Karri Kivelä

The Design Process of an Open Source Mobile Phone

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

23.5.2018

Tekijä Otsikko Sivumäärä Aika	Karri Kivelä Avoimiin lähdekoodeihin perustuvan puhelimen kehitystyö 32 sivua + 5 liitettä 23.5.2018
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Sulautetut järjestelmät
Ohjaaja	Yliopettaja Markku Nuutinen

Insinöörityön tarkoituksena oli suunnitella avoimen lähdekoodin piirilevy ja avoimen lähdekoodin ohjelmiston suorittaminen laitteella.

GSM-modeemin ja muun laitteiston ohjaukseen valittiin avointen lähdekoodien käyttöjärjestelmä Linux, jonka ydin myös käännettiin valitulle piirisirulle. Sen lisäksi perehdyttiin käyttöjärjestelmäydintä käyttävät kirjastot ja ohjelmat sisältävän Linux-käyttöjärjestelmälevyn muodostamiseen.

Puheluiden soittamista ja vastaanottamista varten kehitettiin Python-skriptejä, jotka ohjasivat GSM-modeemia suorittimelta UART-väylää pitkin. Python-skriptit vastasivat myös näppäimistön ja näytön ohjauksesta.

Proof-of-concept-soveltuvuusselvitys suoritettiin yhdistämällä suoritin- ja GSM-laitteistokehityssarjat toisiinsa käyttäen ulkoista LED-valodiodia näyttönä ja yhtä painiketta näppäimistönä. Kummankin laitteistokehityssarjan saatua virtansa ja suoritinkehityssarjan käynnistyttyä Linux-käyttöjärjestelmä ajoi laitteiston ohjaukseen käytetyt Python-skriptit.

Lopputyössä perehdyttiin myös piirilevyn kehityksen vaiheisiin. Kaikki projektin kehitystyö tallennettiin avointen lähdekoodien hengessä julkisesti saataville git-lähdekoodiversiointijärjestelmää käyttäen tunnettuun GitHub-palveluun.

Soveltuvuusselvitys osoitti, että niin puheluiden soittaminen kuin niihin vastaaminen laitteella onnistui. Python-skriptikokoelmassa todettiin kuitenkin suunnitteluvirheitä, joiden takia ohjelman jatkokehitys nykyisessä muodossa olisi haastavaa. Sen lisäksi paljastui, että piirilevyn kehityksessä käytetyt piirisommittelun rajasäännöt eivät sopineet piirilevyn valmistajalle eikä piirilevyä voitu valmistaa.

Koska piirilevysuunnittelu ja Python-skriptit ovat julkisesti saatavilla, kuka tahansa voi vapaasti jatkaa laitteen ja se toiminnan hiomista eteenpäin.

Avainsanat	Freescale i.MX233, SIM900, Olimex, sulautettu Linux

Author Title Number of Pages Date	Karri Kivelä The design process of an open source mobile phone 32 pages + 5 appendices 23rd May 2018
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Embedded Software Engineering
Instructor	Markku Nuutinen, Principal Lecturer

This project's primary aim was to develop an open source based GSM phone by using open source PCB design, and running an open source software on the device to control the hardware.

To control the GSM modem and other hardware components, the open source Linux operating system was chosen, and the steps on how to compile the Linux kernel for the specific CPU of this project were described. In addition, this project covered the generation of a simple Linux system image that consists of a set of libraries and binaries for user interfacing with the operating system.

Python scripts were developed in order to operate the GSM modem over the UART bus from the CPU, and thereby allow making and answering calls. User interfacing with a keypad and a screen were handled from the Python scripts as well.

The proof of concept was executed by physically connecting the hardware development kits of both the CPU and the GSM chipsets, and adding a LED to act as the screen and a button as they keypad. After both boards were powered up by external power supplies, the operating system on the CPU development kit board started running the main Python script.

The project also included designing the PCB for the end device. In the spirit of open source, all the development work was saved in a git source tracking repository in the public GitHub service.

The proof of concept showed that calls can be made, and can be answered, and that incoming calls were indicated on the screen. However, it was found out that the Python software had design flaws, so extending it for features might be challenging. Also it was discovered that the PCB design did not match the design rules of the fabrication house, and could not be printed.

Since the Python scripts and PCB design are publically available online, a basic GSM phone system can be made by anyone, and the concept can be further developed.

Keywords	Freescale i.MX233, SIM900, Olimex, Embedded Linux

Contents

Abbreviations

API	Application Programming Interface.
ARM	Originally Acorn RISC Machine, later Advanced RISC Machine.
AT	Attention.
ASIC	Application Specific Integrated Circuit.
CPU	Central Processing Unit.
DMA	Direct Access Memory.
ELF	Executable and Linkable Format.
GB	Gigabyte.
GPIO	General Purpose Input Output.
GSM	Originally Groupe Spécial Mobile, or nowadays Global System for Mobile Communications.
I2C	Inter-Integrated Circuit.
IC	Integrated Circuit.
LED	Light Emitting Diode.
MB	Megabyte.
MHz	Megahertz.
MMU	Memory Management Unit.
OS	Operating System.
PC	Personal Computer.

PCB	Printed Circuit Board.
PCI	Peripheral Component Interconnect.
RAM	Random-access Memory.
SD	Secure Digital.
SIM	Subscriber Identity Module.
SMD	Surface-Mount Device.
SMS	Short Message Service.
UART	Universal Asynchronous Receiver Transmitter.
USB	Universal Serial Bus.

1 Introduction

The GSM (Global System for Mobile Communications) standard was initially developed by the European Telecommunications Standards Institute (ETSI) in an attempt to find a common wireless telephone service standard across Europe [1]. In the 1980s, a wide variety of wireless radio systems was used by a large group of users, utilizing both private and public networks. Officials and radio manufacturers alike started recognizing the need for a shared radio spectrum between countries that would be reserved for a public telephony system. Despite the difficulties in finding such a common standard, such as the vetoing of each country for a technical standard that would be the easiest for its industries to manufacture, most suitable for its projected usage and closest to the systems used in its respective countries, and although the available radio spectrum aligned all over Europe was small, a consensus was found and the success story of connecting the whole world started. Since then, the use of mobile phones has become widespread.

Embedded computers have been around the consumer market for several decades. In today’s world, the cheaper processors used in these devices are becoming more powerful, being able to power more advanced operating systems, and Linux, the best-known and most-used open source operating system is often used on these devices in some form, due to its ease of customization.

It is not just the processor integrated circuit market that has experienced a significant evolution in the past years, but also the general semiconductor market, and especially the wireless communication market. The world is moving towards “Internet of Things” – more and more products for businesses and everyday consumers are connecting to the internet. There are several technologies to connect to the internet wirelessly over a radio channel, and the GSM-based technologies which are applied in cellular phones, offer a direct connection from anywhere under the coverage of the local GSM cellular operator.

The objective of this final project is to take a low-cost processor and a GSM cellular modem integrated circuit, and connect them to make a prototype of an open source based phone. This prototype is running a Linux operating system, and is able to initiate and receive calls. It is also a modular platform that can, for instance, be extended in software for transferring data over the GSM cellular network. The goal is to come up with a device that anyone at home can construct easily and develop further, and therefore, both the hardware and the software running on top are open source.

In this essay, we will go through every step of the procedure, from choosing the right processor and GSM IC, to building and compiling the Linux system, all the way to demonstrating how the final software application is running on the processor and controlling the hardware. Eventually, we will have a simple proof-of-concept of an embedded Linux system that is able to initiate and receive cellular phone calls. Additionally, we will review the custom hardware design, and analyze the difficulties that emerged during the process.

2 Requirements and the Main Components of the Open Source Phone

2.1 Hardware System Architecture

2.1.1 Main CPU and Peripherals

A CPU (central processing unit) is a microprocessor, or more precisely, a machine that consists of electronical components, that executes the orders that are given to it by the memory, decodes them and generates the required control signals to perform them [2]. The components that build up a CPU are basic logical gates essentially implemented by single transistors. Logical gates compare two input signals, and output one signal based on this comparison. In the case of systems that consist of several logical gates operating on interdependent signals that must be synchronized, such as the case of a CPU, the output change is done by the change of the system clock pulse.

Eventually, when putting such input signals in parallel, a bus of signals is created.
The system bus width is an important factor in CPUs, defining the bit width of the CPU, usually referred to as the “CPU architecture” (e.g. the usual 64 bit CPU architecture used in Intel PCs) [3]. The orders that are received by the CPU are given in the width of the CPU bus. In addition, the CPU always uses different input and output values to operate on. These values and the location they are saved in adhere to the CPU bus width as well. A smaller bit width requires fewer transistors to implement the logic, and thus the CPU is more power efficient. However, a smaller bus width also means that there will be fewer available instructions in the set of CPU commands, less possible

operations happening in only a single cycle without any follow-up cycle needed, smaller values that can be handled in one cycle, a smaller amount of memory that can be pointed to in a single cycle, and ultimately, a higher amount of CPU instructions and cycles will be required in order to perform a single action. Smaller buses, if so, are less efficient in terms of the operations they perform to achieve an end result.

The CPU is often perceived as the main core that performs the basic arithmetic calculations and logical operations on input values, receives the next operation from the memory, fetches the input values, and saves the output. However, in order to operate, the CPU needs peripheral devices around the core. Examples of such peripherals are basic communication buses, keyboard, screen, sound card etc. Usually, these peripheral devices refer to the CPU as a memory address, and therefore, the CPU can access them as it would access any regular memory device. Depending on the CPU architecture, there could be only the simplest peripherals in the core CPU silicon, and the CPU would expose several external buses for the purpose of adding different peripherals in a modular and unconstraint way. Alternatively, the CPU core could have many peripherals included in the silicon (system on a chip), instead of letting the user build the end system on the PCB, and requiring less modularity in terms of the buses it exposes [4]. The regular x86 IBM PC CPU architecture used in personal computers does not directly expose any peripherals, but a high speed internal bus that is then used to communicate to an external control chip. This chip could control memory and peripherals, but it also has wide modularity through the buses it exposes for the user. Some examples of such buses include the USB bus for on-the-air plug ‘n play of slower user devices (e.g. keyboard, hard drive, sound card), as well as the PCI bus for extremely fast peripherals (e.g. graphics card, networking card), with quick access to the RAM memory over DMA controllers. At the other end are the current ARM processors used for cellphones and other hand held and low power devices. These usually include everything on the same silicon with the CPU, from graphics controller and network controller, to even the actual data modem of the cellphone network.

No matter how many peripherals surround the CPU, it must always be connected to external interfaces. The logic the user requires from the CPU is not always possible, efficient or even economic to pack inside the same CPU chip. There is an endless amount of different use cases, each with its own specific hardware implementation. Additionally, some of the logic in the chip might require a complicated, higher voltage or higher current analogical circuit which might even be impossible to implement with the microscopic technology inside a silicon chip. In these cases, in order to power the chip, the IC exposes signal lines between the chip and the external circuit. For these reasons, a connection outside the chip through connection pins is often required [5]. During the phases of design, prototyping and testing, easy access to these pins is crucial, and even in the production phase it can be important, depending on the production method.

In the case of a mobile phone, one extremely important factor is the amount of power that the CPU consumes from the portable battery. Thus, the most power saving CPU is the most desirable one, as long as it answers the user's additional needs from the system. If the purpose of the design is not only cellular use, but necessitates calling for an operating system and internet browsing, the required CPU will be more complicated and power consuming.

When searching for a specific CPU IC for product development, the important properties are how much external circuitry is required, and how good is the reference design available for that IC. Also important is the memory bus, and what constraints does it have from the PCB. For prototyping, easy soldering of the CPU IC, as well as of the external RAM memory will be a useful feature.

2.1.2 GSM Modem

A GSM modem is a device that is used for creating and maintaining a wireless connection to the mobile phone operator's base network through the usage of certain radio bands and channels [6]. It handles the user identification and communication to the base network. The user initiates and receives calls through the modem, and exchanges SMS (Short Message Service) text messages. The modem can also be used for the transfer of data by connecting the device to the Internet.

Since the introduction of the Hayes command set (a set of commands that is named after the Hayes Microcomputer Products Company that defined it, and that is employed by most modem software), widely referred to as the AT (abbreviation for Attention) command set, most data modems have exposed the communication to the host CPU through the use of AT commands over a serial connection [7]. This means that the only requirement from the host CPU is the ability to use a universal asynchronous receiver-transmitter (UART) device.

The cardinal properties one has to look for in a GSM IC for product development are similar to those we focus on when looking for a specific CPU IC for the same purpose – The required external circuitry and the quality of reference design available for the IC should be primarily considered. Here, too, for the purpose of prototyping, easy soldering will be a useful feature.

2.1.3 Main Communication Bus

The most important function of a CPU that is used in a phone is to reliably communicate with the cellular modem on one of its external buses. This means that the architecture of the communication bus of the cellular modem must match with the buses available on the CPU, and the CPU must be able to attend to the received data quickly enough, as well as send responses in the defined time range, without data corruptions and errors on the communication bus. Asynchronous serial ports have been in use in computers for decades. The RS-232 (Recommend Standard n.232) based serial port has been widely applied as the communication bus for the exchange of commands and data since the early days of digital communications. The RS-232 is a hardware serial port protocol that uses duplex communication, utilizing an independent signal line per each direction [8]. There is no separate signal line for clock signal, so the latching of the information on the signal line is based on the clock of each of the devices. The lack of clock means that this protocol requires a highly accurate clock on the devices using it, and it also means that the practical speed limit is not very high, which, of course, was not a problem at the time of introduction. The speed of the data, the baud rate, is chosen beforehand and same rate is used on both devices. It is also possible to automatically detect the baud rate, but this can obviously be done on one side only – the other side will always need to have a predefined baud rate to allow baud rate detection. Although the most popular asynchronous serial connection is based on the RS-232 standard, there are other serial communication standards in the industry. When referring to all of these different asynchronous standards, the term UART is used.

The protocol that is used by data modems for exchange of data and control is called the AT command set. Each command would start with an AT-prefix. Nowadays, the AT command set is a standard used by all data modems. It was made popular by the Hayes modem manufacturing company in the 1980s. Up until then, every manufacturer had its own set of commands to control the data modem. The command set could vary even between different modems of the same manufacturer. The wide adoption of the AT protocol happened accidentally, when Hayes was rushed to publish their 2400 baud modem and to save time, kept the command set identical to their previous 1200 baud model. Users that wanted to update their modems were able to do that without changing their software. Other manufacturers started publishing modems mimicking the Hayes AT command set, turning it into an industry standard [9].

2.2 Software System Architecture

2.2.1 Operating System

An operating system is a software running on top of a CPU, and controlling its usage and the usage of other peripherals on the host device. The operating system controls the hardware of the host device, and exposes an API for the end application to use for implementing a certain functionality. With some operating systems, such as Linux, this API can be supported in a wide range of CPU architectures, meaning that the same application code can be easily compiled and run across physically different CPUs. The operating system is in charge of scheduling different tasks, and in certain cases also of the concurrent execution of tasks. In many advanced CPUs, it also separates the application space into kernel space and user space. This practically means dividing the running software into trusted and protected spaces, offering better runtime protection. The operating system code and the drivers controlling the underlying hardware usually have access to the trusted space, while end user applications are denied access to certain spaces. This protects user applications from performing malicious, such as accessing the memory of another running application, or accessing hardware while someone else is using it, intentionally or accidentally.

2.2.2 Linux Operating System

The kernel is the Linux operating system core. It provides a static, unchangeable user space API for any user space applications to work across any different Linux kernel versions. The kernel also has internal APIs for trusted kernel application space programs (i.e. drivers). The kernel’s job is to schedule different tasks, and to access or provide means to operate the underlying hardware through registerable drivers.

Usually, the Linux kernel is distributed in a binary form along with the root filesystem and user and system applications. A released version of the kernel is called a distribution [10].

The Linux operating system is a perfect fit for mobile phones, due to its abundance of capabilities, and since it is totally open-source. Linux offers many modules for hardware control, higher level logic and user space applications. If the CPU architecture has Kernel support, the mobile phone developer is mainly required to configure which GPIOs are in use. If the CPU has many copies of a peripheral such as UART, the developer must configure which peripheral instance is in use. These configurations are based on the system design and connections on the PCB.

2.2.3 Operating System Drivers

2.2.4 Hardware Control

In Linux there are several hardware control modules. Individual drivers can be registered to these modules in order for the operating system to be aware hardware’s existence. User application can then access the kernel API to control the hardware.

The simplest way to access hardware from the user space is to expose a character driver, i.e. a file handle that exists as a regular file system path. This way, any user space application that has permissions to access that file in the file system can read and write the file, and the actual driver in the kernel that is registered under this file will take the written data, and put back data that the user can read. The only thing the user application must know is the type of data the driver handle exchanges, i.e. what is the protocol the driver expects the data to be in.

In Linux, another way to access the hardware is by using block devices. Block devices are good for a fast and constant transfer of data between the user space and the device driver. As the name indicates, block drivers allow transferring bulks of data between the driver and the user [11]. For the integration of this project’s GSM-specific hardware, only character based drivers will be used.

In this project, the controlled hardware are the cellular modem, the keyboard and the screen. The cellular modem works over the UART bus, which has a character device in Linux. The character device must be opened, and the specific UART bus connection attributes of the port, such as the baud rate, must be set. This allows to read from the character device, and write to it to operate the serial data connection. In the proof of concept, the screen and keyboard are handled as single GPIOs; the screen is a single LED, and the keyboard is a single button. Accessing a GPIO hardware is simple. Linux supports GPIO control through the sysfs filesystem used to expose different hardware through file handles. It is possible not only to set and read the value of the GPIO, but also set other GPIO specific parameters, such as the direction and the edge of the interrupt triggering. Once these parameters have been set up per GPIO, there is a single file handle for each GPIO to access its value. Reading the file handle of the button returns the button’s state - closed or open, depending on whether we placed the button between the input pin and the voltage reference, or between the ground and the input pin. Writing to the file handle of the LED simply controls whether the LED is on or off. If this prototype is further developed in the future, the keyboard and screen could be connected over a slow speed I^{2}C bus. Possibly, both devices could even share the same slave I^{2}C device. The Linux kernel supports operating the physical I^{2}C hardware, so the I^{2}C device driver developer should register the driver to use the Kernel’s I^{2}C bus.

2.3 The Effect of Open Source on Device Requirements

The decision of going for an open source solution means facing a wide selection of predeveloped open-source resources. The principle of open source is allowing anyone to freely access any part of the project’s design, and often also making it legally possible for the public to use the design and edit it limitlessly. This also includes using a new design on top of the base open-source code for profit purposes [12]. In this project, the meaning of open source is designing hardware that is freely available for anyone who might be interested to further develop. It also means using an existing open source software, as well as developing some additional open source software for the benefit of the public.

To allow embedded developing, software (usually an operating system of some sort) must run on the hardware. It could be a simple loop process that is a hard platform to develop on for complicated runtime requirements and difficult to maintain in the long run, or a complex operating system that will have a high initial development cost, but very flexible and usually future-proof support for user needs. The main benefit of choosing an open source platform is that it removes the need for most of the development around an operating system for the device. Since Linux supports a wide range of embedded CPUs, it is highly customizable, and new support and updates are very easy to handle.

This, however, comes at a price. The CPU and main memory requirements for running Linux are particularly high. In order to run a regular Linux system in a comfortable fashion, the CPU is generally required to have an MMU, and the CPU data bus architecture should be at least 32 bit wide. Linux could basically run even on an 8-bit CPU, but it would do so very slowly, as it would have to handle more than 8 bit numbers and addresses most of the time. In addition, more time will be required to combine several CPU operations for those larger numbers, due to the need to store temporary results in the CPU registers.

The MMU requirement from the CPU is not as critical, and the Linux kernel even has a build option that allows running Linux without an MMU support in the hardware. This, however, will result in a low range of usable applications on the operating system. Both of the above mentioned features make the Linux-compatible CPU hardware more expensive to produce.

When looking for a solution that is easy to produce in small numbers and with small-scale tools, it is also important to pay attention to the packaging of the main ICs (in this case, the CPU, RAM and the cellular modem), and to their requirements for external circuitry. For this project, the ICs must be available in a packaging type that can be hand-soldered. In addition, the required external circuit should be reviewed carefully in the IC datasheet. If the IC of the modem can be hand soldered, but requires an external line encoder IC that cannot, it will not answer the requirements.

3 How to Work with Hardware Development Kits

3.1 Using a Hardware Development Kit to Boost Design

Once the main hardware requirements have been determined, in most cases it is preferred to start the software work on real hardware. If the product is new, the developing team is small and the CPU is available, starting the work on a hardware development kit or on a hardware reference design is often the most efficient choice. Oftentimes, especially with a CPU, the silicon manufacturer will publish its reference design and the appropriate drivers around it, as well as software examples. When using reference design, any support needed to bring up and maintain a custom own hardware design will be much easier to acquire. This might also be the only way for small and mid-sized customers to obtain support from the chip manufacturer if any issues arise.

A hardware development kit is developed by the chip manufacturer or by a third party, in order to assist in the development process. There are a few purposes for a hardware development kit. In almost all cases any integrated circuit requires at least some level of support from an outside circuit. The external circuit could be anything from a few resistors connected to the chip using a prototyping breadboard, to a very complicated external circuit requiring dozens of external components and strict PCB tracing requirements. A hardware development kit should answer the minimum requirement for an external circuit to run the IC, with all the components neatly placed on a PCB and tested to work. The simplest hardware development kits also expose the pins of the IC for the user in a more easily accessible way. The more advanced hardware development kits offer additional external circuitry to allow the users a wider range of components that they might implement in their final product.

The most complete form of a hardware development kit are the hardware reference designs. A reference design is a full design created by the IC manufacturer. It can be considered a fully designed product, that in the case of a CPU reference design, even comes with a wide variety of compatible software. This way, the developers can copy the reference design into their own design, performing only minor changes where needed. This also benefits the silicon manufacturers, as they can concentrate on the IC product support, and even be assisted by the community to maintain that support.

3.2 Choosing the Hardware Development Kits Based on Requirements

3.2.1 Main CPU Board

In the requirements phase we have identified the need for a CPU that is able to run an embedded Linux system. Olimex Olinuxino was selected for this project. It is a system that is based on the Freescale i.MX23 ARM processor line. The CPU has a nominal clock speed of 454 MHz, and the available RAM memory is 64MB. Olimex is manufacturing and selling hardware development kits for a variety of different ICs and CPUs, offering products that ease the prototyping of a single IC, or a whole hardware system board.

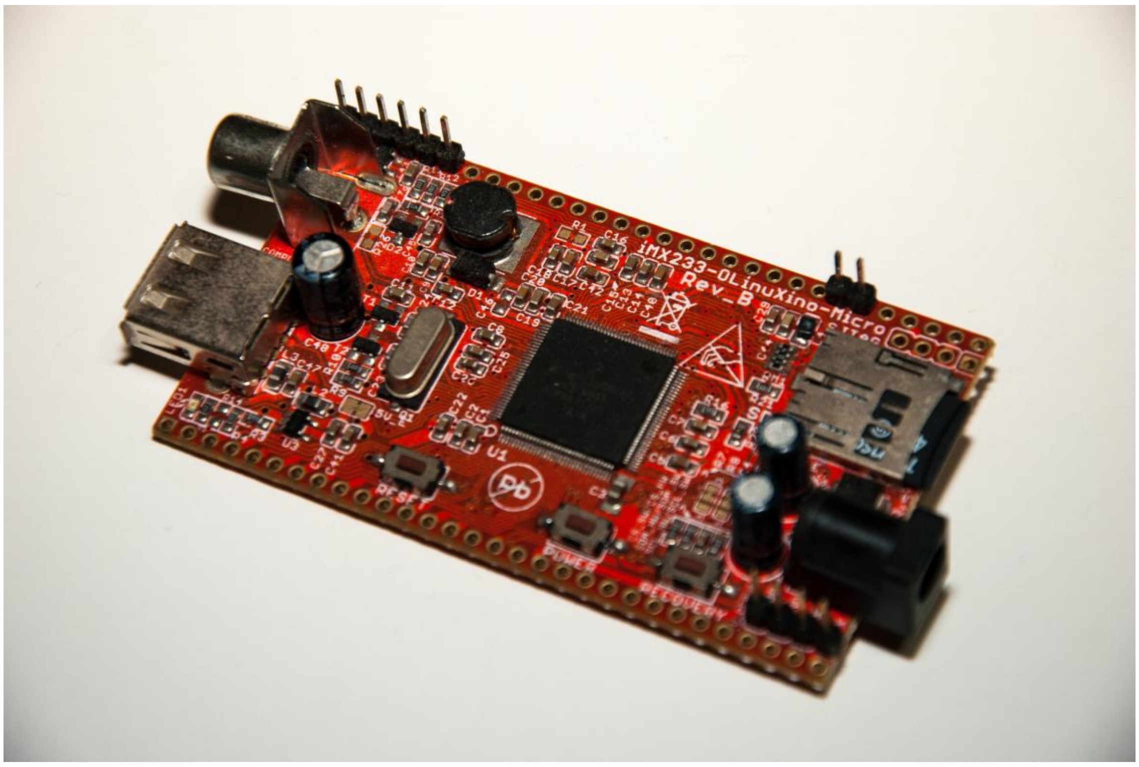


Figure 1 - Freescale multimedia iMX23 based Olinuxino Micro development board assembled and manufactured by Olimex.

Olimex offers the hardware schematics for most of their development boards, as well as at least simple software support to bring up the product in question, making it easy to learn and develop their kits further to an end product. The original Olinuxino line with the i.MX23 CPU was specifically planned to for simple use by home electronics enthusiasts, as it is hand-solderable, its components are highly available, and it consist of a simple two-layer PCB.

3.2.2 GSM Board

The GSM modem for this project is required to possess the capability to make phone calls, and a basic data transferring functionality.

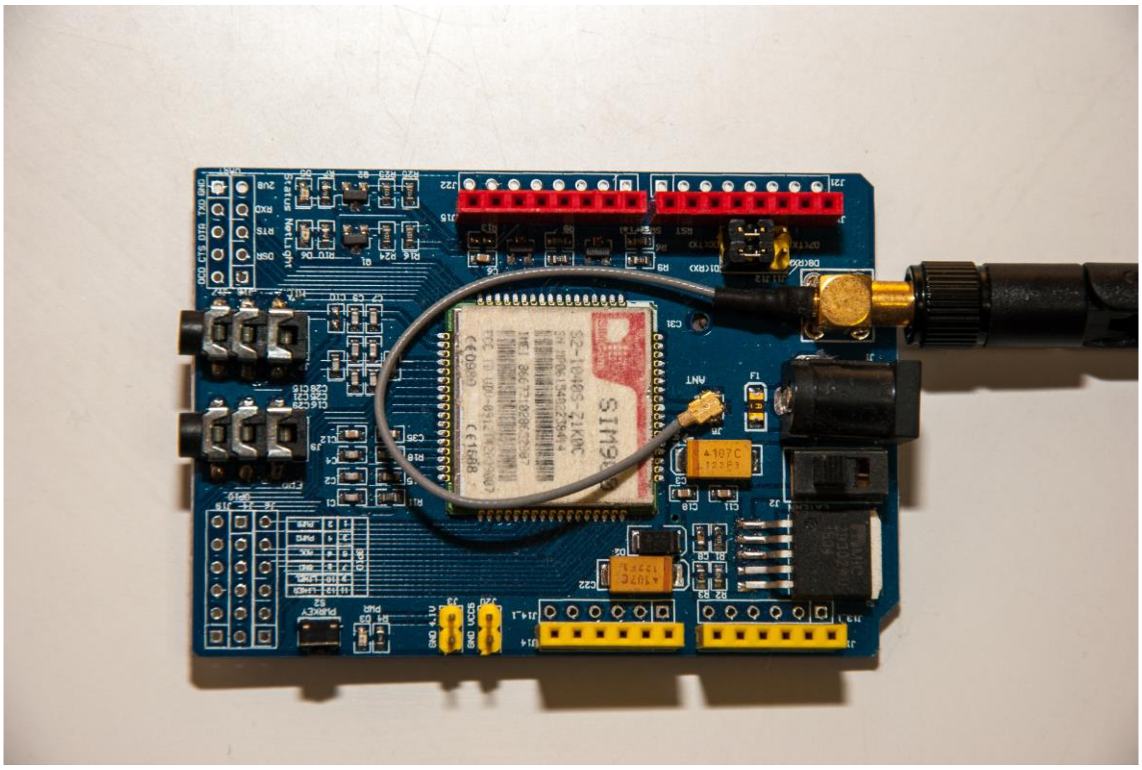


Figure 2 - A hardware development kit based on the SIM900 GSM chipset.

Ultimately, the SIM900 IC was selected for the project. It is widely available for a low price, it is fairly easy to hand-solder, and it is commonly used, especially in hobbyist projects. It uses a simple AT command interface over a serial port, and it is well documented. It also includes the capabilities of a data modem providing a future proof design for the product. The board has a socket that fits mini-sized SIM cards.

4 Hardware Design

4.1 Hardware System Design

The complete system design consists of choosing the ICs that match the product requirements, and plotting them on a circuit board with their required external circuitry, as well as the electrical connections between the ICs and other components. It also consists of verifying that the mechanical dimensions of the PCB match, and that the placed components will physically fit in the casing. In addition, it is essential to make sure that the signal integrity is good, and that the noise ratio is low, especially for high speed connections such as external RAM memory. The system design must also consider the temperature challenges.

For this project, I used an open source electronic design automation suite called KiCad. It includes the complete set of tools needed for every step of the design. It enables the user to design the logical connections between components, to place the components on the PCB, and to convert the final printing to a file that can be sent to the PCB manufacturer for printing.

Olimex designed the Olinuxino as a hobbyist-friendly hardware project, and published the hardware design (logic design) openly in their github repository online. While designing the hardware for this project, I followed the Olimex example from their Olinuxino micro product line. This enabled me to get a better idea of how to efficiently design the PCB layout of the components.

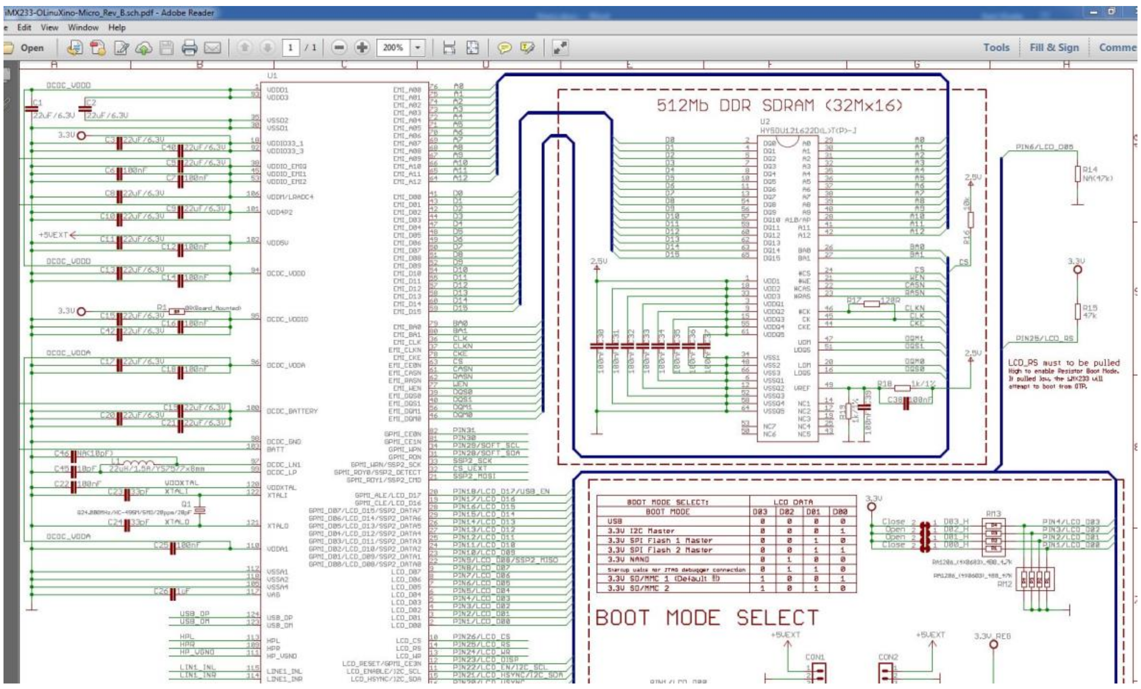


Figure 3 - Olimex publishes the logic connections of the Olinuxino PCB.

4.2 Main Circuit Board Logic Design

In the logic design stage the electronical circuit is developed. The logic design describes how different components are interconnected using isolated electrically conductive tracks. It also defines how many components there are in the design.

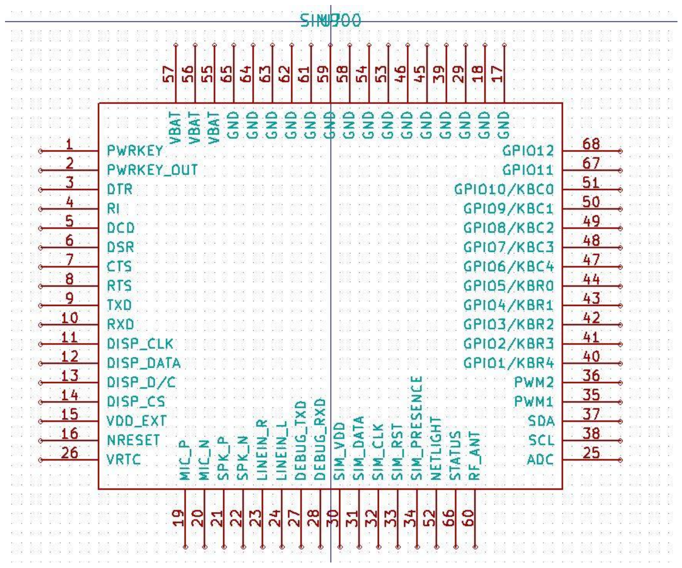


Figure 4 - SIM900 chipset in KiCAD part editor.

From the logic design point of view, each component has the following required attributes:

- Each component has a number of pins connecting it to the rest of the circuit.
- Each pin is defined to be of a certain electrical type, such as input, output, bidirectional, power input or power output.
- Each component has a reference number, that will be incremented for any other component added into the design. This would be the component’s name, and in case of analogical components such as resistors, the value of the component will make it easier for the eventual board assembler and the end user to work with the board. As a part of the component creation, the component will also get a symbol for the logic design.

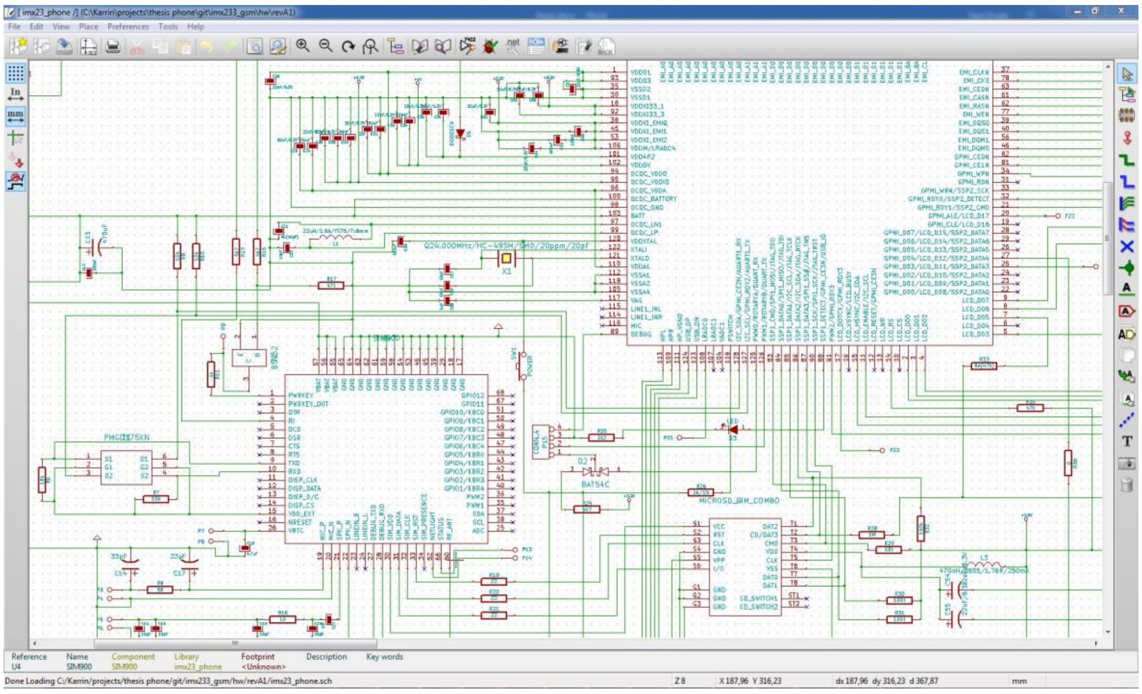
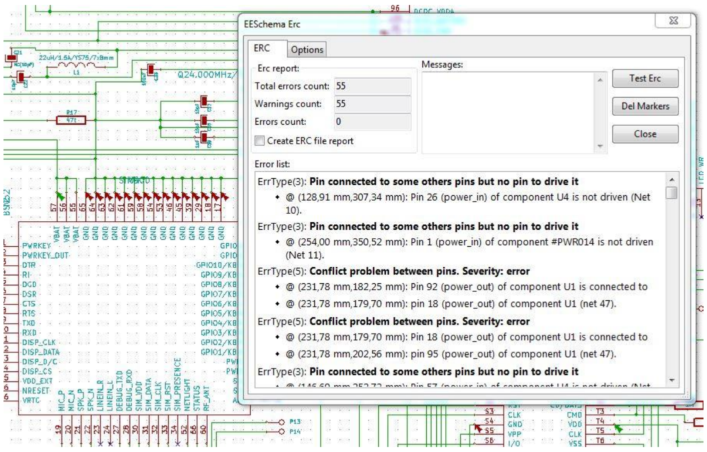

[imx23_phone /] (C:\Karrin\projects\thesis phone\git\imx233_gsm\hw\revA1)
File Edit View Place Preferences Tools Help
Reference U4
Name SIM900
Component SIM900
Library imx23_phone
Footprint <Unknown>
Description
Key words
Done Loading C:/Karrin/projects/thesis phone/git/imx233_gsm/hw/revA1/imx23_phone.sch
Z 8
X 187,96 Y 316,23
dx 187,96 dy 316,23 d 367,87
mm

Figure 5 - A part of the logic design of the GSM phone PCB.

After the components were obtained (either created by the user, or imported from ready-made libraries), they can be added to the logic design. At this stage, the electrical connections between the components must be plotted by placing wires between them. Depending on the electrical type of the pins in the design, all of the pins must be connected to another component. On a per-component basis, a pin can also be left unconnected in the end design.

In the final stage of the logic design, it is required to run the Electrical Rules Check, to make sure all the pins are connected, and that the connections are correct (e.g. no regular input pins connected to power output wires). Running the error correction is a useful and easy way to reduce redesign and reprinting of a PCB. At the end of the process, the logic design will be exported as a netlist, a list of the components



Erc report:
Total errors count: 55
Warnings count: 55
Errors count: 0
Create ERC file report
Messages:

Error list:

ErrType(3): **Pin connected to some others pins but no pin to drive it**
- @ (128,91 mm,307,34 mm): Pin 26 (power_in) of component U4 is not driven (Net 10).

ErrType(3): **Pin connected to some others pins but no pin to drive it**
- @ (254,00 mm,350,52 mm): Pin 1 (power_in) of component #PWR014 is not driven (Net 11).

ErrType(5): **Conflict problem between pins. Severity: error**
- @ (231,78 mm,182,25 mm): Pin 92 (power_out) of component U1 is connected to
- @ (231,78 mm,179,70 mm): pin 18 (power_out) of component U1 (net 47).

ErrType(5): **Conflict problem between pins. Severity: error**
- @ (231,78 mm,179,70 mm): Pin 18 (power_out) of component U1 is connected to
- @ (231,78 mm,202,56 mm): pin 95 (power_out) of component U1 (net 47).

ErrType(3): **Pin connected to some others pins but no pin to drive it**

Figure 6 - Electrical rules check is an important process in the logic design to reduce the amount of faulty PCBs.

which will be used as an input for the layout design, and can be also used as a Bill of Materials for purchase of the project's required components.

4.3 Main Circuit Board Layout Design

While the logic design defines connections between components and their pins, the layout design dictates how and where the components are placed on the board, and especially how the connections between components are drawn in the actual board.

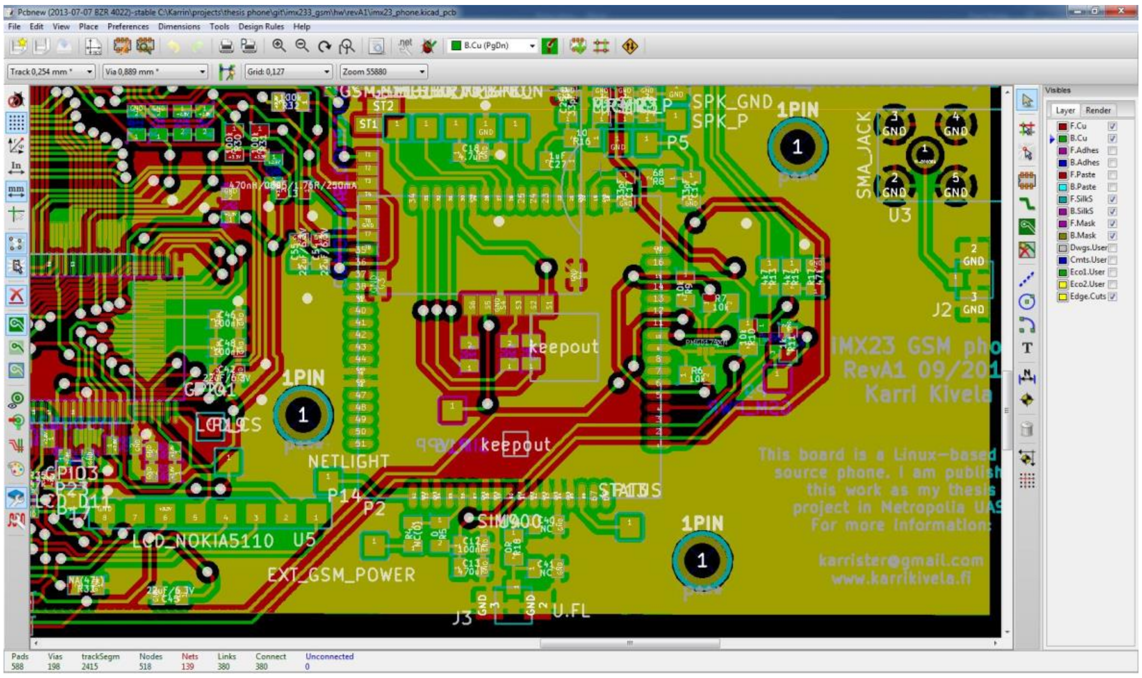


Figure 7 – The GSM board PCB layout.

Once the required size of the physical board is reduced to a certain level, and the circuitry and amount of components are complex enough, traces and components must be placed on several layers. One layer is rarely sufficient, unless the circuit is very simple, as often traces cannot be run around other traces and components, but must jump to the other side of the circuit board. The easiest solution is to use the board's other side as the second layer. It is also possible to get more layers by stacking up boards, having several internal layers with invisible traces running around. To connect to another layer, one must drill holes called "vias". These holes function as tunnels that channel the traces to another layer of the board [13].

4.4 Circuit Board Printing

There are several circuit board printing methods. In boards that are entirely covered by a metal layer, usually copper, the connections of component pins and the electrical connections between components are made by etching lines into the metal layer, to break metal surfaces from each other on the board.

Another method is to make a mask layer, essentially a drawing of the final circuit, containing the drawn soldering points and the electrically conductive traces, and etching the circuit with the assistance of chemicals. In this method, the circuit is printed on a special paper that has a substance that protects the metal coper surface from the etching chemical. After the printed circuit is put on the copper metal board, the copper covered board is sank into a chemical etching liquid that will etch off all the copper except the parts that are protected by the mask that was transferred from the printed circuit. The same effect can also be reached with a special pen, by drawing the surface on the copper. Eventually, the circuit board is mostly a plastic platform that has a metal surface in specific places to allow connecting to components, and running the traces between those metal pads [14].

4.5 Circuit Board Assembly

Once the circuit board is printed, it has all the electrical connections on it, as well as the designated places or pads allowing connection to components and to external circuits. The next stage is to assemble all the components onto the board. The components must have an electrical connection with their respective pads on the circuit board. To achieve this, soldering should be performed. The assembler will heat up both the pad on the circuit board and the pin of the component, and apply the joining, gluing substance that will melt into both the pin and the pad, keeping the component attached to the board through its legs.

There are two types of connections of components to the board – The first type is the SMD (surface-mount device) method, in which the component is mounted directly onto the pad of the board. The second type is the through-hole connection, in which a hole is drilled in the circuit board, and the component is then soldered onto the opposing side of it.

4.6 External Keyboard

The keyboard of the system is placed on a separate PCB that is connected to the main PCB. The keyboard is the interface that enables the user to control the device. In the proof of concept device, there is a single button that is used as the keyboard. When pressed, it either answers an incoming call, or initiates an outgoing call to a pre-saved phone number.

4.7 External Screen

The screen is connected to the main PCB with an internal cable. The screen outputs information for user about the current state of the device. In the proof of concept device, the screen is a single LED. The LED is lit when the device is ready to make and receive a call, and blinking when there is an incoming call.

5 Software Design

5.1 Operating System Required Parts

Before the CPU can start running the operating system code, it must first get to a stable state, where all the core CPU hardware and peripherals have gone through an initialization stage. Once this is accomplished, the CPU must also know where the start of the operating system code resides. This is the job of the bootloader. When the power of the device is turned on, the CPU will first start reading the program code from a hardcoded, previously defined address. The bootloader will handle the loading of the basic peripherals, and jumping to the beginning of the kernel code, which will also have its own boot phase. In this phase, the bootloader will detect and initialize more generic peripherals, such as full hard drive capabilities, bringing up more advanced graphical modes of the display adapter, and detecting and configuring the network adapters and other hardware elements present in the system. This will finally lead into running a set of system and user binaries that run in the user space of the operating system, as defined in the boot configuration, and the operating system will reach the final operating state.

In Linux, the operating system is divided into two parts – the kernel and the functions around it. The kernel is in charge of accessing the hardware, protecting it from erroneous usage as much as possible, and scheduling tasks that the user prompts. The kernel in itself, however, does not provide the desired user functionality. Therefore, a basic set of applications is required. Such set can be further divided into system applications and user applications. The system applications implement logic that require a privileged mode of execution in the operating system, such as direct configuration of the hardware. The user applications execute user related tasks, such as interfacing with other processes, or using the hardware through system provided libraries, such as drawing something on the screen, or sending a packet to the network.

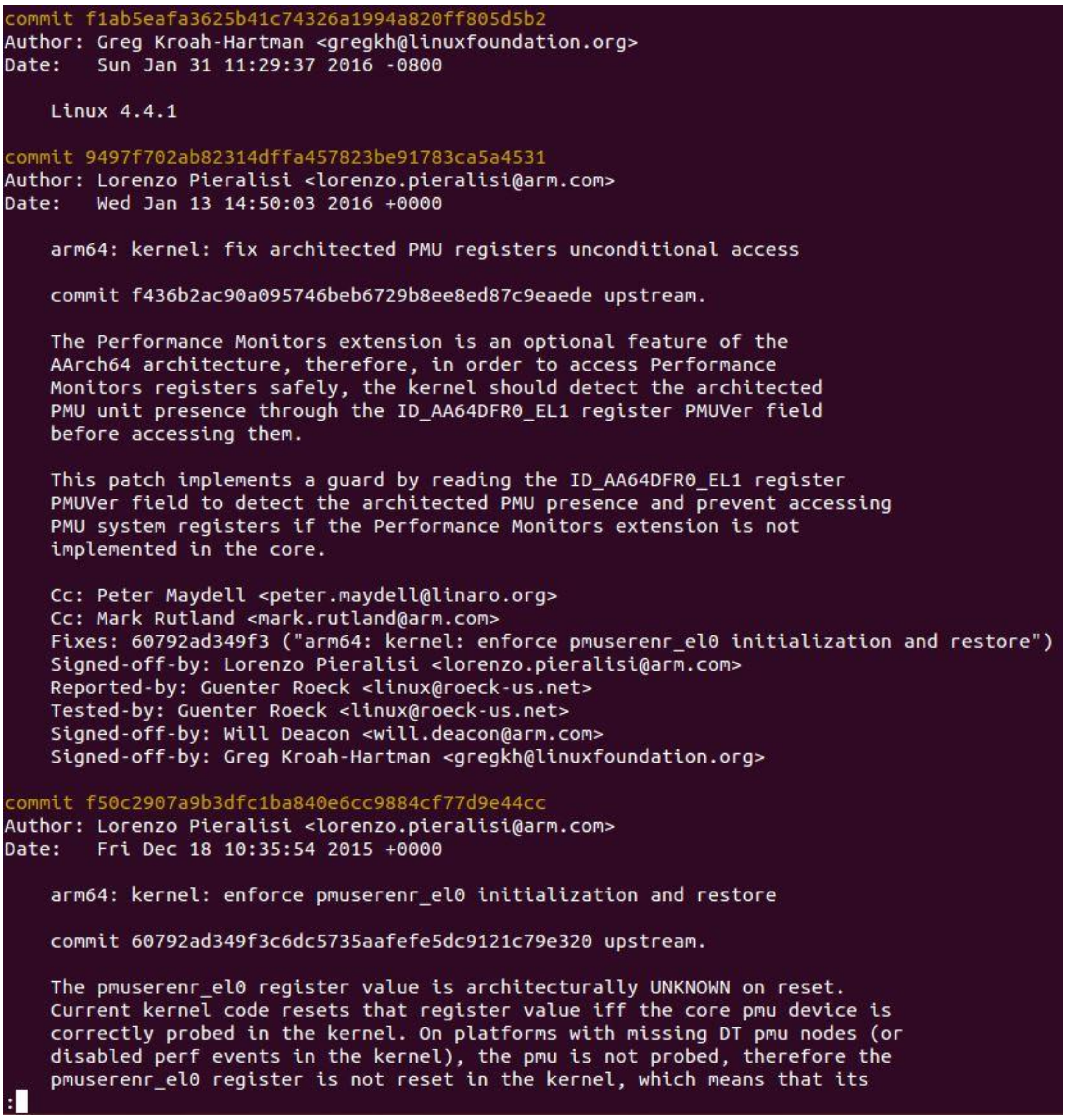
5.2 Linux Kernel Build Process

The kernel is basically the core of the operating system. Similarly to a car engine, it has different programs that use the underlying hardware, such as CPU, memory, hard drive, and graphics cards, making sure that the hardware and resources are used in an organized manner. It then exports interfaces for the Linux system user to operate the hardware and the system, resulting in a generalized way for applications to access the operating system.

Most of the distributions publish the kernel as a prebuilt binary for the designated system. In the case of an embedded computer with a different CPU architecture, many times it is the CPU manufacturer that takes care of publishing the kernel binary. It is required, however, to build the kernel independently if there are any changes that would need to be performed, such as updating or fixing one of the drivers or modules that come with the kernel, or writing a new module to be loaded with the kernel. It is possible, for instance, to implement a driver module that handles the operation of the phone’s screen hardware.

To be able to build the kernel, a compiler is, of course, required. Unless one has access and patience to compile on the actual CPU that the kernel will eventually run on, a cross compiler should be obtained. This compiler will not compile code into the CPU architecture of the computer that the compiler is run on, but actually for a different CPU. Thus, one can use a regular PC to compile code, for example, compiling code for ARM.

After the compiler is set, the next step is to download the kernel source code. The founder of Linux, Linus Torvalds, has developed a source control system called git. This system is used to control different versions of the Linux kernel source code. Git enables the user to get the latest version of the kernel source code, possibly with the latest fixes, as well as checking out an older version of the kernel source code that might be more stable on a specific system.



```
commit f1ab5eafa3625b41c74326a1994a820ff805d5b2
Author: Greg Kroah-Hartman <gregkh@linuxfoundation.org>
Date:   Sun Jan 31 11:29:37 2016 -0800

    Linux 4.4.1

commit 9497f702ab82314dffa457823be91783ca5a4531
Author: Lorenzo Pieralisi <lorenzo.pieralisi@arm.com>
Date:   Wed Jan 13 14:50:03 2016 +0000

    arm64: kernel: fix architected PMU registers unconditional access

    commit f436b2ac90a095746beb6729b8ee8ed87c9eaede upstream.

    The Performance Monitors extension is an optional feature of the
    AArch64 architecture, therefore, in order to access Performance
    Monitors registers safely, the kernel should detect the architected
    PMU unit presence through the ID_AA64DFR0_EL1 register PMUVer field
    before accessing them.

    This patch implements a guard by reading the ID_AA64DFR0_EL1 register
    PMUVer field to detect the architected PMU presence and prevent accessing
    PMU system registers if the Performance Monitors extension is not
    implemented in the core.

    Cc: Peter Maydell <peter.maydell@linaro.org>
    Cc: Mark Rutland <mark.rutland@arm.com>
    Fixes: 60792ad349f3 ("arm64: kernel: enforce pmuserenr_el0 initialization and restore")
    Signed-off-by: Lorenzo Pieralisi <lorenzo.pieralisi@arm.com>
    Reported-by: Guenter Roeck <linux@roeck-us.net>
    Tested-by: Guenter Roeck <linux@roeck-us.net>
    Signed-off-by: Will Deacon <will.deacon@arm.com>
    Signed-off-by: Greg Kroah-Hartman <gregkh@linuxfoundation.org>

commit f50c2907a9b3dfc1ba840e6cc9884cf77d9e44cc
Author: Lorenzo Pieralisi <lorenzo.pieralisi@arm.com>
Date:   Fri Dec 18 10:35:54 2015 +0000

    arm64: kernel: enforce pmuserenr_el0 initialization and restore

    commit 60792ad349f3c6dc5735aafefe5dc9121c79e320 upstream.

    The pmuserenr_el0 register value is architecturally UNKNOWN on reset.
    Current kernel code resets that register value iff the core pmu device is
    correctly probed in the kernel. On platforms with missing DT pmu nodes (or
    disabled perf events in the kernel), the pmu is not probed, therefore the
    pmuserenr_el0 register is not reset in the kernel, which means that its
:
```

Figure 8 - Git version control system is used to track and develop Linux kernel source code.

Before starting the compilation of the Linux kernel, the appropriate kernel module for compilation should be selected out of the great amount of available kernel modules. Each computer system has its own default configuration that defines what is built and what is not, but the user can further edit the list of modules. This easiest way to do that is using an interactive command line utility. Each module can have one of 3 states for the compilation. First of all, the user can decide that the module will not be compiled nor included in the kernel at all. If the user chooses to build a specific module, it can also be determined whether this module is built statically into the kernel, or will it be built as a loadable kernel module. For instance, one of the hardware buses on the board can be utilized for a network chip with Wi-Fi capabilities, and thus, it is important to verify that the driver module is compiled during the kernel build.

As a part of the kernel build, it is also required to setup the bootloader that handles the initial boot of the system into the operating system. A usual way is to use U-boot, which is an open source, widely supported software with several features. In this case, Freescale also offers a simple bootloader called Bootlet, which is very CPU specific, but takes care of the necessary steps to boot into the Linux kernel. The Bootlet is a simple code running on the CPU, bringing the CPU and its peripherals to a state that the operating system image can be loaded.

5.3 Operating System Image Generation

Distributions are publishing a solution with kernel and all the system and user applications as a package of components predefined by the distributer. Most of the time, the user can further customize the system using a package managing system, that allows the user to download and install new features in an organized manner. In an embedded system, this is not always feasible, since not all embedded systems have a networking interface. In addition, it could be difficult to access and use the system remotely in order to install new packages. It is also more convenient for the user of that specific embedded system to get a customized version of a distribution, that can be used without any further package managing or software updates. Thus, it is important to have tools to customize a distribution. In this project, I was mostly interested in the ability to run Python scripts for performing a part of the hardware control over them.

Special tools are needed in order to be able to make a distribution that will be used on a portable ARM system that does not have internet access. Such tools exist for Linux, and there are a few steps required to obtain a fully functioning system image. First, the user must change the operating system root directory in a local shell session to the root directory of the target device, such as the locally connected media that will be used on the target device. Second, an emulation software is needed in order to emulate the hardware architecture on the target system. This is required for running some applications for image generation, such as basic shell commands. After the emulator is in place, we can use the applications of the target device architecture for downloading and generating the root file system.

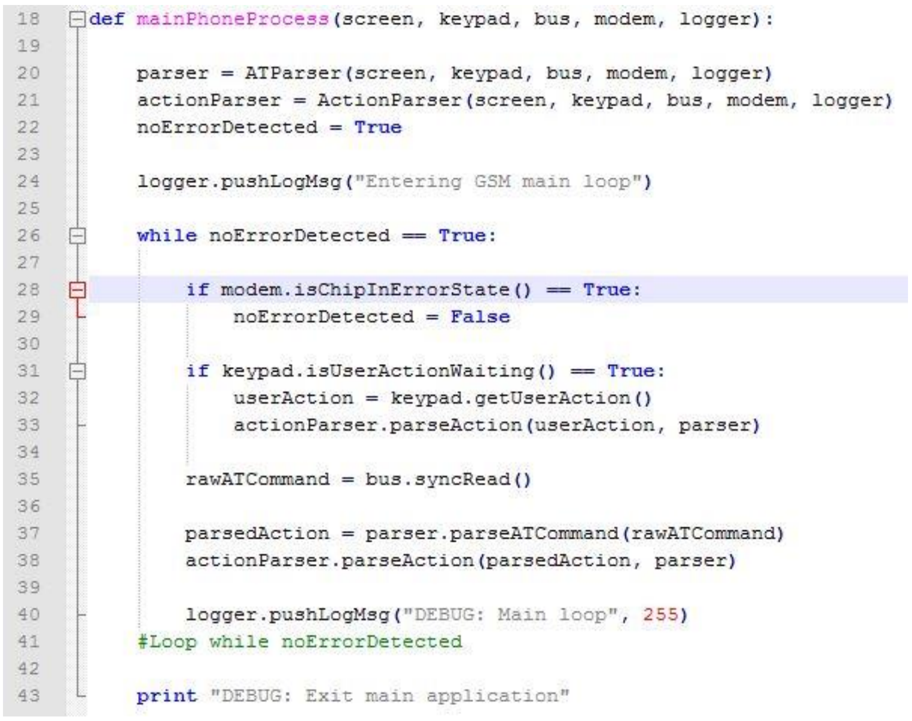
5.4 The Phone Application

Every device with a CPU has a main application that is started in the beginning of the CPU power on. After initializing and bringing up the hardware to an operating state, the main application will start executing the application-specific code. Traditionally in embedded application-specific computers, the main application is a single process in the whole system, being executed line after line, containing the application specific code. As embedded CPUs are getting more powerful, running an operating system on the CPU to control the hardware and to schedule the tasks is becoming a popular method. This way, the application or applications can be written as executable programs that are run and scheduled by the operating system, and can be easily run on different hardware platforms, providing that the operating system remains the same.

Several operating systems also have support of compilers for other languages than the operating system’s native language. In this project, a Python application is run over the shell of an embedded Linux. Python is a script-like, object-oriented programming language that is quite easy and intuitive to use. It has a diverse selection of standard libraries coming in the default installation, providing the developer with many usable features.

The Python application used in this project is written in a single threaded, poll-based architecture. After initializing and setting up the needed phone hardware, the main application flow will run in a loop. In the beginning of the loop, the keyboard is polled to determine whether there is any user action to take place. If a user action exists, the action is read and then passed on to the action parser that will take the required action and update the screen. The action could be, for instance, initiating a call based on the user pressing the button. In this proof of concept stage, this is the only user action supported. Next, the serial port is polled for any commands from the GSM modem. Incoming commands are passed on to the AT commands parser module, which will then, at the end of the loop, either result in an action for updating the screen, or for generation of a command response to the modem.

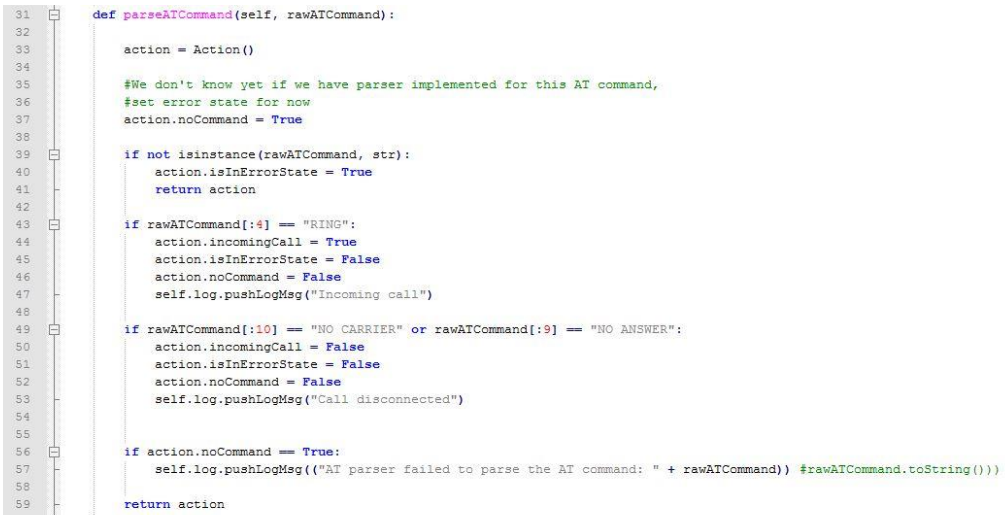
The Python application is divided into different modules. The main module is responsible for the initialization and calling of other modules, as well as the main application flow and the main loop.



```python
def mainPhoneProcess(screen, keypad, bus, modem, logger):

    parser = ATParser(screen, keypad, bus, modem, logger)
    actionParser = ActionParser(screen, keypad, bus, modem, logger)
    noErrorDetected = True

    logger.pushLogMsg("Entering GSM main loop")

    while noErrorDetected == True:

        if modem.isChipInErrorState() == True:
            noErrorDetected = False

        if keypad.isUserActionWaiting() == True:
            userAction = keypad.getUserAction()
            actionParser.parseAction(userAction, parser)

        rawATCommand = bus.syncRead()

        parsedAction = parser.parseATCommand(rawATCommand)
        actionParser.parseAction(parsedAction, parser)

        logger.pushLogMsg("DEBUG: Main loop", 255)
    #Loop while noErrorDetected

    print "DEBUG: Exit main application"
```

Figure 9 - The main Python process handles the main logic.

The AT commands parser module parses AT command strings, and turns them into actions and vice-versa. The parsing and generation of the AT commands of the modem are easily written in Python, which has good and clear standard library functions for string manipulation.

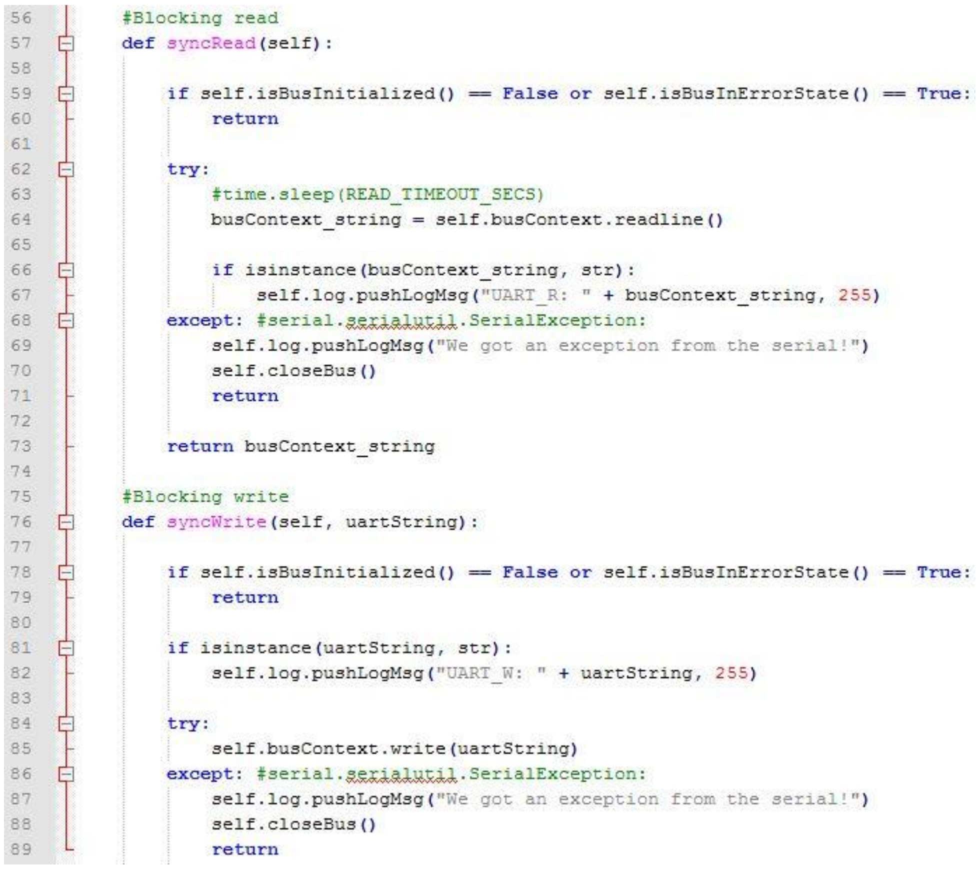


```
def parseATCommand(self, rawATCommand):

    action = Action()

    #We don't know yet if we have parser implemented for this AT command,
    #set error state for now
    action.noCommand = True

    if not isinstance(rawATCommand, str):
        action.isInErrorState = True
        return action

    if rawATCommand[:4] == "RING":
        action.incomingCall = True
        action.isInErrorState = False
        action.noCommand = False
        self.log.pushLogMsg("Incoming call")

    if rawATCommand[:10] == "NO CARRIER" or rawATCommand[:9] == "NO ANSWER":
        action.incomingCall = False
        action.isInErrorState = False
        action.noCommand = False
        self.log.pushLogMsg("Call disconnected")


    if action.noCommand == True:
        self.log.pushLogMsg(("AT parser failed to parse the AT command: " + rawATCommand)) #rawATCommand.toString()))

    return action
```

Figure 10 - The AT command parser translates string messages from the modem into actions inside the Python application.

The phone project has separate modules for both the screen and the keyboard. These modules handle the hardware interfaces of those devices, and exposing API towards the main module to use for user interface.

The serial module is a low level bus driver that communicates with the hardware serial port through the open-source Pyserial library. It also takes care of opening the serial port handle and configuring it at the initialization stage.



```
    #Blocking read
    def syncRead(self):

        if self.isBusInitialized() == False or self.isBusInErrorState() == True:
            return

        try:
            #time.sleep(READ_TIMEOUT_SECS)
            busContext_string = self.busContext.readline()

            if isinstance(busContext_string, str):
                self.log.pushLogMsg("UART_R: " + busContext_string, 255)
        except: #serial.serialutil.SerialException:
            self.log.pushLogMsg("We got an exception from the serial!")
            self.closeBus()
            return

        return busContext_string

    #Blocking write
    def syncWrite(self, uartString):

        if self.isBusInitialized() == False or self.isBusInErrorState() == True:
            return

        if isinstance(uartString, str):
            self.log.pushLogMsg("UART_W: " + uartString, 255)

        try:
            self.busContext.write(uartString)
        except: #serial.serialutil.SerialException:
            self.log.pushLogMsg("We got an exception from the serial!")
            self.closeBus()
            return
```

Figure 11 - The serial module of the application takes care of the communication with the serial hardware.

The main Python process is configured to be run at Linux startup by the initialization configuration. This way, when the power is applied to the device after the initial boot stage of the operating system, the Python application of the main process is run in the background of the operating system. This can be confirmed by observing the LED of the device when it powers up.

6 Performing a Simple Proof of Concept

6.1 The Principle

A proof of concept is the practice of conducting a small-scale test to demonstrate that a certain idea works. The conductor of the proof of concept might have planned a very complex system that is rich with features and robust and stable enough in any environment. However, as a proof of concept, the purpose is to simply show that the main idea or feature of the system or product works, even if not even nearly in the final form. In this project, I have set out a target of building an open source phone running a Linux operating system. The phone's planned capability was to receive and start calls. I have chosen the target hardware (CPU and GSM chipsets). The proof of concept was aimed to demonstrate an incoming and an outgoing call with the GSM development kit. The CPU development kit running the Linux operating system, initiated a call through the use of the serial port. The serial port was operated via user space device handle from the Python application. The incoming call into the device from another regular GSM phone, was answered on the proof of concept device by the press of a button, similarly operating the Python application.

6.2 Getting up a Working System on the Hardware and Running the Proof of Concept

In the proof of concept, the device is not a finished product, but rather a prototype that has functionally the same hardware as the planned finished product. In this project, I am using two separate boards, one for the CPU and its close peripherals, and a second board for the GSM connectivity. The boards are connected together over a UART bus for serial connection, and the command of the GSM modem is done from the CPU. Both boards have their own 5-volt power supplies, and both are placed in a transparent plastic box. The cover of the box acts as the user interface panel that consists of a button and a LED.

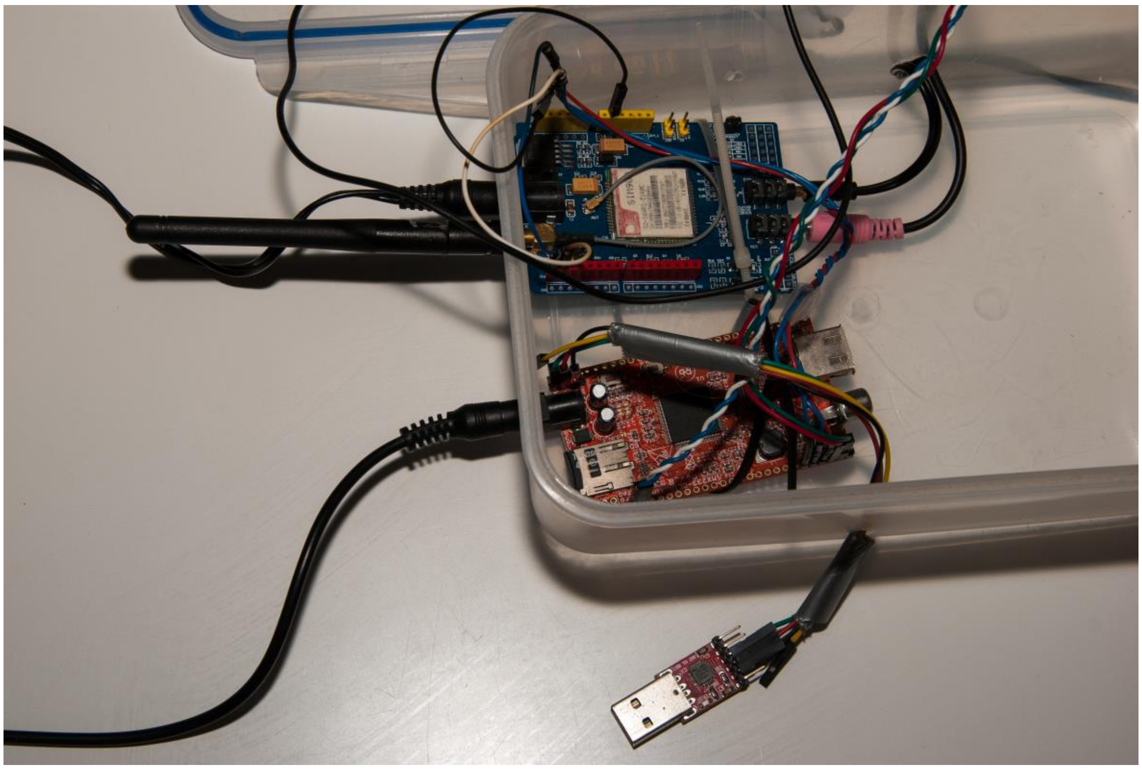


Figure 12 - The proof of concept prototype enclosure, with GSM hardware development kit board (on top), and the Olinuxino main CPU board below it, with the debug serial port coming out from the side of the prototype enclosure.

After plugging in both power supplies, the board containing the CPU will boot on its own, and the GSM board requires a push of a button on the development board in order to continue booting the GSM modem. After both devices have fully powered up, the CPU board has initiated a successful communication with the GSM board when the LED is constantly on. At this point, when pressing the button on the user panel, a call is initiated to a pre-determined phone number. If instead, the LED is blinking, it means that there is an incoming call, which can be answered by the press of the button.

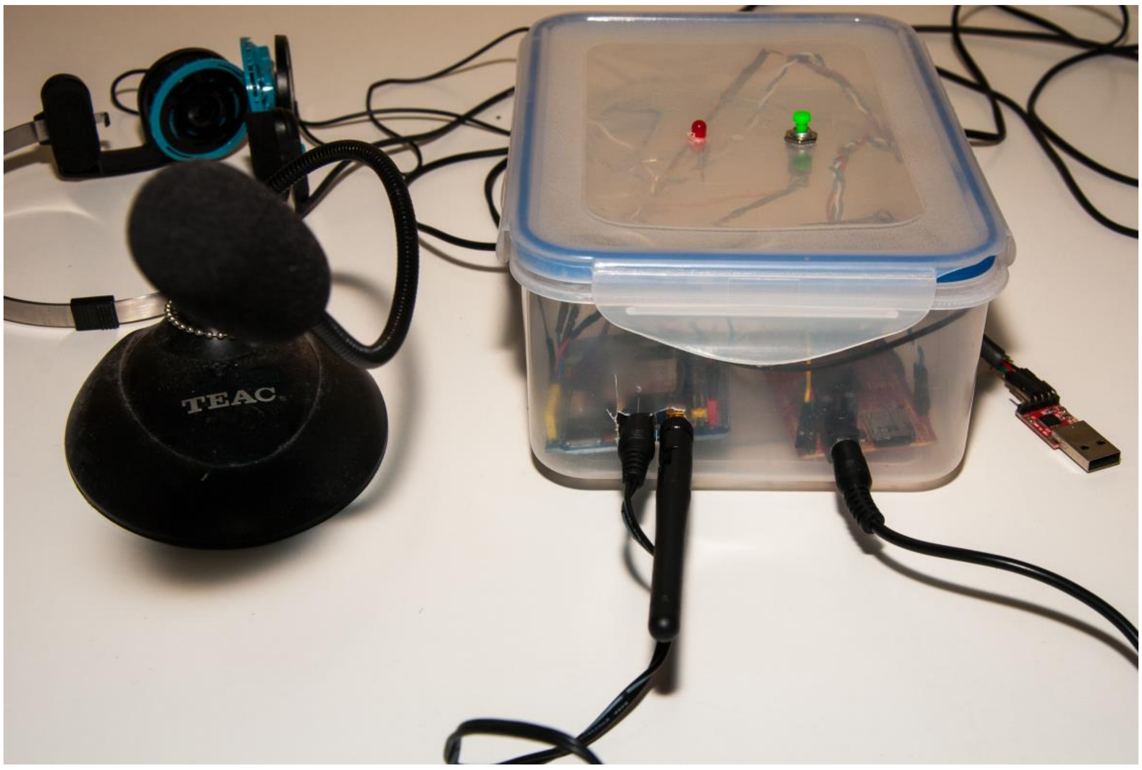


Figure 13 - The final proof of concept setup. On the left side the microphone and headphones for verifying the GSM call. The user panel can be seen on to, with the button to answer and initiate a call, and the LED as an indicator for user feedback.

The GSM development board has two separate connectors for audio devices, one for a microphone and one for headphones. These are directly connected to an audio analog-to-digital converter on the GSM modem. Using the microphone and headphones, it is possible to verify that the phone call is actually initiated successfully.

7 Conclusions

7.1 Proof of Concept Results

The proof of concept was able to achieve the confidence that this idea works. During the execution of the proof of concept, it was shown to be possible to initiate a call to the pre-determined phone number promptly after pressing the button on the user panel, with the LED indicating that the call is underway. It was also proved that an incoming call is being notified by lighting the LED, and the call could be answered by the press of the button on the front panel.

However, a limitation was discovered in the current form of the Python application, that between any incoming or outgoing call, the devices must be reset.

7.2 Hardware Design Failures and Lessons Learned

The PCB for the end device was designed and sent to “Seeedstudio”, the PCB printing company, for fabrication. Following their review of the project design, they responded that it is not possible to manufacture the PCB in their facility. The reason for this was that the distance between traces was too small for fabrication.

When starting the PCB layout design, it is possible to change the set of rules for the layout of traces, pads and nets. The physical width of the traces on the board, as well as their distance between them can be defined in the design rules. The meaning of these rules is that it is not physically possible to plot the traces and other physical PCB layout elements too close to other physical elements, as the rules define. Each manufacturer has its own requirements for the minimum and maximum dimensions for each rule, depending on the capabilities of its manufacturing equipment. In this project, these requirements were overlooked in favor of getting a smaller PCB area size, and thus, an error was made that resulted in a non-printable PCB.

7.3 Python Infrastructure Design Failure and Lessons Learned

Although the Python application is properly communicating with the GSM chipset, and makes it possible to start and receive calls through the user interface, in retrospective, not enough time was used for the overall planning of the Python application and interaction between its modules. In the first place, it would have been important to fully review current and future requirements, and based on that, to make a clear model that includes interactions between modules and the attributes of each module. The current implementation was made without a proper planned design before starting to write the code, but instead with some basic features in mind.

The current Python application works for both initiating a call to pre-determined number, as well as answering an incoming call. This is sufficient in order to complete the proof of concept. However, after each time a phone action is finished, the user must reset both devices. This happens because the Python application architecture was not well planned before starting to write the code. The reason is that the abstraction of the states of the modem in the Python application are not handled well, and the application is not stable because AT commands are being interpreted while ignoring the current state of the modem.

8 Improvements for the Proof of Concept and Potential Future Versions

The proof of concept prototype worked, and it was possible to make phone calls through the device, as well as answer incoming calls. However, there are several missing features that are required for the next version. In the current version, there is only a single button that initiates a call to the preprogrammed phone number, and the same button is used to answer calls as well. The device needs a better keypad, one that has at least one button per each of the ten numbers. This way, the user could input the number to call. In addition, some action buttons need to be added, such as a button for initiating and answering a call, and a button for ending or rejecting a call.

In a similar way, the screen requires an update as well. The screen must be able to show at least 10 to 15 characters, for the user to be able to view the outgoing and incoming phone number.

In the proof of concept setup, the prototype boards required external power supplies. In the next version, there should be a battery charging circuit and a battery to provide power to the device.

The current Python architecture is mostly designed for a simple proof of concept scenario, where either a call is being made or received, with the only user interface being a single button and a single LED. Whenever a new requirement comes up, changes are required system-wide in all the modules, and some significant rewriting is needed, so the Python architecture should be planned well in a flow chart before continuing to the next stage of the proof of concept, or the product itself.

Sources

1. Stasiak, Maciej. Modelling and Dimensioning of Mobile Networks : from GSM to LTE. Chichester, West Sussex, U.K: Wiley, 2011.
2. Godse, A.P. Microprocessor, Microcontroller & Applications. Technical Publications, Pune, 2008.
3. Heathcote, P. M. A' Level Computing. Ipswich: Payne-Gallway Publishers, 2000.
4. Clements, Alan. Principles of Computer Hardware. Oxford New York: Oxford University Press, 2006.
5. Norton, Peter, and Scott H. Clark. Peter Norton's New Inside the PC. Indianapolis, Ind: Sams, 2002.
6. Unnikrishnan, Srija, Sunil Surve, and Deepak Bhoir. Advances in Computing, Communication, and Control: Third International Conference, ICAC3 2013, Mumbai, India, January 18-19, 2013. Proceedings. Berlin New York: Springer, 2013.
7. Brooks, Charles J. A+ : Training Guide. Indianapolis, Ind: Que Certification, 2004.
8. Bai, Ying. The Windows Serial Port Programming Handbook. Florida: CRC Press, 2004.
9. Cowley, John. Communications and networking : an introduction. New York London: Springer, 2007.
10. Ciccarelli, Patrick. Networking Basics. New York: Wiley, 2013.
11. Simmonds, Chris. Mastering Embedded Linux Programming: Unleash the Full Potential of Embedded Linux. Birmingham, UK: Packt Publishing, 2017.
12. Feller, Joseph. Perspectives on Free and Open Source Software. Cambridge, Mass: MIT Press, 2005.

13. Manko, Howard H. Soldering Handbook for Printed Circuits and Surface Mounting. New York: Van Nostrand Reinhold, 1995.

14. Robertson, Christopher T. Printed Circuit Board Designer's Reference: Basics. Upper Saddle River, NJ: Prentice Hall Professional Technical Reference, 2004.

Instructions for Building Kernel 4.4.1 for Olinuxino

Download cross compiler package. Unpack package, and move to a place under a system PATH:

```
wget https://launchpad.net/gcc-arm-embedded/5.0/5-2015-q4-major/+download/gcc-arm-none-eabi-5_2-2015q4-20151219-linux.tar.bz2
```

```
tar jxf gcc-arm-none-eabi-5_2-2015q4-20151219-linux.tar.bz2
```

```
sudo cp -r gcc-arm-none-eabi-5_2-2015q4/* /usr/local/
```

Make sure the path /usr/local/bin exists in the system path environment variable:

```
printenv PATH
```

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
```

Next step is to git clone the source code of the kernel that we will want to compile later:

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
```

And checkout the corresponding kernel version. We should be good with the latest “stable” – let’s not stay with a release candidate version but rather checkout the last stable tag:

```
git checkout v4.4.1
```

For compiling the Linux kernel, we will need to decide what of the massive amount of kernel modules we will want to compile into our kernel. First we will make the base configuration file that will be used to parse what will be compiled and how:

```
make ARCH=arm CROSS_COMPILE=arm-none-eabi- mxs_defconfig
```

We can also further run an interactive shell application if we are not satisfied with the defaults and to include or exclude additional modules:

```
make ARCH=arm CROSS_COMPILE=arm-none-eabi- menuconfig
```

There is not much that need customizing in the kernel building process. However there is an important step related to image creation. As a part of the bootloader integration with the kernel, the following options under “Boot options” menu need to be included as a part of the kernel:

```
[*] Use appended device tree blob to zImage (EXPERIMENTAL)
```

```
[*] Supplement the appended DTB with traditional ATAG information
```

And below that there is field called “Default kernel command string”, where the following line must be entered:

```
console=ttyAMA0,115200 root=/dev/mmcblk0p2 rw rootwait
```

After that I quit the tool, and continued onto compiling the kernel:

```
make ARCH=arm CROSS_COMPILE=arm-none-eabi- zImage modules
```

Now the compilation takes some time, at least some minutes. In the end I would get something like this:

```
Kernel: arch/arm/boot/zImage is ready
```

Next thing we need to do is to make a device tree blob .dbt file. This file is a system specific file that maps the kind of “static”, non-discoverable hardware for kernel use:

```
make ARCH=arm CROSS_COMPILE=arm-none-eabi- imx23-olinuxino.dtb
```

One last thing to do for the kernel is to merge the kernel and the .dtb file together:

```
cat arch/arm/boot/zImage arch/arm/boot/dts/imx23-olinuxino.dtb > arch/arm/boot/zImage_dtb
```

Instructions for Setting Up the Bootloader

A usual way is to use u-boot, which is an open source, widely supported software with several features. Freescale also offers a simple bootloader called bootlet, which is very CPU specific but takes care of the required to boot into the Linux kernel. The bootlet is simple code running on the CPU that takes care of getting the CPU and it’s peripherals to a state that the OS image can be loaded. We will need to compile the bootlet against our kernel headers. We will also need elftosb2 (elf to bootstream) tool, which we will download as a part of the private repository.

We will start by using git to check out a private repository (by a Freescale employee) that has a useful patch for the bootlet, as well as the tool that converts from ELF to bootstream.

```
git clone https://github.com/koliqi/imx23-olinuxino
```

Next, let’s go to the directory of elftosb-0.3, and make the elftosb2 tool appear under the path that the compiler would be looking it for:

```
sudo ln -s `pwd`/elftosb2 /usr/sbin/
```

You can make sure that the compiler sees it correctly by doing:

```
which elftosb2
```

```
/usr/sbin/elftosb2
```

Next step is to take the source code for the bootlets and compile against the kernel headers. A specific version of the source code is provided as an archive as a part of the private repository. We will need to unarchive it, and apply a board specific patch:

```
tar xvzf imx-bootlets-src-10.05.02.tar.gz
```

```
cd imx-bootlets-src-10.05.02/

patch -p1 < ../imx23_olinuxino_bootlets.patch
```

Next thing is that we will need to have the zImage in the directory with the source code. We will make a link to our file that we generated above after the compilation of the kernel itself:

```
ln -s ../../kernel/linux-stable/arch/arm/boot/zImage_dtb ./zImage
```

Now we can finally start the compilation:

```
make CROSS_COMPILE=arm-none-eabi-  clean

make CROSS_COMPILE=arm-none-eabi-
```

Finally the build process was outputting some prints about boot section, and the final line being:

```
To install bootstream onto SD/MMC card, type: sudo dd if=sd_mmc_bootstream.raw of=/dev/sdXY where X is the correct letter for your sd or mmc device (to check, do a ls /dev/sd*) and Y is the partition number for the bootstream
```

The final output file for me is around 4MB in size (depends on how many kernel modules were compiled into the kernel).

Instructions for Creating a Root File System

We will start by creating a local directory that we will use to temporarily hold the target rootfs. We will download the base Debian system into it, using a tool called debootstrap. It will download the base system packages for a target system inside another computer:

```
sudo debootstrap --verbose --arch armel --variant=minbase --foreign jessie /home/karrister/imx23_project/sd-image/rootfs/
```

This command actually installs a basic Debian system on the specified directory, inside a running system. This way we will be able to make the basic Linux system for our device. After some time of downloading packages, the debootstrap is finished. Next we can change the root directory of the host system locally on one shell window to the directory of the mount on the SD card. The point of this is to move working from a different root, with different system binaries as well. Thus, we will need an emulator of the target system to be able to change root to one with binaries from different CPU architecture. We can use emulator called QEMU. We will need to download QEMU and architecture specific binaries to emulate the imx233 CPU environment, including the base qemu packet, as well as the binfmt support which includes a script to register interpreters with the binfmt kernel module (which needs to be included in the initial kernel build) for making the arm executables work with QEMU:

```
sudo apt-get install qemu-user-static
```

```
sudo apt-get install binfmt-support
```

After we have the support for interpretor registration for binfmt, we'll need load the binfmt kernel module in the host system to be able to proceed with the chroot with the “foreign” binaries. We will also need to copy a QEMU binary into the rootfs:

```
sudo modprobe binfmt_misc
```

```
sudo cp /usr/bin/qemu-arm-static /home/karrister/imx23_project/sd-image/rootfs/usr/bin
```

This is done since the binfmt support package has by default registered the emulator binary to be under /usr/bin/qemu-arm-static for binaries of arm type. The command line option “update-binfmts --display” can be used to find the path system is looking for this binary. Next we will need to mount virtual filesystem devpts and proc, to make sure we will have no compatibility issues of running binaries that would need to access these device nodes:

```
sudo mkdir /home/karrister/imx23_project/sd-image/rootfs/dev/pts

sudo mount -t devpts devpts /home/karrister/imx23_project/sd-image/rootfs/dev/pts

sudo mount -t proc proc /home/karrister/imx23_project/sd-image/rootfs/proc
```

Now we have set up the emulator successfully and can continue with the configuration of rootfs by changing the host root into the SD card root file system partition:

```
sudo chroot /home/karrister/imx23_project/sd-image/rootfs/
```

Next we should continue the debootstrap setup, meaning in a cross-arch setup continuing into the second stage. The meaning is that during first stage all the packages and files are downloaded, but because the downloaded binaries are of a foreign CPU architecture, we must have first setup the emulator and done chroot correctly, before continuing with the second stage, which will run some scripts to continue with setting up the rootfs:

```
/debootstrap/debootstrap --second-stage
```

Eventually, after everything is finished we should get the following print:

```
I: Base system installed successfully.

I have no name!@imx233-gsm:/#
```

Now we have a basic rootfs system working, but we will still need to continue with the configuration finalization. Let’s start by setting the default language (EN/US):

```
export LANG=C
```

Let’s set a password for root:

```
passwd root
```

Now we should start downloading and setting up some packages. Let’s first setup the list of sources to look packages. We will need to add some package sources to the file /etc/apt/sources.list. However, the problem is that it seems that we don’t even have vi/vim or any text editing tool on this root system! After I listed the available binaries under /bin, I thought maybe I could use echo to write to this file:

```
echo deb http://ftp.uk.debian.org/debian jessie main contrib non-free >> /etc/apt/sources.list

echo deb-src http://ftp.uk.debian.org/debian jessie main contrib non-free >> /etc/apt/sources.list

echo deb http://ftp.uk.debian.org/debian jessie-updates main contrib non-free >> /etc/apt/sources.list

echo deb-src http://ftp.uk.debian.org/debian jessie-updates main contrib non-free >> /etc/apt/sources.list

echo deb http://security.debian.org/debian-security jessie/updates main contrib non-free >> /etc/apt/sources.list

echo deb-src http://security.debian.org/debian-security jessie/updates main contrib non-free >> /etc/apt/sources.list
```

And it worked! And now also we need to update the indexing of packages:

```
apt-get update
```

Next up, let’s install some basic packages using apt-get, to widen our apt-get support (apt-utils) and enable better setup script interface to user (dialog), as well as enable localization support:

```
apt-get install apt-utils dialog locales
```

```
dpkg-reconfigure locales
```

Now we will install the most important packages for me:

```
apt-get install vim nano less cron man python python-serial
```

Now we should be done. We can write exit, to exit the virtual console, and get back to the host system console

Instructions for Creating a Bootable Linux SD Card

First we will need a SD card of at least 4GB of size. We will partition it to two partitions, one for boot and one for the root file system. We will start by using fdisk to delete all the current partitions of the current SD card, and creating primary partition of 32MB, and assigning the rest to the second primary partition. What is special is that we must write the boot partition with a type of “OnTrack DM6 Aux” (0x53 in hexadecimal). At the end we will write the configuration and exit fdisk. Next step is to format the rootfs partition into Linux ext2 type (depending on the device node of the SD card media in the system):

```
sudo mkfs.ext2 /dev/sdc2
```

Next thing is to copy both kernel and rootfs (which we already created before) onto a SD card. So first I wrote the kernel image file onto the first partition:

```
sudo dd if=sd_mmc_bootstream.raw of=/dev/sdc1
```

where sdc is the device name for my sd card. You could find yours by e.g. following dmesg messages, running fdisk etc. You should make sure of the device name, and not blindly use sdc. In the worst case, it could result in losing all the data on your computer! Next thing is to also copy the rootfs. Let’s first mount the second partition:

```
sudo mount -t auto /dev/sdc2 /mnt/sd-rootfs/
```

Next up is to copy the loadable kernel modules into the rootfs. We will do it first locally, and then copy the whole rootfs into the mounted partition on the SD card. Let’s go back to the dir where we compiled kernel, and copy the loadable modules:

```
sudo make -j4 ARCH=arm CROSS_COMPILE=arm-none-eabi-
INSTALL_MOD_PATH=/home/karrister/imx23_project/sd-image/rootfs  modules_install
```

Finally, we will copy the rootfs into the SD card’s rootfs partition:

```
sudo cp -R /home/karrister/imx23_project/sd-image/rootfs/* /mnt/sd-rootfs/
```

Make sure all files are synced, and unmount the rootfs:

```
sync

sudo umount /dev/sdc2
```

The address of the git repository containing the development work for PCB design in KiCAD and for Python scripts developed that control the hardware:

https://github.com/karrister/imx233_gsm