Escola Politècnica Superior
de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

TREBALL DE FI DE CARRERA

TÍTOL DEL TFC: Middleware ALOE for DSP TMS320C6455

TITULACIÓ: Enginyeria Tècnica de Telecomunicació, especialitat Sistemes de Telecomunicació

AUTOR: Arnau Mata Llenas

DIRECTOR: Antoni Gelonch Bosch

DATA:

Títol: Middleware ALOE for DSP TMS320C6455

Autor: Arnau Mata Llenas

Director: Antoni Gelonch Bosch

Data:

Resum

L'objectiu bàsic de partida és el de donar més flexibilitat als sistemes de quarta generació dels sistemes mòbils. Donat que cada cop més s'utilitzen sistemes reconfigurables-reprogramables en la implementació dels sistemes de comunicacions, es planteja analitzar aquells mecanismes dels dispositius processadors actuals (DSPs) que tinguin una alta importància alhora de proporcionar flexibilitat en la gestió del sistema.

D'aquesta manera, es porta el middleware ALOE, usat per aplicacions de Software Radio, de la seva implementació en ordinadors x86 corrent Linux a els processadors DSP de Texas Instruments.

Pel camí s'avalua la interfície d'Ethernet de que disposa la DSP per tal de poder-ne caracteritzar les prestacions que ofereix en entorns de temps real.

Title: Middleware ALOE for DSP TMS320C6455

Author: Arnau Mata Llenas

Director: Antoni Gelonch Bosch

Date:

Overview

The starting aim of this project is to give more flexibility to the 4th mobile generation systems. Since reconfigurable-reprogrammable systems are becoming each time more widespread in the communications system implementations, it is set out to analyze those mechanisms of the current processor devices (DSPs) that have high importance when flexibility has to be provided to the system management.

This way, the ALOE middleware - used in Software Radio - is ported from its original x86 computer running Linux to the DSP processors from Texas Instruments.

Meanwhile, the Ethernet interface that the DSP provides is evaluated with the objective of its characterization in real time environments.

INDEX

1. INTRODUCTION 1

1.1. The Software Radio Concept 1

1.2. Software Radio Challenges 2

2. ALOE 4

2.1. The ALOE Concept 4

2.2. ALOE's structure 5

3. PROJECT ENVIRONMENT 8

3.1. Digital Signal Processor 8

3.2. Programming environment 8

3.3. DSP operating system 10
3.3.1. TI DSP/BIOS Basics 10

4. TMS320C6455 DSP ETHERNET INTERFACE 16

4.1. Choosing a TCP/IP Stack 16

4.2. Network usage computational cost 17
4.2.1. Measure tools & procedure 18
4.2.2. Results analysis: throughput vs. time slot 23
4.2.3. Results analysis: used CPU vs. time slot 24

5. PORTING ALOE TO TI'S TMS320 DSP'S 32

5.1. Hardware Abstraction Layer 32

5.2. Hardware API Reference 33

6. FUTURE WORK 44

7. ACRONYMS 45

8. FINAL NOTES 46

9. BIBLIOGRAPHY 47

10. ANNEX I: MATLAB CODE 48

1. INTRODUCTION

Nowadays, it is usual that in a single electronic device there are multiple telecommunication hardware elements. This is because users need to communicate using different kinds of RAT (Radio Access Technology) at the same time, while maintaining always the contracted QoS (Quality of Service). It is clear that having one device per each RAT is not a viable option: imagine acquiring one device to access Internet, another to make mobile calls, another for device and accessories interconnection and another to receive GPS signals. The sum of the mentioned gadgets forms what we today know as a mobile phone. And while the first ones are not portable all at once, the second one is.

The current solution is, as we have seen, embedding several communication hardware elements (one for every RAT) in the same device. They are switched depending on user's requirements. But there is an inherited problem; if a new technology appears, a user has to buy a new gadget because the old one is unable to access such RAT: it has not been defined at design time for this purpose neither any mechanism enabling it to reconfigure itself has been designed. Also, the user ends up throwing away his devices in order to acquire new ones with up-to-date technology.

Therefore, the next problem to solve in the telecommunications devices designing field is being able to reuse an existing hardware for accessing different RATS – not necessarily all of them yet available. It's at this point were the idea of Software Radio appears.

1.1. The Software Radio Concept

The Software Radio concept was first introduced by J. Mitola 1 as "a radio whose channel modulation waveforms are defined by software". Traditionally in communications devices, most of signal processing operations were done using hardware; Software Radio tries to move as many processes as possible to the software world.

This is being currently possible thanks to the great evolution in computing machines, since billions of operations per second are a usual order of magnitude for Software Radio. However, and due to current limitation in processing power, Software Radio also needs to include hardware non-programmable but reconfigurable devices (digital or analogue). Implementing a Software radio concept following these guidelines would provide the following example features:

- Reconfiguration "on-the-fly": the same device can act as a cordless phone, a mobile phone, a WiFi adaptor, a GPS receiver, etc. The terminal can switch between one service to another (or even use more than one at once) during its normal execution.

- Quick and easy upgrade to new technologies or enhanced features. This upgrade could be even done over-the-air.

- Smart or cognitive radio that adapt the radio environment (background operator network and user terminal) to instantaneous population distribution and QoS requirements.

Also, we should also be concerned about the benefits that such technology should bring in terms of environmental issues. Reusing hardware elements or the possibility to upgrade some device capabilities by terms of attaching another device (but keeping the old one) should contribute to reduce the overall refuses produced by the developed world as, nowadays, huge amounts of old devices are currently being thrown away when they are not useful anymore.

1.2. Software Radio Challenges

Once the Software Radio model is accepted as the first step on providing the previously stated features, we easily realize how it will not be an easy challenge. Software Radio is a technology; is a guideline on how communications systems should be designed if we want to obtain a set of benefits. Its implementation, however, is not a direct step.

First of all, the designer will try to find a platform where the communications system (implemented via Software Radio) is going to be executed. The designer will find a great diversity of platform families available in today's market: several companies manufacture different DSP, different FPGA, different microprocessors (GPP), etc. The designer will have to choose one of these platforms. When a new technology appears or a new feature is desired to be included, it might occur that the selected device is not powerful enough to process a determinate algorithm or a required functionality is not available. The designer will find the problem of selecting another device and as a consequence rewrite some pieces of code, or in most of the cases, the entire code. The Software Radio concept is not efficiently implemented. Two of its main objectives, the ability of reusing code and the flexibility, are lost as soon as the platform changes.

The reason why moving code from one platform to another is so difficult is because of the strong dependence existing between platform architecture and radio applications. These applications require a low-level access to platform resources as timers (real time control is very important in this applications), codecs, memory controllers, interfaces configuration and interaction, etc. and these interactions are very architecture dependant.

Also when we think in which is the common structure that these radio applications share, we easily find out how most of them share some algorithms or pieces; at most, they change a few parameters or variables but they maintain the main structure of the algorithm. For example, UMTS and WLAN standards use both CRCs and Convolutional Codes (although their parameters are not the same). We could take advantage of this issue and develop applications reusing

some parts of other ones; that means, to implement a concept similar to Object Oriented Programming used in common computing platforms but in signal processing applications. This would drastically reduce the time-to-market of new devices as developer could just link some of the pre-written objects and configure them to match the current application. In the near future, this could not be a feature anymore and become a must.

We will begin by drawing down some challenges Software Defined Radios are currently facing up:

- Most of the RATs share some algorithms although their parameters are different, a tool enabling the possibility to reuse software pieces is necessary.

- Developing or maintaining several of these applications would require of organizations in order to maintain common algorithms unified.

- Large matrixes of embedded and heterogeneous chips require a tool to homogeneously manage them.

- Signal processing applications require a close contact with hardware devices. As a consequence, it is difficult to move code from one platform to another.

These and other problems leads to a conclusion: it is needed to establish a common design policy surrounded by a background framework where homogeneous solutions for these problems are established or at least a future solution is enabled.

It is here were FlexNets open source initiative comes into play. Its goal is to provide the design policy and background framework that we have seen that are needed for developing Software Radio applications. This essay pretends to port the FlexNets's ALOE (the background framework) from its current implementation in x86 computers using Linux to the Digital Signal Processors (DSP) from Texas Instruments.

This way, all the computing power provided by DSPs will be available jointly with the provided by x86 computers. In chapter 3 we will see why DSP are so much used in this kind of processing. Also, it will be the first time that ALOE is ported to another platform, so we will be able to evaluate how much time it costs and what kind of problems we have.

2. ALOE

In this chapter, the ALOE (Abstraction Layer & Operating Environment) framework will be described. Those who already know ALOE should skip it since here is not explained anything new. All the others are encouraged to follow, because all the next presented work has been made following the ALOE guidelines.

2.1. The ALOE Concept

One of the most relevant objectives that a common framework to develop and deploy software radio applications need to meet is to eliminate platform (hardware and support software) dependencies. In this context, the possibility to add as much processing resources as required for a given application, making not necessary to change it at all, is of must importance. The possibility of (re)configuring the radio terminal and/or the base station built by using different hardware from different providers requires the capability of adding (and removing) plug-and-play hardware to (from) the system. Different hardware topologies, configurations and, above all, assigned tasks impose restrictions on the integration of different hardware to construct software radio platforms.

Also, from the point of view of a radio application, it is commonly assumed that such application is built based on a set of software modules (hardware or software based) that communicates among them. The common used terms denominates such modules as “objects”. Therefore each one of such objects are signal processing blocks, some of them with important requirements in terms of computing resources, that needs to acquire/deliver information from/to other objects.

Being this said, ALOE supports de following functionalities:

- Flexibility

  The framework is based in the capacity of reconfiguration as the basic mechanism providing flexibility.

- Execution control management

  The coordinated execution of the whole system is assured in order to fulfil the QoS required by the waveform being used.

- Hide the platform heterogeneity to the radio application

  There are two basic mechanisms to provide such feature: abstraction layers and using a high-level language like C (so the compiler will translate to machine code).

- Computing resource management.

The different objects to execute are distributed among the available computing resources. In addition, it is capable to assure the overcoming of the real-time constraints.

- Computing object's data packet oriented messaging.

Packet oriented in order to let each object to have its input and output of information in a way where it doesn't matter if the sender/receiver is or not on the same processor.

- Parameter evolution capture and processing during execution.

Allows support for Cognitive Radio strategies.

- Time slot based division of the transmission medium.

As in ALOE each RAT is broken down to several objects which transfer data among them, we do not have the typical continuous flow. Instead, each object is continuously receiving, processing and sending information in blocks. As we use A/D and D/A converters, we must make sure that this flow goes at the right speed. Therefore, the time is divided in slots, the objects are designed so they make a cycle each slot and ALOE assures that the messages sent in a certain slot are available for reception in the next one.

2.2. ALOE's structure

ALOE has been designed as a set of layers (to provide abstraction) together with daemons responsible of providing required functionalities. In Figure 2.1, a representation of the abstraction layers involved is drawn. On top, we see the connection graph of the object pieces building a radio application. This is the Abstract Application Layer, where the RAT is shown as a group of objects. On the next level, objects are mapped into real Processing Elements (PE): they physically need a place to be executed or located on, like a computer, DSP, FPGA... The ALOE Layer is the abstraction layer found between the real application processes and the Hardware Layer. From above, an ALOE Platform is represented although different PE's (each of them enabling ALOE functionalities) are under it. To the user it's a unique virtual platform which is accessed by means of a Software Library.

Underneath, we have the Hardware Abstraction Layer - HAL - (which is the only piece that is platform-dependent) and the Software Daemons.

The first one can be understood as a software layer sometimes included inside the operating system which from one side is in close contact with the hardware while in the other side deals with other software components. Thus, from the point of view of the software running over the HAL, there isn't any detail about the underlying hardware. All the hardware procedures, like communication between several processors, in/out interfaces, memory ranges, context switching, etc. are totally hidden to the software piece running over.

The HAL does not introduce any new concept at all. Some tasks are encapsulated and offered as software services to higher layers. But the most important feature is the ability to make portable between heterogeneous platforms these encapsulated tasks. An example of this is a common Operating System. In such architectures, most of the functions and services can be described regardless of the hardware.

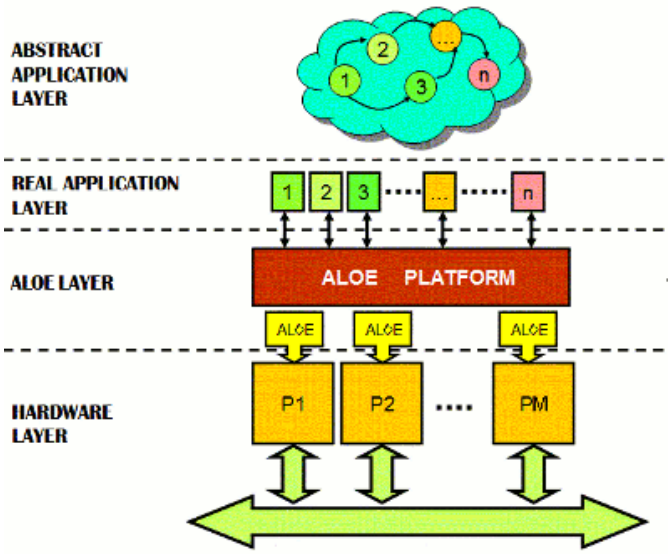

Fig. 2.1 ALOE layers schematic

Then, if we incorporate the HAL in a system, building applications requiring some kind of close contact with the hardware becomes easier because it's not necessary to make different code for different hardware configurations. For example, accessing a resource located outside the common bus of the processor would require a different code depending on the bridge device architecture. HAL routines are ready to deal with these differences. The *real* access to the resource will be transparent for the application.

The other component, software daemons, is a group of extra services that have been defined to help in the management and execution control of radio applications. The provided functionalities will be always available in the ALOE platform. However, as it is formed by joining different devices (PE) and interconnecting them in some defined way, it is not necessary that they are present in each PE. Moreover, there are some of these daemons which only exist in one master PE. In figure 2.2 we can see a sample ALOE and object installation in a device.

In the scope of this project, only the Frontend, SWLoad (software load), and Exec daemons will be considered. These are the minimum ones required to run objects. With this and the Hardware Abstraction Layer we have sufficient to manage the processor, the loaded objects and their status.

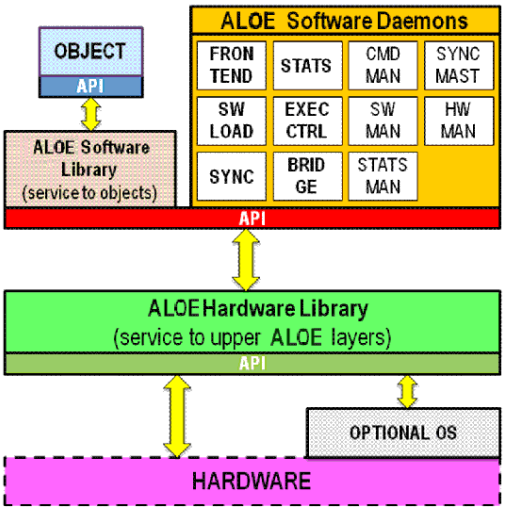

Fig. 2.2 ALOE components diagram

To end this chapter, we have to introduce another functionality of ALOE. We have previously mentioned that each object and daemon can transfer messages to others. The method to do so is by creating internal interfaces, usually with queues, where the sending and receiving endpoints register to. This way, when the sender has a message it leaves it there and the receiver goes continuously polling this interface to retrieve it.

3. PROJECT ENVIRONMENT

This chapter presents the way of how this work started. It defines the DSP that we are using, as well as the development IDE, special programming rules and Operating System.

3.1. Digital Signal Processor

The idea of using a DSP comes from the need and type of computation that we are doing. As our applications are in the Software Radio scope, what we usually need to do are calculations with a stronger presence of multiplications and additions, like on filter processing or calculating Fast Fourier transformations.

Therefore, using a GPP is not the best idea. This is because such type of processor has been thought to offer programmability, and therefore flexibility. Moreover, in this type of processors it usually takes various cycles to perform a single multiplication.

In the other hand, a DSP is based on fast instruction execution in a less flexible environment. In order to accomplish that, DSP's offer different architectural features like pipelining, parallel multiply-accumulate operations, hardware-controlled looping, a Very Long Instruction Word format, and others parallelization-friendly characteristics. Then, a DSP is able - for example - to perform multiple multiplications in one cycle.

Note that this kind of DSP, using VLIW architecture, forces the code developer to use a high-level language like C; relaying in a compiler which optimizes the code to the characteristics of each processor. Assembly code gets pushed to a second plane: when more optimization is needed.

In order to choose a processor we decided to follow the Texas Instruments C6000 High Performance family that we have already used in the university (specifically, a C6416). That family has been used for telecommunications purposes, as it is a high-performance range. Among others, there is available the TMS320C6455 DSP Development Starter Kit (DSK). This board includes a 1.2 GHz C6455 DSP processor, a 10/100Mpbs Ethernet interface, a Serial RapidIO bus interface, 128Mbytes of DDR2 memory and other features. Also, the DSK includes a copy of the TI's integrated development environment called Code Composer Studio which we also do already know.

3.2. Programming environment

In the Figure 3.1 we can see the Code Composer Studio IDE (CCS). We will be using this one in order to produce, optimize and debug the code.

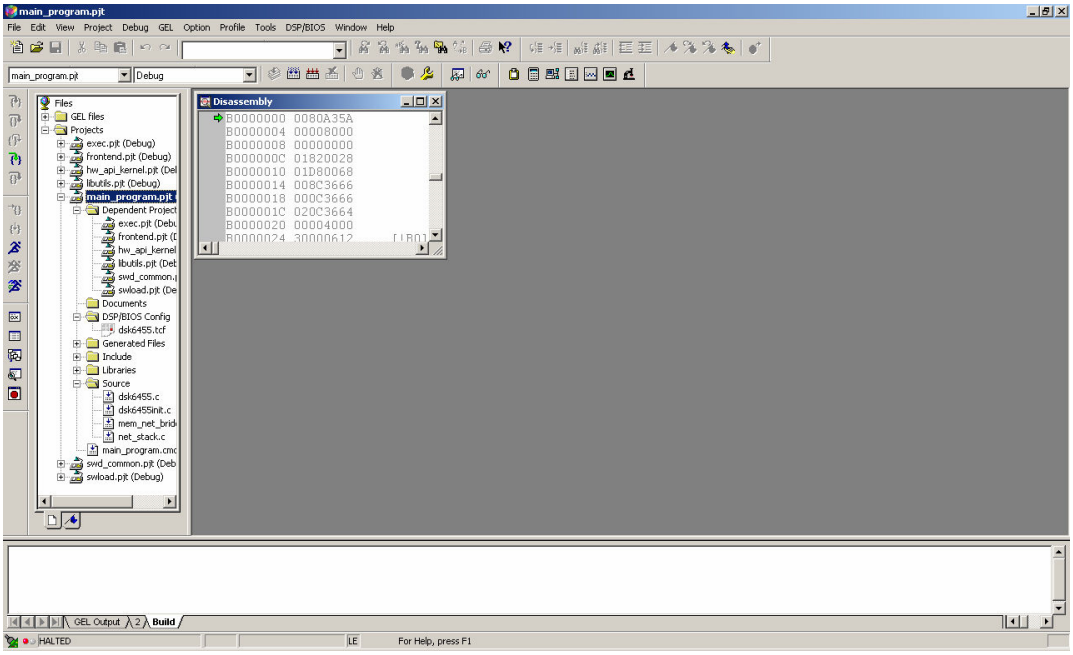

Fig. 3.1 Code Composer Studio v3.3 default view

In production, is of our interest the capability of CCS to manage projects as executables or libraries; being able to make dependencies between them. Also, it gives us a wide range of compilation and linking options. The rest is like any other IDE.

About the optimization, it has code profiling tools. In our case we haven't used them since the fact of using a network interface doesn't make it too much straightforward.

Finally, when debugging, this IDE allow us to load the program directly to volatile memory, allowing its visualization and modification in real time. Also, we can set breakpoints, execute line by line, see the assembler code being executed for each C instruction and save(load) memory ranges to(from) the computer among other things.

Aside, there is another kind of programming environment which has also to be explained: the mandatory software guidelines. These are three:

1. The code must be time-slot division compatible
2. It has to be optimized
3. There can't be any blocking functions

The first two are obvious given our project scope. As for the third, as we are dealing with a middleware for Software Radio applications, we have to assure that there won't be - never - any kind of possibility that the code gets interrupted because it has to wait for some kind of resource to get available. This way, we can assure that all what has to execute in one time-slot does so.

3.3. DSP operating system

In this work, the ALOE middleware has to end up running in a DSP processor. Therefore, the Hardware Abstraction Layer has to be created for such device. We have two options: to make from this layer a basic, custom, integrated operating system or to use an existing one (TI's DSP are known to have run various OS).

As we want to spend more time with ALOE than creating an OS, the decision was made to use an already created operating system. This also would provide and assure us stable base functionalities. Moreover, we need a real-time operating system (RTOS), since this characteristic is a must for Software Radio.

There are two different RTOS for DSPs: the one from Texas Instruments (DSP/BIOS) and various flavours of Linux. As the DSK board that we have chosen contains a royalty-free license for DSP/BIOS, which has been created specially for this platform, we decide to give it a try. Its advantages exceed the defects: it has just the basic needed features, is lightweight, is easy to work with, is under continuous development, it does integrate with Code Composer Studio and it supports all TI's DSPs. In the other hand, is proprietary code, so we have to expect to have all the characteristics we might need available.

3.3.1. TI DSP/BIOS Basics

In the first place, if we are going to use this RTOS we should have at hand the API Reference guide of one's version [2] and the User's Guide [3]. The first one is platform and version dependant, and should be our first knowledge source. The second one is useful when not knowing how to do something and for checking if a feature is present on all platforms or not. Both of them are available online and in the folder where DSP/BIOS got installed (there are also some code examples using different functionalities).

In the second place, as it gets installed from the same CD where Code Composer Studio comes; it is advisable to perform an upgrade to a latest version (Texas Instruments releases new versions when bugs are found). There shouldn't be any worries, since DSP/BIOS is pretty integrated with Code Composer Studio and we can have various versions at the same time, being able to switch to the one that suits us more. This should be done from the *Help* menu of Code Composer Studio, then *About* and finally *Component Manager*. Note that in the *About* window we can see which versions do we have of the IDE, the DSP/BIOS and the code generation tools (the set of compiler, linker, etc.).

There are three ways we are going to interact with this RTOS: by its static configuration, with the API we can use in our code and with its real-time information exchange from the DSP to Code Composer Studio.

The static configuration of the DSP/BIOS is where compile-time objects and options are set (this file gets compiled, generating some useful headers for us).

It is saved in text files with the *tcf* extension. We can edit it graphically or by hand. Although a graphic interface is a lot eye-catching, we have to bear in mind the way it edits the source code: each time there is a modification, it is saved at the end of the file, even if it is actually modifying a previous declared value. In the Figure 3.2 we can see how the graphic interface looks like.

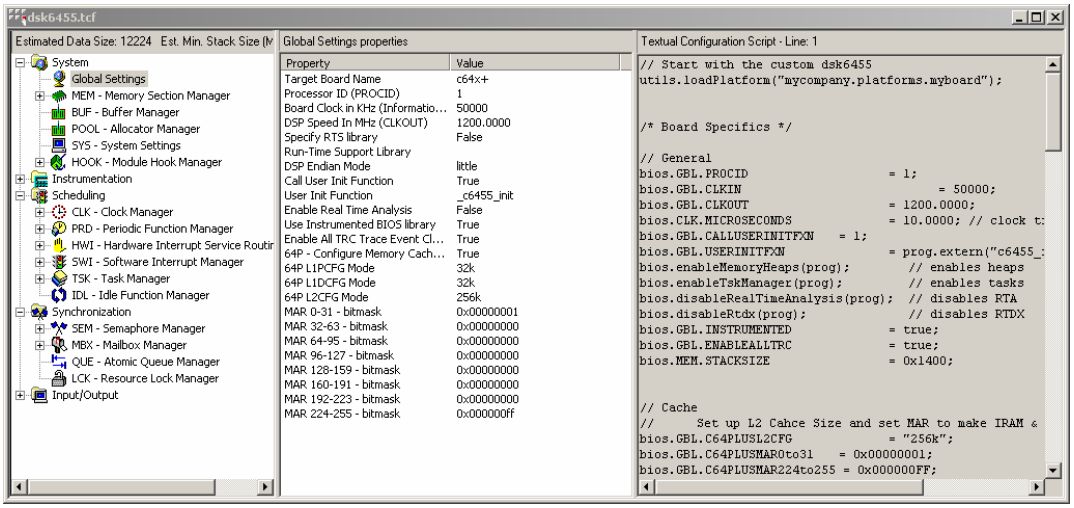

Fig. 3.2 DSP/BIOS graphical editor

By default, when we create a new DSP/BIOS configuration we are offered to choose between a few pre-defined DSP boards. After, the main configuration file will import setting of the selected device. In order to assure compatibility with different versions of the RTOS, we chose to unify the entire configuration to a copy-per-project format, except for the basic board-dependant parameters like available memory, processor speed, etc which are stored in a custom pre-defined board created by us.

Let's now comment the sections where useful functionalities that are usually set at compile-time are found.

- Global Settings

We can choose, among others, the same endianess configured in the DSP board, to call a function before starting the DSP/BIOS and in the case of our board (a C64x+) the cacheability of the memory.

- Memory Section Manager

DSP/BIOS divide the continuous-addressable memory back to its basic modules: internal, external, cache… It also allows us to set options like:

  - the RTOS stack size
  - where the allocatable memory is
  - where to store the DSP/BIOS objects and specific compile sections

  - have compile sections duplicated in a fast volatile memory for its usage and slow non-volatile memory for storing
  - create custom memory segments which can be used afterwards on our code

- System Settings

Where we can set debugging options like how much space reserve for the RTOS's traces and declare what to execute on an error, at exit…

- Log

It allows us to use DSP/BIOS printf-like functions which are stored in a buffer in the DSP. When it does not have any code to execute, the prints get transferred to the computer. This lets us avoid time interruptions due to the DSP to start transferring data to the computer, with POSIX printf.

- Clock Manager

Here, we can set up which code-accessible timers we want to use as well as which period they have. It's important to remember the period of the main timer interrupt, since some of the DSP/BIOS functions do use this base unit.

- Periodic Functions Manager

This RTOS allows us to define C or assembler functions and set them up to be executed each certain time. An example usage would be to call a function where a day-hour-minute-second timer is incremented for example each second. The DSP/BIOS always tries to have them executed on time. Otherwise, it writes to the main log to alert us.

- Hardware Interrupt Service Routine

A full list of the available interruptions is given to the user, including the typical on-pin-value-change. One can configure what is executed in each one. They have maximum priority.

- Software Interrupt Manager

Following in priority there are the software interrupts. They are like the hardware ones, but with the difference that they are called from the program being executed in the DSP. They are arranged by priority.

- Task Manager

The last being executed are the typical user threads. Those are internally ordered also by priority. Here we can configure if we want automatic scheduling (round-robin type) or manual, management functions hooks, which stack has each thread…

- Idle Function Manager

When there is nothing else to do, the DSP starts executing the idle thread. In this section we can manage which functions are going to be executed and in which order. If we use the real-time data exchange between DSP and PC, here is where the transfer is done.

- Semaphore Manager

  They are useful if we want to share resources. We can choose between binary and counting semaphores, depending on if we want a busy/available semaphore or a busy with Y waiting/available one. Both of them are managed in the same way, but after, one must remember to use them correctly.

- Mailbox Manager

  They are basic queues used to transfer data typically between threads. They do copy the message, so the user must specify the maximum number and size of them.

- Atomic Queue Manager

  The objects managed are the same as the mailboxes, but with the difference that it stores a pointer to the message and have more access options. Therefore, they are recommended if we need speed or with big messages.

The second way of interacting with DSP/BIOS, using the API, is the one we will use when writing code. We have the same objects as in the static configuration, and we can use and manage them dynamically. It is recommended to have its related documentation at hand, as most of the function calls have special characteristics. Also, when writing in C it is necessary to include headers which reference these objects. Its name convention is simple: one should include a file with the same name as the first three characters of the functions that are being used. For example, if we need to call *MEM_alloc* then we need to include <*mem.h*>.

As a note, we should comment that clock and task functions related to time don't use the same base of times. For example, we can get the current time in two formats (low and high precision). Each one goes at a different speed, and both of them depending on the processor. In the other hand, functions like *sleep()* do require that the time is given in portions of the previously seen main timer period. Unfortunately, this value is not accessible from code, so is responsibility of the programmer to assure that the same is on his code and the DSP/BIOS configuration file. At least, the API provides functions in order to know the relation between the first type of time and the second one.

Finally, the last way of using this RTOS is for debugging. Code Composer Studio lets us interact with the DSP/BIOS while being executed or when stopped to see its behaviour. We have seen that for our needs, three of them are necessary:

- Execution Graph

  After refreshing this window, one can see what the DSP was doing before that. The quantity of time displayed depends on how much messages have the mandatory *LOG_system* log of the DSP/BIOS. In the Figure 3.3 we can see an example output. There are two things that one has to bear in mind: only the statically created objects are displayed by itself, and that the horizontal time-base is not actually the real time (on should look on the *Time* values, where a tick is made each time the main timer is incremented).

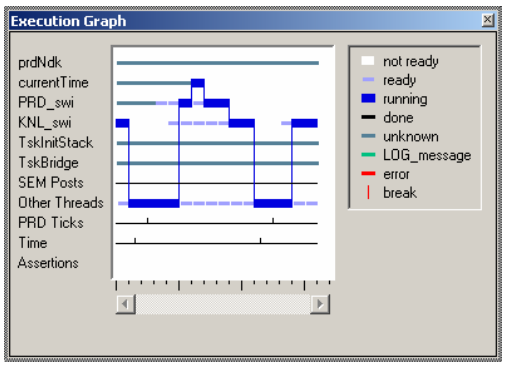

Fig. 3.3 Code Composer Studio Execution Graph window

- Message Log

As we can see in Figure 3.3, it allow us to select the log object from which we want to see its output; without affecting the running code. Also, if we double-click over one of the lines of the output of the DSP/BIOS – in the *Execution Graph Details* log - then we see the corresponding time marked in the Execution Graph.

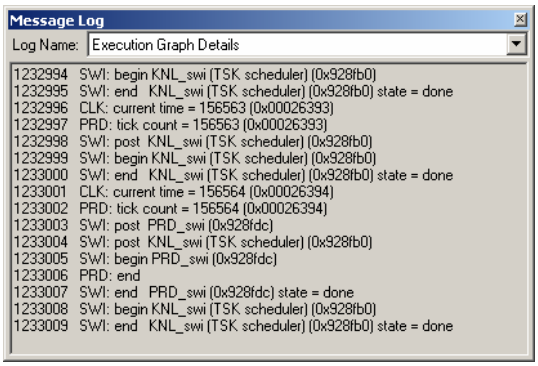

Fig. 3.3 Code Composer Studio Message Log window

- Kernel/object view

Here one can look up the status of the objects created statically and dynamically. As we can see in Figure 3.4, it allow us to check parameters like free and used memory, task stack status and peak usage, semaphores and mailboxes usage…

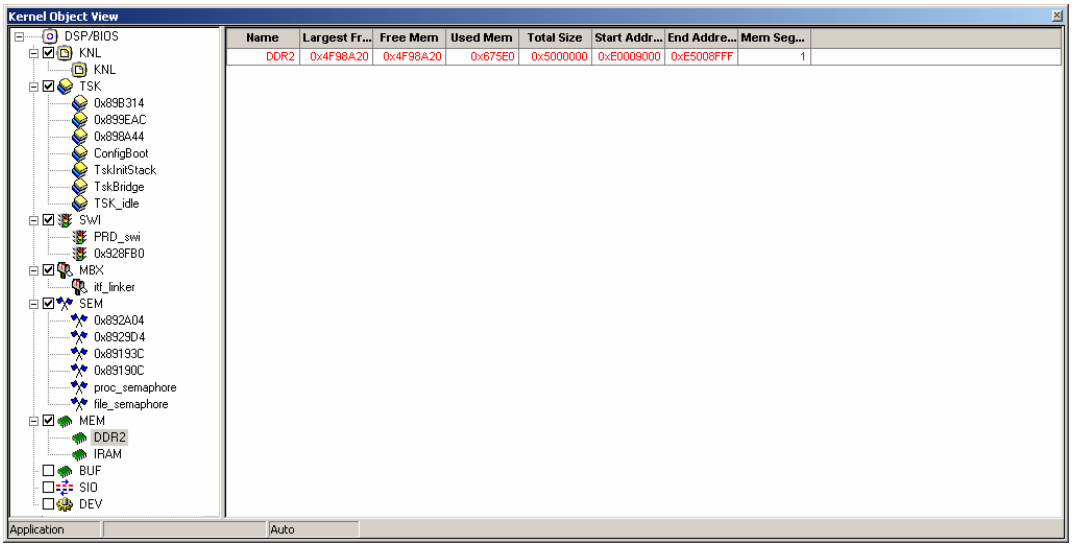

Fig. 3.4 Code Composer Studio Kernel Object View window

4. TMS320C6455 DSP ETHERNET INTERFACE

Our objective is to port ALOE to the DSP processors; thence knowing how to use interfaces for external communication is a must. As the current ALOE implementation on x86 GPP processors is using Ethernet as its main intercommunication network, it's obvious that the first interface from the DSP that should be used is the same one.

The SDK being used in this project has a 10/100 Ethernet interface embedded in the board. It also comes with a basic package for accessing such interface, coded with C. As we do need to use the TCP protocol to communicate with the Linux x86 machines and dealing directly with the interface is not nice to deal with, we will need a TCP/IP Stack.

4.1. Choosing a TCP/IP Stack

As we do want the DSP processor to spend as much time as possible with objects computation, there are four requirements for the TCP/IP Stack:

1. Reduced CPU usage
2. Reduced memory footprint
3. Easy porting to DSP platform
4. Capable of integrating with provided Ethernet module access software

In other words, we need a light-weight stack that allows us to use the TCP protocol - as well as other like ARP and ICMP- while its use in DSP platforms should be feasible.

There exist two stacks which can be useful to us: LwIP and Texas Instruments' NDK (Network Development Kit). In order to choose the best one a short description and comparative is presented in the following lines.

- LwIP
    - Designed for embedded systems where memory is rather limited. They don't specify how much CPU it needs to work properly.
    - Its platform porting capability is good enough. There are few files to edit, easily organized depending on its connections to the operating system and Ethernet device driver.
    - It has a modular structure, so a custom stack could be built with our specific requirements.
    - Is open source. Therefore, access to all source code is granted and is cost-free.
    - It has a very good documentation, and there are multiple ways to ask for support (also for free).

- NDK
    - Designed for Texas Instruments' own DSP boards.
    - It does not need to be ported. Moreover, it has the Ethernet device drivers built in.
    - There are benchmarks on the same board that we are using, which show that it does not need many resources.
    - Supports advanced features like NAT and application-level protocols.
    - It's proprietary software. There are just a few source code files released to the public.
    - Although there is an evaluation version with run time restrictions, its cost is quite expensive. At least, it is royalty-free.
    - The documentation is somewhat limited to a description of how does the API work [5].

In a first approximation, the LwIP stack looks like the best option. It would allow us to create a reduced version. Also, it would be easier to debug it and to adjust it to the real-time requirements that we must meet. Meanwhile, the NDK looks ugly by the fact that it does not allow too much customization. In fact, we only can remove components like NAT and high-level protocols.

In the other hand, after checking how much it would cost to use the Ethernet device driver in LwIP, we change our opinion: TI's NDK is assured to work in conjunction with DSP/BIOS and the Ethernet device whereas LwIP would require to spend time trying to get it to work. Also, the NDK has been tested to work alright after being compiled by the TI's compiler with DSP-specific optimizations. About the documentation, as we are going to use the Berkeley Sockets, we see that what provides TI will be sufficient. And last, but not least, TI's does assure quality in their Stack: they have customers how have paid for it and therefore may not allow a faulty behaviour.

4.2. Network usage computational cost

As we want to be able to share out processing modules among a network of processors, we need to know how much can contribute each one. Hence is of our interest to determine which overhead introduces the fact of being using the network and maintaining it.

Also, we need to know how the network throughput does behave, as we must ensure that the real time constraints are met (remember that ALOE has to be able to transfer messages between objects which can be on the same processor or not while ensuring that are available on the next time slot).

Even more, the data analysed in this chapter will be useful when designing a Software Radio application because the relation between time slot value and network throughput.

Finally, the development of the software needed to make the measurements allows us to play with TI NDK TCP/IP Stack's and TI DSP/BIOS's configuration values; apart from the way in which we interact with the network. This way, we can optimize the base system where ALOE will be ported to.

4.2.1. Measure tools & procedure

We can split up the needed tools in two parts: pre-processing and post-processing. Let's introduce them as they went being necessary.

Before introducing them, we should talk about the software running in the DSP. Due to the initial decision of using TI DSP/BIOS, we have to bear in mind the disadvantages of using proprietary software. There were two main problems. The first one was that the sockets, configured as non-blocking, did not do so from time to time. The documentation did not tell that it was needed anything more. After trying multiple possible solutions, we found that if we put the option MSG_DONTWAIT when calling send and receive functions then it behaves always in a non-blocking mode as we wanted. For the record, this option is used when a socket is in blocking mode and a send/receive call must be made without blocking. The other problem was that in the scope of knowing how much CPU cycles are being used for the network, the fact of not being able to modify or see the code does not allow us to make such measurements by software. This is because we are not able to change the threading scheduler in order to mark when a task is started and when stopped. Furthermore, we don't even know how many threads are running for sure: at least one for the Stack, one for the scheduling and another for our functions. We also discarded using the DSP/BIOS's Execution Graph, as we had problems obtaining all the data if the log size was bigger than about 64 Kbytes (and if in 10 microseconds it needs ~60 bytes, for capturing 100 slots we would need ~600 Kbyte – and we would need more than to obtain coherent results).

Then, the solution comes by means of hardware; as we can create a thread with the lowest priority (idle task) and can modify the output value of the two control bits of DC_REG register. This register is used to monitor and control a DSP daughter card. We will be using its two control ports. They can be looked up in the DSP board description book [4].

As a resolution of 1us is more than sufficient for time slots in the range of milliseconds, the idle task will consist of a never-ending loop where the output value of one of these control ports changes approximately each microsecond. It is not necessary to have a determined period, since we know that in each time slot there will be at least one period without interruptions which will be measured (we are not running anything else). Also, we could not assure that the compiler does not translate the C function as the same machine-code each time we build the program with different configurations, so we should analyse for each compilation which period does it have.

Besides, we will make our network using and maintaining thread - which executes in the start of every time slot - to modify the other control port. This way we will now also when a time slot starts and ends. This makes us program the thread in a way which we have to know when the next time slot starts, so when done we can sleep the task until then and with nothing interrupting it.

In order to measure those pins variations we will use a Logic Analyser. In the laboratory we have at our disposal a Hewlett Packard 16702A Logic Analysis System – like the one in the Figure 4.1.

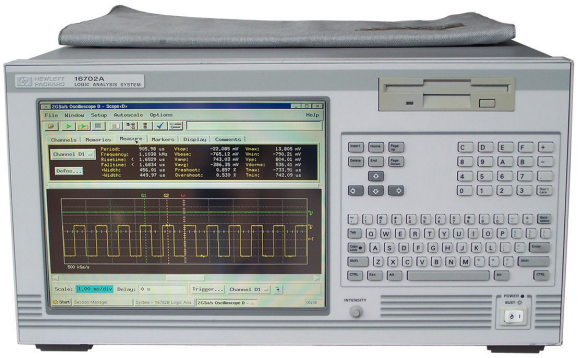

Fig. 4.1 Hewlett Packard 16702A Logic Analysis System

Let's take a look at the main characteristics which we will be using:

- 32 data channels (input and output)

We are going to use two of them, plus a ground connection, for the control ports pins.

- 64Kbits or 128Kbits deep channels

From which we choose 64Kbits since it's the only one that lets us choose different sampling speeds. As it captures high or low states, in each capture there will be 64K samples. We will have to assure that in those samples at least a time slot has been covered.

- Capture period of 500ps, 1ns, 2ns, 4ns and so on up to 65536us.

Depending on which time slot are we using, we will have to choose on value or another. In this work we always tried to cover one time slot in 64Kbits*period time.

- Programmable

Lets us choose triggering logic conditions, what to do with sampled data and which post-processing we might want to do.

- Network capabilities with support for SAMBA shared folders.

  Will be useful for transferring sampled data to the computer where it will be analysed.

- 4 GBytes of available hard disk space

  This will be enough to save configuration and data information.

- Three different file type for captured samples

  The obtained data can be saved in three formats: internal, fast binary and ASCII. Its size and time-to-store characteristics do increment consecutively.

As we want to capture the values in the output of the idle thread and our specific interest is in what happens in a time slot, we use the other output as the trigger condition. Also, we configure a sample period such that covers an entire slot. Therefore, when a time slot starts we also start sampling values at a speed such that when the 65536^{th} is acquired, the time slot has passed.

With initial test measurements we have seen that with a number of captures starting on 500 there is sufficient to obtain correct results.

The logic analyser that we use saves each capture on a file. Consequently, it has to be saved to the hard drive. This entails that between each capture there will be some lost time. Moreover, the place where stored and the type of the file makes this time to change, obtaining speeds from about 10 to 60 captures per minute. It is recommended that the captures are saved with a pathname as short as possible. Also, the internal filetype should be used.

With the given configuration, a usual measurement with 800 captures takes about 15 minutes to complete. This seeking for fast measuring comes from the need of testing different values of time slots. This way, the human-required part is done as fast as possible, and all the post-processing is done without necessary intervention. The other option would have required human presence in a much slower frequency, but needing to be there each time a measurement ends (which would take at least 2h instead of 15min). There is also a second reason which has got us to do this: the post-processing.

As we decided to save the data in the logic analyser's internal format, we need after to transform it to a data type that can be read for analysis. The available tool that is most useful for us in order to plot and examine the data is MathWork's Matlab. Therefore, we choose to save in ASCII format so we can read it on Matlab directly. This has the disadvantage that the conversion takes some time (at about 4 seconds per file), but does not require for us to be there as all can be scheduled. Following the second option, the file conversion would not have been necessary – but programming a tool to read it would be a must.

We can find in the Annex I the Matlab script code created to read those ASCII files. If someone should again do something similar, it can save him some time. All of them share a format like the one in Figure 4.2. The first value is the number of the sample (0 is when trigger condition happens). In the second column there is the value of the idle task output. The third is the time slot

beginning notification. The last column is the time passed since the trigger condition happened.

As per processing, each file takes up 1.5Mbytes and for 800 of them that is 1.2GBytes which Matlab has to parse line at a time. Also, that takes a lot of the computer's RAM memory. Therefore, files were read and processed a few at a time (depending on computer's available RAM). The procedure was to save each field of a sample in a structure, and later apply the math to calculate how much time was the CPU without doing anything (in the idle thread).

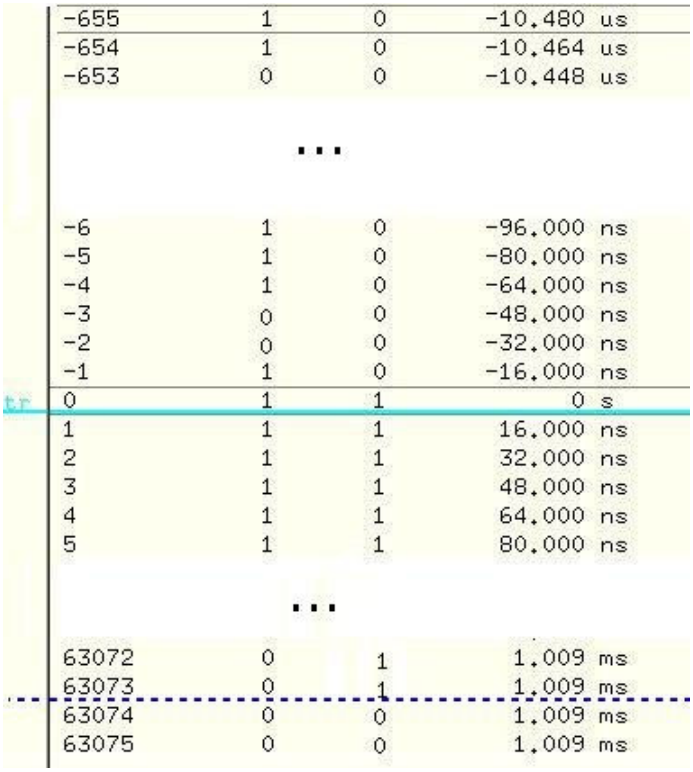

Fig. 4.2 Data captured in the logic analyser

The related math is simple: as we can see in Figure 4.3 and zoomed at the trigger at Figure 4.4, we can identify a time slot - 1 millisecond in the figure - and therefore, obtain the idle task output changes that are produced in there. If we then measure the minimum time that it has been without changing its value – in this example these white hole - it remains and check if this happens most of the time, then we know the time that lasts each loop of the idle function (we know that there isn't anything which executes with a period smaller than 10 microseconds, so most of the idle time is spend alone). If we then subtract this minimum value in each variation, we obtain how much time it has been doing something else in that slot.

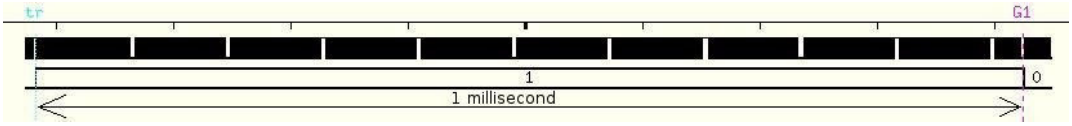

Fig. 4.3 Captured slot with a period of 1 millisecond. Done with while receiving large amounts of data, hence the white holes where the CPU is busy. Above: fast idle task output changes. Below: one time slot indication

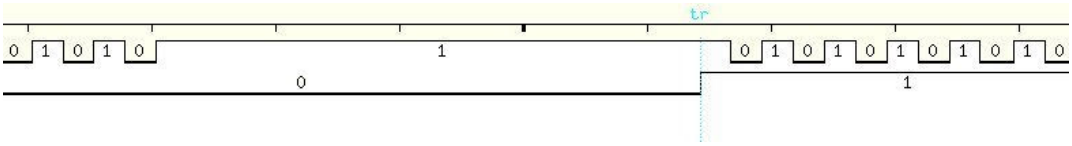

Fig. 4.4 Detail a slot start. Above: fast idle task output changes (with a CPU busy region). Below: time slot variation.

Finally and returning to the measurement, there is another tool to be mentioned: the computer which communicates with the DSP via Ethernet. We firstly wrote a simple read-write socket program in Linux that allowed us to do the firsts tests. However, after looking for documentation about best practices to test Ethernet interfaces, we found two programs which do this task. They are *iperf* and *nuttcp*. Their usage is straightforward after reading the man page. We used them in order to transfer data to the DSP, at the maximum possible speed.

As just said; only computer-to-DSP transfers were tested. This is because as we are using a dedicated Ethernet point-to-point network, it is the same to receive and to send. As there aren't TCP retransmissions all send-related buffer aspects do not apply - after all there is nothing else in the network; and retransmissions would indicate faulty DSP behaviour-. This kind of network will be rather common in our applications, since at design time we know how much data goes thought the network (so we know in advance what it has to support). Therefore, we decided to proceed with; and put more effort in the middleware.

Once we know the measurement related tools and how they are used, it's presented the procedure by itself in a simple way:

1. Connect the DSK board to Code Composer Studio. This will allow us to load the program as well as know if the TCP/IP Stack has any warnings (like timeouts or packet loss), as it gets printed in it.

2. Connect the DSK board with a computer by Ethernet.

3. Connect the data acquisition inputs of the logic analyser to the output pins of the board. Note that one ground connection must be also done, and that is much advisable to power off the DSP while doing this.

4. Configure the logic analyzer for the time slot that will be used.

5. Run the DSP and PC programs & start sampling.
6. Convert the captures to text files.
7. Matlab processing.
8. Repeat with different time-slot values.
9. Analysis of the data.

4.2.2. Results analysis: throughput vs. time slot

Because of the time-slotted design of our application, we are retrieving data from the stack once every time slot. This shouldn't affect the throughput of the interface if we assure that the buffers are big enough. However, as we can see in Figure 4.5 it does not happen so. We found that the NDK POSIX socket receive function returns as much as 7300 bytes of data, but never more. Trying with bigger buffers, function optional flags, different sending packet sizes, TCP window sizes and even changing the program that sends from the computer were useless efforts.

With this experience and previously ones, we can say that the Texas Instruments NDK has not been thought to carry out Software Radio-like characteristics. It does give importance to typical blocking sockets, but not that much on non-blocking. As for now, we keep using it; having the option of reading multiple times in each time slot (like when sending) if we need more throughput. This reminds us too that, like its name says, it's a development kit for evaluating applications. Therefore, it's not their intention to distribute a TCP/IP Stack to be used in final designs. Anyway, this was an unexpected result.

For the record, starting with a time slot of 0.6 milliseconds the stack starts to work at 83Mbps. Quite good bearing in mind that we are using TCP on a 100Mbps Ethernet network. From there, it starts to decay slowly. At a time slot of 1 millisecond we have 58Mpbs and at 10 ms, 5.8Mbps. As is of our interest to have a rather big slot in order to have more time to execute objects code, we should check which network speed we will have, and in case of being to slow modify the Hardware Abstraction Layer to receive from the stack more than one time.

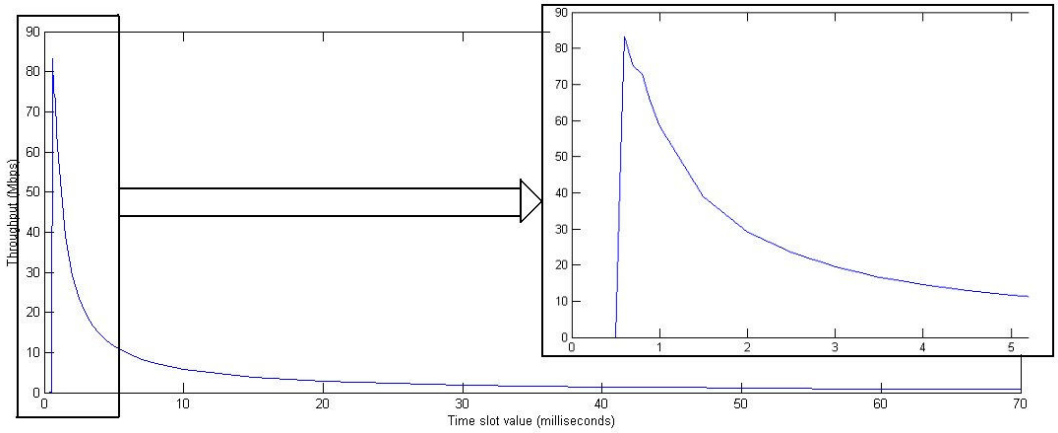

Fig. 4.5 Network throughput for various values of time slot. Optimized code used, one socket receive call each slot.

4.2.3. Results analysis: used CPU vs. time slot

There are two parameters, apart from the time slot value, that are relevant here. They are if we set up optimization or not when compiling and if we are receiving from Ethernet or not. For the first one, a simple comparison is sufficient as the code does not change too much when changing the time slot value.

Used optimization options: Opt Level (-o3), Program Level Opt. (-pm -op0).

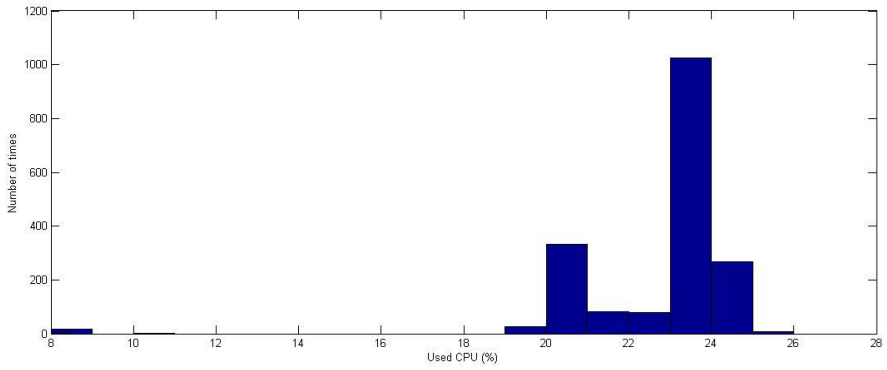

Fig. 4.6 Percentage of CPU used in 1840 time slots. Code not optimized.

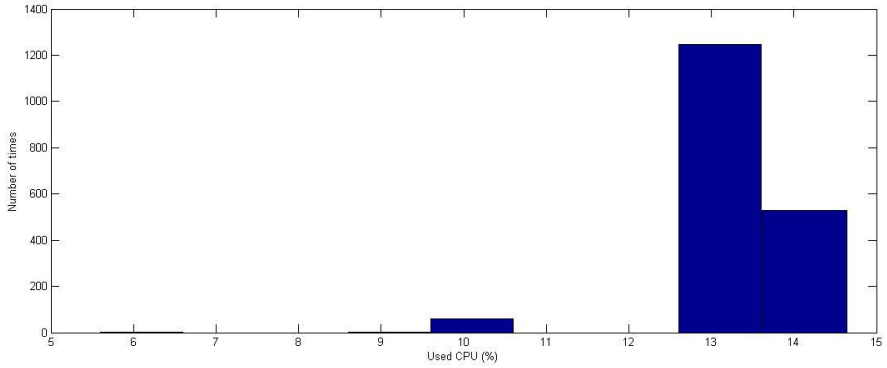

Fig. 4.7 Percentage of CPU used in 1840 time slots. Code optimized.

In Figures 4.6 and 4.7, we have the histograms of the percentage of used CPU while receiving data for a non-optimized and an optimized version of the same code respectively. A 1 millisecond time slot was used. As we can see, the optimized version takes about half the CPU than the other version. This happens since the DSP platform has a characteristic platform which the compiler is adapted to. Once optimization has been enabled, the compiler tries to parallelizate the code as much as possible so high optimization marks are achieved. We can also appreciate that the optimized version does stay in a shorter range of values, also approximately half of the non-optimized version. This indicates that the optimization has been proportional in the whole code. Therefore, it is highly recommended that when debugging is not necessary, an optimized compilation is done.

The remaining measures have been done with an optimized version. Also, as we previously mentioned, 500 time slots captures are statistically sufficient and therefore the results correct.

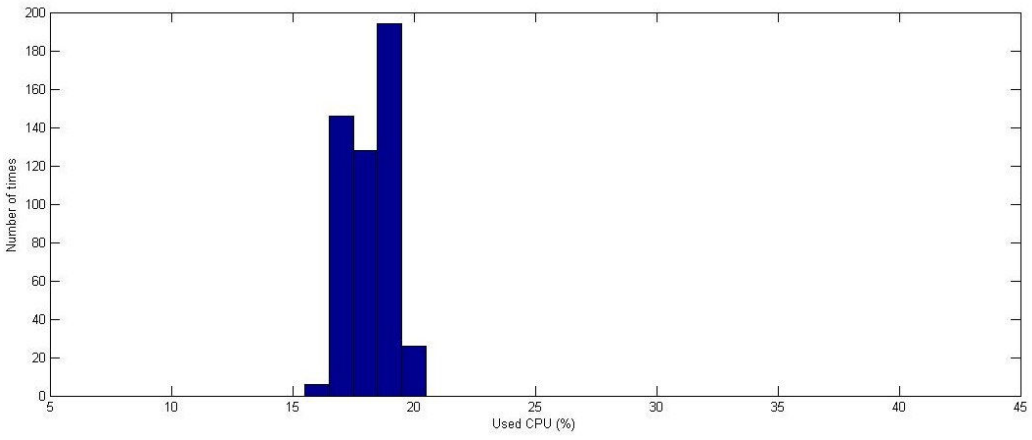

Fig. 4.8 Used CPU, 500 captures, 0.6 milliseconds time slot, receiving.

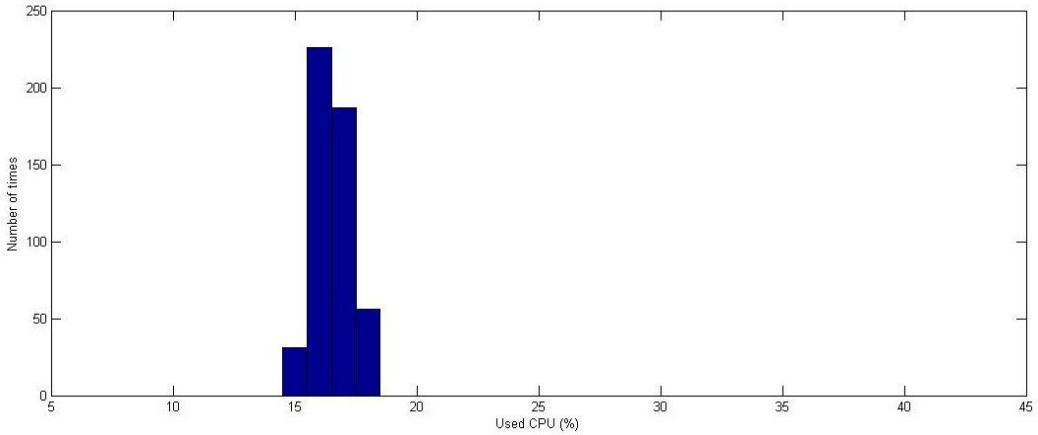

Fig. 4.9 Used CPU, 500 captures, 0.7 milliseconds time slot, receiving.

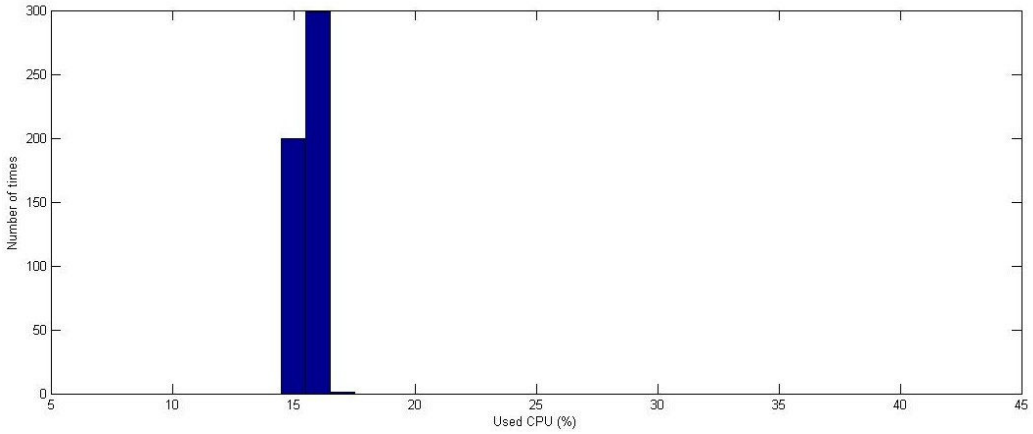

Fig. 4.10 Used CPU, 500 captures, 0.8 milliseconds time slot, receiving.

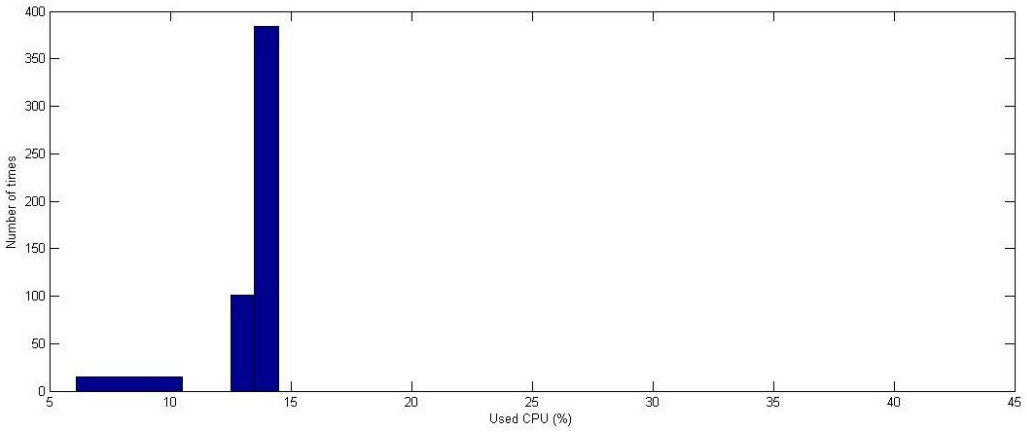

Fig. 4.11 Used CPU, 500 captures, 1 millisecond time slot, receiving.

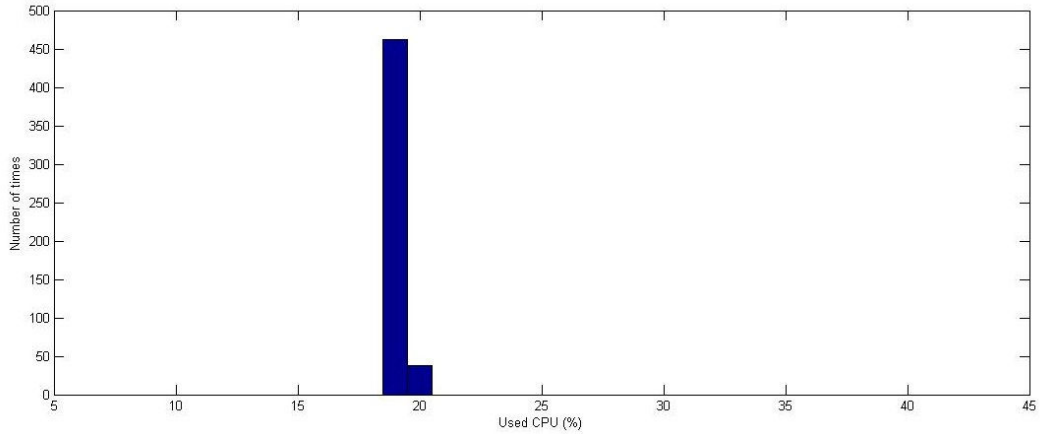

Fig. 4.12 Used CPU, 500 captures, 1.5 milliseconds time slot, receiving.

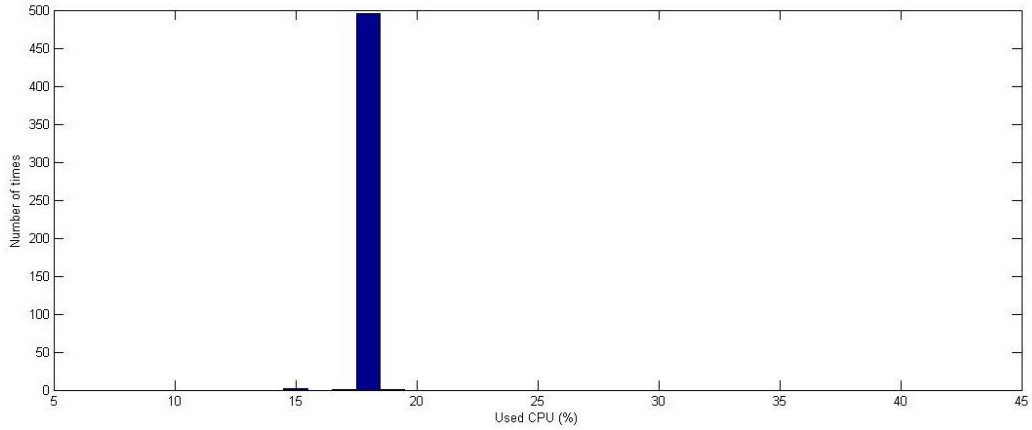

Fig. 4.13 Used CPU, 500 captures, 2 milliseconds time slot, receiving.

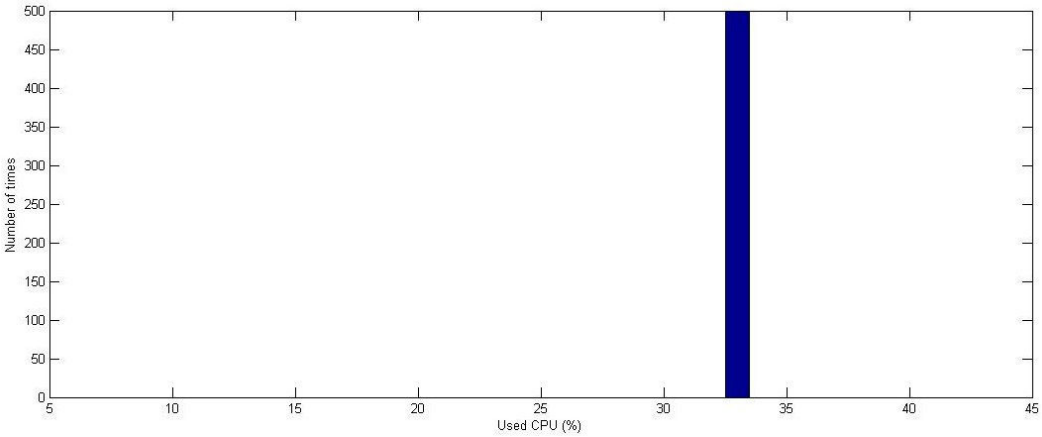

Fig. 4.14 Used CPU, 500 captures, 5 milliseconds time slot, receiving.

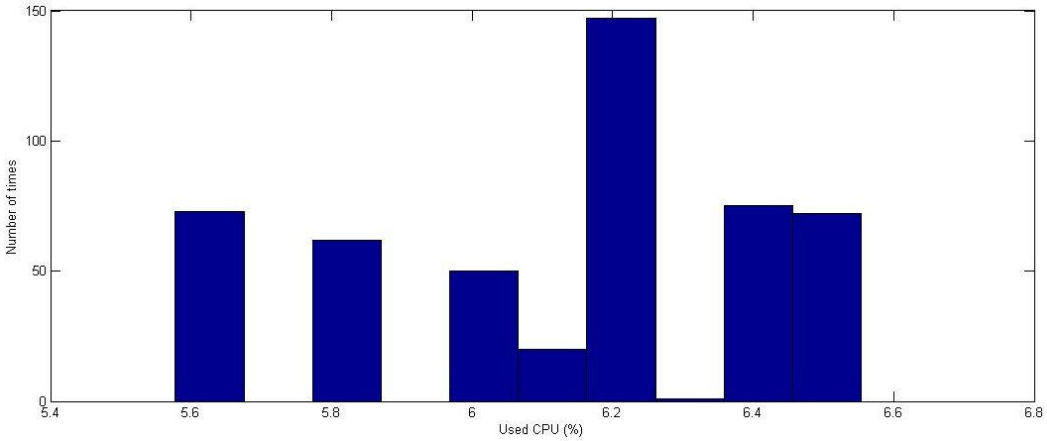

Fig. 4.15 Used CPU, 500 captures, 0.6 milliseconds time slot, not receiving.

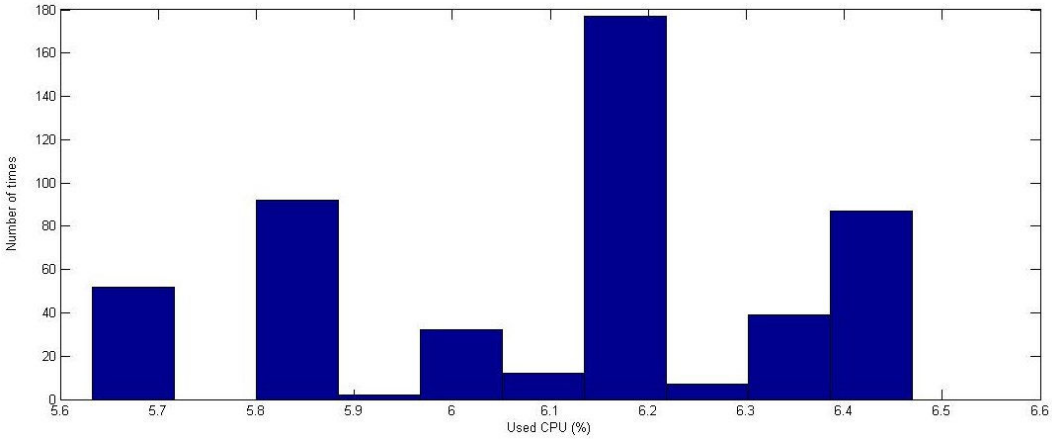

Fig. 4.16 Used CPU, 500 captures, 0.7 milliseconds time slot, not receiving.

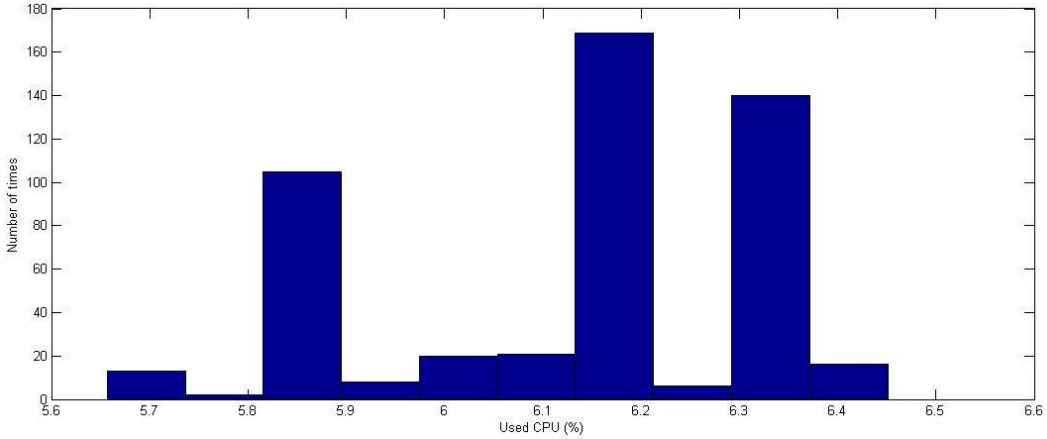

Fig. 4.17 Used CPU, 500 captures, 0.8 milliseconds time slot, not receiving.

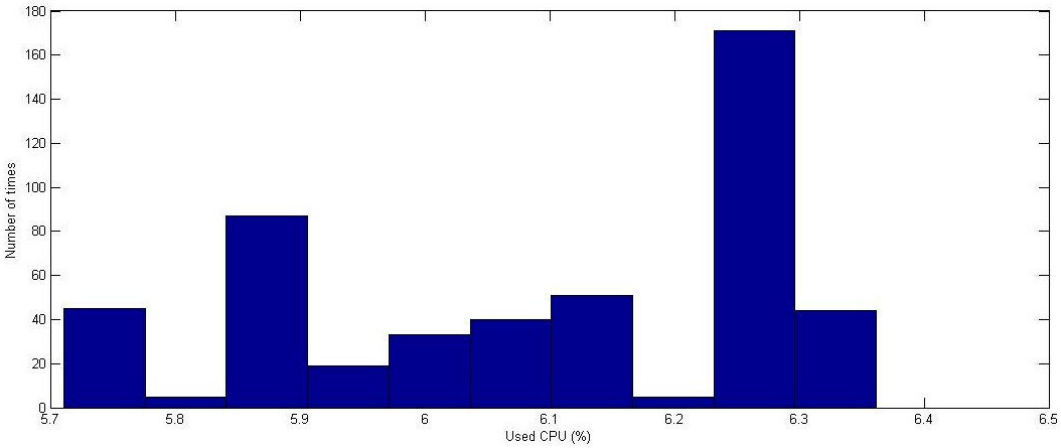

Fig. 4.18 Used CPU, 500 captures, 1 millisecond time slot, not receiving.

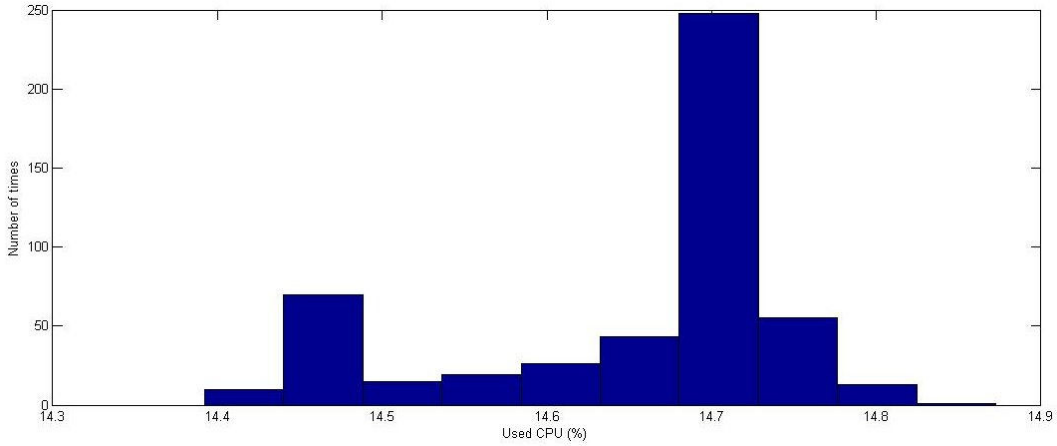

Fig. 4.19 Used CPU, 500 captures, 1.5 milliseconds time slot, not receiving.

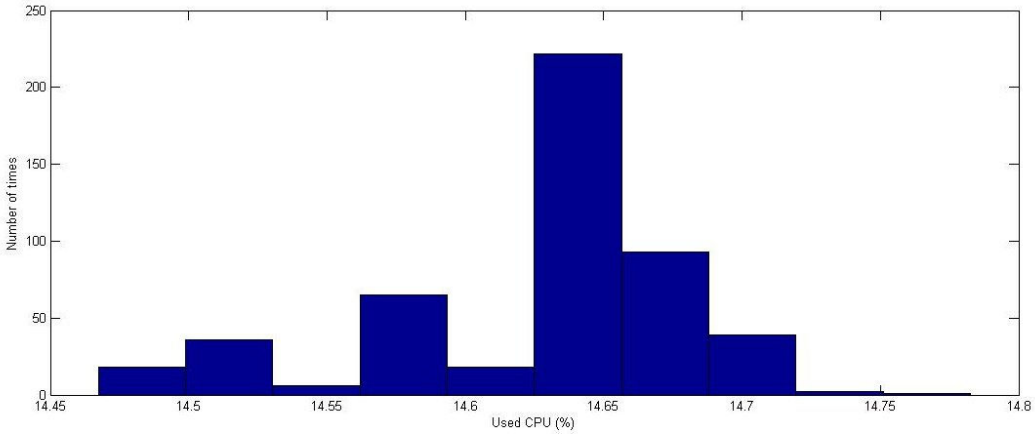

Fig. 4.20 Used CPU, 500 captures, 2 milliseconds time slot, not receiving.

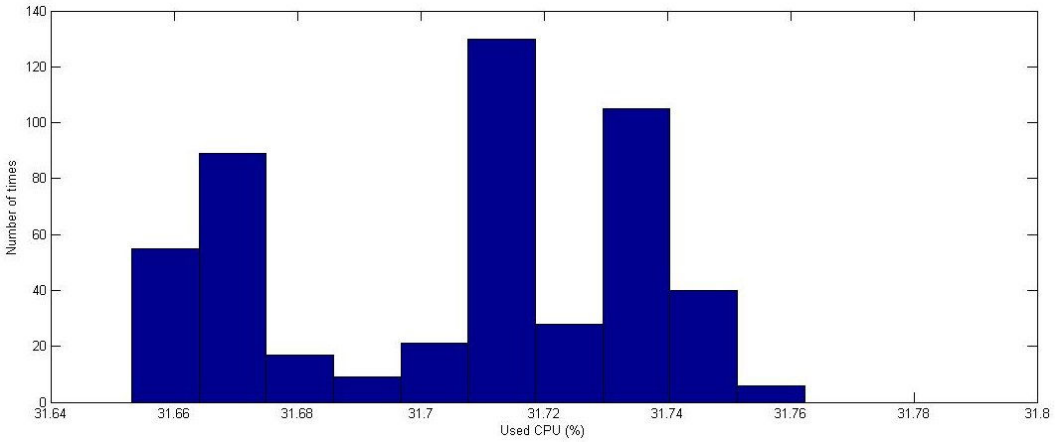

Fig. 4.21 Used CPU, 500 captures, 5 milliseconds time slot, not receiving.

When analyzing the used CPU in a time slot; is logical to think that as bigger gets the slot as lower will be the used CPU. We have a clear example between Figures 8 to 11. It starts at about 18% and ends up in 14%. At the same time, the range of values in each slot gets lower too. This happens because as the slot gets bigger, the ratio time slot divided by the variation of the time it takes to execute the code gets bigger too because the variation does not get modified and therefore the resulted percentage of used CPU gets more delimited. We can say that this is due to the fact that it's using the network because if we look at the corresponding measures without transfers (Figures 15-18) the variation is always the same, less than 1%.

A not anticipated result is what happens in Figures 11-14 and 18-21. In both of them the CPU usage gets bigger as it does the time slot. The only possible explanation to this unexpected behaviour is that the NDK might be blocking when in the non-transferring measure; because each time slot it tries to accept new incoming connections. In the case of when there is transmission, something wrong has to be in the TCP buffers, since if we put them as big as we want, we always obtain a maximum data size in each receive call. Therefore, as we call the receive function more slowly and it does not returns us more data, it has to reduce the TCP window size in order to not overflow the buffers.

Looking at the Figures 15-18 we can see how much processing it takes a basic system. The set of DSP/BIOS + TCP/IP Stack + basic network read/write & management takes up about a 6.2% of the CPU.

We can also appreciate the fact that the TCP/IP Stack is only managing because on all non-transfer measures (Figures 15-21), the range of values taken is quite low (less than 1%) and we previously saw that it was due to it.

From all of this data we obtain two conclusions:

1. This stack is not yet too much stable. It has been designed to use blocking sockets (even all of the sample code use them), and are giving more importance to include application level protocols than to proportionate a basic and stable TCP/IP stack. Please refer to the Notes section for related information.

2. If we need to use slot values bigger than 1 millisecond, we will have to make the network thread to receive form it more than once. In fact, this is not a wild idea, since when ALOE is sending, it does so multiple times (one per each message in the queue). The network code should be then patched to perform receive until no data is available. This should resolve, apart from the slow throughput, the high CPU usage previously stated problem.

5. PORTING ALOE TO TI's TMS320 DSP's

In this chapter we port ALOE from x86 architecture to DSP's. As we have seen, what we have to do is modify the Hardware Abstraction Layer and use the same code of the daemons and objects as the available for Linux.

The basic daemons that are needed to run an object are the Frontend, the Software Load and Exec. The first one is mandatory in each processor. It bridges incoming control packets to the daemons they are addressed. Also it registers to the network at boot, as well as discover the surrounding processors. The second one deals with loading objects locally. It also creates the interfaces they need, as well as downloads needed objects. The last one, Exec, is in charge of controlling and watching the object execution state.

5.1. Hardware Abstraction Layer

This layer is implemented as a C file (where all the functions are defined) and a header, which is included by all daemons and objects. The last one allows them to access the HAL functions. The header is called Hardware API.

In Linux, each object and daemon is an executable. Each one has its own HAL library and shares a memory space where the management information of interfaces, objects and daemons is stored. In the DSP, as it works with absolute addressing, we need a relocation utility if we want to load various executables. In other words, as the addresses of the code and variables are set at compile time and within a physical range, we cannot assure that there won't be anything else in there when loading the program. We would have to reserve some specific space for each additional executable. As we don't have yet the relocating functions, we choose to create a shared library and port the basic daemons as a library too. This way, we can start with a basic executable where the DSP/BIOS + network threads are present together with the daemons and HAL library. In the future, this would allow saving memory because instead of multiples copies of this library, there is only one.

If we save the address of the Hardware API functions in a known reserved memory space, afterwards we can load executables which don't have the entire HAL library, but just a set of redirections to retrieve these stored values and call them. Therefore, there are two different implementations of the HAL in the DSP. The first one is used only once. In it there is the real code, and also a set of redirections to these functions from a reserved and fixed memory range. The second one just implements redirections. This one is used by executable elements. We call the first one HAL kernel library and the other just HAL library.

This decision requires some modifications in the existing code. As the HAL library is shared between processes, one must ensure that these functions are re-entrant. In other words, they can be called without risking other simultaneous calls integrity. To do so, we principally must not use global variables or objects.

Another problem to face is how to do a shared memory space. In order to do that, in the DSP/BIOS we have defined special little memory regions reserved for this. By using compiler instructions like *#PRAGMA DATA_SECTION()*, we indicate that a C structure or variable which is shared has to be stored in one of these regions. After, semaphores have to be created for these variables that get modified at run-time, so we keep the re-entrant propriety.

In the HAL kernel library we also set two C files that are used as configuration. In there the user creating an ALOE middleware for a specific target can specify which daemons to load at startup as well as which Ethernet connections must be made.

The structure of this library is based in the Hardware API, which are called by others, and a never-ending task that executes each time slot. This task is in charge of maintaining TCP connections (by opening new ones and listening to incoming), to listen for Hardware API requests of linking TCP connections with internal interfaces and finally to send and receive data by this linked external interfaces.

Another important thing to notice is the task priority levels. This is needed because we have two different types of tasks: ones that do execute every certain time and others that never end. As the network task and the stack do work from time to time, like with interruptions, and they must follow a certain period; they have the highest priority. This way we assure that when they need to run there would not be anything else in its way. We must point out that the stack must have a priority over the network task, since this last one calls the stack API and so tells the documentation. About the never ending tasks, they are objects and daemons. They will be running when the network task and the stack are done with its job. A DSP/BIOS round-robin scheduling will assure that each one of tem gets a fair computation time.

5.2. Hardware API Reference

In this section each function in the HAL library that is accessible to daemons and objects is explained. Also, DSP unique characteristics are also mentioned.

Declaration `int hwapi_init(void);`

Parameters

Returns

`1`	If successful
`0`	If error

Description Initializes Hardware API for an object or daemon.

This function must be called before calling any other function of this API.

Declaration `int hwapi_exit(void);`

Parameters

Returns

`>0`	If successful
`<0`	If error

Description Clears all HAL resources used by the daemon or object who calls the function.

Once this function is called, no more functions from this API can be called.

Declaration `int hwapi_myobjid(void);`

Parameters

Returns

`integer`	The identifier of the object calling this function.

Description This function is used by objects in order to know who they are.

Declaration `void hwapi_relinquish(int slots);`

Parameters

`slots`	Number of time slots to relinquish

Returns

Description Processes release their ownership of the CPU until next time slot after this function is called. The execution is suspended during *slots* time slots. After this amount of time slots have been elapsed, execution thread continues with the next line.

Declaration `void hwapi_relinquish_daemon(void);`

Parameters

Returns

Description Does the same as hwapi_relinquish, but for daemons. As they don't follow time slots, they are paused during certain period of the current slot.

Declaration `int hwapi_itf_create(int id, int size);`

Parameters

`size`	Maximum size in words for the packet to be written to the interface.

Returns

`!=0`	Id of the interface
`=0`	If error

Description Creates or allocates the resources needed to allow the interchanging of packets inside the Platform.

This call just prepares a side to side interface but can not be used yet because read and write sides must be assigned (see hwapi_itf_attach).

The returned Id will be later used when assigning interfaces.

In the DSP, it has a semaphore that assures that while creating the interface nobody overwrites it. Also, memory-pointer queues are created in order to send and receive the messages. Each interface gets a fixed number of messages which go form free to send or receive queue.

Declaration `int hwapi_itf_delete(int id);`

Parameters

`id`	Id of the interface to delete

Returns

`1`	If deleted
`0`	If error

Description Deletes the resources allocated or reserved.

Once this function has been called, the interface can not be used anymore.

It also deletes all the messages that the interface had, as well as the queues.

Declaration `int hwapi_itf_delete_own(char *obj_name);`

Parameters

`obj_name`	Name of the object

Returns

`1`	If deleted
`0`	If error

Description Given an object name, deletes all interfaces which it has been matched.

Declaration

```
int hwapi_itf_attach(int id, int mode);
```

Parameters

`id`	Id of the interface to delete
`mode`	1 read side 2 write side

Returns

`>0`	File descriptor for the interface for sending or receiving packets
`-1`	Interface not available
`-2`	Error attaching interface

Description

Creates a file descriptor to use the interface. This function should be called 'after' creating the interface and 'before' using it.

It must be attached in FLOW_READ_ONLY (for reading) or FLOW_WRITE_ONLY mode (for writting). A file descriptor will be returned which can be used to read/write packets.

This function also uses a semaphore to assure that nobody modifies shared variables while modifying them.

Declaration

```
int hwapi_itf_unattach(int fd, int mode);
```

Parameters

`fd`	File descriptor
`mode`	1 read side 2 write side

Returns

`>0`	If successful
`<0`	If error

Description

Un-attaches a file descriptor from the (previously attached) interface in mode 'mode' .

Declaration `int hwapi_itf_snd(int fd, int *buffer, int size);`

Parameters

`fd`	File descriptor of the interface
`buffer`	Pointer to data to send
`size`	Number of words (32 bits) to send

Returns

`>0`	Number of words send
`=0`	Data NOT send, full queue
`<0`	If error

Description Writes the first *size* words located at **buffer* to the interface assigned to file descriptor *fd*.

Function is none blocking thus, if packet can not be sent immediately because of full buffer or any other reason function returns 0 and packet is not send.

This function uses DSP/BIOS queue objects and buffer data memory copy to the message.

Declaration `int hwapi_itf_rcv(int fd, int *buffer, int size);`

Parameters

`fd`	File descriptor of the interface
`buffer`	Pointer to save data
`size`	Length of user buffer in words (32 bit)

Returns

`>0`	Number of words received
`=0`	No data is available
`<0`	If error

Description Receive the first packet available at interface assigned to file descriptor *fd*. Data is saved to the buffer pointed by *buffer*.

Packets are received entirely although user must provide the length of the buffer to avoid overflow, so, if packet is bigger a truncation of the data will occur.

This function is none blocking thus, if no packet is available when function is called, 0 will be returned.

This function uses DSP/BIOS queue objects and message-to-buffer data copy.

Declaration

```
int hwapi_itf_link_phy(int itf_id, int phy_itf_id, int mode);
```

Parameters

`itf_id`	Local interface Id
`phy_itf_id`	External Physical Interface Id
`mode`	FLOW_READ_ONLY or FLOW_WRITE_ONLY constant

Returns

`>0`	Successfully linked
`<0`	If error

Description Links a physical interface (external) to an internal packet oriented interface. After the link is successfully realized, all packets written to the internal interface will be bridged (by the network task) towards the physical interface. Conversely, all packets received to the phy. itf. will be bridged to the internal interface.

Interface must be created before calling this function. Then, in order to use it, it must be attached

This function uses DSP/BIOS mailboxes. Therefore, the link is not created immediately but it is surely done on the next time slot.

Declaration

```
int hwapi_itf_match(int itf_id, char *w_obj, char *w_itf, char *r_obj, char *r_itf);
```

Parameters

`itf_id`	Local interface Id
`w_obj`	Write-end object name
`w_itf`	Write-end interface name
`r_obj`	Read-end object name
`r_itf`	Read-end interface name

Returns

`1`	If successful
`0`	If error

Description Function used to assign names to the object and interface write and read sides of a local interface.

Declaration `int hwapi_itf_find(char *obj_name, char *itf_name, int mode);`

Parameters

`obj_name`	Object name
`itf_name`	Interface name
`mode`	FLOW_READ_ONLY or FLOW_WRITE_ONLY constant

Returns

`>0`	Id of the local interface
`0`	If error

Description Function used to search for the Id of an interface, given its name, attached object name and mode.

Declaration `int hwapi_hwinfo_xitf(struct hwapi_xitf_i *xitf, int max_itf);`

Parameters

`hwapi_xitf_i`	Pointer to information structure
`max_itf`	Maximum number of interfaces

Returns

`>=0`	Number of interfaces read
`0`	If error

Description Obtains information of the Physical External interfaces. It is read from the shared memory structure.

User must provide the maximum amount of external interfaces he wants to read. It depends on the length of the xitf buffer (number of elements).

Declaration `void hwapi_hwinfo_cpu(struct hwapi_cpu_i *info);`

Parameters

`info`	Pointer to an structure of type *struct hwapi_cpu_i*

Returns

Description Function used to gather processing resource (CPU) information.

Data is written to the struct pointed by info.

Declaration `void hwapi_hwinfo_setid (int pe_id);`

Parameters

`pe_id`	New processor id

Returns

Description Overwrites the current processor id for the given one.

Declaration `Int hwapi_proc_create(char *pdata, struct hwapi_proc_launch *pinfo);`

Parameters

`Pdata`	Pointer to program data
`Pinfo`	Pointer to process information structure

Returns

`>0`	Pid of the process
`<0`	If error

Description This function creates a process and launches it. As the process resides in user memory (pdata buffer), it must be written to memory before launching it. When deleting a process, its code is also erased.

DSP/BIOS task creation utilities are used in order to start the process. Also, a special function parses the executable format (COFF if compiled by Code Composer Studio).

Declaration `int hwapi_proc_remove(int obj_id);`

Parameters

`obj_id`	Id of the object to remove

Returns

`1`	If successful
`0`	If error

Description Removes a process from the system and de-allocates all resources it was using.

Declaration `int hwapi_proc_status_get(void);`

Parameters

Returns

`integer`	The status to run as

Description Obtains the status the object must run in the current timestamp.

This function is called by Services API during the Status() call, in every object execution's loop.

During an status change procedure, the new status won't be returned after the timestamp is same or greater than the one passed as the third parameter in the hwapi_proc_status_new() function.

Declaration `int hwapi_proc_status_new(int obj_id, int next_status, int next_tstamp);`

Parameters

`obj_id`	Object identifier
`next_status`	Next status to setup
`next_tstamp`	Tstamp where it is valid

Returns

`1`	If successful
`0`	Can't change
`-1`	If error

Description Function used to change the status of an object.

Declaration `int hwapi_proc_status_ack(int obj_id);`

Parameters

`obj_id`	Object identifier

Returns

`1`	If already changed
`0`	If yet not changed

Description Confirm a successful status change.

Declaration `int hwapi_proc_info(int obj_id, struct hwapi_proc_i *pinfo);`

Parameters

`obj_id`	If of the object assigned to the process
`pinfo`	Pointer to an structure of type *struct hwapi_proc_i*

Returns

`1`	Successful
`0`	If error (not found)

Description Function used to gather information about current process status, future status, CPU timing and other related information

Declaration `int hwapi_proc_myinfo(struct hwapi_proc_i *pinfo);`

Parameters

`pinfo`	Pointer to an structure of type *struct hwapi_proc_i*

Returns

`1`	If successful
`0`	If error

Description Fills hwapi_proc_i structure with the caller's info.

Declaration `int hwapi_proc_list(struct hwapi_proc_i *pinfo, int nelems);`

Parameters

`pinfo`	Pointer to an structure of type *struct hwapi_proc_i*
`nelems`	Maximum number of processes to get

Returns

`integer`	Number of processes retrieved

Description Gets the processes list.

Declaration `void get_time(time_t *tdata);`

Parameters

`tdata`	Pointer to a Time Structure

Returns

Description Fills the Time Structure with the current HAL Library time. The structure consists in two unsigned integers. First one is the amount of seconds and the second one is the amount of microseconds.

The DSP/BIOS updates this time each 100 microseconds.

Declaration `void get_time_interval(time_t *tdata);`

Parameters

`tdata`	Pointer to a Time Structure

Returns

Description The first element stores the interval between the second element (start time) and the third element (end time).

Declaration `int get_tstamp(void);`

Parameters

Returns

`unsigned int`	Time Stamp Value

Description Returns the current HAL Library time stamp value. Value is an unsigned integer.

Declaration `TSK_Handle getppid(void);`

Parameters

Returns `TSK_Handle` Identifier of a task

Description Returns the identifier of the network task.

Declaration `TSK_Handle getpid(void);`

Parameters

Returns `TSK_Handle` Identifier of a task

Description Returns the identifier of the task who has called this function.

And finally, in order to change the code to another DSP the following should be done:

1. Configure DSP/BIOS
2. If memory map is modified, update *hwapi_dsp.h*'s `SHARED_MEMORY_POSITION` value
3. Update ALOE configuration as desired
4. In *mem_net_bridge.h* modify the `CLK_GRANULARITY` as configured in DSP/BIOS, and `TIME_SLOT` to match the desired one.

6. FUTURE WORK

There are several things to still do on the DSP in the scope of the FlexNets project. Hopefully, the end of this essay doesn't suppose any problem.

The following is a list of features that should be implemented:

- Rapid IO: for fast, low delay communications with DSPs and FPGAs.
- Relocation functionality: in order to be able to load any executable, regardless of its memory positions.
- Remote message sending: in order to have a unique machine which receives all messages of the PE's. This would facilitate ALOE Platform debugging.
- Optimize ALOE and objects with CCS provided comments-on-compilation.
- Add big endian support: necessary when the PlayStation 3 gets added as a new platform for executing ALOE.
- Port the synchronization daemon: in order to have time control between all existing PE.
- Upgrade NDK to a latest version and if necessary ask for support on non-blocking sockets.
- Prepare DSP's ALOE port to be able to compile in Linux.
- Use the Direct Memory Access (DMA) controller of the DSP: to make all the memory copy operations done outside the processor.

7. ACRONYMS

- ALOE (Abstract Layer & Operating Environment)

Middleware form FlexNets for Software Radio applications.

- API (Application programming interface)

Piece of software destined to be used by someone else.

- CCS (Code Composer Studio)

- CPU (Central Processing Unit)

- DSK (Development Starter Kit)

It's a pack that contains a DSP board and a CCS license.

- DSP (Digital Signal Processor)

- FlexNets

Open source initiative, both hardware and software based, interested to strength the collaboration among different institutions in developing a common framework to explore the introduction of flexibility in the different layers of the wireless communication systems.

- FPGA (Field Programmable Gate Array)

- GPP (General Purpose Processor)

- HAL (Hardware Abstraction Layer)

- IDE (Integrated Development Environment)

- NDK (Network Development Kit)

- PE (Processing Element)

- QoS (Quality of Service)

- RAT (Radio Access Technology)

- RTOS (Real Time Operating System)

- TI (Texas Instruments)

- VLIW (Very Long Instruction Word)

8. FINAL NOTES

The software of this project is not available with this essay. It is part of the ALOE from FlexNets. Therefore, if interested one should contact the Tutor of this work (at antoni@tsc.upc.edu).

Also, ALOE is under continuously under development, so in the case of wanting the code it is highly recommended to ask for acces to our Subversion code repository.

A note on the NDK TCP/IP stack should also be done: as of the writing of this document Texas Instruments has released a new version (1.94.01). It should be tested to see if our issues have been resolved, or else contact them since they might already know faulty issues.

Finally, and due to the lack of documentation on programming DSPs, we leave into record two websites which gather most of the help on might find for free in the Internet. These are [6] and [7]. Note also that when installing CCS, one might find a lot more of TI's documentation in *%CCS Install Directory%/docs/PDF*.

9. BIBLIOGRAPHY

1. J. Mitola, “The software radio architecture”, *IEEE Commun. Mag.*, vol. 33, no. 5, pp. 26–38, May 1995.

2. Texas Instruments, “TMS320C6000 DSP/BIOS 5.32 Application Programming Interface (API) Reference Guide”, literature number SPRU403O, September 2007.

3. Texas Instruments, “TMS320 DSP/BIOS User's Guide”, literature number SPRU423F, November 2004.

4. Spectrum Digital, “TMS320C6455 DSK Technical Reference”, revision 6, September 2006.

5. Texas Instruments, “TMS320C6000 Network Developer's Kit Programmer's Reference Guide”, literature number SPRU524E, May 2008.

6. http://www.dsprelated.com/ ; last checked on 15 January 2009.

7. http://tiexpressdsp.com/wiki/index.php?title=Main_Page ; last checked on 15 January 2009.

10. ANNEX I: Matlab Code

```
function [cpu_working, slot_duration] = LoadAnalizer( file_path,
pos_start,pos_end, slot_time )

%% Prepare things
files_array = repmat(struct('m_fast', {repmat(uint8(0),65536,1)},  'm_slot',
{repmat(uint8(0),65536,1)},  'm_num', {repmat(int32(0),65536,1)},  'time',
{repmat(int32(0),65536,1)},  'units', {'xx'}), (pos_end-pos_start+1), 1);
num_captura=0;

for i=pos_start:pos_end

    %% Open file
    file = fopen(strcat(file_path,'.',  num2str(i) ), 'r');
    if (file < 0)
        error('Could not open %s for input.',strcat(file_path, '.',
num2str(i)));
    else
        num_captura=num_captura+1;
    end

    %% Read useless 14 lines
    fseek(file, 0, 'bof');
    for j=1:14
        fgetl(file);
    end

    %% Start reading data
    buffer = textscan(file,'%u8 %u8 %d %d %s ');
    files_array(num_captura).m_fast = buffer{1};
    files_array(num_captura).m_slot = buffer{2};
    files_array(num_captura).m_num = buffer{3};
    files_array(num_captura).time = buffer{4};
    files_array(num_captura).units = buffer{5}(1);

    fclose(file);
end

clear buffer file;
files_array = files_array(1:num_captura);
tmp = length(files_array);
slot_duration = zeros(tmp, 1);
cpu_working = zeros(tmp, 1);
idle_array = repmat(int16(0), 65536, 1);

%% Calculate cpu load for each capture
num_captura = 1;
for i=1:tmp

        % Make sure it's a valid sample
        if (files_array(i).m_num(1) >= 0) || (length(files_array(i).m_fast) ~=
65536) || ~((strcmp(files_array(i).units, 'ps')) ||
(strcmp(files_array(i).units, 'ns')))
            fprintf(1, '\nFound invalid capture file\n');
            continue;
        end

        % Find where slot should start
        slot_start_position =  find(files_array(i).m_num == 0, 1, 'first');
        slot_value = files_array(i).m_slot(slot_start_position);

        % Find where the slot should end
        slot_end_position = find(files_array(i).m_slot(slot_start_position :
end) ~= slot_value, 1, 'first') + slot_start_position - 2;
```

```
        % Find sample period
        sample_rate = files_array(i).time(slot_start_position+1);
        if strcmp(files_array(i).units, 'ps')
            sample_rate = sample_rate/1000;
        end

        % Calculate the slot time
        slot_duration(num_captura) = (slot_end_position - slot_start_position
+ 1)*sample_rate;

        % Make sure a full slot has been sampled
        while 1
            if(slot_duration(num_captura) < slot_time*800)
               if(slot_start_position > 32768)
                    fprintf(1, '\nSlot has not been sampled completely,
next!\n');
                    slot_start_position = 0;
                    break;
               end
               slot_start_position = slot_end_position + 1;
               slot_value = ~slot_value;
               slot_end_position =
find(files_array(i).m_slot(slot_start_position : end) ~= slot_value, 1,
'first') + slot_start_position -2;
               if isempty(slot_end_position)
                   fprintf(1,'\nCould not find an end position for the slot');
                   slot_start_position = 0;
                   break;
               end
               slot_duration(num_captura) = (slot_end_position -
slot_start_position + 1)*sample_rate;
            else
                break;
            end
        end
        if (~slot_start_position)
           continue;
        end

        % Calculate how much samples are IDLE and how much not.
        idle_pos = slot_start_position;
        for ii=1:65536
            idle_value = files_array(i).m_fast(idle_pos);
            idle_next = find(files_array(i).m_fast(idle_pos : end) ~=
idle_value, 1, 'first') + idle_pos - 1;
            if isempty(idle_next)
                idle_next = slot_end_position;
            end
            idle_array(ii) = idle_next - idle_pos;
            idle_pos = idle_next;
            if(idle_next >= slot_end_position)
                idle_array = idle_array(1:ii);
                break;
            end
        end

        idle_array = idle_array - min(idle_array);
        cpu_working(num_captura) = sum(idle_array)*sample_rate;

        num_captura = num_captura + 1;
end

slot_duration(1:(num_captura-1));
cpu_working(1:(num_captura-1));

fprintf(1, '\nTime is in nanoseconds.\nDone.\n');

end
```