CHALMERS

Mobile Broadband Module Simulation Framework

Simplifying Client Driver Development through Test Environment Control

Master of Science Thesis in the programme Computer Science

FREDRIK SANDELL
DAVID RUNEMALM

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, March 2012

Mobile Broadband Module Simulation Framework
Simplifying Client Driver Development through Test Environment Control

F. SANDELL,
D. RUNEMALM

Examiner: R. SNEDSBÖL

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover:
The mobile broadband simulator deployed and connected
to a laptop. The laptop screen shows running instances
of the applications SimDebug and Wireless Manager.

Department of Computer Science and Engineering
Göteborg, Sweden, March 2012

Acknowledgement

We would like to thank our supervisors - Rolf Snedsböl at Chalmers and Torgny Johansson at Ericsson - for their help and support throughout this thesis project.

Mobile Broadband Module Simulation Framework
Simplifying Client Driver Development through Test Environment Control
FREDRIK SANDELL, DAVID RUNEMALM
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Software drivers are typically hard to debug since their operation is closely linked to the functionality of the hardware for which they are developed. The problems with driver debugging is especially complicated if the hardware device has got interdependencies on complicated external systems, as is the case with 3G modems. The purpose of this thesis is to develop a 3G modem simulator framework that can be used to control the environment in which 3G modem drivers are developed. Using a simulator to control a development environment is not a new approach. However, up to this point a modem simulator has not been used during 3G modem driver development.

The modem simulator described in this thesis is implemented on a separate piece of hardware which ensures that it can be used with any host system. The simulator is highly configurable and can be used to induce 3G modem failures to test edge cases and stress test the driver software. Tests have been performed which shows that the system is indeed capable of simulating a 3G modem when attached to different host systems.

The mobile communication industry is constantly evolving, therefore future development of the simulator system is proposed that would increase the capabilities of the system and keep it up to date and ready for the next generation of modems.

Keywords: modem simulator, 3G modem, broadband modem, BeagleBoard-xM, MBM, MBMSF

Contents

Lexicon

ACM	-	Abstract Control Model
ASCII	-	American Standard Code for Information Interchange
ARM architecture	-	32-bit reduced instruction set architecture
AT-command	-	A command consisting of a string of one byte characters.
CDC	-	Communication Device Class
DDR memory	-	Double Data Rate memory
EDGE	-	Enhanced Data rates for GSM Evolution
GPS	-	Global Positioning System
GSM	-	Global System for Mobile Communication
IP	-	Internet Protocol
MBIM	-	Mobile Broadband Interface Model
MBMSF	-	Mobile Broadband Module Simulation Framework
NCM	-	Network Control Model
NMEA data	-	GPS data format standard
kbuild	-	"Kernel Build", the build system used in the Linux kernel
PCI	-	Peripheral Component Interconnect
TTY	-	Unix terminal. Name originates from TeleTYpewriter
UMTS	-	Universal Mobile Telecommunications System
USB	-	Universal Serial Bus

Chapter 1

Introduction

The gradual increase in cell phone network bandwidth over the last decade has given rise to a new industry within computer networking. The goal of this industry is to offer consumers Internet access by transmitting data through a link to the cell phone network. Through this link; laptops, tablets and other similar devices, from now on referred to only as hosts, can stay connected even without the presence of an Ethernet or a Wi-Fi connection.

The task of maintaining the link to the cell phone network is usually handled by a separate piece of hardware, a mobile broadband module, from now on referred to only as module. The module is usually connected to the host by using a USB link. By using an interface exposed by the connected module the host may transmit data through a cell phone network. The interface between the module and host does however require the host to have access to a set of drivers which controls the low level communication with the module.

1.1 Problem definition

The functionality of driver software is per definition intimately tied to the operation of the hardware with which it interacts and the module drivers are no different. This presents a problem when the drivers are to be tested since it means that it is hard to isolate and test the driver software separated from the rest of the module system functionality.

The development of driver software is usually carried out in parallel with the development of the module itself. Therefore, both the module drivers and the module are bound to contain errors. This leads to several complications when debugging the software drivers:

- Isolating the origin of a failure is often difficult since it may well be due to a fault in the module itself.
- It is also hard to stress test the driver software since only limited low level control over a module is available.

1.2 Purpose & Outcome

The purpose of this thesis have been to create a Mobile Broadband Module Simulator Framework(MBMSF) which enables isolated testing of the host drivers without the presence of a real module. The benefits of using a simulated module when developing drivers are threefold:

- A fault in a system using a simulated module can easily be traced to the driver software.
- Simulating a module allows development of drivers to start before the module hardware platform is available; thereby reducing time to market[43].
- Using a simulated module drastically increases the control over the module. This can be used to inject faults in the simulated module and thereby stress test the driver software.

Beyond the MBMSF implementation an evaluation of the completed system is presented to verify the implementation. Further, future system features are suggested and possible implementations are described to provide guidance when the MBMSF system is to be upgraded.

During the research for this thesis no information was found regarding systems that have been developed to simulate mobile broadband module functionality. Simulations with the purpose of facilitating debugging is however used in many areas of software development, for example embedded system development such as presented by A. Gosh et al. [24]. The most closely related work is the open source project Ofono phonesim which has been used when creating the implementation presented in this thesis. The Ofono phonesim is discussed in section 2.2.4.

1.3 Limitations

The thesis is based on problems faced by the Mobile Broadband Module(MBM) department within the company Ericsson AB. Because of this the implementation of MBMSF will be guaranteed to be compatible *only* with the Ericsson mobile broadband driver software. To implement a general solution compatible with other module systems is likely possible but most information about module to host interfaces are proprietary and therefore inaccessible.

Further the MBMSF is *only* intended to simulate the host to module interface. No attempts have been made to simulate other types of module interfaces, such as with SIM-card or cell phone network interfaces.

Chapter 2

Theoretical Framework

This chapter contains basic information for the context of this thesis project. Initially an overview of mobile broadband communication is given, seen from a module point-of-view. The chapter also presents an overview of the BeagleBoard-xM hardware as well as software frameworks which together will form the platform for the MBMSF.

2.1 Mobile Data Communication

The goal of a mobile data communication link is to enable devices connected to the cell phone network Internet access. For this to be achieved the cell phone network core is linked to the Internet infrastructure through a wired high speed connection. For the data packets to be made available for routing in the cell phone network core they need to be transmitted between the connected device and the cell phone network. This is done by means of a radio link. The cell phone network core is connected to a large number of base stations covering the geographical area in which the connected device is used. Each of these base stations contains radio receivers and transmitters and forward data packets between the connected device and the cell phone network core. The radio transmission between the base station and the connected devices are controlled by several different communication protocols and which of these are used at any given moment is determined by the physical environment in which the device is located and network settings. Figure 2.1 displays an overview of a mobile data communication link. Currently in a European network a 3G compatible device is required to support GSM, EDGE and several different releases of the UMTS protocol (commonly referred to as 3G) [48]. If the device is to be sold globally even more protocols

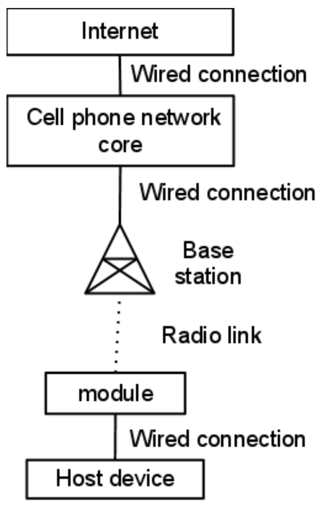


Figure 2.1: Network Overview

are required. Within each of the protocols several modes of operation exists, each with its own control messages [42]. In the end the control data complexity required to maintain a continuous radio connection towards a cell phone network is considerable. Because of the complexity associated with the radio transmission and the requirement of special hardware, radio transmitter and receiver, the cell phone network communication functionality is usually removed from the rest of the device and placed in a separate physical module. Having this functionality in a separate module enables it to be added to a device more easily than if it were to be integrated into the device architecture. A single version of such a module can then be integrated into a variety of devices with a minimal amount of device adaption required.

2.1.1 Mobile Broadband Module

The task of the mobile broadband module is to handle the complexities of cell phone network communication and present a simple standardized network interface to the host. A common way to connect a mobile broadband module to a host is to use the Universal Serial Bus (USB) standard. More information about the USB protocols are available in section 2.3 in this chapter.

The module interfaces

Even though the USB protocol is most known for connecting external units to a host device it is also common to use the USB bus for attaching units built into the host device. This is the case with the Ericsson mobile broadband module which is embedded into a host device and usually never visible to the end user. A picture of a detached Ericsson F3607gw mobile broadband module can be viewed in Figure 2.2. The modules supplied by Ericsson may have a form factor of either full or half size PCI Express mini card [25]. The module displayed in Figure 2.2 have the full size form factor. Each module regardless of its form factor have three external interfaces, the main one being the PCI mini express interface marked with A in Figure 2.2. This interface connects directly to the PCI bus of the host and it is over this interface all module to host communication occur. The PCI express standard supports the use of either the USB or PCI protocols for communication [41]. The Ericsson mobile broadband module uses the USB protocol standard for communicating with the host. This fact in combination with the ubiquitous presence of USB interfaces on modern computers greatly simplified the development of the MBMSF. Interfaces marked with B and C, on Figure 2.2, are the sockets for attaching GPS and Radio antennas. The antennas for these two interfaces are normally integrated into the host device to increase the antenna area.

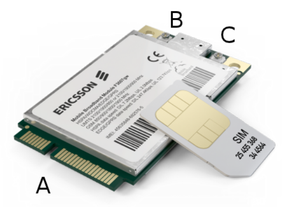


Figure 2.2: Detached Ericsson Mobile Broadband Module F3607gw with SIM-card. A: PCI mini express interface. B: Radio antenna socket. C: GPS antenna socket.

Subscriber Identity Module (SIM)

To connect to a cell phone network a device needs to be registered with a mobile operator. The registration allows the device to use the network infrastructure maintained by the operator. The authentication process when connecting to a cell phone network requires the connecting device to provide the network with details of the users subscription. In particular the International Mobile Subscriber Number(IMSI) is provided to the network for subscription identification. The IMSI number and other subscriber details are usually stored in a Subscribed Identity Module(SIM) card [36]. Apart from the IMSI number and other parameters used for subscriber identification several other properties are stored on the SIM card. Such properties may include specific billing options, quality of service parameters and many more. An Ericsson mobile broadband module communicates with a SIM card reader embedded in the host to retrieve the parameters stored on the sim card trough the USB interface.

Module AT-command communication

As previously explained the module communicates with the host by using a USB protocol. As explained in section 2.3 the USB protocol can be extended with additional protocol layers to let a USB interface provide several different types of transmission interfaces. One of these interfaces used in the module to host connection is a simple serial character interface. Over this interface; control data is transmitted between the host and module in the form of AT-commands.

AT-commands is an old standard for communicating with computer modems by serially transmitting character strings [37]. The host simply transmits a character string to the modem over a serial link, in this case a special interface exposed by the USB interface, and waits for a serial character response. The basic structure of AT-commands have remained largely unaltered over the years and AT-command can roughly be divided into four basic variants, as seen in table 2.1.

Type	Syntax
Basic AT-command	ATCMD1
Extended AT-command	AT+CMD2
Set parameter(s)	AT+CMD3=1,2
Read parameter(s)	AT+CMD3=?

Table 2.1: AT command types

Common for all forms of AT-commands are the prefix of "AT" which denotes that what follows is an AT-command. Even though the fundamental principle of AT-commands is very simple, the specification of the standard AT-commands is complex. A large number of AT-commands are required to cater to the various functions that a modem is likely to support. Each vendor usually also add their own proprietary AT-command specification which extends the standard specification with product-specific AT-commands. Ericsson is no exception and provides their own AT-command specification that allows the host to exert greater control over the module than otherwise possible through the standard AT-command set.

2.2 Development Tools and Material

Many different tools have been used in the creation of the MBMSF system. Physically the MBMSF runs on stand-alone hardware with a processor architecture that differs from what is normally used in PCs. Because of this cross-compiling have been necessary for any program that is intended to run on the MBMSF system. The most important tools and contributions to the MBMSF system is described below.

2.2.1 BeagleBoard-xM

BeagleBoard-xM [27] is a hardware board with interfaces and processing power comparable to that of a PC, with the additional properties that it is significantly smaller and cheaper. It is an evolved version of the original BeagleBoard [26] that was developed at Texas Instruments (TI) to demonstrate the OMAP3530 system-on-a-chip ARM processor. It has been widely adopted by development- and hobby projects and has a relatively large open-source community supporting it. The most notable differences between the BeagleBoard-xM and its predecessor is the DM3730 1GHz-processor which replaced the OMAP3530 720 MHz processor, double amount of DDR memory (512MB) and that it comes with four instead of one USB ports. Furthermore it has no NAND memory whereas the BeagleBoard came with a NAND module of 256MB capacity. [27] The community support, price, performance, extensive peripheral support and low power consumption of the BeagleBoard-xM makes it a suitable platform for many types of embedded projects. Several operating systems has already been ported to it, among them Linux and Unix distributions such as Ubuntu [17] and Angstrom [16], but also Android [15] and non-UNIX based ones such as Windows Embedded [18].

2.2.2 Cross-Building

A common way to develop software for an embedded system is to use a cross-compiling toolchain to first build the binaries on a host computer and then transfer them to the embedded device. This is sometimes a necessary method e.g. when developing for a micro controller without an OS with software development support. In that case it would be unfeasible to develop directly on the target machine, but there are also several convenience reasons for using a cross-build system. For example the same workstation can be used to develop for multiple platforms and a potential integrated development environment (IDE) does not have to be setup every time the target machine has to be restarted. Cross-building support for the BeagleBoard-xM platform exists in form of toolchains as well as automated cross-building environments. A commonly used toolchain is the GNU ARM toolchain [5].

OpenEmbedded (OE) [14] is a cross-building environment supporting a wide-range of embedded devices, among them the BeagleBoard-xM. This tool automates the process of building Debian packages or whole Linux distributions by using predefined ”recipes-files” and is explained in more detail in section 2.2.6.

2.2.3 Ofono

Ofono is a software framework which can be used by a host device to handle interaction with cell phones and mobile module equipment [13]. The Ofono framework includes features common for module modem interaction such as call handling, network selection and several other features. The Ofono framework uses module specific drivers for communication with the module itself but presents a generic interface for module interaction to higher layers in the host software stack. The generic software interface given by Ofono can thus be used by host software independently of module type and vendor. The Ericsson Linux drivers are written as a plugin to the Ofono framework and conforming to the Ofono specifications. This allows the host driver development in Linux systems to become independent of higher layers in the host software stack and thereby allowing for the use of generic connection management software.

2.2.4 PhoneSim

A key software component used in the thesis is Ofono phonesim, from now on referred to as phonesim. Phonesim is a project connected to the Ofono project and more information regarding the phonesim project can be found in on the Ofono web site [13]. Phonesim was originally written to simulate a cell phone on the host. The phonesim software establishes an interface through which the Ofono framework is able to transmit AT-commands. By doing this, phonesim bypasses the driver layer of the Ofono framework and allows higher layer Ofono software sections to be tested without the need for hardware or drivers. It accomplishes the simulation by maintaining a state machine which specifies in what logical state the simulated module is. Most of the state machine related functionality in the MBMSF is based on the phonesim implementation and a detailed description of that MBMSF subsystem can be found in B.1.1.

2.2.5 GNU Make

GNU Make[6] is a free version of a common development tool, Make. The purpose of the tool is to ease compilation of software projects by automatically resolving dependencies between source files and thereby compile them in the correct order. GNU Make, as other versions of Make, ensures that only updated versions of the source files are compiled. This ensures short build times when only a few source files have been updated in a large software project. Gnu Make also have several other benefits such as hiding build information from end users and providing an easy method of installing and un-installing software from a system.

2.2.6 OpenEmbedded

OpenEmbedded [14] is a build tool used for building software targeted towards embedded systems. The actual building core of OpenEmbedded consists of the build tool *BitBake* [1]. OpenEmbedded additionally incorporates a framework on top of BitBake for cross-compilation, packaging and installation of packages. With these additional capabilities, OpenEmbedded is able to cross-build whole distributions for embedded devices.

BitBake uses *recipes* to define packages, its dependencies on other packages, and in what order to build them. These recipes is analogous to make's Makefiles (see Section 2.4.1), although recipes are much more independent in relation to each other than make's Makefiles. This recipe independence naturally provides a higher flexibility in creating and maintaining, as well as composing recipes. This has lead to OpenEmbedded being a popular tool for maintaining distributions that with a high degree of automation is able to be cross-built for multiple hardware platforms. Furthermore, OpenEmbedded provides a user-generated database of recipes, encouraging distribution and re-usability of community work for platform support of software packages.

OpenEmbedded was used in the MBMSF project to cross-build the Linux kernel (along with some additional user-land tools). In the build process, a cross-compilation toolchain was created that could be used later in the project for cross-building the kernel module (see section 4.1), together with make.

2.2.7 Qt

The simulation framework which is described in the method section of the report is written using the C++ framework Qt. Qt was originally developed by Nokia [20] as a way of increasing C++ portability by introducing a set of libraries, a special build system and additional syntactic features that are pre-processed by the Qt build system. It is the Qt build system that enables the introduction of extra syntactic features and a higher degree of portability. The build system is called qmake and is essentially a multi layered Make file system[19]. The qmake is very similar to the more general cmake [2] pre-processing system and the two systems can to some extend be used interchangeably. Before software written in Qt is compiled the qmake parses a special configuration file, the project file, and the source code; qmake then generates additional source files when needed to support the extra Qt syntax features. The qmake also generates ordinary Makefiles when performing the pre-processing based on the input in the project file. A standard gcc compiler can then be invoked to compile the source code based on the instructions in the generated Makefiles. Since the Makefiles are generated based on a single project file, only this file requires modification when new directives needs to be given to the compiler. This simplifies cross-compilation since the switch for target platform is reduced to a single variable in the project file.

2.3 Universal Serial Bus (USB)

USB [38] is a serial bus standard in the personal computer (PC) interconnectivity domain. It is commonly used to connect peripheral devices to a PC to extend its functionality. Examples of common type of devices to connect are printers, keyboards and web cameras. The protocol is currently at revision 3.0 (USB 3.0) [39] but revision 2.0 (USB 2.0) [38] is still of most widespread use in consumer products at the time of writing. USB supports plug n' play, which makes it flexible to the end-user, and it also provides high data transfer speeds. The maximum theoretical speed limits are 480 Mbps for USB 2.0 and 5.0 Gbps for USB 3.0.

This chapter is a summary of the USB architecture, providing a knowledge base for the discussions throughout the paper involving the USB protocol. It is especially relevant for the implementation section 4.1 of the USB driver that has been developed in this project. That driver is formally defined as a *Linux USB gadget driver* but from now on it's referenced simply as a *gadget driver*. The area of focus for this driver is marked out as a logical entity in Figure 2.3 which depicts a logical overview of the USB system.

For information in greater detail than what is provided by this chapter, refer to the latest specification of USB 3.0 [39].

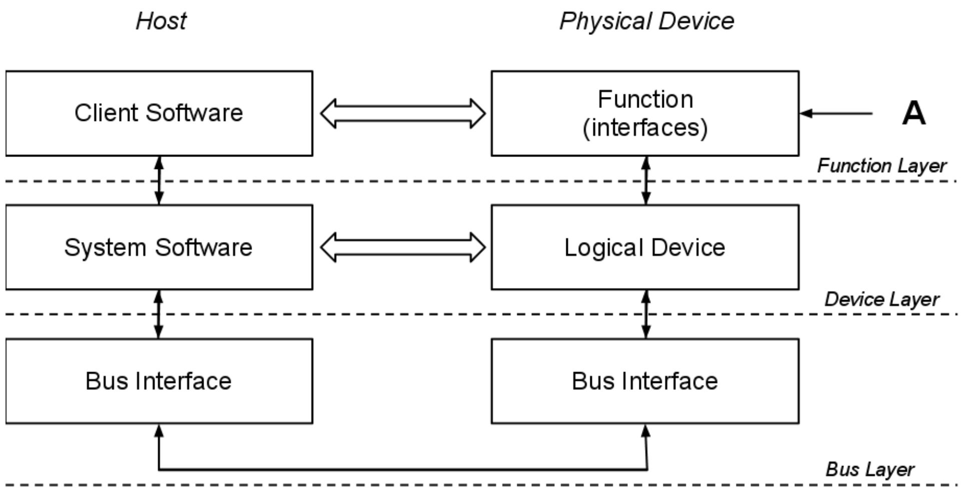


Figure 2.3: Architectural overview of the USB system. The focus of the implementation for the gadget driver is marked out with *A*.

2.3.1 Specification

The USB protocol is maintained by the USB Implementers Forum [22] and is currently at its third revision. Note however that because this project uses software implementing USB 2.0, we will from now on refer to USB 2.0 whenever the USB specification is referenced throughout the text.

2.3.2 Layering

USB is divided into three logical layers. These are depicted in Figure 2.3 and from the bottom up they are; *bus layer*, *device layer* and *function layer*. The bus layer, is where the actual data communication between the physical devices takes place through cables, hubs and other types of infrastructure. At the middle layer, functionality specific to the devices operating systems are implemented. This software supports the USB system on respective device and is independent of the devices that are actually connected to the host. Finally, the top layer is where the USB *functions* are implemented. These functions are a collection of (mostly standardized) interfaces, such as Ethernet, serial or mass-storage. It can also expose a proprietary protocol interface such as a GPS or modem. For this project, the gadget driver will expose ACM and NCM interfaces at the USB function entity (section 4.1.1).

2.3.3 Bus Topology

The physical USB bus topology is of a tiered-star type and because of this creates a tree-like structure when multiple nodes are connected, as seen in Figure 2.4. The nodes in the tree are divided into classes called *hubs* and *functions*. The hubs provides ports for functions to connect to. Thus, the functions are leaves in the USB topology tree and as already mentioned they are endpoints in the communications. The host, which also is responsible for constructing and maintaining this bus tree, provides a *root hub*. From this hub, all other extensions to the tree are connected; which can be both hubs and functions.

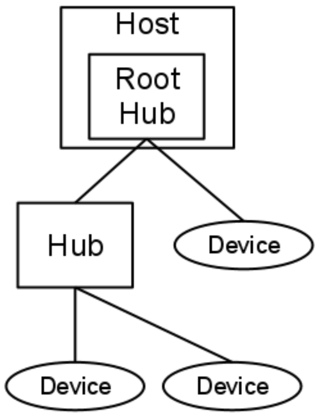


Figure 2.4: USB physical topology example.

2.3.4 Device Types

Two terms that is used to classify devices are *compound-* and *composite-*devices. Compound devices contains a hub which has one or more functions permanently connected to it. The emphasis here is that all nodes are contained in one single package. Composite devices however does not include a hub. They are simply functions but they all have multiple interfaces, which are controlled independently of each other. As an example, the MBMSF is a composite device because it does not include a hub but it provides multiple interfaces (one NCM and three ACM). If a device does not provide multiple interfaces and is not a compound device it is simply called a *device*.

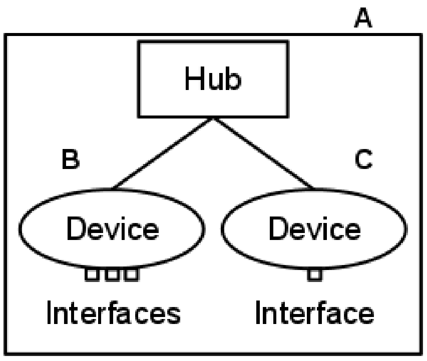


Figure 2.5: USB device types:
A) Compound device
B) Composite device
C) Regular device

2.3.5 Configurations, Interfaces and Endpoints

There are some concepts in the USB protocol that is vital to understand in order to understand the enumeration process. These are *configurations*, *interfaces* and *endpoints*, as can be observed in Figure 2.6.

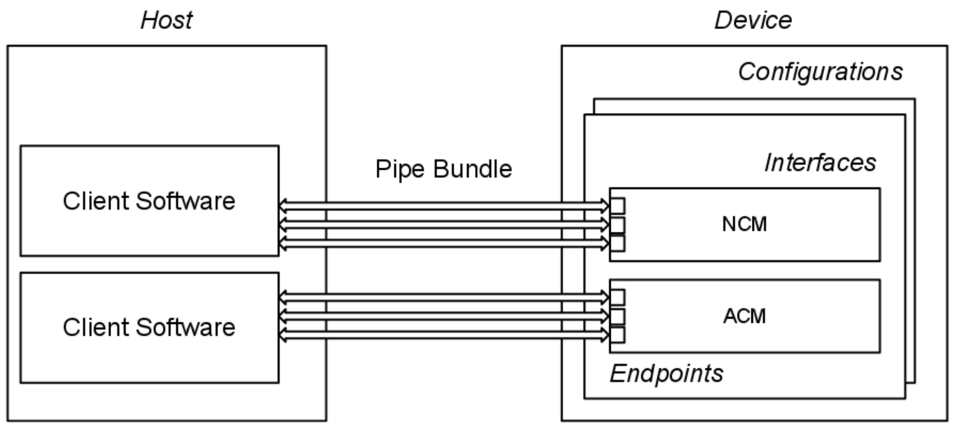


Figure 2.6: The concept of configurations, interfaces, endpoints and pipes.

A configuration is fundamentally defined as a set of interfaces where interfaces provides functionality to the host device. A device may implement several configurations and the host can instruct the device to switch configuration. By switching configuration the device exposes different functionality at the function layer. As an example, a device may have an initial configuration that makes the device show up to the host as mass-storage, containing drivers intended to be installed. When these drivers has been installed and brought into operation, they will instruct the device to switch configuration, after which the device instead of mass-storage will provide whatever functionality it was created for, such as a modem or printer.

Pipes

While data is physically transmitted between host and device over cables at the physical layer, the very same flows are also abstracted as pipes interconnecting entities at the upper layers, these pipes are depicted in Figure 2.6. The pipes are all terminated at so called endpoints which are bundled together and connected to interfaces on either side of a USB connection.

In all USB connections a special pipe is required, the Default Control Pipe (DCP). This pipe is special because it is not connected to any functions in the function layer of the USB stack. Instead, it is used to generically manipulate the device by sending control messages, and plays a vital role in the enumeration of a device.

2.3.6 Enumeration

Enumeration is the process carried out when a device is attached to the USB. During this process, the host uses the default control pipe to query the device for information, giving it an address and configuring it. The configuration involves activating a configuration and by this selecting which group of interfaces the device should activate.

2.4 Linux Kernel

The MBMSF utilizes a Linux version 2.6.39 [31] operating system kernel. This chapter is devoted to describe the subsystems and parts of that kernel relevant for this project. This means that if someone wants to reproduce the MBMSF gadget driver, they will find useful background information for this here.

The chapter begins with explaining the concept of kernel modules in the Linux core, as well as how these are developed by using "kbuild", the Linux kernel automated build script. After that follows information about the USB subsystem including the architecture and development of USB device drivers (gadget drivers). Finally, the end of this chapter will specifically explain the ACM- and NCM interface drivers developed by the Linux community for Linux kernel 2.6.39, since these are the drivers utilized by the MBMFS gadget driver in this thesis.

2.4.1 KBuild

The build system used by Linux is called *kbuild*, which stands for "kernel build" [45]. This build system consists of hundreds of *Makefiles* residing in different directories throughout the Linux source code. Every Makefile defines what to be built (the *targets*) in that specific directory, and also in what way. The target is specified to be built as either a *built-in* or a kernel module, (the latter being the case for the gadget driver). These targets in the Makefiles that explains how and what to build are used by the *make* [35] command line tool that parses and executes relevant commands needed to generate corresponding output.

The Makefiles also has means to specify to make how to recursively descend down specific directories in the source tree, passing arguments to targets to execute in those directories authoritative Makefiles.

There are five parts of the Makefiles [29]:

- Makefile (the "top" Makefile)
- .config (kernel configuration file, referenced by kbuild Makefiles)
- arch/<arch>/Makefile (the Makefile specific to <arch> architecture)
- scripts/Makefile.* (common rules for all Makefiles)
- kbuild Makefiles (authoritative Makefiles used in kernel build)

The top Makefile together with the kbuild Makefiles forms the tree of Makefiles that is involved in building the kernel or kernel modules. To start such a build, the *make* command is executed on the top Makefile.

For more information regarding the kbuild system, please refer to the Makefile documentation [29].

2.4.2 Kernel Modules

A kernel module is fundamentally a loadable piece of code that may extend a running kernel with new functionality when loaded. The gadget driver is built in shape of a loadable kernel module. By allowing code to be loaded on-demand like this offers advantages such as flexibility and minimal memory footprint by not bloating the running kernel with unused program code. However, this flexibility can come to the cost of lower efficiency due to the overhead involved in the kernel mechanisms that provides this functionality.

Building an External Module

Kernel modules are usually built at the same time as the kernel itself. The modules to build is then specified in the .config file mentioned earlier and the module source code is part of the kernel source tree. However, it is sometimes desirable to build a module of which the source code is separated from the kernel source tree. It might even be cross-built from another machine, as is the case for the gadget driver module in this project. Modules built this way in Linux is referred to as *out-of-tree-* or *external* modules [28].

There are some prerequisites for building an external module. The source code of the target kernel as well as the configuration and header files generated in the build is needed. If this cannot be acquired from e.g. the source directory of the target kernel, or as a Linux distribution package, there are other ways of setting these files up. The steps to take then are:

1. Get the source code for the target kernel version.

2. Make sure the .config file contains the same parameter values as those used for building the kernel, (these are commonly stored by Linux distributions at */boot/config-<kernel-version>*).

3. Run *$ make modules_prepare* in the source directory to prepare the source for external build. This will make sure all information that is required for an external build exists, such as e.g. header files.

4. Make sure the top Makefile defines the exact same version information as the target kernels Makefile.

When the kernel source code, configuration and header files are set up as defined above, one last file is needed before starting the build. This is a Makefile residing in the module source directory, containing information that make parses in order to resolve object- and source file dependencies for the *.ko* module file that is to be built. See [28] for more information about the syntax for defining this. After this, all is set for externally building the module by executing the following command:

```
$ make -C <kernel> M=<source>
```

In the above command, <*kernel*> and <*source*> are file system paths to the pre-built kernel source and module source, respectively. What basically will happen is that make will first switch to the <kernel> directory before operating on the authoritative Makefile there. The *M* parameter tells make that external modules are to be built and so the *modules* target in that Makefile will be executed. This target is defined to parse the Makefile in <*source*> as explained above, and build modules accordingly.

Loading

Loading a kernel module means installing the module into a running kernel. Most Linux distributions based on Linux kernel 2.6.39 provides a package suitable for this task named *module-init-tools* [30]. This package contains some useful user-space tools, among which are:

- insmod (load a single module)
- modprobe (load a module and it's dependencies)
- modinfo (show information about a module)
- lsmod (list modules currently loaded)
- depmod (create list of module dependencies)
- update-modules (generate modules.conf)

The recommended tool for loading a module is modprobe [50], because it loads not only the wanted module but also any other modules it may be dependent on. A module is dependent on another if it uses *symbols* exported by it. These symbols are logically module services and are exported by a module by using EXPORT_SYMBOL in the module's source code [49].

The recommended way to load a module (such as e.g. the gadget driver module g_mbm), is by executing the following command in a terminal:

```
$ modprobe g_mbm
```

The following chain of events are:

1. The file modules.dep is scanned for module dependencies.
2. If modules are found in step 1; insmod is issued on every one of these to load them.
3. Insmod is finally called on g_mbm, effectively loading it.

The file modules.dep is maintained by the depmod utility. When called, depmod will search modules for symbols and derive dependencies between modules which are then recorded in module.dep. To refresh this file with dependencies among all modules, issue the following command in a terminal:

```
$ depmod -a
```

Auto-Load

The following minimal actions is performed to make a module automatically load at kernel boot time:

- Create file <module-name> in */etc/modutils/<module-name>* with content <module-name>.
- Run the command *update-modules*.

update-modules will thereafter make sure the module is loaded at every boot.

2.4.3 Linux on USB devices

A peripheral board embedding Linux and acting in the USB device (slave) role, will utilize the Linux-USB Gadget API Framework (gadget API), see Figure 2.7 below. This framework is part of the Linux kernel since version 2.4 and is used to support gadget driver development in Linux. It supports writing simple device drivers as well as composite ones, incorporating one single interface and configuration as well as multiple ones.

Currently, there are several fundamental drivers providing "basic" interfaces such as serial, Ethernet and file-storage. The task of creating a composite driver is thereby simplified by putting two or more of these interfaces together.

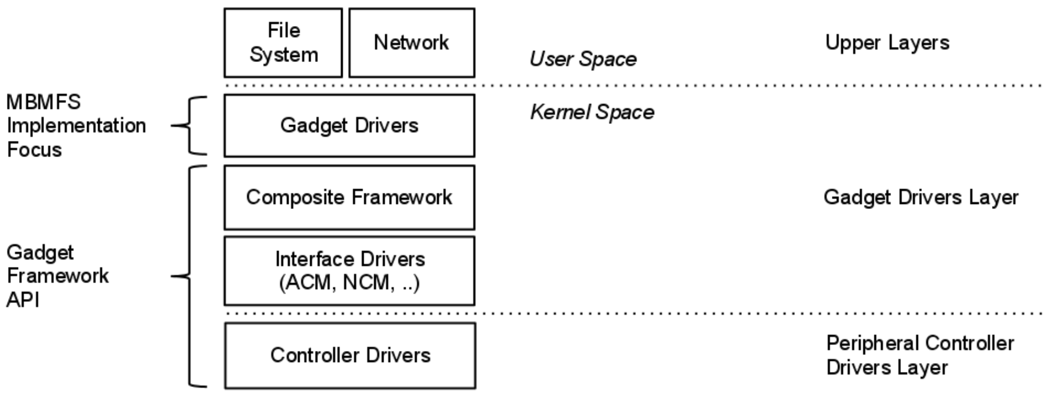


Figure 2.7: Linux-USB Gadget API Framework and gadget driver overview. The implementation focus for the gadget driver in this project is marked out.

The *peripheral controller layer* at the bottom is sometimes further divided into a platform-dependent and a platform-independent layer. The platform-dependent layer contains controller drivers which are specific to the controller used in the system. This means the other code throughout the entities can be generic. Exceptions may occur if the hardware does not support the functions implemented in e.g. an interface driver.

Gadget drivers are implemented at the *gadget drivers layer*. These drivers may use the composite framework if a composite driver is to be developed. (Read section 2.3.4 for more information about composite devices). The composite driver exploits the interface drivers library, fundamentally providing the glue to tie these interfaces together into a driver with multiple interfaces. This approach was chosen for the gadget driver development in this project. More about the implementation of the gadget driver is found in section 4.1.

2.4.4 Interface Drivers

Fundamentally, the interface drivers already mentioned above are "building blocks" that can be combined into a composite driver. There has been several interface drivers developed by Linux community members and they are available through the Linux source tree. Among these are [11]:

- GadgetZero (Essential for controller driver testing).
- GadgetFs (Fundamentally a wrapper, providing a user-mode API).
- File-backed Storage (Implements the USB mass-storage class).
- Serial (Used for serial data communication).
- CDC-Ethernet and -NCM (Subclasses of USB Communications Device Class).

Serial driver

The serial.c [32] interface driver implements the CDC-ACM protocol to provide a serial interface. The CDC-ACM protocol is suitable for transferring character data over a USB link, such as e.g. AT-commands.

Network driver

The CDC specification [23] defines two sub-classes for network data. These are CDC-ECM and CDC-NCM [33]. CDC-ECM sends a single ethernet frame in every USB package. By aggregating frames there is room for performance improvements in the protocol, which is the main reason behind the development of CDC-NCM. The frame aggregation implies less interrupts to be triggered and a better overall performance.

However, the CDC-NCM implementation in Linux drivers lacks the frame aggregation support. This means that the data throughput and performance overall is affected negatively. This does not directly affect the usage of the MBMSF today but might be an issue in a future deployment. This topic is discussed in section 7.2.

Chapter 3

Design Decisions

During the course of the project several important decisions have been taken regarding the design of the MBMSF implementation. In each case the benefits of a design decision have been weighted against the alternative options that were available. This chapter is intended to shed light on the most important of these design decisions and thereby provide an explanation to the implementation.

To understand the design decisions it is important to understand the technical requirements imposed on the implementation. The final objective of the MBMSF implementation is to simulate a module in a way which makes it indistinguishable from a real module, from the hosts perspective. This minimally requires at least three things:

- Make the MBMSF present a correct interface towards the drivers on the host.
- Ensure that the MBMSF is capable of responding to control communication, i.e. AT-commands.
- Ensure that data traffic can be routed through the interface that MBMSF presents to the host.

The natural starting point in the development of the MBMSF was to establish on which platform it would run, since this would decide how the MBMSF presented itself to the host.

3.1 Platform

Inarguably, the most important design decision have been regarding what platform to use. Two alternatives existed in the beginning of the project: Creating an all software simulation of a module run on the host or run a software simulation of a module on a separate piece of hardware.

Running on the host

Creating simulation on the host had the benefit of being simpler to implement and thereby likely to have a shorter development time. A framework for simulating hardware devices in the Windows operating systems exist, and could likely have been used to further speed up the development process [3]. The approach did however suffer a major drawback. It is completely platform dependent. The implementation of hardware enumeration and handling differ between different operating systems [12][21]. Creating a simulator implementation on the host machine would require the MBMSF to interfere with the operating systems normal hardware procedures. Communication bound for the external hardware module that MBMSF simulated would need to be re-routed to the MBMSF software system running on the host operating system. Interfering with the hardware procedures in an operating system independent way would in practice be difficult because of the differences that exist in the hardware enumeration and handling. A software only implementation of the MBMSF system would therefore likely need a different implementation for each operating system that it was to support. The Ericsson department that ordered the thesis performed driver development on at least Linux, Windows and Android and this property was therefore considered a significant drawback.

Running on separate hardware

The second alternative was to implement the MBMSF on a separate piece of hardware. As explained in Section 2.1.1 a real module communicates with the host through the use of the USB bus. This property of the real system would allow a separate system to simulate a module, connect to the host through an external USB interface, and the host system would not be able to tell the difference. This approach would be independent on which operating system the host was running since the hardware interface for a real module is common for all systems. A drawback with this approach was that the implementation of the MBMSF was likely to be more complicated due to the fact that the operating system that was to be used on the platform needed to be cross compiled along with all software that were to run on the MBMSF. Furthermore drivers for the MBMSF system would need to be modified or developed to present the interface that the host system would expect from a real module.

In the end the benefits of operating system independence in the second approach outweighs the benefit of simplicity in the first approach. This especially since the driver development for each new module in Ericsson is performed for Android, Linux and Windows. A multi-platform solution would therefore be of much more use to the Ericsson driver development team. Because of this the MBMSF is implemented on a system-on-a-chip connected to the host by a standard USB cable.

3.2 Software

A real module uses a state machine to organize the communication with the cellphone network and the host. That the MBMSF would have to use the same method was never in doubt since the AT-communication which it maintains are of a statefull nature. For example the AT-command response to many AT-commands differ depending on whether or not the module is in a connected state. However the implementation of the MBMSF state machine was under discussion. Two alternatives existed, implementing the state machine from scratch, or try to use some already existing implementation.

Implementing the MBMSF software from scratch was attractive because it would allow the use of any language. Using a familiar language would possibly have shortened the development time and therefore left room for more features to be included. However since the implementation would include a fairly standard state machine it was deemed beneficial to look for an already existing solution for at least the state machine part of the simulator implementation.

Such a solution was found, namely the open source project called ”phonesim”. The implementation supplied the state machine system that was sought and it was decided that much of the development time could be cut if phonesim could be used. Even though the documentation was non existent and phonesim had been developed to run on x86 architecture the benefits of using the phonesim outweighs the implement-from-scratch alternative.

3.3 The configuration system

Since the purpose of the MBMSF is to aid in the debugging of software drivers it is important that the module can be easily configured to conform to a specific test case. When deciding how to implement the configuration process two alternatives were examined.

XML approach

The first alternative was a straight forward XML configuration system. The design of this system would encompass two XML files that would specify all state machine configurations that could apply to the MBMSF system. One file would fill the same purpose that the state machine definition XML file do in the current implementation, simply defining the correct module behavior. The other XML file, the AT alteration XML, would apply the alterations to the AT-command responses. Keeping the AT-command alterations in a separate XML file would ensure greater modularity and each specific test scenario could be kept in a separate AT-command alteration file.

This XML approach was appealing since the XML parsing facilities were already implemented and used to interpret the state machine definition XML. However the approach also had drawbacks. Since it was clear that some AT-command modifications would only need to be applied under certain environment conditions, such as for example low network coverage, it was obvious that some kind of conditional statement would be required. The only conditional mechanisms that could be used without any additions to the XML configuration syntax lay in the state machine. Changing the XML syntax sufficiently to support conditional statements would be cumbersome to use and an alternative was needed.

ECMAScript

Since the configuration required evaluation of possibly complex boolean expressions it was obvious to look for a script solution, which would naturally support such operations. In a recent version of the Qt framework an ECMAScript engine was included into its library. The ECMAScript language standard is based on the several script systems, for example the well known JavaScript technology, and ECMAScript have support complex conditional statements[40].

However, some uncertainties regarding the script solution were still present since no documentation of the use of the script system for ARM architecture existed. Because of the instruction set used on ARM processor on the BeagleBoard-xM some features in the logging library had proven to be unusable; fears were that the same could have been the case for features of the script library. The script solution did also mean a more complex implementation and a slightly increased MBMSF configuration complexity.

In the end the benefits of the script solution resulted in the current implementation using scripts and a set of configuration files. The configuration process may require the user to apply changes in more files than what would have been the case using the XML solution, but the flexibility in terms of configuration is far greater. Since the end user in the scope of this thesis is software engineers the slightly increased configuration complexity was considered a low price to pay for the flexibility the script implementation offers.

3.4 Network Forwarding

The very purpose of a module is to provide network access to the host. Therefore an important part of the module simulation is the ability to forward network traffic in realistic manner. To accomplish the network forwarding several alternatives was available. One alternative was to let the network traffic be routed between the network interface of the host to the network interface of the BeagleBoard-xM without any interference. This approach is as displayed in Figure 7.1 in Section 7.1. This alternative had the appeal of being easy to implement. A transfer of raw Ethernet data between the two interfaces would suffice to implement this solution. The second alternative was to implement some kind of IP routing on the BeagleBoard-xM that would ensure that datagram bound received on the BeagleBoard-xM network interface would reach the host.

The first alternative, a simple bridge between the two network interfaces, was deemed sufficient and was initially implemented. During the course of the project however new constraints became apparent and the implementation was revised. A detailed discussion about the change between the two network forwarding alternatives are described in Section 7.1.

Chapter 4

Implementation

The implementation chapter is intended to explain how the MBMSF system has been constructed and more specifically the architecture of each subsystem. An overview of the subsystems of which the MBMSF consists can be viewed in Figure 4.1. The chapter is divided into three sections signifying the three major subsystems developed to enable the MBMSF to successfully simulate a 3G module. The communication subsystem enables the MBMSF to enumerate as a real module would and creates several logical data channels. It thereby exposes the two other subsystem to the host, the simulator and the network forwarding subsystem.

The simulator handles control data communication with the host and essentially defines the behavior of the MBMSF. The network forwarding subsystem provides the host with an IP address and ensures that network communication from and to the host is routed correctly.

Some implementation details have been omitted from this chapter but are available in Appendix B. For the USB driver implementation in section 4.1, it is recommended that the reader has got some background knowledge in the Linux USB framework. This information can be acquired in section 2.3.

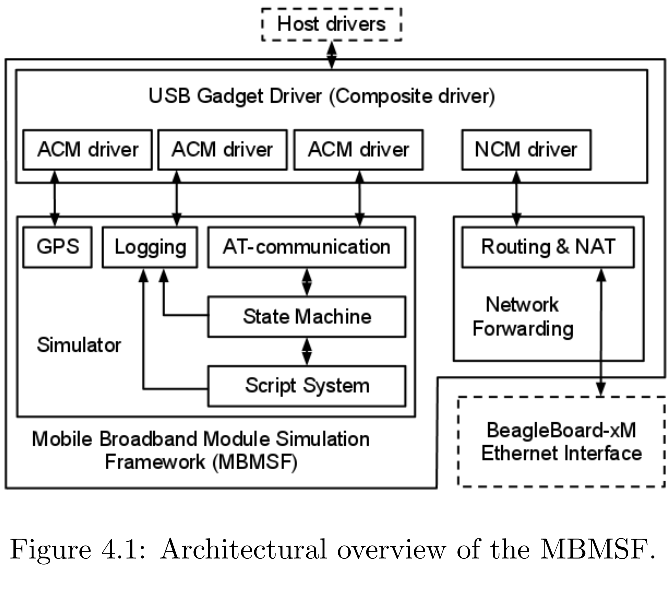


Figure 4.1: Architectural overview of the MBMSF.

4.1 Gadget Driver

The USB gadget driver implements the communication subsystem of the MBMSF so that it may communicate with the host it connects to. This subsystem's responsibilities is to identify the MBMSF device on the bus as the F5321gw module it simulates, and to support in establishing communication channels upon host and device interconnection, which is as part of the enumeration process. The channels will transport serial data constituting control data, Ethernet frames, GPS coordinates and debug output. These channels are the only interfaces available to the simulator- and networking-subsystem as can be seen in figure 4.1 above, meaning that all communication with the host is routed through these channels.

Using channels to communicate over the USB is an inherent necessity imposed by the USB standard specification. Moreover, the development efforts of implementing gadget drivers for peripheral devices depends on supporting frameworks and libraries provided by the operating system it will run in. In the case of the MBMSF, which runs a Linux operating system on top of a BeagleBoard-xM hardware chip, the supporting framework is the Linux Gadget API, (more closely described in chapter 2.4.3). The design of the framework means that the work of developing a gadget driver for the MBMSF system is fundamentally about describing the USB configuration, (including interfaces abstracting the channels), in a source code file written in the *C* language. The underlying framework takes care of the generic internal workings common to all gadget drivers needed to make a device successfully operate on the bus.

The rest of this section will describe the configuration and interfaces as implemented in the *g_mbm.c* source code. However, the writing of source code is by itself not enough to create the driver. A cross-compilation environment has been set up on a development system and has been used to cross-build the driver for the target architecture of the MBMSF. The output of this build is a Linux kernel module that needs to be deployed and installed in the running MBMSF system. These additional efforts are not specific for this project but common to most software projects in which the development and target system has different processor architectures. Therefore, these methods are described in appendix B.2 and recommended for the reader interested in the complete workflow of developing this gadget driver.

4.1.1 Source Code

The gadget drivers source code is written in C and based upon the reference driver *multi.c* [46] provided by the Linux Kernel Project. This driver was originally written by Michal Nazarewicz and is part of the Linux Kernel source tree since Linux version 2.6.33 [47].

The device simulated by the MBMSF is assigned an identity identical to the real module it is required to mimic according to the project specifications. This means it has been given a vendor id of *0x1917* and a product id of *0x0bdb*, matching the mobile broadband module *F5321gw* [4] developed by Ericsson AB. The driver incorporates a single configuration utilizing five interfaces as explained in the following section.

Configuration & Interfaces

One of the requirements on the MBMSF is that existing client drivers developed for hosts to communicate with the real module must be compatible with the simulated module. This further imposes requirements on the gadget driver in addition to correct identification strings. Figure 4.2 depicts the identity strings, as well as the interfaces and their positioning inside the configuration. The ACM interface number 8 is a so called *dummy* interface. It is of no other use to the gadget driver than to make it compatible with the Windows Vista client drivers from Ericsson. The order of the interfaces and the existence of a dummy interface is not an issue for client drivers on Linux hosts though, this is solely a solution for making the driver compatible with the Windows platform.

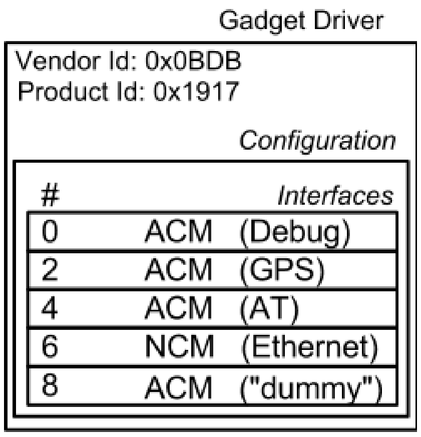


Figure 4.2: The gadget driver identity strings, configuration and interfaces. The unused interface is positioned at interface number eight.

The dummy interface is from now on excluded when discussing the MBMSF's interfaces. The following is a list of the interfaces incorporated into the driver along with their respective type and purpose:

- ACM - Debug channel (Output generated by simulator.)
- ACM - GPS channel (GPS coordinates encoded in NMEA sentences.)
- ACM - Control channel (AT-commands.)
- NCM - Network channel (Ethernet data.)

The hosts client drivers communicates with the simulator by sending and receiving AT-commands over the *control channel*. The *GPS channel* is used for transporting NMEA sentences (spatial coordinates), generated by the simulator for the host. The *debug channel* outputs debug data from the MBMSF; this data stream is intended to be used for debugging purposes during software development involving the simulator. If a task-specific filter is applied to this stream, it may show to be even more effective and useful. An example of a filtering tool is *SimDebug* provided in appendix D. The *network channel* is used for transferring network data between the host and the MBMSF.

4.2 Simulator

The simulator is the engine that controls the behavior of the MBMSF. This means listening to AT-commands sent from the host and reacting to these in some way.

The simulator is also responsible for a number of other tasks. These include the a GPS simulation and a logging system. Both of these tasks are described later in this chapter. However, the main and most important task of the simulator is to maintain AT-command communication with the host.

4.2.1 State Machine

Since a real mobile broadband module maintains a complex communication with the cell phone network, it is usually modeled as a state machine. The simulator also maintains a state machine that is used to decide which AT-command response is appropriate when receiving an AT-command from the host. AT-commands enter the simulator trough a tty terminal exposed by the USB composite driver described in section 4.1. Each command arrives as a series of characters and are buffered until a newline character is received. When a newline is registered the character buffer is forwarded to the AT-command parsing unit of the simulator. This unit will determine the appropriate action to take based on the received AT-command and the current state. What states are available in the MBMSF is entirely based on the content of configuration files loaded on system startup.

The simulator is entirely event-driven and in the current MBMSF implementation all events, and thereby subsequent actions that are taken to address the events, are originating from received AT-commands. In reality a series of actions are carried out whenever an AT-command is received in the simulator, in the future this series of actions are referred to as an AT-action. This since each AT-command are associated with one particular set of actions, an AT-action. The result of an AT-action execution may be a character string sent back through the USB composite driver, change of state or one of several other possibilities. For a more extensive explanation please refer to section B.1 in Appendix B.

The AT-action execution is divided into several discrete steps, which are:

- Load static AT-action based on configuration file.
- Possibly modify AT-action parameters based on script.
- Commit AT-action parameters.
- Send response string to tty port (if response have been specified).

In the *first* step of the AT-action execution a set of static parameters are loaded. These parameters are originally specified in a XML configuration file interpreted at simulation startup. But included in the simulator is also a possibility to attach ECMAScripts to an AT-action. The execution of a script is the *second* step in the AT-action execution. How the script-to-AT-action binding is implemented and used is described in greater detail in B.1 in Appendix B. A script attached to an AT-action enables the AT-command response to be more dynamic, since the ECMAScript is more flexible than the statical AT-action configuration based on the XML configuration. Each script attached to an AT-action is executed at run time and have, through an API, the ability to read, test against and change all possible AT-action parameters. This enables script writers to modify the behavior of the simulator without the need to alter the simulator source code.

Examples of parameters associated with an AT-action are:

- Response string to send to the host.
- State transition in the simulator State Machine.
- Time delay applied to the response sent back to the host.
- Assignment to variables maintained within the State Machine.

The *third* and *fourth* steps of an AT-action execution is simply to carry out the directives set in place by the first and second step of the execution. The *second* and *third* step needs to be separated because multiple ECMAScripts may be attached to a single AT-action and each must be executed before the final AT-action parameters are available.

GPS simulation

In addition to the handling of AT-commands the simulator also has a simple GPS simulator that allow it to simulate a GPS receiver. Some of the models of the Ericsson modules have a built in GPS receiver. Once the GPS receiver is activated in a real module, it locates at least three GPS satellites and continually outputs coordinates on a serial interface.

Several different operational modes exist in a real module. In the MBMSF the implementation is simple. A thread within the simulator is dedicated to reading coordinates from a prerecorded route file and output this to a dedicated serial interface with a static time interval between output.

To support more than simple verification of GPS output interface features on the host, this functionality would have to be more thoroughly integrated into the state machine of the simulator.

4.2.2 Logging & Debugging

Since the purpose of the MBMSF system is to aid in the debugging of host software client drivers, it is essential that the test engineer is informed of what goes on inside the MBMSF. Apart from the setup of the configuration files, this is achieved with the help of a logging system. The logging system implemented in the MBMSF is based upon the log4Qt open source library, which in turn borrows heavily from the well known Apache foundation logging implementation log4J for Java.

The MBMSF logging implementation is configured separately from the rest of the simulator to ensure that an MBMSF erroneous configuration setup does not interfere with the logging. If no logging configuration file exist a default one is created at system startup to prevent log message loss. The log system configuration allows several files to be created, each associated with its own log tag. Each log entry within the simulator has got a log tag attached and this can be used to filter what log messages should be routed into which file. The configuration of the log filtering is done by editing a text file and can be changed before each startup of the MBMSF system without the need to recompile the source code. The script system has access to a feature that lets scripts log messages. This feature allows introduction of new log statements without recompiling the source code.

The log files are locally stored in the MBMSF's filesystem. However since most use cases does not include access to the filesystem of the MBMSF, an additional serial interface has been set up in the usb gadget driver. By default this interface forwards the base log to the host. This log stream can then be filtered and stored on the host system in close to real time.

4.2.3 SimDebug Java Application

SimDebug is a small host side debug application written in Java. It was not written as part of the project requirements but to aid in the development of the MBMSF. The application description is provided in appendix D.

4.3 Network Forwarding

The purpose of the Network Forwarding Subsystem is to provide the host with the ability to communicate with a network as if it were connected to a real module. The final implementation of the network forwarding subsystem runs a DHCP server using a NAT configuration on the MBMSF. This allows the MBMSF to forward data between the two Ethernet interfaces it has access to. The first of the two interfaces is exposed by the NCM gadget driver which is used to establish a network interface between the module and host device. The other Ethernet interface is exposed by the physical Ethernet connector which is located on the BeagleBoard-xM. The external network interface, NIC 2 in Figure 4.3, is using a DHCP client to retrieve an external IP address, alternatively a static IP setup can easily be configured. The internal network interface, NIC 1 in Figure 4.3, runs a DHCP server that provides the host with an internal IP address from a local address range.

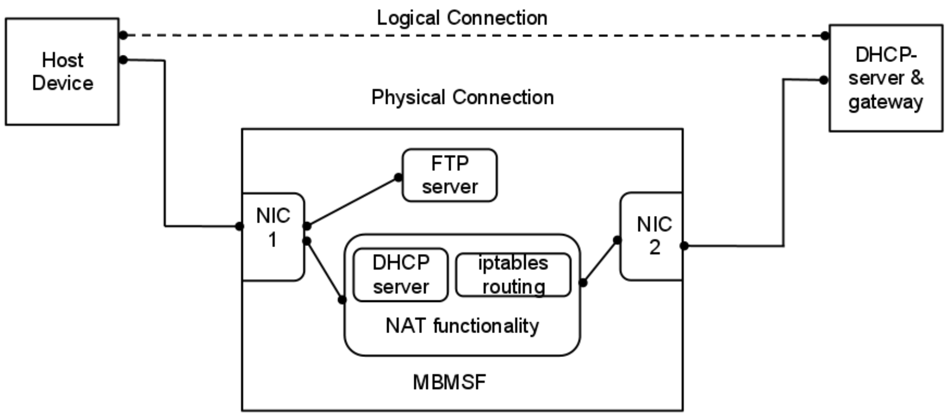


Figure 4.3: An overview of the network forwarding implementation.

Network Access to MBMSF

Apart from providing the host device with a way to forward data between an external and an internal network interface using NAT, this setup provides another benefit. It allows access to the BeagleBoard-xM itself through the use of dedicated ports. This direct network access is very useful for reconfiguration as well as file transferring. Accessing services on the MBMSF from the external network is done by addressing the IP and UDP/TCP port of a service routed directly to the MBMSF. The routing on the MBMSF is configured by an instance of iptables[10]. Currently only the TCP port 22(SSH) is reserved for the MBMSF itself, traffic to this port will be routed and consumed by the system on the BeagleBoard-xM, all other traffic is routed to the host.

Even though network access to the MBMSF from the external network interface is limited to the SSH port, no such restrictions apply for connections to the internal network interface where the host resides. The MBMSF is the gateway for the host and does therefore posses a unique, even though internal, IP address. This is useful since it theoretically allows the MBMSF to be used for a larger set of tests. The Ethernet connection can potentially be eliminated and only the USB link used when a service is run on the BeagleBoard-xM itself, for example a FTP server as seen in Figure 4.3. What effects this may have on tests are discussed in greater detail section 7.2.

Chapter 5

Evaluation

This chapter intends to explain how the MBMSF have been evaluated and what actions have been taken to ensure that the system is functional. To ensure that the MBMSF system works as intended a number of tests have been created. Most of these tests are performed on a system level and verify the correct behavior of system level functionality. Tests of smaller subsystems have been performed in a less structured manner and have therefore been excluded from the thesis report. A few tests have however been created to test non functional properties of the MBMSF. These tests have yielded some unexpected results regarding the network throughput; these results are discussed in section 7.2. The results, and a more precise explanation of each test can be found in appendix C while the general outcome of the tests are presented in chapter 6.

5.1 Functional tests

As explained above the main focus of the evaluation, and thereby the tests, have been to verify that the functional requirements of the MBMSF have been fulfilled. No pretense of full test coverage is made and only vital system functions are covered within the test scope.

The most important of the functional tests can be seen in table C.1, C.2 and C.3 in appendix C. These tests verify that the AT-command communication channel and script system work as expected. To support this statement, tests that run for long time durations have been created and during these tests the MBMSF system is put under more stress than it is estimated to endure during normal operations.

Another large section of functional tests that have been performed are the integration tests listed from test T.5 and beyond. These tests aim to prove that the MBMSF system is Operating System(OS) independent. This was important to prove since much of the reason for implementing the MBMSF on a separate piece of hardware was to make the system OS independent. These tests also show that the MBMSF work with software written for real Ericsson 3G modules.

5.2 Non functional Tests

The intention of these tests are not as with the functional tests, to establish that the system is fulfilling requirements, but to determine the limits of the usage of the MBMSF.

As seen in appendix C the non-functional tests are all built around the data forwarding subsystem of the MBMSF. The result of these tests implies a limitation of the usages of the MBMSF system; this fact and its implications are presented in Chapter 7.2.

Chapter 6

Result

The goal and subsequent requirements, specified in section A.1, have guided the project development and have resulted in a mobile broadband module simulator framework (MBMSF) running on a system-on-a-chip BeagleBoard-xM stand-alone hardware. In this chapter, the result of the thesis project is presented. Important features of the MBMSF system are enumerated and their role in fulfilling the project goal is explained.

The module simulation

The MBMSF implementation described in chapter 4 have been verified to be able to simulate a number of the operations that a real module performs. The MBMSF have also been shown to work well with networking software on different host operating systems. Software written specifically for Ericsson modules are compatible with the MBMSF. All of this indicates that the MBMSF system is likely to work well as a 3G module simulator. Equally important is the ability to get the MBMSF to act incorrectly but deterministically when debugging software drivers. To accomplish this the MBMSF have been made highly configurable.

Flexible configuration

To provide as much flexibility as possible a number of configuration files have been created. All these files use plain ASCII character encoding and have a standardized formatting. Through the use of the configuration files it is possible to alter the response that the MBMSF may give to any AT-command. The configuration possibilities are further extended by the introduction of a script engine which provides the possibility to create dynamical responses to AT-command input. Conditional script statements may be used to create sophisticated failure models which can be used when debugging host client drivers. The scripts may also be used as assertions to verify that the AT-command input from the host drivers are consistent with any number of conditions. For a more thorough description of the configuration files and script system please refer to section B.1.

Logging

To enable tracking of the MBMSF execution a highly configurable logging system is implemented. It allows the log output configuration to be altered and does in combination with the script system allow introduction of new log statements without the need to recompile the MBMSF source code. Messages may be filtered by log level or message priority and then routed to a file or terminal within the MBMSF system. By default a copy of the raw unfiltered log is sent to a tty terminal connecting the MBMSF to the host. This allows the user to review and filter the log from a running MBMSF instance in close to real time without the need to mount the MBMSF file system.

Operating System independence

Because the MBMSF is implemented on a physically separate device, the system is host operating system independent and the MBMSF could theoretically be used to debug client driver on any operating system. Test have verified that the MBMSF can be used for driver development on Windows Vista and Linux hosts.

Data transmission simulation

The MBMSF system does not only simulate the serial communication between the host and module but also offers the possibility to transfer data over a network interface which the MBMSF composite driver expose in the host. This allows the network interface subsystem of the host to be tested along with the AT-command communication. This data forwarding capability is implemented in a way which allows it to be isolated from the rest of the network. For a discussion regarding this feature please refer to section 7.1. The isolation of the USB link could for instance be used if properties of the link were to be tested without interference from the rest of the network. Tests have shown that this data link is sufficient to test data throughput levels used in current 3G networks, about 21Mbit/s. To allow use cases with data transfer rates sufficiently high to simulate LTE level data throughput would require software modification of the MBMSF, if at all possible with the currently used MBMSF hardware. A discussion on this topic is available in chapter 7.

Chapter 7

Discussion and Future Development

The intention of this chapter is partly to discuss problems that arose during the course of the project and partly to give advice for further development of the MBMSF system. A more more exhaustive discussion regarding the future development sections is provided in Section 7.3.

7.1 Two methods of network forwarding

During the course of the thesis project two different implementations of the network forwarding subsystem were created. The first version made use of a so called Ethernet Bridge. The Ethernet Bridge software creates a virtual link between two network interfaces on a machine. The data sent to one of the interface was forwarded to the other. The forwarding was done on the data link layer which meant that entire Ethernet frames were forwarded without ever being opened. This had the effect that no routing was possible in the MBMSF. Moreover the MBMSF, for all practical purposes, vanished from the network architecture once the bridge became active. A DHCP request message sent from the host was indiscriminately forwarded to the other network interface. This network forwarding setup worked well during active use of the simulator. However, problems occurred as soon as something needed to be reconfigured on the MBMSF. Either a serial link needed to be used to access the MBMSF system, or the Ethernet bridge needed to be switched off to allow the MBMSF to be addressed and accessed through SSH. Solutions mitigating this problem was investigated and in one instance even implemented. A script was created binding the network forwarding process to one of the two physical buttons available on the BeagleBoard-xM, pressing the button toggled the network forwarding on the BeagleBoard-xM.

NAT solution

The first iteration of the network forwarding implementation did also have other drawbacks. The main one being that the data throughput in the MBMSF system would be limited by the throughput available on the BeagleBoard-xM physical

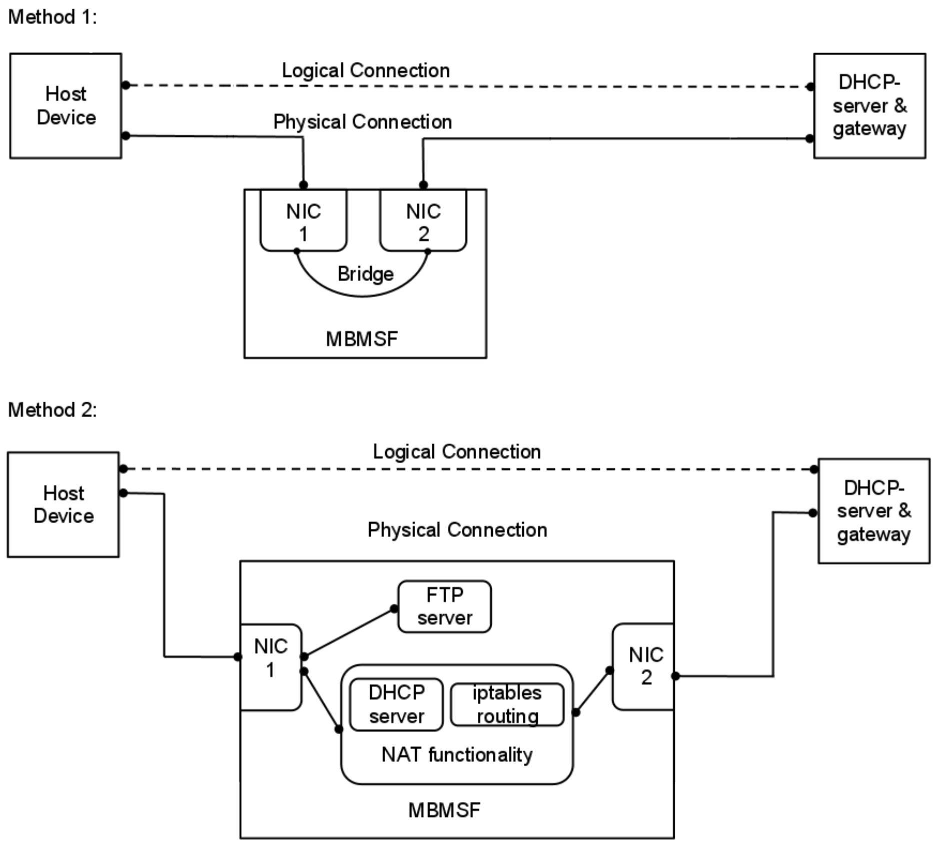


Figure 7.1: An overview of the two methods for achieving network forwarding.

Ethernet interface. In many use cases the Ethernet throughput available on the BeagleBoard-xM Ethernet interface could be as low as 100Mbit/s due to limitations in the network architecture. This throughput is lower than what is planned for the LTE mobile communication system. To be able to test throughput related issues in drivers developed for LTE modules the MBMSF would have to be able to support the throughput of such systems. Using a 2x2 multiple-input and multiple-output(MIMO) configuration LTE should theoretically reach throughput speeds of up to 73.40 Mbit/s [44]. Using a 4x4 MIMO configuration, likely to be used in the future, even higher throughput will be achieved [44]. To simulate a 4x4 MIMO LTE configuration the potential bottleneck that the BeagleBoard-xM Ethernet interface presented a new network forwarding method needed to be implemented.

A hint of a solution to the problems with the first network forwarding implementation was given in the implementation of a real Ericsson module. In a real Ericsson module a DHCP server setup is used to provide the host with an IP for the network interface exposed by the module. The information regarding the real module network setup lead to the second implementation of the network forwarding subsystem described in section 4.3. Using a DHCP server with a NAT configuration in the MBMSF enables the host to directly address the MBMSF with IP traffic. This does not only mean a permanently available ssh server for MBMSF configuration but also allows data traffic to be sent to the MBMSF system itself. This also removed the potential throughput bottleneck of the physical Ethernet interface by letting network services be run on the MBMSF system. In Figure 7.1 this is exemplified with an FTP server. The tests of the second implementation explained in Chapter 5 have been performed by sending data throughput over the USB link isolated from the rest of the network.

7.2 USB data throughput limit

In the ideal case the data throughput on the USB link should be close to the theoretical limit of USB 2.0, about 480Mbit/s, enough to support LTE level throughput testing. However as the test results in figure 7.2 show the actual measured throughput only achieve a fraction of the theoretical USB limit.

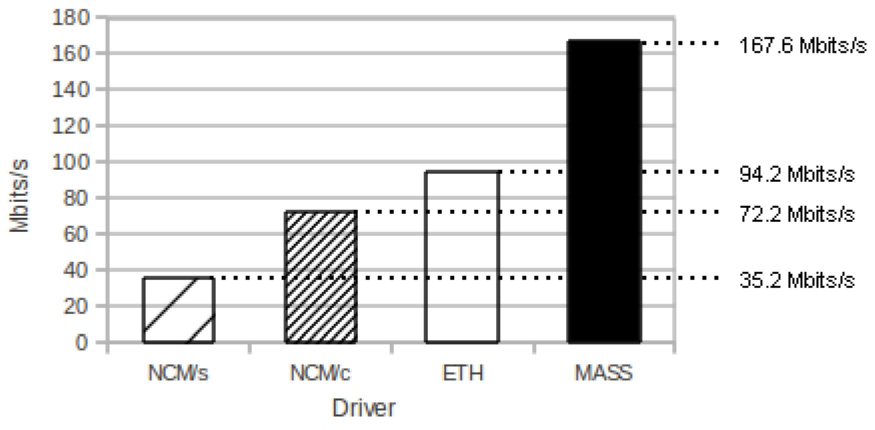


Figure 7.2: Average throughput measurement using different drivers. NCM/x is the results using the NCM channel of the gadget driver where x denotes direction; s = receiving, c = transmitting. ETH and MASS is tested using Linux Ethernet driver and gadget mass-storage driver, respectively.

During the first examination of the network forwarding throughput the first version of the network forwarding implementation was used, described in section 7.1. This implementation left no room to separate the USB link throughput from that of the Ethernet interface on the BeagleBoard-xM. The throughput bottleneck could therefor well be within the Ethernet link to the rest of the network as seen in figure 7.3 or in the Ethernet Bridge forwarding method. Partly to amend this the second version of the network forwarding implementation was created. The second implementation allows the throughput of the USB link to be isolated and tested separately by letting either the host or module act as a server during a transfer. As shown in the test results presented in figure 7.2 the throughput did even after the isolation of the USB link not reach anywhere near the theoretical limits specified by the USB standard. Initially the reason for this was assumed to be the incomplete

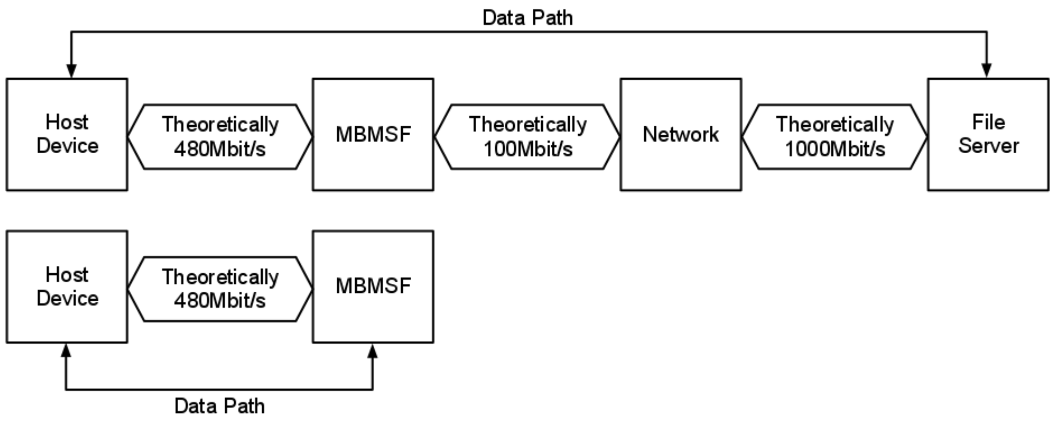


Figure 7.3: An overview off the bottleneck in the network forwarding subsystem.

NCM gadget driver implementation used in the MBMSF. The driver implementation used in the MBMSF for the data transmissions over the USB link is the NCM gadget driver available in the Linux kernel. The NCM gadget driver implements the minimal set of protocol functions to support the NCM protocol. One of the limitations that this have introduced concerns the frame aggregation feature of NCM. The NCM protocol is an addition to the underlying ECM protocol used for transferring network data over a USB link. The ECM protocol induces a high number of interrupt on the receiving end once the throughput of the data transmission increases since only one Ethernet package can be transmitted in each USB transfer. Among other effects, this can reduce the maximal throughput possible over a link using ECM. The incomplete NCM gadget driver implementation used in the MBMSF does, just as the ECM protocol, not aggregate Ethernet packages. For this reason it is suspected that the relatively low data throughput experienced over the data link is caused by the lack of frame aggregation.

By examining the data throughput achieved by other unrelated USB bulk transfer protocols it was verified that the general throughput limit over the USB link is not limited to the throughput exhibited by the interface exposed by the NCM driver. A possible solution to the limitation induced by the usage of the gadget NCM driver is to implement the changes proposed in section 7.3.4. Doing so would inevitably mean exchanging the NCM driver for a driver that supports the Mobile Broadband Interface Model (MBIM) protocol [34].

Even though the throughput limitation discovered during the tests prevents the MBMSF to be used as a simulator for LTE modules it is still more than enough to reach the throughput speed achieved by the fastest 3G modules available at the time of writing, 21Mbit/s.

7.3 Future Development

Even though tests in appendix C shows that the system is operational and fulfill the requirements imposed by Ericsson many possibilities for enhancement exist. Part of the requirement of the thesis project was to create a system which could easily be extended and maintained. Providing a detailed description of possible future development of the MBMSF system is a way to argue for such properties. Other reasons for the future development section does also exist e.g. the section 7.3.4 regarding the MBIM protocol adaptation was specifically requested by Ericsson to ensure that a plan for the MBIM implementation existed as the actual implementation could not be fitted into the scope of the MBMSF project.

7.3.1 Event system

Currently the MBMSF lacks a complete event system. The only event to which callbacks such as scripts may be applied to currently is on reception of AT-commands. It would be beneficial to be able to bind callbacks, such as scripts, to other events in the system. It would for example be very useful to bind a script to the event of a variable modification. This would greatly ease emulation of unsolicited messages. Attaching callbacks to state transitions would likewise be useful.

To implement this event system a general callback hook in each of the possible events would need to be created. This could be implemented as an additional function call whenever a variable modification or a state transition occurred. An example of an event callback execution is displayed in figure 7.4.

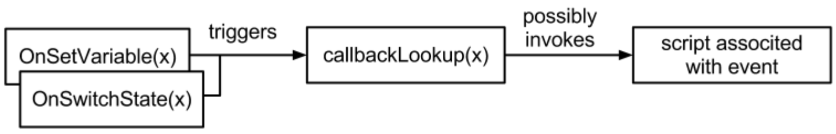


Figure 7.4: Proposed event triggering. In the second function call, the callbackLookup, the data structure displayed in the next figure would be used.

The "callbackLookup" function call would iterate, or use a hash indexing technique, to find a script that is associated with a particular instance of the event. To maintain the callback function list a data structure containing all the registered events in the system would need to be created. A figure of the proposed data structure for maintaining event callback methods are displayed in figure 7.5.

Since the data structure resembles the structure used to hold the AT-actions described in section 4.2.1 it may even prove beneficial to extend the same data structure.

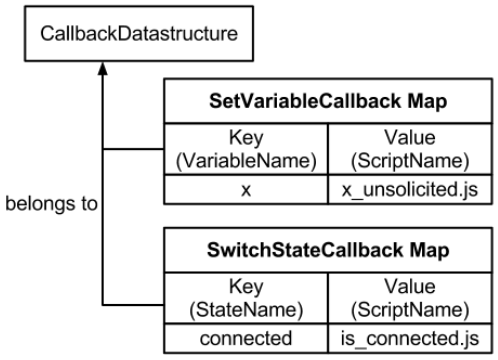


Figure 7.5: Example of data structure maintaining the event callback information.

Only events with actual callbacks registered would need to be maintained in the event data structure. The actual registration of events in the data structure could be done at initialization through an XML file similar to the state machine definition file and/or by allowing runtime registration of event callbacks.

7.3.2 Traffic shaping

Presently the bandwidth available to the host is always the maximal throughput which the MBMSF system can provide. No bandwidth limitation techniques have been applied on the current version of the network forwarding implementation. The previous iteration of the network forwarding implementation used a traffic shaping filter that allowed the maximal throughput to be modified at runtime. This feature potentially allowed the MBMSF to simulate bad network conditions and even packet loss.

As is described in section 7.1 the first network forwarding implementation was abandoned due to its various drawbacks. The drawbacks was however not related to the traffic shaping functionality and it is possible to implement in the current MBMSF system. The traffic shaping in the first network forwarding iteration was done using the Linux Class Based Queuing(CBQ) qdisk program. The program queues datagrams bound for transmission on a network interface and transmits them in a timed interval which will produce a data throughput that corresponds to a value specified at program start. The interval between transmissions is dynamically calculated based on the the total throughput available on the interface.

By letting the MBMSF create and modify the a CBQ qdisk filter applied on the outbound network interface during runtime the MBMSF would be able to simulate changes to the data link. This could be done by letting the MBMSF spawn and execute separate processes that applied or modified a CBQ filter. The facilities for spawning and executing such processes from within the MBMSF are implemented but poorly tested.

7.3.3 Network AT-command Control

In the current implementation the link between the simulator and the network forwarding subsystem is missing. This means that the state of the network connection simulated on the MBMSF is not reflected on the actual network interface which the MBMSF provides to the host. The MBMSF may respond to AT-commands as if the connected state of the module is off line, but in reality the network interface that MBMSF provides will accept any network traffic sent to it. In other words, the network link is not severed when the state machine in the simulator enters an off line state. The modifications required to accomplish a link between the MBMSF state machine and the network forwarding subsystem should be relatively small since the simulator currently has the ability to execute arbitrary bash commands on the MBMSF system. Altering properties of the data forwarding setup would suffice to sever or restore the connection. However if this is enough to successfully simulate the correct disconnected behavior of the network interface provided to the host needs to be investigated further.

7.3.4 Adaptation to Mobile Broadband Interface Model

The main functionality of the MBMSF is built around parsing AT-command messages and responding to these. The AT-command standard was specified the first time during the mid 1980 to define communication between the modems and computers of that day. Because of this the communication is not perfectly suited for the communication between modern modules and hosts. As a response to this a new standard for module to host communication is emerging called Mobile Broadband Interface Model(MBMIM) [34]. This new standard is based on an entirely new communication protocol and completely replaces the AT-communication standard.

MBIM Specification Introduction

AT-command communication uses a separate USB interface, the CDC ACM serial interface (Section 2.4.4), for control data communication and second USB interface for data transmission, CDC NCM (Section 2.4.4). Unlike the communication performed with the AT-command protocol the MBIM control communication is transmitted on the default pipe in the device layer of the USB protocol stack. This usage of the default pipe is made possible by utilizing the interrupt channel on the MBIM interface. This means that no separate serial drivers and pipes are needed to create a control channel over which instructions can be sent to the module. Figure 7.6 compares the approaches to control data transmission.

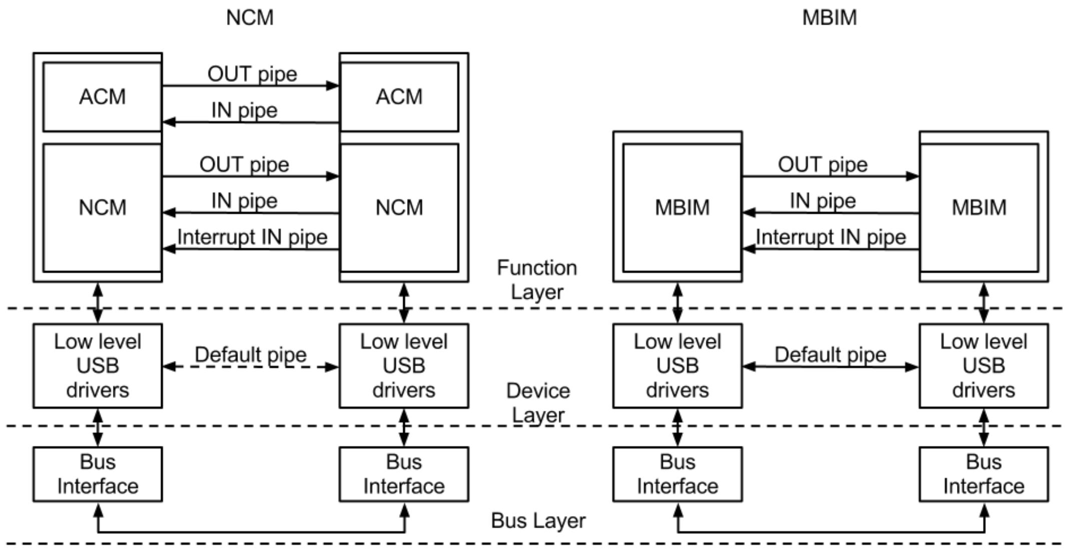


Figure 7.6: The architectural difference between NCM and MBIM. MBIM have only one interface used for both data and control message transfers.

As previously mentioned the control message transmission uses the default pipe and since a single pipe can not handle duplex communication the MBIM interrupt pipe is used to signal upcoming data transmission. By using the interrupt pipe the default pipe is able to transmit data in both directions, but never at the same time. A sequence diagram of a control message transaction in the MBIM protocol is displayed in figure 7.7.

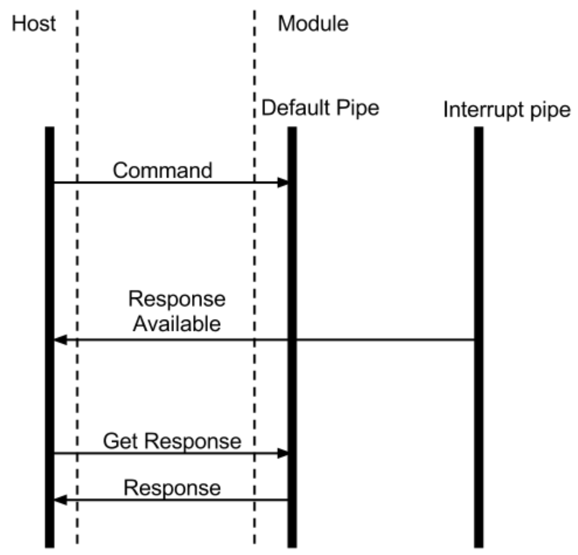


Figure 7.7: Sequence diagram of a MBIM control message transaction. Note that the "Response Available" message is sent asynchronously and therefore does not require the host to wait for the response. An unsolicited message only uses the last three messages.

A special packet is transmitted over the interrupt pipe, the receiving device responds and a control packet data transmission over the default pipe can start[34]. Since a control data transmission is always preceded by an interrupt neither device needs to poll the incoming data buffer as was required in the AT-command protocol. Another big difference between the MBIM protocol and using AT-commands are that the transmitted control data is not sent as ASCII characters. The control data is transmitted in binary form and marshalled to C structs on the receiving end.

Changes Required to Support MBIM

Rather drastic changes to the MBMSF are required to be able to support the MBIM protocol. The most obvious, and likely the hardest to implement, is the requirement of drivers supporting the MBIM protocol. Currently no such drivers have been implemented and added to the Linux kernel. And more importantly no MBIM gadget(peripheral) drivers exist. The host side MBIM drivers are bound to be implemented by the Linux community in the near future to provide support for new hardware. However the support for the corresponding gadget drivers may take time. Developing MBIM driver will be a rather complex task, possibly suitable for a master thesis.

Apart from the requirements of a new gadget driver supporting MBIM a few other changes would also be necessary to the MBMSF. The changes are required to support the altered message format which the MBIM change brings. Since the current MBMSF implementation is based on AT-commands, and thereby ASCII pattern matching, a new command matching process needs to be developed. Instead of matching incoming commands against AT-command strings listed in the state machine definition file the incoming commands will be matched against a binary data structure. Each such structure defined in the MBIM have a unique id called SID[34]. This id would be used to match the incoming command against an action in the MBMSF. How the responses and actions to incoming commands are defined would also need to be revised. The current method of defining these actions in the state machine definition XML file would need to be reconsidered. Either an entirely new way of entering these action would need to be created, possibly by developing a tool which enabled the actions to be stored directly as a binary data structure. Or by creating a tool which converted information stored in the state machine definition XML file to data matching the data structures that the MBMSF were to receive. In either case the method for defining command response actions would likely be more complicated than in the current implementation that uses AT-commands.

7.3.5 Command Line Interface

It have been suggested that a a command line interface(CLI) towards the MBMSF would be convenient feature. Such an interface would expose variables, scripts and states to the end user. The user could through the CLI change these by entering ASCII strings, just as in a normal UNIX-terminal. This would be useful in circumstances where the exact conditions of a fault were unknown and slight tuning of the simulator may be required to find just the right setting to trigger a fault in the host drivers. The implementation of such an interface would require some restructuring of the internal system, however the most straight forward way would be to implement the CLI on top of the already existing AT-command parser. CLI commands would be passed as a parameter to a specially designed AT-command. This AT-command would deliver the CLI-command to a parser which would carry out the instructions that a command symbolizes. The commands could perhaps even be interpreted as ECMAScripts, which would allow the script engine in MBMSF to be reused. Using this method would allow CLI-commands to be issued over the normal serial communication channel exposed with the ACM driver. No separate serial interface to the module would thereby need to be implemented.

Chapter 8

Conclusion

This thesis has produced a Mobile Broadband Module Simulation Framework (MBMSF) that runs on a separate piece of hardware connected to a host device through a USB cable. The MBMSF simulates a mobile broadband module to facilitate development of module driver software. The implementation is host platform independent and can be used with any operating system intended to support a module.

Testing of the complete system indicates that the MBMSF is able to handle communication with the host system and communicate with software written for interaction with real modules. Furthermore, the system supports a high level of configurability through the use of an internal script engine that allows for conditional alterations in the module state machine behavior. The high level of configurability could potentially be used to test edge cases, stress test driver software or create assertion tests. All of the activity within the MBMSF is logged and is by default stored locally and also send over the USB link to the host for consumption. This ensures that the developer is never in doubt of what is communicated between the drivers and the MBMSF.

Moreover, the MBMSF is capable of simulating data communication between the host and the module. Network traffic sent to the MBMSF can be transparently re-routed to an external network to simulate a normal cell phone network connection or handled locally on the MBMSF hardware to prevent external interference with the data throughput.

System tests have showed that the maximal data throughput speed of the MBMSF easily allows for 3G module data throughput levels but does not reach LTE data throughput levels. This has lead to the conclusion that the MBMSF implementation described in this paper is suitable for 3G modules simulations but will likely not be sufficient to simulate LTE modules.

Beyond the implementation of the MBMSF an examination of future extensions to the system has been conducted. The examination provides descriptions and advice regarding the implementation of features that would further extend the capabilities of the MBMSF.

References

[1] Bitbake build tool. http://developer.berlios.de/projects/bitbake. Accessed Feb 16, 2012.

[2] cmake. http://www.cmake.org/. Accessed Dec 12, 2011.

[3] Device simulation framework. http://msdn.microsoft.com/en-us/windows/hardware/gg454516.

[4] Ericsson f5321gw mobile broadband module. http://www.ericsson.com/solutions/mobile_broadband_modules/docs/9_287_01_fgb_101_576-h5321gw_r1a-screen.pdf. Accessed Feb 27, 2012.

[5] Gnu arm toolchain. http://www.gnuarm.com/. Accessed Nov 9, 2011.

[6] Gnu make. http://www.gnu.org/s/make/. Accessed Dec 13, 2011.

[7] hdparm man page. http://linux.die.net/man/8/hdparm. Accessed Feb 27, 2012.

[8] Hp compaq 110c-1011so. http://h10025.www1.hp.com/ewfrf/wc/document?docname=c01815721&cc=ad&dlc=en&lc=en&jumpid=reg_r1002_usen. Accessed Feb 27, 2012.

[9] iperf man page. http://manpages.ubuntu.com/manpages/lucid/man1/iperf.1.html. Accessed Feb 27, 2012.

[10] Iptables. http://www.netfilter.org/projects/iptables/index.html. Accessed Feb 29, 2012.

[11] Linux-usb gadget api framework. http://www.linux-usb.org/gadget/. Accessed Dec 12, 2011.

[12] The linux usb sub-system. http://www.linux-usb.org/USB-guide/book1.html. Accessed Mar 23, 2012.

[13] Ofono. http://http://ofono.org/about. Accessed Feb 8, 2012.

[14] Openembedded build tool. http://www.openembedded.org/. Accessed Nov 9, 2011.

[15] Port of android for beagleboard-xm. http://beagleboard.org/project/Beagledroid/. Accessed Mar 23, 2012.

[16] Port of angstrom for beagleboard-xm. http://beagleboard.org/project/angstrom/. Accessed Mar 23, 2012.

[17] Port of ubuntu for beagleboard-xm. http://beagleboard.org/project/Ubuntu/. Accessed Mar 23, 2012.

[18] Port of windows embedded for beagleboard-xm. http://beagleboard.org/project/WEBB/. Accessed Mar 23, 2012.

[19] qmake. http://doc.qt.nokia.com/latest/qmake-manual.html. Accessed Dec 12, 2011.

[20] Qt. http://qt.nokia.com/. Accessed Dec 12, 2011.

[21] Usb driver stack architecture. http://msdn.microsoft.com/en-us/library/windows/hardware/hh406256%28v=vs.85%29.aspx. Accessed Mar 23, 2012.

[22] Usb implementers forum. http://www.usb.org/. Accessed Dec 6, 2011.

[23] Universal serial bus class definitions for communication devices. http://www.usb.org/developers/devclass_docs/CDC1.2_WMC1.1_012011.zip, note = Accessed Nov 22, 2011, November 2010.

[24] R. Casley C. Chien A. Jain M. Lipsie D. Tarrodaychik O. Yamamoto A. Ghosh, M. Bershteyn. A hardware-software co-simulator for embedded system design and debugging. Technical report, Mitsubishi Electric Research Laboratories, Inc., 1995.

[25] Ericsson AB. *Mobile Broadband Module Screen*, May 2010.

[26] Gerald Coley. *BeagleBoard System Reference Manual, Revision C4*, December 2009.

[27] Gerald Coley. *BeagleBoard-xM System Reference Manual, Revision C.1.0*, April 2010.

[28] Linux Community. Building external modules, (source tree documentation, linux kernel ver. 2.6.39). http://git.kernel.org/?p=linux/kernel/git/stable/linux-stable.git;a=blob_plain;f=Documentation/kbuild/modules.txt;hb=HEAD. Accessed Jan 19, 2012.

[29] Linux Community. Linux kernel makefiles, (source tree documentation, linux kernel ver. 2.6.39). http://git.kernel.org/?p=linux/kernel/git/stable/linux-stable.git;a=blob_plain;f=Documentation/kbuild/makefiles.txt;hb=HEAD. Accessed Jan 19, 2012.

[30] Linux Community. module-init-tools readme-file, (source tree documentation, linux kernel ver. 2.6.39). http://git.kernel.org/?p=utils/kernel/module-init-tools/module-init-tools.git;a=blob;f=README;h=5c314576ead39cfb91d8fcee4c27756e1b1bae6f;hb=HEAD. Accessed Jan 25, 2012.

[31] Linux Community. Source tree, linux kernel version 2.6.39.4. `http://git.kernel.org/?p=linux/kernel/git/stable/linux-stable.git;a=summary`. Accessed Jan 19, 2012.

[32] Al Borchers David Brownell. serial.c (serial gadget driver in linux). `http://git.kernel.org/?p=linux/kernel/git/stable/linux-stable.git;a=blob;f=drivers/usb/gadget/serial.c;h=ad9e5b2df64267d9b18af3b770351a04069dd0e1;hb=HEAD`, 2008. Accessed Feb 28, 2012.

[33] Balden et. al. Universal serial bus communications class subclass specifications for network control model devices. `http://e-tools.info/project/documents/18001_19000/18516/ncm10.pdf`, note = Accessed Nov 16, 2011, April 2009.

[34] Björn Boden et. al. Universal serial bus communications class subclass specification for mobile broadband interface model. Technical report, USB-IF, 2011.

[35] Stallman et. al. Gnu make. `http://www.gnu.org/software/make/`. Accessed Jan 20, 2012.

[36] ETSI. Digital cellular telecommunications system (phase 2+);specification of the subscriber identity module - mobile equipment (sim - me) interface. Technical report, ETSI, 1995.

[37] ETSI. Digital cellular telecommunications system (phase 2+); universal mobile telecommunications system (umts); lte; at command set for user equipment (ue) (3gpp ts 27.007 version 10.6.0 release 10). Technical report, ETSI, 2012.

[38] USB Implementers Forum. *Universal Serial Bus Specification*, 2 edition, April 2000.

[39] USB Implementers Forum. *Universal Serial Bus 3.0 Specification*, 3 edition, May 2011.

[40] ECMA International. Ecmascript language specification, 5.1 edition. Technical report, ECMA International, 2011.

[41] National Intruments. *PCI Express - An Overview of the PCI Express Standard*, Aug 2009. Found at: http://zone.ni.com/devzone/cda/tut/p/id/3767; Accessed: 2012-02-08.

[42] ITU. Detailed specifications of the terrestrial radio interfaces of international mobile telecommunications-2000(imt-2000). Technical report, ITU, 06/2010.

[43] Mikko Kerttula. *Virtual Design, A Framework for the Development of Personal Electronic Products.* PhD thesis, Faculty of Technology, University of Oulu, 2006.

[44] Tommy Svensson Marilynn P. Wylie-Green. Throughput, capacity, handover and latency performance in a 3gpp lte fdd field trial. 2010.

[45] mec. kbuild: the linux kernel build project. http://kbuild.sourceforge.net/. Accessed Jan 19, 2012.

[46] Michal Nazarewicz. multi.c (multifunction composite driver in linux). http://kernelnewbies.org/Linux_2_6_33#head-e43ebdd2327e592842e46603dd3df49aa7394cb1. Accessed Feb 22, 2012.

[47] Linux Kernel Newbies. Changelog linux version 2.6.33. http://kernelnewbies.org/Linux_2_6_33#head-e43ebdd2327e592842e46603dd3df49aa7394cb1. Accessed Feb 22, 2012.

[48] ETSI organization. Universal telecommunication system; requirements for the umts terrestrial radio access system (utra). Technical report, 3GPP, 1997-10.

[49] Rusty Russel. depmod man page. http://linux.die.net/man/8/depmod, 2002. Accessed Jan 25, 2012.

[50] Rusty Russel. modprobe man page. http://linux.die.net/man/8/modprobe, 2002. Accessed Feb 13, 2012.

Appendices

Appendix A

Requirements

A.1 Requirements, Limitations and Their Results

When the project was devised this short description was used to describe the intended outcome of the thesis work:

Create a mobile broadband module (from now on "module") simulation framework. The framework will be used to support device driver development before hardware is available and to simulate module failures during driver testing.

In addition to the description several requirements where imposed on the project to ensure that the desired outcome was achieved. Both the thesis description and the requirements was originally provided by the Mobile Broadband Module department of Ericsson. Some additions to the original requirements have during the course of the project been appended to further extend and specify the MBMSF behavior, these changes are reflected in the list below. The thesis requirements include a functional specifications that described the intended behavior of the simulator.

- **It should be possible to configure the simulated module to different module configurations. It must be possible to control the simulated module through XML or URC files.**
- **The simulated module must be able to read a definition file which states which AT-commands are supported.**
- **It should be possible to configure the simulated module to simulate module malfunctions, failures and fail rates in a set of XML files.**
- **The framework should be documented and structured in a way which enables further extension.**
- **Test cases simulating real-life scenarios should be created to verify that the framework is functional and provide guidance for how the framework is to be used.**

Coupled with the requirements of the project several limitations was created to further specify that requirements of the project:

- **The device framework will not simulate any interface other than the interface between module and host drivers.**
- **No efforts will be made to copy the existing internal structure of real modules.**
- **Only a subset of the existing device and host system interface will be implemented. Minimally those interface parts that are required to fulfill the specified test sessions.**

Appendix B

Detailed Implementation & Deployment

This appendix provides a more detailed description of two of the simulators subsystems. It starts with the architecture of the simulator and ends with the implementation and deployment of the gadget driver.

B.1 Simulator

As mentioned in section 4.2 the simulator is the engine which controls the behavior of the MBMSF. This appendix section will in detail explain how the simulator has been constructed. Each functionally separate part of the simulator will be explained separately to provide a deeper understanding of the MBMSF implementation.

B.1.1 State Machine Model

Just like in a real mobile broadband module, the simulated module maintains an internal state machine. However, its state machine is heavily simplified. This simplification can be done without any loss of simulation accuracy. This is due to the fact that one of the main goals of placing the cell phone network communication in a separate module is to present the host device with a simplified and standardized network interface. In a real setting the host device is therefore unaware of most of the complexities associated with the maintenance of the cell phone network data link. The state machine in the simulator is constructed from a XML definition file which is interpreted upon simulation startup. The state machine definition file is used to generate a tree-like data structure which the simulator searches through when an AT-command is received from the host. When a matching AT-command entry is found in the tree structure the associated actions are executed and the AT-response found in the entry is returned to the host. An example of the state machine configuration XML can be viewed in Listing B.1. As seen in Listing B.1, AT commands responses are also specified in the same file. Each state is associated with several AT-commands and the appropriate AT-command response is in this way determined by which state the simulator currently resides in. Two states may have different AT-command responses for the same AT-command. For example the

"AT+COMMAND1" command which is found in "state1" on line 6 in listing B.1 is also defined in "state2" on line 18 in the same listing. This command have different responses depending on in which state the state machine resides in.

```
 1 <?xml version="1.0"?>
 2 <simulator>
 3     <set name="variable1" value="100"/>
 4     <state name="state1">
 5         <chat>
 6             <command>AT+COMMAND1</command>
 7             <response delay="500">${variable1}\n\nOK</response>
 8             <set name="variable1" value="200"/>
 9         </chat>
10         <chat>
11             <command>AT+CPIN=*</command>
12             <response>+CPIN: READY\n\nOK</response>
13             <switch name="state2"/>
14         </chat>
15     </state>
16     <state name="state2">
17         <chat>
18             <command>AT+COMMAND1</command>
19             <response delay="1000">Active\n\nOK</response>
20             <set name="variable1" value="300"/>
21         </chat>
22     </state>
23 </simulator>
```

Listing B.1: State machine definition file example.

Transitions between states is also defined in the state configuration XML. This is seen at line 13 in Listing B.1. Issuing the command AT+CPIN=anything in state state1 will trigger a state transition in the simulator and the AT command responses will be altered accordingly. One other property of the state configuration XML needs to be mentioned. The definition of global variables seen at line 3 in Listing B.1. This allows global variables to be registered in the simulator. These variables are used in AT-command responses, as seen at line 7 in Listing B.1. This allows the MBMSF to respond with dynamic values. These variables may also be altered by receiving particular AT-commands. The definition of an an AT-command which alters a variable can be seen at line 8 in Listing B.1. All of this functionality is augmented by the possibility of attaching scripts to an AT-command response. How this is done is described in the next section.

B.1.2 Simulation Scripts

Beyond the functionality offered by the state machine definition file, a script system have been created to increase the system flexibility. These scripts are read by the simulator at startup and attached to the tree-like data structure of the AT-command responses that is generated from the state machine definition file. To understand how these scripts are able to modify variables, state transitions, AT command responses and response delays it is important to understand the processing of AT-commands in the simulator.

When the simulator receives an AT-command a process of parsing the string data will start. The AT-command needs to be matched against an existing AT-command entry in the internal AT-command tree structure. This is done by iterating over the AT-command entries in the current simulation state. The actual matching between a received AT-command and the appropriate AT command response entry is done by executing a regular expression match operation on the received AT-command. If a match is found the process of executing the AT command response will start. If no match is found in the current state the simulator will resort to searching the default state for an appropriate AT-command response. If a match is found the AT-command response execution will start, but if no match is found in the default state a response value of "ERROR" is returned to the host device to signal that the AT-command is not supported.

The execution of an AT command response have four distinct steps, as depicted in Figure B.1. The *first step* is the loading of parameters associated with the AT-command response. These parameters are specified in the state machine definition file described in previous section. The *second step* of the AT-command response execution is the execution of associated scripts. How these scripts are associated with an AT-command response is explained in the next section. When a script is run, get and set methods for variables and access to several functions are made available to the script by exposing C++ functions to the script engine. The scripts are through the invocation of these C++ functions allowed to read and modify the parameters that can be associated with an AT-command response. The *third step* of the AT-command response execution is to carry out the instructions associated with the AT command response. These instructions may have been modified by the scripts in the previous execution step. Performing these operations may involve modifying global variables and spawning additional threads to execute external programs. The *fourth and final step* of the AT-command response execution is the transmission of the response string back to the serial interface. This response string can be modified by a script and a function call from a script may also introduce a delay before the AT-command is sent to the serial interface.

Each script also returns a boolean value upon termination. This value is used if the scripts is marked as an assertion test in the script bindings file. The script binding file is described in the next section. Because the parameters specified in the state machine definition file is loaded each time an AT-command response execution occurs, the values stored in the local variable scope is always the same when script execution starts. This prevents a previous AT-command response execution to interfere with the next execution.

B.1.3 Unsolicited AT messages

Unsolicited messages are in the current simulator implementation executed with a periodic interval of two seconds. Each time the interval expires the script file periodic.js is run. To create an unsolicited message the periodic.js can be altered to

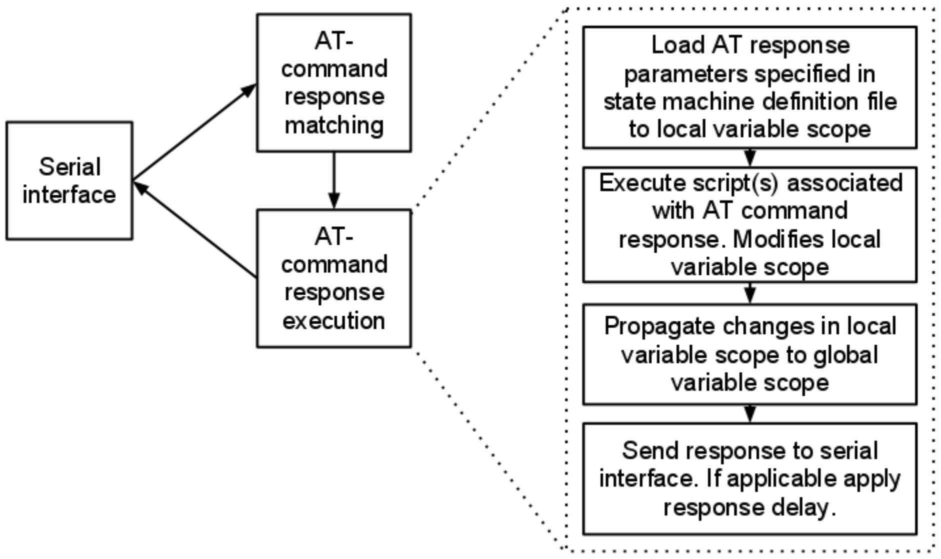

Serial interface
AT-command response matching
AT-command response execution
Load AT response parameters specified in state machine definition file to local variable scope
Execute script(s) associated with AT command response. Modifies local variable scope
Propagate changes in local variable scope to global variable scope
Send response to serial interface. If applicable apply response delay.

Figure B.1: Script Execution overview

look for changes in variables, states or other system parameters.

B.1.4 Configuration of Scripts

As described in the previous section, each AT command response may have one or more scripts attached. To enable a high degree of configuration flexibility a set of configuration xml files have been created named script configurations. A section of such a file can be viewed in Listing B.2. These files specify to what AT command and state a script file is to be associated. One script may be associated with several AT command responses in several states. A script binding may also be tagged as an assertion, as is done on line 20 in Listing B.2. This will trigger the simulator to regard the script return value as a test result. A failed test result will generate an error message in the simulator log file to allow easy tracing of the cause of the failure.

```
1  <?xml version="1.0"?>
2  <bindings>
3      <binding>
4          <!-- what state(s) the script is applied to -->
5          <states>
6                  <state>ALL</state>
7          </states>
8          <!-- what command(s) that the script is applied to -->
9          <ATs>
10                 <AT>AT+CFUN?</AT>
11         </ATs>
12         <scripts>
13                 <script>
```

```
                        <!-- The path to the script relative from the
                            simulator executable -->
                        <path>definitions/scripts/test.js</path>
                        <!-- If this is an assertion, the return value of
                            the script must be true. Otherwise an error
                            is triggered -->
                        <assertion>true</assertion>
                </script>
        </scripts>
    </binding>
</bindings>
```

Listing B.2: Script binding file example.

Several script xml files may be parsed and used in succession. This feature enables the creation of specific failure scenarios or tests sessions that easily can be added to a simulation instance. The section below describes how to configure which script configuration files that are to be used and several other simulation configuration options.

B.1.5 Global simulation configuration

The configuration of the global simulation settings is done in a separate file which is parsed upon simulation startup. This configuration file contains important information about what script files to load, what state machine definition file to use and several other properties. This file allows a user to quickly change the behavior of the simulation without the need to change file names or edit XML files or scripts. Failure and test scenarios can be prepared in separate script configuration files and can be enabled or disabled by adding the script configuration files in the global simulator configuration file. An example global simulator configuration file can be seen in listing B.3.

```
#base state machine and at configuration file
base_config=definitions/state_machines/default.xml

#the script binding files that are to be used
section script_xml
    definitions/script_bindings/scripts.xml
sectionEnd

#content for nmea output
nmea_file=definitions/gps/nmea.txt
#where to output nmea data
nmea_tty=/dev/ttyGS0
```

Listing B.3: Global configuration file example.

B.2 Gadget Driver

Whereas chapter 4.1 describes the final implementation of the USB gadget driver used by the MBMSF, this appendix contains a more detailed view of the development work flow.

For the interested, build scripts, source code, cross-build toolchains etc. can be achieved by mailing the authors of this thesis project. The addresses are found in the beginning of this paper.

The work flow steps is depicted in figure B.2, and more carefully explained in the following sections. The steps are:

1. Write the driver source code.
2. Compile kernel and cross-compilation tools.
3. Cross-build the driver using toolchain acquired in step 2.
4. Install the driver.

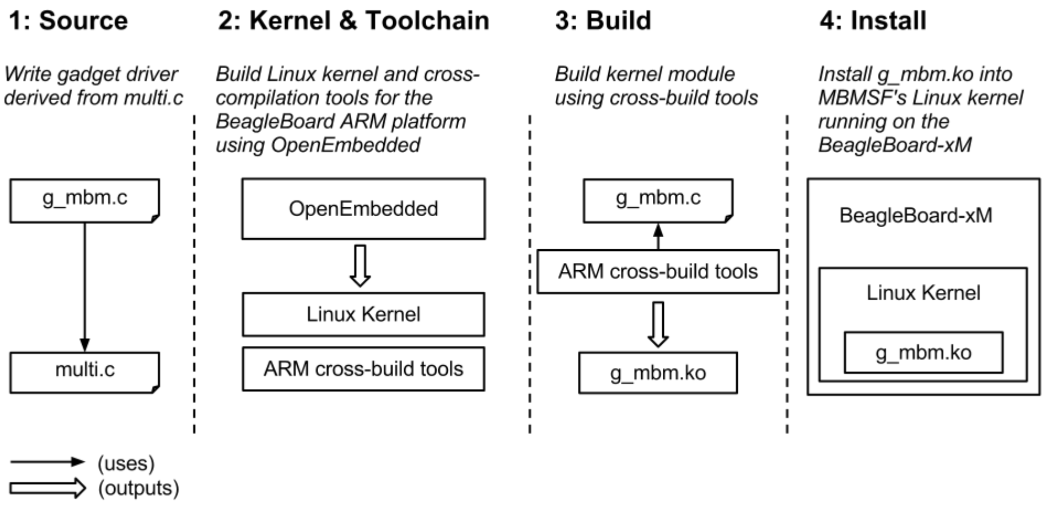

1: Source
Write gadget driver derived from multi.c
g_mbm.c
multi.c
2: Kernel & Toolchain
Build Linux kernel and cross-compilation tools for the BeagleBoard ARM platform using OpenEmbedded
OpenEmbedded
Linux Kernel
ARM cross-build tools
3: Build
Build kernel module using cross-build tools
g_mbm.c
ARM cross-build tools
g_mbm.ko
4: Install
Install g_mbm.ko into MBMSF's Linux kernel running on the BeagleBoard-xM
BeagleBoard-xM
Linux Kernel
g_mbm.ko
(uses)
(outputs)

Figure B.2: Gadget driver implementation work flow.

B.2.1 Step 1 - Source Code

The gadget drivers source code is contained in one file named *g_mbm.c* which borrows code from the reference driver *multi.c* [46]. multi.c is provided by the Linux Kernel Project as of version 2.6.33.

B.2.2 Step 2 - Build Kernel and Toolchain

The Linux kernel running the MBMSF is cross-compiled using *OpenEmbedded* which is a "build framework tool for embedded Linux" [14]. An overview of OpenEmbedded is found in section 2.2.6 of the theoretical framework chapter 2. More information is naturally found on their homepage [14]. The bitbake recipe used to produce the kernel was *linux-omap*. The cross-compilation toolchain derived from the process was found in the staging directory of OpenEmbedded and used in the build of the driver in the next step.

B.2.3 Step 3 - Build driver

The driver was ultimately built as an external loadable kernel module named *g_mbm.ko*, using the cross-build toolchain produced in the previous step. The build process was automated by the use of *kbuild* and custom Makefiles. The technical background of building external kernel modules for Linux is described in section 2.4.2 of the theoretical framework chapter 2.

B.2.4 Step 4 - Install driver

The recommended, and probably simplest, way to install/load the gadget driver is to use tools in the module-init-tools package. The general process of loading a kernel module in Linux is described in section 2.4.2. We take the following actions to install our module into the MBMSF after which it will load upon every system boot.

1. Place the module *g_mbm.ko* generated by *buildmodule.sh* at */lib/modules/2.6.39/kernel/drivers/usb/gadget/g_mbm.ko.*
2. Run $ echo "g_mbm" > /etc/modutils/g_mbm
3. Run $ update-modules

Appendix C

Tests

A serious of tests has been performed to ensure correct functioning and properties of the MBMSF. These results are presented throughout this report along with arguing around them. This appendix will describe more thoroughly how these tests were implemented along with some raw test data.

The appendix is divided into two sections where the first covers functional tests and the other section contains driver throughput tests.

C.1 Functional Tests

This section intends to provide a description of the functional tests performed on the MBMSF system to verify that is conforms to the requirements specified in appendix A.1.

C.1.1 Implementation

The tests were performed using bash scripting and execution in a terminal in Linux. Fundamentally, the scripts fed AT-commands into the simulator and analyzed the output to produce test results.

C.1.2 Results

The results, depicted in table C.1 below, are divided into six sections. These sections are:

- T.1 - Log Channel Properties.
- T.2 - State Machine.
- T.3 - Variable Assignment.
- T.4 - Script Execution.
- T.5 - Host Operating System.
- T.6 - Software Integration.

Num	Test	Required Outcome	Passed
T.1	Log Channel Properties		
T.1.1	Log output over ACM log channel works	Log data can be transmitted from MBMSF to host device.	OK
T.1.1	Large log volumes. Induce high log volumes sent from the MBMSF to the host device.	All messages needs to be received. No message loss may occur	OK
T.1.2	Long log duration. Require the system to record log over long time duration, at least 1 hour, 15 messages per second.	No log messages may be lost.	OK
T.2	State Machine		
T.2.1	State machine transition works.	State machine transition based on AT-command input works.	OK
T.2.2	Induce large amount of state changes over short time period based on AT-command input	Ensure that all states that AT-commands require have happened.	OK
T.3	Variable Assignment		
T.3.1	Basic variable assignment based on AT-command input should work.	Assigning, reading and reassigning variable work in MBMSF.	OK
T.3.2	Assignment of very long variable values	Assigning very long(more than 1000 characters) variable values should not cause problems	OK
T.3.3	Assignment of 0 length variable strings	Assignment of 0 length variable strings should not cause problems	OK

Table C.1: Functional tests.

Num	Test	Required Outcome	Passed
T.4	Script Execution		
T.4.1	Basic assignment of scripts to AT-commands by the use of the configuration files should work.	Using the script bindings XML, ECMAScripts should be attached and executed once an AT-command execution is run.	OK
T.4.2	Modifications of variables in scripts should work	Altering the value of a variable in a script should have permanent effect on the variable	OK
T.4.3	Changing state within a script should work	After the execution of a script changing the state the change should be complete	OK
T.4.4	Logging strings from within a script should be possible.	Invoking the log function call from within a script the log string should be written immediately to the log file.	OK
T.4.5	Altering the response string within a script should work	Invoking a function from the script should be enough to alter the response string sent back to the host device	OK
T.4.6	Applying delays to response strings should work	Invoking a function from within the script should be enough to apply a time delay when the AT-command response is sent back	OK
T.4.7	Chaining scripts together should create predictable results	Attaching and thereby executing several scripts after each other when an AT-command arrives should result in the following: if variable "A" is modified in script 1 and later "A" is modified in script 2, the modification applied in script 2 should apply to "A" once the script execution is finished. Other similar operations should act in a similar fashion	OK
T.4.8	Persistence in script execution	Ensure that the correct script is executed every time the AT-command is received. two executions every second for 4 hours	OK

Table C.2: Functional tests.

Num	Test	Required Outcome	Passed
T.5	Host Operating System		
T.5.1	Linux(Ubuntu) compability	Test to ensure that MBMSF enumerates correctly on Linux(Ubuntu)	OK
T.5.2	Windows Vista compability	Test to ensure that MBMSF enumerates correctly on Windows Vista	OK
T.6	Software Integration		
T.6.1	Linux interaction	Test to ensure that MBMSF can be controlled from built in network controller in Linux(Ubuntu)	OK
T.6.2	Windows Vista interaction	Test to ensure that MBMSF can be controlled from built in network controller in Windows Vista	OK
T.6.4	MBMSF Ericsson software compatibility	MBMSF is detected by Ericsson Wireless Connection Manager software	OK
T.6.5	MBMSF to Ericsson software information exchange	MBMSF information is displayed in Ericsson Wireless Manager	OK

Table C.3: Functional tests.

C.2 Driver Throughput Tests

These tests are devised to acquire the average data throughput speed between a host computer and the BeagleBoard-xM for three different drivers. The results are used in the discussions in section 7.2.

The drivers tested are two gadget interface drivers (NCM and mass-storage) and the Ethernet driver that comes with the Linux kernel 2.6.39 used for the MBMSF. Throughput limit is tested in both directions where *in* indicates a direction from the MBMSF to the host and *out* consequently means data directed from host to the MBMSF. Both directions are tested except for the mass-storage driver where only the in direction is measured.

The host computer used for these tests is a HP Compaq Mini 110c-1011SO laptop [8] which is suitable as representative for deployed MBMSF hosts, because it's specifications are assumed to be that of an average laptop.

The results section below contains the raw results which are further summarised into a diagram represented in figure 7.2 of section 7.2, around which arguing and conclusions are discussed.

C.2.1 Implementation

Three drivers are tested. These are:

- NCM (NCM interface of the *g_mbm.ko* gadget driver.)
- Mass-Storage (*g_mass_storage.ko* driver in the Linux 2.6.39 kernel)
- Ethernet (default driver for the Ethernet interface of the BeagleBoard-xM.)

The NCM and Mass-Storage drivers are interfaces of a gadget driver each. The Ethernet driver is the driver installed by default by the Linux kernel for the Ethernet controller of the BeagleBoard-xM used to control the physical Ethernet interface.

iperf [9] is used to test the NCM and Ethernet drivers. iperf is a utility that measures throughput by starting a server instance on one of the devices and a client instance on the other. The client instance then sends data continuously for a given amount of time and reports average throughput with a specified interval. See the results section for the output of the iperf invocations.

For the mass-storage test, a utility named *hdparm* [7] is used. This utility measures the in-directed data throughput of a storage device in Linux. To eliminate throughput bottlenecks imposed from reading from the BeagleBoard-xM's flash memory, the file read from the mass-storage device is memory-resident. That is, it is placed in the RAM memory of the BeagleBoard-xM which provides fast reading speeds.

All tests are repeated several times to ensure that no fluctuations of test results occurres.

C.2.2 Results

Following is an output of the test invocations of iperf and hdparm for the different drivers as presented in the summarised figure 7.2.

NCM/c (direction: in)

```
command executed on the MBMSF:
$ iperf -c 172.17.207.139 -t 70 -i 10
------------------------------------------------------------
Client connecting to 172.17.207.139, TCP port 5001
TCP window size: 16.0 KByte (default)
------------------------------------------------------------
[  3] local 172.17.207.1 port 54295 connected with 172.17.207.139 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  82.3 MBytes  69.1 Mbits/sec
[  3] 10.0-20.0 sec  86.0 MBytes  72.2 Mbits/sec
[  3] 20.0-30.0 sec  86.0 MBytes  72.2 Mbits/sec
[  3] 30.0-40.0 sec  86.1 MBytes  72.2 Mbits/sec
[  3] 40.0-50.0 sec  86.1 MBytes  72.2 Mbits/sec
[  3] 50.0-60.0 sec  86.0 MBytes  72.1 Mbits/sec
[  3] 60.0-70.0 sec  86.1 MBytes  72.2 Mbits/sec
[  3]  0.0-70.0 sec    599 MBytes  71.7 Mbits/sec
```

NCM/s (direction: out)

```
command executed on host:
$ iperf -c 172.17.207.1 -t0 -i 10
------------------------------------------------------------
Client connecting to 172.17.207.1, TCP port 5001
TCP window size: 16.0 KByte (default)
------------------------------------------------------------
[  3] local 172.17.207.139 port 35785 connected with 172.17.207.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  41.9 MBytes  35.1 Mbits/sec
[  3] 10.0-20.0 sec  41.9 MBytes  35.1 Mbits/sec
[  3] 20.0-30.0 sec  42.1 MBytes  35.3 Mbits/sec
[  3] 30.0-40.0 sec  42.2 MBytes  35.4 Mbits/sec
[  3] 40.0-50.0 sec  42.0 MBytes  35.2 Mbits/sec
[  3] 50.0-60.0 sec  42.1 MBytes  35.3 Mbits/sec
[  3] 60.0-70.0 sec  42.2 MBytes  35.4 Mbits/sec
[  3]  0.0-70.0 sec   295 MBytes  35.3 Mbits/sec
```

ETH/c (direction: in)

```
command executed on the MBMSF:
iperf -c 192.168.1.1 -t 70 -i 10
------------------------------------------------------------
Client connecting to 192.168.1.1, TCP port 5001
TCP window size: 16.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.2 port 35652 connected with 192.168.1.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec    113 MBytes  95.1 Mbits/sec
[  3] 10.0-20.0 sec    112 MBytes  93.9 Mbits/sec
[  3] 20.0-30.0 sec    112 MBytes  94.3 Mbits/sec
[  3] 30.0-40.0 sec    112 MBytes  94.0 Mbits/sec
[  3] 40.0-50.0 sec    112 MBytes  94.0 Mbits/sec
[  3] 50.0-60.0 sec    112 MBytes  94.4 Mbits/sec
[  3] 60.0-70.0 sec    112 MBytes  94.0 Mbits/sec
[  3]  0.0-70.0 sec    786 MBytes  94.2 Mbits/sec
```

ETH/s (direction: out)

```
command executed on host:
iperf -c 192.168.1.2 -t 70 -i 10
------------------------------------------------------------
Client connecting to 192.168.1.2, TCP port 5001
TCP window size: 16.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.1 port 43659 connected with 192.168.1.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec   113 MBytes  94.7 Mbits/sec
[  3] 10.0-20.0 sec   112 MBytes  94.2 Mbits/sec
[  3] 20.0-30.0 sec   112 MBytes  94.1 Mbits/sec
[  3] 30.0-40.0 sec   112 MBytes  94.2 Mbits/sec
[  3] 40.0-50.0 sec   112 MBytes  94.1 Mbits/sec
[  3] 50.0-60.0 sec   112 MBytes  94.2 Mbits/sec
[  3] 60.0-70.0 sec   112 MBytes  94.1 Mbits/sec
[  3]  0.0-70.0 sec   786 MBytes  94.2 Mbits/sec
```

MASS (direction: out)

```
command executed on host:
$ sudo hdparm -t /dev/sdb1
/dev/sdb1:
 Timing buffered disk reads:  64 MB in  3.07 seconds =  20.84 MB/sec

$ sudo hdparm -t /dev/sdb1
/dev/sdb1:
 Timing buffered disk reads:  64 MB in  3.05 seconds =  20.95 MB/sec

$ sudo hdparm -t /dev/sdb1
/dev/sdb1:
 Timing buffered disk reads:  64 MB in  3.06 seconds =  20.94 MB/sec

$ sudo hdparm -t /dev/sdb1
/dev/sdb1:
 Timing buffered disk reads:  64 MB in  3.06 seconds =  20.89 MB/sec
```

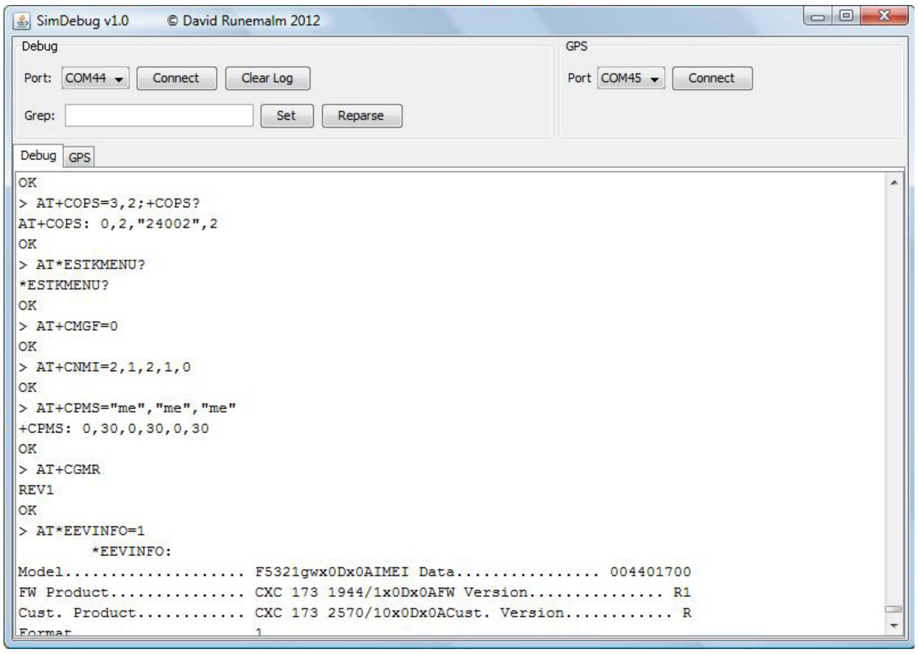
Appendix D

SimDebug

SimDebug is a platform-independent tool developed in Java. It was developed during this project to aid in configuration of the MBMSF's state machine and basically provides an output of the GPS and debug channels from the MBMSF. It incorporates filtering capabilities in UNIX *grep* style on the debug output stream.

The applications user interface is very simple. At program startup, the streams needs to be manually configured for the two output windows through drop-down boxes. After this assignment, the data coming from the MBMSF will be printed to the output windows which are provided as tabs, see an example in the figure below.

SimDebug is not a part of the project requirements but provided as proof-of-concept for the discussions in section 4.1 about debug output and filtering of the same.



```
OK
> AT+COPS=3,2;+COPS?
AT+COPS: 0,2,"24002",2
OK
> AT*ESTKMENU?
*ESTKMENU?
OK
> AT+CMGF=0
OK
> AT+CNMI=2,1,2,1,0
OK
> AT+CPMS="me","me","me"
+CPMS: 0,30,0,30,0,30
OK
> AT+CGMR
REV1
OK
> AT*EEVINFO=1
        *EEVINFO:
Model.................... F5321gwx0Dx0AIMEI Data................ 004401700
FW Product............... CXC 173 1944/1x0Dx0AFW Version.............. R1
Cust. Product............ CXC 173 2570/10x0Dx0ACust. Version........... R
Format                    1
```

Figure D.1: The SimDebug application in action. Debug and GPS output from MBMSF is supported as well as filtering capabilities.