Final Master Thesis

Improving the BeagleBone board with embedded Ubuntu, enhanced GPMC driver and Python for communication and graphical prototypes

By

RUBÉN GONZÁLEZ MUÑOZ

Directed by

MANUEL M. DOMINGUEZ PUMAR

FINAL MASTER THESIS 30 ECTS,
JULY 2015, ELECTRICAL AND ELECTRONICS ENGINEERING

Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Abstract

BeagleBone is a low price, small size Linux embedded microcomputer with a full set of I/O pins and processing power for real-time applications, also expandable with cape pluggable boards.

The current work has been focused on improving the performance of this board. In this case, the BeagleBone comes with a pre-installed Angstrom OS and with a cape board using a particular software "overlay" and applications. Due to a lack of support, this pre-installed OS has been replaced by Ubuntu. As a consequence, the cape software and applications need to be adapted.

Another necessity that emerges from the stated changes is to improve the communications through a GPMC interface. The depicted driver has been built for the new system as well as synchronous variants, also developed and tested.

Finally, a set of applications in Python using the cape functionalities has been developed. Some extra graphical features have been included as example.

Contents

Abstract 5
List of figures 7
List of tables 9
2 Introduction 11
 2.1 Work done before 12
 2.2 Scope 12
3 Internet of Things 13
 3.1 Internet of Things 13
 3.2 Embedded systems 13
 3.3 Single Board Computers 14
  3.3.1 Arduino 15
  3.3.2 Raspberry Pi 16
  3.3.3 Other boards 17
4 BeagleBone 20
 4.1 Description 20
 4.2 Device tree 21
  4.2.1 Device Tree Blob 22
  4.2.2 Device Tree Overlay 23
 4.3 Laboratory Cape TT01v1 24
5 Methodologies and development 27
 5.1 Brief 27
 5.2 Setting up the development environment 27
  5.2.1 Virtual machines 28
  5.2.2 Cross compiling 32
 5.3 Ubuntu embedded 34
  5.3.1 From Angstrom to Ubuntu 34
  5.3.2 Build for Ubuntu 3.8 RCN Kernel 36
  5.3.3 Alternative builds of Ubuntu. Kernel 3.14 in ARMhf 41
  5.3.4 Some improvements 45
 5.4 GPMC driver 46
  5.4.1 First approach to the cross-compilation 46
  5.4.2 Compiling the simple module driver beginning 47
  5.4.3 General Purpose Memory Controller GPMC 50

5.4.4 Asynchronous GPMC 51
5.4.5 Synchronous GPMC 56
5.4.6 Burst GPMC 59
5.5 Cape programs 63
5.5.1 Previous work: regenerate the cape EPROM contents 63
5.5.2 Building the cape programs 64
5.5.3 Inspection of the cape programs behavior in the new environment 65
5.5.4 Integrating both cape-bone-TT01v1 and cape-bone-iio capes 66
5.5.5 Adapting the cape programs 69
5.6 Python 69
5.6.1 Introducing Python 69
5.6.2 Adafruit Python modules for BeagleBone 70
5.6.3 Installing Python and Python modules 71
5.6.4 Writing TT01 Cape programs to Python. Adafruit modules 71
5.6.5 Introducing graphics in the BeagleBone 82
6. Results 91
6.1. Ubuntu system 91
6.1.1 Getting into Embedded Ubuntu 91
6.1.2 Startup process 92
6.2. GPMC 93
6.2.1 Testing GPMC asynchronous module 94
6.2.2 Testing GPMC synchronous module 94
6.2.3 Testing GPMC synchronous burst module 95
6.3. Cape verification 97
6.3.1 Applications verification 98
6.4. Python programs 101
7. Conclusions and future work 103
8. Bibliography 104
9. Glossary 106

List of figures

Fig 3.1: Microprocessor family of ARM 14
Fig 3.2: Arduino DUE (left) and Arduino UNO (Right) 15
Fig 3.3: Raspberry Pi B+ (left) and Raspberry Pi 2 B 1.1 (right) models 16
Fig 3.4: BeagleBone Black board 17
Fig 3.5: Intel Galileo Gen 2 board 18
Fig 3.6: Odroid X2 (left) and Odroid U2 (right) boards 18
Fig 4.1: BeagleBone 21
Fig 4.2: Peripherals in ARM architecture 21
Fig 4.3: Device tree compiler schema 23
Fig 4.4: Cape TT01v1 top view 24
Fig 5.1: Left: connected scenario. Right: connectionless scenario 27
Fig 5.2: Host PC inner system relationships 30
Fig 5.3: Moba Xterm interface 31
Fig 5.4: GPMC to 16 bit address and data multiplexed device 50
Fig 5.5: Interface and internals of the memory DE2 FPGA board 51
Fig 5.6: Asynchronous read operation 53
Fig 5.7: Asynchronous write operation 54
Fig 5.8: Synchronous read operation 57
Fig 5.9: Synchronous write operation 58
Fig 5.10: Burst read operation 60
Fig 5.11: Burst write operation 61
Fig 5.12: Signals generated by the rotating encoder depending on the rotating direction 73
Fig 5.13: SPI master & slave communication scheme 77
Fig 5.14: Virtual shift register in SPI master & slave model 78
Fig 5.15: interconnection schema in an I2C system 79
Fig 5.16: VNC Interconnection scheme 83
Fig 5.17: New VNC viewer tab in Moba Xterm 86
Fig 5.18: Common X applications thrown in BeagleBone viewed using VNC viewer 86
Fig 5.19: Basic structure of Tkinter application 87
Fig 5.20: Position and direction of the XY axis in a Tkinter canvas 87
Fig 5.21: Accelerometer with the cape at a neutral X position 90
Fig 5.22: Accelerometer bent in the X direction and opposite to X direction, respectively 90

Fig 6.1: TT01 cape board CN-2 connector pinout..93
Fig 6.2: GPMC asynchronous write chronogram..94
Fig 6.3: GPMC synchronous write chronogram...95
Fig 6.4: GPMC synchronous burst write chronogram...96
Fig 6.5: LED LD1 in cape TT01v1 switched on, corresponding to app GPIO1 98
Fig 6.6: LEDs LD1, LD2 and LD3 in cape TT01v1 switched on, corresponding to app GPIO2 99
Fig 6.7: LEDs in LED matrix in cape TT01v1 switched on, corresponding to app LEDMATRIX..... 101

List of tables

Table 3.1: Comparison between embedded and PC technologies 14
Table 3.2: Main features of Arduino Uno and Arduino Due boards 15
Table 3.3: Main features of Raspberry Pi computers 17
Table 5.1: Current version of TI Sitara SDK for BeagleBone and BeagleBone Black 32
Table 5.2: Description of the GPMC interface 50
Table 5.3: Asynchronous read selected timings 54
Table 5.4: Asynchronous write selected timings 55
Table 5.5: Synchronous write selected timings 57
Table 5.6: Synchronous write selected timings 58
Table 5.7: New timings in a burst read operation 59
Table 5.8: Synchronous burst read selected timings 60
Table 5.9: Synchronous burst write selected timings 62
Table 5.10: Summary of results in cape program execution with the current cape overlay 66
Table 6.1: Relevant configuration parameters in the asynchronous GPMC 94
Table 6.2: Relevant configuration parameters in the synchronous GPMC 95
Table 6.3: Relevant configuration parameters in the synchronous GPMC 96

2 Introduction

The BeagleBone board is the focus of the current work. Because it is a proper system on which to develop particular projects as well as commercial projects for the industry, it is also an ideal system to teach and learn.

The BeagleBone is currently used for learning purposes in the Electronics Engineering department of the UPC. This board is used in an optative degree subject whose target is focused on embedded and real time operating systems RTOS. Its name is "**Digital Systems Using Embedded Linux**" (DSX). Most of the work done in this subject takes place in the laboratory with the students programming this board.

Before this work started, two principal necessities were needed to be covered for this subject:

- The operating system had a caducity date. The Angstrom system does not have support anymore although compatibility is granted in other projects such as Yocto. New shipments come with Debian embedded instead.
- One driver, the General Purpose Memory Controller GPMC, needed to be improved. Part of the subject is based on learning how to develop drivers, but the GPMC one supplied had no compatibility with synchronous systems.

Both necessities have become the main focuses of the current work, covering the 60% of the overall work.

In order to find a solution for the statements above, some options are purposed. For the problem with the OS, it must be evaluated the option of installing a new one with a good support level granted. Some options in the embedded world covering this restriction are Linux like distributions such as Ubuntu, Debian, Arch or even Android. There is an important volume of documentation in internet about how to proceed with this issue.

The question of the driver needs to be tacked in a different way. The information about how to program a GPMC interface is all gathered in the technical reference of the processor, and so it is the only document that needs to be used. There, several options such as timings and other parameters are introduced, as well as configuration modes such as asynchronous, synchronous or burst. The problem is focused on designing and programming a proper configuration for the driver.

What else?

In addition to the work mentioned, some extra tasks have been done. The first of them, and not less important, has been ensuring the compatibility of the software. A set of applications specifically designed to the deprecated OS need to be tested and adapted, if necessary. These applications rely on a cape board which comes coupled to the BeagleBone through a set of expansion pins. So, the driver handling this board, also called overlay, needs to be checked and modified, if necessary.

Finally, an optional work has been done in order to demonstrate the capacity of the board from another point of view. A set of simple applications handling the cape board has been designed, implemented and tested using Python as programming language. It proves the capacity of the board and this language to

develop good and fast application prototypes without relying on a compiled language such as C. This work ends up with the development of a graphical application also using the advantages that the Python modules provide to the developer. These two last tasks together compose the resting 40% of the overall work.

2.1 Work done before

Before this project, a solid basis to work was stablished. This project can be seen partially as the continuation of the project "*Design of a dedicated cape board for an embedded system lab course using BeagleBone*" of the author Raúl Pérez López [2.1]. In this project, the cape board TT01v1 was designed and implemented. This cape worked properly under the Angstrom system.

Moreover, the GPMC driver had an asynchronous version which also worked under the old OS. This driver was designed to handle write and read operations on a memory of a FPGA of the laboratory. The current version of this driver was developed by the professor Manuel Domínguez Pumar, which also imparts the subject stated before.

2.2 Scope

The main objectives of the present project are the following:

- Installing an Ubuntu distribution into the BeagleBone providing a reliable alternative to the deprecated Angstrom OS.
- Adapting the GPMC asynchronous driver to the new OS.
- Adapting the cape software to the new OS, as well as the applications which use it expecting to behave as before the changes
- Provide additional versions for the GPMC driver based in synchronous or burst behavior, if possible
- Play with the cape using Python and basic I/O handling libraries, as well as developing simple but effective applications with and without graphical interface.

It is important to highlight that this project does not include hardware design or development. Every improvement or development is done in the software concerns to the system or the applications in the BeagleBone. No laboratory specific material has been required except a personal computer and the BeagleBone plus the cape board. In order to test the GPMC driver, the aid of a logical analyzer has been required.

3 Internet of Things

3.1 Internet of Things

The concept "Internet of Things" (IoT) first appeared in 1999, when the English entrepreneur Kevin Ashton used it to describe it as the network of physical objects that are embedded with electronic technology such software, sensors and connections. Those connections provide an advanced level of connectivity between devices and systems achieving a greater value exchanging data with other devices, manufacturers or operators.

A wide variety of devices compose the physical layer of IoT, from the simplest ones to the most complex. Monitoring implants or biochip transponders are examples of embedded devices composed of sensors and transponders that work gathering data and sending it to later process it.

More sophisticated devices are also considered in the category of *embedded devices*. For instance, a computer aiding the management of the doors, the acoustic signals and the information in the panels of a railway is also included. This type of devices can be seen as a flavor of general purpose computers whose hardware and software has been modified to adapt its functionality to a particular purpose.

3.2 Embedded systems

But, what can be understood under the name of *embedded device*?

A simple definition can be the following: an embedded device is a computing system embedded within an electronic device [3.1]. Being more specific, these two following points should be met:

- It is almost any computing system except desktop computers, laptops or servers
- The embedded device takes advantage of its specific application to optimize its design

Nowadays, a large group of embedded systems is based in microprocessor architecture. The account of new microprocessors into embedded systems is over the 99%. Most of them are applied in the fields of consumer electronics, vehicle control systems, medical equipment and sensor networks [3.2].

In terms of the system, what makes the main difference between a desktop based computer and an embedded system is its microprocessor and chipset architecture. Whereas a PC normally has an Intel x86/x64 CISC based processor, most of the market of embedded devices based in microprocessor has chosen to embed an ARM RISC based processor. The following table summarizes the main differences between both:

	Embedded	PC
Microprocessor	ARM	x86, x64
Architecture	RISC	CISC
Technology	SoC	CPU + Chipset
Operations	Less capacity	More capacity
Price	Reduced	Elevated
Electrical purchase	Reduced	Elevated
Purpose	**Specific Tasks**	**General Purpose Tasks**

Table 3.1: Comparison between embedded and PC technologies

3.3 Single Board Computers

There is a particular branch in the group of embedded devices which could be allocated between the embedded and desktop computers world. These devices are known as *single board computers* or *microcomputers*, and are capable to perform different tasks on the same low cost small sized device.

Although most of these computers are used in industrial products, many where conceived for educational purposes. The increasing processing capacity of their microprocessors has made this task easier through the years. Some common examples of educational single board computers are *Arduino*, *Raspberry Pi* and *BeagleBone* boards. As most of the microcomputers, the three of them are based in a microprocessor of the ARM family:

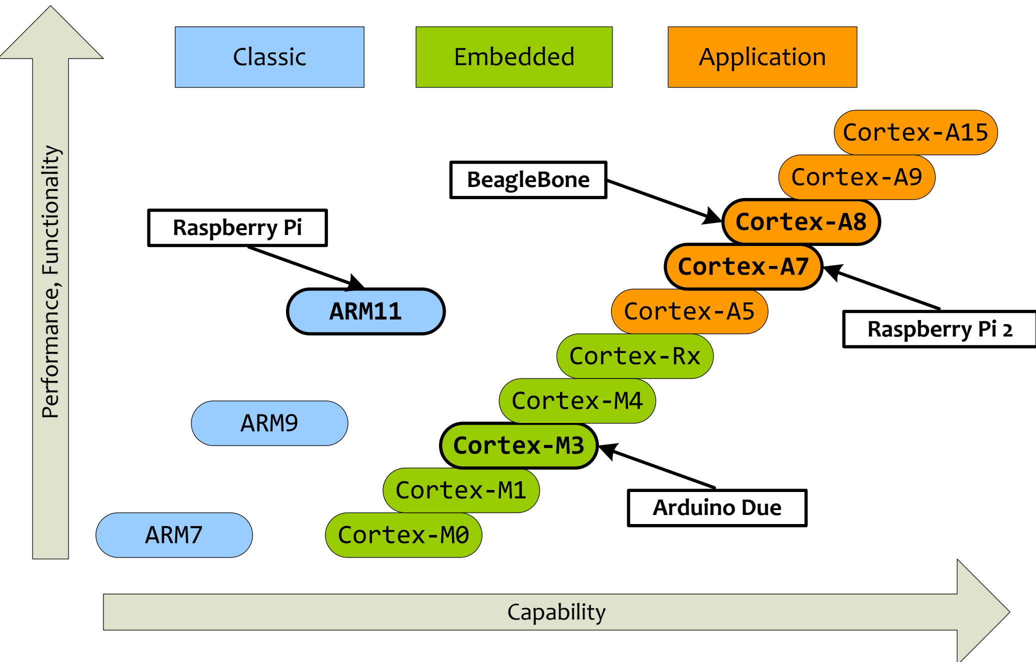


Fig 3.1: Microprocessor family of ARM

The capacity of performing different tasks is provided by its microcomputer based architecture and the capacity of integrating devices in SoC technology. Moreover, those computers frequently include a variety of media and communication interfaces as well as communication protocols available.

In the following sections, a slight overview of the Arduino and Raspberry Pi boards as well as other also popular boards will be given. Arduino can be seen as one of the precursors of the current embedded device developments, whereas the Raspberry Pi has brought the possibility of arriving nowadays to all the user and developer community due to its complete set of capabilities, ease to learn, and low cost. A more complete description of the BeagleBone board will be provided in the next chapter.

3.3.1 Arduino

Arduino is an open source micro-computer aimed to make interactive projects. Projects are designed and built by the Arduino Company , but also by the community of users [3.3].

The development board is based on 8-bit Atmel AVR processors or 32-bit ARM (newest) processors, providing sets of digital and analog I/O pins which can interface with a variety of expansion boards, also called *shields*. In order to program the microcontroller, the Wiring IDE is provided. It is a full IDE based in the Processing project [3.4] which includes support to a simplified version of C and C++.

The project Arduino started at 2005, when a group of students at Interaction Design Institute of Ivera considered too expensive using a BASIC Stamp board (a microcontroller with a specialized BASIC interpreter built into ROM) at a cost of 100$. The name of *Arduino* comes from a bar where some of the students used to meet. Between them there was Massimo Banzi, one of the Arduino founders and student at Ivera.

At present, there exists a wide variety of official Arduino boards and shields. A basic feature comparison is given at the official web of Arduino. For instance, Arduino UNO and Arduino DUE are a representation of the most popular Arduino boards.

Fig 3.2: Arduino DUE (left) and Arduino UNO (Right)

	Release	Processor	Operating/In Voltage	CPU (MHz)	Analog I/O	Digital IO/PWM	EPROM	SRAM	UART
Uno	09/2010	ATmega328	5V/7-12V	16	6/0	14/6	1	2	1
Due	10/2012	AT91SAM3X8E (Cortex-M3)	3.3V/7-12V	84	12/2	54/12	-	96	4

Table 3.2: Main features of Arduino Uno and Arduino Due boards

Nowadays, Arduino is perceived by the community as one the precursors boards for educational purposes. A large variety of amazing user projects based in Arduino can be found in the Internet [3.5]. Widely used in

universities and laboratories, this board has also contributed in a variety of automation projects including this board in third parties products.

3.3.2 Raspberry Pi

The Raspberry Pi is a single-board microcomputer developed by the Raspberry Pi Foundation with the aim of promoting the teaching of basic computing at schools. This board can be plugged into a computer monitor or a TV using an HDMI interface and can handled connecting it to a standard keyboard and mouse. From exploring the internet with a web browser to playing high definition videogames, the Raspberry Pi is designed to reproduce everything a common purpose PC can do [3.6].

Applications in the Raspberry Pi can easily be built and tested using Python, which is the main programming language intended for this board. Other languages such as C, C++, Java and Perl are also feasible to develop applications in the Raspberry Pi.

As well as other current microcomputers, the Raspberry is equipped with a set of generic input and output pins, which can vary depending on the board version. These pins are depicted to interface the Raspberry Pi with a variety of devices which can support several communication protocols: GPIO, UART, SPI, I2C and PWM, amongst others. Devices are easily handled using the Python modules which are explicitly developed for the Raspberry Pi.

The Raspberry Pi is inspired in the BBC Micro educational computer developed in 1981 Acorn Microcomputers. In particular, models A, B and A+ of the Raspberry Pi are references to that computer. At present, most of the success of this platform is due its low price and to the contribution of a vast community of developers around the world.

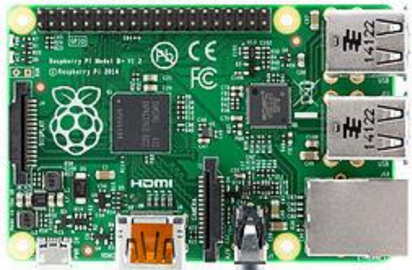


Fig 3.3: Raspberry Pi B+ (left) and Raspberry Pi 2 B 1.1 (right) models

Some of the features of the current Raspberry Pi models in the market, including the most recent Raspberry Pi 2 from February of 2015, are summarized in the following table.

	Release	Soc / Processor	V In	CPU (MHz)	I/O pins	GPIOs	Interfaces	Storage	RAM	USB	Media ifaces	OS
A	02/2012	BCM2835 / ARM11	5V	700, 1 core	40	8	I2C, SPI, UART, PWM	SD	256	1	HDMI, Audio out, Composite video	Raspbian (default). RISC OS, BSD, & other
B	10/2012					8		SD	512	2		
A+	07/2014					8		micro SD	256	4		
B+	11/2014					17		micro SD	512	4		
2	02/2015	BCM2835 / Cortex-A7	5V	900, 4 core	40	17	Idem	-	1GB	4		Idem + Win. 10

Table 3.3: Main features of Raspberry Pi computers

3.3.3 Other boards

The current market of microcomputer boards does not only consist of Arduinos, Raspberries and BeagleBones. From years, many companies and particulars with different purposes have developed this type of hardware with more or less success in their objectives. At present, the world is evolving to a scenario where most of the objects need to be smart, many of them including some type of memories and capacity to process inputs and transmit information [3.7].

BeagleBone Black

The Black model of the BeagleBone is a low-cost powerful board which is the successor of the white model of the BeagleBone. It is provided of a faster processor, which is an ARM Cortex A8 at 1GHz. The RAM memory is also enhanced, providing 512MB DDR3. One important improvement in this board compared to the white model is that it includes a micro HDMI interface, which added to its power makes this board ideal for media applications. This board can be obtained by 60 € approximately [3.8].

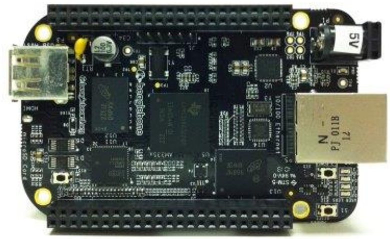


Fig 3.4: BeagleBone Black board

Galileo Gen 2

This board appears as one alternative from using ARM based architectures in embedded systems. The manufacturer, Intel, includes the 32 bit Quark SoC X1000 processor, which is compatible with the set of instructions of the Intel Pentium series. In general, the features in this board, processor and pinout are less powerful compared with other computers such as the BeagleBone or the Cubie Board. Nevertheless, one interesting feature that Galileo Gen provides is the certified compatibility with the Arduino IDE and commercial shields, and includes a set of pins to perform on-board debugging. The board is compatible with

Yocto Linux, VxWorks and even with Windows operating systems, and can be acquired by 140€ approximately [3.9].

Fig 3.5: Intel Galileo Gen 2 board

Cubie Board 2

The Cubie Board 2 is also a high performance low cost microcomputer which is provided of similar features as the BeagleBone Black. It includes approximately the same ports: USB client and host, Ethernet connector, power connector and two important rows of programmable pins, as well as the BeagleBone. It also supports a variety of operating systems such as Android, Ubuntu and other Linux distributions. It can be obtained by 60€ or less [3.10].

Odroid X2 and Odroid U2

The Odroid board is definitely the strongest board currently on the market. By its features, it seems to be a direct competitor of the Raspberry Pi but with a better performance. The processor is a Samsung Exynos4412 Cortex-A9 Quad Core at 1.7 GHz, which is much powerful by far than its rivals. Moreover, the 2GB of RAM memory is much more in this device than in other boards. The board is also provided of a GPU 400 MHz Quad processor, and media interfaces such as HDMI, and audio and microphone jacks. Moreover, a variety of connectivity interfaces such as USBs, one Ethernet interface and a micro SD card slot are provided.

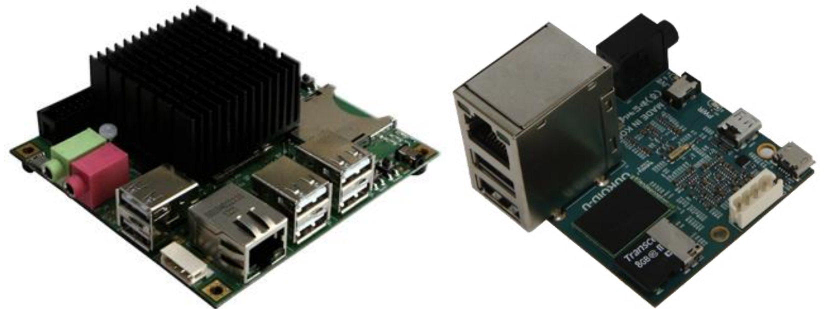


Fig 3.6: Odroid X2 (left) and Odroid U2 (right) boards

The most significant difference between both models X2 and U2 is the size and the number of connections. The model U2 is much smaller, having the surface of half a credit card. Moreover, the number of USB slots have been reduced from 6 to 2, and the pinout raw has been suppressed (although an external module can be attached recovering those pins). Those reductions are traduced to an important decrease of the price. The model X2 can be found by 140€ whereas the U2 has a price of 90€ approximately [3.11].

This has been a brief introduction of some popular micro PC boards in the market today that are perfectly suitable to develop applications in a world of embedded devices. But there is a lot more of micro computers and peripheral devices that can be found to perfectly fit the application it needs to cover. Internet is full of information and comparisons between different boards that makes the current document a short introduction to this subject.

4 BeagleBone

The BeagleBone is a low price, small size Linux embedded microcomputer that connects to the Internet and runs different operating systems such as Angstrom, Ubuntu or Android. It is equipped with a full set of I/O pins and processing power for real-time applications through its ARM Cortex A8 processor. This board can be enhanced with cape pluggable boards allowing extra functionalities.

The following sections in this chapter will describe some of the relevant features of the BeagleBone, as well as the system in which is based to manage the hardware and pluggable boards: the device tree.

4.1 Description

Released in October 2011 with educational purposes, the BeagleBone is equipped with a Sitara ARM Cortex-A8 processor running at 720 MHz. The board was designed by beagleboard.org, a community of developers of Texas Instruments. It was initially priced to 90€ approximately. Nowadays it is difficult to find this board on sale, and so its successor BeagleBone Black is now sold by a more affordable price.

Some relevant technical features of the BeagleBone are the following [4.1]:

- SoC name: AM3358/9
- CPU: ARM Cortex A8, 500MHz (USB powered) or 720 MHz (DC jack powered)
- GPU: PowerVR SGX530 200MHz
- RAM memory: 256 MB DDR2
- USB: 1 standard host, 1 mini client
- Video and audio interfaces: none, can be included through some capes
- Storage: Micro SD card
- Networks: Fast Ethernet interface
- Other interfaces: UART (4), PWM (8), LCD, GPMC, SPI (2), I2C (2), ADC, CAN (2), FTDI, JTAG (USB)
- Power rating: 300-500 mA, 5V
- Power source: mini USB or 2.1 mm x 5.5 mm 5V jack

The BeagleBone is provided by two expansion connectors P9 and P8 that are frequently used to insert cape boards. These connectors are formed by two rows of 23 pins allowing a full set of GPIOs (65) and interfaces for user configuration and use, as depicted above.

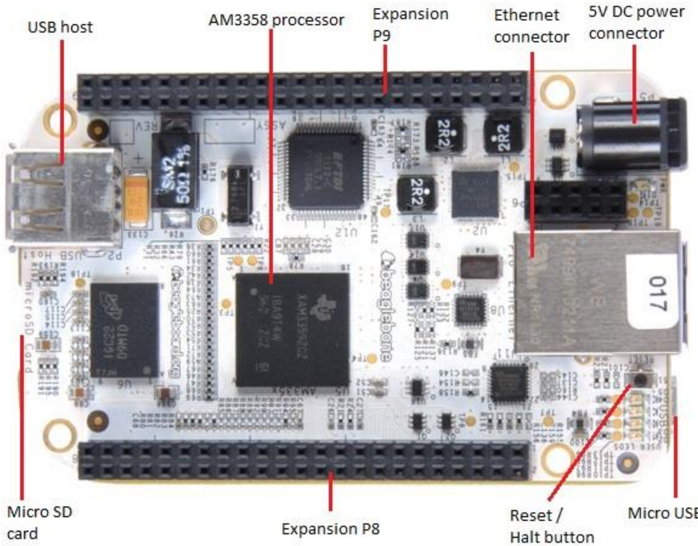


Fig 4.1: BeagleBone

Cape board is the name that an expansion board has in the BeagleBone. The concept is analog to the shields used in Arduino. A cape board can have a variety of peripheral devices that are controlled by the processor. Capes also need to have an EEPROM with particular information in order for the system to recognize them and load the proper hardware description. Capes can be acquired commercially or can be user hand made.

The first shipped BeagleBones included the Angstrom OS. Nevertheless, the processor architecture is also compatible with other embedded OS such as Debian, Ubuntu, Arch or Android. The latest shipped boards of BeagleBone Black come with Debian due to the current lack of support to the Angstrom system.

4.2 Device tree

In March of 2011 Linus Torvalds decided that there were not going to be more ARM device drivers in the Linux kernel anymore [4.2]. A huge amount of ARM board details was accumulated in the Linux kernel mainline. A different board file needed to be stored for each ARM board.

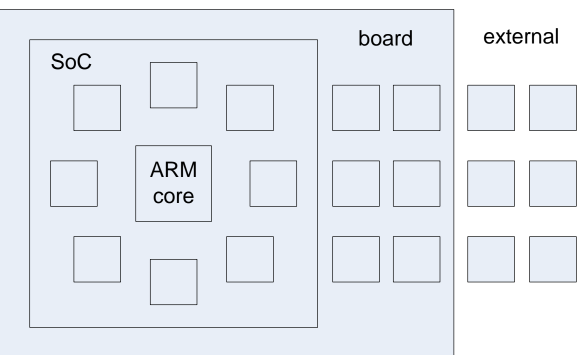


Fig 4.2: Peripherals in ARM architecture

In order to solve this issue, device trees were introduced. Now, data describing these boards is structured in nodes and passed to the kernel at boot time. The following sections will provide a brief review of these data structures and a relationship with the *cape overlays*.

4.2.1 Device Tree Blob

As depicted, the device tree is a data structure that describes hardware instead of hard coding every detail of a device into a driver. The file that contains this information is located in a particular binary which is called Device Tree Blob (DTB). Parameters of the hardware are described in a tree structure that is passed to the operating system at boot time.

The data structure is formed by a tree composed of nodes and properties. Nodes can be formed by properties and other nodes. Properties are tuples formed by a name and a value holding any type of data.

In summary, the bootloader loads two binaries: the kernel image and the DTB [4.3]:

- The kernel image remains as *uImage* or *zImage*
- The DTB is located in /boot, having one in particular for each board. The bootloader passes the DTB to the kernel to adjust it with memory information, kernel command line, and other information.
- There is no more machine type required in the kernel source.

A device tree source file has the following structure:

```
/ {
	node@0 {						//node name and unit address
		string_property = “value”;			//pair property-value
		string_list_property = “value1”, “value2”, …;
		byte_data_property = [0x01 0x02 …];		//byte string

		child_node@0 {
			…
			reference = <&node1>			//reference to other node
		}

		child_node@1 {
			…
		};
	};

	node1: node@1 {						//label before node name
		empty_property;
		cell_property = <1 2 3 4>;			//cells

		child_node@0 {
		}
	};
};
```

This hardware description is written on a file called Device Tree Source (DTS). On ARM, all DTS files are located in arch/arm/boot/dts in the kernel sources:

- DTS files are used for board-level definitions.
- DTSI files (Device Tree Source Include) are *include* files which contain SoC-level definitions and are included in DTS files or other DTSI using *include* directives.
- The inclusion works by overlaying the tree of the including file over the tree of the included file.

The Device Tree Compiler is a tool which compiles these sources obtaining the binary file

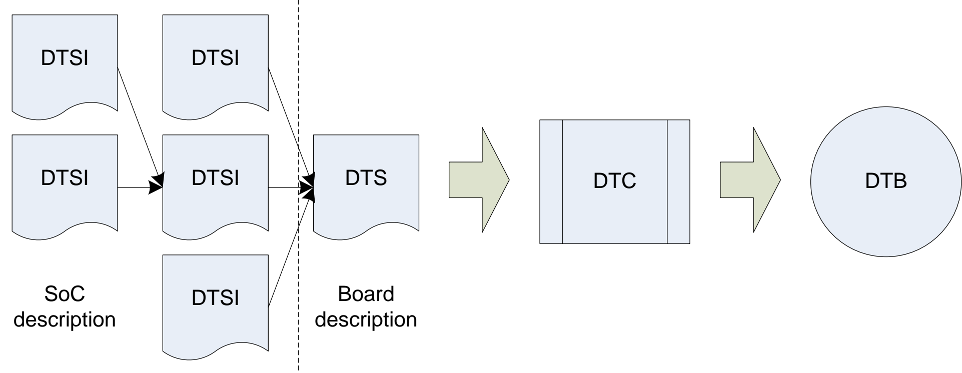


Fig 4.3: Device tree compiler schema

The Device Tree Blob (DTB) is the binary file which is produced by the DTC describing the complete system hardware. It is loaded by the bootloader at boot time and parsed by the kernel. The kernel file arch/arm/boot/dts/Makefile lists which DTBs should be generated at build time.

4.2.2 Device Tree Overlay

Whereas the DTB data structure is referred as parsed at boot-time, BeagleBone capes are not static: a different cape set might be connected each time the board boots. The only way to find out what kind of cape is connected is to read the serial EEPROM present on each cape which contains identification information.

The EEPROM in the cape stores different parameters of information such as a user readable name, serial number, part number, revision information and others. The only information that the cape manager requires are part-number and revision [4.4].

The method used to dynamically load the cape definition to the running kernel is called a Device Tree Overlay (DTBO). What a DTBO does is to apply changes to the kernel internal device tree representation and trigger the changes that this action carries. For example, adding a new enabled device node should result in a new device being created, while removing a node removes the device.

DTBO is a general purpose mechanism, which is not tied to any platform. For actual platforms like the BeagleBone, the mechanism must be supplemented by the platform specific logic.

Device tree overlays were introduced in kernel 3.8 and were handled through the cape manager. Unfortunately, the overlay concept in kernels 3.13 and on, as well as the cape manager, disappears. If one needs to use a cape with these kernel branches he/she must modify the original board DTS introducing the hardware description of this cape. Current kernel branches from 4.0 and on retrieve the cape manager and overlay concepts. Nowadays, if it is required to use capes in the BeagleBone, the best option is to use the 3.8 branch. Nevertheless, it seems that soon it will be feasible to do it with the newest kernel branches.

Quoted by Robert C. Nelson on December 9th of 2014:

[...]

If you need capes, you downgrade to 3.8:

sudo apt-get update

sudo apt-get install linux-image-3.8.13-bone68

sudo reboot

[...]

4.3 Laboratory Cape TT01v1

The laboratory cape TT01v1 is designed to work using the cape manager under the kernel 3.8. At boot time, the kernel loads the board DTB, which for the BeagleBone it is normally the DTB file **am335x-bone.dtb**. The overlay for the laboratory cape can be loaded or unloaded dynamically, or loaded at boot time too. The name of the cape overlay is **cape-bone-TT01v1-00A0.dtbo**.

The cape-bone-TT01v1 hardware is a BeagleBone cape aimed to educational purposes. It is equipped with several devices and interfaces which provide a wide set of feasible laboratory experiments. The cape is compatible with both models BeagleBone and BeagleBone Black.

Fig 4.4: Cape TT01v1 top view

The cape is mounted over the BeagleBone board using the two expansion headers P8 and P9. In the following sections, a brief overview of the devices in the cape board is provided.

Cape EEPROM

The EEPROM in the cape is used by the cape manager to physically identify the cape when it is plugged to the BeagleBone. Amongst others, it stores the part number and revision of the cape.

The following are some relevant features of the EEPROM:

- Size of 32 KB
- Page size of 64 bytes
- I2C serial access

- Random and sequential Read access modes
- Operation voltage from 2.5V to 5.5V

Cape LEDs

There is a set of three general purpose LEDs which are bounded to another three GPIOs that are set up as outputs. Some information about them is the following:

- Reference number: HSMG-C170
- Manufacturer: Avago
- Colours: Green, orange and red
- DC forward current voltage: 20 mA

Cape push button

There is an input push button connected to a GPIO which is set as input.

Cape analog input

The cape is able to generate an analog signal using a source voltage at 3.3V and a set of resistors that can be set up as different voltage dividers using the S1 switch. The resulting signal is connected to an analog input in the BeagleBone which is called AIN0. This is the analog channel 0 out of 8 in total.

The BeagleBone has a successive-approximation 12 bits ADC. The reference voltage for the ADC is 1.8V, so the equivalent voltage to the ADC counts lecture is:

- voltage = (number_counts_ADC / (2^12)) * 1.8

The voltage range for the ADC is set between 0V and 1.8V.

Cape rotary encoder

A rotary encoder is also included in the cape. It monitors 2 input GPIO signals whose changes indicate the direction of the encoder. Measuring the interval of signal changes, it is also possible to measure the speed that encoder turns.

Cape LED matrix

The cape is also provided of a LED matrix of four rows of four LEDs each. This matrix is controlled by a shift register, which is the 74HC595D model. At the same time, the shift register is controlled by a SPI bus, in particular the SPI1 in the BeagleBone.

Cape accelerometer

A 3-axis accelerometer is also installed in the cape. It can be used to detect orientation, shake, fall, tilt, motion, shock, vibration in the three space dimensions. The model is a Freescale MMA8453QR1, and these are some of its features:

- 1.95V to 3.6V supply voltage

- ±2g/±4g/±8g dynamically selectable full-scale
- 10 bit and 8 bit digital output resolution
- Controlled by I2C digital output interface

Cape NFC

Near Field Communication (NFC) is an open short-range radio technology that enables communication between devices at short distance. The cape is provided with a NFC tag implemented with the ST Microelectronics memory M24LR04E-RMN6T, which is an EEPROM with a NFC antenna control.

The memory in the NFC device can be acceded using an I2C bus interface.

Cape CAN bus

The Texas Instruments ISO1050DUB Isolated CAN transceiver bus is also installed in the cape. The following are some features of this transceiver:

- ISO11898-2:2003 compliant. High-speed (up to 1 Mbit/s)
- 2500 Vrms isolation

Cape GPMC interface

The General Purpose Memory Controller (GPMC) is a memory controller aimed to interface external memory devices:

- Asynchronous SRAM-like memories and ASIC devices
- Asynchronous, synchronous, and page mode burst NOR flash devices
- NAND Flash
- Pseudo-SRAM devices

GPMC is connected to the expansion headers CN1 and CN2 to allow communication with external devices.

5 Methodologies and development

5.1 Brief

The methodologies and developments of this project are the focus of the current chapter. It includes all software developments and methodologies to reach the objectives exposed in the introductory chapter. In particular, the following topics will be explained:

- How to set a proper environment to work and develop with the BeagleBone
- How to get to get a proper OS in the BeagleBone with the restrictions stated before
- How to get an asynchronous, synchronous, and synchronous burst GPMC driver for the BeagleBone
- How to adapt the legacy system cape overlay and cape applications in the new system
- How to save time developing for the BeagleBone using Python

The laboratory work, tests and tests results are explained in the next chapter. Now, let's start with the configuration of the development environment.

5.2 Setting up the development environment

The following section is going to describe how the development environment is set. It is focused on the system which handles the connection to the BeagleBone and the tools within. In order set up the BeagleBone properly and work pleasantly with it, two main scenarios have been prepared:

- Connected scenario. The BeagleBone board is physically connected to a router and its IP address belongs to its network. The board can be handled with a PC within the same network, and can also access to internet favoring the installation and upgrade of parts of its system.
- Connectionless scenario. The BeagleBone is physically connected through Ethernet to the PC that is used to handle this board. There is no internet connection available for the BeagleBone.

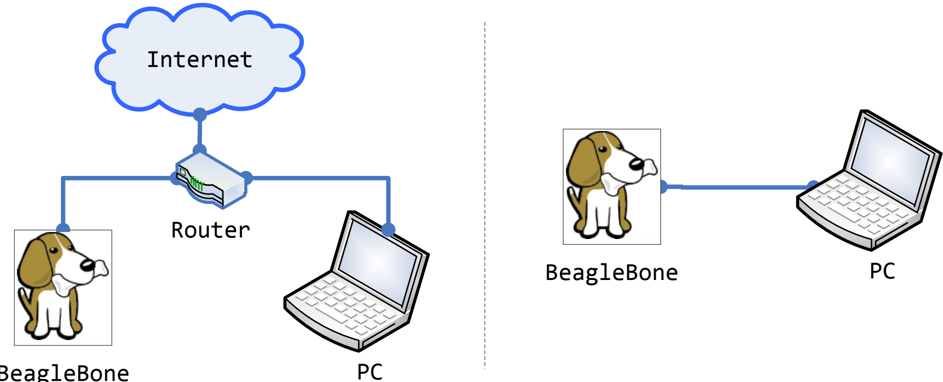


Fig 5.1: Left: connected scenario. Right: connectionless scenario

5.2.1 Virtual machines

In computing, a virtual machine can be defined as an emulation of a particular computer guest system into a real computer host system. Virtualization allows operations in the guest computer architecture using specific software or hardware. Virtualizing a system allows having a complete set of tools for a particular architecture and operating system without using a new hardware.

Part of the work in this project has been done using a virtualized system. This part is the development environment, which was used to develop and build the system and applications in the BeagleBone. In order to do so, two popular virtualization frameworks can be used: Virtual Box and WMWare.

- ***Virtual Box*** is an easy to use free virtualization application which can be installed in Windows, Linux, Mac and Solaris environments. It provides an extensive set of options to fully customize the features and performance of the system to virtualize. Several machines can be virtualized by using this framework.
- ***VMWare*** is also a popular virtualization environment. Although this application is not free, and its use, options and interface are definitely very similar to the previous one, its performance is better enough to decide to use it.

For this work, the virtualized guest system needs to accomplish a set of requirements in order to develop the BeagleBone system in a proper way. Basically, the system has to be a recent Linux distribution which supports a wide set of common tools and some particular ones. Particular tools can be understood as those required to cross compile or build the applications in the BeagleBone, as well as those to create the full system image and building its kernel. Different classical and popular systems are proper to this purpose. Amongst them there are Ubuntu, Debian and SuSE:

- ***Ubuntu*** is the most popular Linux distribution providing a beautiful and friendly user interface based in the Unity desktop. The system is reliable because it has received support of a vast user community along the years.
- ***Debian*** does not have such a beautiful interface as Ubuntu, bus has gained an important prestige among the community of developers. System architecture and file system, as well as the package system and other components are very likely in both distributions. Gnome is its default desktop in Debian, which is much lighter than Unity.
- ***SuSE*** is also a very friendly distribution, constituting a solid alternative to Ubuntu and Debian.

5.2.1.1 Choosing a proper virtual system

In this case, the choice of a virtual machine depends on the performance of the guest system. Both virtual machine platforms are similar in their basic usage. Nevertheless, the difference is clearly appreciated in the behavior of its guest system. For this reason, **VMWare** was finally selected to be the virtualization platform.

The choice of the guest system was made in accordance to internet documentation. Several candidates are feasible: Ubuntu, Debian or SuSE and others. The Texas Instruments web page, which produces Sitara

development environment for ARM processors, has the tendency to orientate the user to install their SDK in Ubuntu systems [5.1]. So, an Ubuntu guest system sounds as an ideal candidate.

In order to improve the performance in the tuple VMWare-Ubuntu, a specific flavor of Ubuntu is chosen. Particularly, Ubuntu Gnome 14.04 is selected. Unity becomes the default desktop of Ubuntu since 2010 [5.2], but it requires higher graphical performance compared to Gnome. Canonical (the community behind Ubuntu) also provides the option to get Ubuntu with Gnome 3.0 desktop as default, which is thinner than Unity.

5.2.1.3 Installing Ubuntu 14.10 Gnome on the VMWare

The flavor of Ubuntu which provides Gnome as the default desktop is called Ubuntu Gnome. Although it is also an official version of Ubuntu, it is not directly provided at the official Ubuntu/Canonical web page.

Installing Ubuntu Gnome on a VMWare virtual machine is much easier and quicker than in a Virtual Box machine. Some testing was made comparing both installations, and the combination of VMWare with Ubuntu Gnome was clearly victorious due to the following:

- The Unity desktop in VirtualBox is absolutely unusable. In VMware, the Gnome desktop works as if a native system were installed in a fluent machine.
- The connection to internet is clearly better in WMWare. For instance, cloning the Linux Kernel from a Git repository in VMWare is between one and two magnitude orders faster than in VirtualBox.

Connecting the virtual machine to the physical network interfaces is done by bridging them to the virtual ones. The Virtual machine can define a set of virtual interfaces. In this case, there are two Ethernet virtual interfaces: the first is bridged to the WLAN physical interface, and the second is bridged to the Ethernet physical interface. These interfaces are locally set by the router using DHCP.

In order to work with this virtual machine and prepare the environment for some operations that will be needed to work with the BeagleBone, some actualizations are done:

- Download and install *git* in order to get a tool to clone and handle source repositories.
- Download and install *ncurses*, which provides a kind of graphical environment to compile a kernel
- Download and install *lzop*, which is a kernel compressing tool
- Download and install *cpp-arm-linux-gnueabihf*, which is the cross-compilation tool for the ARM processor architecture in the BeagleBone and the same tool that is included in the Texas Instruments Sitara SDK. Sitara is a development environment that will be commented later.

In addition, some other components needed to be installed and configured. Installing Ubuntu in VMWare is evidently slighter than in VirtualBox, taking some configuration parameters by default such as the language or keyboard. Extra components that have been installed are:

- A secure shell server SSH. It is necessary to get connected remotely. Getting this service can be done by downloading and installing the *ssh* package

- A network file system service NFS. It is necessary to mount a file system in a remote machine (in this case, this is the BeagleBone). Getting this service can be done by downloading and installing the *nfs-kernel-service* package.

In addition, the network-manager service, which automatically manages the network interfaces and connections, has been disabled in order to ease manually configuring the network infrastructure through the configuration files in /etc/network/interfaces (configure network interfaces) and /etc/resolv.conf (determine the DNS servers).

5.2.1.2 The host PC

The host PC is a common high performance personal computer. It is used as development environment as well as an interface to the BeagleBone. It runs a Windows 7 OS and virtualizes an Ubuntu 14.04 environment, which is used to work and develop the software and system of the board.

The following are the main features of the host PC:

- Intel Core i7-2630QM 2GHz (year 2011)
- 6GB RAM memory
- 512 GB SSD (Solid State Disk)
- Virtual machine VMWare. Virtual guest system Ubuntu Gnome 14.04

The selected virtual machine and the guest system are a good combination to work with. Performance results are better compared to other combinations which were finally discarded.

After installing an Ubuntu 10.04 system in both virtual machine platforms, it is easily noticeable that the behavior of the distribution desktop is absolutely different. In VMWare Workstation, the fluency of the environment is such that one could think that he/she is using a native OS. In the other hand, the difficulties shown in the use of the desktop interface in Virtual Box are strong enough to discard this option.

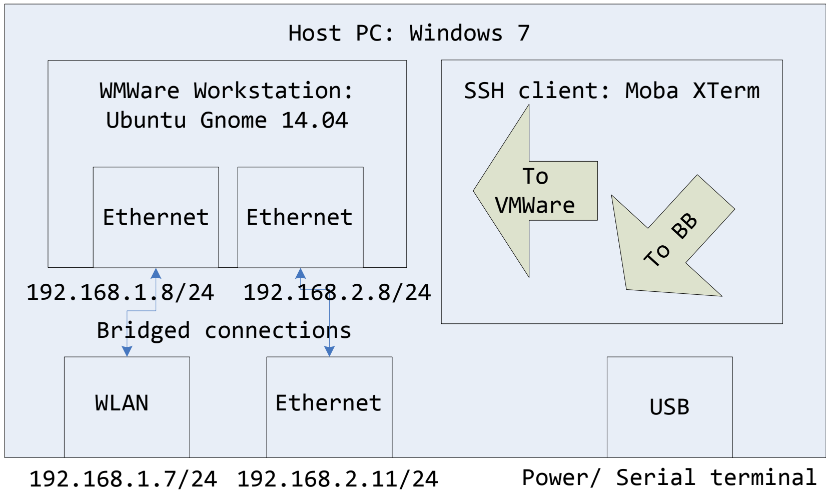


Fig 5.2: Host PC inner system relationships

The sketch in figure above depicts how the host PC system is handled. The operation is basically the following: the virtual machine is started with the Ubuntu guest system, and is configured to have two virtual network interfaces. These interfaces are bridged to the physical ones of the host. In that way the guest has the same connectivity as the host which now can be connected to the BeagleBone board.

In general, the virtual machine can be minimized and handled through a SSH client. Using a SSH client for Windows can ease interfacing both the guest machine and the BeagleBone at the same time. For this purpose, the application Moba Xterm has been used.

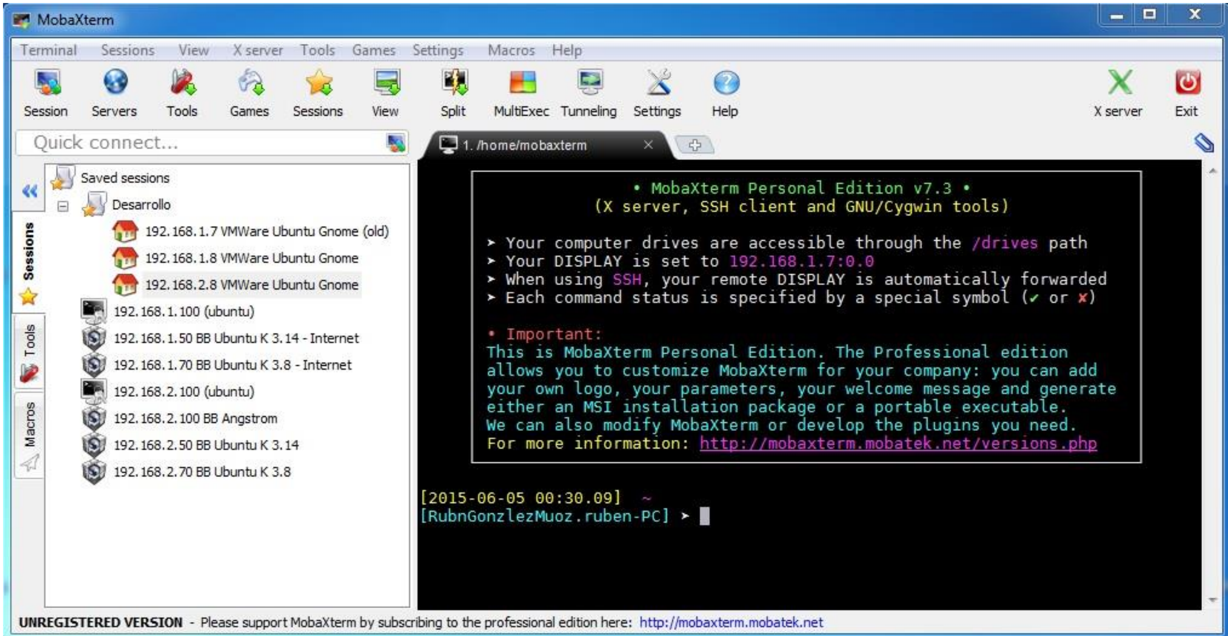


Fig 5.3: Moba Xterm interface

Moba Xterm is composed by a simple and intuitive interface which provides network connection clients such as SSH, FTP, SFTP, VNC or XDMCP, among others. A set of session configurations can be stored and retrieved each time a connection needs to be made, allowing ease of use.

Connecting the BeagleBone to internet

Like in any other Ubuntu, the process of connecting the BeagleBone to internet is identical. The particular connected scenario is provided by a fixed IP address defining its static primary interface parameters in /etc/network/interfaces file:

```
# interfaces(5) file used by ifup(8) and ifdown(8)

# loopback network interface
auto lo
iface lo inet loopback

# primary network interface
auto eth0
iface eth0 inet static
address 192.168.1.100
netmask 255.255.255.0
network 192.168.1.0
broadcast 192.168.1.255
```

```
gateway 192.168.1.1
```

Moreover, the DNS needs to be defined through the file /etc/resolv.conf. In order to ease the development Google DNSs are set in the BeagleBone:

```
# Dynamic resolv.conf(5) file for glibc resolver(3) generated by resolvconf(8)
nameserver 8.8.8.8
nameserver 8.8.4.4
```

5.2.2 Cross compiling

Some particular tools are required in order to generate executable software for computers such as the BeagleBone. In most cases, these tools come integrated into Software Development Kits (SDK). This is the case of the *Sitara SDK* from Texas Instruments (TI). The software is free and can be easily downloaded from the web of TI:

Part Number	License	Status	Version	Version date	OS	Kernel	Core
LINUXEZSDK-BONE: **Linux EZSDK for BeagleBone, Beaglebone Black**	Free	ACTIVE	v01.00.00	10-Apr-2015	Linux	v3.14.26	ARM Cortex-A8

Table 5.1: Current version of TI Sitara SDK for BeagleBone and BeagleBone Black

This SDK has a size of nearly 2GB because it is composed of a variety of tools in addition to the cross compilation software. For instance, this SDK provides the Code Composer Studio, which is an Integrated Development Environment (IDE) to program the microcontroller focused on devices of TI.

The installation of this SDK is done through an interactive script and not difficult to follow in a Debian or Ubuntu distribution. Nevertheless, installing the full SDK is not required in order to build BeagleBone applications and system: just the cross compiler toolchain is enough in order to do so.

5.2.2.1 GCC toolchain

The GCC compiler toolchain used for this work is provided by Linaro. This is a non-for-profit engineering organization which works with open source and free software focusing mainly in the Linux kernel and the GNU Compiler Collection, among others. The Linaro GCC compiler is also included in the Sitara SDK [5.3].

Linaro provides a set of pre-built hard float compilers (hf) which come into the same package. Moreover, the toolchain is completed with other linked tools composed basically by binary utilities (binutils), the GLIBC library and a debugger.

The compiler of GCC runs on a host system of a specific architecture, typically x86, but produces executables to run on a different architecture such as ARM. This is called cross compilation and is the typical way of building embedded software. This is the case of the BeagleBone in the current scenario.

Linaro has released optimized toolchains for current ARM processors such as Cortex A8 and Cortex A9. Cross toolchains are available to Ubuntu users through special packages that can be obtained from the Linaro FTP servers.

It is important to highlight that in order to set a proper development environment just the Linaro GCC toolchain needs to be installed. Installing the Sitara SDK is too expensive in terms of time and memory resources, and the functionalities that it provides are not required for the current work.

5.2.2.2 Kernel resources

In order to get a proper kernel, first of all it is needed to get a proper repository. Recently the BeagleBone kernel mainline Git repository has moved from one location to another, so the first one has become deprecated. Nevertheless, it has stable versions which work in the system with no problems.

- git://github.com/beagleboard/kernel.git.

This deprecated repository has branches of kernel versions 3.2, 3.8 to 3.14, and including other branches which are exclusive for the BeagleBoard (not BeagleBone). The latest commit in this repository was done on August 2014.

The location of the new kernel BeagleBone kernel repository is the following:

- git://github.com/beagleboard/linux.git

This repo also provides retro compatibility with the older one, allowing access to older releases of the different kernel branches. At present, the repository also includes access to the kernel branch 4.1, which provides some important changes respect the 3.14 branch that are related to capes. The responsible of this repository is Robert C. Nelson, which is also the main contributor in the BeagleBone kernel development.

But this is not the only place where to find kernel resources. The same contributor has also its own BeagleBone repository, which is the one that will be finally used for the current work:

- git://github.com/RobertCNelson/bb-kernel

The reason of choosing this repository is due to the instructions the same Robert C. Nelson provides for developing a complete Ubuntu BeagleBone micro SD card image. There, he uses this repository which is also supposed to be optimal for these purposes. This repository provides kernel branches from 3.1 to 3.19, 4.0, 4.1 and 4.2 for ARM am335x processors.

One advantage selecting this repository can be seen when building the kernel once cloned. This process is almost automatically carried by executing the *build_kernel.sh* script, also included.

Two kernel branches have been considered for the present work: branches 3.8 and 3.14. The developments started with 3.14, which was the latest branch under development by these days. Nevertheless, problems with the cape that were expounded before lead the development to downgrade to 3.8. By the time of the current work it was recommended to use kernel 3.8 if the purpose was to use capes. This branch is fully extended and supported by the BeagleBone community of developers. Because it is a rather new concept, kernels from versions 3.13 and on try to change or evolve the device tree philosophy suppressing the cape manager. Current kernel branches from 4.0 and on retrieve the cape manager concept.

5.3 Ubuntu embedded

5.3.1 From Angstrom to Ubuntu

Before starting to search solutions to the stated problems in this work, some research on the current system running into the BeagleBone has been done. The objective is to find relevant information in the Angstrom system that could be useful or to serve as a basis for the rest of the project. This process is aimed to gather relevant files and file systems within the micro SD with Angstrom.

5.3.1.1 Files for the EEPROM in the cape

In the file system, there is a script called **eeprom.js** which parses and generates the EEPROM data file that configures the EEPROM cape. The specification of the content is done in JSON format.

In the Angstrom file system, this script is located at:

- /var/lib/cloud9/eeprom/bonescript-master/node_modules/bonescript/eeprom.js

In addition, the file which describes the contents in the EEPROM needs to be found. Its name is **cape-bone-TT01v1.json**, and was designed and implemented in a previous project. It has been found in the following location:

- /var/lib/cloud9/eeprom/bonescript-master/node_modules/bonescript/cape-bone-TT01v1.json

The whole file system under **/var/lib/cloud9/** is retrieved and copied to the development environment in order to do the required modifications, if any.

The content file that will be dumped to the EEPROM can be generated locally in the BeagleBone. This can be achieved by changing to the depicted directory and executing the following:

```
# cd ./var/lib/cloud9/eeprom/bonescript-master/node_modules/bonescript/
# node ./eeprom.js -w cape-bone-TT01v1.json
```

This command generates a binary file named **cape-bone-TT01v1** which can be loaded into the BeagleBone cape by short circuiting the pins 4 and 7 of the EEPROM chip (they are actually short circuited) and executing the following at the same time:

```
# cat cape-bone-TT01v1.eeprom.data > /sys/bus/i2c/devices/1-0057/eeprom
```

5.3.1.2 Generating a dtb test file for Angstrom

In order to build a testing device tree overlay, the example source file **cape-bone-iio-00A0.dts** is taken as reference. This is an official file that can be found under the kernel file system of the BeagleBone. In the micro SD file system, it is located in the following folder:

- /lib/firmware/cape-bone-iio-00A0.dts

In order to generate the device tree overlay, the device tree source file must be compiled executing the following instruction:

```
# dtc -O dtb -o cape-bone-iio-00A0.dtbo -b 0 -@ cape-bone-iio-00A0.dts
```

The resulting overlay is named **cape-bone-iio-00A0.dtbo**. Its purpose is enabling the management of the analog interfaces or pins in the processor, which are called from AIN0 to AIN7.

Now, the *dtbo file can be copied to /lib/firmware. That folder contains all the system device tree overlays. In order to load the functionality, the identifier of this overlay needs to be copied to the slot system in the cape manager:

```
# echo cape-bone-iio.dtbo >  /sys/devices/bone_capemgr.7/slots
```

The operating system is continuously monitoring this type of device files. After executing the instruction before, there should not be any message thrown by the console if everything has gone alright. Otherwise, the following message will be printed:

```
-sh: echo: write error: File exists
```

In this case, the system rejects writing this file because this overlay has been loaded previously.

5.3.1.3 Device tree cape-bone-TT01v1-00A0

The device tree source file cape-bone-TT01v1-00A0.dts can be found in the following location in the file system:

- /lib/firmware/cape-bone-TT01v1-00A0.dts

Its corresponding binary file, the device tree overlay cape-bone-TT01v1-00A0.dtbo, can also be found at the same location:

- /lib/firmware/cape-bone-TT01v1-00A0.dtbo

Now, it can be checked that the binary *.dtbo file can be generated as explained before:

```
# cd home/root/boneDeviceTree/TT01v1/firmware/
# cp cape-bone-TT01v1-00A0.dtbo cape-bone-TT01v1-00A0.dtbo.bak
# dtc -O dtb -o cape-bone-TT01v1-00A0.dtbo -b 0 -@ cape-bone-TT01v1-00A0.dts
```

Note that the original *.dtbo file has been saved before compiling the device tree source file. Once done, the compilation generates the overlay **cape-bone-TT01v1-00A0.dtbo**. Now a checksum can be done in order to see if the new overlay is different from the version in Angstrom:

```
# md5sum cape-bone-TT01v1-00A0.dtbo
f51b15fb4fb24e7d88938ae0a5ba24eb  cape-bone-TT01v1-00A0.dtbo
# md5sum cape-bone-TT01v1-00A0.dtbo
f51b15fb4fb24e7d88938ae0a5ba24eb  cape-bone-TT01v1-00A0.dtbo
```

The md5 checksum shows that the recently generated file is exactly the same as the one existing in the Angstrom system. If no modifications were needed to be done on the sources of this file, it could be copied to /lib/firmware in the micro SD card and loaded in the system through the cape manager:

```
# echo cape-bone-TT01v1-00A0 > /sys/devices/bone_capemgr.7/slots
```

Later on this document it will be shown that some modifications are needed to have all the cape functionalities running on the new Ubuntu system.

Note that in the Angstrom system the device tree overlays and the system device tree files are located in different folders. The earlier come into /lib/firmware whereas the later come into /boot, which is a boot partition:

```
# ls /boot/
am335x-bone.dtb
[…]
```

The BeagleBone loads its *.dtb file which is also present in /boot partition. In general, this file is called **am335x-bone.dtb** in the BeagleBone. The folder /lib/firmware contains the *.dtbo files which are the overlays redefining and enhancing functionalities described in the *.dtb file loaded in the /boot partition.

5.3.1.4 Programs in Angstrom using the cape

The binaries of the programs in the file system that use the cape are located in /home/root. In order to use them in another systems such as Ubuntu (which is the target system of this project), the source code needs to be retrieved and compiled in a development environment.

```
# ls /home/root
ACCELEROMETER  Desktop           ENCODER  GPIO2             LEDMATRIX      boneDeviceTree
pruebaseclipse
ANALOG         DeviceTreeAnalog  GPIO1    HelloWorldRemote  SPI1devicetree  npm-debug.log
```

The executable applications are those described in capital letters. After executing each of them as root, it is checked that their behavior is the expected as described in a previous project.

5.3.2 Build for Ubuntu 3.8 RCN Kernel

The current section describes how to build the whole system for a BeagleBone into a micro SD card using a set of customized options such as choosing the Linux embedded distribution (Ubuntu, Debian) or the kernel version (3.8 or other) for the system. The steps in this section are taken from the Linux on ARM web page, BeagleBone section, which is frequently updated by the main maintainer of the BeagleBone and BeagleBone Black systems Robert C. Nelson [5.4].

- Main page: https://eewiki.net/display/linuxonarm/Home
- BeagleBone: https://eewiki.net/display/linuxonarm/BeagleBone

This site describes how to create a full embedded Linux system for different ARM processor based boards. Among them, there are the different development boards from Texas Instruments:

"This is the home of the Linux on ARM space. From these links you will find quick instructions for getting Linux running on these development boards."

[...]

- Texas Instruments:
- BeagleBoard
- BeagleBone
- BeagleBone Black
- PandaBoard
- OMAP5432 uEVM

[...]

The process of creating a basic system using embedded Ubuntu 14.04 and kernel 3.8 is going to be detailed next. The system is deployed on a 4GB size micro SD card. The whole process is done on the development environment, so the intervention of the BeagleBone board is not required at any time until its own the system startup.

5.3.2.1 Creating the new system

In order to start with this process, it is considered that the development environment does not have any development tool yet. As explained before, the development environment is a virtualized Ubuntu Gnome 14.10 running over a VMWare machine. Only the required tools will be installed.

In the development environment a new file folder is created, namely Workspace. This workspace has the objective of gathering all the tasks in a unique location. Root privileges are assumed during the execution of all the tasks.

- Starting location in the file structure of the development environment: ~/

```
# mkdir Workspace
# cd Workspace
```

This will become the current working location: ~/Workspace

The cross compiler

The ARM cross compiler is downloaded and installed:

```
# wget -c https://releases.linaro.org/14.09/components/toolchain/binaries/gcc-linaro-arm-linux-gnueabihf-4.9-2014.09_linux.tar.xz
# tar xf gcc-linaro-arm-linux-gnueabihf-4.9-2014.09_linux.tar.xz
```

A cross compiler path environment variable is exported in order to ease the processes:

```
# export
CC=`pwd`/gcc-linaro-arm-linux-gnueabihf-4.9-2014.09_linux/bin/arm-linux-gnueabihf-
```

U-boot

The u-boot loader source code is cloned from a Git repository, and a checkout to the proper branch is done. It is a loader for the BeagleBone.

```
# git clone git://git.denx.de/u-boot.git
# cd u-boot/
# git checkout v2015.01 -b tmp
```

The proper patch for this loader and type of ARM processor is downloaded and applied:

```
# wget -c https://raw.githubusercontent.com/eewiki/u-boot-patches/master/v2015.01/0001-am335x_evm-uEnv.txt-bootz-n-fixes.patch
# patch -p1 < 0001-am335x_evm-uEnv.txt-bootz-n-fixes.patch
```

After that, the binary loader is produced by building the project using its own Makefile:

```
# make ARCH=arm CROSS_COMPILE=${CC} distclean
# make ARCH=arm CROSS_COMPILE=${CC} am335x_evm_defconfig
# make ARCH=arm CROSS_COMPILE=${CC}
```

Device Tree Compiler

Current location: ~/Workspace/

The device tree compiler for ARM devices is downloaded and updated. This compiler produces the system and custom overlays for the capes used in the BeagleBone.

```
# wget -c https://raw.github.com/RobertCNelson/tools/master/pkgs/dtc.sh
# chmod +x dtc.sh
```

The script *dtc.sh* automatically gets and updates from Internet the source of the DTC and builds this executable

```
# ./dtc.sh
```

Kernel

The source code of the kernel is cloned from the Robert C. Nelson Git repository. A checkout to the proper branch of the kernel is done. In this case and in order to work properly with the cape, the kernel branch needs to be the 3.8 version.

```
# git clone https://github.com/RobertCNelson/bb-kernel.git
# cd bb-kernel/
# git checkout origin/am33x-v3.8 -b tmp
```

The kernel now can be built using the script *build_kernel.sh*, which has been downloaded once done the steps above. The process of building the kernel is almost automatic but rather long and complex. The Linux kernel from Linus Torvalds is also cloned and built within this process. Some aspects to bear in mind before executing this script and building the kernel are the following:

- The username and email account of Git needs to be configured before compiling the kernel. Otherwise, the process is aborted.

- Some basic complementary tools needed by this script need to be installed. These tools are: *bc, build-essential, device-tree-compiler, fakeroot, lsb-release, lzma, lzop, man-db*.
- A stable branch of the kernel will be built using make menuconfig. At this point, an editor based in the *ncurses* library will be executed allowing the customization of the kernel. The default configuration (without enabling or disabling additional options) is valid.

```
# ./build_kernel.sh
```

Ubuntu File System

The Ubuntu 14.04.1 minimal based file system for ARM devices is downloaded in the development environment. It has the following pre-installed user:

- Username: ubuntu
- Password: temppwd

```
# wget -c https://rcn-ee.net/rootfs/eewiki/minfs/ubuntu-14.04.1-minimal-armhf-2015-01-
20.tar.xz
```

Its privileges can be arisen modifying the file /etc/sudoers

The *tarball* file is uncompressed releasing the file and folder system structure.

```
# tar xf ubuntu-14.04.1-minimal-armhf-2015-01-20.tar.xz
```

Preparing the micro SD card

In order to ease the installation, the unit name using the micro SD card in the development environment needs to be determined. A variable with its whole path is exported in the current shell environment. In operating systems such as Ubuntu the *lsblk* tool is useful to get this unit name. This path uses to be /deb/sdb in systems such as Ubuntu.

```
# export DISK=/dev/sdb
```

Before the installation, the contests in the micro SD card are erased.

```
# dd if=/dev/zero of=${DISK} bs=1M count=10
```

The next step is installing the bootloader files MLO and u-boot.img produced at the early stages of this reference.

```
# dd if=./u-boot/MLO of=${DISK} count=1 seek=1 conv=notrunc bs=128k
# dd if=./u-boot/u-boot.img of=${DISK} count=2 seek=1 conv=notrunc bs=384k
```

The micro SD card partition scheme is created at this point.

```
# sudo sfdisk --in-order --Linux --unit M ${DISK} <<-__EOF__
1,,0x83,*
__EOF__
```

The next step is formatting this partition properly:

```
# sudo mkfs.ext4 ${DISK}1 -L rootfs
```

Afterwards, it is mounted in /medis/rootfs/

```
# mount ${DISK}1 /media/rootfs/
```

Installation into the micro SD card

The kernel version number is retrieved from the kernel compilation process which was started with the execution of the *build_kernel.sh* script.

```
-----------------------------
Script Complete
eewiki.net: [user@localhost:~$ export kernel_version=3.X.Y-Z]
-----------------------------
```

In this case, the version is **3.8.13-bone70**. As indicated in the end of the execution of this script, an environment variable for the kernel version is exported again:

```
# export kernel_version=3.8.13-bone70
```

The next step is uncompressing the minimal Ubuntu file system into the micro SD card.

```
# tar xfvp ./*-*-*-armhf-*/armhf-rootfs-*.tar -C /media/rootfs/
```

A new file called *uEnv.txt* is created. This file contains the information of the kernel version.

```
# sh -c "echo 'uname_r=${kernel_version}' > /media/rootfs/boot/uEnv.txt"
```

The next step is to copy the compressed image of the kernel into the partition of the micro SD card using the proper name:

```
# cp -v ./bb-kernel/deploy/${kernel_version}.zImage /media/rootfs/boot/vmlinuz-
${kernel_version}
```

The directory containing the device tree binary files (among them, am335x-bone.dtb) is created and the *tarball* which contains them is uncompressed in it. This *tarball* was also generated during the execution of the script *build_kernel.sh*.

```
# mkdir -p /media/rootfs/boot/dtbs/${kernel_version}/
# tar xfv ./bb-kernel/deploy/${kernel_version}-dtbs.tar.gz -C
/media/rootfs/boot/dtbs/${kernel_version}/
```

The module drivers are installed as well:

```
# tar xfv ./bb-kernel/deploy/${kernel_version}-modules.tar.gz -C /media/rootfs/
```

The file containing the partition table is created next:

```
# sh -c "echo '/dev/mmcblk0p1  /  auto  errors=remount-ro  0  1' >> /media/rootfs/etc/fstab"
```

In addition, a configuration for the network interface is also defined using the file /etc/network/interfaces. A static address is provided to the eth0 interface in order to access the BeagleBone board using the Ethernet connection directly using a PC and without using an access point or router. In case that software download or update needs to be done, the network topology and file contents would be modified.

```
# vi /etc/network/interfaces
```

Contents of the /etc/network/interfaces file:

```
# interfaces(5) file used by ifup(8) and ifdown(8)
# Include files from /etc/network/interfaces.d:
source-directory /etc/network/interfaces.d

auto lo
iface lo inet loopback

# primary network interface
auto eth0
iface eth0 inet static
address 192.168.2.70
netmask 255.255.255.0
network 192.168.2.0
gateway 192.168.2.1
```

If the micro SD card is going to be used in different BeagleBone boards, the next information needs to be stored in the/media/rootfs/etc/udev/rules.d/70-persistent-net.rules file:

```
# vi /media/rootfs/etc/udev/rules.d/70-persistent-net.rules
```

Contents:

```
# BeagleBone: net device ()
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*", ATTR{dev_id}=="0x0", ATTR{type}=="1",
KERNEL=="eth*", NAME="eth0"
```

In order to enable the serial port login, the following file needs to be modified:

```
# vi /media/rootfs/etc/init/serial.conf
```

Contents:

```
start on stopped rc RUNLEVEL=[2345]
stop on runlevel [!2345]
respawn
exec /sbin/getty 115200 ttyO0
```

At this point, the micro SD card is ready to get plugged to the BeagleBone slot. The micro SD card is unmounted and unplugged from the host PC:

```
# sync
# umount /media/rootfs
```

5.3.3 Alternative builds of Ubuntu. Kernel 3.14 in ARMhf

The instructions in the site *"Linux for ARMhf devices"* are followed in order to make an alternative build and deployment of an Ubuntu system over embedded ARM systems such as the BeagleBone [5.5]. This site provides a complete description of the process of installing prepackaged ARM Linux images for BeagleBone and other boards. Note that **this process has not been used to build the final image of the system**. For this purpose, the process described in the previous section has been used instead.

The steps to deploy a prepackaged image of the Ubuntu system from those available in this site can be summarized in the steps exposed along the following lines.

First of all, it is necessary to determine which will be the device file of the micro SD card that will be used to deploy the image. Often, this device file uses to be /dev/sdb. This will be taken as example for the current explanation.

Next, the micro SD card partitioning process needs to be started:

```
# fdisk /dev/sdb
```

Now, an interactive shell application is started. A new partition table needs to be initialized by selecting "o", and then verify the partition table is empty by selecting "p".

A boot partition needs to be created next by selecting "n". It is needed to indicate that it has to be a primary partition by selecting "p", and that is the first partition by selecting "1". The first default sector needs to be accepted by pressing "enter" and the last one has to be specified at "4095".

Now, the partition type needs to be set to FAT16 by selecting "t" for type and "e" for W95 FAT16 (LBA). This partition needs to be set as bootable by selecting "a" and then choosing "1" for the first partition.

The next step is to create a partition for the root file system by selecting "n" for new partition, "p" for primary partition, and "2" to indicate that it is the second partition. The default values of the first and last sectors can by selected by pressing enter twice.

Now that the partition table has been completed, it can be checked by pressing "p":

```
Disk /dev/sdb: 7948 MB, 7948206080 bytes
255 heads, 63 sectors/track, 966 cylinders, total 15523840 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0xafb3f87b

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1   *        2048        4095        1024    e  W95 FAT16 (LBA)
/dev/sdb2            4096    15523839     7759872   83  Linux
```

At this point, the partitions need to be committed by typing "w" for write and exiting the *fdisk* application.

The next step consists on formatting the partitions. Format partition 1 as FAT:

```
# mkfs.vfat /dev/sdb1
```

Then format the partition 2 as ext4:

```
# mkfs.ext4 /dev/sdb2
```

The next step is to download and install the u-boot loader into the first partition of the micro SD card:

```
# wget http://s3.armhf.com/dist/bone/bone-uboot.tar.xz
# mkdir boot
# mount /dev/sdb1 boot
# tar xJvf bone-uboot.tar.xz -C boot
# umount boot
```

Then, download and install the Ubuntu root file system into the second partition of the micro SD card:

```
# wget http://s3.armhf.com/dist/bone/ubuntu-trusty-14.04-rootfs-3.14.4.1-bone-
armhf.com.tar.xz
# mkdir rootfs
# mount /dev/sdb2 rootfs
# tar xJvf ubuntu-trusty-14.04-rootfs-3.14.4.1-bone-armhf.com.tar.xz -C rootfs
# umount rootfs
```

The micro SD card can be now removed from the host PC and inserted into the BeagleBone. The board is now ready to boot.

The first connection to the BeagleBone board needs to be done through the serial connection of the USB port. Programs such as screen *provide* a serial terminal application which can get connected easily form the host PC to the remote system using a serial connection:

```
# screen /dev/ttyUSB0 119200
```

The parameters that have been passed to the application have the following meaning:

- /dev/ttyUSB0 indicates the device file which handles the serial connection in a USB port
- 119200 indicates the connection bitrate of the serial port

More parameters can be configured, but in this case the default ones are taken.

Some seconds after executing this command, the prompt of the acceded machine appears. The user credentials are queried:

- Username: *ubuntu*
- Password: *ubuntu*

```
ubuntu@ubuntu-armhf:~$
```

Some basic operations can be performed in order to check the system. Navigating across the file system, checking its structure, connecting the board to internet and pinging remote servers are some examples. One can easily raise administrator privilege by executing:

```
# sudo -s
Intoduce the same password as before
root@ubuntu-armhf:~#
```

5.3.3.1 Building another kernel from scratch

A satisfactory build and deploy of the kernel for the BeagleBone can be done following the steps depicted in the blog "*Random Codes – Elementz Tech*" [5.6].

There are several kernel versions which can be generated using this procedure. By the time of tis report, the kernel version 3.14.1 was the last generated.

In order to get and compile the kernel it must be highlighted that the whole process could last between 2 and 3 hours depending on the hardware. Due to this, the system should be working with the highest performance, if possible. For instance:

- The virtual machine should have assigned most of the processor cores in the host

- If the host processor has some turbo or high performance mode, it should be activated. Processors such as Intel i7 can increase their performance in rates up to 40%.
- In addition, the compilation process can be optimized by allowing multiple execution threads using the parameter –jX, being X the number of threads.

Now that the conditions are optimal, the steps to follow are the following:

- Starting location in the file structure of the development environment: ~/

The repository with the kernel sources can be cloned from the official BeagleBone and BeagleBoard kernel repository:

```
# git clone git://github.com/beagleboard/linux.git
```

Now, a checkout to the desired development branch can be made. For instance, the kernel branch 3.14 can be chosen. This will be the base kernel version for the system.

```
# cd kernel
```

- Now the working directory becomes: ~/kernel/

```
# git branch -t 3.14 origin/3.14
# git checkout 3.14
```

The patch script *patch.sh* can be applied now:

```
# ./patch.sh
```

Then, kernel, dtb files and modules can be built:

```
# cd kernel
```

- This is the working directory now: ~/kernel/kernel/

```
# make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- beaglebone_defconfig
```

Now it is time to build the kernel compressed image and device tree binaries:

```
# make -j8 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- zImage dtbs LOADADDR=0x80008000
```

The modules need to be built and installed apart:

```
# make -j8 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- modules
# mkdir ~/rootfs
# make -j8 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- INSTALL_MOD_PATH=~/rootfs
modules_install
```

At this point, all the necessary files to deploy a new kernel have been generated: *zImage*, *am335m-bone.dtb* and the *modules* in the *rootfs* folder. In fact, if the kernel version is not very far away from the one that we want to replace, only the compressed image of the kernel should be enough.

The next step is to copy the new files to the BeagleBone. It can be done by copying them to the micro SD card or sending them to the connected BeagleBone through the network interface.

Sending *zImage*:

```
# scp ~/kernel/kernel/arch/arm/boot/zImage ubuntu@beaglebone:/boot/zImage
```

Sending *am335x-bone.dtb*:

```
# scp ~/kernel/kernel/arch/arm/boot/dts/am335x-bone.dtb ubuntu@beaglebone:/boot/dtbs/
```

Sending the modules, although it is not necessary for the BeagleBone to start and login:

```
# tar czvf modules.tgz ~/rootfs/*
# scp ~/modules.tgz ubuntu@beaglebone:/
```

Now, the next step is to login into the BeagleBone and uncompress the modules:

```
# tar xzvf modules.tgz
```

This installs the modules in the BeagleBone file system, in particular in /lib/modules/3.14.1+/

Now everything required for the new kernel is present in the micro SD card of the BeagleBone. The board can be rebooted and so the new kernel will be loaded. This easily can be checked:

```
# uname -a
Linux ubuntu-armhf 3.14.1+ #3 SMP Sun Dec 28 17:42:30 CET 2014 armv7l armv7l armv7l GNU/Linux
```

The kernel version is 3.14.1+, whereas the prebuilt kernel image in the Ubuntu system was 3.14.25+. The deployment of the new kernel has been completed successfully.

Moreover, it can be checked the correct loading of the kernel modules:

```
# lsmod
Module                  Size           Used by
nfsd                    243068  2
omap_aes                11820          0
omap_sham               16979          0
ti_am335x_adc           4712           0
kfifo_buf               2513           1 ti_am335x_adc
industrialio            46529          2 ti_am335x_adc,kfifo_buf
rtc_omap                5145           0
uio_pdrv_genirq         3215           0
uio                     8796           1 uio_pdrv_genirq
```

5.3.4 Some improvements

In order to enhance the usability of the BeagleBone, some improvements have been introduced in the system. These changes are not mandatory, but provide a better user experience. The most relevant ones are the following:

- Installing a Network File System NFS [5.7].
- Improving the profile of the shell.
- Installing some extra applications and tools.

The NFS server is installed in the host PC in the virtual machine. A NFS client is installed in the BeagleBone. Both of them are included in official Ubuntu packages called *nfs-kernel-server* and *nfs-common*, respectively. Sharing files and folders mounting a NFS system can speed up the some processes because the BeagleBone

could see some files as if they were of its own. An example can be sharing with the BeagleBone compiled binaries that have been built in the host PC.

Some additional changes have been done in the prompt and the shell. Options such as coloring the prompt and the file and folder types have been enabled. Aliases which ease the navigation along the file system have also been enabled.

Extra applications have been installed. The most relevant would be the editor *Vim*, which improves the usability regarding to its predecessor Vi. and the application *aptitude*, that has also been installed with the purpose to improve the *apt-get* tool or the apt-* toolset.

Finally, the Python framework as well as additional *Adafruit Python* modules has been installed in order to develop applications in this language. This will be explained in later sections in this document.

5.4 GPMC driver

Before going to the issues derived to the particular behavior of the GPMC driver itself, an introduction to the cross-compilation of programs and drivers for the BeagleBone will be given.

This section will provide an approximation to the compilation of basic programs and modules (drivers), and will serve as an introduction to compose more complex modules. In addition, a General Purpose Memory Controller (GPMC) will be implemented and probed providing a communication protocol between the processor and a memory in a FPGA. The next sections will deal with some of the solutions provided to the problematics derived.

5.4.1 First approach to the cross-compilation

Before trying to compile a complete driver for the new system, a first compilation approach needs to be done. The purpose is to get a simple application compiled, deployed into the BeagleBone and executed correctly. The main idea is getting a "Hello World" application running.

For this purpose, some Makefile-based program can be taken as example. The Makefile will serve as the recipe for building the application:

```
#-------------------------------------------------------------------------------
# Makefile code for a helloworld application

# General definitions

ARCH := arm
CROSS_COMPILE=arm-linux-gnueabihf-
CC=$(CROSS_COMPILE)gcc
CFLAGS=-c -Wall -O2 $(INCLUDE)
LDFLAGS=-lpthread

# Program specific definitions

SOURCES=src/helloworld.c
CHECK_INCLUDES=
EXECUTABLE=helloworld
OBJECTS=$(SOURCES:.c=.o)

# Make rules
```

```
all: $(SOURCES) $(EXECUTABLE)

clean:
	rm -rf $(PWD)/src/*.o $(PWD)/$(EXECUTABLE)
	touch $(PWD)/*
	touch $(PWD)/src/*

$(EXECUTABLE): $(OBJECTS)
	$(CC) $(LDFLAGS) $(OBJECTS) -o $@

%.o: %.c $(CHECK_INCLUDES)
	$(CC) $(CFLAGS) $< -o $@
#-------------------------------------------------------------------------------
```

It is important to indicate which compiler will be set in order to do the cross compilation. In this case, with a processor ARM Cortex in the BeagleBone, the proper compiler is *arm-linux-gnueabihf-gcc*. Note that the path to this compiler is included in the PATH environment variable.

The code for the “Hello World” application could be simply the following:

```
//-------------------------------------------------------------------------------
// Code of application helloworld.c

#include <stdio.h>

int main(void)
{
   printf("Hello World!!");
   return 0;
}
//-------------------------------------------------------------------------------
```

If properly set, compiling the application is as easy as executing the command *make* in the path of the Makefile and the source code. If everything has gone alright, no error message should be thrown by console after executing *make*.

Executing the resulting application in the host PC results in error indicating that the binary file could not be executed (as expected). To see if the application is executable, it must be deployed to the BeagleBone or executed after mounting the NFS system. Due to the fact that the BeagleBone has enough disk space, applications developed for this project will be deployed via secure copy (scp) to the micro SD card instead of mounting the NFS.

```
# ./helloworld
Hello World!!
```

The result above is the one expected. After building, deploying and executing this simple program the next step consists on doing something a bit more complex. This is compiling the GPMC asynchronous driver, which was built for a particular older version of the kernel and probed under the Angstrom OS. Now the new Linux kernel will be same as the one included in the image of the Ubuntu in the BeagleBone.

5.4.2 Compiling the simple module driver beginning

First of all, and before introducing the compilation of the module itself, a brief overview of the Makefile for a module or Linux driver will be given. Note that all the driver compilations under the following sections have

been made using the RCN Linux 3.8 Kernel source, which is exactly the same kernel that is built and used in the BeagleBone system.

5.4.2.1 Makefile module overview

To get introduced into the problematics of a compilation process of a driver, a simple laboratory module *driver beginning* is proposed: the module reads a particular register in the processor and retrieves its information. This is reached by implementing the corresponding read directive of the developed module. Note that in a Linux context, driver and module are synonyms.

An inspection of the original module Makefile gives the following information:

```
//-------------------------------------------------------------------------------
KSRC=../../../kernel-3.2.42-psp27/KERNEL

ARCH := arm
CROSS_COMPILE := arm-angstrom-linux-gnueabi-

MAKEARCH := $(MAKE) ARCH=$(ARCH) CROSS_COMPILE=$(CROSS_COMPILE)

EXTRA_CFLAGS := -I$(PWD)/src -I$(PWD)/include

obj-m += module.o
PWD := $(shell pwd)

all:
	$(MAKEARCH) -C $(KSRC) M=$(PWD) modules
clean:
	$(MAKEARCH) -C $(KSRC) M=$(PWD) clean
//-------------------------------------------------------------------------------
```

The rules provided by this Makefile (all and clean) have a similar format than those used to produce the Hello World application, and use the same variables:

- ARCH: provides the architecture of the cross-compilation destination system: arm
- CROSS_COMPLIE: provides the cross-compilation toolchain: arm-angstrom-linux-gnueabi
- MAKEARCH: provides the full command to cross compile with the given architecture
- KSRC: indicates the path to the Kernel source files
- PWD: indicates the current directory

Given that, it is easy to see that there are two variables whose values need to be modified: *KSRC* and *CROSS_COMPILE*. KSRC will indicate the current path with the target sources of the selected Kernel. In this case, the Kernel version used to compile the driver in Angstrom 3.2.42-psp27 is replaced for a current release of the 3.8 kernel. The CROSS_COMPILE variable will indicate the new toolchain *arm-linux-gnueabihf*, as depicted before.

```
//-------------------------------------------------------------------------------
KSRC=/home/ruben/gitlocal/linux

ARCH := arm
CROSS_COMPILE := arm-linux-gnueabihf-

MAKEARCH := $(MAKE) ARCH=$(ARCH) CROSS_COMPILE=$(CROSS_COMPILE)
```

```
EXTRA_CFLAGS := -I$(PWD)/src -I$(PWD)/include

obj-m += module.o
PWD := $(shell pwd)

all:
	$(MAKEARCH) -C $(KSRC) M=$(PWD) modules
clean:
	$(MAKEARCH) -C $(KSRC) M=$(PWD) clean
//---------------------------------------------------------------------------------
```

5.4.2.3 Getting to compile the driver beginning

The next step now is getting to compile the module that comes with the name of "driver beginning". There, the Makefile explained before is used proceeding with the compilation. Only one compilation error is shown when trying to compile the module without any modification. This error can be is easily corrected:

- One include path had to be included in the module.c source code:

```
#include <asm/io.h> // used for "ioremap" directive
```

Including this source file, the simple version of the driver can be compiled without errors. Now the driver can be sent to the BeagleBone, loaded, and tested:

```
[once copied module.ko to the Beaglebone]

# insmod module.ko
# lsmod
Module         Size       Used by
module         1979       0
[...]
```

A simple application is made with the purpose of testing the module. This application opens the file the module uses, reads its contents and prints them on the screen:

```
# ./app

SDBL: Microprocessor feature reference AM3359 = 020fd0383
```

Having tested a simple module, the following sections will describe the relevant features about building and probing more complex drivers. These modules implement the communication protocol between the processor and an extern FPGA memory through a GPMC.

5.4.3 General Purpose Memory Controller GPMC

As introduced before, the solution provided for the driver implements read and write operations in a GPMC bus. Part of the work done before considers an attached NOR memory within a FPGA which is accessed in an asynchronous way using an address and data multiplexed bus. The following schema provides a description of the connections between the GPMC interface and this type of memory device:

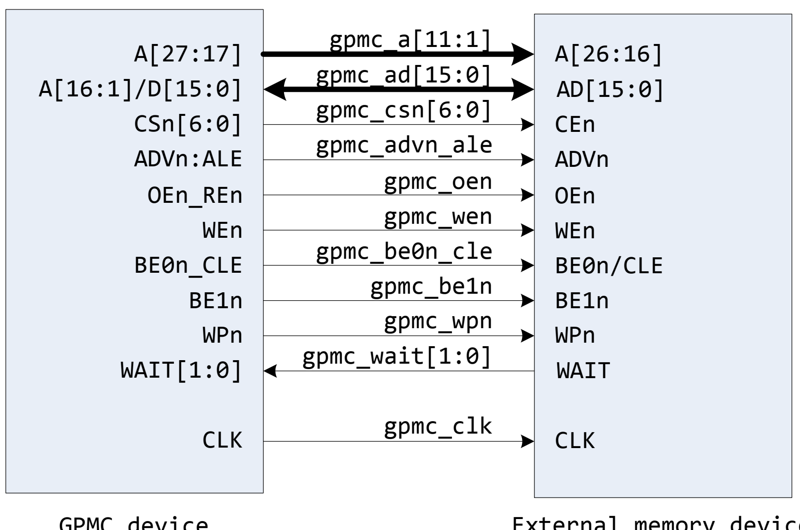


Fig 5.4: GPMC to 16 bit address and data multiplexed device

The complete set of signals in the GPMC is described in the following table:

Signal name	I/O	Description
GPMC_FCLK	Internal	Internal time reference
GPMC_CLK	O	External clock for synchronous modes
GPMC_A[27:17]	O	Address
GPMC_AD[15:0]	I/O	Data multiplexed with addresses
GPMC_CSXn	O	Chip select 0 and 1
GPMC_ADVn_ALE	O	Address valid enable
GPMC_OE_REn	O	Output enable (write access operation)
GPMC_WEn	O	Write enable (read access operation)
GPMC_WAIT[1:0]	I	Ready signal from memory device

Table 5.2: Description of the GPMC interface

More information about the GPMC and programming and configuration features of the BeagleBone ARM processor can be obtained in the *Technical Reference Manual of the AM335x ARM® Cortex™-A8 Microprocessor* [5.8].

Work done before included a FPGA memory side mounted on a DE2 board which is programmed to work as described previously (asynchronous) through the set of signals available. The figure bellow provides a brief sketch of its interface and internals:

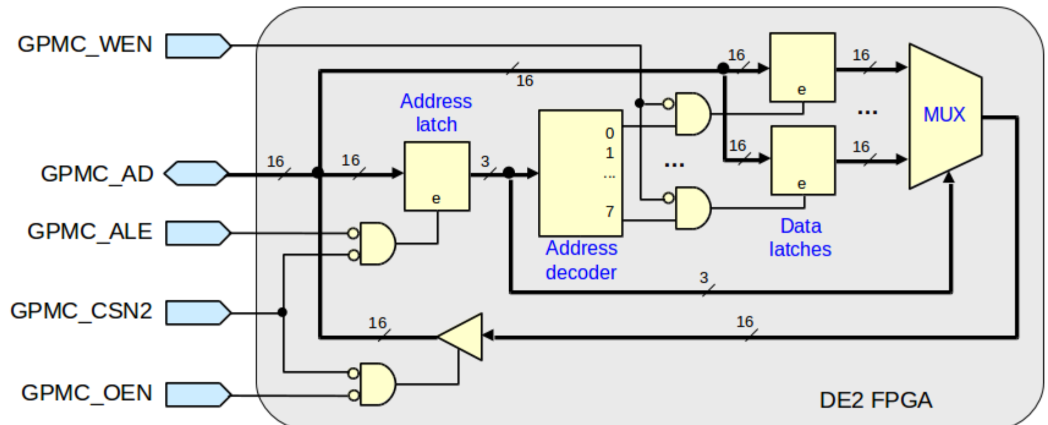


Fig 5.5: Interface and internals of the memory DE2 FPGA board

The system in the FPGA is composed of a set of 8 data latches. Setting up the signals in the GPMC properly, 16 bit words of data can be written in the memory coming from the gpmc_ad[15:0] bus. It is also possible to read 16 bit words of data from any of the latches to the gpmc_ad[15:0] bus asynchronously.

Given this brief introduction to the GPMC system, the following sections will introduce the particularities of different interface configuration types. In particular, three modes will be treated: asynchronous, synchronous, and burst. Implementations of their corresponding drivers will also be discussed.

5.4.4 Asynchronous GPMC

The current section is divided into three main parts. The first one introduces a seven step model to implement the configuration settings of the GPMC interface. The second one introduces the particular features and timings of the implemented asynchronous GPMC interface. Finally, the third one explains which are the necessary steps to build and probe the module within the current scenario.

5.4.4.1 Seven-step GPMC implementation model

This 7 step model is based on the guidelines that are provided in the technical reference of the processor. It can be seen as a recipe to program the GPMC interface. The following lines provide a qualitative sketch of the parameters to program in the GPMC interface module starting from a simple module code template.

- Step 0. Prepare the GPMC.
    - Enable the GPMC clock through the MODULEMODE field in the CM_PER_GPMC_CLKCTRL register
    - Reset the GPMC subsystem using the GPMC_SYSCONFIG_SOFTRESET field in the GPMC_SYSCONFIG register and wait until it is enabled monitoring the GPMC_SYSSTATUS register
    - Enable the “no idle mode” (an idle request is never acknowledged) using the GPMC_SYSCONFIG_IDLEMODE field in the GPMC_SYSCONFIG register
    - Mask all the interruptions setting the GPMC_IRQENABLE register to 0x00

- Disable the timeout control feature in the processor (It is fast enough) setting the GPMC_TIMEOUT_CONTROL register to 0x00

- Step 1. Set up GPMC_CONFIG_1 register: NOR memory attached and granularity x2
  - Set up the device size to 16 bits through the GPMC_CONFIG1_0_DEVICESIZE field
  - Set up the attached device page length to 4 words through the GPMC_CONFIG1_0_ATTACHEDDEVICEPAGELENGTH field
  - Indicate that data and addresses are multiplexed on the same bus
- Step 2. Set up GPMC_CONFIG_2 register: chip select assert and deassert times.
  - Set up the CS assert time using the GPMC_CONFIG2_0_CSDONTIME field
  - Set up the CS read deassert time using the GPMC_CONFIG2_0_CSRDOFFTIME
  - Set up the CS write deassert time using the GPMC_CONFIG2_0_CSWDOFFTIME
- Step 3. Set up GPMC_CONFIG_3 register: assert and deassert address enable
  - Set up the ADV assert time using the GPMC_CONFIG3_0_ADVONTIME field
  - Set up the ADV read deassert time using the GPMC_CONFIG3_0_ADVROFFTIME field
  - Set up the ADV write deassert time using the GPMC_CONFIG3_0_ADVROFFTIME field
- Step 4. Set up GPMC_CONFIG_4 register: assert and deassert output enable and write enable
  - Set up the OE assert time using the GPMC_CONFIG4_0_OEONTIME field
  - Set up the OE deassert time using the GPMC_CONFIG4_0_OEOFFTIME field
  - Set up the WE assert time using the GPMC_CONFIG4_0_WEONTIME field
  - Set up the WE deassert time using the GPMC_CONFIG4_0_WEOFFTIME field
- Step 5. Set up GPMC_CONFIG_5 register: read and write cycle times
  - Set up the read cycle time using the GPMC_CONFIG5_0_RDCYCLETIME field
  - Set up the write cycle time using the GPMC_CONFIG5_0_WRCYCLETIME field
  - Set up the read accress time using the GPMC_CONFIG5_0_RDACCESSTIME field
- Step 6. Set up GPMC_CONFIG_6 register: configure write data on admux bus, write access time and other parameters
  - Set up write data on admux bus using the GPMC_CONFIG6_0_WRDATAONADMUXBUS field
  - Set up the write access time using the GPMC_CONFIG6_0_WRACCESSTIME field
- Step 7. Set up GPMC_CONFIG_7 register: configure the base address for the CS with the minimum size (16MB).

- Set up properly the GPMC_CONFIG7_0_BASEADDRESS, GPMC_CONFIG7_0_CSVALID and GPMC_CONFIG7_0_MASKADDRESS fields

These steps are implemented in a source code file which is called *module.c*. This file is included in a folder structure which is composed of the required files which compose a Makefile project. These files are necessary to build both the module and an example application using the module.

5.4.4.2 Asynchronous timings and features

The timing design in the GPMC for an asynchronous behavior is based on the chronogram specifications in the processor technical reference. Consider the following for a read operation:

- The valid address signal needs to be enabled once the destination address is set. The time instant the address is captured in the attached memory device falls between falling and rising edges of this signal at the same time that the output enable signal remains disabled.
- The time instant the data is read from the address and data bus is set using the read access time parameter. The address valid signal needs to be disabled before reading the data and output enable needs to be set.

The following chronogram has been used to design the timing values in the GPMC interface for a read operation in an asynchronous NOR 16 bit multiplexed memory and address device:

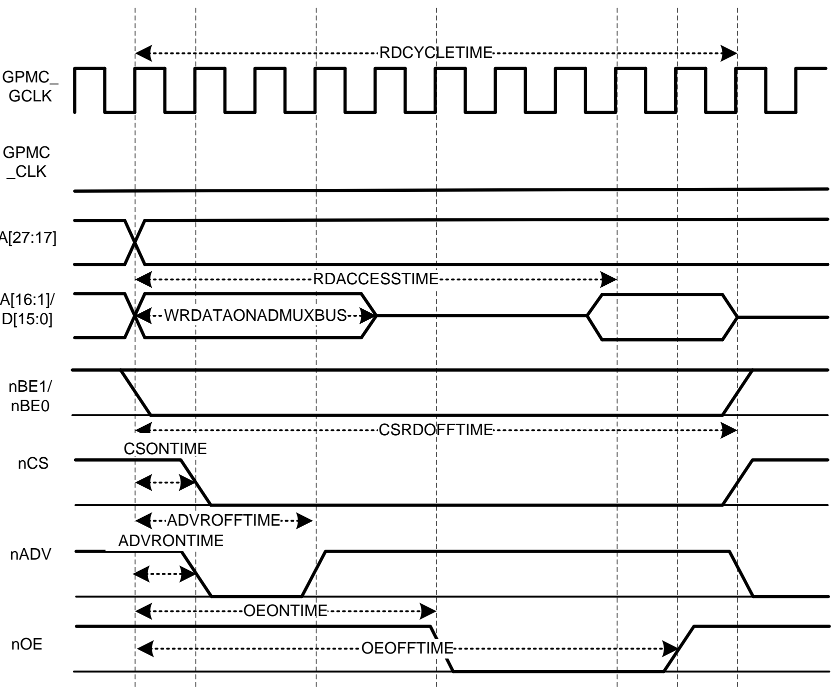


Fig 5.6: Asynchronous read operation

This is a summary of the timing values which have been set for this operation (values are given in clock cycle times):

RDCYCLETIME = 10	ADVROFFTIME = 3
RDACCESSTIME = 8	OEONTIME = 5
CSONTIME = 1	OEOFFTIME = 9
CSRDOFFTIME = 10	WRDATAONADMUXBUS = 4
ADVRONTIME = 1	

Table 5.3: Asynchronous read selected timings

For a write operation, consider the following:

- As well as for a read operation, the valid address signal needs to be enabled once the destination address is set. The time instant the address is captured in the attached memory device falls between falling and rising edges of this signal at the same time that the output enable signal remains disabled.
- The time instant the data is written to the address and data bus is set between write enable signal assertion and deassertion. Moreover, the address valid signal needs to be disabled before writing the data.

The following chronogram has been used to design the timing values in the GPMC interface for a write operation in an asynchronous NOR 16 bit multiplexed memory and address device:

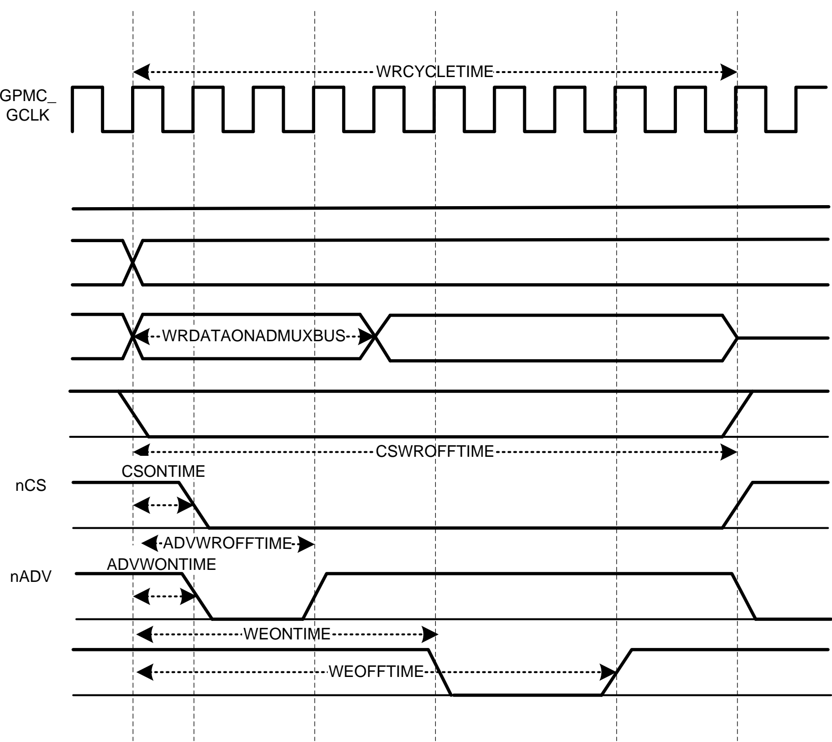


Fig 5.7: Asynchronous write operation

This is a summary of the timing values which have been set for this operation (values are given in clock cycle times):

WRCYCLETIME = 10	ADVWOFFTIME = 3
WRACCESSTIME = 0	WEONTIME = 5
CSONTIME = 1	WEOFFTIME = 8
CSWDOFFTIME = 10	WRDATAONADMUXBUS = 4
ADVWONTIME = 1	

Table 5.4: Asynchronous write selected timings

5.4.4.3 Build and probe the GPMC asynchronous module

The following is the folder structure of the Makefile project providing the GPMC module and an example application to test it:

```
├── application
│  ├── app.c
│  └── Makefile
├── Makefile
└── module
   ├── include
   │   ├── am33xx.h
   │   ├── cm3xxx.h
   │   ├── driver.h
   │   ├── gpmc.h
   │   ├── hw_cm_per.h
   │   ├── hw_gpmc.h
   │   ├── mux.h
   │   └── soc_AM335x.h
   ├── Makefile
   └── module.c
```

In order to compile these binaries with the new kernel and particular cross-compiler, some corrections had to be done. Here, a summary of these modifications is recalled:

- The KERNEL path in the module Makefile has to point to the kernel source code that was used to build the BeagleBone kernel. In this case:

```
KSRC=~/Workspace/bb-kernel/KERNEL
```

- The CROSS_COMPILE variable in the module Makefile has to point to the tool that is used to cross compile the kernel as well as the applications in the BeagleBone:

```
CROSS_COMPILE := arm-linux-gnueabihf-
```

- Almost two include paths had to be included in the module.c source code:

```
#include <asm/io.h> // in order to use “ioremap” directive
#include “am33xx.h” // in order to use “AM33XX_CTRL_BASE” macro
```

Building the module produces the following output:

```
# make
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -C ~/Workspace/bb-kernel/KERNEL
M=/home/ruben/Lab/driver_solucion/module modules
make[1]: Entering directory '/home/ruben/Workspace/bb-kernel/KERNEL'
 CC [M]  /home/ruben/Lab/driver_solucion/module/module.o
 Building modules, stage 2.
 MODPOST 1 modules
 CC      /home/ruben/Lab/driver_solucion/module/module.mod.o
 LD [M]  /home/ruben/Lab/driver_solucion/module/module.ko
```

```
make[1]: Leaving directory '/home/ruben/Workspace/bb-kernel/KERNEL'
```

The previous process produced a binary file which name is *module.ko*. It can be copied into the BeagleBone and be loaded in the kernel:

```
# insmod module.ko
```

Using the command *lsmod* it can be checked that the module has been loaded correctly.

5.4.5 Synchronous GPMC

The next task consists on introducing the work done before into a synchronous scenario. In order to do so, the designed module needs to be adapted, and the logics in the FPGA of the DE2 development board will need to be programmed and dumped as well.

The first step is to look for the changes that apparently concern to a synchronous communication. The technical reference provides some tips to program the timings and the registers which define the synchronous behavior. The chronograms in this manual about synchronous write and read operations in 16 bit NOR multiplexed devices are taken as reference.

Timing values in the resolution of the asynchronous module are kept for the synchronous model. Some additional fields in registers had to be set in order to indicate the new synchronous behavior. The following section is a proposal for the chronograms and their respective timings.

5.4.5.1 Synchronous timings and features

The timing design in the GPMC for a synchronous behavior is based on the chronogram specifications in the processor technical reference. Consider the following for a read operation:

- The valid address signal needs to be enabled once the destination address is set. The time instant the address is captured in the attached memory device falls between falling and rising edges of this signal at the same time that the output enable signal remains disabled.
- The time instant the data is read from the address and data bus is set using the read access time parameter. The address valid signal needs to be disabled before reading the data and output enable needs to be set.

The following chronogram has been used to design the timing values in the GPMC interface for a read operation in a synchronous NOR 16 bit multiplexed memory and address device:

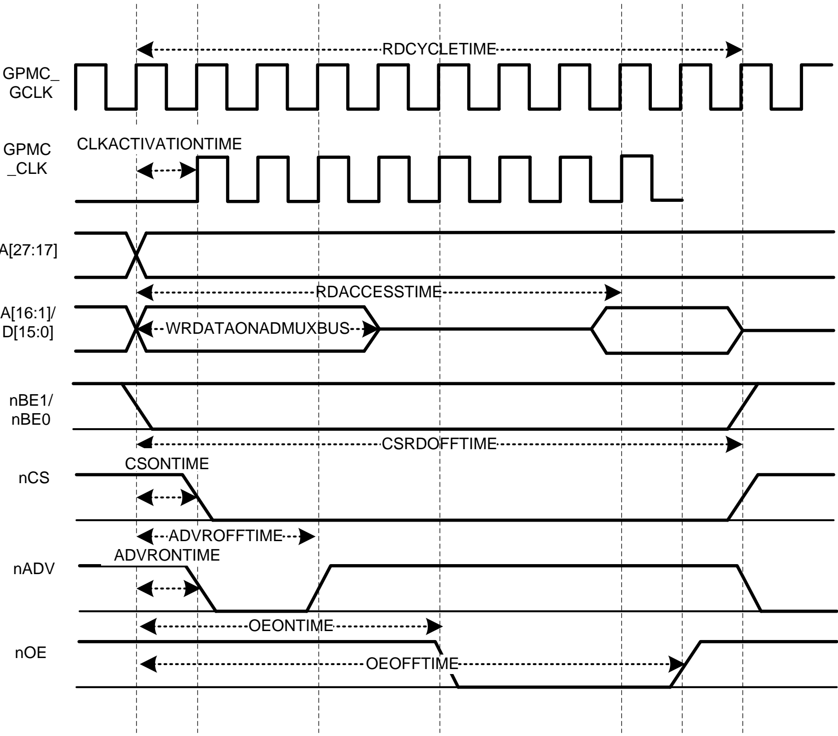


Fig 5.8: Synchronous read operation

This is a summary of the timing values which have been set for this operation (values are given in clock cycle times):

RDCYCLETIME = 10	ADVROFFTIME = 3
RDACCESSTIME = 8	OEONTIME = 5
CSONTIME = 1	OEOFFTIME = 9
CSRDOFFTIME = 10	WRDATAONADMUXBUS = 4
ADVRONTIME = 1	CLKACTIVATIONTIME = 1

Table 5.5: Synchronous write selected timings

For a write operation, consider the following:

- As well as for a read operation, the valid address signal needs to be enabled once the destination address is set. The time instant the address is captured in the attached memory device falls between falling and rising edges of this signal at the same time that the output enable signal remains disabled.
- The time instant the data is written to the address and data bus is set between write enable signal assertion and deassertion. Moreover, the address valid signal needs to be disabled before writing the data.

The following chronogram has been used to design the timing values in the GPMC interface for a write operation in a synchronous NOR 16 bit multiplexed memory and address device:

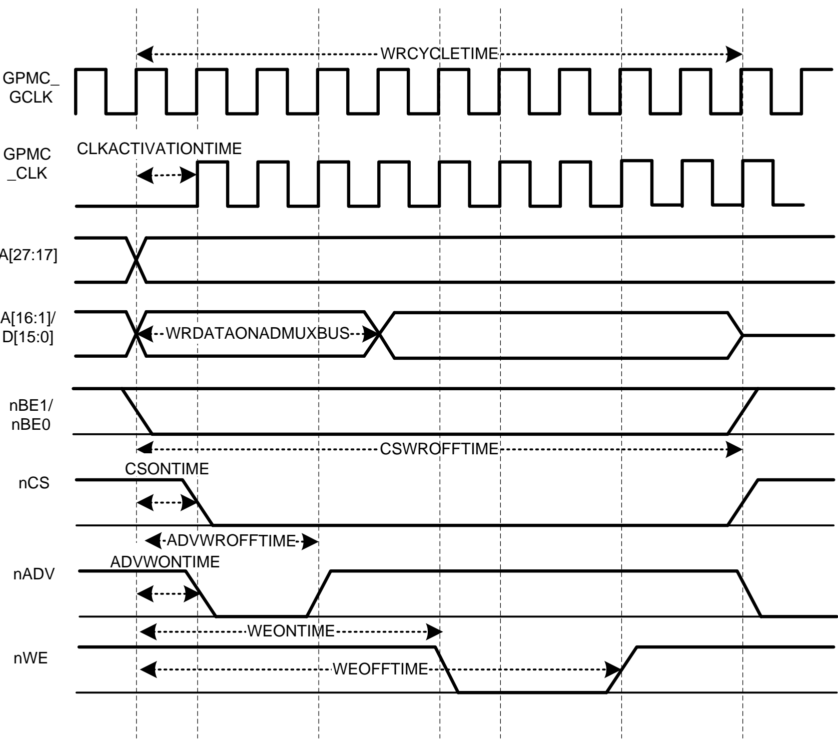


Fig 5.9: Synchronous write operation

This is a summary of the timing values which have been set for this operation (values are given in clock cycle times):

WRCYCLETIME = 10	ADVWOFFTIME = 3
WRACCESSTIME = 0	WEONTIME = 5
CSONTIME = 1	WEOFFTIME = 8
CSWDOFFTIME = 10	WRDATAONADMUXBUS = 4
ADVWONTIME = 1	CLKACTIVATIONTIME = 1

Table 5.6: Synchronous write selected timings

5.4.5.2 Build and probe the GPMC synchronous module

The following register gathers the modifications that need to be made in order to get the GPMC working synchronously. In particular, it is the register GPMC_CONFIG1_0 in step 1 depicted before:

- Step 1. Set up GPMC_CONFIG_1 register:
  - Set up the read type as synchronous using the GPMC_CONFIG1_0_READTYPE field
  - Set up the write type as synchronous using the GPMC_CONFIG1_0_WRITETYPE field
  - Set up the CLK activation time using the GPMC_CONFIG1_0_CLKACTIVATIONTIME field.

The fields GPMC_CONFIG1_0_READTYPE and GPMC_CONFIG1_0_WRITETYPE are used to indicate which type of access is performed. In the asynchronous design these values were set to 0. Now these values are set to GPMC_CONFIG1_0_READTYPE_RDSYNC and GPMC_CONFIG1_0_WRITETYPE_WRSYNC respectively. The field GPMC_CONFIG1_0_CLKACTIVATIONTIME indicates the delay in clocks in the output signal GPMC_FCLK. This value has been set to 1, but could also be set to 0.

The technical reference provides the description of these fields of the GPMC_CONFIG1_i register.

As done before, changes have also been introduced in the Makefiles in order to compile the drivers with the required kernel sources and cross compiler. Simple test applications doing some interactive write and read data operations with the GPMC bus have also been compiled and deployed in the BeagleBone board together with the drivers.

The module has been built, probed and in the following sections will be tested.

5.4.6 Burst GPMC

The next task consists on converting the previous synchronous GPMC interface into a burst synchronous one. In order to do so, the designed module needs to be adapted, and the logics in the FPGA of the DE2 development board will need to be programmed and dumped as well.

The first step is to look for the changes that concern to a synchronous burst communication. The technical reference provides some tips to program the timings and the registers which define the burst behavior. The chronograms in this manual about synchronous burst write and read operations in 16 bit NOR multiplexed devices are taken as reference.

5.4.6.1 Burst timings and features

The timing design in the GPMC for a synchronous burst behavior is based on the chronogram specifications in the processor technical reference. These are the new timings and parameters to be considered compared to a non-burst read access:

READCYCLETIME = READCYCLETIME0 + READCYCLETIME1	
PAGEBURSTACCESTIME	(GPMC_CONFIG5_i)
CSRDOFFTIME = CSRDOFFTIME0 + CSRDOFFTIME1	
WRAPBURST	(GPMC_CONFIG1_i)
ATTACHEDDEVICEPAGELENGTH	(GPMC_CONFIG1_i)

Table 5.7: New timings in a burst read operation

Consider the following for a read operation:

- Chip select, address valid and output enable signals are called in the same way as in a synchronous read operation.
- When the read access time is completed, control-signal timings are frozen during the multiple data transactions, corresponding to the page bust access time multiplied by the number of remaining data transactions.
- Initial latency for the first read data is controlled by the read access time. Successive read data are provided by the attached memory device each one or two clock cycles. The page burst access time must be set accordingly with the GPMC clock divider and the memory internal configuration.
- Burst wraparound can be enabled, and allows accesses of 4, 8 or 18 word bursts wrapped within its burst-length boundary.

For timing design, the model of the asynchronous module has been taken as reference. Some common parameters have been modified whereas new ones have been introduced. The following picture shows the waveforms that have been programmed:

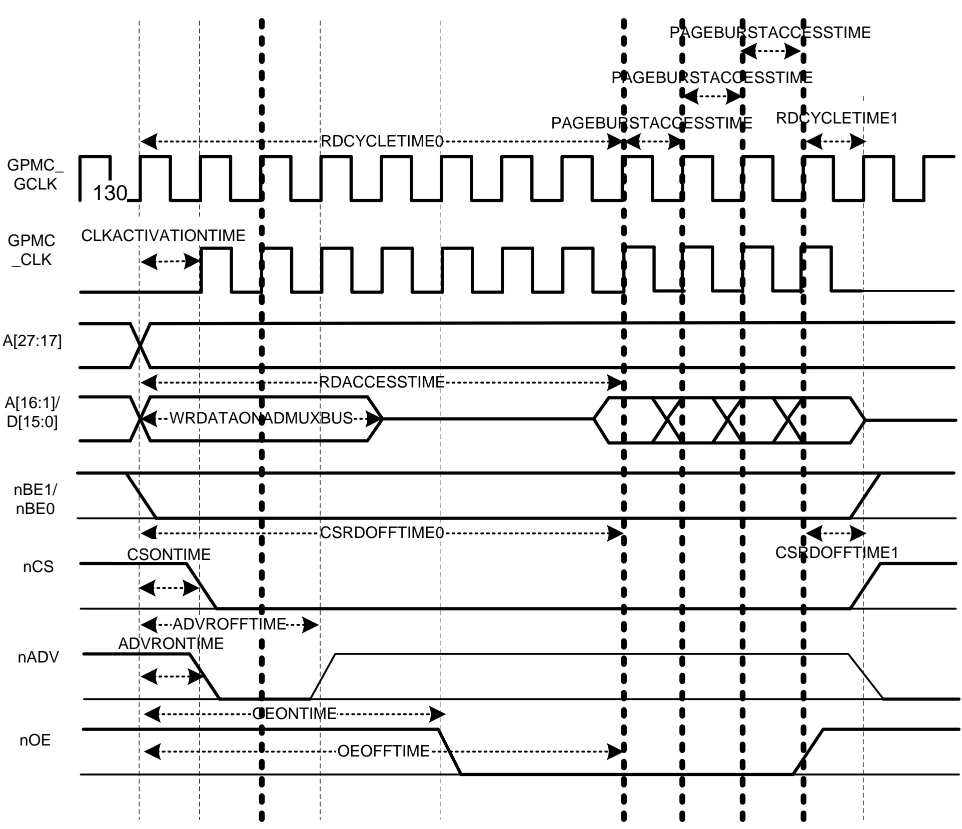


Fig 5.10: Burst read operation

Note that the first vertical thick dotted line indicates the instant of address capture whereas the other four vertical thick dotted lines indicate the instants of each word data capture.

These are the timings that have been introduced in the design:

RDCYCLETIME = 9	ADVROFFTIME = 3
PAGEBURSTACCESSTIME = 1	OEONTIME = 5
RDACCESSTIME = 8	OEOFFTIME = 8
CSONTIME = 1	WRDATAONADMUXBUS = 4
CSRDOFFTIME = 9	CLKACTIVATIONTIME=1
ADVONTIME=1	

Table 5.8: Synchronous burst read selected timings

In addition, these two parameters have been programmed:

- WRAPBURST (GPMC_CONFIG1_i): Enables the wrapping burst capability. Must be set if the attached device is configured in wrapping burst. This parameter has been set to 0, so no wrapping is supported initially.
- ATTACHEDDEVICEPAGELENGTH (GPMC_CONFIG1_i): Specifies the attached device page (burst) length (1 Word = Interface size). This parameter has been set to 0, which means 4-word burst and no wrapping as a consequence.

For a read operation, these are the new timings and parameters to be considered compared to a non bursted read access:

WRACCESSTIME

This parameter was not necessary for a single asynchronous write operation. It was set to 0. Now things are different in a burst synchronous access scenario. This time needs to be set in order to indicate when to start switching values in the data bus. The technical reference provides more details about programming the required parameters for a write access operation.

For the design of the timings, the model of the asynchronous module has been taken as reference. As well as in the burst read operation, some common parameters have been slightly modified whereas new ones have been introduced. The following picture shows the waveforms that have been programmed:

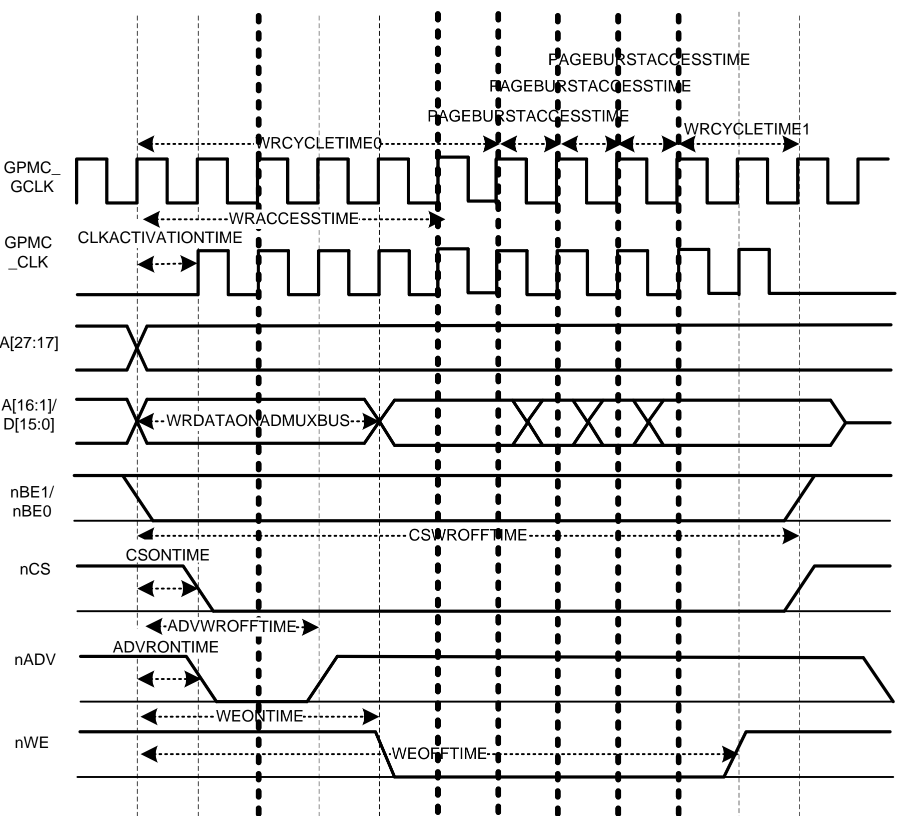


Fig 5.11: Burst write operation

Note that the first vertical thick dotted line indicates the instant of address capture whereas the other four vertical thick dotted lines indicate the instants of each word of data writing.

These are the timings that have been introduced in the code:

WRCYCLETIME = 8	ADVROFFTIME = 3
PAGEBURSTACCESSTIME = 1	WEONTIME = 4
WRACCESSTIME = 5	WEOFFTIME = 10
CSONTIME = 1	WRDATAONADMUXBUS = 4
CSWROFFTIME = 11	CLKACTIVATIONTIME = 1
ADVONTIME = 1	

Table 5.9: Synchronous burst write selected timings

5.4.6.2 Build and probe the GPMC synchronous module

The following register gathers the modifications that need to be made in order to get the GPMC working synchronously and bursted. In particular, it is the register GPMC_CONFIG1_0 in step 1 depicted before:

- Step 1. Set up GPMC_CONFIG_1 register:
    - Set up the wrapping bust to not supported using the GPMC_CONFIG1_0_WRAPBURST field
    - Set up multiple access for reading using the GPMC_CONFIG1_0_READMULTIPLE field
    - Set up the read type as synchronous using the GPMC_CONFIG1_0_READTYPE field
    - Set up multiple access for writing using the GPMC_CONFIG1_0_WRITEMULTIPLE field
    - Set up the write type as synchronous using the GPMC_CONFIG1_0_WRITETYPE field
    - Set up the CLK activation time using the GPMC_CONFIG1_0_CLKACTIVATIONTIME field.
    - Set up the attached device page length to 4 words with the ATTACHEDDEVICEPAGELENGTH field

The fields GPMC_CONFIG1_0_READTYPE and GPMC_CONFIG1_0_WRITETYPE are used to indicate which type of access is performed. In the asynchronous design these values were set to 0. Now these values are set to GPMC_CONFIG1_0_READTYPE_RDSYNC and GPMC_CONFIG1_0_WRITETYPE_WRSYNC respectively. The field GPMC_CONFIG1_0_CLKACTIVATIONTIME indicates the delay in clocks in the output signal GPMC_FCLK. This value has been set to 1, but could also be set to 0.

In addition, the GPMC_CONFIG1_0_WRAPBURST field has also been explicitly set to not support synchronous wrapping in the attached memory device. The fields GPMC_CONFIG1_0_READMULTIPLE and GPMC_CONFIG1_0_WRITEMULTIPLE have been set to enable reading and writing in multiple synchronous burst mode, respectively.

The technical reference provides the description of these fields of the GPMC_CONFIG1_i register.

One more time, changes have also been introduced in the Makefiles in order to compile the drivers with the required kernel sources and cross compiler. Simple test applications doing some interactive write and read data operations with the GPMC bus have also been compiled and deployed in the BeagleBone board together with the drivers.

The module has been built, probed and in the following sections will be tested.

5.5 Cape programs

The cape TT01v1 came with a set of executable applications for the Angstrom system which were depicted to demonstrate the different functionalities of this cape. Each program implements a different feature, giving an easy relation of the cape capabilities.

A quick overview of these programs is given among the following sections also explaining the modifications that were necessary for them to run properly in the new Ubuntu environment. The objective is to get them working with the minimum changes.

5.5.1 Previous work: regenerate the cape EPROM contents

At this point, some information from the Angstrom system that was previously saved for future work needs to be retrieved. This moment has arrived. The directory containing the cloud9 file system has to be copied in the new image of the BeagleBone that runs Ubuntu. The destination directory with the original contents in /var/lib/cloud9 of the Angstrom system are now also in /var/lib/cloud of the Ubuntu system.

The next step is to check out if the application node.js, which produces the contents for the EEPROM, comes installed with the original contents of the current Ubuntu distribution.

```
# aptitude search nodejs
p   nodejs                                                    - evented I/O for V8 javascript
[...]
```

The "p" before the package name indicates that the package exists in the repository cache but it is not installed in the system. In order to solve this issue, the BeagleBone board is connected to internet and the package is downloaded and installed:

```
# aptitude install nodejs
```

Here, *nodejs* is an analog application to *node*, which was previously tested in Angstrom.

Once installed *nodejs*, the next step is to produce the binary contents for the EEPROM. This can be achieved by executing the following instruction in the location of the cape-bone-TT01v1.json script:

```
# cd /var/lib/cloud9/eeprom/bonescript-master/node_modules/bonescript
# cp cape-bone-TT01v1 cape-bone-TT01v1.bak
# nodejs ./eeprom.js -w cape-bone-TT01v1.json
```

Before generating the new file, the old one has been renamed with the *.bak* suffix. It has been done in order to compare the newly generated file with the old one. The comparison of these two binary files can be easily done executing a checksum of these two files. In particular, *md5sum* can be used:

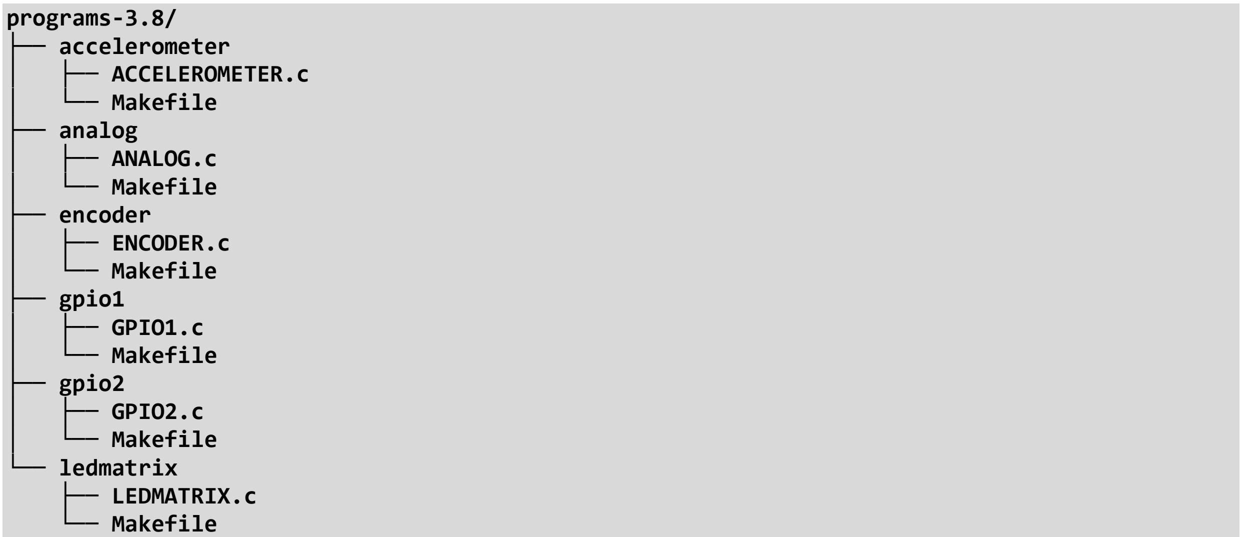
```
# md5sum cape-bone-TT01v1
f4fb9f287aabf3f4c54daea4c06d5b28  cape-bone-TT01v1
# md5sum cape-bone-TT01v1.bak
f4fb9f287aabf3f4c54daea4c06d5b28  cape-bone-TT01v1.bak
```

This application demonstrates that the new and old files are exactly the same because both of them generate the same checksum. Having viewed that, the conclusion is that it is not necessary to generate and dump the contents of the EEPROM for the new system.

5.5.2 Building the cape programs

The source code files of a set of applications using the cape-bone-TT01v1 have been retrieved from the legacy Angstrom system micro SD card. In particular, there is the source code of GPIO1, GPIO1, ENCODER, ANALOG, LEDMATRIX and ACCELEROMETER applications. These source codes have been copied to the development environment in order to compile them and generate new binaries for the BeagleBone.

Instead of creating a new project in the Eclipse IDE (or other similar IDE) the construction of a new executable is based on Makefile files. The following is the file folder structure for the applications to be build:



```
programs-3.8/
├── accelerometer
│   ├── ACCELEROMETER.c
│   └── Makefile
├── analog
│   ├── ANALOG.c
│   └── Makefile
├── encoder
│   ├── ENCODER.c
│   └── Makefile
├── gpio1
│   ├── GPIO1.c
│   └── Makefile
├── gpio2
│   ├── GPIO2.c
│   └── Makefile
└── ledmatrix
    ├── LEDMATRIX.c
    └── Makefile
```

Each Makefile has been adapted depending on the filename in the source code and the name of the binary executable file they generate.

Using an IDE has been discarded due to the fact that the applications are basically composed of a rather short single source code file. It is more practical editing the source code using a simple editor such as Vi and compiling the code from a command line shell. IDEs such as Eclipse are better in large projects which require indexation among multiple source code files.

The following is an example of the resulting Makefile proposed to build or clean the application just by executing the make or make clean commands, respectively:

```
# Emdedded Linux Makefile
# Project: accelerometer

ARM=arm
CROSS_COMPILE=arm-linux-gnueabihf-

PWD     := $(shell pwd)

CC=$(CROSS_COMPILE)gcc
CFLAGS=-c -Wall -O2 $(INCLUDE)
LDFLAGS=-lpthread

SOURCES=ACCELEROMETER.c

EXECUTABLE=ACCELEROMETER
```

```
OBJECTS=$(SOURCES:.c=.o)

INCLUDE=\
	-I. \
	-Iinclude

# Make rules

all: $(SOURCES) $(EXECUTABLE)

clean:
	rm -f $(PWD)/*.o $(PWD)/$(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
	$(CC) $(LDFLAGS) $(OBJECTS) -o $@

.c.o:
	$(CC) $(CFLAGS) $< -o $@
```

In the example given, executing make on the project root folder will produce the object file ACCELEROMETER.o and the executable file ACCELEROMETER. Executing make clean will erase these two binary files.

The binary files obtained from the compilation process have been copied to the BeagleBone and tested. The next section introduces the results of the execution and the actions taken to correct software defects, if any.

5.5.3 Inspection of the cape programs behavior in the new environment

In order to test the cape programs, the original cape overlay **cape-bone-TT01v1-00A0.dtbo** has been maintained and loaded into the BeagleBone with Ubuntu through the cape manager. Results explained in previous sections have showed that the overlay is compatible with the device tree in the system.

The execution of the binaries using the cape in the section before has resulted satisfactory in all cases except in the analog application. The next table summarizes these results:

Application	Pin or device	Function	Status
GPIO1	gpio48	LED	**ENABLED**
GPIO2	gpio48	LED	**ENABLED**
	gpio51	LED	**ENABLED**
	gpio50	LED	**ENABLED**
	gpio20	Push button	**ENABLED**
ANALOG	AINx	Analog input	**WRONG**
	gpio20	Push button	**ENABLED**
	S1	Switch	**ENABLED**
ENCODER	gpio66	Rotating clockwise input	**ENABLED**
	gpio69	Rotating counterclockwise	**ENABLED**
	S2	Switch GPMC	**ENABLED**

LEDMATRIX	SPI interface	Manage Shift Register	ENABLED
	gpio20	Push button	ENABLED
ACCELEROMETER	I2C interface	R/W accelelerometer registers	ENABLED
	gpio20	Push button	ENABLED

Table 5.10: Summary of results in cape program execution with the current cape overlay

In order to detect why the analog program fails, a simple test can be done. The cape-bone-iio-00A0.dtbo overlay can be loaded alone using the cape manager, that is, without loading cape-bone-TT01v1-00A0.dtbo . Then the analog program is executed:

- If the analog reading fails, it means that the program has a defect that produces that bug
- If the analog reading is proper, it means that there is a mistake somewhere in the cape-bone-TT01v1-00A0 device tree source file description.

Before loading the system analog overlay and executing the application, it can be seen that the application runs properly. It provides the values of the analog input in pin AIN0 (which depend on the voltage that is set through a switch in the cape).

5.5.4 Integrating both cape-bone-TT01v1 and cape-bone-iio capes

In order to get all the possible features working at the same time, some experiments have been considered. Recall the following:

- The cape worked perfectly only with the custom binary tree cape-bone-TT01v1-00A0 in Angstrom. Its corresponding *.dts file also included the tree provided in the source cape-bone-iio-00A0.dts.
- When testing same cape in Ubuntu with the same binary, it was found that all the features, except one, worked correctly. The feature which failed was just the analog input.
- Isolating the analog binaries, i.e., using the cape-bone-iio-00A0.dtbo which comes with the kernel, it was found that the analog program using this feature worked perfectly.

A first overview of the device trees shows that the code in cape-bone-TT01v1-00A0.dts also includes what is depicted in cape-bone-iio-00A0.dts. So, the idea that emerges is to compare both overlay source codes and see what is missing from the analog overlay to the cape-bone-TT01v1-00A0.

A quick overview of both files reveals the following information about the analog section:

- Analog section in device tree cape overlay:

```
[…]
        fragment@4 {
                target = <&ocp>;
                __overlay__ {
                        /* avoid stupid warning */
                        #address-cells = <1>;
                        #size-cells = <1>;
```

```
                        tscadc {
                                compatible = "ti,ti-tscadc";
                                reg = <0x44e0d000 0x1000>;

                                interrupt-parent = <&intc>;
                                interrupts = <16>;
                                ti,hwmods = "adc_tsc";
                                status = "okay";

                                adc {
                                        ti,adc-channels = <8>;
                                };
                        };
                };
        };
[…]
```

- Analog section in cape-bone-iio overlay:

```
[…]
        fragment@0 {
                target = <0xdeadbeef>;
                __overlay__ {
                        #address-cells = <0x1>;
                        #size-cells = <0x1>;

                        tscadc {
                                compatible = "ti,ti-tscadc";
                                reg = <0x44e0d000 0x1000>;
                                interrupt-parent = <0xdeadbeef>;
                                interrupts = <0x10>;
                                ti,hwmods = "adc_tsc";
                                status = "okay";

                                adc {
                                        ti,adc-channels = <0x0 0x1 0x2 0x3 0x4 0x5 0x6 0x7>;
                                };
                        };

                        helper {
                                compatible = "bone-iio-helper";
                                vsense-name = "AIN0", "AIN1", "AIN2", "AIN3", "AIN4", "AIN5",
"AIN6", "AIN7";
                                vsense-scale = <0x64 0x64 0x64 0x64 0x64 0x64 0x64 0x64>;
                                status = "okay";
                                linux,phandle = <0x1>;
                                phandle = <0x1>;
                        };
                };
        };
[…]
```

Note that the version of cape-bone-iio-00A0.dts is the one that comes with the kernel version in Robert C. Nelson official repository, kernel branch 3.8. Two main differences can be easily appreciated:

- The analog channels are defined differently in both overlay source files
- The cape-bone-iio-00A0.dts source also provides a sub section in the tree which is called "*helper*". This section is not relevant by the moment, but later in this project it will.

The problem integrating the analog input functionality into cape-bone-TT01v1-00A0.dts had to be easy to resolve. This is not the first time someone has found this problem, and the solution for a proper configuration is provided in Internet forums [5.9].

From this information and following the analogy between am335x-bone.dts and am335x-boneblack.dts, it was found that the code of the cape-bone-TT01v1-00A0.dts needed fixing in a particular field: the description of the channels used. This field is a list with the channels that will be enabled. Possible values are in the range from 0 to 7 (8 in total). Number 8, which was the previous value, is not allowed. For this reason the cape did not load any analog input capability.

In order to allow analog inputs, the section "*exclusive-use*" of the cape-bone-TT01v1-00A0.dts device tree sources has been enhanced including the required pins that refer to AIN0 to AIN7. As depicted above, the vector in section "adc" in "tscadc" (fragment 4) has been modified in order to include channels 0 to 7.

To fix this issue, the following are the changes introduced in the device binary source code from cape-bone-TT01v1-00A0.dts:

```
[…]
	/* state the resources this cape uses */
	exclusive-use =
		/* the pin header uses */
		"P9.18",	/* i2c1: i2c1_sda */
		"P9.17",	/* i2c1: i2c1_scl */
		"P9.39",	/* AIN0 */
		"P9.40",	/* AIN1 */
		"P9.37",	/* AIN2 */
		"P9.38",	/* AIN3 */
		"P9.33",	/* AIN4 */
		"P9.36",	/* AIN5 */
		"P9.35",	/* AIN6 */
		"P9.14",	/* led: led3 */
		"P9.15",	/* led: led1 */
[…]

[…]
	fragment@4 {
		target = <&ocp>;
		__overlay__ {
			/* avoid stupid warning */
			#address-cells = <1>;
			#size-cells = <1>;

			tscadc {
				compatible = "ti,ti-tscadc";
				reg = <0x44e0d000 0x1000>;

				interrupt-parent = <&intc>;
				interrupts = <16>;
				ti,hwmods = "adc_tsc";
				status = "okay";

				adc {
					ti,adc-channels = <0 1 2 3 4 5 6 7>;
				};
			};
		};
	};
};
[…]
```

These sources can now be compiled and the resulting binary overlay can be loaded in the BeagleBone using the cape manager:

```
# dtc -O dtb -o cape-bone-TT01v1-00A0.dtbo -b 0 -@ cape-bone-TT01v1-00A0.dts
```

Copy the overlay file in /lib/firmware and load it:

```
# cd /sys/devices/bone_capemgr.8
# echo cape-bone-iio > slots
# cat slots
0: 54:PF---
1: 55:PF---
2: 56:PF---
3: 57:P---L cape-bone-TT01v1,00A0,TuDuTech,cape-bone-TT01v1
```

5.5.5 Adapting the cape programs

The behavior of the applications once compiled with the new cross compiler toolchain and deployed to the BeagleBone is almost perfect, except for the analog application whose changes have been explained before. This is due to the fact that the system and application libraries used in Angstrom and Ubuntu embedded are the same or almost the same; at least they are compatible.

Despite that the code of these applications can be optimized, no modifications have been done.

5.6 Python

Python is a scripting language which is widely used at present due to its general-purpose nature. It is presented as a high-level programming language, but also includes functionalities which allow low-level programming. The focus of its syntax is put on easing code readability and reducing complexity, allowing writing shorter codes than those expected in other high-level languages.

5.6.1 Introducing Python

The following are some of the features that define Python:

- It is an interpreted multiplatform language
- The type casting is dynamic
- Files containing Python code end with .py
- Indentations are mandatory in order to define sections of code

Python is constituted as an ideal language to quickly develop prototypes in a scenario of an embedded device such as the BeagleBone. In general, primitives are intuitive and can be used by someone introduced to programming. In this context, it is easy to write a code which handles a device in a few lines.

With Python in the BeagleBone, it is not necessary to build, cross-compile and deploy an application. Just write myapp.py code in an editor of the BeagleBone and execute the resulting script in a console:

```
# python myapp.py
```

In order to deal with some of the capabilities of the BeagleBone ARM processor such as the management of the I/O pins, the *Adafruit Company* has developed a free library which provides a set of tools to handle them. This library is called *Adafruit BBIO* and some of its features will be explained in the following sections.

Adafruit was founded in 2005 by MIT engineer, Limor "Ladyada" Fried. Her goal was to create the best place online for learning electronics and making the best designed products for makers of all ages and skill levels. For more information refer to https://www.adafruit.com/

5.6.2 Adafruit Python modules for BeagleBone

The BeagleBone board has two expansion headers that are composed of 46 pins each one. These pins are available to be configured to be used isolated or as a part of a communications subsystem. Not only pin configuration for GPIO purposes is available. Some of the supported functions are:

- 7 analog input pins
- 2 I2C buses
- 2 SPI interfaces
- 2 CAN buses
- 4 timers
- 4 UART interfaces
- 8 PWM interfaces
- A/D Converter
- And of course 65 GPIO pins at 3.3V

Despite at present Adafruit's BeagleBone Python library does not support all of the functionalities above, many of them are available and can be used to test the peripherals included in the BeagleBone cape. It is expected that the library grows up and includes new features progressively.

The following is the list of disposable capabilities in this library:

- AIN: Analog Inputs
- I2C: Inter-Integrated Circuit
- SPI: Serial Peripheral Interface
- UART: Universal Asynchronous Receiver-Transmitter
- PWM: Pulse Width Modulator
- GPIO: General Purpose Input Output

See the library description and source code in the public GitHub repository [5.10].

5.6.3 Installing Python and Python modules

Before installing Python in the BeagleBone in the current development environment, the board needs to be connected to internet. This is done giving it a fixed IP address within the network of the router and setting the name servers to point to Google ones, as explained before. The procedure that is shown below assumes that Python is being installed in a BeagleBone with Ubuntu system using the 3.8 branch of the Linux kernel.

The first step is to set an accurate date and time in the BeagleBone. It can be done manually using the application date or synchronizing the system date and time using a NTP server:

```
# ntpdate pool.ntp.org
```

The next step is to install *build-essential* tools and *Python* packages, as well as its dependencies:

```
# apt-get update
# apt-get install build-essential python-dev python-setuptools python-pip python-smbus
```

At this point one can start writing code in Python language and executing it in the BeagleBone. Nevertheless, to get the devices in the cape working, the BBIO Adafruit library needs to be installed.

5.6.3.1 Installing the BBIO Adafruit Python modules

Python has its own package installer which is called *pip*. In this case, it can be used to get and install Adafruit BBIO Python library:

```
# pip install Adafruit_BBIO
```

Alternatively, the library can be cloned from GitHub and installed in the system with the same result:

```
# git clone git://github.com/adafruit/adafruit-beaglebone-io-python.git
# cd adafruit-beaglebone-io-python
# python setup.py install
```

These steps are enough to start developing Python applications directly in the BeagleBone which use the BBIO module provided by Adafruit.

5.6.4 Writing TT01 Cape programs to Python. Adafruit modules

5.6.4.1 Using the GPIOs

The Adafruit_BBIO Python library provides a simple interface to handle GPIOs called GPIO. After importing this module, some functions are available:

- setup: sets the direction of a GPIO pin (IN or OUT)
- output: writes the value of a GPIO set as output (HIGH or LOW)
- input: reads the value of a GPIO set as input
- cleanup: restores the original configuration of the GPIO pinout

The following is a test program that switches on a LED during three seconds. This LED is present on the cape with the label LD1 and it is bound to P9_15 pin in the BeagleBone. In addition, a timer has also been imported for sleep function:

```
#-------------------------------------------------------------------------------
# Code of application GPIO1.py

import Adafruit_BBIO.GPIO as GPIO
import time

GPIO.setup("P9_15", GPIO.OUT)
GPIO.output("P9_15", GPIO.HIGH)
time.sleep(3)
GPIO.cleanup()
#-------------------------------------------------------------------------------
```

The following code is similar to the previous one but including a GPIO pin set as input. The application makes LEDs labeled as LD1, LD2 and LD3 go blinking each second until a push button (also present in the cape) is maintained pressed. The application remains checking the value of the P9_41 pin in a while loop. This pin is set as input and bound to a physical push button in the cape. When the push button is kept pressed, the value of this pin is set to 1, the while condition is not met, and then the application ends.

```
#-------------------------------------------------------------------------------
# Code of application GPIO2.py

import Adafruit_BBIO.GPIO as GPIO
import time

GPIO.setup("P9_15", GPIO.OUT)
GPIO.setup("P9_16", GPIO.OUT)
GPIO.setup("P9_14", GPIO.OUT)
GPIO.setup("P9_41", GPIO.IN)

output=1
while (GPIO.input("P9_41") == 0):
	if output:
		GPIO.output("P9_15", GPIO.HIGH)
		GPIO.output("P9_16", GPIO.HIGH)
		GPIO.output("P9_14", GPIO.HIGH)
		output=0
	else:
		GPIO.output("P9_15", GPIO.LOW)
		GPIO.output("P9_16", GPIO.LOW)
		GPIO.output("P9_14", GPIO.LOW)
		output=1
	time.sleep(1)

GPIO.cleanup()
#-------------------------------------------------------------------------------
```

In order to go deeper in the capabilities provided by Python in prototyping applications, some improvements can be done in the codes shown before. For instance, the implementation of the push button code can be done without polling the value of its associated pin at each while cycle. It can be done setting and unsetting an event associated to the change of the state value of its pin. In order to do so, the Adafruit Python library provides the following functions:

- add_event_detect: detects the state change of an associated pin at RISING, FALLING or BOTH edges, and executes an associated callback function at an expected bounce time
- remove_event_detect: removes the event associated to a state change in a pin

A common code should also have a mechanism of exceptions in order to protect the system of undesired or unexpected behaviors.

- try: this label indicates a portion of code which is going to be executed and could throw some type of exception. If an exception is thrown, the rest of the code is aborted immediately and the execution falls into the except label
- except: this label executes the depicted set of procedures under it in case that one of the procedures under the try label throws an exception
- raise: this identifier is normally put at the end of a set of primitives executed at the except section. It forces the exception to happen and to be raised to the section or module which called this code

In addition, functions are also used in order to ease clean and reusable code in Python, as well in many other compiled or interpreted languages.

The following section is showing a code of a slightly more complex application which interprets a rotary encoder. It is based in both falling and rising edge detection of two signals which are simulated through two associated input pins. These pins, P8_7 and P8_9, are connected to a rotating switch generating the following signals, depending on the rotation direction:

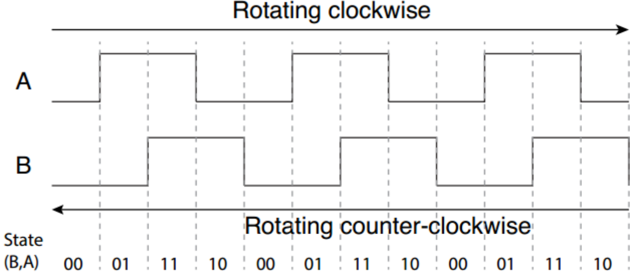


Fig 5.12: Signals generated by the rotating encoder depending on the rotating direction

Due to the fact that the encoder shares some GPIO pins with the GPMC, the 3 switches of the cape that are named GMPC EN (switch 2) must be set to ON.

```
#---------------------------------------------------------------------------------
# Code of application ENCODER.py

import Adafruit_BBIO.GPIO as GPIO
import time

GPIO.setup("P8_7", GPIO.IN)
GPIO.setup("P8_9", GPIO.IN)
GPIO.setup("P9_41", GPIO.IN)
pushed=0

def fun_push (pin):
       global pushed
       pushed=1

def fun_encode_A(pin):
       print "ready: "
       print "\tLead A"
       print "\t\tA: ", GPIO.input("P8_7"), "  B: ", GPIO.input("P8_9")
```

```
def fun_encode_B(pin):
	print "ready: "
	print "\tLead B"
	print "\t\tA: ", GPIO.input("P8_7"), "  B: ", GPIO.input("P8_9")

try:
GPIO.add_event_detect("P8_7",GPIO.BOTH,callback=fun_encode_A,bouncetime=20)
GPIO.add_event_detect("P8_9",GPIO.BOTH,callback=fun_encode_B,bouncetime=20)
	GPIO.add_event_detect("P9_41",GPIO.RISING,callback=fun_push,bouncetime=20)

	while (pushed==0):
		time.sleep(0.1)

except:
	GPIO.remove_event_detect("P8_7")
	GPIO.remove_event_detect("P8_9")
	GPIO.remove_event_detect("P9_41")
	GPIO.cleanup()
	raise

GPIO.remove_event_detect("P8_7")
GPIO.remove_event_detect("P8_9")
GPIO.remove_event_detect("P9_41")
GPIO.cleanup()
#--------------------------------------------------------------------------------
```

Note the following features:

- The code that is used to finish the application and to free resources is replicated under the except label, and will be executed if an exception arises
- The program remains in a while loop until an event occurs. If events are both edge detection in state change of P8_7 and P8_9, these events are attended and the application remains looping. If the event is a rise edge detection in P9_41, the global variable pushed is set to 1 and so the execution escapes the while loop
- Events are detected and handled through their corresponding callback function:
    - fun_encode_A and fun_encode_B prints the current value of P8_7 and P8_9, and the rotating direction of the encoder
    - fun_push detects the push button associated pin P9_41 state change and sets a global variable in order to aid finishing the application

The following is an example of the output shown by the terminal console when rotating one step clockwise the encoder:

```
ready:
	Lead B
		A:  1   B:  0
ready:
	Lead A
		A:  0   B:  0
ready:
	Lead B
		A:  0   B:  1
ready:
	Lead A
		A:  1   B:  1
```

And this is an example of the output shown by the terminal console when rotating one step counterclockwise the encoder returning to the starting point:

```
ready:
       Lead A
               A:  0   B:  1
ready:
       Lead B
               A:  0   B:  0
ready:
       Lead A
               A:  1   B:  0
ready:
       Lead B
               A:  1   B:  1
```

5.6.4.2 Using the ADC

In order to use the ADC capabilities of the Adafruit_BBIO Python library it is required to import the ADC module into the script. After that, the following functions are available:

- setup: starts the BeagleBone analog capabilities
- read: reads a normalized value of the input voltage between 0 and 1 in the selected channel. It is possible to convert it to the actual voltage multiplying the value by 1.8
- read_raw: reads the raw value of the selected channel

The following program reads every two seconds the value of the analog input depicted as AIN0 (pin P9_39). This value is printed on the terminal in raw and normalized formats, as well as in actual voltage values.

```
#--------------------------------------------------------------------------------
# Code of application ANALOG.py

import Adafruit_BBIO.GPIO as GPIO
import Adafruit_BBIO.ADC as ADC
import time

GPIO.setup("P9_41", GPIO.IN)

ADC.setup()

while (GPIO.input("P9_41") == 0):

       value_raw = ADC.read_raw("AIN0")  #"P9_39" also valid
       print "AINO raw reading:", int(value_raw)

       value = ADC.read("P9_39")
       print "AINO normalized value:", value
       print("AINO voltage [V]: %.2f" % (value * 1.8))

       print ""
       print "keep pushbutton pressed to break"
       print ""

       time.sleep(2)

GPIO.cleanup()
#--------------------------------------------------------------------------------
```

The following is an example of the result printed on the console terminal:

```
AINO raw reading: 656
AINO normalized value: 0.365000009537
AINO voltage [V]: 0.66

keep pushbutton pressed to break

AINO raw reading: 657
AINO normalized value: 0.365000009537
AINO voltage [V]: 0.66

keep pushbutton pressed to break

[...]
```

Note that the raw reading has been formatted to an integer value given that the print function provides it as a float value by default. The actual voltage has also been formatted to two digits of decimal precision in order to ease the reading.

Changing the position of the switches in the analog switch of the cape alters the composition of the resistive voltage divider at the input of AIN0 and so the read values vary.

Important device tree modification to use Adafruit's Python analog library

Before using Python to test the analog input, the applications written in C did not expect a piece of the device tree description which now is required to use the analog capabilities through Adafruit BBIO BeagleBone modules. This part is the analog *helper*.

When executing the Python code without the given helper, an error is thrown:

```
Traceback (most recent call last):
 File "ANALOG.py", line X, in <module>
   adc.setup()
RuntimeError: Unable to setup ADC system. Possible causes are:
 - A cape with a conflicting pin mapping is loaded
 - A device tree object is loaded that uses the same name for a fragment: helper
```

The helper has the functionality of aliases AIN0 to AIN7 and voltage scale to convert the reading to millivolts for each analog channel. In fact, the analog readings are registered in the files "/sys/bus/platform/devices/tiadc/iio:device0/in_voltageX_raw" (in which X is equal to the number of the analog input), which can be acceded just reading them. This helper was introduced in previous sections of this document modifying the *.dtb file of the cape. Note that it is necessary to include this helper in the cape overlay, rebuild and deploy it to the BeagleBone, as explained before.

```
[…]
	fragment@4 {
		target = <&ocp>;
		__overlay__ {
			/* avoid stupid warning */
			#address-cells = <1>;
			#size-cells = <1>;

			tscadc {
				compatible = "ti,ti-tscadc";
				reg = <0x44e0d000 0x1000>;
```

```
                                interrupt-parent = <&intc>;
                                interrupts = <16>;
                                ti,hwmods = "adc_tsc";
                                status = "okay";

                                adc {
                                        ti,adc-channels = <0 1 2 3 4 5 6 7>;
                                };
                        };

                        test_helper: helper {
                                compatible = "bone-iio-helper";
                                vsense-name  = "AIN0", "AIN1", "AIN2", "AIN3", "AIN4", "AIN5",
"AIN6", "AIN7";
                                vsense-scale = <100  100  100  100  100  100  100  100>;
                        status = "okay";
                        };
                };
        };
[…]
```

5.6.4.3 Using the SPI

The Serial Peripheral Interface (SPI) is a synchronous communication standard which is frequently used in devices or peripherals of embedded systems. It is based on a bus which handles bi-directional communication in a master-slave model. The system is composed of one master and one or more slaves. The figure depicts a simple master-slave scenario:

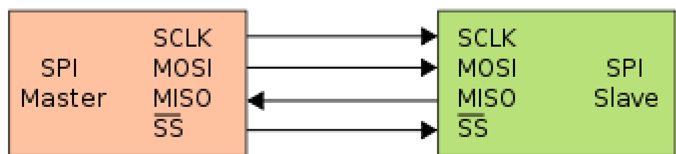


Fig 5.13: SPI master & slave communication scheme

The logical signals in the SPI interface are the following:

- SCLK: Serial Clock
- MOSI: Master Output, Slave Input
- MISO: Master Input, Slave Output
- SS: Slave Select (for multiple slaves in collaborative or non-collaborative mode)

Basically, the operation mode is as follows: the master device sends a bit on the MOSI and the slave reads it, whereas the slave sends a bit on the MISO and the master reads it. This bidirectional communication occurs each SPI clock cycle, even if only one direction data transfer is required. The communication can be seen as if both master and slave keep two registers which actually combined act as a virtual shift register [5.11].

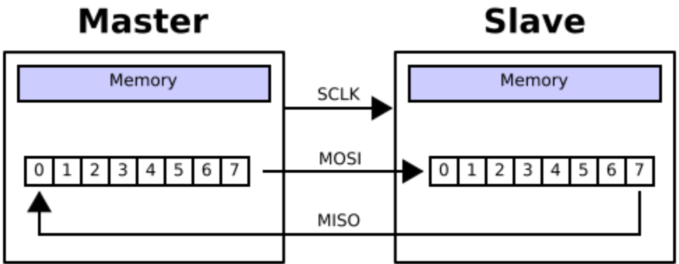


Fig 5.14: Virtual shift register in SPI master & slave model

One advantage of using Adafruit Python modules to handle SPI is that the developer does not need to know almost anything about this communication protocol. He/She only has to know which bus and devices the peripheral is attached to, and perform write/read operations.

Some of the commands executed to handle a master-slave communication before loading the required SPI library are the following:

- SPI: initiates a handler to the required bus and device
- writebytes: indicates a list of bytes to be sequentially written to the MOSI

The following code toggles on and off each second the LEDs on a 4x4 LED matrix. It uses the SPI protocol handling a slave shift register, which finally enables or disables the LEDs on the matrix.

```
#--------------------------------------------------------------------------------
# Code of application LEDMATRIX.py

import Adafruit_BBIO.GPIO as GPIO
from Adafruit_BBIO.SPI import SPI
import time, thread

GPIO.setup("P9_41", GPIO.IN)
spi01 = SPI(1,0) #/dev/spidev1.0
pushed=0

def func_matrix ():
	output=1
	global pushed
	while (pushed==0):
		if output:
			print "Switch ON LEDMATRIX"
			spi01.writebytes([0x00])
			output=0
		else:
			print "Switch OFF LEDMATRIX"
			spi01.writebytes([0x0F])
			output=1
		time.sleep(1)

def func_push (pin):
	global pushed
	pushed=1

try:
	pushbutton="P9_41"
GPIO.add_event_detect(pushbutton,GPIO.RISING,callback=func_push,bouncetime=20)
	id=thread.start_new_thread(func_matrix,())
	while (pushed==0):
		time.sleep(0.1)
```

```
except:
        GPIO.remove_event_detect(pushbutton)
        GPIO.cleanup()
        raise

GPIO.remove_event_detect(pushbutton)
GPIO.cleanup()
#--------------------------------------------------------------------------------
```

Note the following:

- The values 0x00 and 0x0F are written alternatively in the MOSI bus switching on and off the LED matrix respectively
- start_new_thread: The resource of threading has been included. Threads can be used after importing the required module. This function needs a callback function to be passed as an argument, which will be the executed routine in the thread
- Improvements such as using functions, try-except and events have been used in order to generate a better code and application

Although not necessary at all in this application, threads are often useful when dealing with more than one task at the same time. In some cases, some extra signaling resources need to be used in order to avoid concurrency (semaphores, mutexes, condition variables, etc.)

5.6.4.4 Using the I2C

The Inter-Integrated Circuit (IIC or I2C) is a serial half-duplex communication bus composed of two lines, which allows more than one master and slaves under the same bus:

- SDA: Serial Data Line
- SCL: Serial Clock Line

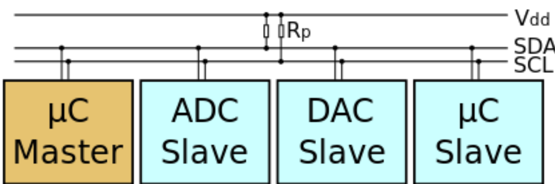


Fig 5.15: interconnection schema in an I2C system

Each device has a 7 bit direction, and the communication is done using 9 bit frames: 8 bits of data and 1 of acknowledgement. Basically, the protocol is the following:

- The data transfer is initiated with a start bit signaled by SDA being pulled low while SCL stays high
- SDA sets the first data bit level while keeping SCL low
- The data is sampled (received) when SCL rises for the first bit
- This process repeats, SDA transitioning while SCL is low, and the data being read while SCL is high
- A stop bit is signaled when SDA is pulled high while SCL is high [5.12]

In a Linux environment such as Ubuntu in the BeagleBone, there exists a tool which provides some information about one I2C bus in the system. This tool is called *i2cdetect*. It scans the selected I2C bus searching for attached devices.

Different I2C buses can be scanned in the BeagleBone:

```
root@arm:~/python# i2cdetect -y -r 0
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- UU -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: UU -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

The program that is invoked above is using two options: the option -y disables the interactive monitoring mode, whereas option -r sends a specific byte at this address in order to probe if there is a device attached to that. The printed results can be the following:

- "--" Address probed but no chip answer received
- "UU" Address not probed because it is currently used by a driver (chip attached to that)
- A specific address. A chip is attached to that address.

The I2C Adafruit's Python module provides a simple interface to handle devices attached to an I2C bus at a given address. Before importing this module, some operations can be done:

- Adafruit_I2C: initiates the handler of the attached device indicating its address and bus
- readX: Adafruit provides a complete Python API in order to perform read operations with different types of data and size: readList, readS16, readU8,...
- readX: Adafruit also provides a complete Python API depicted to perform write operations with different types of data and size: writeList, write8,...

More information about Adafruit I2C API can be seen in the official Adafruit web page [5.13]

In order to test how this library works, the cape provides a 3-axis accelerometer which is handled through an I2C bus. The model is MMA8453Q and is made by Rohs. The device is attached at the address 0x1D of the I2C bus number 2, which can be proven with *i2cdetect* command:

```
root@arm:~/python# i2cdetect -y -r 2
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- 1d -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- 53 -- -- -- 57 -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

The technical reference manual of MMA8453Q accelerometer depicts which registers must be written to set up the device and which ones provide the values for the acceleration. Synthesizing, these are the required values:

- Control register 1: address 0x2A (8 bits). Needs to set up the active mask to 0x01
- X axis acceleration, most significant byte: address 0x01 (8 bits)
- X axis acceleration, least significant byte: address 0x02 (8 bits)
- Y axis acceleration, most significant byte: address 0x03 (8 bits)
- Y axis acceleration, least significant byte: address 0x04 (8 bits)
- Z axis acceleration, most significant byte: address 0x05 (8 bits)
- Z axis acceleration, least significant byte: address 0x06 (8 bits)

The following is a Python code which uses the features provided by Adafruit's I2C library. The application reads the value of the three axis of acceleration and represents it on the terminal console refreshing this information every two seconds:

```
#--------------------------------------------------------------------------------
# Code of application ACCELEROMETER.py

from Adafruit_I2C import Adafruit_I2C
import time

#MMA8453QR1 device address
MMA8453QR1_address = 0x1D

#MMA8453QR1 device configuration
REG_CTL_REG1 = 0x2A
activeMask =0x01

#MMA8453QR1 X, Y and Z acceleration registers
ACC_X_MSB=0x01
ACC_X_LSB=0x02
ACC_Y_MSB=0x03
ACC_Y_LSB=0x04
ACC_Z_MSB=0x05
ACC_Z_LSB=0x06

#Get handler to the device address at i2c bus 2
i2c = Adafruit_I2C(MMA8453QR1_address,2,False)

#Setup accelerometer
i2c.write8(REG_CTL_REG1,activeMask)

while True:

       acc_X=(i2c.readS8(ACC_X_MSB) << 2)|((i2c.readS8(ACC_X_LSB) >> 6)& 3)
       acc_Y=(i2c.readS8(ACC_Y_MSB) << 2)|((i2c.readS8(ACC_Y_LSB) >> 6)& 3)
       acc_Z=(i2c.readS8(ACC_Z_MSB) << 2)|((i2c.readS8(ACC_Z_LSB) >> 6)& 3)

       if (acc_X > 511):
              acc_X = acc_X - 1024
       if (acc_Y > 511):
              acc_Y = acc_Y - 1024
       if (acc_Z > 511):
              acc_Z = acc_Z - 1024
```

```
        print "ACC X: ", acc_X
        print "ACC Y: ", acc_Y
        print "ACC Z: ", acc_Z
        print ""

        time.sleep(2)
#-----------------------------------------------------------------------------------
```

Note the following:

- Due to the fact that the accelerometer has 10 bits of resolution at each axis, two registers of 8 bits need to be read to obtain the complete value for the acceleration by axis. Values read at each pair of need to be manipulated In order to get a readable value:
    - acc_X=(i2c.readS8(ACC_X_MSB) << 2)|((i2c.readS8(ACC_X_LSB) >> 6)& 3)
    - T corresponds to X, Y or Z axis
- The acceleration is given in a range of values between 0 and 1023. Values between 0 and 511 correspond to positive acceleration. Values between 512 and 1023 correspond to negative acceleration. To have a clearer representation of this value a simple transformation is done:
    - if (acc_T > 511):
    - acc_T = acc_T - 1024
    - T corresponds to X, Y or Z axis

The code does not include extra features such as threads, exceptions or event detection in order to ease readability (is just an academic exercise).

5.6.5 Introducing graphics in the BeagleBone

At this point it would be interesting to see if the current environment that has been created for the BeagleBone is proper to develop some graphical features. The tuple formed by the BeagleBone plus the cape (due to its peripherals) results in a combination providing information that can be represented graphically. This information is frequently much clearer than raw data at the moment of interfacing with humans. For instance, if we want to know how strong the acceleration on one axis is, we will better figure it out it if we see a figure that bends to one side or another depending on the strength of this value.

The current environment in the BeagleBone is proper to develop some kind of graphical applications using the peripherals in the cape. In order to do so, it is necessary to install some additional features in the operating system before using graphical features.

It is important to see that the BeagleBone lacks a specific hardware to connect the board to any kind of monitor. The board does not have resources to reproduce graphical environments such as a typical desktop, so a solution based in virtualization can be provided.

5.6.5.1 Description of the solution. Tkinter & VNC

The basic idea is the following: the image that is generated by a graphical application in the BeagleBone needs to be stored somewhere in the system memory. This data can be retrieved and served in order that other application present in a remote system can get it and represent it using a monitor.

At present, different Linux based operating systems provide a set of tools in order to do so. These tools are also present in the Ubuntu distribution that is used for this project:

- Virtual Frame Buffer (VFB): this is software that permits creating a virtual monitor in the memory of the system. This tool can be used in systems with graphical card resources as well as in those without
- VNC server: this software is present in some Linux distributions. It enables a video server which can be fed by the image data in the console 7 (tty7) or by other virtualized monitors such as those created by VFB
- VNC client: this is a program which is able to establish a connection to a VNC server through a specific port. It is able to reproduce the image that the server provides

The following is the interconnection schema between applications:

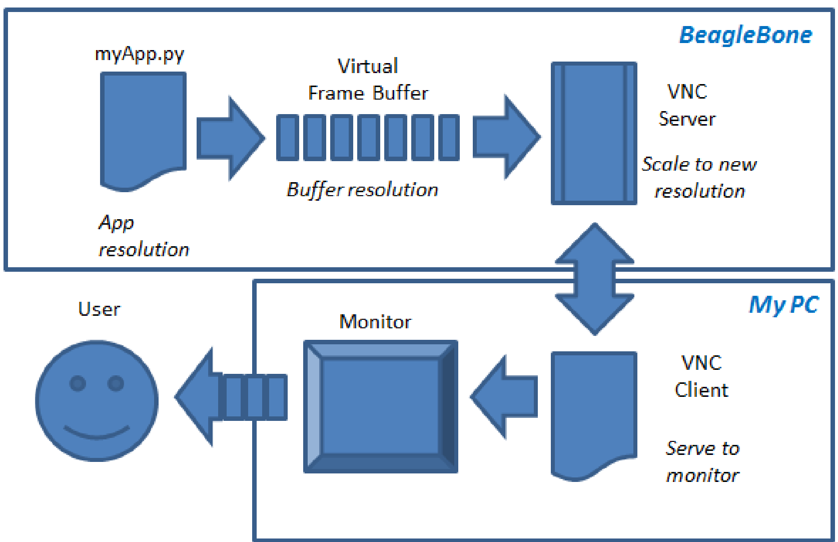


Fig 5.16: VNC Interconnection scheme

Note that both VFB and VNC server need to be installed into the BeagleBone, whereas the VNC client needs to be present in the host PC which is connected to the BeagleBone. There exists a variety of applications which can provide a VNC client, including Moba XTerm. In addition, the application providing graphical features in the BeagleBone needs to use a specific library in order to do so. This is a Python library which is called Tkinter. Some features of that resource will be explained in the following sections.

5.6.5.2 Installing graphical resources in the BeagleBone

Again, it is necessary to connect the BeagleBone to internet in order to download and install the required software. The steps to set up the environment properly are described in the following lines.

First of all, the repositories need to be updated. Doing this is a proper action to get the latest security fixes in the software that is going to be retrieved from the repositories:

```
# apt-get update
```

After that, it is necessary to get the VFB software and install it as well as its dependencies. Note that it is possible to check the package that is going to be installed using the tool apt-cache. This tool is also proper to check if the software has been installed before:

```
# apt-cache search xvfb
xvfb - Virtual Framebuffer 'fake' X server

# apt-cache pkgnames | grep xvfb
[prints nothing]

# apt-get install xvfb
[proceeds with the installation of the package and its dependencies]
```

After installing the VFB, it is necessary to install the VNC server. It is possible to proceed using apt-cache just as shown before:

```
# apt-cache search x11vnc
ssvnc - Enhanced TightVNC viewer with SSL/SSH tunnel helper
x11vnc - VNC server to allow remote access to an existing X session
x11vnc-data - data files for x11vnc

# apt-cache pkgnames | grep x11vnc
[prints nothing]

# apt-get install x11vnc
[proceeds with the installation of the package and its dependencies]
```

At this point the system has the required tools to perform some tests in order to see if the system behaves as expected. Some typical X application (using graphical capabilities) such as Xeyes or Xcalc can be downloaded to the BeagleBone and proved:

```
# apt-get install x11-apps
```

5.6.5.3 Testing graphical resources in the BeagleBone

One option to compose a graphical environment in the BeagleBone can be given by the following steps:

- First: set up a virtual frame buffer. The following command will do what is expected:

```
# Xvfb :20 -dpi 75 -screen 0 800x600x16 -nolisten tcp &
```

  - :20 - a server will hear for connections as server number 20
  - -dpi 75 - depicts the number of dots per inch of the virtual screen
  - -screen 0 800x600x16 - is the setup of the virtual screen. Width, length and bit depth can be set with this option
  - -nolisten tcp - is an additional option which disables listening for possible annoying client TCP connections

    - & - execute in background

- Second: set up a VNC server. The following command will do what is expected:

```
# x11vnc -display :20 -scale 800x600 -rfbport 4000 -forever &
```

    - -display :20 - depicts which display the VNC server is binded to
    - -scale 800x600 - resizes the image to the desired resolution or scale (can be a scalar number), the width and height that is going to be served
    - -rfbport 4000 - is the port to listen to
    - -forever - keeps listening to client connections instead of disconnecting once the client has disconnected
    - & - execute in background

- Third: export the new value for the DISPLAY variable:

```
# export DISPLAY=:20
```

    - This command exports a new value for the DISPLAY variable. It is used to tell X applications which screen they are going to be run on.

After setting the environment, a VNC client running in the host PC needs to be run. The application that is being used to connect through SSH to the BeagleBone (Moba XTerm) also provides VNC client features. It is very simple so establish a VNC connection to the BeagleBone in the environment that has been prepared before:

1) Start a new session in Moba XTerm
2) Indicate that it is a VNC connection
3) Write the name of the remote address or the IP address of the BeagleBone: 192.168.2.100
4) Write the port number of the remote VNC server to be connected to: 4000

A new tab is opened in the Moba XTerm application showing the contents of the virtual screen:

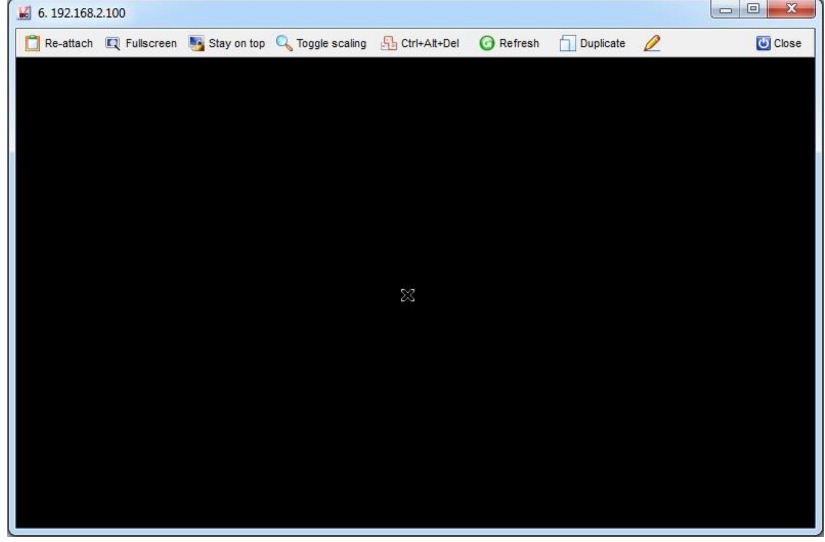


Fig 5.17: New VNC viewer tab in Moba Xterm

Note that there is nothing to be shown in this tab yet. In order to see some application running, it is possible to execute them in the other BeagleBone SSH console tab. In the following figure, it can be seen typical X applications running:

```
# xeyes & xcalc & xclock &
```

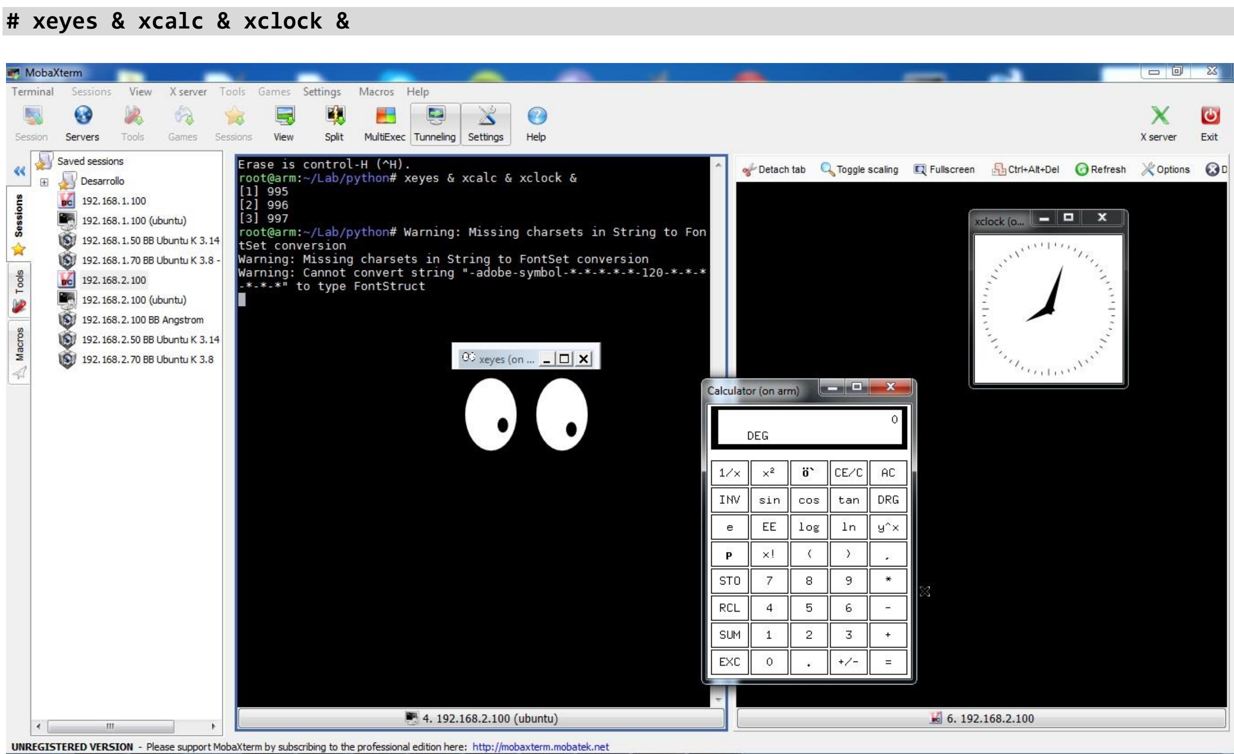


Fig 5.18: Common X applications thrown in BeagleBone viewed using VNC viewer

These applications run detached from the terminal they are shown by default. It results in a visualization that seems to be program executed locally, but they are not. Actually they are executed in the BeagleBone but visualized in the host PC.

5.6.5.4. Using Python for graphical applications: Tkinter

Tkinter is the most common free library used by Python to implement graphical applications. It is built as a binding to the standard Tk GUI toolkit, and it is included as de-facto standard in basic Python installations of Windows and Mac OS X. Its name comes from the contraction of Tk-interface [5.14].

The basic structure of a code written in Tkinter is the following:

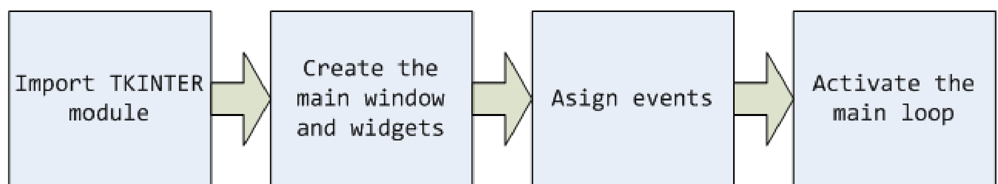


Fig 5.19: Basic structure of Tkinter application

There is a set of basic operations that are useful to know to handle graphics in Python after importing the library module. Some of them are summarized in the following lines.

Setting the main window of the application and some attributes:

- myWindow = Tkinter.Tk(): constructs the main window of the application
- myWindow.title('title'): sets the name of the window
- myWindow.geometry('400x300'): sets the size of the window (dots)
- myWindow.after(delay, function): the main window executes the depicted function after the given delay

Running the main loop of the window:

- myWindow.mainloop()

Creating a widget label and setting some of its attributes:

- myLabel = Tkinter.Label(myWindow): constructs a widget label into the depicted main window
- myLabel.configure(text="my label"): sets the text of the label
- myLabel.place(x=20, y=60): sets the position of the label into the main frame
- myLabel.update(): updates the text in the label

Creating a widget canvas and setting some of its attributes. Coordinates in the canvas bitmap are given following this direction:

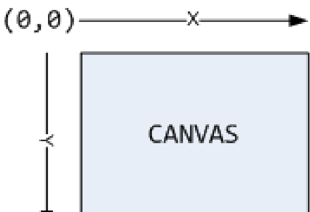


Fig 5.20: Position and direction of the XY axis in a Tkinter canvas

- myCanvas = Tkinter.Canvas(mainWindow): constructs a canvas into the depicted main window
- myCanvas.configure(width=10, height=20): sets width and height attributes
- myCanvas.create_line(x0, y0, x1, y1, width="6", fill="orange2): creates a line in the canvas which starts at (x0,y0), ends at (x1,y1) with the given width and colour
- myCanvas.create_polygon(x0, y0, x1, y1, x2, y2, … , fill="orange"): creates a polygon with vertexes at each pair (xi, yi) with the given colour
- myCanvas.coords(object, x0, y0, x1, y1, x2, y2, …): sets the new coordinates of the given object [5.15].

With the set of functions depicted above, it is possible to write a graphical Python application allowing a better visualization of some of the peripherals that are present in the cape. In order to show how it works, an application based in the accelerometer has been implemented as example.

5.6.5.5. Example program: Taking advantage of Tkinter with the accelerometer

The idea is to use the Tkinter module in order to show how the accelerometer is bent to one side or another. In this case, only the acceleration in one of its axis will be taken into consideration. The objective is to represent a bar bending to both sides in function of the values that are read from the X axis of the accelerometer.

An extra module will be imported in order to aid using some mathematical functions. This module is called math and it will provide capabilities to calculate sinus and cosine.

- math.cos(rad_X)
- math.sin(rad_X)

The following is the code of the application ACCELEROMETER_TK.py. There are no new elements introduced except those referring to math and Tkinter modules.

```
#--------------------------------------------------------------------------------
# Code of application ACCELEROMETER_TK.py

from Adafruit_I2C import Adafruit_I2C
import time, thread, Tkinter, math

# Define MMA8453QR1 accelerometer parameters
MMA8453QR1_address    = 0x1D               #MMA8453QR1 device address
REG_CTL_REG1          = 0x2A               #MMA8453QR1 register configuration
activeMask            = 0x01               #MMA8453QR1 value active mask config
ACC_X_MSB             = 0x01               #MMA8453QR1 X MSB acceleration register
ACC_X_LSB             = 0x02               #MMA8453QR1 X LSB acceleration register

# Set up MMA8453QR1 accelerometer
i2c = Adafruit_I2C(MMA8453QR1_address,2,False)    #Device address at i2c bus 2
i2c.write8(REG_CTL_REG1,activeMask)               #Setup accelerometer

# Set tkinter main window
rootWin = Tkinter.Tk()                     #Create the main window
rootWin.title('MMA8453QR1 data')    #Name of the window
rootWin.geometry('400x300')         #Define the geometry/resolution
```

```
# Define some parameters of the canvas
axis_x = 150                                #Supporting point at x axis
axis_y = 150                                #Supporting point at y axis
length_bar = 100                            #Length of each ahlf of the bar

# Set a canvas to write some objects on
myCanvas = Tkinter.Canvas(rootWin, width=300, height=300)
myCanvas.pack()

# Write some objects on the canvas: moving bar and static triangle
bar = myCanvas.create_line(axis_x + length_bar, axis_y, axis_x - length_bar, axis_y,
width="6", fill="green")
myCanvas.create_polygon(axis_x, axis_y, axis_x + 10, axis_y + 20, axis_x - 10, axis_y + 20,
fill="orange")

# Set a text label to show the inclination
printVal = Tkinter.StringVar()                                  #Set a string variable
printLabel = Tkinter.Label(rootWin, textvar=printVal)    #Asign it to a label
printLabel.place(x=200,y=250)                                   #Set label position
#printLabel.pack()

def func_accelerometer():

       #Execute the same function after "delay" milliseconds
       rootWin.after(20, func_accelerometer)

       #Get the acceleration from the accelerometer X MSB and LSB registers
       try:
              acc_X=(i2c.readS8(ACC_X_MSB) << 2)|((i2c.readS8(ACC_X_LSB) >> 6)& 3)

       except IOError, err:
              print err
              raise

       if (acc_X > 511):                                 #Transform to readable values
              acc_X = acc_X - 1024

       deg_X = acc_X*90/256                       #Convert to degrees
       printVal.set(str(deg_X))                          #Set text value for the label

       rad_X = math.radians(deg_X)                #Convert to radians
       inc_x = int(length_bar * math.cos(rad_X))  #Set the X extreme of the bar
       inc_y = int(length_bar * math.sin(rad_X))  #Set the Y extreme of the bar

       #Set the new position of the bar
       myCanvas.coords(bar, axis_x+inc_x, axis_y+inc_y, axis_x-inc_x, axis_y-inc_y)

try:
       func_accelerometer()
       rootWin.mainloop()

except IOError, err:
       print err
       raise
#---------------------------------------------------------------------------------
```

The previous code is executed in a virtual frame buffer in the BeagleBone and visualized through a VNC client. The following image can be seen if the accelerometer is settled at a neutral X axis position:

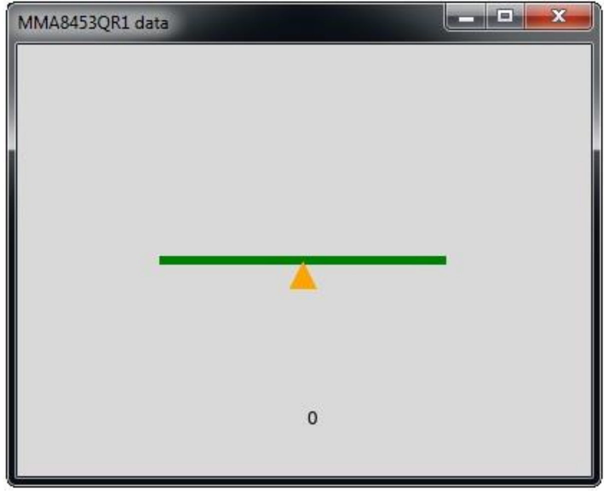


Fig 5.21: Accelerometer with the cape at a neutral X position

When the BeagleBone with the cape is bent to some side in the X direction, the bar in the graphical application also bends. The pictures below show how it is represented. The window also shows a scalar value with the degrees the board has bent.

Fig 5.22: Accelerometer bent in the X direction and opposite to X direction, respectively

Through this kind of application one can easily figure out the position of some device which has an embedded accelerometer. For instance, it could be useful as reference to easily know the relative position of a robotic arm.

6. Results

6.1. Ubuntu system

Due to the fact that a full system test is not easy to be done, the evaluation of the new operating system can only be qualitative. Instead, some comments can be done about the overall performance that has been seen during the processes that have involved the current project.

In first place, it is necessary to highlight that the system behaves exactly as required. Having a well-known system that is also widely supported by the Linux community (in this case, Ubuntu community) eases handling the embedded device which is the BeagleBone board. The currently deprecated Angstrom system had a variety of particularities that made some common tasks such as handling a network interface much more complex than in Ubuntu.

Downloading and updating programs and part of the software in the system in Ubuntu is almost automatic. The Ubuntu repository system is one of the most completes, making it very easy to get the proper tool for a particular task. In other words: you want a tool, you download and install this tool and all its dependencies automatically. It has been seen previously in this document when it was needed to install different tools such as *nodejs*, *Vim*, or *aptitude*, even *Python*.

Network interfaces configurations can be easily built through editing typical configuration files such as /etc/network/interfaces or /etc/resolv.conf. Network configuration tools such as *iptables* and *route* are also available. Their behavior is the same as in other not embedded systems.

6.1.1 Getting into Embedded Ubuntu

In order to access the BeagleBone, it can be done in two ways: one is using the serial port connection and the other one (which is more useful) is by using the Ethernet connector. The Ubuntu OS is equipped with a serial port service allowing serial connections:

```
# screen /dev/ttyUSB0 115200
```

In a similar way, a SSH server is also available:

```
# ssh ubuntu@192.168.2.70 [passwd: temppwd]

Welcome to Ubuntu 14.04.1 LTS (GNU/Linux 3.8.13-bone70 armv7l)

* Documentation:  https://help.ubuntu.com/
Last login: Tue Jan 20 19:40:23 2015 from 192.168.2.11
ubuntu@arm:~$
```

The serial connection has only been used in this work to demonstrate that it works. Instead, all the connections to the BeagleBone have been done with SSH.

Similarly, the network file system NFS has not been used. It has been implemented and tested only for academic purposes. Instead, and due to the fact that the system is powerful enough, file transfers have been performed between the host PC and the micro SD card in the BeagleBone. The tool used is *scp*, which can be easily called from the shell.

In order to perform privileged actions in the system, a privileged user must be used. The default user *ubuntu* can raise privileges easily:

```
ubuntu@arm:~$ sudo -s [passwd: ubuntu]
ubuntu@arm:~#
```

Note that the password was changed from the original that came with the reduced Ubuntu file system for the BeagleBone.

Nevertheless, this user has full access to a full set of options in the system using its privileges. It can be deduced by reviewing the groups it is assigned to:

```
root@arm:~# id Ubuntu
uid=1000(ubuntu) gid=1000(ubuntu)
groups=1000(ubuntu),4(adm),15(kmem),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),44(vid
eo),46(plugdev),100(users),102(netdev),106(i2c),999(admin),998(spi),997(systemd-
journal),996(weston-launch),995(xenomai)
```

Amongst them, there is the group “admin”, which has full privileges in the system. It can be checked in the file system /etc/sudoers:

```
[…]
# Members of the admin group may gain root privileges
%admin ALL=(ALL) ALL
[…]
```

6.1.2 Startup process

The first lines of the /var/log/dmesg log file shows details from the Linux kernel version and compilation, as well as from the system processor. A quick overview of this file reveals that there has not been any problem in the startup process.

```
# dmesg
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Linux version 3.8.13-bone70 (root@ubuntu) (gcc version 4.7.3 20130328
(prerelease) (crosstool-NG linaro-1.13.1-4.7-2013.04-20130415 - Linaro GCC 2013.04) ) #1 SMP
Wed Feb 18 16:01:36 PST 2015
[    0.000000] CPU: ARMv7 Processor [413fc082] revision 2 (ARMv7), cr=50c5387d
[…]
```

In particular, the kernel compilation version can also be seen with *uname*:

```
root@arm:~# uname -a
Linux arm 3.8.13-bone70 #1 SMP Wed Feb 18 16:01:36 PST 2015 armv7l armv7l armv7l GNU/Linux
```

Details from the processor can be determined with *lscpu*:

```
root@arm:~# lscpu
Architecture:          armv7l
Byte Order:            Little Endian
CPU(s):                1
On-line CPU(s) list:   0
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):             1
```

It is important to highlight that no problems have been detected in the startup of the system. The hardware has been detected correctly and the modules have been loaded properly.

6.2. GPMC

The prototype modules of the asynchronous, synchronous and burst synchronous driver have been copied to the BeagleBone, as well as a simple application that uses it doing both write and read operations. The test application has been called *testdriver*, and essentially consists on a loop which alternatively offers the possibility of reading data from the GPMC or writing data to it.

The current section will test the capability of writing data in the GPMC. For this option, an attached external memory is not required, only a logical analyzer such as the laboratory one. The training sequence of data that will be written on the GPMC will be deterministic and always the same:

```
DATA = [0x0000, 0x0101, 0x0202, 0x0303, 0x0404, 0x0505, 0x0606, 0x0707]
```

That is a sequence of 8 words of 2 bytes each. Due to the fact that the laboratory analyzer has a limited set of lines to read and visualize, just the less significant byte will be displayed. This byte also coincides with the address bus. Remember that the implemented GPMC multiplexes addresses and data on the same bus.

Before loading the module and executing the application, the pins of the GPMC need to connected to the logical analyzer of the laboratory. The connection is done following the pinout label number connection described in the following schema:

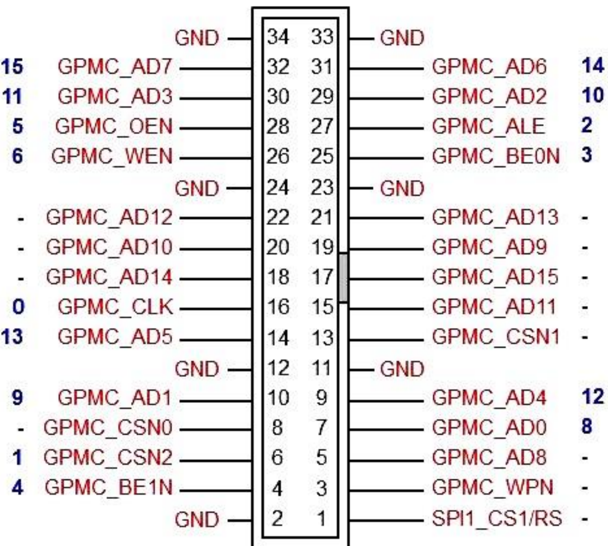


Fig 6.1: TT01 cape board CN-2 connector pinout

The labels in red indicate the GPMC pin functions whereas the numbers in blue correspond to the wires that are connected to the logical analyzer following the depicted order.

Once connected the chosen signals in the GPMC interface to the logical analyzer, a set of tests is performed in order to check the behavior of this interface.

6.2.1 Testing GPMC asynchronous module

The asynchronous version of the module is loaded in the BeagleBone. After that, the program *testdriver* is executed, and the depicted sequence of information is written in the bus when it asks to. The following figure provides the output in the display of the logical analyzer when performing a write operation using the asynchronous module:

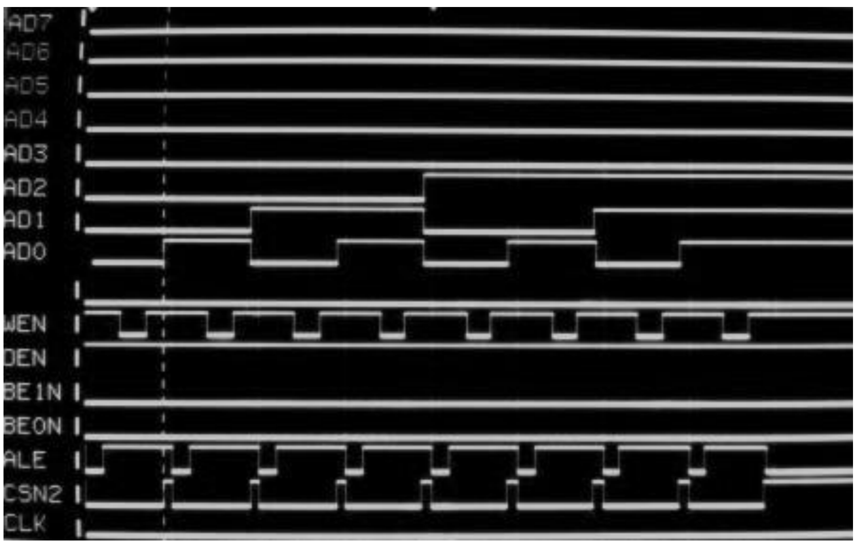


Fig 6.2: GPMC asynchronous write chronogram

The trigger has been set to be the falling edge in the ADV signal. The figure shows how the same pattern is repeated 8 times for the control signals. It corresponds to eight accesses to the data bus putting a different word in it ranging from 0 to 7. The display only shows the less significant byte of this word.

It must be highlighted that in this particular case the addresses that are put in the AD bus are same as the data that is being written in the same bus. This is why it is not appreciated a change in the AD bus after the address is set and when the data needs to be set.

The control signals Chip Select, Address Valid and Write Enable and Output Enable behave as expected. For this type of access the signals Byte Enables are not required nor set. Recall the relevant configuration parameters in the asynchronous GPMC:

<table>
<tr><td colspan="5">GPMC_CONFIG1
READTYPE = GPMC_CONFIG1_0_READTYPE_RDASYNC
WRITETYPE = GPMC_CONFIG1_0_WRITETYPE_WRASYNC</td></tr>
<tr><td>GPMC_CONFIG2
CSONTIME=1
CSRDOFFTIME=10
CSWROFFTIME=10</td><td>GPMC_CONFIG3
ADVONTIME=1
ADVRDOFFTIME=3
ADVWROFFTIME=3</td><td>GPMC_CONFIG4
OEONTIME=5
OEOFFTIME=9
WEONTIME=5
WEOFFTIME=8</td><td>GPMC_CONFIG5
RDCYCLETIME=10
WRCYCLETIME=10
RDACCESSTIME=8</td><td>GPMC_CONFIG6
WRDATAONADMUXBUS=4
WRACCESSTIME = 0</td></tr>
</table>

Table 6.1: Relevant configuration parameters in the asynchronous GPMC

6.2.2 Testing GPMC synchronous module

The synchronous version of the module is loaded in the BeagleBone. After that, the program *testdriver* is executed, and the depicted sequence of information is written in the bus when it asks to. The following

figure provides the output in the display of the logical analyzer when performing a write operation using the synchronous module:

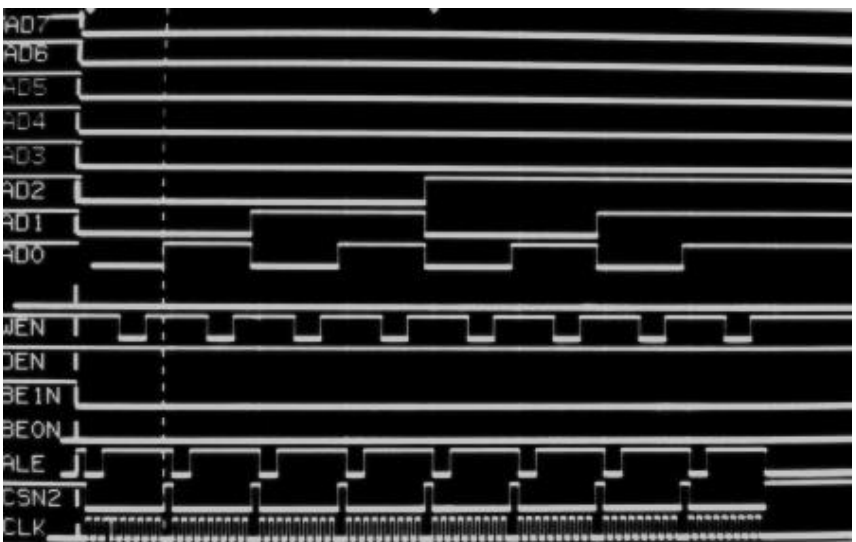


Fig 6.3: GPMC synchronous write chronogram

In this case, the signals behave again in the same way as for the asynchronous driver. The only difference between asynchronous and synchronous chronograms is the CLK signal. In the synchronous chronogram, this signal is enabled one clock cycle after each write cycle begins, as expected.

The control signals Chip Select, Address Valid and Write Enable and Output Enable behave as expected. For this type of access the signals. The external clock signal is also enabled when it is expected. Byte Enables are not required nor set. Recall the relevant configuration parameters in the synchronous GPMC:

<table>
<tr><td colspan="5">GPMC_CONFIG1
READTYPE = GPMC_CONFIG1_0_READTYPE_RDSYNC
WRITETYPE = GPMC_CONFIG1_0_WRITETYPE_WRSYNC
CLKACTIVATIONTIME = GPMC_CONFIG1_0_CLKACTIVATIONTIME_ONECLKB4</td></tr>
<tr><td>GPMC_CONFIG2
CSONTIME=1
CSRDOFFTIME=10
CSWROFFTIME=10</td><td>GPMC_CONFIG3
ADVONTIME=1
ADVRDOFFTIME=3
ADVWROFFTIME=3</td><td>GPMC_CONFIG4
OEONTIME=5
OEOFFTIME=9
WEONTIME=5
WEOFFTIME=8</td><td>GPMC_CONFIG5
RDCYCLETIME=10
WRCYCLETIME=10
RDACCESSTIME=8</td><td>GPMC_CONFIG6
WRDATAONADMUXBUS=4
WRACCESSTIME = 0</td></tr>
</table>

Table 6.2: Relevant configuration parameters in the synchronous GPMC

6.2.3 Testing GPMC synchronous burst module

The synchronous burst version of the module is loaded in the BeagleBone. After that, the program *testdriver* is executed. Now, in order to get a clearer view the written sequence it has been changed by the following:

```
DATA = [0x00AA, 0x0055, 0x00AA, 0x0055, 0x00AA, 0x0055, 0x00AA, 0x0055]
```

Only the LSB of the sequence is changed due to the fact that the MSB is not displayed in the analyzer. The sequence can easily be recognized because it represents 10101010 and 01010101 alternatively in binary.

The sequence above is written in the bus when it asks to. The following figure provides the output in the display of the logical analyzer when performing a write operation using the asynchronous module:

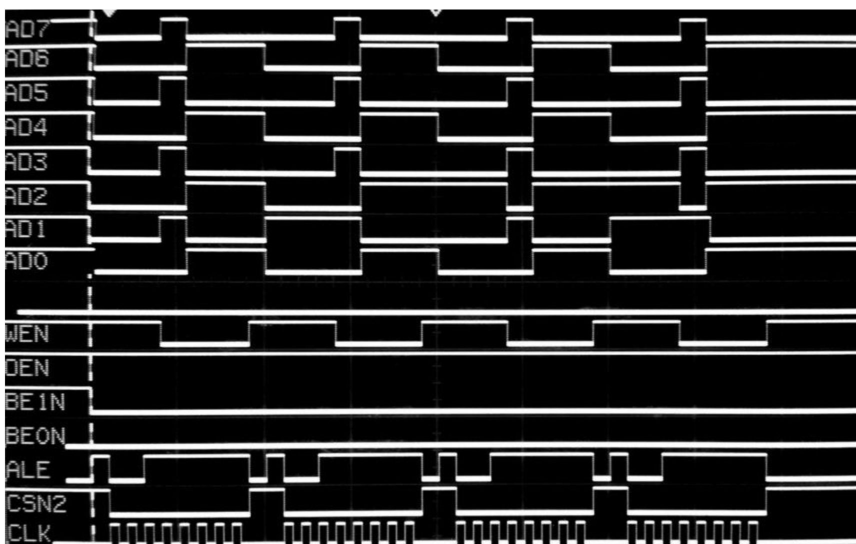
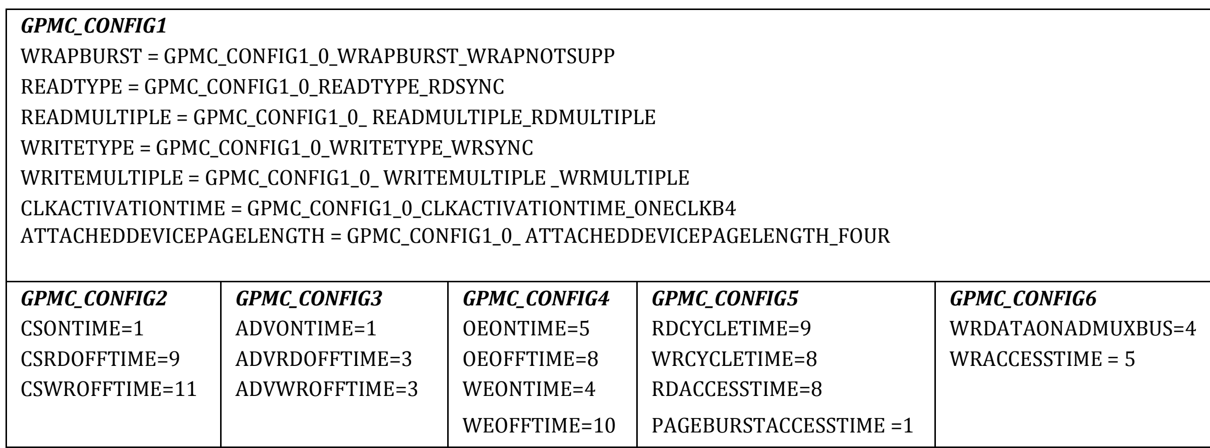


Fig 6.4: GPMC synchronous burst write chronogram

In this case, the behavior of the driver is not exactly the expected. The design was made in order to perform four write operations at each burst. So, for a write operation of 8 words, two bursts are needed. Now the chronogram is showing 4 burst operations. Only 2 different words of data are being written in the bus instead the expected 4.

The burst gets freeze in the second value of the burst and remains in this state during the rest of the burst time. Other 2 transactions should happen within this period (until write enable is deasserted), but unfortunately this is not happening. This issue should be fixed in future work.

Recall the relevant configuration parameters in the synchronous burst GPMC:



<table>
<tr><td colspan="5">GPMC_CONFIG1
WRAPBURST = GPMC_CONFIG1_0_WRAPBURST_WRAPNOTSUPP
READTYPE = GPMC_CONFIG1_0_READTYPE_RDSYNC
READMULTIPLE = GPMC_CONFIG1_0_ READMULTIPLE_RDMULTIPLE
WRITETYPE = GPMC_CONFIG1_0_WRITETYPE_WRSYNC
WRITEMULTIPLE = GPMC_CONFIG1_0_ WRITEMULTIPLE _WRMULTIPLE
CLKACTIVATIONTIME = GPMC_CONFIG1_0_CLKACTIVATIONTIME_ONECLKB4
ATTACHEDDEVICEPAGELENGTH = GPMC_CONFIG1_0_ ATTACHEDDEVICEPAGELENGTH_FOUR</td></tr>
<tr><td>GPMC_CONFIG2
CSONTIME=1
CSRDOFFTIME=9
CSWROFFTIME=11</td><td>GPMC_CONFIG3
ADVONTIME=1
ADVRDOFFTIME=3
ADVWROFFTIME=3</td><td>GPMC_CONFIG4
OEONTIME=5
OEOFFTIME=8
WEONTIME=4
WEOFFTIME=10</td><td>GPMC_CONFIG5
RDCYCLETIME=9
WRCYCLETIME=8
RDACCESSTIME=8
PAGEBURSTACCESSTIME =1</td><td>GPMC_CONFIG6
WRDATAONADMUXBUS=4
WRACCESSTIME = 5</td></tr>
</table>

Table 6.3: Relevant configuration parameters in the synchronous GPMC

6.3. Cape verification

Using the Angstrom overlay cape-bone-TT01v1-00A0.dtdo without modifications produced almost a perfect result in the new Ubuntu OS. Recall that some information in the Angstrom system was saved:

- /lib/firmware/cape-bone-iio-00A0.dts
- /lib/firmware/cape-bone-iio-00A0.dtbo
- /lib/firmware/cape-bone-TT01v1-00A0.dts
- /lib/firmware/cape-bone-TT01v1-00A0.dtbo

The overlays were copied to /lib/firmware in the new system and the device tree sources were saved in the development environment. Before rebuilding and copying the cape applications in the BeagleBone, the results of the execution showed the following:

- GPIO1: worked correctly
- GPIO2: worked correctly
- ENCODER: worked correctly
- ANALOG: did not find the file containing the value of the analogic input AIN0. (It did not exist)
- LEDMATRIX: worked correctly
- ACCELEROMETER: worked correctly

As explained in previous sections, the execution error in ANALOG was produced by a misunderstood in the definition of the channels in the analog fragment of the overlay. Since the proper change was made, the application worked properly. Recall this fix:

```
[…]
        fragment@0 {
                target = <0xdeadbeef>;
                __overlay__ {
[…]
                        tscadc {
[…]
                                adc {
                                        ti,adc-channels = <0x0 0x1 0x2 0x3 0x4 0x5 0x6 0x7>;
                                };
                        };
                };
        }
[…]
```

The following is a summary of the information lines registered in /var/log/dmesg which concern to the cape and cape manager once fixed the problem and with the complementary helper used for the analog section in Python:

```
# dmesg
[…]
[    1.732241] bone-capemgr bone_capemgr.8: Baseboard: 'A335BONE,00A6,1913BB000058'
[    1.740174] bone-capemgr bone_capemgr.8: compatible-baseboard=ti,beaglebone
[    1.778496] bone-capemgr bone_capemgr.8: slot #0: No cape found
```

```
[    1.815597] bone-capemgr bone_capemgr.8: slot #1: No cape found
[    1.852706] bone-capemgr bone_capemgr.8: slot #2: No cape found
[    1.883272] bone-capemgr bone_capemgr.8: slot #3: 'cape-bone-TT01v1,00A0,TuDuTech,cape-
bone-TT01v1'
[    1.893766] bone-capemgr bone_capemgr.8: initialized OK.
[    1.899491] bone-capemgr bone_capemgr.8: loader: before slot-3 cape-bone-TT01v1:00A0 (prio
0)
[    1.908584] bone-capemgr bone_capemgr.8: loader: check slot-3 cape-bone-TT01v1:00A0 (prio
0)
[…]
[    1.935008] bone-capemgr bone_capemgr.8: loader: after slot-3 cape-bone-TT01v1:00A0 (prio
0)
[…]
[    1.950593] bone-capemgr bone_capemgr.8: slot #3: Requesting part number/version based
'cape-bone-TT01v1-00A0.dtbo
[…]
[    1.989057] bone-capemgr bone_capemgr.8: slot #3: Requesting firmware 'cape-bone-TT01v1-
00A0.dtbo' for board-name 'cape-bone-TT01v1', version '00A0'
[…]
[    7.014983] bone-capemgr bone_capemgr.8: slot #3: dtbo 'cape-bone-TT01v1-00A0.dtbo'
loaded; converting to live tree
[    7.015525] bone-capemgr bone_capemgr.8: slot #3: #5 overlays
…
[    7.124935] bone-iio-helper helper.11: ready
[    7.125915] bone-capemgr bone_capemgr.8: slot #3: Applied #5 overlays.
[    7.125941] bone-capemgr bone_capemgr.8: loader: done slot-3 cape-bone-TT01v1:00A0 (prio
0)
```

Here it can be seen how the system device tree is loaded using the cape manager. It can be easily detected the process of correct loading the cape-bone-TT01v1-00A0.dtbo overlay into the system.

6.3.1 Applications verification

The current section illustrates the correct behavior of each application using the cape. The objective is to check if each application does wat is depicted to do.

6.3.1.1 Application GPIO1

This application just switches on the LED LD1 in the cape for 3 seconds. After that, the application finishes. The picture below shows an image of the cape with this LED switched on:

Fig 6.5: LED LD1 in cape TT01v1 switched on, corresponding to app GPIO1

6.3.1.2 Application GPIO2

This application switches on and off the LEDs LD1, LD2 and LD3 each second until the push button is pressed. After that, the application finishes. The picture below shows an image of the cape with these LEDs switched on:

Fig 6.6: LEDs LD1, LD2 and LD3 in cape TT01v1 switched on, corresponding to app GPIO2

6.3.1.3 Application ENCODER

The rotary encoder is based in both falling and rising edge detection of two signals which are simulated through two associated input pins. These pins, P8_7 and P8_9, are connected to a rotating switch generating two different signals, depending on the rotation direction, as described before in Python sections.

Remember that in order to test this application the 3 switches of the cape that are named GMPC EN (switch 2) must be set to ON because they are shared with the GPMC.

The following is the console output when rotating the encoder clockwise:

```
ready: 1
         Lead B
                 A: 1  B: 0
ready: 1
         Lead A
                 A: 0  B: 0
ready: 1
         Lead B
                 A: 0  B: 1
ready: 1
         Lead A
                 A: 1  B: 1
```

And this is the console output when rotating the encoder counter-clockwise:

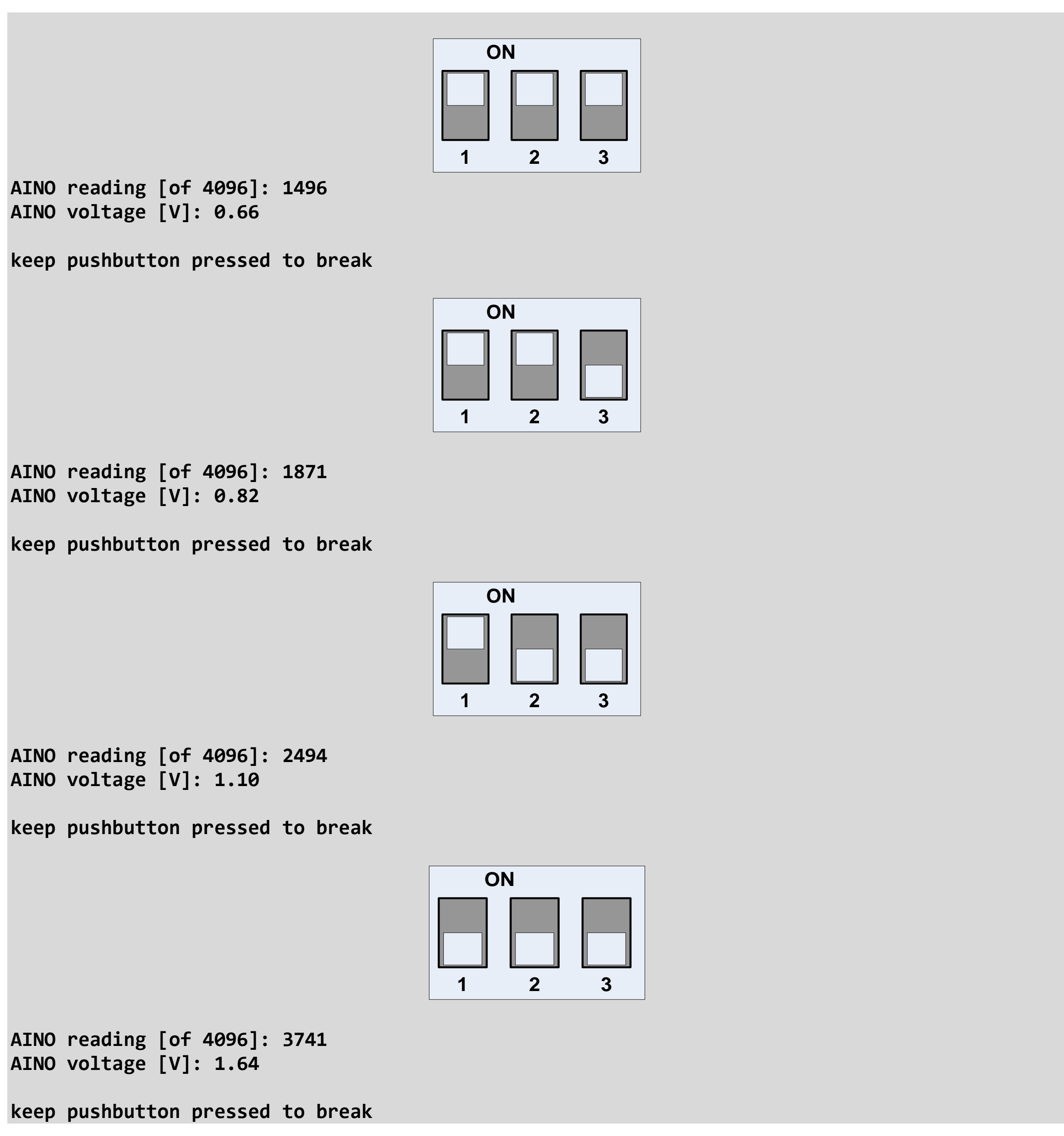
```
ready: 1
         Lead A
                 A: 0  B: 1
ready: 1
         Lead B
                 A: 0  B: 0
ready: 1
         Lead A
                 A: 1  B: 0
ready: 1
         Lead B
                 A: 1  B: 1
```

The output values in A and B produce the same waveform that is expected from the depicted encoder explained before.

6.3.1.4 Application ANALOG

The current program reads every two seconds the value from the analog input depicted as AIN0 (pin P9_39). This value is printed on the terminal in raw and normalized formats, as well as in actual voltage values.

Changing the position of the switches in the analog switch of the cape alters the composition of the resistive voltage divider at the input of AIN0 and so the read values vary.



6.3.1.5 Application LEDMATRIX

This application switches on and off the LEDs in the LED matrix each second until the push button is pressed. After that, the application finishes. The picture below shows an image of the cape with these LEDs switched on:

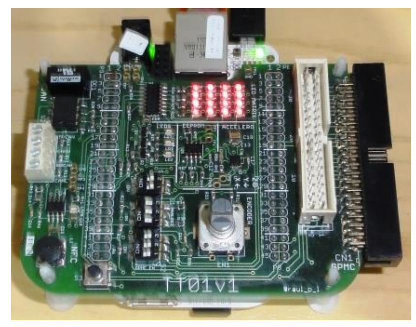


Fig 6.7: LEDs in LED matrix in cape TT01v1 switched on, corresponding to app LEDMATRIX

6.3.1.6 Application ACCELEROMETER

This application reads the value of the three axis of acceleration and represents it on the terminal console refreshing this information every two seconds.

When the cape is in a flat resting position over the table, the values are the following:

```
ACC X: -6
ACC Y: 0
ACC Z: 255
```

While the acceleration in X and Y axles are rather 0 (except some bias), the acceleration in Z is maximum. This acceleration is caused by the gravity. If the cape is bounced along the X position and put in vertical, the values change:

```
ACC X: 255
ACC Y: 3
ACC Z: -4
```

In this case, the gravity affects the X axis being this the maximum whereas the other two axis remain close to zero.

The analog situation happens if the cape is bounced along the Y axis and also put in vertical:

```
ACC X: -1
ACC Y: 259
ACC Z: -2
```

The gravity affects the Y axis being this the maximum whereas the other two axis remain close to zero.

6.4. Python programs

The accuracy of the programs written in Python has been demonstrated in previous sections in this work. Using the available modules in Phyton and Adafruit Python modules for the BeagleBone is not only easy, it is often more practical in order to quickly develop prototypes in a laboratory. What is more, the use of these libraries or modules is simpler and safer because they embrace more complex actions in order to keep a correct system status. For instance, if an application is using a GPIO that in C needed to be declared, set, and opened, the same module handles automatically these actions and prevents the system from leaving this GPIO in a wrong state after having used it.

Additionally, using some features of this language such as threads and exceptions has made the applications more flexible and safe than the original ones written in C. Such changes can be written in a much easier way than in C.

The other advantage of Python in this environment is that it is not necessary compiling the code because it is an interpreted code. The interpreter is installed in the BeagleBone and codes can be written using any editor such as Vi. In addition, the interpreter can be executed and lines of code in this language can be introduced as if it was a shell console. By this way, the code can be checked quickly and easily.

The output of the developed Python applications that use the cape is almost the same as the described in the section before, except in some formats in the output thrown by the shell. So, these outputs are not going to be showed again. To sum up: *all the cape programs run properly and safer than C programs doing what is expected from them to do.*

7. Conclusions and future work

This work was basically focused on improving the performance of the BeagleBone board. Some objectives were stablished before it started:

- Perform a proper changeover from the legacy system Angstrom to a Ubuntu or similar
- Make the GPMC driver work properly in the new system and evolve it, if possible
- Make the cape TT01v1 work properly in the new system

The overall of the objectives has been reached during the development of the current work. In addition, some extra work has been done in order to demonstrate the capacity of this board to use trending programming technologies such as Python.

The three main points of this work have required coordination between them. A good understanding on the new trends in device trees, which by the time of this project is still a new technology, has been required in order to do a proper selection.

To handle capes, the kernel 3.8 is required. Although some tests have revealed that using other kernels such as 3.14 is possible, the lack of a cape manager makes it difficult to introduce the cape hardware description in them. Overlay philosophy is not allowed in kernels from 3.13 to 3.19. What is more, the selection of this kernel also conditions the build of the GPMC module, which depends on its sources.

In future works, it should be interesting to test the performance of the kernel 4.X series. This kernel restores the cape manager, so it could be a feasible evolution for the current kernel 3.8. By the time of writing these lines, the latest kernel branch including the cape manager is 4.2.

In order to perform a proper changeover between the new and old versions of the GPMC interface driver, a good comprehension of a driver structure plus the GPMC interface in the BeagleBone processor is required. Asynchronous and synchronous prototype versions of this interface have been developed and write operations have been monitored using a test application and a laboratory logical analyzer. The analysis shows an expected behavior of the GPMC interface signals.

Configuring the GPMC can be a basis of different future works to attach the BeagleBone to other type of memories different from asynchronous and synchronous address and data multiplexed NOR memories.

Finally, works around Python have demonstrated the capability of both board and scripting language to easily and quickly develop prototype applications. Ease of syntax, modularity, and safeness are some adjectives that describe the experience with Python. To conclude the demonstration, a basic visual application using the cape accelerometer has been implemented and tested with a sensationalist result.

In addition, some future work or part of it can put its focus on developing additional applications in order to test the CAN bus and the NFC in the cape. These two devices have not been tested in the current and previous works.

8. Bibliography

[2.1] R. Pérez López. "Design of a dedicated cape board for an embedded system lab course using BeagleBone". M.S. thesis, Electrical Engineering department, Technical University of Catalonia (UPC), Barcelona, Spain, 2014

[3.1] M. M. Ahmad Zabidi, "Embedded Systems ". *UTM OpenCourseware*. [Online] Available: http://ocw.utm.my/file.php/79/01-68k-Embedded.Systems.ppt.pdf. [Accessed: 18 July 2015].

[3.2] Various contributors, "Embedded Systems", *Wikipedia,* 2015. [Online] Available: https://en.wikipedia.org/wiki/Embedded_system. [Accessed: 18 July 2015].

[3.3] Various contributors, "Arduino", *Arduino,* 2015. [Online] Available: https://www.arduino.cc/. [Accessed: 18 July 2015].

[3.4] Various contributors, "Processing (programming language)", *Wikipedia,* 2015. [Online] Available: https://en.wikipedia.org/wiki/Processing_(programming_language). [Accessed: 18 July 2015].

[3.5] Various contributors, "Arduino Projects", *Instructables,* 2015. [Online] Available: http://www.instructables.com/id/Arduino-Projects/. [Accessed: 18 July 2015].

[3.6] Various contributors, "Raspberry Pi", *Raspberry Pi Foundation,* 2015. [Online] Available: https://www.raspberrypi.org/. [Accessed: 18 July 2015].

[3.7] Iq Jar, "An overview and comparison of today's single-board microcomputers*", Iq Jar*, 2013. [Online] Available: http://iqjar.com/jar/an-overview-and-comparison-of-todays-single-board-micro-computers/. [Accessed: 18 July 2015].

[3.8] Various contributors, "BeagleBone Black", *beagleboard.org,* 2015. [Online] Available: http://beagleboard.org/black. [Accessed: 18 July 2015].

[3.9] Various contributors, "Intel® Galileo Gen 2 Development Board", *intel.com*, 2015. [Online] Available: http://www.intel.com/content/www/us/en/embedded/products/galileo/galileo-overview.html. [Accessed: 18 July 2015].

[3.10] Various contributors, "Cubieboard", *cubieboard.org,* 2015. [Online] Available: http://cubieboard.org/. [Accessed: 18 July 2015].

[3.11] Various contributors, "Odroid", *hardkernel.com,* 2015. [Online] Available: http://www.hardkernel.com/main/products/prdt_info.php. [Accessed: 18 July 2015].

[5.1] Various contributors, "TI Software", *Texas Instruments*, 2015. [Online] Available: http://downloads.ti.com/sitara_linux/esd/AM335xSDK/latest/index_FDS.html. [Accessed: 18 July 2015].

[4.1] Various contributors, "BeagleBoard", *Wikipedia.org*, 2015. [Online] Available: https://en.wikipedia.org/wiki/BeagleBoard#BeagleBone. [Accessed: 18 July 2015].

[4.2] D. Büchi, "Linux Device Tree", *Software Services AG*, 2014. [Online] Available: http://www.embeddedcomputingconference.ch/pdf_2014/4A1_Buechi.pdf. [Accessed: 18 July 2015].

[4.3] T. Petazzoni, "Device Tree for Dummies", *Embedded Linux Wiki*, 2014. [Online] Available: http://elinux.org/images/f/f9/Petazzoni-device-tree-dummies_0.pdf. [Accessed: 18 July 2015].

[4.4] Various contributors, “BeagleBone and the 3.8 kernel”, *Embedded Linux Wiki*, 2014. [Online] Available: http://elinux.org/BeagleBone_and_the_3.8_Kernel. [Accessed: 18 July 2015].

[5.2] Various contributors, “Unity (User Interface)”, *Wikipedia*, 2015. [Online] Available: https://en.wikipedia.org/wiki/Unity_(user_interface). [Accessed: 18 July 2015].

[5.3] Various contributors, “Linaro”, *Wikipedia*, 2015. [Online] Available: https://en.wikipedia.org/wiki/Linaro. [Accessed: 18 July 2015].

[5.4] Robert C. Nelson, “BeagleBone”, *eewiki.net*, 2015. [Online] Available: https://eewiki.net/display/linuxonarm/BeagleBone#BeagleBone-LinuxKernel. [Accessed: 18 July 2015].

[5.5] Various contributors, “ARMhf Linux for ARMhf devices”, *armhf.com*, 2015. [Online] Available: http://www.armhf.com/. [Accessed: 18 July 2015].

[5.6] Elementzoneline, “Compiling BeagleBone kernel 3.14 from scratch*”, Random Codes – Elementz Tech Bog*, 2014. [Online] Available: https://elementztechblog.wordpress.com/2014/11/18/compiling-beaglebone-kernel-3-14-from-scratch/. [Accessed: 18 July 2015].

[5.7] S. Kishore, “How to configure a NFS server and mount NFS shares on Ubuntu 14.10”, *HowtoForge Linux Tutorials*, 2014. [Online] Available: http://www.howtoforge.com/nfs-server-on-ubuntu-14.10. [Accessed: 18 July 2015].

[5.8] AM335x ARM® Cortex™-A8 Microprocessors (MPUs) Technical Reference Manual, Texas Instruments Incorporated, 2011.

[5.9] Various contributors, “Enabling the ADC on the 3.8 BBB Kernel without the cape manager”, *Google Groups - BeagleBoard*, 2014. [Online] Available: https://groups.google.com/forum/#!topic/beagleboard/7ejOSh9Fs5s. [Accessed: 18 July 2015].

[5.10] B. Croston, “Adafruit's BeagleBone IO Python Library”, *GitHub*, 2015. [Online] Available: https://github.com/adafruit/adafruit-beaglebone-io-python. [Accessed: 18 July 2015].

[5.11] Various contributors, “Serial Peripheral Interface Bus”, *Wikipedia*, 2015. [Online] Available: https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus. [Accessed: 18 July 2015].

[5.12] Various contributors, “I2C”, *Wikipedia*, 2015. [Online] Available: https://en.wikipedia.org/wiki/I%C2%B2C. [Accessed: 18 July 2015].

[5.13] Various contributors, “Setting up IO Python library on BeagleBone Black”, *Adafruit*, 2013. [Online] Available: https://learn.adafruit.com/setting-up-io-python-library-on-beaglebone-black/i2c. [Accessed: 18 July 2015].

[5.14] Various contributors, “Tkinter”, *Wikipedia*, 2015. [Online] Available: https://en.wikipedia.org/wiki/Tkinter. [Accessed: 18 July 2015].

[5.15] Various contributors, “Tkinter”, *Python*, 2014. [Online] Available: https://wiki.python.org/moin/TkInter. [Accessed: 18 July 2015].

9. Glossary

API – *Application Programming Interface*

BB – *BeagleBone*

BBB – *BeagleBone Black*

CAN – *Controller Area Network*

CISC – *Complex Instruction Set Computer*

DHCP – *Dynamic Host Configuration Protocol*

DTB – *Device Tree Blob*

DTC – *Device Tree Compiler*

DTBO – *Device Tree Blob Overlay*

DTBI – *Device Tree Binary Include*

DTS – *Device Tree Source*

EEPROM – *Electrically Erasable Programmable Read Only Memory*

FPGA – *Field Programable Gate Array*

FTP – *File Transfer Protocol*

GCC – *GNU Compiler Collection*

GPIO – *General Purpose Input Output*

GPMC – *General Purpose Memory Controller*

HDMI - *High Definition Media Interface*

I2C – *Inter-Integrated Circuit*

IDE – *Integrated Development Environment*

IoT – *Internet of Things*

LED – *Light Emitting Diode*

NFC – *Near Field Communications*

NFS – *Network File System*

OS – *Operating System*

PWM – *Pulse Width Modulation*

RISC – *Reduced Instruction Set Computer*

RTOS – *Real Time Operating System*

SFTP – *SSH File Transfer Protocol*

SDK – *Software Development Kit*

SoC – *System on Chip*

SPI – *Serial Peripheral Interface*

SSH – *Secure Shell*

TI – *Texas Instruments*

UART – *Universal Asynchronous Receiver Transceiver*

VFB – *Virtual Frame Buffer*

VNC – *Virtual Network Computing*