaded from the master phones memory
2. The message is compressed
3. The compressed message is sent using the ”Cluster Calls” method described in section 2.1.3 on page 13. Each call must be hung up by the receiver to let the sender know that the call is received.

Exceptions
If a call is not hung up by the receiver a time out is raised and the call is repeated. If the phones in the sender or receiver group is disconnected the user must get a warning on the display.

2.3.4 Receive message

This use case have the same critical points described under ”Send message” above.

Goal
When the message is received by the receiver, the user must be able to read the message.

Main scenario
The message is received as a sequence of phone calls from the phones in the sender group, and the message is then reassembled and shown to the receiver.

1. The compressed message is received and stored in the master phones memory
2. The compressed message is decompressed and stored in the master phones memory
3. The receiver chooses ”Show message” and the message is shown

Exceptions
If the user choose not to view the message right away, it needs to be available for later viewing.

2.3.5 limitation

The exceptions in each use case will not be taken into account in the further development of the application because the functionality is prioritized above handling exceptions.

2.4 Activity diagram

Both the contextual scenario and the use case describes the system from the user's point of view, the diagram 2.5 on the following page shows the different functions of the system. The diagram is separated in a sender and receiver system even though the final system will include both.

When the user has written a message it is compressed and converted into a bit sequence which the system interprets as a call sequence which is used in the transmission of the message.

On the receiver side the system is waiting for the first call. When a call is registered the system hangs up and waits for the next call. When the system has received all the calls, they are interpreted into a message which is then decompressed.

2.4. ACTIVITY DIAGRAM

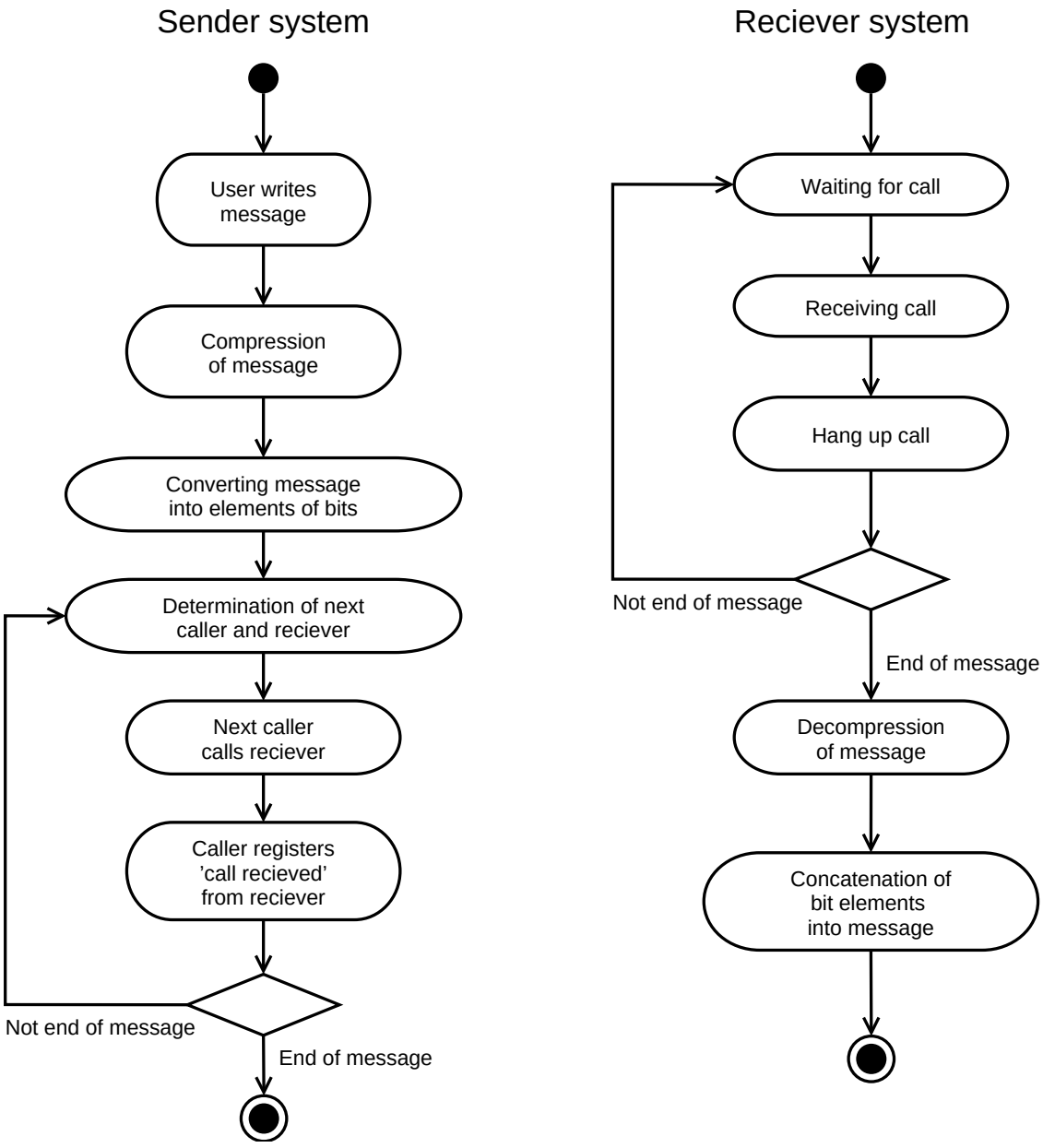


Figure 2.5: Activity diagram of the different tasks of the system

2.5 Requirements Specification

The requirements to the system descried below is defined and specified based on the previous sections. The requirements to the GUI is mainly found from the use case analysis and the functional requirements is found from the activity diagram.

2.5.1 Requirements to GUI

To summarize from the use case analysis, the user needs a menu to navigate, which contains an editor for writing messages, an inbox for received messages, and a part for setting up the phone correctly.

- GUI1: The GUI must contain an editor where the user can write a message
  - GUI2: It must be possible to write at least 160 characters in the editor
  - GUI3: The editor must support T9-dictionary
  - GUI4: It must be possible to send the message from the editor
- GUI5: The user must be able to view an incoming message
- GUI6: The user must be able to view a list of received messages
- GUI7: The user must be able to set up the phone to be master or slave in a settings menu
- GUI8: From the settings menu, the user must be able to change the bluetooth MAC address of the local partner
- GUI9: From the settings menu, the user must be able to change the phone numbers of the remote master and slave

2.5.2 Functional Requirements

The activity diagram shows the functions of the system, and these are described below through different requirements.

- These requirements is for both the receiver and the sender side of the system:
  - FR1: The system must run on mobile phones with Symbian (S60-series) OS

  - FR2: The system must connect the phones and exchange data via bluetooth
  - FR3: The system must have the ability to compress/decompress the message
- Requirements for the sender side
  - FR4: The system must be able to make a phone call
  - FR5: The system must register that the receiver have received the call
  - FR6: The system must convert the message into a bit sequence
  - FR7: The The system must know which phone is the next to call from the bit sequence
- Requirements for the receiver side
  - FR8: The system must be able to register an incoming call
  - FR9: The system must be able to hang up immediately (in one second)
  - FR10: The system must be able to identify an incoming call
- FR11: The system must be able to send a message of 20 characters in 3 out of 5 attempts.

The acceptance test specification is set up in appendix A.1 on page 76. This specification describes how to test the different requirements and a summarized conclusion of this can be found in section 6.3 on page 67.

Chapter 3

Technical analysis

In this chapter the technical aspects of the project are reviewed. The operating system of the phones used, are explained to show the benefits of this operating system. Different programming languages and scheduling principles are described to make the best selection for the intended application. Finally a test is made to show the delay in the mobile network service.

3.1 Symbian phones

In this section the phones used in conjunction with this project and the operation system (OS) of these will be examined. The phones used are Nokia 6600 and Nokia N70. These use the Symbian OS which will be explained later on in the chapter.

Smart phones come in many different variants manufactured by a number of different companies. They generally have large color displays, cameras and often a wide variety of entertainment abilities such as games, video and mp3-capability. The smart phone marked is at the moment primarily dominated by Nokia and their Series 60 phones. The Series 60, 80 and 90 utilizes the Symbian OS, an operating system developed by the major actors in the field of mobile phones, e.g. Nokia, Samsung and Siemens.

The operating system used on the new smart phones is, as previously mentioned, the Symbian OS which gives the phones a modular built structure. This enables fast development of new capabilities and allows the developed systems to be used on various phone types, not only Nokia or Samsung, without having to recode the entire program. The Symbian OS is the most commonly used OS on the smart phone mobile market, accounting for 66.5% of the marked see table 3.1 on the next page. Of these 66.5% Nokia has a 47.9%

OS vendor	Q1 2005	% share	Q1 2006	% share	Growth Q1 05/Q1 06
Symbian	6,618,370	72.7%	11,504,920	66.5%	73.8%
Linux	1,704,270	16.5%	3,378,320	24.4%	98.2%
PalmSource	474,120	5.2%	543,840	3.1%	14.7%
Microsoft	338,650	3.7%	411,740	2.4%	21.6%
RIM	90,320	1.0%	599,530	3.2%	518.5%
Others	83,570	0.9%	65,728	0.4%	-21.3%
Total	**9,309,300**		**16,464,078**		76.9%

Table 3.1: Worldwide total smartphone device market - Market shares 2006 Q1 05 / Q1 06 [Sym06a]

share see figure 3.1, making them the driving force in the Symbian developement. The OS makes it possible to program C++, Java and Python onto the phone.

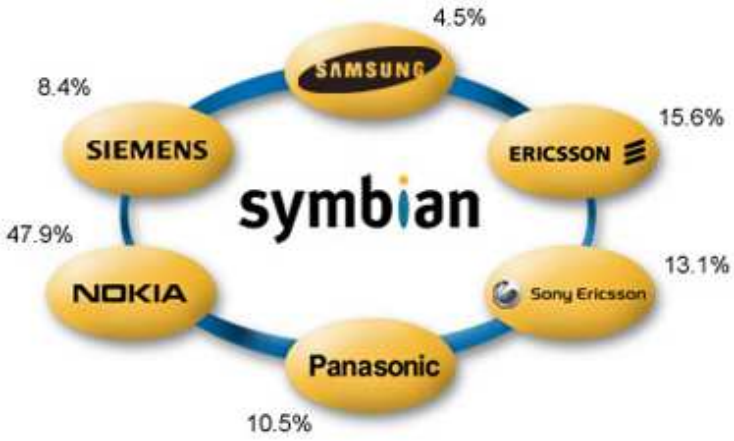


Figure 3.1: The diagram shows the different Symbian developers and their share [Sym06b]

Nokia has different Symbian phones in the series 60, 80 and 90 intended for different market segments. The key features of the series 80 and 90 is large screens (up to 640 x 320 pixels) and touchscreens for the business segment where the series 60 is for the general user.

3.1.1 Phone specifications

The series 60 phones used in this project is Nokia 6600 and Nokia N70 as shown in figure 3.2. The specifications are shown in table 3.2 on the next page.

Figure 3.2: Nokia 6600 and Nokia N70

	Nokia 6600	**Nokia N70**
Processor:	104 MHz ARM	220 MHz ARM
Memory:	6 MB internal	20 MB internal
Display:	176 x 208 pixels, 16 bit colors	176 x 208 pixels, 18 bit colors
Symbian:	v7.0s (2nd. ed.)	v8.1a (2nd. ed. FP3)

Table 3.2: The specification for the Nokia 6600 and N70 phones [Nok06a] [Nok06b]

3.2 Programming languages

To implement the system specified in the requirements specification in section 2.5 on page 21, it must be decided which programming language is the most suitable for this purpose. On the Symbian S60 platform there are several possibilities of development. Three of them will be taken into consideration: Symbian C++, Java 2 Micro Edition and Python. Symbian C++ and Java are widely distributed and Python was released for Nokia phones only a few years ago. Every language is very well documented and several tutorials are available for each of them as well as large communities with forums and discussion boards.

In this section the three languages will be briefly described to determine the advantages and disadvantages in relation to this project. They all provide access to the functionality needed for the implementation which is:

- Local wireless connectivity (Bluetooth)
- Graphics for GUI construction
- Threads for parallel processing
- Interface to the GSM network (Making phone calls)

3.2.1 Symbian C++

C++ is a language most programmers are familiar with and it is very versatile and can be implemented onto a variety of platforms, making it ideal for practically any purpose. It is the natural choice for application development on the Symbian platform and it lets the programmer access every level and functionality of the system. This leaves the programmer in complete control of the interaction between the application and the operating system.

Especially time critical applications can be made very efficient in C++ because the source code is compiled to the specific operating system. This is also a downside as the application needs to be recompiled when the platform is changed. Also the programmer must understand the Symbian OS to make an application work properly, and it can be very hard to learn this [Wik06].

3.2.2 Java 2 Micro Edition

Java 2 Micro Edition (J2ME) is a platform for developing applications to memory limited devices e.g. mobile phones. The platform is available on a majority of the mobile phones on the market today where it is mostly used for games and other graphical applications. It has an average runtime speed due to the compilation into Java bytecode, but this also makes it portable to other devices without recompiling the source code [Nor03].

3.2.3 Python

The programming language Python bears some similarity to pseudo code, making it very fast and easy programming applications. With little training, it is possible to make a Python script for a mobile phone with a GUI containing menus and editors for accessing e.g. system information and SMS sending. Python is an interpreted script language which makes the runtime speed below average but no compilation is needed. Python for mobile platforms is still in its early stages and only available for Nokia series 60, but its user-friendly approach should secure future spread to other platforms and devices.

Python is great for developing software prototypes which is the goal of this project, but it still lacks a lot of system functionality which is not included in the current edition (version 1.3). This problem can be solved through the Python/C API which allows the programmer to extend the Python standard library with C or C++ modules to gain access to e.g. Symbian OS functionality.

3.2.4 Selection

The programming languages discussed above is all useful for developing a prototype of the desired application. Table 3.3 on the next page shows a comparison between them. Symbian C++ is of course the best choice for an optimal application but it is also very extensive for developing a small prototype application. Java is less extensive but still requires insight in object oriented programming to utilize system functionality. Python is easy and fast to develop but lacks functionality.

Python has been chosen for the implementation because of the simple syntax, little complexity and the fact that it is possible to make extension modules in C/C++ for the lacking functionally, creating a hybrid between Python and Symbian C++. The low Python runtime speed is not an issue as the final application will have a lot of idle time due to the time it takes to establish a phone call between the sender and the receiver. The time to establish a call between two mobile phones is measured in a test described in section 3.5 on page 37.

3.3 Python for Series 60

Python is an interpreted programming language developed since 1990 [Fou05]. It is an easy and flexible language that applies to many different platforms and purposes. The Python for Series 60 Platform (PyS60) is a new branch for Nokia mobile phones and an example of the increasing mobility of the Python programming language. The first version of PyS60 was released from Nokia in 2004 and with the current version (1.3.1) the source code has been released for open source development. PyS60 version 1.3.1 is based on Python version 2.2.2.

PyS60 offers many build-in modules which are useful and makes it easy to develop applications without knowledge of the underlying system. PyS60 does not includes all standard modules, but many of them (some with modifications and extensions) plus special modules for using the native resources of the S60 mobile phones [Nok05]. This section describes some of the modules in PyS60 which will be used in this project, how they are beneficial and how their functionality on some points are limited.

3.3.1 Appuifw module

The appuifw module constitutes with the e32 module the two built-in extension in PyS60. This means that they are loaded together with the PyS60 interpreter and can be used right away. The appuifw module provide access to the UI framework which contains many of

	Symbian	**J2ME**	**PyS60**
Foundation	C++	Java	Python
Learning Curve	Steeper	Average	Excellent
Cross-Platform Deployment	Compile per target	Excellent - Bytecode	Interpreted language only on Nokia Series60
Graphical Interface	2D, 3D Graphics (newer phones) Many Widgets, Visual Form-Based GUI Builder	2D, 3D graphics, Many widgets, Visual Form-Based GUI Builder	2D Graphics access, some simple widgets
Functionality	No restrictions	Varies by handset - dependent on available included JSR's. No high-resolution pictures, No Cell ID, limited file access	Partial through API: High resolution pictures, Cell ID
Runtime Speed	Best (Compiled language), lots of memory management options	Average due to Java bytecode	Below Average due to Interpreted language

Table 3.3: Comparison between programming languages for mobile platforms [Wik06]

the UI widgets known from Symbian OS. It allows the programmer to set up menus and dialogs or show notes to the user on the display. The services from this module may only be used in the main thread of an application, often referred to as the UI thread, to avoid conflicts on the access to the display. Figure 3.3 shows the layout of the UI in the normal screen mode and how to access the varies areas on the screen.

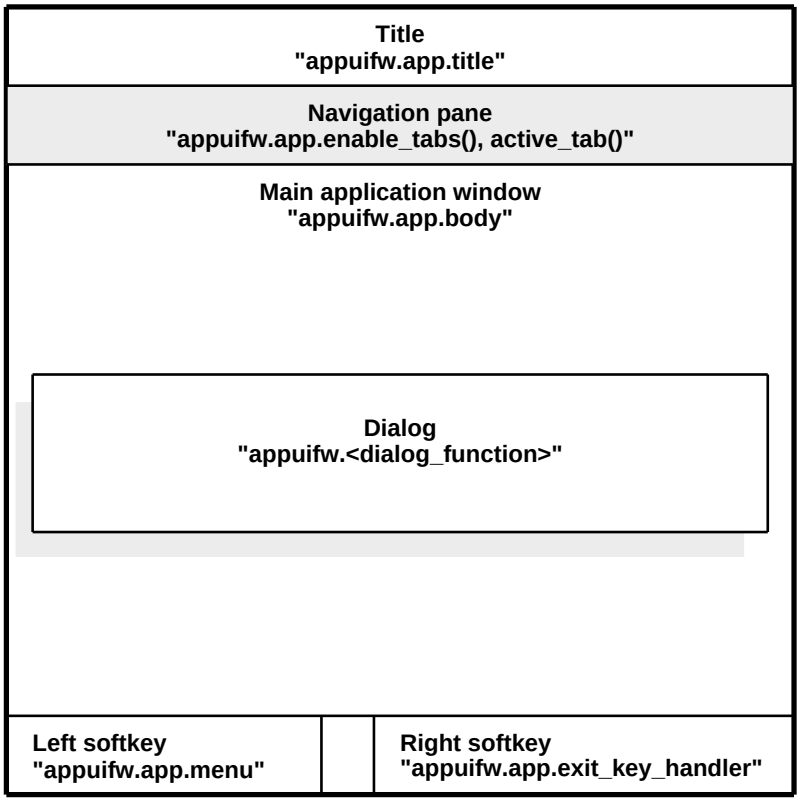


Figure 3.3: Screen map of a S60 phone with appuifw objects[Nok05]

The UI is set from the application type within the appuifw module (appuifw.app). A description of the expressions from figure 3.3 is shown below:

- `appuifw.app.title` sets the title/headline of the application
- `appuifw.app.enable_tabs` creates a multi-view application with tabs for the user to scroll through.
- `appuifw.app.body` defines the body of the application. E.g. plain text or a canvas for graphic elements.
- `appuifw.app.<dialog_function>` shows a dialog. E.g. a note or a query to allow the user to type inputs. Dialogs are always on top of the application body

- `appuifw.app.menu` and `appuifw.app.exit_key_handler` defines the action of the right and left softkeys. E.g. a popup menu or a callable object.

3.3.2 e32 module

The e32 module provide Symbian related utilities to communicate with the Symbian active scheduler and active objects on the phone. It also provides access to information about which PyS60 version is currently running and which version of Series 60 it is running on. Some examples of module level functions are [Nok05]:

- `ao_sleep(` *`interval`*`)`: Makes a process or thread sleep for a given *interval* without blocking the active scheduler or the UI.
- `pys60_version` : Returns the version number of the PyS60 currently running.
- `in_emulator()` : Checks if the an application is running in an emulator or on a phone.

The e32 module also defines the type `Ao_lock()` which is a very useful way of creating locks (semaphores) in a multi threaded application on the S60 phones. The `Ao_lock()` has the methods `wait()` and `signal()` for locking and unlocking the lock. When `wait()` is called, the thread that called it, is blocked from running until the the lock is signaled by another thread. Meanwhile other threads and active objects is serviced by the scheduler. This is useful when the application must wait for a user input, be prevented from exiting or if multiple threads needs access to the same data. (Read more in the section 3.4 on page 33).

3.3.3 Telephone module

The telephone module is a dynamically loadable module. In order to be used it must first be loaded with the Python import statement (`import telephone`). The module provides an API to the telephone functionality. The methods available in this module are [Nok05]:

- `dial(` *`number`*`)` dial a number given as the string *`number`*.
- `hang_up()` hangs up a call made by `dial()` .

It is not possible to use `hang_up()` for hanging up incoming calls or other outgoing calls not made by `dial()` . This makes telephone module very limited and insufficient for use

in this project as it is necessary to have a functionality for hanging up incoming calls. The `dial()` function however is suitable for use.

3.3.4 Socket module

The socket module is a standard module in Python which allows low level networking. In PyS60 the socket module is extended with bluetooth support. It is possible to communicate in two ways over bluetooth either by simulating a RS-232 serial connection by RFCOMM or to send binary content via an OBEX connection. RFCOMM will be used to connect the phones because it is more suitable for the task of sending text. This means that OBEX is not discussed any further.

Sockets are created as objects which have a numbers of methods to control the connection. These are all defined in the standard socket module and some examples of them are [Nok05][Fou05]:

- `bind(` *`address`*`)` Binds the socket to *`address`*.
- `listen(` *`backlog`*`)` Listens for incoming connections to the socket. The *`backlog`* argument specifies the maximum number of queued connections and should be at least 1.
- `connect(` *`address`*`)` Connects to a remote socket on the specified address.
- `accept()` Accepts the connection.
- `send(` *`string`*`)` Sends the specified string.
- `recv(` *`bufsize`*`)` Receives data from the socket connection. *`bufsize`* specifies the maximum data amount that can be received.
- `close()` Closes the socket.

Some examples of the added bluetooth methods are:

- `bt_advertise_service(` *`name`*`,` *`socket`*`,` *`flag`*`,` *`class`*`)` Advertises the service *`name`* on the channel bound *`socket`*. The service is turned on/off by setting *`flag`* True/False. The service is an OBEX or RFCOMM service defined in *`class`*.
- `bt_discover([` *`address`*`])` Search for other bluetooth devices. If *`address`* is not specified the BT device address and a dictionary of services of the remote devices is returned.

- bt_rfcomm_get_availabe_server_channel(*socket*) Gets the available RFCOMM server channel for the specified socket.

3.3.5 Thread module

The thread module is also a part of the standard Python. The module makes it possible to have tasks running simultaneously and parallel. Threads are also useful for running methods with blocking behavior in context of the main application, such as reading from a socket or waiting for other asynchronous input. There are certain limitations on what can be shared between threads, e.g. sockets. This means that it is not possible to send and receive on the same socket from two different threads. In PyS60 the appuifw module can only be used from the main thread.

Some of the most important functions to control thread execution are:

- start_new_thread(*function, args*) starts a new thread calling *function* with the tuple *args* as arguments.
- allocate_lock() returns a new lock. The lock is initially unlocked.
- acquire() acquires the lock when it is called. A lock can only be acquired by one thread at a time.
- release() releases the lock. The lock has to be acquired previously by a thread (not necessary the same thread).
- locked() returns the state of the lock, 1 if it has been acquired 0 if not.
- exit() causes the thread to exit.

3.3.6 Extensions

As mentioned in section 3.2.3 on page 26 the Python/C API makes it possible to write extensions in C/C++ to the standard library if a desired functionality is lacking. Appendix C on page 99 contains a howto for writing a simple PyS60 extension. Before an extension is written it is necessary to consider several issues as:

1. Conversion of I/O variables and objects between PyS60 and the extension code.
2. Preparation of the extension code so it is visible and callable from PyS60.

3. Release of the PyS60 interpreter lock if blocking calls is made from the extension code.

The extension example in listing 3.1 shows issue 1 and 2:

Listing 3.1: Source code for a simple extension

```
1  static PyObject* sum_args(PyObject*     /*self*/, PyObject* args){
2    int number1, number2;
3    if (!PyArg_ParseTuple(args, "ii", &number1, &number2)){
4      return NULL;
5    }
6    int sum = number1 + number2;
7    return Py_BuildValue("i", sum);
8  }
9
10 static const PyMethodDef sum_methods[] = {
11   {"sum", (PyCFunction)sum_args, METH_VARARGS, "sums numbers"},
12   {0, 0}
13 };
14
15 DL_EXPORT( void) MODULE_INIT_FUNC(){
16   Py_InitModule("summe", sum_methods);
17 }
```

Line 4 uses the function `PyArg_ParseTuple()` to convert the Python input to two C-integers. Lines 10-13 is a list of functions available in the extension module and an initialization function which makes the extension visible from PyS60. For more information see appendix C on page 99.

Issue 3 is not shown in listing 3.1 because there is no blocking calls in that extension code. Otherwise the release of the interpreter lock is done as in listing 3.2:

Listing 3.2: Interpreter lock release

```
1 Py_BEGIN_ALLOW_THREADS
2 /* Blocking call */
3 Py_END_ALLOW_THREADS
```

3.4 Scheduling

In complex systems there are often a need for several processes running simultaneously. Therefore it is important to schedule how the processes are completed to meet the dead-

lines in the system.

If the processes have less time to run than required, it may cause a system error that may crash the system. Therefore the minimum running time of all processes has to be ensured by scheduling the system.

To schedule the system a number of different terms are available which make the completion of the processes as smooth as possible.

3.4.1 Interrupt

Interrupts can be used for time critical processes. If a process needs to run it can send an interrupt which means that the process currently running is put on the stack until the process which asked for the interrupt is finished. On figure 3.4 it can be seen that an interrupt call is interrupting the main program and that it takes less than 5 μ seconds to start the interrupt routine and the same time to end it [Nie06b].

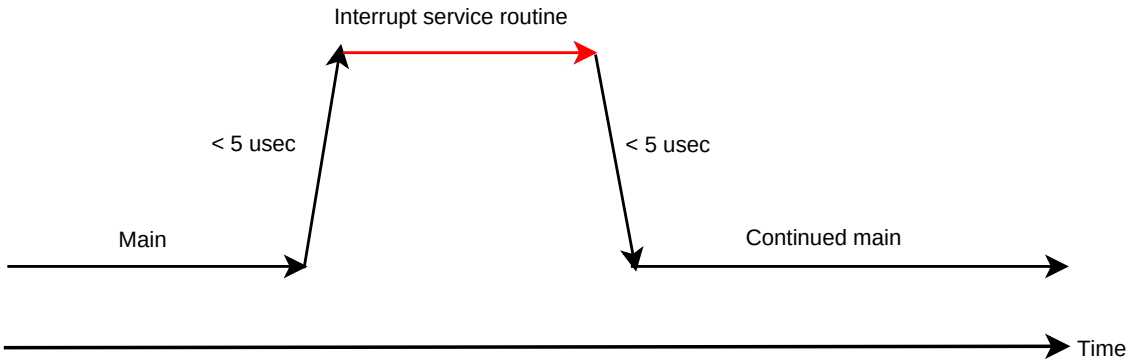


Figure 3.4: Interrupt service routine

3.4.2 Parallelism trough threads

Threads can be used to make processes run virtual at the same time instead of having a sequential program running that has to finish a process before another can be started. Threads are often used to divide tasks into subtasks that runs independently in context of a main thread that waits on the results from the other threads. Modern operating systems are using timesharing to give the single threads runtime on the CPU. Threads containing I/O operations are relatively slow compared to the runtime of the CPU which lets the threads appear to run parallel from the users point of view. The system time is divided

into time slices as seen on figure 3.5. This gives the different threads time to run and they can be prioritized by scheduling the system.

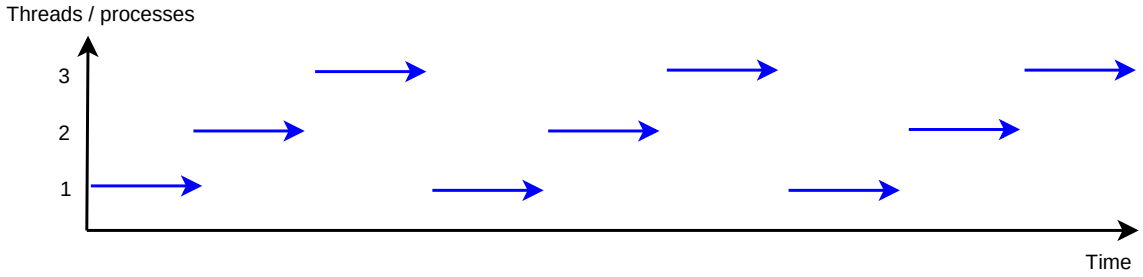


Figure 3.5: The CPU time is divided into time slices which gives the threads run time

Threads are not making processes / tasks running parallel but if they are scheduled correct it can be referred to as potential parallelism. If threads should be parallel in reality they would need a CPU for each thread in the program [MH05].

3.4.3 Semaphores

Semaphores are often referred to as a mutex (mutual exclusion-lock) and are used in synchronization problems e.g. avoiding threads from sharing the same resources or variables at the same time and preventing threads from doing non intended processes at the same time. A semaphore has the operations wait and signal and a counter. If a thread decrements the counter the thread is blocked until another thread unblocks it by incrementing the counter. `Wait` decrements the counter and `signal` increments it. When a thread is blocked it means that it notifies the scheduler that it should not proceed until it gets the signal command. A normal semaphore can unlike a thread lock be de / incremented more than once which means it may need to be signaled several times to proceed. This can be seen on figure 3.6. The thread lock as mentioned before has a boolean value 0 or 1 respectively `wait` or `signal` .

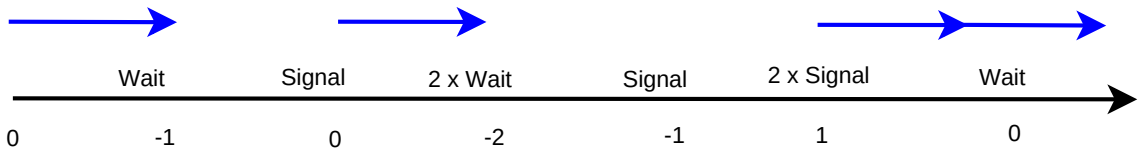


Figure 3.6: The thread is allowed to continue if the semaphore is 0 or positive

3.4.4 Scheduling principles

To ensure the runtime in the system it can be scheduled. There are numerous algorithms to schedule a system and the main principles can be categorized as following.

- **Fixed schedules**
  Schedules derived at pre run-time, rather simple and effective, but less flexible.
- **Round robin**
  Fixed time slots are granted periodically, simple and flexible but less effective.
- **Fixed priority**
  CPU-time is granted to the highest priority among active tasks, simple and flexible, medium efficiency.
- **Dynamic priorities**
  CPU-time is granted according to runtime criteria, complex, flexible and effective.

  [Dor02]

In the proposed system for this project there are a number of processes. In order for these processes to run smoothly, they need to be given sufficient time to execute. The processes that will make up the program can at this point be listed as:

- The communication between master and slave, via bluetooth
- The conversion of the message to and from bit elements
- The call ordering that is established to send the bit elements
- The actual call between the sender phones and the receiver phones

The conversion and call ordering can be done prior to the start of the transmission of the message, and are therefore not time dependent. The communication and the actual calls do however require a longer period of time to execute and new calls must wait for the current call to be completed. This means that the system has a lot of idle time which can be used for other processes and no time critical scheduling is needed.

3.5 Testing the delay in the mobile network

The purpose of testing the delay in the mobile network service is to determine the average time for a call from one phone to another. This is necessary for the design of the application. The test is a black-box test, because the communication between the mobile phones are unknown. Furthermore it is not possible to tweak the communication to benefit the project, so it is therefore necessary to know the limitation of the mobile network service.

The test is performed at different times of the day, to determine the worse case scenario that may occur during the peak period(s) of the day. It will also be run on phones with sim cards from different countries.

To execute the test there has been developed two PyS60 programs, one for the caller phone and one for the receiver phone.

3.5.1 How the test programs work

The program on the caller phone runs in a loop and calls the receiver phone every time the phone is idle, and ready to establish a new call. Because the call status occasionally returns a ready to establish a new call status, before the phone actually is ready, due to the processes running in threads, the program sleeps for 1 second before establishing the next call.

The program on the receiver phone runs in a loop as well, listening for incoming calls. If a call is received the program writes the phones UNIX time into a log on the phone and includes the phone number of the caller. It then hangs up the incoming phone call, and listen for new incoming calls.

3.5.2 Results

The test results are based on the log files created on the mobile phones by the test programs running for more than 8 hours making more than 600 calls. From the test it is possible to conclude the average time for establishing a call between two phones located in Denmark with German sim cards is 34,57 seconds, see table 3.4 on page 39. The worst-case connection time on the accepted calls is 57 seconds, but it should be noticed that 31,5% of the calls made between the two phones are recorded as errors.

The test has also been run on two phones, where both phones are located in Denmark and the caller has a German sim card and the receiver has a Danish sim card. The average time for establishing a call is 26,51 seconds, see table 3.5 on page 39. The worst-case

connection time on the accepted calls is 64 seconds, in this test the error rate of the calls being made is 44%.

The call test has also been run on two phones, both located in Denmark and using Danish sim cards. The average time for establishing a call is only 17,48 seconds, see table 3.6 on the facing page. The worst-case connection time on the accepted calls is 26 seconds, and in this test the error rate was 0%.

It should be noticed that the time it takes to establish a call between to mobile phones is depending on where the sim cards are from. The fastest way of making a call in Denmark is to use two Danish sim cards. The reason for the high error rate is properly caused by an error in the PyS60 function `telephone.dial()` . When the sender program is idle and ready to make a call, it occasionally does not start dialing for 60 seconds but stays idle. If a user presses a key on the phone while it is idle it starts dialing immediately.

The German sim cards have a higher failure rate due to problems of getting through the network. This failure results in a longer call time which gives a higher chance of a failed call. In appendix E on page 108 several figures are shown to give a better overview of the test results. Figure E.1 on page 108, E.2 on page 109 and E.3 on page 109 shows the results from each test. Figure E.4 on page 110 shows a comparison between the three tests.

Time of test	22-04	2006
Duration of test	111,92	min
Sim cards	DE to DE	
Phones	Nokia 6600	
# of usable data	89	
# of non-usable data	41	
Average call time	34,57	sec
Shortest call time	25	sec
Worst case	57	sec
Error rate	31,54	%

Table 3.4: The results of the call test between two phones using German simcards

Time of test	23,24 and 30-4	2006
Duration of test	316,77	min
Sim cards	DE to DK	
Phones	Nokia 6600	
# of usable data	235	
# of non-usable data	168	
Average call time	26,42	sec
Shortest call time	18	sec
Worst case	56	sec
Error rate	44,01	%

Table 3.5: The results of the call test between two phones using a German and a Danish simcard

Time of test	05-05	2006
Duration of test	43,98	min
Sim cards	DK to DK	
Phones	Nokia 6600	
# of usable data	151	
# of non-usable data	0	
Average call time	17,48	sec
Shortest call time	16	sec
Worst case	26	sec
Error rate	0,00	%

Table 3.6: The results of the call test between two phones using Danish simcard

Chapter 4

Design

As the design is made it needs to fulfill the requirements specified in section 2.5 on page 21 otherwise the final product might differ from the intended purpose for the application. A requirement set in the requirement specification is the compression of the message. An application named smsZipper contains a module for compressing text which can be used. It interfaces directly to the standard editor for creating messages and should be a selectable menu point when pressing ”send”. The compression functionality is an optimization and will not be implemented at this point because the focus is on developing the transmission method.

The main purpose of this program is to send a message from one group of telephones to another. This must be done by making phone calls in a pattern specified by the message. In the design considerations it is important to take account of the fact that no data is being transmitted between the two groups of phones. In this project it has been chosen to organize the design phase in 3 different parts:

1. Sender side
2. Receiver side
3. GUI

The same application should be able to both send and receive a message thus the correct part of the program must be initialized when needed. On the sender side this is done when the user chooses to send a message. On the receiver side it is done when the phone receives the first call which must be regarded as a initialization call. This initialization takes place on the master phones on each side which subsequently must inform their slaves about the pending transmission.

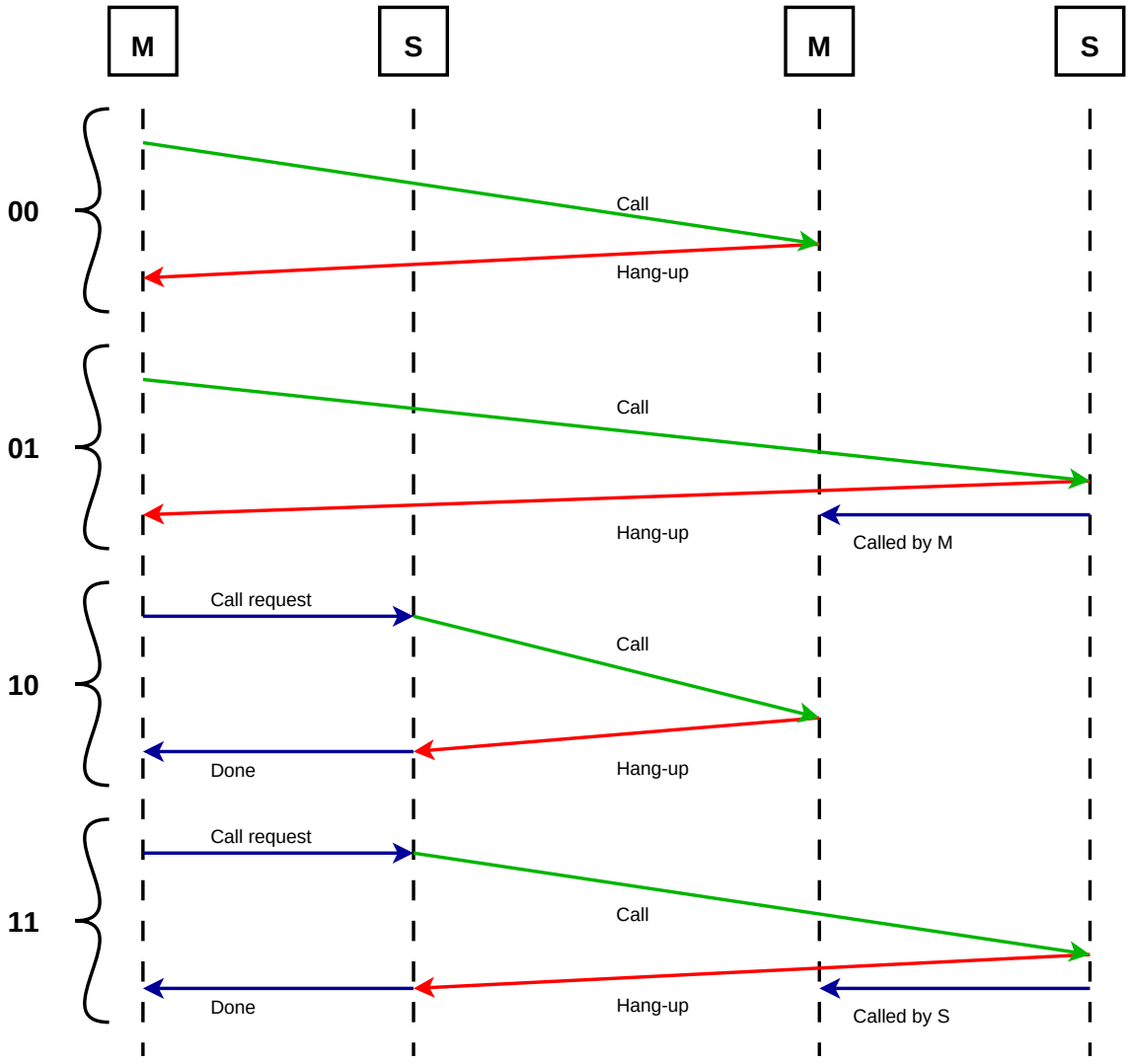


Figure 4.1: The protocol of the transmission with the sender on the right and the receiver on the left, both with a master and a slave (M and S)

When the transmission is initialized the message must be sent via a common protocol which is visualized in figure 4.1 on the previous page. The message is sent two bits per phone call as both the caller and the callee represents one bit each.

The bits are interpreted as follows:

00 : Master calling master

01 : Master calling slave

10 : Slave calling master

11 : Slave calling slave

Whenever a slave must make a call it must be told to do so by its master and if a slave receives a call it must inform its master about it. This is done via a bluetooth connection between the master and its slave indicated by the blue arrows in figure 4.1 on the preceding page. The slaves must either execute orders from the master (making calls) or inform about incoming calls thus the slaves must be ready for communication before the transmission starts. The transmission of the message is described in the following sections 4.1 and 4.2.

At the end of a transmission a signal is needed to inform the receiver side that the entire message is transmitted. This could be done by sending an ending character e.g. a newline, or it can be done by a timeout. The ending character-method has the advantage that it is very robust compared to the timeout, which might give timeout if a call is delayed or if it fails. Yet an ending character requires more calls and a long time to send thus the timeout-method is chosen for this application. The timeout should be chosen according to the time for establishing a call. From the results of table 3.5 on page 39 it can be seen that the average time is 27 sec with a maximum time of 56 sec. The timeout should therefore be set to 60 sec when calling from a German to a Danish sim card.

4.1 The sender side

On the sender side, the transmission is started when the user chooses to send the message. This executes the send-method which runs the sequence of steps shown in figure 4.2 on the facing page. The following entries corresponds to the boxes in the figure.

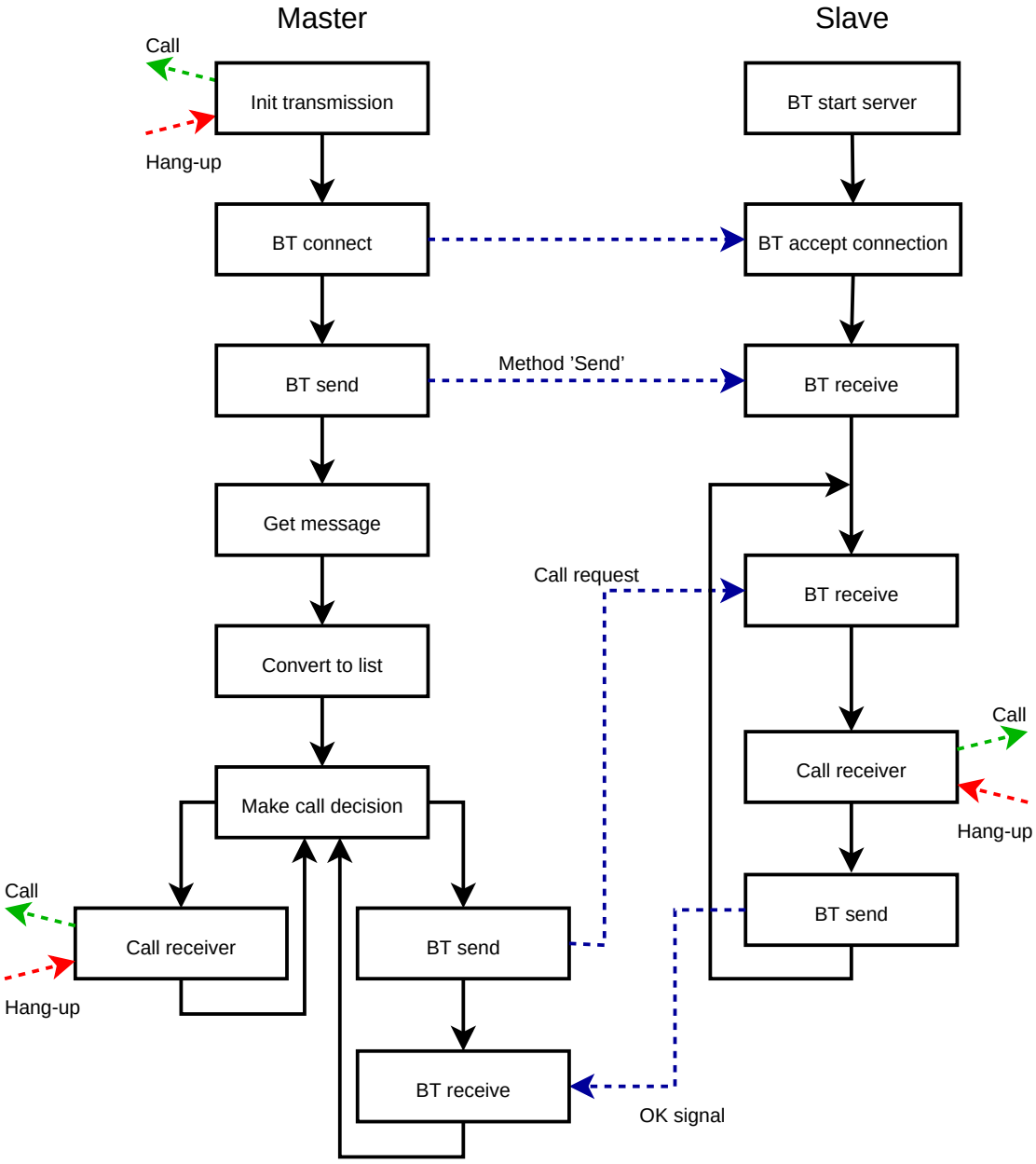


Figure 4.2: Flowchart of the system on the sender side with communication between master and slave

Master

Init transmission When the user chooses to send the message the send-method makes a call which initialize the transmission on the receiver side.

BT connect: The master connects to the bluetooth server started by the slave.

BT send To inform the slave about the pending transmission the master connects via bluetooth and sends a signal which indicates that the slave should participate in the process of sending a message.

Get message: The message is fetched from the editor

Convert to list: The message is converted into a list containing elements of two bits which defines the caller and callee of each call in the transmission.

Make call decision: The list of bits is interpreted one element at the time to decide the next caller and callee.

Call receiver: If the master is the next caller, it calls the callee on the receiver side, waits for a hang-up signal and returns to make the next decision.

BT send: If the slave is the next caller, the callee's phone number is send to the slave.

BT receive: The master waits for an OK signal which means that the slave has finished the call.

Slave

BT start server: When the user chooses to use the phone as a slave a bluetooth RF-COMM server is started.

BT accept connection: The server is listening for and accepting incoming connections.

BT receive: The slaves receives the signal which indicate that it should participate in a sending process.

BT receive: The slave waits for incoming bluetooth transmissions until it receives a phone number of the next callee.

Call receiver: The slave calls the number received via the bluetooth connection with the master and waits for a hang-up signal.

BT send: When the slave gets a hang-up signal from the callee it sends a signal to its master to inform about the finished call. Then it returns to listen for new phone numbers.

At the end of the transmission the master sends an end signal to the slave which causes the slave to close the bluetooth connection. This is not included in figure 4.2 on page 43 to keep it simple and avoid confusion.

4.2 The receiver side

On the receiver side the transmission starts when the master receives the initialization call. This executes the receiver-method which, like the sender-method, controls the sequence of steps needed to complete the process. Figure 4.3 on the following page shows the sequence of steps needed on the receiver side.

Master

Wait for call: If the application is idle in the main menu it is ready to receive a message.

Get caller ID: The transmission must start with the initializing call from the master on the sender side thus the incoming call must be correct.

The receiving process on the master must be designed with parallel processing as the master must listen for and process incoming calls both on itself and on the slave. This may result in blocking calls which must be separated process phone calls when they occur. This is done by dividing the tasks into threads.

BT connect: Like on the sender side the master connects to the slave via bluetooth.

BT send: The master sends the signal which indicates that the slave is part of a receiving process.

BT receive: The master listens for incoming signals on the bluetooth connection.

Wait for call: The master listens for incoming calls and hangs up.

Get caller ID: The caller ID is fetched from the phones caller log.

Derive bits: A bit element of two bits is created from the caller and callee and appended to a list of bit elements. To determine if the transmission is over, a timestamp must be set after each incoming call. This can be compared to the current time before the

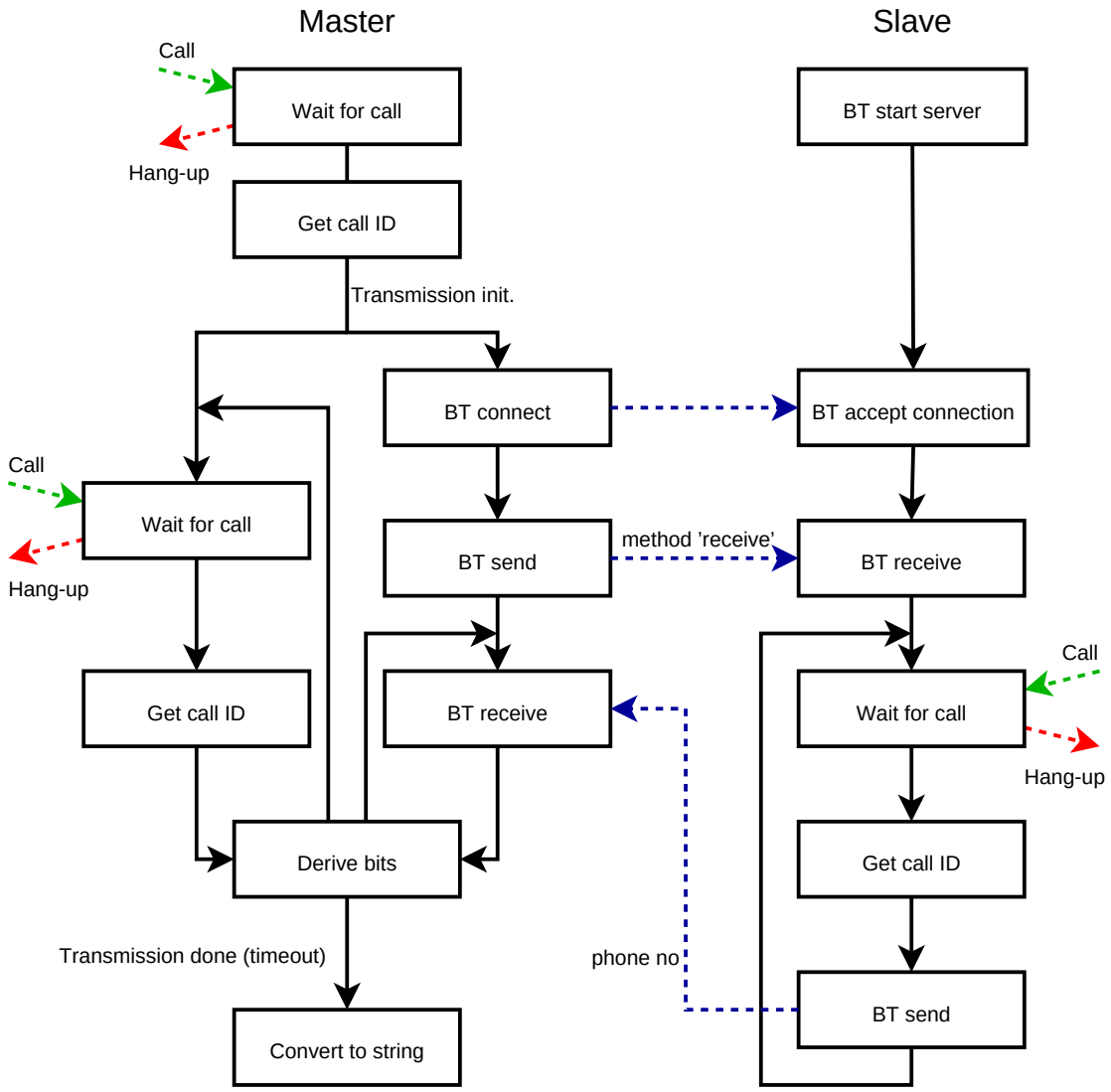


Figure 4.3: Flowchart of the system on the receiver side with communication between master and slave

application waits for new calls. If there is no timeout, the application return to wait for incoming calls and signals on the bluetooth connection.

Convert to string The list is converted into a string when the transmission is over, which makes the message readable to the user.

Both the thread that waits for calls, and the thread that listen on the bluetooth connection is accessing the functionality 'Derive bits' through which both threads can write to the same list in the memory. To avoid simultaneous access to this memory a semaphore should be created to protect it. This is not necessary because this functionality is only used after incoming calls which never occurs more than once every 15 seconds.

Slave

BT start server: Like on the sender side, the slave starts a bluetooth server when it is started.

BT accept connection: The slave accepts an incoming connection from the master.

BT receive: The slave receives a signal to start listening for incoming calls.

Wait for call: The slave waits for incoming calls and hangs up when a call occurs.

Get caller ID: The caller ID is fetched from the phones caller log.

BT send: The phone number is send to the master.

Like on the sender side the end of the transmission is not included for the slave, but is described below:

A timestamp must be set after the bluetooth transmission. This can be compared to the current time to determine whether the slave has timeout while waiting for incoming calls. When a timeout occurs, the slave must send a request to the master asking if the timeout is real. The slave can only reach timeout after the master reaches it, thus the slave must ask for timeout often to reach timeout as close to the master as possible. This is important because the slave is running the bluetooth server and have to be the one to shut it down.

4.3 Graphical User Interface

The programs user interface will be structured as a menu the user can navigate through. This allows the user the access different features in the application. In response to the requirements specification following features need to be available to the user:

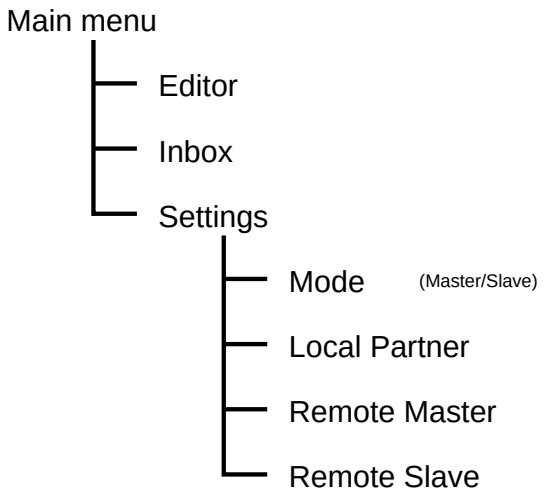


Figure 4.4: The menu of the application

- Editor (Writing and sending the message)
- Inbox (Viewing incoming messages)
- Settings (Specifying the settings of the system)

A full menu tree on the master phone is shown in figure 4.4. On a slave phone the entries ”Editor” and ”Inbox” will not be available as the only purpose of a slave phone is to provide services to the master e.g. making and receiving calls.

The entries under settings have the following options:

- **Mode** is either master or slave. This entry determines which entries are available under the main menu. The following entries is only used by the master.
- **Local Partner** contains the MAC address of the other phone used to send or receive (the slave).
- **Remote Master** contains the phone number of the master phone to send to or receive from.
- **Remote Slave** contains the phone number of the slave phone to send to or receive from.

4.4 Input/output table

To provide an overview of the system functionality and define the I/O of the different parts of the application, a design table has been made. The various modules in table 4.1 match the entries in section 4.1 on page 42 and 4.2 on page 45. The parameters of the table are further described in chapter 5 on the next page.

Module	Input	Output	Access
Init transmission	-	-	phone no. (remote master)
BT start server	-	-	-
BT connect	slaves MAC-address	socket object	-
BT accept connection	-	socket object	-
BT send	socket object, string ("OK", "send", "receive", phone no.)	-	-
BT receive	socket object	string ("OK", "send", "receive", phone no.)	-
Get message	-	string (message)	Contents of the editor
Derive bits	phone no. (remote master/slave), callee ("m", "s")	-	List of bit elements
Convert to list	string (message)	List of bit elements	-
Convert to string	List of bit elements	string (message)	-
Make call decision	List of bit elements	-	phone no. (remote master/slave)
Call receiver	phone no. (remote master/slave)	-	Status of phone line
Wait for call	-	-	Status of phone line
Get caller ID	-	Caller ID	-

Table 4.1: Input/output table for the modules of the application

Chapter 5

Implementation

In this chapter the implementation of the application will be described. First the different functionalities of the GUI will be explained which includes the editor, the inbox and the settings menu. Then functionalities that are needed for the bluetooth connection and for the bit mapping will be explained and finally a description of three modules which extends PyS60 with call functionalities that are not included.

To give a better overview only the code that is needed for explaining a functionality is shown in the listings which result in the fact that the code listings in the different sections does not correspond exactly with the code for the actual application.

5.1 GUI

As described in section 4.3 on page 47 the application is made as a menu which the user can navigate through. Each entry in the menu triggers a function e.g. to start the sender side or to change the settings of the phone. Because the user also needs to navigate back through the menu it needs to be build into a main loop, which causes the menu to be shown again when the user leaves an entry instead of leaving the program all together. An overview of the GUI in both master and slave mode can be seen in appendix B on page 97

5.1.1 Menu

When a phone is selected to be slave a different menu is shown. This menu only has two entries, the setting entry and an entry called ”slave”. The first entry is necessary so the user can change the phone back to master and the second entry locks the phone in slave-

mode. To control which menu should be displayed, the dictionary entry ”mode” is used, if this contains character ”m” the phone should be master and if it contains the character ”s” the phone should be slave. This is shown in listing 5.1 in line 4 and 21. The dictionary will be explained later in this section.

Figure 5.1 shows the menu when the phone is selected to be master.

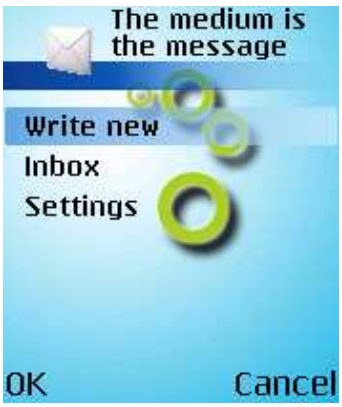


Figure 5.1: The main menu on a master phone

The menu is made as a selection list which is a menu style from the appuifw module. It allows the user to choose an item from a list and returns an integer.

Listing 5.1: The main menu loop

```
1  while running:
2    active = 1
3
4    if dict["mode"] == 'm':
5      appuifw.app.exit_key_handler = exit_key_handler
6      MenuList = [u'Write new', u'Inbox', u'Settings']
7      index = appuifw.selection_list(choices=MenuList,
           search_field=0)
8
9      if index == 0:  #Editor
10          active = 0
11       editor()
12     elif index == 1:  #Inbox
13          active = 0
14       inbox()
15     elif index == 2:  #Settings
16          active = 0
17       settings()
18     elif index == None:
```

```
19        exit_key_handler()
20
21    elif dict["mode"] == 's':
22      slave(MenuListSlave)
```

The selection list is called with the command in line 7 and its first argument is the list in line 6. The list is the items in the menu.

To keep track of which level in the menu the user is on, `active` is 1 in the main menu and 0 when an entry in the menu has been chosen. In listing 5.1 line 5 the `appuifw.app.exit_key_handler` which is triggered by the right softkey is assigned to the function in listing 5.2.

Listing 5.2: The exit key-handler

```
1  def exit_key_handler
2    if active:
3      running = 0
4    menu_lock.signal()
```

Here `active` is checked and if it is set the application is terminated.

5.1.2 Editor

The editor is simply made by changing the body of the appuifw module to ”Text” which makes it possible for the user to input text using the T9-dictionary. To make it possible for the user to send the message that had been written in the editor a menu is made. This menu is visible if the user presses the left softkey. Listing 5.3 shows a code example of the editor:

Listing 5.3: The editor

```
1  def editor():
2    appuifw.app.title = u'Editor'
3    appuifw.app.body = appuifw.Text()
4    appuifw.app.menu = [(u'Send sms',send)]
5
6  def get_contents():
7    string = appuifw.app.body.get()
8    return string
```

Lines 1-4 is the function for the editor. The arguments of the `appuifw.app.body` is a list of tuples, one for each entry in the menu. The first item in each tuple is the entry in the menu and the last item is the function which is called when the user chooses the entry.

Lines 6-8 is the function which returns the contents of the editor. It is important that this function is called before the user leaves the editor because the `appuifw.app.body` changes. Figure 5.2 shows the editor with the message ”Hi”.

Figure 5.2: The editor of the application

5.1.3 Inbox

Like the main menu the inbox is made as a selection list but the inbox has only one entry which will display the most resent message received. When the message is received it is saved in a file on the phone and the same file must be opened when the message should be displayed. This is shown in listing 5.4 below.

Listing 5.4: Retrieval of a message from the inbox

```
1 try:
2   f = open(u'c:\\message.txt', 'r')
3   sms = f.read()
4   f.close()
5 except IOError:
6   sms = ''
7 appuifw.note(u'' + sms, 'info')
```

The opening of the file is made as a try/except statement because the user might try to view a message before any message is received. In lines 2-4 the file is opened and the contents is loaded in the variable `sms`, if this fails because there is no file yet, the `sms` is made with no content. When the `sms` is either loaded from the file or made with no content it is shown in a note. This is done with the command in line 7.

5.1.4 Settings

To prompt for user input the appuifw module provides the function query which pops up and ask for one input. In the settings entry in the main menu several input is needed therefore this entry is made as a `appuifw.Form` type. The Form is an editable multi-field dialog which gives a better overview for the user than the query used several times. This in shown in figure 5.3.

Figure 5.3: The settings menu of the application

The Form is put together with a dictionary which is a list of `key:value` pairs. The key is the name of the field and the value is the user input. The dictionary is loaded from a file using the command: `dict = eval(open(dir).read())` . The eval function interprets a string as a Python expression, in this case a dictionary. The command is used in a try/except statement where the dictionary is created if no file is found.

The following code example in listing 5.5 shows the creation of Form and use of both the Form and the dictionary:

Listing 5.5: Creation and use of the settings menu

```
1  def settings(self):
2    settings_form = appuifw.Form(SettingsList)
3    settings_form.flags = appuifw.FFormDoubleSpaced
4    settings_form.save_hook = self.save_settings
5    settings_form.execute()
6
7  def save_settings(self, form):
8    dict["mode"] = form[0][2]
9    dict["local_partner"] = form[1][2]
10   dict["remote_master"] = form[2][2]
11   dict["remote_slave"] = form[3][2]
```

```
12
13    open(u'c:\\settings.txt', "w").write(repr(dict))
14    return True
```

The appuifw.Form made in line 2 takes one argument which is a list of tuples. Each tuple consist of three items which describes the fields in the form:

1. Title
2. Type
3. Value

The type is text for all the fields and the value is the different dictionary items.

The form has different attributes of which one is the `flags` . This can be set to different values e.g. `FFormViewModeOnly` which means that the form can not be edited. In this application the `FFormDoubleSpaced` value is used which puts the title and the value of the field on two different lines instead of one.

In line 4 the attribute `save_hook` is used. This ensures that the form is saved when the user leaves this menu entry. The correct parts of form must be saved in the dictionary, this is shown in line 8-11. `form[1][2]` means the the third item from second field in the form and in line 9 this is inserted as the value to the key `local_partner` in the dictionary.

When the form is made it must be visible in the GUI, this is done in line 5 with the `execute()` command. In line 13 any change the user might have made is saved in the dictionary file using the `repr()` command which saves the dictionary as a string because a Python dictionary can not be saved in a text file.

5.2 Bluetooth

This section describes the implementation of the communication between the master and the slave in a cluster of phones. Table 4.1 on page 49 states the need of five bluetooth actions (listed with prefix ”BT”). The communication is designed as a client/server connection with the slave as server and the master as client. This way the master can connect to the slave when it is needed.

BT start server and **BT accept connection** is implemented as one function, handling the bluetooth server using the socket module in PyS60. The server makes a blocking call waiting for a client to connect and returns a socket object which is a descriptor of the

connection. This can be used for reading from or writing to the socket. **BT connect** is implemented as one function with the MAC-address of the server as input and a socket object as output. Like on the server the socket object can be used for reading or writing. The code example in listing 5.6 below shows some of the code used to set up the connection.

Listing 5.6: Methods used to establish a bluetooth connection

```
1 sock = socket.socket(socket.AF_BT, socket.SOCK_STREAM)
2
3 port = socket.bt_rfcomm_get_available_server_channel(sock)
4 sock.bind(("", port))
5 socket.bt_advertise_service(u"BT_service", sock, True,
      socket.RFCOMM)
6
7 (sock,peer_addr) = sock.accept()
8
9 sock.connect(server_address)
```

Line 1 is used on both server and client to create a socket object using the bluetooth address family. Lines 3-5 is used on the server to assign a channel to the socket and advertising an RFCOMM service. Line 7 is used on the server to accept a connection from the client modifying the socket object to be returned and line 9 is used on the client to connect to the server.

BT send uses the returned socket object from the server/client to write a string to the socket. It calls the method `send(string_to_write)` on the socket. To let the receiver of `string_to_write` know when the transmission is done, an ending character (newline) is added.

BT receive uses the method `recv(1)` on the input socket object. `1` specifies that only one character can be received and stored in the buffer at the time, thus the BT receive-function is constructed as loop appending incoming characters to a string which is returned when the ending character is received. This is shown in listing 5.7.

Listing 5.7: The BT receive function

```
1 while 1:
2   ch=sock.recv(1)
3   if(ch=='\n'):
4     break
5   line = line + ch
6 return line
```

5.3 Conversion / Bit mapping

Before the transmission can begin the message string must be transformed into a sequence of numbers which corresponds to a sequence of phone calls. This module implements the same functionality as the data link layer known from computer networks. At the sender side it transforms a data packet into raw bits which can be transmitted over the available communication channel. Though the raw bits is not being send, but derived from the call sequence. Furthermore this module should not operate on the data one bit at the time, but split the data into elements of bits depending on how many phones is available in a cluster.

Figure 5.4 shows such a mapping in the case where each cluster consist of two phones which is the current implementation. It shows the mapping on the sender side known from table 4.1 on page 49 as **Convert to list**. The receiver side has an inverse module known as **Convert to string** which is mapping from bit elements to a data string.

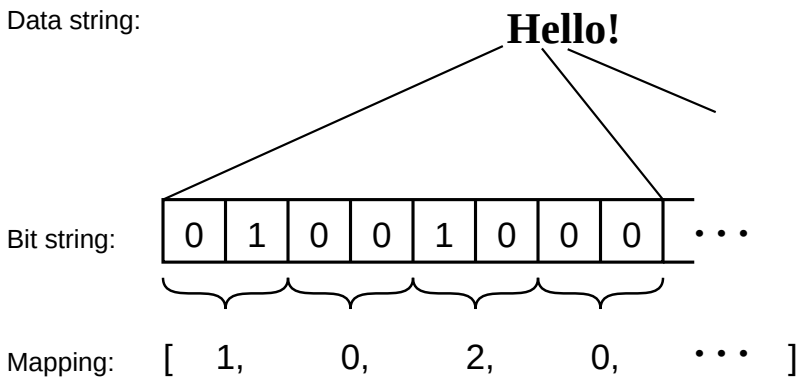


Figure 5.4: Mapping of a string. The bit representation of the character "H" is mapped into elements of two bits. The mapping continues for the rest of the string.

The mapping is done using binary operations on one character at the time. The steps in the process are:

1. A bit element is created from the most significant end of the bit representation.
2. The element is appended to a list.
3. The bit representation is shifted (`char = char « 2`).
4. Continue with step 1.

5.4 Threads

The master on the receiver side must both listen for incoming calls and for incoming data on the bluetooth connection which requires different threads due to the blocking behavior of the bluetooth receive function. These threads are started from the main loop with the commands in listing 5.8:

Listing 5.8: Thread creation

```
1 thread.start_new_thread(bt_thread,())
2 thread.start_new_thread(receive,())
```

Two new threads are started which means that the application is running with three threads with the following purpose:

1. Handles the GUI
2. Listen for incoming calls
3. Listen for incoming data on the bluetooth connection

Both 2 and 3 handles the processing of the incoming data. To control the different threads two locks are made. The code example in listing 5.9 shows the commands used in relations with threads.

Listing 5.9: Thread lock operations

```
1 bt_lock = thread.allocate_lock()
2 master_lock = thread.allocate_lock()
3 master_lock.locked()
4 master_lock.acquire()
5 master_lock.release()
```

The first two lines return a new lock object each. Line 3 returns the status of the lock, 1 if it has been acquired by some thread, 0 otherwise. The commands in line 4 and 5 are used to acquire and release the lock. The phone must be able to both send and receive a message, thus it both checks for incoming calls and show the menu on the screen. When either the first call of a message is received or the user choose to write and send a message the other function must be stopped to avoid errors. This is done by acquiring `master_lock` before the threads do any tasks. If the lock is acquired by the other thread no action should be executed.

`bt_lock` is acquired by the main thread before the new thread is started and not released until the first call is received. Otherwise the phone would not be able to send a message thus the bluetooth socket can only be used in one thread.

5.5 Extension modules for PyS60

In this section the extensions modules made for PyS60 will be described. First `HangUp()` is described. This module handles the hang up of an incoming call. The second module is `CallStatus()` which look up a variable and returns the value. The last module is `LastCall()` . This module make a request of the latest phone number in the call log through an Active Scheduler which is described later in this section.

The entries **Call receiver** and **Wait for call** in table 4.1 on page 49 both need `CallStatus()` . **Wait for call** also handles the hang up of incoming calls and therefore uses `HangUp()` . The entry **Get caller ID** is equivalent to `LastCall()` .

The three extensions is made as modules in Symbian C++ because the functionalities is not yet available in PyS60.

5.5.1 `HangUp()`

This function works by simulating the same key presses that is used when a user wants to disconnect an incoming call on a phone.

To get a better overview of the functionality a code example of the C++ code is shown in listing 5.10.

Listing 5.10: A code segment of `HangUp()`

```
1 RWsSession ws;
2 TKeyEvent key;
3 ws.Connect();
4 key.iCode = EKeyDevice0;
5 key.iScanCode = EStdKeyDevice0;
6 ws.SimulateKeyEvent(key);
7 ws.Close()
```

The first three lines make a connection to the window server session. The session between the client and the window server can be used for client interface control and system control. The `TkeyEvent` is used to set the key code of the key which should be pressed.

The next three lines define the key that should be pressed, and then execute the key press.

The last line closes the windows server session.

5.5.2 `CallStatus()`

This function makes a connection to the client handle through the `systemAgent` , and is then able to access the information from the state variables on the phone, among them the status of the phone.

The C++ code example in listing 5.11 shows the functionality of the module.

Listing 5.11: A code segment of `CallStatus()`

```
1 RSystemAgent systemAgent;
2 systemAgent.Connect()
3 TInt CallValue = systemAgent.GetState(KUidCall);
4 systemAgent.Close();
5 return CallValue;
```

In lines 1-2 initializes and connects to the `systemAgent` . In line 3 the variable `CallValue` is assigned to `KUidCall` that has a value between 0 and 9 depending on what status the phone is in, see table 5.1.

Values	Meaning
0	No call
1	Voice call
4	Calling
5	Incoming call
9	Call Disconnecting

Table 5.1: The meaning of the returned values from `CallStatus()`

The last two lines close the connection to the system agent and return the value from `KUidCall` .

5.5.3 `LastCall()`

This function must return the last number in the call log of the phone. To get access to this the function needs to use two different objects: an active scheduler which the request of the phone number will be made through and an active object which handles the request.

Normally when a UI framework is used it will automatically create, install and start the Active Scheduler, but in this case the function must be written as a DLL (Dynamic Link Library), which needs an Active Scheduler to be started explicitly. This is done by the PyS60 interpreter.

To give a better overview the Active Schedulers life cycle is shown on figure 5.5.

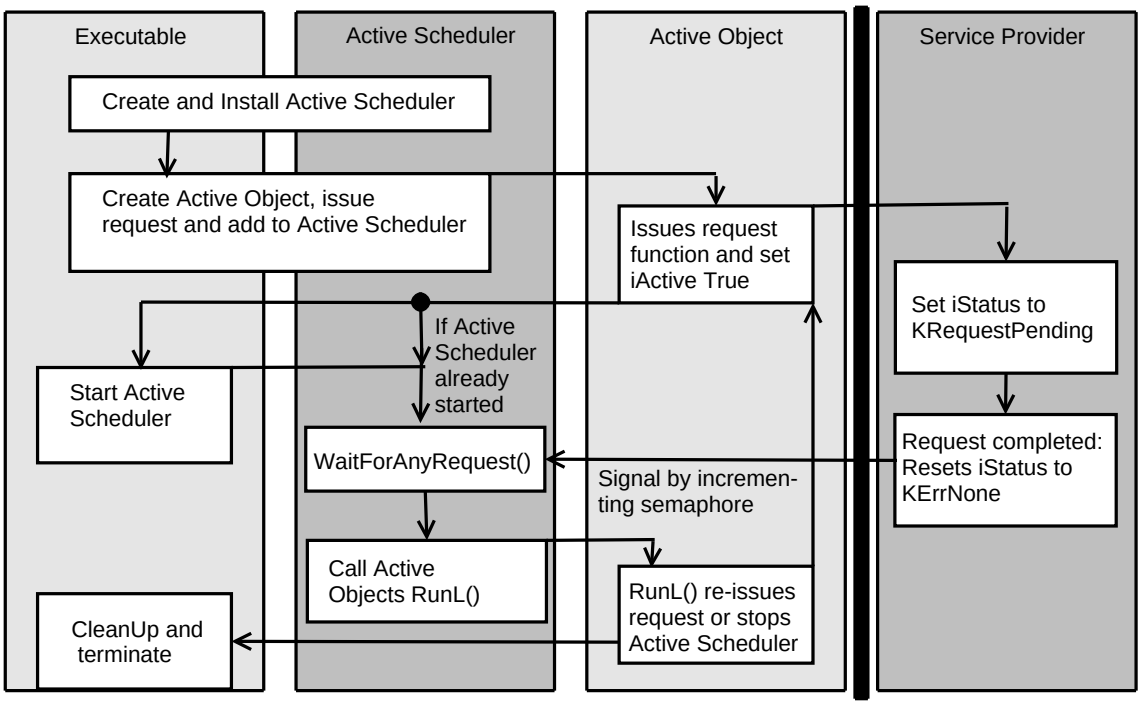


Figure 5.5: The Active Schedulers lifecycle [Edw04]

Before an Active Scheduler can be started, a number of steps must be done. First an Active Scheduler is created and installed, then an Active Object is created and a request is raised to the service provider which contains the different variables e.g. the phone number of the last received call. When the Active Scheduler is started, the thread goes into a ”wait loop” and is therefore suspended and the ”wait loop” is not stopped until the request that was raised to the service provider is done. This causes the Active Scheduler to call `RunL()` in the Active Object. `RunL()` is the handler function which will carry out different actions given to the request. In this case the phone number from the phone log is assigned to a local variable.

It is very important to manually delete the variables because the runtime environment does not automatically clean up when the module returns. Even if it is only a few hundred bytes such a leak will block in the memory and it can become critical issue if it happens enough times. This is due to the fact that the Symbian OS runs on devices that may have only a couple of megabytes of RAM and there may go months between reboots.

The code example in listing 5.12 shows a part of the code for `LastCall()` . The module needs to return the phone number from the call log. It uses an Active Scheduler to get access to the service provider.

Listing 5.12: `GetNumberL()` from `LastCall()`

```
1  TInt GetNumberL(){
2       CGetNumber* getNumber = CGetNumber::NewL();
3       CleanupStack::PushL(getNumber);
4
5       CActiveScheduler::Add(getNumber);
6
7       getNumber->GetNumberFromLogL();
8
9       CActiveScheduler::Start();
10      TInt result = getNumber->iNumber;
11      CleanupStack::PopAndDestroy(getNumber);
12      return result;}
```

In line 2 an Active Object is created to handle the request for the phone number and in line 3 this object is added to the `CleanupStack` which ensures that it is deleted when the module returns. The Active Object which is called `getNumber` is added to the Active Scheduler in line 5 and in line 6 the request for the number is issued. When the request is done the `iNumber` is assigned to `KErrNone` (see figure 5.5 on the previous page) and `RunL()` shown in listing 5.13, is called.

Listing 5.13: `RunL()` from `LastCall()`

```
1  void CGetNumber::RunL()
2  {
3       TBufC<40> numberTemp(iSession->Event().Number());
4       TPtr ptr = numberTemp.Des();
5       ptr.Delete(0,3);
6
7       TLex temp(numberTemp);
8       temp.Val(iNumber);
9
10      CActiveScheduler::Stop();
11 }
```

In line 3-5 the number is fetched and the country code is removed. The reason for this is explained in the end of this section. The number is not fetch as an integer and because this is needed a `TLex` is used to change the data type.

To avoid having `LastCall()` making a call back to PyS60 when the Active Scheduler

is done working with the request the function `GetNumberL` is called with a trap which is shown in listing 5.14. This will catch a leave in the function.

Listing 5.14: Catching a leave with a trap

```
1 TRAPD(result, number = GetNumberL());
2 if(result != KErrNone)
3   return Py\_BuildValue("i", result);
4 else
5   return Py\_BuildValue("i", number);
```

In line 1 `TRAPD` gets two values. The first one is the status of the request and the second is the number fetched from the Service Provider. In line 4 and 6 either the value from the function `GetNumberL` is returned or the error code if the function failed and an error occurs.

It should be noticed that `LastCall()` returns a long int, with a size of 32-bit. This means that the returned value can not be higher than 4.249.967.295, it is therefore not possible to return a Danish phone number with the country code code. The phone number is therefore returned without the country code, like ”+45” [Edw04].

Chapter 6

Test

In this chapter a number of test methods on different levels will be described. It is necessary to test software as it is developed as well as testing it when it is finished. Testing the software while developing will help to ensure that the components which constitutes the software, are not faulty. The purpose is to eliminate the errors that make the software crash or perform in a fashion other than the intended one.

When testing software, the test type must first be chosen. Figure 6.1 on the facing page shows a diagram of different types of testing defined by three parameters:

1. Method of testing
2. Characteristics to test
3. Level of detail to test

A test could be performed as e.g. a white-box test for robustness at module level.

White-box is a test that is based on the source code and performed step by step. It can also be tested as black-box, where the code is tested without knowledge of the steps the input goes through. The white-box testing method will not be used in conjunction with the software for this project and will therefore not be treated further. A description of the black-box testing method will follow.

When testing a system it is sometimes necessary to test parts of the system as a black-box test. This means that these parts of the system has a functionality where the input given undergoes an unknown process. In black-box testing these parts are tested by giving them a specific input and checking that the given output corresponds to the expected output, see figure 6.2 on page 66.

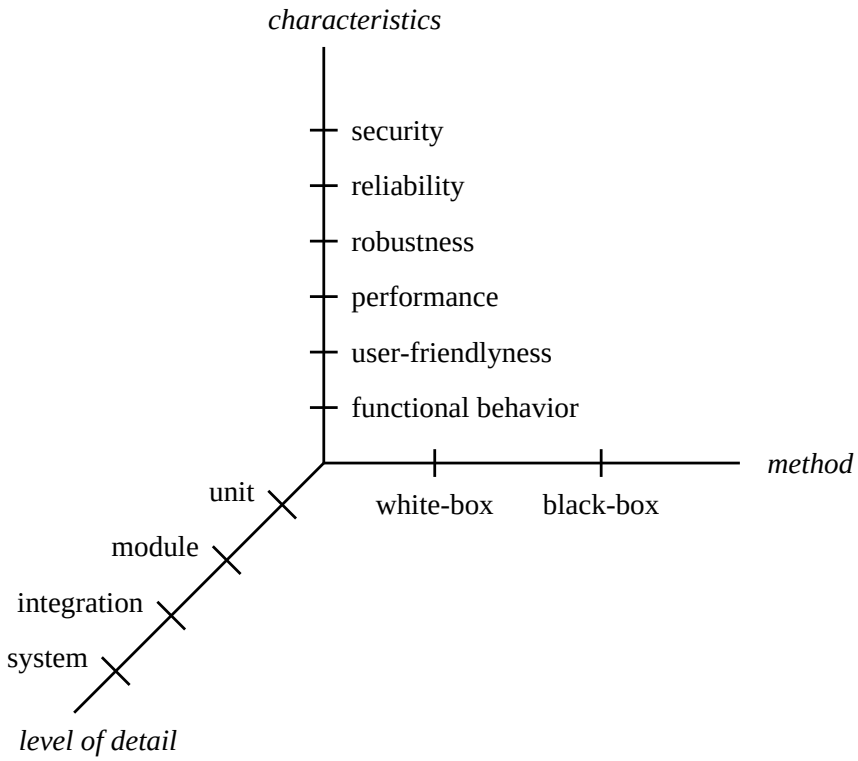


Figure 6.1: Different types of tests [Nie06a]

In order to do this properly the inputs and their expected outputs must first be specified, listing all the inputs that is to be tried and what outputs are expected.

6.1 Tests performed

The tests performed in this project is black-box tests. The tests are performed on the following levels: module and system, as seen on figure 6.1. The acceptance test is a system test that tests the entire system. It does not point to where an error might have occurred, like a black-box test does, but proves that the application works as specified in the requirements specification. The acceptance test can be seen in appendix A.1 on page 76. Furthermore is a module test and a test of error handling made. These tests are described in the following sections. To test the efficiency of the application a test that measures the speed of sending messages is carried out.

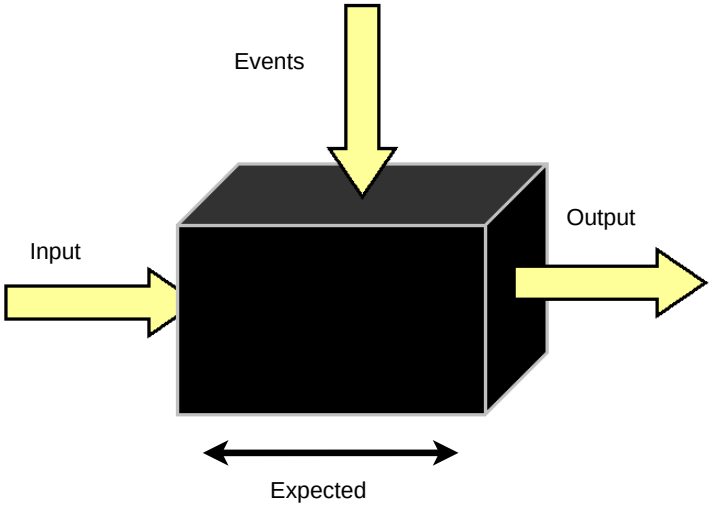


Figure 6.2: The picture illustrates the principle behind black-box testing

6.2 Module test

The tests performed in the test specification in appendix A.2 on page 84 is a module test, which tests every single module described in the implementation chapter 5 on page 50. Table 6.1 shows the results from the module test.

Test:	**Result:**
Testing the GUI functionalities	The GUI on the master works as expected but the right soft key does not work in slave mode
Testing the call functionality	Works as expected
Testing the bluetooth communication	Works as expected
Testing the converting process	Works as expected
Testing `CallStatus()`	Works as expected on Nokia N70 but on Nokia 6600 the module returned 1 instead of 9.
Testing `HangUp()`	Works as expected
Testing `LastCall()`	Works as expected

Table 6.1: The results from the module test

6.3 Acceptance test

The acceptance test has been performed to see if the application performs as intended and whether it lives up to the requirements specified in section 2.5 on page 21. The acceptance test specification is specified in appendix A.1 on page 76. The results of the acceptance test is summarized in table 6.2.

Sequence number:	Result:
1. Run on a Symbian S60 phone	Passed
2. Altering the settings	Failed
3. Sending a message	Passed
4. Editor demands	Passed
5. Handling calls	Passed
6. Compression	Failed
7. Sorting the calls	Passed
8. Full message test	Passed

Table 6.2: The results of the acceptance test. As it states, sequence 2 and 6 has failed.

As it is stated in table 6.2 sequence 2 and 6 has failed. In the following possible breaking points that could have made the tests fail will it be described.

Sequence 2
Sequence 2 failed because the changed settings were not shown when entering the settings without restarting the application. The settings are loaded from a dictionary file when the application starts, not when entering the settings, therefore the changed settings will only be shown after a restart of the application. To solve this problem the loading procedure needs to be implemented in the settings function so it is started every time the user enters the settings menu.

Sequence 6
Sequence 6 failed because the compression feature has not been implemented in the application. If this feature should be included in the application a DLL from the smsZipper could be used.

6.4 Error handling

A test of wrong inputs made to test where the application fails. This test can be seen in appendix A.3 on page 93. The results of this test can be seen in the table 6.3 on the following page.

Test:	Result:
Sending special characters	
Entering wrong MAC/numbers	Wrong MAC, does not connect to the slave Wrong number, calls the wrong number.
Wrong master slave setting	Displays a white screen and the phone needs to be restarted.
Not starting the slave	Does not connect via bluetooth, and crashes after receiving the initial call.
Hidden caller ID	The receiver, initializes to receive but does not receive any data.
Interrupting the transmission	Writes ”wrong number”, hangs up. The right incoming call is not detected and the message is corrupted.
Distance between the phones	Fails to connect via bluetooth and application shuts down.

Table 6.3: This table shows how the application handles errors.

6.5 Measurements

Sequence 8 in the acceptance test specification in appendix A.1 on page 76 tests the performance of the entire system. As seen in appendix A.4 on page 96 the test result is that the application could send a message of 20 characters in 65% of the attempts. In most cases the failure is caused by the network i.e. some calls fail to reach their destination. The average time of sending a succesful message was 31,46 minutes which is 1 minute and 34 seconds per character. This result is for 2 phones on each side with German to Danish sim cards. The table 2.2 on page 14 and the formula in that section can be used to calculate the transmission speed with more phones on each side.

If one character is send in 1 minute and 34 seconds (94 seconds) with two phones on each side it takes 23,5 sec. per call ((*t*) and the following formula can be used to calculate the time (*T*) it takes to send one character if more phones are available:

$$T = t \times \frac{8}{bits/call}$$

The result of this calculation is shown in table 6.4.

Phones/side	Time to transmit a character
2	1 minute and 34 seconds
4	47 seconds
8	31,33 seconds
16	23,5 seconds

Table 6.4: The calculated time to send a character

Chapter 7

Conclusion

The main purpose of this project was to develop an application to send a message between two groups of phones by making unanswered calls, which makes it possible to send free SMS messages. This idea takes advantage of the Italian communication system Squillo and the principles of channel shifting to send a media coded message. The application was developed for the Nokia S60 mobile phones and is tested on Nokia 6600 and N70.

Several methods for sending a message this way was presented and the method ”Cluster calls” was selected because of its robustness and transmission speed. There were three different languages available for the implementation of the application. PyS60 was chosen due to its possibilities for design of a graphic user interface and its short and easy learning process which makes it ideal for prototyping. PyS60 offers only limited functionality on the S60 platform but it has been extended with new modules through the PyS60/C API.

The result is an application which illustrates the principles of sending a message using this method. It can send messages between two clusters of phones using two phones for sending and two for receiving. The phones on the sender side calls the phones on the receiver side in a pattern corresponding to the message for transmission.

The application has been tested in Denmark using German sim cards in the sender phones and Danish sim cards in the receiver phones because of the free dial-ups in Germany. The acceptance test shows that six test sequences out of eight has passed. On of the failed sequences is the compression which is not yet implemented and as a result the transmission time is longer. It was possible to send 20 characters successfully in 65% of the cases which is concluded to be acceptable in this prototype application. The avarage time of the transmission of 20 characters is 31,46 minutes which is 1 minute and 34 seconds per character.

There are a number of bugs and uncompleted modules like the compression which will

be considered in chapter 8 on the next page, but it is concluded that the application is working and is worth further development to make it available for public use.

Chapter 8

Future perspectives

In this chapter the ideas that could be added to the application for its further development and to make it a viable product for distribution is presented.

Developing for Symbian

The current version of the application has been developed in PyS60. If the application were developed in a Symbian version, the application would be available for a significantly larger number of phones. The implementation in Symbian C++ would be the next logical step since the PyS60 application is being interpreted by the Symbian core on the used phones. Developing the application for Symbian presents a more viable solution since a number of smart phones do not support PyS60 as a standard.

Implementing the compression

The compression of the message should be implemented because it would give the application less data to transmit. The compressor from the smsZipper can achieve a compression of 1/3 of the original message which would increase the transmission speed of the message. The compressor could be implemented directly into the application by using it before the conversion and forwarding the compressed message to the converter.

Improving the user interface

In the current version of the application the slaves are manually added in the settings menu by entering the MAC-addresses. However if the application could actively search for phones within bluetooth range, that are available for slave use, and automatically integrate them into the transmission. As the slaves are detected the system will also have to keep track of the slaves already integrated and eliminate them if they are no longer available. This could possibly even be done while a message is being sent, as suggested in the scenario 2.2.2 on page 15. This would increase the speed and the flexibility of the application.

In the current menu the user has to type the mode in ”Settings”. This opens the possibility of typing a wrong input. Ideally the ”mode”-selection would be limited to two choices, where the user uses the navi-key to select. The selection of remote phones could also be improved by implementing an interface to the phone book.

The current inbox shows only the latest received message. The inbox could be extended to enable the storing of multiple messages which could be done by creating a selection list with entries for each text file in a message folder. Incoming messages could be appended to this folder.

Error handling

While the current version of the application has successfully sent messages between the two clusters of phones, there is however no error handling implemented. As a result of this there is no possibility for catching errors that may occur e.g. if one of the phones becomes unavailable for a period of time, the other phones in the system is currently not capable of adapting to the changes in their environment. Error detection and handling should be implemented in order to prevent a system failure while transmitting.

New character set

When converting the message into bits, the application currently uses the ASCII character set where each character is represented as 8 bits. The creation of a new character set, specifically designed for the transmission would increase the efficiency. In the new set the bit representation would be limited to 6 bits, thereby eliminating the need for one of the four calls in a 2 by 2 scheme. If the set is constructed using 6 bits, there will be 2^6 = 64 possible character in the set. This is enough to make all the letters in the English alphabet as well as special characters such as period and decimal points. By eliminating one phone call of every character in the message, the message is shortened by 25%.

Increasing the transmission possibilities

To further increase the speed of the transmission each phone in the system could establish more than one type of communication e.g. data or fax calls. If this is possible the phones could switch between the different communication channels like the frequency hopping in bluetooth. This would provide more parameters in the communication making it possible to derive more bits for each call.

Bibliography

[Dor02] Mads Doré. Real-time systems - lecture notes - lecture 2. PDF, 2002.

[Edw04] Leigh Edwards. *Developing Series 60 Applications*. Addison Wesley, 2004.

[Fle02] Luke Fletcher. *With this ring*. The Age, http://www.theage.com.au/articles/2002/07/26/1027497404219.html , 2002.

[Fou05] Python Software Foundation. *Python introduction*. http://www.python.org/doc/intros/introduction/ , 2005. Accessed 25.04.06.

[Hik99] Ann M. Hikey. Setting a foundation for collaborative scenario elicitation. PDF, 1999.

[MH05] Norman Matloff and Francis Hsu. *Tutorial on Threads Programming with Python*. http://heather.cs.ucdavis.edu/~matloff/Python/PyThreads.pdf , 2005. Accessed 12.05.06.

[Nie06a] Brian Nielsen. *Modellering, Test og Verifikation - lecture 2 PDF*. http://www.cs.aau.dk/~bnielsen/MTV/ , 2006.

[Nie06b] Jens Dalsgaard Nielsen. Analyse og design af indlejrede realtidssystemer - lecture 1. PDF, 2006.

[Nok05] Nokia. *Python for Series 60 Platform API Reference*. Forum Nokia, 2005.

[Nok06a] Nokia. *Nokia - Device details - Nokia N70*. http://www.forum.nokia.com/main/0,,018-2578,00.html?model=N70 , 2006. Accessed 24.05.06.

[Nok06b] Nokia. *Nokia - Phone Features - Nokia 6600*. http://europe.nokia.com/nokia/0,8764,33211,00.html , 2006. Accessed 27.04.06.

[Nor03] Jacob Nordfalk. *Javabog.dk*. http://javabog.dk/VP/kapitel13.jsp , 2003. Accessed 14.04.06.

[oT06] IT og Telestyrelsen. *Teleprisguiden*. http://www.teleprisguide.dk/ , 2006.

[Sym06a] Symbian. *Fast facts*. Symbian, http://www.symbian.com/about/fastfacts/fastfacts.html , 2006. Accessed 24.05.06.

[Sym06b] Symbian. *Ownership*. Symbian, http://www.symbian.com/about/overview/ownership/ownership.html , 2006. Accessed 24.05.06.

[Wik06] Wikipedia. *Mobile Development Comparison*. http://en.wikipedia.org/wiki/Mobile_development , 2006. Accessed 11.04.06.

Appendix A

Tests

A.1 Acceptance test

In the following appendix is the acceptance test specification arranged to test whether the requirements specification in section 2.5 on page 21 is fulfilled for the final system. Different test sequences are set up to test the individual requirements. Under each test sequence is it specified how the test will be executed in practice.

Sequence 1

Requirements:

- FR1 - The system must run on mobile phones with Symbian (S60-series) OS

Starting conditions:
No additional

Sequence:
Turn on the phone and start the application.

Success criteria:
If the application starts and a menu is shown, requirement FR1 is fulfilled.

Result:
The success criteria was fulfilled, so FR1 is met.

Sequence 2

Requirements:

- GUI7 - The user must be able to set up the phone to be master or slave in a settings menu
- GUI8 - From the settings menu, the user must be able to change the bluetooth MAC address of the local partner
- GUI9 - From the settings menu, the user must be able to change the phone numbers of the remote master and slave

Starting conditions:
The application must be started.

Sequence:

1. Select ”settings” in the menu
2. Change the local partner (mac) address in format xx:xx:xx:xx:xx:xx
3. Change remote master in the in the format +[land code][phone number]
4. Change remote slave in the format +[land code][phone number]
5. Save the settings by selecting options and save
6. Exit the settings menu and reenter the settings menu

Success criteria:
The new values in the settings menu should be applied. If the new mode value, mac address and phone numbers are shown the requirements GUI7, GUI8 and GUI9 are fulfilled.

Result:
The success criteria was partly fulfilled. The settings are saved successfully, but it was necessary to exit the program to see the changes in the settings menu. The requirements GUI7, GUI8 and GUI9 are partly met.

Sequence 3

Requirements:

- FR2 - The system must connect the phones and exchange data via bluetooth
- FR4 - The system must be able to make a phone call
- GUI1 - The GUI must contain an editor where the user can write a message
- GUI4 - It must be possible to send the message from the editor

Starting conditions:
Bluetooth must be enabled on the master and slave phones on the sender side. The receive master phone must be turned on.

Sequence:
Part 1

1. Start FExplorer on the master phone and choose a file and send it via bluetooth to the slave
2. From the standby screen enter the phone number of the slave phone and press dial

Part 2

1. Start the application on the sender phones
2. Set up the sender slave to be slave
3. Select ”write message” and choose options in the editor and select ”send”
4. Observe that the sender master and slaves are connecting to each other via bluetooth
5. Observe that the sender master is calling and that the receiver master is getting the call

Success criteria:
Part 1

1. If the slave receives the file the requirement FR2 is fulfilled
2. If the slave phone is getting called the requirement FR4 is fulfilled

Part 2

1. When ”write message” is selected in the menu an editor appears on the screen, if this is correct the requirement GUI1 is fulfilled
2. If the option in the editor is ”send” the requirement GUI4 is fulfilled
3. When send is chosen the requirement FR2 is fulfilled if the slave phone prints ”I’m listening”
4. If the master calls the other master the requirement FR4 is fulfilled

Result:
Part 1: The success criteria was fulfilled, so FR2 and FR4 is met.
Part 2: The success criteria was fulfilled, so GUI1, GUI4, FR2 and FR4 is met.

Sequence 4

Requirements:

- GUI2 - It must be possible to write at least 160 characters in the editor
- GUI3 - The editor must support T9-dictionary

Starting conditions:
The application must be started and the menu point ”write message” must be selected

Sequence:

1. Write 160 characters in the editor
2. Choose the T9-dictionary by holding the #-key down until two lines are showed at the pencil in the upper right corner of the screen
3. Make the key presses ”43556”

Success criteria:

1. If the characters in the editor can be counted to 160 the requirement GUI2 is fulfilled
2. When ”43556” is inputted the written text in the editor must be ”hello”. If this is correct the requirement GUI3 is fulfilled

Result:
The success criteria was fulfilled, so GUI2 and GUI3 is met.

Sequence 5

Requirements:

- FR8 - The system must be able to register an incoming call
- FR9 - The system must be able to hang up immediately (in one second)
- FR10 - The system must be able to identify an incoming call

Starting conditions:
The sender master must be in standby mode and the receiver master and slave must have the application started.

Sequence:

1. Call the receiver master manually
2. Check the receiver master's screen under the call process
3. Check the screen on the receiver slave after the call

Success criteria:

1. If the receiver master rejects the call in one second, it means that it is registered and the requirements FR8 and FR9 is fulfilled
2. After the call sequence the receiver slave screen must have printed "I'm listening. This means that the call was identified and the bluetooth link is established. Then the requirement FR10 is fulfilled

Result:
The success criteria was fulfilled, so FR8, FR9 and FR10 is met.

Sequence 6

Requirements:

- FR3 - The system must have the ability to compress/decompress the message

Starting conditions:
The four phones must be set up with correct modes, macs and phone numbers and the slaves must be in slave mode.

Sequence:

1. On the sender master the editor must be started
2. Write the message ”This is a test of how the application works”
3. Under options choose ”Check message”
4. Send the message by selecting ”send”
5. After the message is sent, read the message on the receiver phone

Success criteria:

1. On the sender phone the ”check message” must show that the compressed message is smaller than the original. If this is correct the requirement FR3 is partly fulfilled
2. On the receiver phone check that the message is identical with the transmitted message

Result:
The success criteria was not fulfilled because it is not implemented in this version of the application.

Sequence 7

Requirements:

- FR5 - The system must register that the receiver have received the call
- FR6 - The system must convert the message into a bit sequence
- FR7 - The The system must know which phone is the next to call from the bit sequence

Starting conditions:
The application must be started on the receiver phones.

Sequence:
Part 1

Value	Call sequence
0	Master -> master
1	Master -> slave
2	Slave -> master
3	Slave -> slave

Table A.1: Elements and the corresponding call

1. Call the receive master manually from the sender master

Part 2

1. Start the application on all four phones
2. On the receive master start the editor and send the message ”a9”

Success criteria:
Part 1

1. When the receive master is called it will hangup in one second. This means that the requirement FR5 is fulfilled

Part 2

1. When sending ”a9” the sequence of the phone call should be ”1,2,0,1,0,3,2,1”. See table A.1.
2. Look at the receive master while the message is being sent to check that the sequence is correct. If so the requirements FR6 and FR7 is fulfilled

Result:
Part 1: The success criteria was fulfilled, so FR5 is met.
Part 2: The success criteria was fulfilled, so FR6 and FR7 is met.

Sequence 8

Requirements:

1. FR11 - The system must be able to send a message of 20 characters in 3 out of 5 attempts.

2. GUI5 - The user must be able to view an incoming message

3. GUI6 - The user must be able to view a list of received messages

Starting conditions:
The four phones must be set up with correct modes, macs and phone numbers and the slaves must be in slave mode.

Sequence:

1. Select ”write message” on the sender master and write the message ”Hello this is a test”. Then select ”send” under options.

2. Observe the screen on the receiver master. If it after some time prints ”Following message is received and saved:” plus the message that was send.

3. Press exit on the receiver master and select inbox. Under inbox select one of the messages.

4. Repeat the three previous sequence test points 4 times.

Success criteria:

1. If the receiver master gets the message of 20 characters in 3 out of the 5 attempts the requirement FR11 is fulfilled.

2. If the receiver after the sending process prints ”Following message is received and saved:” plus the message the requirement GUI5 is fulfilled.

3. When the messages in the inbox is selected they must be showed to the user. If this is correct the requirement GUI6 is fulfilled.

Result:
The success criteria is fulfilled so the requirements FR11, GUI5 and GUI6 is met.

A.2 Module test

Testing the GUI functionalities

Date: 17-05-06
Comments: The purpose of this test is to ensure that the GUI functionalities is working as expected. The test is performed on each of the four phones to ensure that the menu can be accessed from the different phones. This is testing that the navigation through the menu is working as intended.

Test 1:
Testing the functionality of ”Write new”, ”Inbox” , ”Slave” and the ”cancel”-buttons

Test case:
Testing master mode menu

1. Start the application. Select ”Write new”. When inside the editor press the right soft key to return to the menu.
2. Select ”Inbox” in the menu. When inside the ”Inbox”. Open a message.
3. In the menu press the right soft key.

Testing slave mode menu

1. Start the application. Select ”Slave”. Then press the right soft key.
2. In the menu press the right soft key.

Expected results:
Master mode

1. After the application is started a menu screen should be displayed. When ”Write new” is selected an editor should appear on the full screen. The editor should exit when pressing the right soft key.
2. The ”Inbox” opens in the full screen and displays a list of messages. When selecting a message it opens in an info box.
3. The application will exit when the right soft key is pressed.

Slave mode

1. After the application is started a menu screen should be displayed. By selecting ”slave” a console screen should appear and display information. When pressing the right soft key the application should return to the menu screen.

2. The application will exit when the right soft key is pressed.

Actual results:
The buttons work as expected except in slave mode where pressing the right soft key does not have any effect.

Test 2:
Testing the functionality of the ”Settings”

Test case:
Testing the function on both slave and master

1. Select ”Settings” in the menu. In the settings edit the ”mode (m)aster/(s)lave” by entering a small ”s” or ”m”. Save the changes by pressing options and ”save”. Edit the ”local partner” and select the right soft key to exit and press ”No” to ”Save changes?”

2. Repeat the test for each of the items in ”Settings”.

Expected results:

1. When ”Settings” is selected a list of the different items should appear. By editing the ”mode (m)aster/(s)lave” with ”s” or ”m”, saving and leaving the ”Settings” the menu should display the mode-specific items. While choosing not to save e.g. the ”local partner”-value, this should not be saved. Changing the other items in the settings menu will not result in graphical changes in the GUI in any way.

2. The rest of the items in the settings menu can be edited by selecting each one with the left soft key and saving the new value either by pressing ”save” or pressing the right soft key and choosing ”Yes” to the query that appears.

Actual results:
The settings can be changed but the values is not checked by the program and mistyping the settings e.g. using a capitalized ”M” or ”S” instead of ”m” and ”s” will not be detected. Using the capitalized letters will crash the program.

Testing the call functionality

Date: 17-05-06
Comments:
The purpose of this test is to ensure that the calling functionality is working correctly. The first test is testing the sender phones and the second is testing the receivers. The phones that is not tested in each test is controlled manually. In these tests the call sequence will be the initializing call plus ”0,3,2,1” as shown in table A.1 on page 82. This is done by sending the message ”9” from the editor.

Test 1:
Testing the call functionality on the sender phones

Test case:

1. Start the application on the sender phones. The receiver phones must be in standby mode so they con be controlled manually.
2. On the sender slave phone select the slave mode
3. Send a message ”9” from the editor on the master phone.
4. Observe the outgoing calls on the sender phones and hangup on the receivers when they receive an incoming call.

Expected results:

- While sending the ”9” message it is expected that the call sequence is the initializing call plus ”0,3,2,1” as described in comments.

Actual results:
The call was executed as expected.

Test 2:
Testing the call functionality on the receiver master and slave phones

Test case:
The application must be started on the receiver phones and the sender phones must be controlled manually.

1. Start the application on the receiver phones. The sender phones must be in standby mode so they con be controlled manually.

2. On the receiver slave phone select the slave mode
3. Make the call sequence: Initializing call plus ”0,3,2,1” manually from the sender phones and observe the receiver phones.

Expected results:

- While the call sequence: Initializing call plus ”0,3,2,1” is executed it is expected that the receivers get called in this pattern and hangs up when they receive the calls. It is also expected that the receiver will receive the character ”9”.

Actual results:
The receivers rang in the correct order and received the character ”9”.

Testing the bluetooth communication

Date: 18-05-06
Comments:
The purpose of this test is to ensure that the bluetooth communication is working as expected. The test is going to examine at the communication between the devices, when using the the bluetooth module. The test is going to be done on two phones in a Python console with the bluetooth module loaded.

Test
The test is going to send and receive different text strings between the two phones, to make sure that the module works properly.

Test Case
Testing the bluetooth communication between two phones

1. Start Python on both phones, and enter the interactive console.
2. Enter `import bt` on both phones.
3. On the receiver phone enter `sock = bt.server()`
4. On the sender phone enter `sock = bt.client( "address")`, where *"address"* is the bluetooth MAC-address of the receiver phone.
5. On the receiver phone enter `text = bt.listen(sock)` .
6. On the sender phone enter `bt.write( sock,"Text to send")`

7. Now enter `print text` on the receiver phone.

Expected results
Every time the second phone sends a text, the receiver should print the text on the screen when the first phone is set in the `bt.listen` mode followed by `print`

Actual results:
After the connection was made and the text was sent, the text could be printed as expected.

Testing the converting process

Date: 19-05-06
Comments:
The purpose of this test is to ensure that the conversion of the message into bit elements runs correctly. If the conversion makes an error in converting the message, the error will be carried through the call process and give an error in the received message. The conversion module will be tested through the console on one of the phones, where the conversion function will be called.

Test 1
Testing the conversion from the original message into bits as performed on the test phone. The test phone will be running a Python console with a bluetooth connection to a PC.

Test Case:
Conversion of characters into bits

1. Start hyperterminal on the PC
2. Start Python on the phone and connect to the PC via bluetooth
3. The char2list function will be called with random characters from the 7 bit GSM ascii table.
4. The returned bits will be compared to the ascii value for the given character.

Expected results (conversion)
The conversion is expected to return a value of bits, in the order of 4 numbers, 0-3, which corresponds to the ascii value of the character that was used as input.

Actual results:
The function returned successfully all the characters sent to the module. It should be

noticed that if a user wishes to use any special characters such as the ”euro’ sign it is not possible due to the character is made of two characters from the GSM ascii set.

Test 2
Testing the conversion of bits back into character for the received message. The test will be performed on the test phone. The test phone will be running a Python console with a bluetooth connection to a PC.

Test Case:
Conversion of bits back into characters

1. Start a hyper terminal on the PC
2. Start Python on the phone and connect to the PC via bluetooth
3. The test phone will be running a console.
4. The list2char function will be called with 4 numbers (0-3) in random order as input within the range 0-127 in the ascii table.
5. The returned character will be compare to the ascii table to check that the character is the correct one according to the given input number.

Expected results (conversion)
The conversion should return corresponding characters for each number given.

Actual results:
The function returned the correct characters corresponding to the numbers.

Testing `CallStatus()`

Date: 17-05-06
Comments:
The purpose of the test is to ensure that the output that is returned from `CallStatus()` is the same as expected. The test is performed through the bluetooth console and is run on the module as a standalone on the phone.

Test 1:
Testing the output of the module in different call states

Test case:
The test phone must be able to connect via bluetooth to a PC. Another phone is needed to make calls to the test phone and to receive call. The other phone must be set up not to call divert when it is rejecting an incoming call.

1. Start a hyper terminal on the PC
2. Start Python on the phone and choose the bluetooh console to connect to the PC.
3. Run the command `import phone_ext` and `import telephone`
4. No call:

   To test when there is no call run the command `phone_ext.CallStatus()`
5. Incoming call:

   Call the test phone by the other phone. When the test phone rings run the command `phone_ext.CallStatus()`
6. Outgoing call:

   Run the command `telephone.dial("number of callee")` . Then run the command `phone_ext.CallStatus()`
7. Call disconnecting:

   Like before run the command `telephone.dial("number of callee")` . Reject the call when the other phone receives the call and run the command `phone_ext.CallStatus()` repeatedly.

Expected results:

1. When there is no call it is expected that the module returns 0
2. With an incoming call it is expected that the module returns 5
3. When there is an outgoing call it is expected that the module returns 4
4. When the call is disconnected remotely it is expected that the module returns 9

Actual results:
The results was as expected for numbers 1-3. For number 4 Nokia N70 returns 9 as expected while Nokia 6600 returns 1.

Testing `HangUp()`

Date: 17-05-06
Comments:
The purpose of this test is to test the functionality of `HangUp()` . The test is performed through the bluetooth console and is run on the module as a standalone on the phone.

Test 1:
Testing the execution output of `HangUp()`

Test case:
This test case tests that the module can hang up on the test phone. Another phone is needed to make an incoming call to the test phone.

1. Start a hyper terminal on the PC
2. Start Python on the phone and choose the bluetooh console to connect to the PC
3. Run the command `import phone_ext`
4. Call the test phone manually from the other phone
5. When the test phone rings run the command `phone_ext.HangUp()`

Expected results:

1. When the test phone rings and the command is executed it is expected that the phone hangs up

Actual results:
The phone was hanging up when command was executed.

Test 2:
Testing the execution output of `HangUp()`

Test case:
The module simulates key presses to hangup so it is tested that it can do the key presses correct.

1. Start a hyper terminal on the PC
2. Start Python on the phone and choose the bluetooh console to connect to the PC

3. Run the command `import phone_ext`
4. Switch to the menu screen on the phone
5. Run the command `phone_ext.HangUp()`

Expected results:

1. When the command is run on the standby screen it is expected that the module is executing the key presses ”left soft key, down, down, left soft key”

Actual results:
As expected the executed key presses was ”left soft key, down, down, left soft key” when the command was executed.

Testing `LastCall()`

Date: 19-05-06
Comments:
The purpose of this test is to ensure that `LastCall()` returns the expected values when it is called. The test is going to run on a phone where Python is loaded and is setup as a bluetooth console. This is to ease the testing, because the Python interactive consol is the controlled from a terminal window on a laptop that is connected to the phone by bluetooth.

Test 1:
The test is going to test 4 different cases, to ensure that the module works under all conditions. The first case is running the module when the log is empty. The second is where there only is one record stored in the log, so the test is run once with a record in Missed calls, Received calls and Dialled numbers. The third case is going to be run where an anonymous call has been received. The fourth case testing the module is run when all tree logs have some records.

Test case

1. Start a terminal window on the laptop, and let it listen on the incoming bluetooth communication.
2. Start the bluetooth console on the phone and establish a connection it to the laptop

3. Setup the phones call log as described and execute the tests as described in the test case

Expected results

1. The first case should return the error code ”-1000”, which indicates that the log is empty
2. The second case should in all under cases return the same number
3. The third case should return 0, this means that the received call was anonymous
4. The fourth case should return the last number added to the log independently of which log it is added to.

Actual results
The returned values was as expected.

A.3 Error handling of the application

In this section the functionality of the application is tested for its stability to see what happens when the user gives wrong input to the application. The program can run a controlled test environment where the number of errors are limited. If an ordinary user should use the program it is more important to be aware of possible errors caused by wrong inputs.

Sending special characters

In the application it is possible to write a message containing all possible characters in the ascii table. However some of these characters are not supported in the normal ascii table but are included in the extended table. The phone uses the GSM ascii table that uses the normal table and some of the extended.

Test
To test sending of special characters the following sequence will be sent:
”, ′ ? ! " ( ) @ / : \ _ ; + & % * = < > £ $ [ ] { } # ”

Result
The conversion returns a error message, stating that returned string is out of range. This is due to the special characters being from the extended ascII set, which the converter is not designed for.

Entering wrong MAC/numbers in settings

When setting up the application, the user must setup the bluetooth MAC address of its slave and the phone numbers of the receivers. When typing these settings wrong the application might fail.

Test
To test this in the application input wrong MAC and phone numbers in settings and send a random message.

Result
The master sender could not establish a bluetooth connection to the slave and never moved on the transmission. If wrong numbers are entered, the application will try to call these numbers and the message will not be send.

Wrong master slave settings

The mode setting for takes the arguments m/s. If another character or capital letters are written the application might fail.

Test
Change the mode under settings to a character different than m/s.

Result
Changing the mode to a random character other than m/s caused the application to display a white screen with no way to return but to restart the phone.

Not starting the slave mode

The slave has to be ready when sending/receiving a message, this means it must be in slave mode. Otherwise the send/receive part can not start.

Test
Send a message when one of the slaves is not started.

Result
If the sender slave is not available, the master never gets past establishing of the bluetooth communication. If the receiver phone is not available, the receiver master crashes after receiving the initial call from the sender master because it fails to connect to the receiver slave.

Hidden caller ID

The phones must be able to send their caller ID. If this is not the case the sender phones in the transmission can not be identified.

Test
Set up one sender phone to not send the caller ID and send a random message.

Result
If the sender phones do not send their identification along with the call, the receiver master enters the receive state but never initiates the connection to its slave and never gets ready to receive.

Interrupting the transmission

The transmission can be interrupted by an incoming call to the receiver from another phone than the sender phones.

Test
When the application is sending a message, call one of the receiver phones.

Result
The receiver phone labels the incoming call as a wrong number and hangs up. However the incoming call from the sender is not detected and results in an error in the message.

Distance between the phones

The bluetooth standard supports a distance of 10 meters between the devices. If this distance is exceeded the connection will fail and has to be reestablished.

Test
When the application is sending a message move one of the phones more than the bluetooth range from its partner.

Result
The application fails to connect via bluetooth and shuts down.

Test no	Starting time	Finish time	Duration (min)	Passed/Failed
1	8.50	-	?	Failed
2	9.03	9.37	34	Passed
3	9.39	10.09	30	Passed
4	10.21	-	?	Failed
5	10.42	11.05	23	Passed
6	11.18	11.49	31	Passed
7	12.10	12.42	32	Passed
8	12.44	13.16	28	Passed
9	13.18	13.50	32	Passed
10	13.55	-	?	Failed
11	21.04	-	?	Failed
12	21.30	22.02	32	Passed
13	22.15	-	?	Failed
14	22.21	22.54	33	Passed
15	23.07	23.41	34	Passed
16	-	-	32	Passed
17	-	-	33	Passed
18	-	-	?	Failed
19	-	-	35	Passed
20	-	-	?	Failed
Result			**31,46**	**65,00%**

Table A.2: The results of sending 20 characters

A.4 Measurements

Sequence 8 in the acceptance test A.1 on page 76 tests that the application succeed in sending 20 characters in 3 out of 5 attempts. To get a reliable result this is tested with 20 attempts. The results from these tests are shown in table A.2.

Appendix B

Screenmap of the GUI

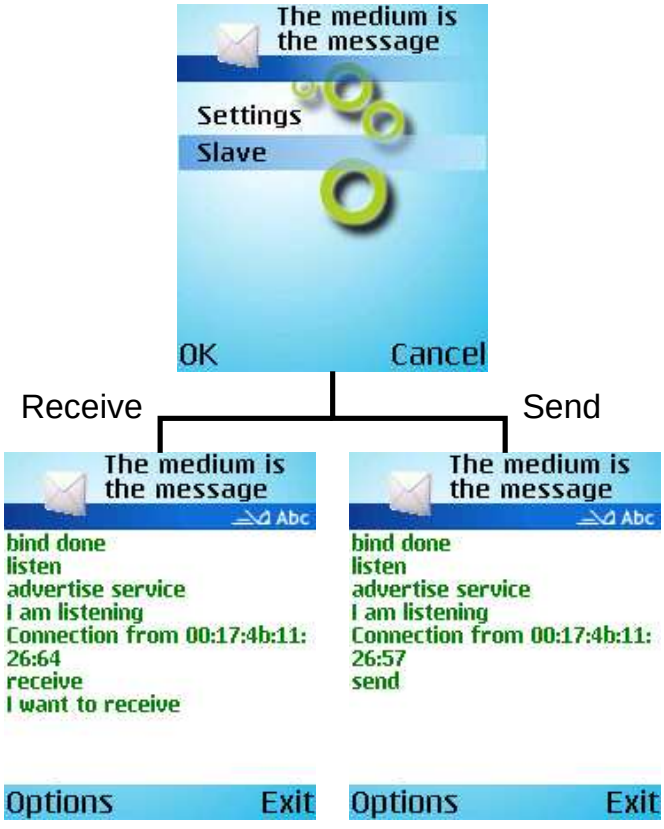


Figure B.1: The GUI of a phone in slave mode

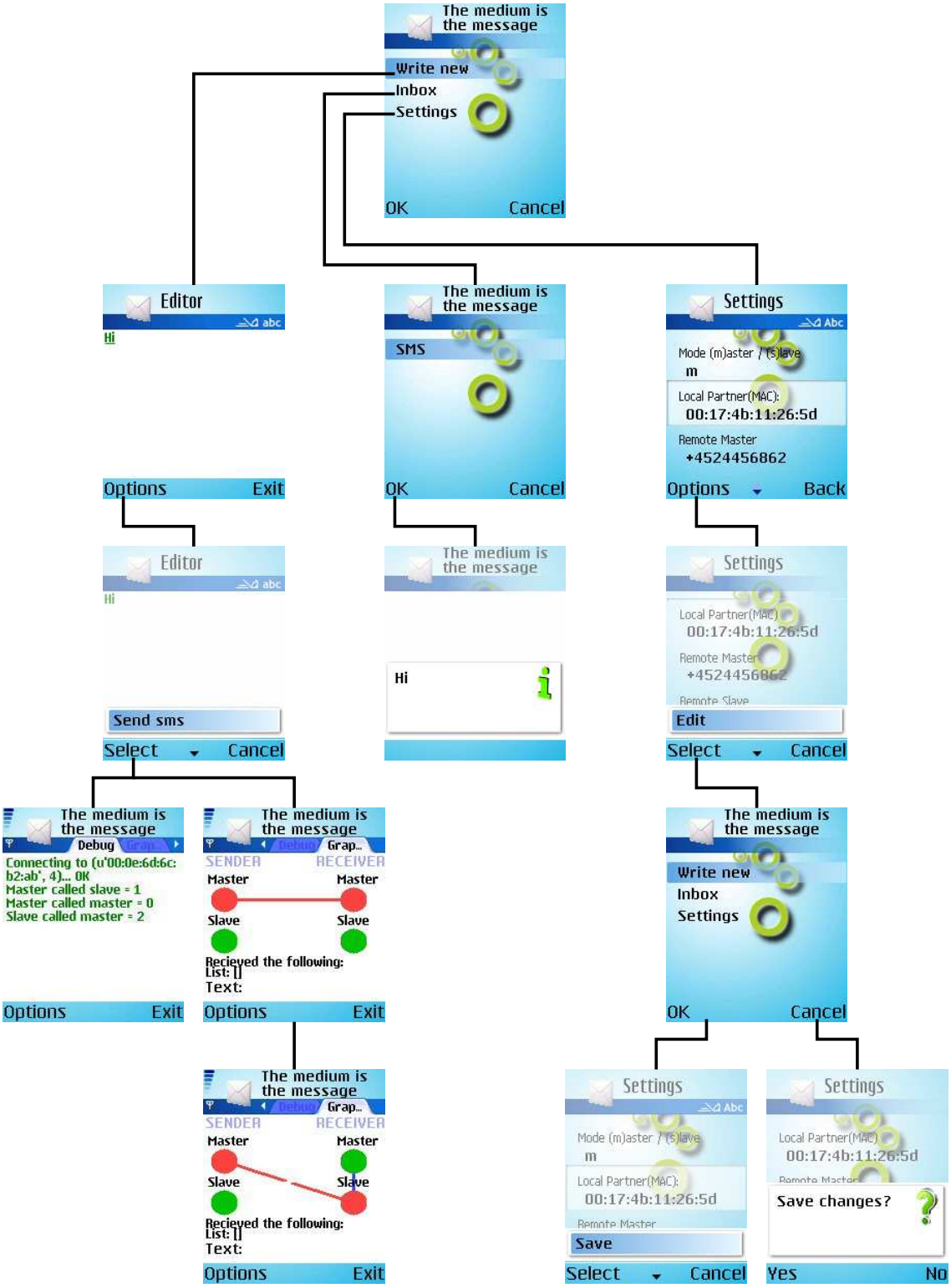


Figure B.2: The GUI of a phone in master mode

Appendix C

PyS60 extensions

To show how to extend PyS60 we have chosen the simple example to add two integers. To do this we need 3 files:

1. Source code (summation.cpp)
2. Project file (summation.mmp)
3. Build file (bld.inf)

They must be saved in the directory `c:\Symbian\8.0a\S60_2nd_FP\src\ext\summation`

Source code (summation.cpp)

The first file (summation.cpp) is the functionality we want to integrate in PyS60. It can be any function in C or C++ for Symbian.

Listing C.1: Source code

```
1 #include "Python.h"
2 #include "symbian_python_ext_util.h"
3 #include <e32std.h>
4
5 static PyObject* sum_args(PyObject*     /*self*/, PyObject* args)
6 {
7     int number1, number2;
8     if (!PyArg_ParseTuple(args, "ii", &number1, &number2))
9     {
```

```
10          return NULL;
11      }
12
13      int sum = number1 + number2;
14      return Py_BuildValue("i", sum);
15  }
16
17  static const PyMethodDef sum_methods[] = {
18      {"sum", (PyCFunction)sum_args, METH_VARARGS, "sums
            numbers"},
19      {0, 0}
20  };
21
22  DL_EXPORT( void) MODULE_INIT_FUNC()
23  {
24      Py_InitModule("summe", sum_methods);
25  }
26
27
28  //This function is mandatory in Symbian DLL's.
29
30  GLDEF_C TInt E32Dll(TDllReason)
31  {
32      return KErrNone;
33  }
```

Lines 5-15 in listing C.1 is the function we would like to call from PyS60. In this case it takes two integers as input and returns the sum. In line 8 the inputs are distributed to the addresses of the integers number1 and number2. `"ii"` indicates that there is two arguments. If no arguments are given the return value is null. The function `PyArg_ParseTuple()` checks the argument types and converts them to C values. It returns nonzero if the number of arguments and their type is correct and zero otherwise. The `Py_BuildValue()` function is the inverse of `PyArg_ParseTuple()` i.e. it converts C values into PyS60 objects which can be returned to the PyS60 environment.

To be able to call the function from PyS60 we need to declare a list of the functions available in the module. In this case there is only one function (`sum_args()`). Lines 17-20 creates our list by the name of `sum_methods[]` . Every element in this list consists of 4 parts:

1. The command used to call the function. (in this case `"sum"`)
2. The C function typecasted to a Python function (`(PyCFunction)sum_args`)

3. The calling convention to be used (generally METH_VARARGS)
4. A description of the function

The last element 0, 0 indicates the end of the list.

The module must be initialized and this is done in lines 22-25 by Py_InitModule() . First argument is Python name of the module and the second is the list of functions.

Lines 30-33 make the extension module work in Symbian and must always be included.

Project file (summation.mmp)

This file tells the compiler which module to compile and what's needed for this module.

Listing C.2: Project file

```
1  TARGETTYPE  dll
2  TARGET      summe.pyd
3
4  TARGETPATH  \system\libs
5
6  SYSTEMINCLUDE   \epoc32\include
7  SYSTEMINCLUDE   \epoc32\include\libc
8
9  userinclude   ..\..\core\Symbian
10 userinclude   ..\..\core\Include
11 userinclude   ..\..\core\Python
12
13 USERINCLUDE   .
14
15 LIBRARY     python222.lib
16 LIBRARY     euser.lib
17
18 LIBRARY   estlib.lib   /* Necessary only if you use the C standard
       library */
19
20 SOURCE      summation.cpp
```

In line 1 in listing C.2 TARGET is the Python name of the module. It is very important that this is the same as the first argument in Py_InitModule() in the source code. Lines 6-11 is paths of the various header files needed in the source code. Lines 15-18 is a list of the libraries needed in the compilation. SOURCE specifies the name of the source code.

Build file (bld.inf)

This file specifies which platforms the module should be compiled for and what project file to use.

Listing C.3: Build file

```
1 RJ_PLATFORMS
2 wins winscw armi
3
4 PRJ_MMPFILES
5 summation.mmp
```

In line 2 in listing C.3 `wins` is the PC emulator and `armi` is the ARM processor of the phone.

Compilation

To compile the module you have to open a DOS shell and make a virtual drive. This is done by:

```
subst t:  c:\Symbian\8.0a\S60_2nd_FP
set EPOCROOT=\
```

Go to the location of the module files, in this case:

```
t:\src\ext\summation
```

Run:

```
bldmake bldfiles
```

This creates an abld.bat file needed for the compiler

Then run either:

```
abld build wins udeb
abld freeze
abld build wins udeb
```

This compiles a summe.pyd file in your `t:\epoc32\release\wins\udeb\z\system\Libs` directory for the debug emulator.

Or:

```
abld build armi urel
```

```
abld freeze
abld build armi urel
```
This compiles a summe.pyd file in your `t:\epoc32\release\armi\urel\` directory for the phone.

The module works straight away for the emulator but to get the module to work in the phone you need to send summe.pyd from the `t:\epoc32\release\armi\urel\` directory via bluetooth to your phone.

Usage in PyS60

To use the new extension module in PyS60 you need to import it. In the Python console this is done by:

```
import summe
```

To use the function "sum" type:

```
summe.sum(8, 9)
```

This should return 17.

Appendix D

Spin-off applications

D.1 Call / No call

One of the applications which has been developed during this project is the ”Call / No call” application. It was made to give a better understanding of how this type of communication works, and to see if it was stable enough to send a message with. In this application the time interval is set to 25 seconds, because it has been developed to be used between two Danish phones with Danish sim cards.

The application is divided into two parts, the sender and receiver side, see figure D.1.

Figure D.1: The application for the sender side to the left and the receiver side to the right.

How to use the application

The two applications need to be installed on two different phones that both have Python installed. After this is done, start the application on the sender phone. when asked to type the phone number of the receiver phone and then type the message to send. Before sending the message the number of characters need to be counted, because the receiver phone needs to know this.

Start the application on the receiver phone and enter the number of characters it is going to receive, but do not press ”ok” until the sender phone informs that the receiver needs to be started. Press send on the sender phone.

The receiver phone will now get called in a sequence based upon the message and show this in on the screen, as on figure D.2. When the sender phone has finished sending the message, it will be displayed on both phones. As seen on figure D.3.

Figure D.2: The screen from both phones when the the text ”A test” is started to be transmited.

Figure D.3: The screen from both phones when the message is received.

The applications can be found on the CD included in this report.

D.2 Cluster call 4 to 1

An applications which have been developed during this project is the ”4 to 1 phone SMS” application. It was made to test how the communication works with Cluster calls, and to see if it was stable to send messages with. Furthermore the application tested the possibility of making a GUI in PyS60.

The application is only made for the receiver phone, which is able to combine the received calls into a text string. A screenshot of the application can be seen on figure D.4.

Figure D.4: The start screen of the ”4 to 1 phone SMS” application.

How to use the application

The application needs to be installed on a phone that have Python installed. After this is done, start the application, and set up the phone numbers of the callers under Options.

Call the phone in the sequence based upon what to be sent. The display will be updated as the calls are received, as seen on figure D.5 on the facing page.

After four calls a letter based on the received calls are displayed under *Received text: ()*. This can be seen on figure D.6 on the next page.

The applications can be found on the CD included in this report.

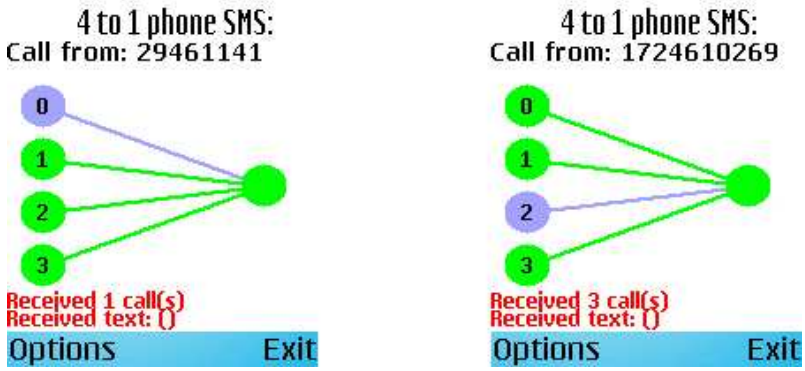


Figure D.5: The screen from the phone while being called, the gray line shows which phone that have just called.

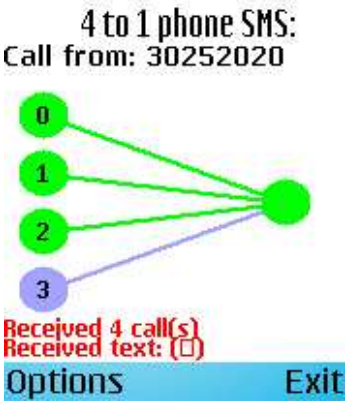


Figure D.6: A screenshot from the phone after it has received four calls.

Appendix E

Graphs of the call establishing time

This appendix contains graphs of the results from the test of the call establishing time. Figure E.1, E.2 on the next page and E.3 on the facing page shows the one test each and in figure E.4 on page 110 an overview is shown. The source data can be found on the CD.

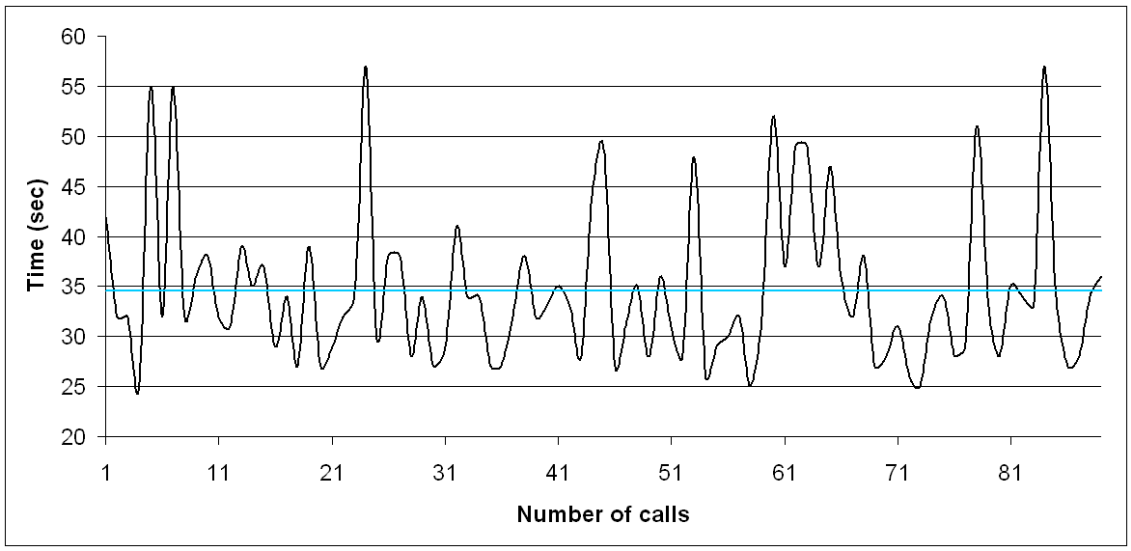


Figure E.1: This figure shows the call establishing time between two phones using German sim cards. The blue line is the average.

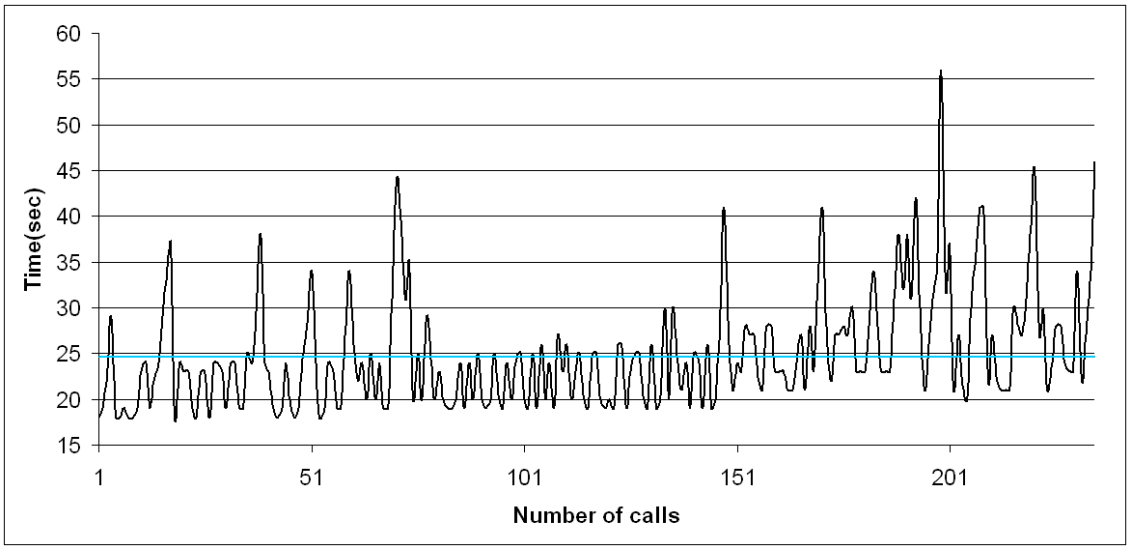


Figure E.2: This figure shows the call establishing time from a phone using German sim cards to a phone using Danish sim cards. The blue line is the average.

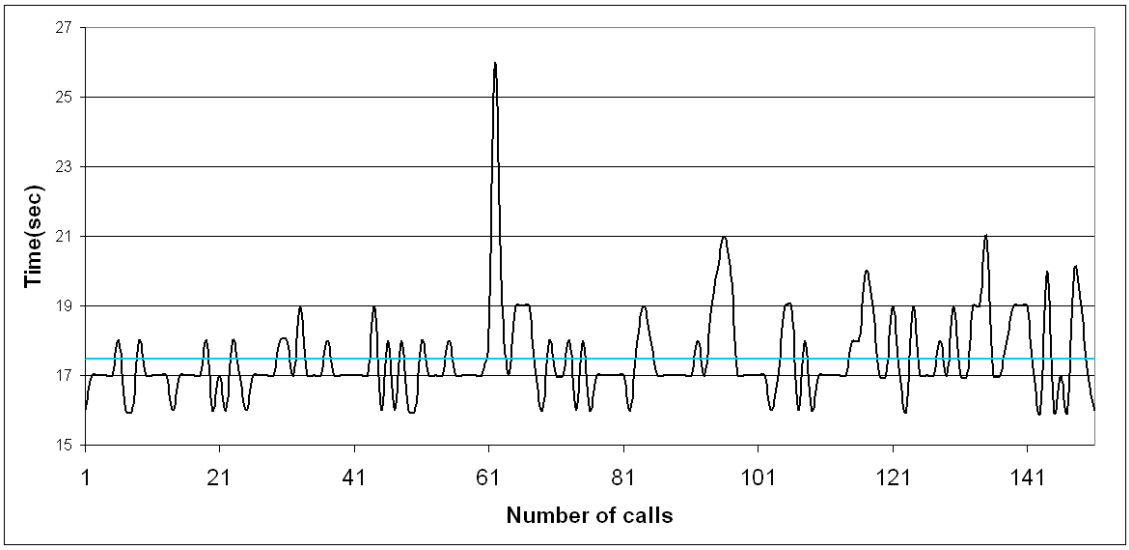


Figure E.3: This figure shows the call establishing time between two phones using Danish sim cards. The blue line is the average.

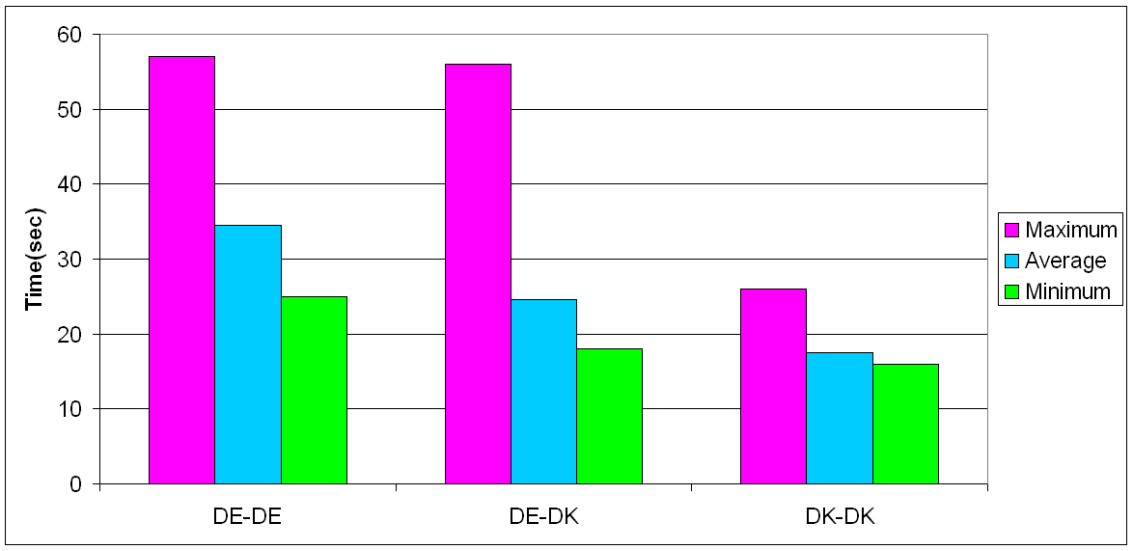


Figure E.4: This figure shows a comparison between the three tests.