Escola Tècnica Superior
d'Enginyeria de Telecomunicació de Barcelona

Final Project

“Analog Signal Adquisition and FFT Application for a Linux Embeded System Lab”

Author : Carlos J. Montes O'Connor
Director : Dr. Manuel Dominguez Pumar

PROJECT SUMMARY

The project goal is to design a platform based on an existing HW/SW embeded Linux system containing a FPGA, within an academic environment. To accomplish that objective, a specific application for signal Adquisition and Fourier Análisis System will be designed and implemented.

The project scope includes developing an Analog I/O Board with interface with the existing FPGA within the Embeded System Board, design and implementation the needed firmware to make the signal adquisition and the FFT signal análisys , the Linux kernel device driver , the user space tcp server program and finally a JAVA Client application to demostrate the final system performance .

As a result, the project will define a general design procedure for developing digital systems and kernel driver for linux embeded systems.

INDEX

1 INTRODUCTION

The project goal is to design a platform based on an existing HW/SW embeded system containing a FPGA, within an academic environment .

To accomplish this objective and show that has been achived, a specific application will be developed. In particular, for this purpose has been selected to make an application for Signal Adquisition and Fourier Análisis in the audio frequency range.

As a result, the project will define a general design procedure to develop digital systems and kernel drivers for linux embeded systems,

The project results could be used in an undergraduate optative subject oriented to digital systems design for embeded systems, filling the gap between both Digital Systems and Embeded Linux programming subjects.

The Operative System selected is Linux, because is open source code is very suitable for real-time applications and the most used in scientific and engineering university environments .

The HW/SW Linux embedded System platform selected is the APF27 from ARMADEUS, because is a open source system, so we can use GNU license agreements.

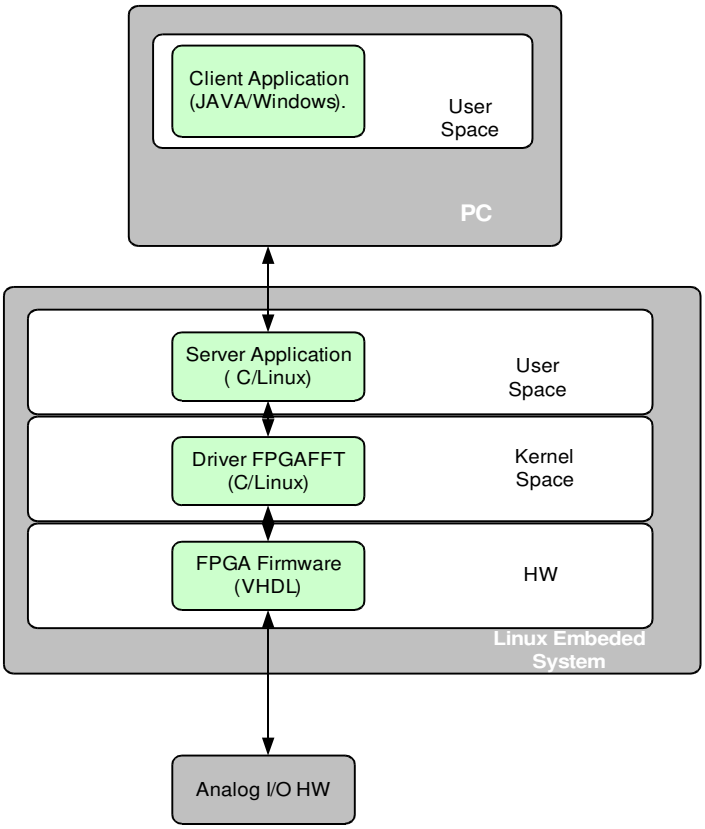
To meet special application requirements the project scope will also include design and implementation of a board including an Analog to Digital Converter with the right hardware interface for the specific APF27 board external interfaces .

It is important to note that the the main goal is to get a procedure covering the FPGA firmware and linux kernel driver , to properly interface a specific digital system with a linux embeded system.

So , the project scope can be broken down in the following parts:

- Design and implementation of an Analog I/O Board to interface with existing FPGA in APF27 board.
- Design and implemetation in HDL the needed firmware for interface with the APF27 system bus and make the specific application functionality (i.e. the signal adquisition and the FFT signal análisys).
- Interrupt driven Linux device driver dessign and implemetation, including specific functionality.
- User space application design and implemenation (i.e. tcp server)
- Demostration Program (i.e. JAVA Client application) to demostrate the final system performance , displaying results.

Next diagram shows the different project task , including the development environment to do the Job. The use of C/C++ in Linux kernel programming is mandatory. In user space is also the easiest way to do the job (because the ioctl call to device file). The JAVA will be use to program the Client application because its portability and high level GUI programming features that make easy to do a good MMI using standard class libraries .



The general design rules followed in this project :

- Use standard and if possible the most extended technology in academic enviroments.
- Use stablished API's and libraries
- Use available opensource code and IP-Cores as initial point for development.

In regard with FPGA firmare, following that rules the standard ANSI/IEEE VHDL laguaje will be used also because is the most extended HDL in academic enviroment.

In following sections the different design options are discused to explain the reasons why decisions where made. There is also a detailed design an implementation explanation.

2 HW/SW PLATFORM

2.1 Initial Considerations and Platform Selection

As it was pointed out in the introduction the Linux Operative System will be selected, because is open source , is very suitable for real-time applications and is the most used in scientific and engineering university environments .

So , the Embeded System Board processor will execute the Linux O.S but should also have an specialized hardware in order to be able to sample signals , at least at 100 Khz frecuency and eventually perform the FFT Transform , while running other task like dataserver . All this task processing is far out of a Processor capability.

So two options were considered ,:

- DSP
- FPGA

There is reasons for chosing the two options, but in the FPGA has been selected for the following reasons:

- FPGA can sample at a higher rates , beacause DSP may strugle to capture process and output data without any loss. This is due to the many shared resources , buses and even the core with the processor.
- DSP are instruction based not clock based. Typically, three to four instructions are required for any mathematical operations on a single sample. The data must be captured at the input , then fordwarded to the processing core , cycled througth that core for each operation and then send it to the aoutput,. In contrast FPGA is clock based , so every clock cycle has the potential ability to perform a mathematical operation on the incoming data stream.
- IP cores are available in FPGAs for many applications. Often it is simpler to break a high-level system block diagram into FPGA modules and IP cores than it is to program into C code for DSP implementation.

On the other hand the new multicores architectures of DSP can achieve very high sample rates and could also easily meet the project requirimentes.

But finally , the FPGA option was taken because the academic value of developing a Linux kernel driver for using in an embedded platform with FPGA firmware.

So to have an FPGA resource , was a requirement for the hardware platform to use.

There is several platforms for this purpose but ARMADEUS has been selected , because the use of Linux , and Open Software policy , so every information about software /harwdware is available and there is forums to help in case of issues , in addition Linux is also the most used OS in academic enviroment.

It also have a Xilinx Spantan 3A FPGA, A bout which there is plenty of documentation, information and free available IP Cores.

In this section the Armadeus APF27 Hardware and LINUX Software platform, will be described.

2.2 Hardware Platform Description

2.2.1 Armadeus APF27 Single Board Computer

The APF27 is a middle/high-end Single Board Computer targeted for low power applications, advanced GUI and extended connectivity. Here is a list of the main features:

- Processor: Freescale i.MX27 (ARM9 @ 400MHz)
- RAM: Mobile DDR. 64 to 256MB. 32 bits data bus. Default capacity will be either 64 or 128MB.
- Flash: Mobile NAND. 256MB, 16 bits data bus.
- Ethernet: onboard Physical (ready to use Ethernet 10/100Mbit link)
- USB: High speed USB OTG (OnTheGo) with onboard Physical (ready to use USB OTG link)
- USB: 1 High speed Host and 1 full speed USB Host ports (external PHY required)
- RS232: onboard Physical (RS232 compatible interface)
- FPGA: Xilinx Spartan 3A (50 to 400k gates, 200k default)
- Supplies: high end DC/DC converters and LDOs on board. Only one external supply of 3.3V required.
- Low power sleep mode (<10mW)
- Mechanical dimensions: ~60x45mm

All the i.MX27 peripherals (LCD, 2xSDIO, 3xSPI, 6xSerial, I2C, CSI, 3xUSB, keypad, PWM, etc...) and the FPGA signals can be accessed through two high density Hirose connectors .

In this project the APF27 SBC has been used connected to an APF27DEV Peropheral board.

For further information visit the armadeus webpage
http://www.armadeus.com/wiki/index.php?title=APF27

2.2.2 Armadeus APF27DEV board

The APF27Dev is a full featured development board dedicated to the APF27 single board computer. This board offers access to the whole functionalities of the APF27 and provides several additional features. Its price and its connectors make it ideal for rapid development of embedded applications.

Features:

- Input Power supply 5 to 16V dc (2.5A required if the whole functionalities are used) On board regulators 5V / 2,5A max high efficiency DC/DC converter. 3.3V / 1,5A max high efficiency DC/DC converter
- 1.8V/250mA LDO for audio and HDMI
- USB Host Controllers One high speed and one full speed USB 2.0 port
- Stereo Audio In/Out Controller Headset stereo audio out
- Touchscreen Controller Resistive touch panel
- CAN Controller CAN 2.0b . **Full version only**
- ADC 7 x 10 bits SPI ADC with internal or external reference. **Full version only**
- DAC 2 x 10bits I2C DAC with external external reference. **Full version only**
- HDMI HDMI 1.2a, up to 720p. **Full version only**
- RTC with backup battery **Full version only**
- Standard Connectors Two Hirose receptacles for the APF27
- Jack 2.5mm for power supply
- 2 x USB host (type A)
- RS232 DSub 9pts
- Ethernet (RJ45) with integrated isolation transformer and leds
- HDMI (**full version only**)
- dual 3,5mm stereo jack for audio in and out
- microSD
- Specific 2,54mm connectors Touchscreen
- LCD interface
- CAN (**full version only**)
- ADC/DAC (**full version only**)
- FPGA signals (**full version only**)
- iMX inputs/outputs
- User leds 2: One connected to the i.MX and the other to the FPGA

- User switches 2 : One connected to the i.MX and the other to the FPGA
- Reset Switch
- Jumper for boot mode selection
- Standby mode: tbdl

See the APF27DEV Datasheet for further information.

2.3 Software Platform

The armadeus APF27 Board has a bootloader called U-Boot whose object is to provide the user the ability to configure and diagnose the board.

Among the task suported by U-Boot are :

- Load a executable image (kernel, root file system or FPGA program) on RAM
- Move data between RAM memory zones
- Load FPGA and programming
- Flash Memory programming
- enviroment variables R/W access
- Boot

From U.Boot promt The Linux Operative System can be loaded over the network , via FTP , and boot. There are also some tools to customice linux kernel and busybox , Please refer to ANEX 1 for further details about setting up the linux development enviroment and available toolkit for kernel and busybox configuration.

In this project a module driver will be integrated into the kernel using this tools.

3 SYSTEM DESIGN

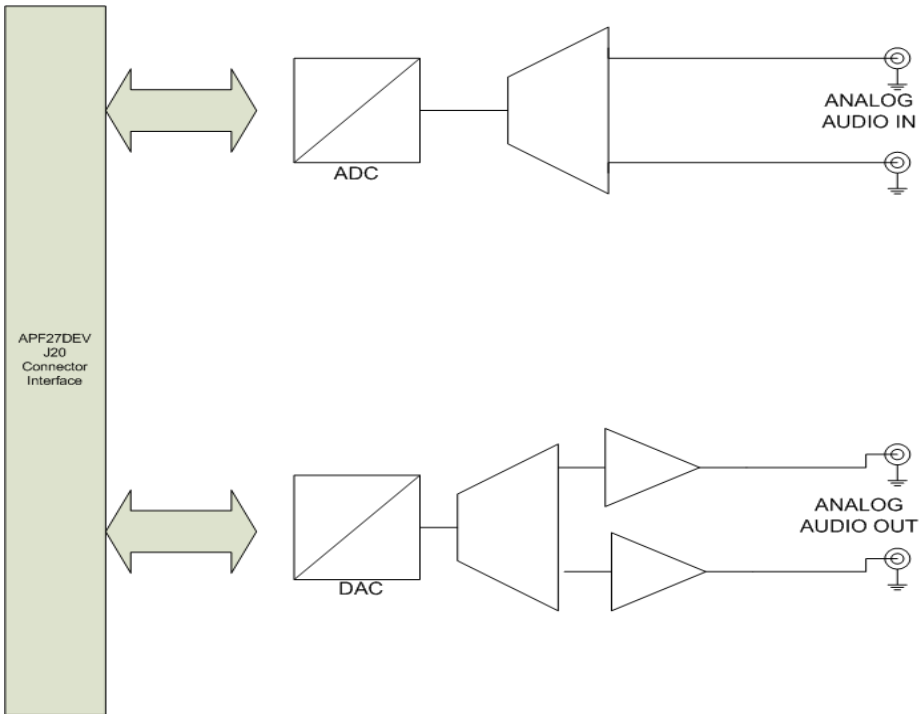
3.1 Analog I/O Board

The analog audio signal will be taken from an external device (phone, cd-player, MP3-player,ipod...), and the frequency band considered will be from DC to 50Khz.

As per the frequency range considered , the system must be able to sample at a rate at least 100 ksamples per second.

To meet this requirement , a dedicated hardware will be used to control the adquisition and interface with the existent FPGA through the APF27DEV board. The PFGA firmware will control the ADC and store the data in a buffer memory, as such a rate is far beyond the microprocessor capability.

The analog board block diagram is shown below. The design is for a stereo analog signal.



The left side is the J20 Connector interface with the APF27_DEV Board. On the other side there are two different parts , the input adquisition part is at the upper section. The two signals are multiplexed into an A/D , and conected to the FPGA through the J20 APF27DEV connector. On the other hand in the lower section the two output analog signals are demultiplexed and then conected to the power amplifiers . Design requirements are basically :

1- Interface through APF27 FPGA (J20 DIL-40 Connector)

2- In order to safe components it should work with 3.3 Vdc voltage supply from the J20 connector

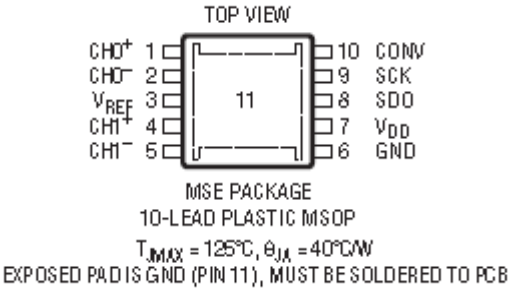
3- ADC and DAC should be able to work at least at 100 ksps

4- Serial interface is preferred to safe conections

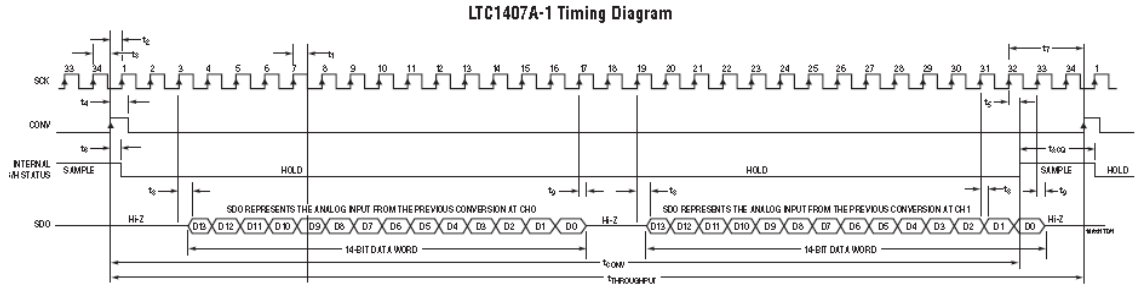
5- Multiplex/Demultiplex capability or dual configuration are preferred to safe components and conections.

3.1.1 ADC device selection

Between the huge variety of ADC devices availables for this pupose the LTC1407A-1 the Linear Technology was selected because it has 2 channels for audio stereo , SPI interface and works with 3.3V supply, that is available at APF27DEV interface connector . Its has two independent channels , thus meeting all requirements and prefferences . It has 14 bit resolution, and finally I found the price is below the average. The pin layout is shown below.



In regard with the maximum working sample frequency, from device datasheet , the timing is shown below



from datashet the minimun SCK period is 19,6 ns. As we have a 100 Mhz clock in the FPGA, we could get a 20 ns period by dividing by two the system frequency. However in order to have a less noise sensitive system a SCK period of 80 ns has been selected by dividing the system clock by 8.

As can be derived from the above timing the conversion time is at least 24 clock periods. The time needed by the FPGA logic to store the data in memory will be added to that time.

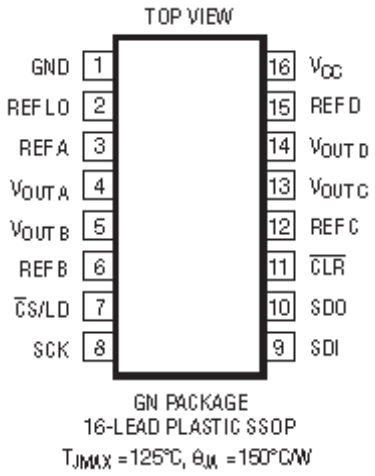
As an initial value a time of sample about 30*80ns = 2,4 us, that would result in about 416 Kps. The final value will depend on the implementation, but is far beyond the speed required to sample signals in the audio frequency range.

We could limit the maximum working frequency in 50 Mhz , put a simple RC antialising filter with a cut frequency in 100 Mhz, Achiving two goals :

- constat gain in 0-40 Mhz frequency range.
- No aliasyng , filtering frequencies over 200 Mhz , as are far from the cut frequency.

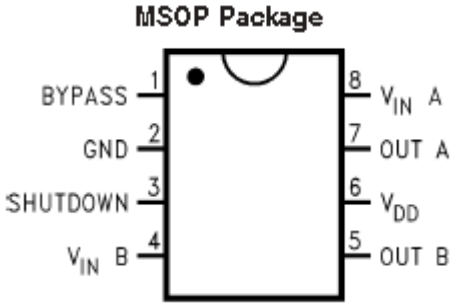
3.1.2 DAC device selection

Between the huge variety of DAC devices availables for this pupose the LTC2624 from Linear Technology was selected because it has 4 channels , SPI/MICROWIRE interface and works with 3.3V supply. Its has four independent channels , thus meeting all requirements and prefferences . The pin layout is shown below. It has 12 bits resolution , enough for this application. The CLK can opperate up to 50 Mhz that is the FPGA clock frequency. Finally I found the price is near the average.

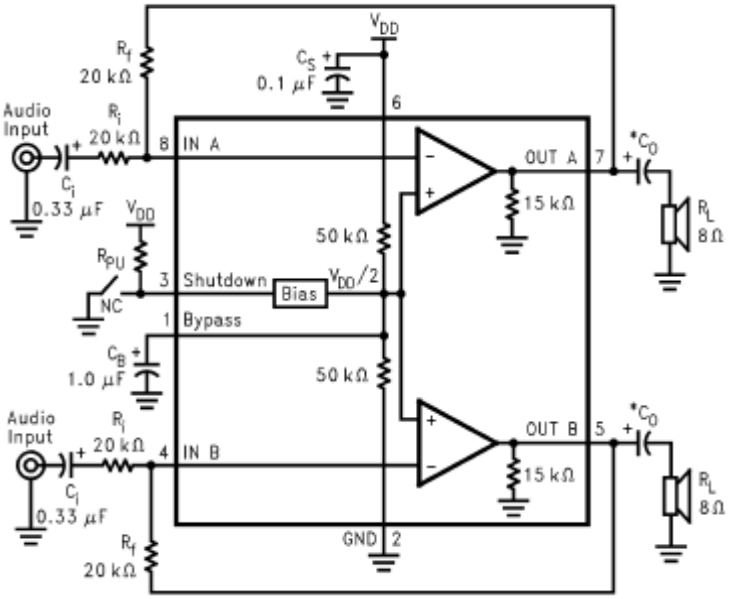


3.1.3 Power amplifier device selection

The LM4881 selected is a Dual 200 mw headphone amplifier , is suited for this application as the supplyy voltage can be from 2.7 to 5.5, so it meets the 3.3V requirement. It also has a low THD+N and good frequency response characteristics, and external gain configuration capability. The pinout layout is shown below.



The typical application circuit shown below, can be applied as it is for this project, where Inputs will be the DAC outputs. However for the specific application will not be used.



3.1.4 Board Interface

The FPGA will be programed to be engaged in the sampling control and signal processing, so the FPGA General purpose input /output capability will be used. The APF27DEV has external access to the FPGA through the J20 connector. This is the pin assignement.

Pin	Description	Type	Supply domain	Pin	Description	Type	Supply domain
1	VCCO1 (in). Bank1 power supply	Power	BANK1	2	IO_L20N_1	IO	VCC1
3	IO_L24P_1	IO	VCC1	4	IO_L20P_1	IO	VCC1
5	IO_L24N_1	IO	VCC1	6	IO_L23N_1	IO	VCC1
7	IO_L22P_1	IO	VCC1	8	IO_L23P_1	IO	VCC1
9	IO_L22N_1	IO	VCC1	10	GND	Power	
11	GND	Power		12	VCCO3 (in). Bank3 power supply	Power	BANK3
13	IO_L24P_3	IO	VCCO3	14	IO_L24N_3	IO	VCCO3
15	IO_L23P_3	IO	VCCO3	16	IO_L23N_3	IO	VCCO3
17	IO_L22P_3	IO	VCCO3	18	IO_L22N_3	IO	VCCO3
19	IO_L20P_3	IO	VCCO3	20	IO_L20N_3	IO	VCCO3
21	IO_L15P_3	IO	VCCO3	22	IO_L15N_3	IO	VCCO3
23	IO_L14P_3	IO	VCCO3	24	IO_L14N_3	IO	VCCO3
25	IO_L12P_3	IO	VCCO3	26	IO_L12N_3	IO	VCCO3
27	IO_L11P_3	IO	VCCO3	28	IO_L11N_3	IO	VCCO3
29	IO_L08P_3	IO	VCCO3	30	IO_L08N_3	IO	VCCO3
31	IO_L03P_3	IO	VCCO3	32	IO_L03N_3	IO	VCCO3
33	IO_L02P_3	IO	VCCO3	34	IO_L02N_3	IO	VCCO3
35	IO_L01P_3	IO	VCCO3	36	IO_L01N_3	IO	VCCO3
37	IP_L25N_3	Input	VCCO3	38	IP_L04N_3	Input	VCCO3
39	+3V3(out)	Power		40	GND	Power	

In addition to the supply pins for 3.3V (39) and GND (40)The signals, the needed signal are

CLK from FPGA , SPI bus signals and ADC, DAC Control signals , for these general purpose IO will be used.

I have chose a separate SPI bus for each device, as its simpler and the speed can be higher at the same cost, as the difference is only two more aditional conections.

3.2 FPGA Firmware

3.2.1 FPGA firmware Functionalities

To meet this project target , the FPGA firmware must implement a device to perform the following functionalities that could be divided into three groups : Interface , Adquisition and FFT processing.

- Interface
  - Processor bus interface
  - ADC Interface
  - Control module to control system operation (sample period , start adquisition, start fft...)
- Adquisition
  - Signal ADC control in order to take signal samples at a selected period.
  - Signal samples values storage in memory
  - processor interruption
  - Processor read access to signal samples memory
- FFT processing
  - FFT IP-Core Control
  - FFT computation processing using memory stored samples.
  - FFT values storage in output memory
  - Processor read access to output memory

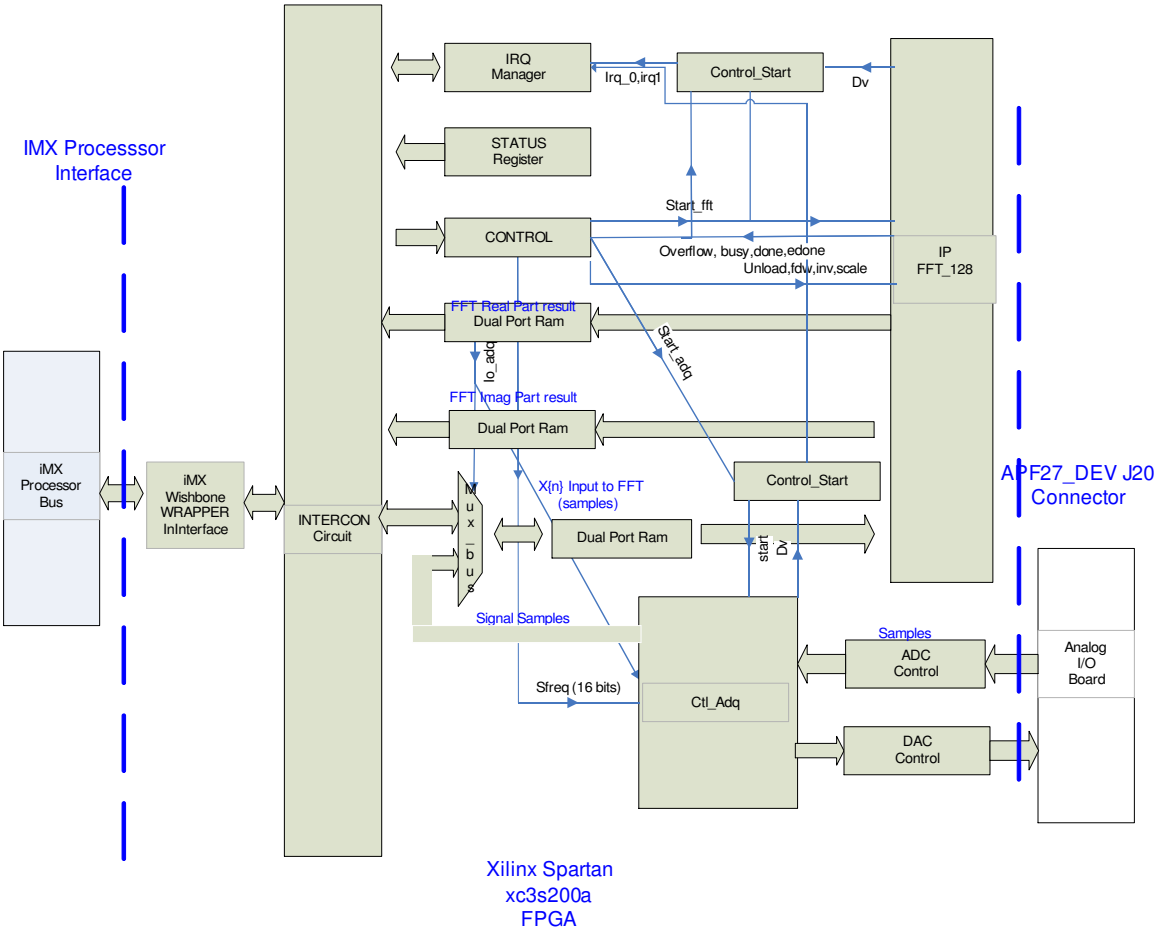
As the FPGA in the APF27 board is from XILINX, the firmware will be implemented using the XILINX ISE webpack development enviroment.

The architecture selected to connect the circuit parts with the iMX processor is based on the wishbone bus. This is a synchronous parallel bus very used for interconeting modules within the same board or chip. Further details can be found in [2] Wishbone Bus B4 Specification.
For the FFT computation an IP-Core from Xilinx will be used. See [3] DS260 FFT v6.0 Datasheet , from Xilinx LogiCore for more information.

The FPGA firmware program block diagram is shown in the next section

3.2.2 FPGA Block diagram

This is the block diagram to be implemented , that would be explained in detail ,in the implementation section.



3.3 LINUX Driver

For this project a character driver will be developed. the basic functionality will be discused in this section.

The device driver should be implemented using the platform device model as it is a device present in the board itself and not in an expansion bus. So the model implements this drivers like device were connected to the platform bus which is a virtual bus.

To make easier the design procedure and use available code will break down the functionality in two parts:

- Interruption manager driver.
- Specific functionality driver.

3.3.1 Interruption manager driver

This driver is who receive the interruption signal that generates the IRQ Manager circuit and process it.

The interruption processing is generic for any application and does the followin g things :

1- Acknowledges The Interruption

2- Find out the interruption cause , for the bit number and calls the corresponding Interrup handler

The code has to be modified only in regard with the number of interrupts covered.

3.3.2 Specific functionality driver

This driver will have asociated a device file /dev/fft0 when installed. This device file , will be the only interface with the user application.

The /dev/fft0 device should support write , read as well as ioctl for special operations . The ioctl operations suported will be :

ioctl FFT_IOCCOMMAND <command_number>

ioctl FFT_SFREQ < period>

ioctl FFT_IOCSTATUS

This Ioctl Operations will be described next:

3.3.2.1 Ioctl COMMAND Operation

<command_number> is the command to execute, that will be transparent , as the command word word will be output at the command register, so this mechanism lets the programmer control the bits transparently, although there will be constants defined in a file header in order to make it easier to understand .
The command word bits will be among others :

- START_ADQ: trailing edge sensitive, the FPGA Ctl_Adq circuit starts the adquisition process . When the adquisition is completed the DV_ADQ status bit is set to 1.

- IO_ADQ: connects the input memory read/write port to the Ctl_Adq circuit , que activated or to the Wrapper circuit when desactivated.
- -START_FFT: when this bit is set to '1' , the FPGA logic starts the FFT computation process , when this is completed the fft values are stored in the FPGA internal memory, The IPCore asserts an interrupt , so the driver stores at fft0 device the values, and the semphor is set, for the processor to read the results.
- FWD_INV_FFT: Selects fordward or Inverse FFT
- FWD_INV_WE: write enable for FWD_INV_FFT signal

3.3.2.2 Ioctl SFREQ Operation

This command changes the samplig period . <period > is an unsigned word value, that represents the number of Ctl_Adq loop for perod, so as one Ctl_Adq period last 2.56 us, the resulting sampling period will be :

$$T_s = 2.56 * period \ \mu s$$

3.3.2.3 ioctl STATUS Operation

returns the internal device status.
The status bits are the following

- DV _ADQ: Adquisitio process finished
- DV_FFT: FFT Transform finished
- FFT_Overflow
- FFT_RFD : FFT ready for data

3.3.2.4 Driver functionality

The read operation , will obtain time domain samples or fft results , depending on the previous ioctl command .

The write operation will store into the input RAM time domain signal samples for the FFT IPCore to compute the fft transform.

So the normal sequence of operations must be done from the aplication program, as follows:

1- assert an ioctl call to the fft0 device with the FFT_IOCOMMAND operation , and a command number mask setting the START_ADQ and IO_ADQ bits, The fft driver will access the FPGA control register. Then, the FPGA next 128 adquisition values area stored in internal input FPGA RAM, and an interruption signal will be asserted to the irq_manager. When the fft_driver interrupt handler is called it will read the 128 samples from the input RAM and will store them in kernel memory, so are availables for the user to read them.

2- the application program shuld then read the fft0 device the samples values and store and/or process them as the application funtion requires.

3- Once the application has processed time sample values , can write them in the device file to write those values in the Input Memory , or simply to pass to the next

step in case it does not want to change the samples.

4- Next the application must assert an ioctl to fft0 device with the START_FFT command number. The fft driver will access the FPGA control reg setting the start_fft bit. Then, the IP- FFT in the FPGA computes the 128 points FFT, and store the values in memory, and an interruption signal will be asserted to the irq_manager. When the fft_driver interrupt handler is called it will read the 128 samples from the Xk Output Memory , and will store them in kernel memory.

2- the application program shuld finally call the read funtion for fft0 device to obtain the 128 fft values real and imaginary parts, so 256 word values must be read.

3.4 User Space Server Program

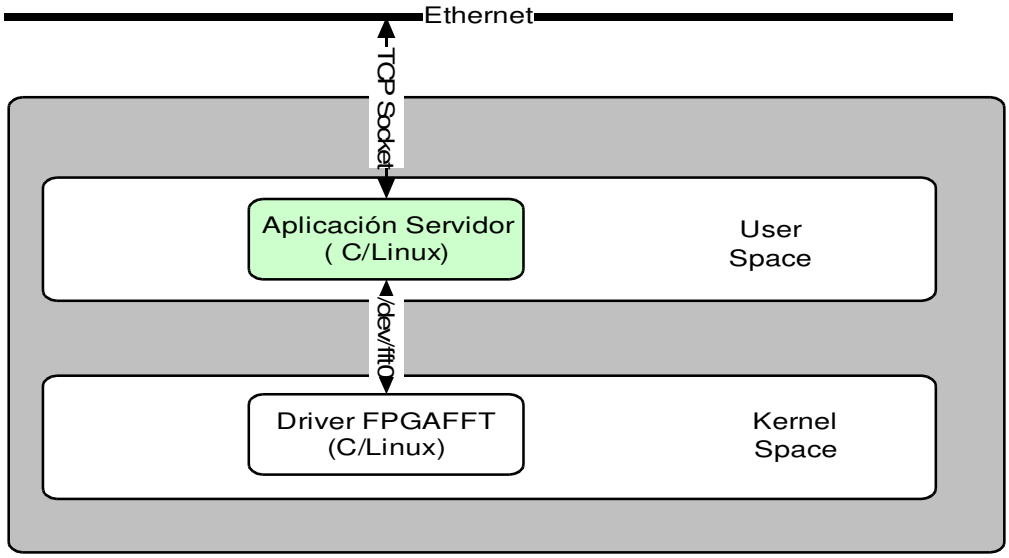
In order to make the data available to external aplications a user space server program should be developed.

As was discused in the above section the device driver enables user space programs to access adquisition and FFT analisys data.

this program should use this driver through the device file /dev/fft0 to get the adquisition and analisys data.

It also must stablish a network TCP service to make those data availble for any client connection. An aplication protocol should be defined for this purpose.

The figure below sumarizes the User Space Program interfaces.



The Server will launch a thread upon a new client connection , and will enter a read process and respond loop until and end of conection message is received , that will cause the thread termination.

The MMI layout will have 4 panels : Control Panel , Signal adquisition , FFT Analisys and Debug Information.

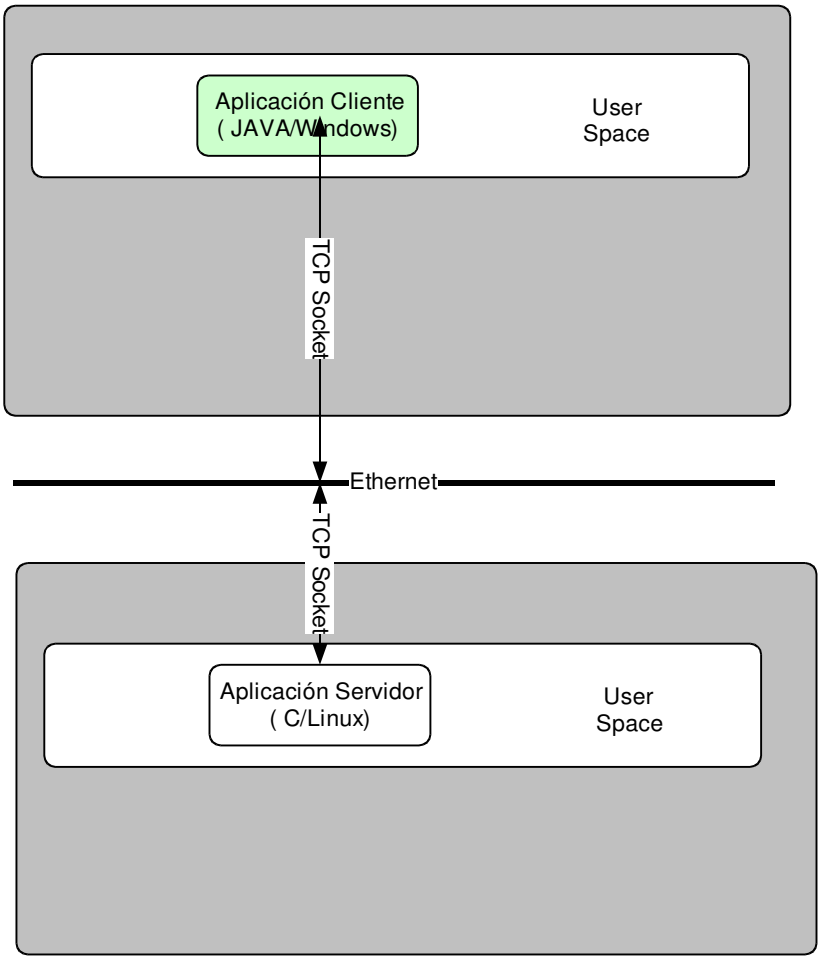
3.5 Client JAVA Aplication

In order to demostarte the overall system performance a Client application program will be developed .

The basic functionality will be connect the Server program, get adquisition and analisys data and display results graphically .

Additionally other funcionalities will be implemented like the sample period selection , the operation mode (single or continuous) and display debug information.

High level programming lenguaje to be used will be JAVA in order to avoid any platform dependencies. The communications protocols will be TCP/IP over ethernet , because a high speed throughput is required. This communication will be implemented using sockets available in both API's , the C/C++ side and JAVA side.

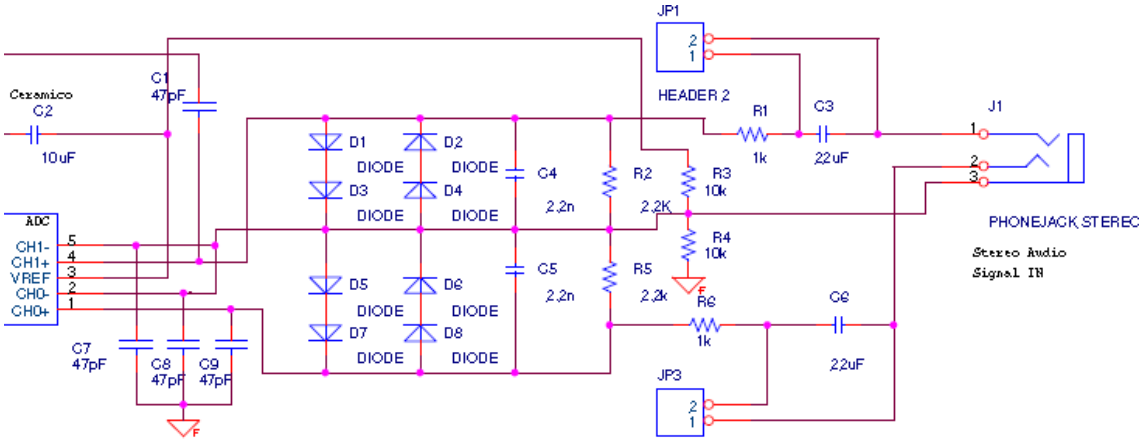


4 SYSTEM IMPLEMENTATION

4.1 Analog I/O Board

4.1.1 Input circuit

The circuit below is used for signal conditionig and overvoltage protection to connect the ADC to a stereo phoneJack. Each signal is connected to one channel. Vref output is about 2.5V, is internally generated by the ADC circuit and is used to set the negative diferential inputs voltage to 1.25V.

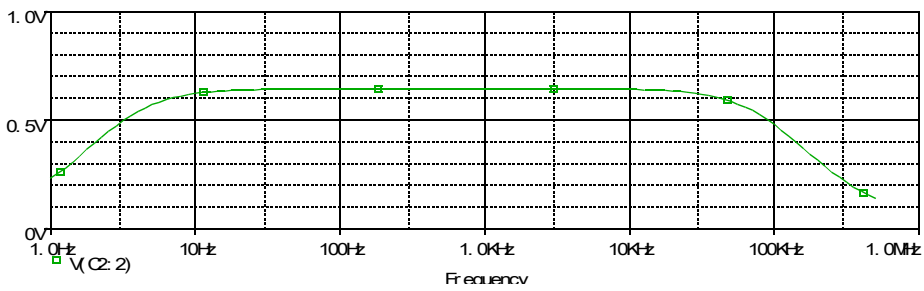


The Diodes arrengement is used to protect the ADC inputs against overvoltage , limiting the voltage to about ±1,25 v , that is the ADC input range.

The coupling capacitors C2 and C6 , together with the input resistances of about 3k forms an high pass firter with corner frecuency about 5 Hz. In case DC coupling is needed can be set the JP2 and JP1 jumper, in order to eliminate the coupling capacitor.

As we saw in the design section, the system features about 400 Khz maximum sample frequency , greater by far than the needed for audio signals, and thereby capable to detect signals with frequency under 200 kHz.

The signal spectrum is supposed to be contained in the audio band (<40Khz) , however in order to attenuate the high frequency noise , over 200 Khz that might exist , a simple RC low pass filter is used.The upper channel filter is formed by capacitor C4 together with the resistors R1,R2 and R3.The cut frequency is about 100 Khz and provides over 7dB attenuation for signals over 200 kHz. The other has an analogous circuit formed by C5,R4,R5 and R6. One channel circuit frequency response simulation plot is shown below .



Capacitor C3 is to Vref DC coupling, finally capacitors C13,C14,C15 and C16 are manufacturer recommended for noise reduction.

4.1.2 ADC and DAC APF Board interface

The board interface JP1 will be connected to the J20 connector in the APF27_Dev board through a flat 40 pin ribbon cable.

The physical FPGA I/O and correspondig conected signals used are:

IO_L15N_3 - CLK signal output from FPGA to ADC and DAC

IO_L20N_3 - SDO signal output from ADC to FPGA

IO_L14N_3 - CONV signal output from FPGA to ADC

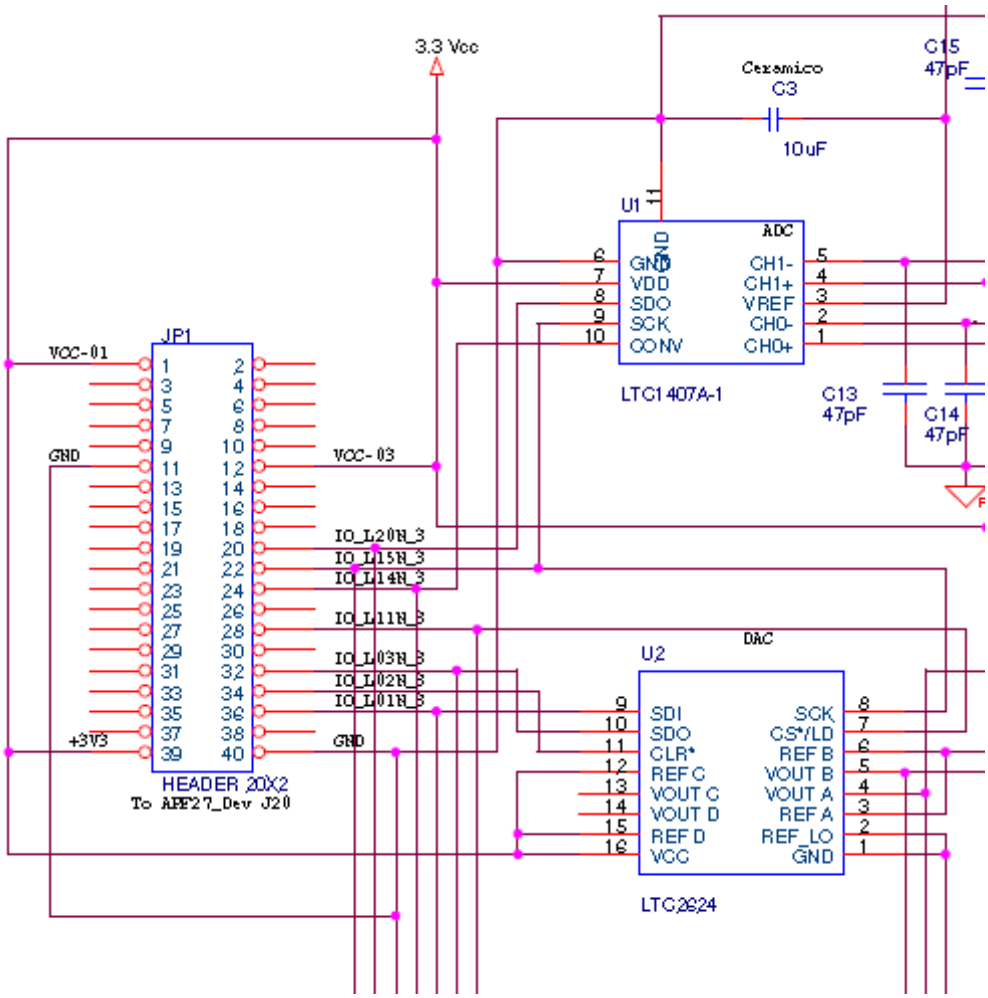
IO_L11N_3 - CS*/LD signal output from FPGA to DAC

IO_L03N_3 - SDO signal output from DAC to FPGA

IO_L02N_3 - CLR* signal output from FPGA to DAC

IO_L01N_3 - SDI signal output from FPGA to DAC

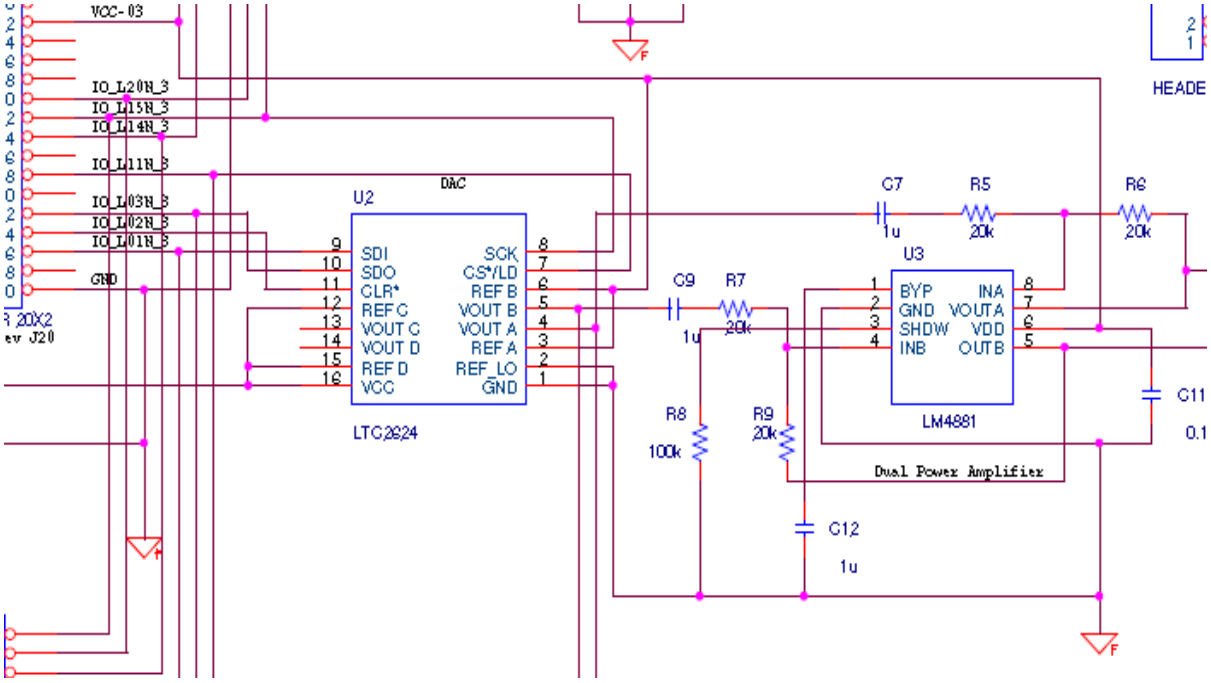
The connection diagram is shown below. As can be seen 3.3v dc supply is taken from the APF27_Dev board.



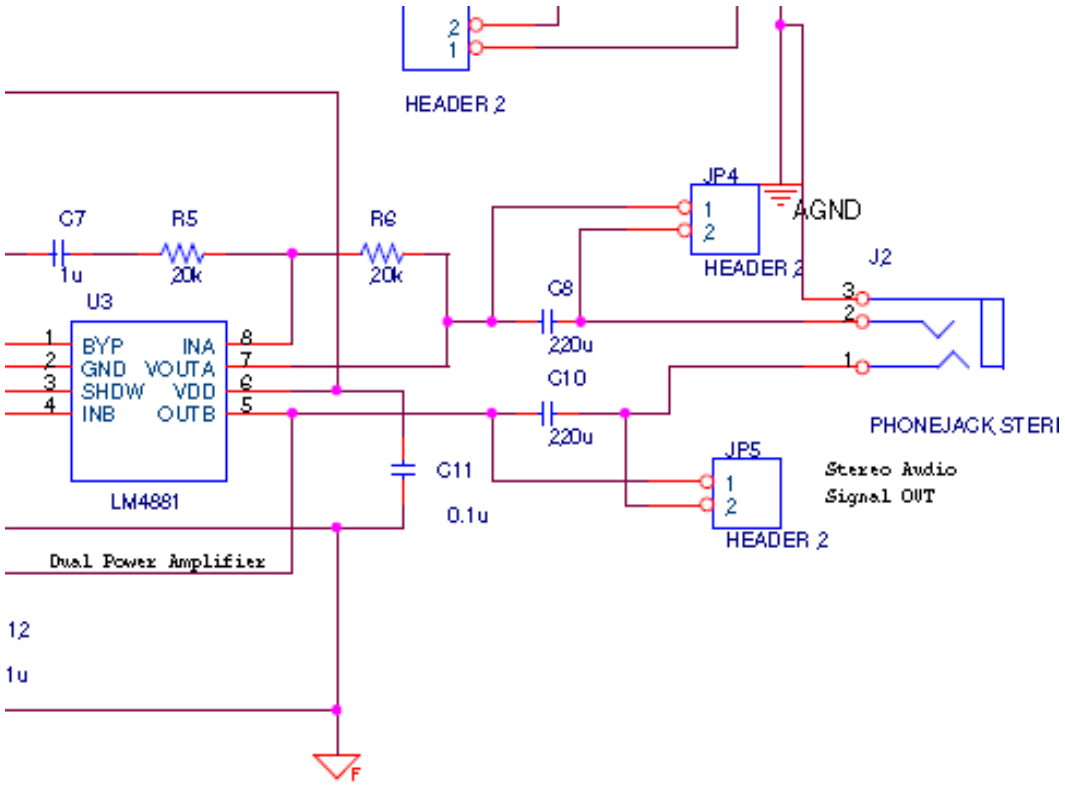
4.1.3 DAC and Output Power amplifier

Given that at the momment the hardware was designed no decision about the final project scope was made, in order to have as many options as posible , this part was included in the board design.So although this part of the circuit is not finally used in this project , is described here for completness.

As can be seen the DAC has 4 channels , but only two are connected, the SDI is the SPI serial data input where the analog value along with the channel number is receibed from the FPGA, while CS*/LD signal is held low, The connection diagram for the dual power amplifier is the rmanufacturer recommended. Finally the amplifiers outputs are conected to a Jack audio connector throught a coupling capacitor.



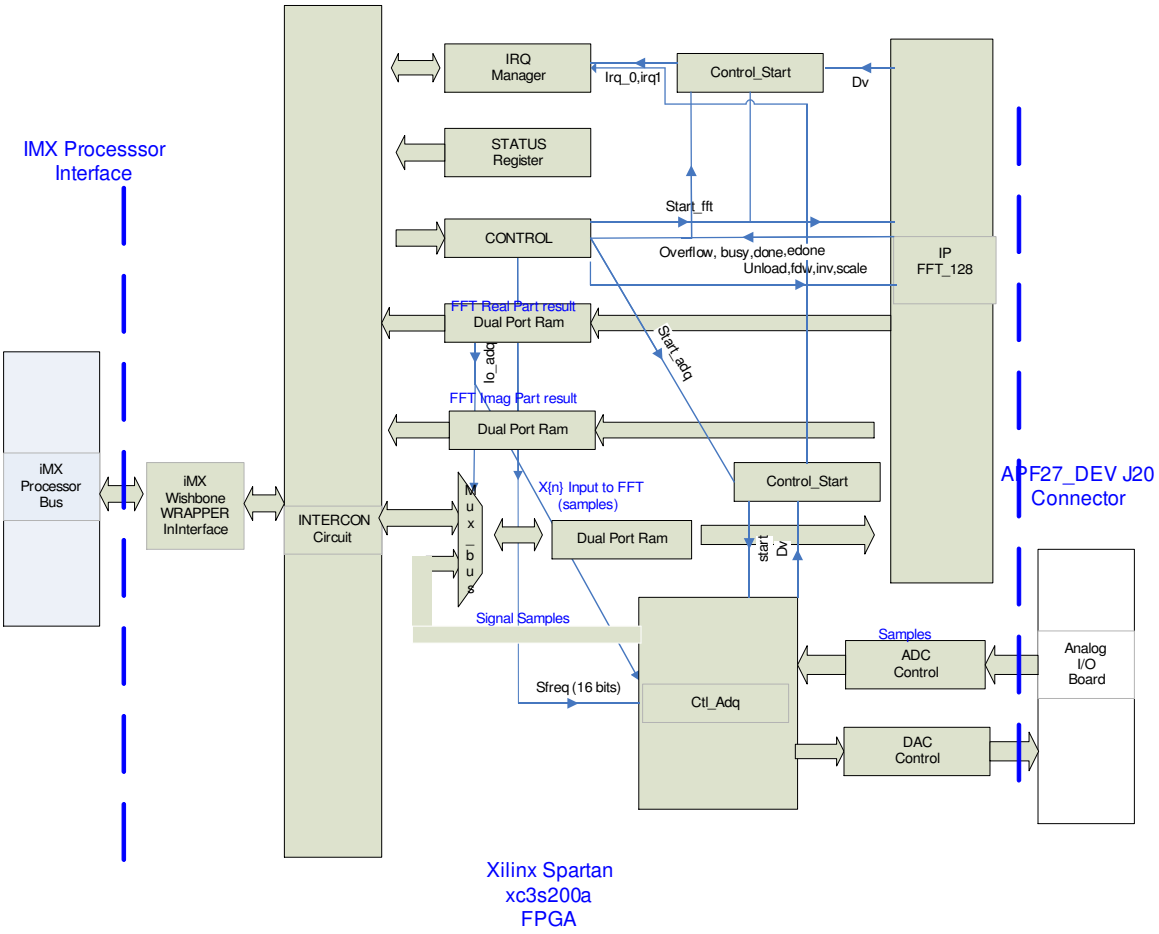
4.1.4 Output interface with Power amplifiers



4.2 FFT Device Firmware implementation using the Spartan 3 FPGA

4.2.1 General Circuit Description

In this section , the implemented firmware will be explained in more detail, to give an overview of the internal architecture, the functionality, the I/O and interface. Finally each module will be explained. In this block diagram the modules are represented in grey boxes conected by buses or signals



The circuit implements the defined functionality to perform the direct signal FFT as well as to sample an analog signal , throught an ADC, using the Spartan 3 FPGA's resources that is available in the ARMADEUS APF27[1] Board and directly connected iMX ARM9 board processor.
The implementation has been made using the Xilinx-ISE Tool suite. Which has a Core Generator Tool integrated with some free IP-Cores from OpenCores that can be directly included in the schematics . Between them is the ds260 FFT IP Core that is used in this project application.
The internal bus selected to connect inside parts is the wishbone bus as is the more extended in OpenCores project.
The wishbone bus is an OpenCores System On Chip (SoC) bus to interconet parts in an IC . For

further information see " OpenCores Wishbone B4 specification "[2]
The FPGA firmware implementation topology is shown in the below schematics, that is formed by several building bloqs which will be expleined in the next section.
The central bloq is an FFT IP-Core from Xilinx LogiCore, which performs the FFT computation, see the Core Datasheet [3] for further information.
The rest of the circuit is to control de ADC in order to sample and store the analog signal , interface the iMX processor and the FFT IP-Core, and to monitor and control its operation through the status and control modules, and finally the interrupt manager to generate the interrupt to the iMX processor.
Wrapper module converts iMX bus signals to Wishbone bus and acts like a wishbone master.
Intercon Module acts decodes addresss activates CS and updates local address for the selected device , the Control , Status , Input and output memorys, Irqmanager and Ctl_Adq devices are wishbone slaves .
The Status has a read register to monitor the operation status.
The Control module has two 16 bit registers, one is for the processor to write, in order to assert control signals, like start_adq, start_fft, and io_adq. The other is for the processor to select the sample period , to do so, this register is connected to Ctl_Adq module.
Ctl_Adq module, controls the ADC comunication and control , so that it samples the analog signal and store the results in external memory input, at a pace depending on the sample_period selected.
When Ctl_Adq receives the start_adq signal it store the 128 next samples in input memory , at the pace selected.To do so, the control register should be writed , so it must assert the io_adq signal to '1' in order to select the access to the input memory. When processor reads the input memory it must be selected io_adq = '0'.

Mux_bus circuit performs the input memory data bus access switch depending on the io_adq signal , to the Ctl Adq to write sample values circuit or processor to read them.

The memory input X(n) implemented with two port RAM is used to store the signal samples, so that them can be read by the processor or used for the FFT IP-Core to make the transform.
The memory output FFT RAM is implemented with a two port RAM to hold 128x16 bit values (see VHDL code for more details). The IP-Core writes this two memories (one for real and other for imaginary parts) with FFT results so that them can be read from the processor.
The sample's imaginary part is hardwired to 'x0000' by conecting a constant signal.

Finally the Irqmanager has four 16 bit registers, each bit in all registers is associated with a particular interruption. In this case two interrupts are used.
The Irq manager circuit activates the allways same irq signal, regardless the source cause, to the iMX processor , setting the corresping interrupt bit in the ISR register, in order the handler to know the interrupt device source.
Note that The FFT implementation is made for 128 points, but is easyly expanded to any power of two, provided not overflowing FPGA memory resources.

4.2.2 FPGA Memory Map

The FPGA addess bus has 13 lines attached to the iMX data bus , as the data bus is 16 bits , the LSB (bit 0) is not used , so from the iMX side only the even address are valid.
The Spartan 3 FPGA 's 4k words address space is mapped to the 0xD6000000 base address and the memory map is:

13 bit offset ADDRESS (HEX)	
0x0000	MASK REGISTER (IRQ manager)
0x0002	PENDING (IRQ manager
0x0004	ID (=1) IRQ manager)
0x0006	NOT USED
0x0008	CONTROL REGISTER 1
0x000A	CONTROL REGISTER 2
0x000C	MASK (Status Register)
0x000E	STATUS REGISTER
0x0100 to 0x01FE	Input Memory Xn(128 word Time domain Signal Samples)
0x0200 to 0x02FE	Output Memory Xk (128 word Spectrum real part)
0x0300 to 0x03FE	Output Memory Xk (128 word Spectrum imag. Part)

As can be seen the top addres space is occupied for the device internal registers.
The botton addres space is occupied to store :

- Xn will be used to store the time domain signal, input to compute the FFT
- Xk (real a imaginary parts) is the computed output FFT from the IP core.

In the following sections the different parts of the circuit are analized.

4.2.3 FPGA External I/O Interface

The FPGA I/O pin assignement is made in the . UCF File (User Constraints File). The complete listing for this project is shown in Annexes. The FPGA I/O used are the ones conected to the J20 connector.

Basically are configured two groups of signals:

the IMX bus interface :

16 Data bus lines addres lines

Lower 12 Address bus lines

Bus Control lines: ext_clk, CS* , eb3*, OE*, gls_irq

External interface signals :

AnalogIO_SDO_ADC

AnalogIO_SCK

AnalogIO_CONV_ADC

AnalogIO_CSLD_DAC

AnalogIO_SDO_DAC

AnalogIO_CLR_DAC

AnalogIO_SDI_DAC

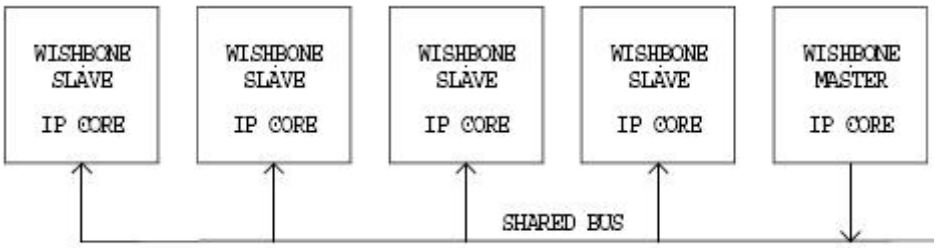
There are also other two signals Conected in the APF27_DEV board:

led_o (in APF27_DEV board)
button_i (in APF27_DEV board)

4.2.4 FPGA Internal Bus: The Wishbone Bus

The **Wishbone Bus [2]** System On Chip (SoC) Interconection Architecture is an open source hardware computer bus intended to let the parts of an integrated circuit communicate with each other. The aim is to allow the connection of differing cores to each other inside of a chip. The **Wishbone Bus** is used by many designs in the OpenCores project.

The topology is a shared bus where the intercon circuit is the master



Wishbone signals

Control signals and the correspondence with the iMX bus is shown below

Wishbone	IMX bus	Description
cyc	not (imx_cs_n) and not(imx_oe_n and imx_eb3_n)	indicates that a valid bus cycle is in progress
stb	not (imx_cs_n) and not(imx_oe_n and imx_eb3_n)	indicates a valid data transfer cycle
we	not (imx_cs_n or imx_eb3_n)	indicates whether the current local bus cycle is a READ or WRITE cycle. The signal is negated during READ cycles, and is asserted during WRITE cycles.
ack		indicates the termination of a normal bus cycle by slave device.

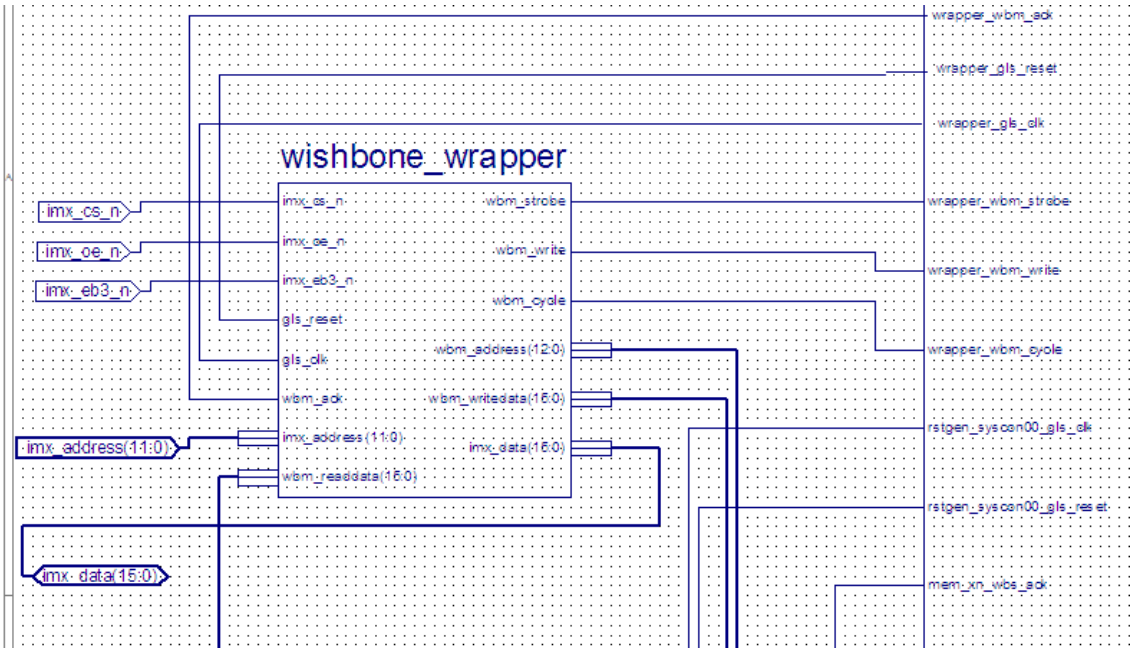
Ack signal is used to acknowledge from slave to master wb device, so its not sended to the iMX processor.

See Wishbone datasheet for more information :
http://cdn.opencores.org/downloads/wbspec_b3.pdf

4.2.5 FPGA Modules Description

4.2.5.1 Wrapper circuit

The wrapper circuit is an opencore used to interface the iMX signals to the standard 16 bit wishbone Bus. So it acts as wishbone bus Master.



So The the circuit I/O is

```
i.MX Signals
  11 lines imx_address   bus ( 11..1)  so LSB is not used
  16 lines mx_data bus
   imx_cs_n   Chip select  signal to FPGA
   imx_oe_n  output enable
   imx_eb3_n

Global Signals
    gls_reset global reset
    gls_clk  global clock

Wishbone interface signals
   wbm_address  , is  the Address bus
   wbm_readdata , is the Data bus for read access
   wbm_writedata  is the Data bus for write access
   wbm_strobe   , DataStrobe
   wbm_write    , Write access
   wbm_ack      , data  acknowledge
   wbm_cycle  , bus cycle in progress
```

As you can see In the VHDL file wraper.vhdl , the interface's ecuations performed are grouped

depending on Synchronization :

```
process(gls_clk, gls_reset)
begin
  if(gls_reset='1') then
    write   <= '0';
    read    <= '0';
    strobe  <= '0';
    writedata <= (others => '0');
    address   <= (others => '0');
  elsif(rising_edge(gls_clk)) then
    strobe  <= not (imx_cs_n) and not(imx_oe_n and imx_eb3_n);
    write   <= not (imx_cs_n or imx_eb3_n);
    read    <= not (imx_cs_n or imx_oe_n);
    address <= imx_address & '0';
    writedata <= imx_data;
  end if;
end process;
```

The other signal are then translated and synchronized with the strobe signal.

```
wbm_address    <= address when (strobe = '1') else (others => '0');
wbm_writedata  <= writedata when (write = '1') else (others => '0');
wbm_strobe     <= strobe;
wbm_write      <= write;
wbm_cycle      <= strobe;
imx_data <= wbm_readdata when(read = '1' ) else (others => 'Z');
```

4.2.5.2 The Intercon Circuit

The intercon circuit functionality is address decoding and wishbone signal generation for each device are connected with: irg_manager , status register, control register, signal memory and transform memory (real and imaginary). The address decodig is performed by this VHDL code fragment, that activates the right CS signal dependig on the address, that activates the correspondig circuit.

```
decodeproc : process(rstgen_syscon00_gls_clk,rstgen_syscon00_gls_reset)
    begin
        if rstgen_syscon00_gls_reset='1' then
            irq_mngr00_swb16_cs <= '0';
            control_swb16_cs <= '0';
            status_swb16_cs <= '0';
        elsif rising_edge(rstgen_syscon00_gls_clk) then

            if wrapper_wbm_address(12 downto 3)="0000000000" and
                        wrapper_wbm_strobe='1' then
                irq_mngr00_swb16_cs <= '1';
            else
                irq_mngr00_swb16_cs <= '0';
            end if;

            if wrapper_wbm_address(12 downto 2)="00000000010" and
wrapper_wbm_strobe='1' then
                control_swb16_cs <= '1';
            else
                control_swb16_cs <= '0';
            end if;
```

```
            if wrapper_wbm_address(12 downto 2)="00000000011" and
wrapper_wbm_strobe='1' then
                status_swb16_cs <= '1';
            else
                status_swb16_cs <= '0';
            end if;

            if wrapper_wbm_address(12 downto ADDR_WIDTH+1)="0000001" and
wrapper_wbm_strobe='1' then
                mem_xn_swb16_cs <= '1';
            else
                mem_xn_swb16_cs <= '0';
            end if;

            if wrapper_wbm_address(12 downto ADDR_WIDTH+1)="0000010" and
wrapper_wbm_strobe='1' then
                mem_xk_r_swb16_cs <= '1';
            else
                mem_xk_r_swb16_cs <= '0';
            end if;
          if wrapper_wbm_address(12 downto ADDR_WIDTH+1)="0000011" and
wrapper_wbm_strobe='1' then
                mem_xk_i_swb16_cs <= '1';
            else
                mem_xk_i_swb16_cs <= '0';
            end if;
          if wrapper_wbm_address(12 downto ADDR_WIDTH+1)="00100" and
wrapper_wbm_strobe='1' then
                mem_xs_i_swb16_cs <= '1';
            else
                mem_xs_i_swb16_cs <= '0';
            end if;

        end if;
    end process decodeproc;
```

Then the address lines for each slave device are assigned to the master wishbone bus counterparts. Each circuit has different address lines depending on his internal registers.

```
      irq_mngr00_wbs_s1_address <= wrapper_wbm_address(2 downto 1);
      control_wbs_addr <= wrapper_wbm_address(1);
      status_wbs_addr <= wrapper_wbm_address(1);
      mem_xn_wbs_addr <= wrapper_wbm_address(ADDR_WIDTH downto 1);
      mem_xk_r_wbs_addr <= wrapper_wbm_address(ADDR_WIDTH downto 1);
      mem_xk_i_wbs_addr <= wrapper_wbm_address(ADDR_WIDTH downto 1)
      mem_xs_wbs_addr <= wrapper_wbm_address(ADDR_WIDTH downto 1);
```

once the chip select signals (CS) are generated are used to send the control wishbone signals to the corresponding wishbone slave and select de readata and ack to send to the master (e.g. the wrapper circuit), for example for irq_manager device the code is :

```
   -- for irq_mngr00
      irq_mngr00_wbs_s1_strobe <= (wrapper_wbm_strobe and
irq_mngr00_swb16_cs );
      irq_mngr00_wbs_s1_cycle <= (wrapper_wbm_cycle and
irq_mngr00_swb16_cs );
      irq_mngr00_wbs_s1_write <= (wrapper_wbm_write and
irq_mngr00_swb16_cs );
```

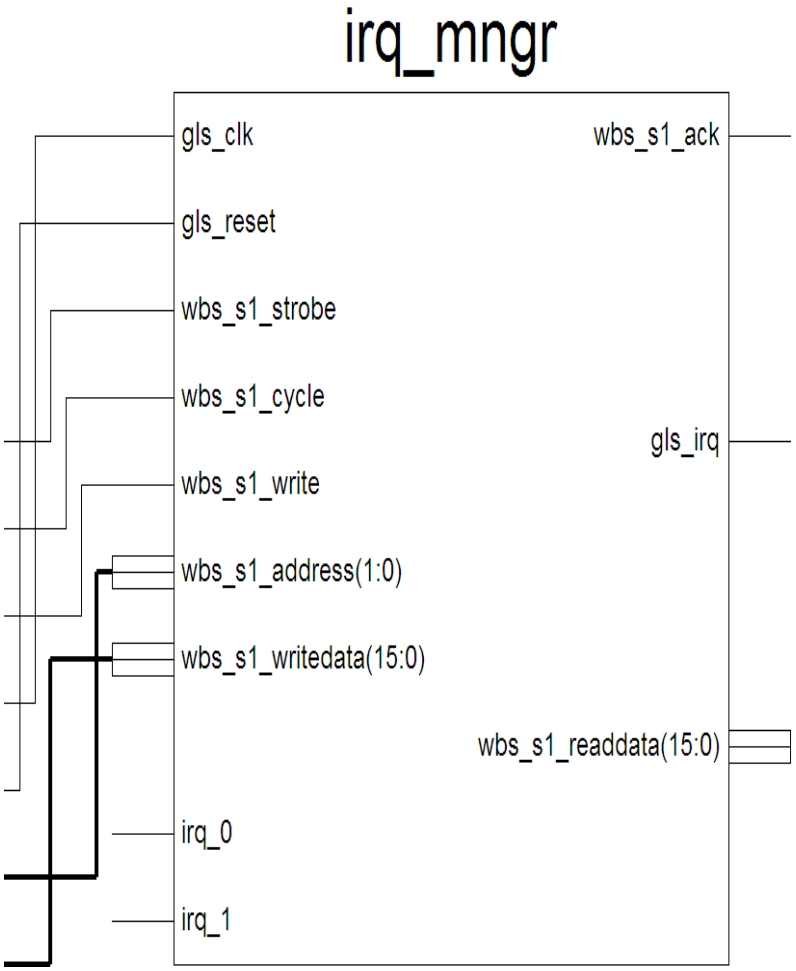
```
      irq_mngr00_wbs_s1_writedata <= wrapper_wbm_writedata when
(wrapper_wbm_write and       irq_mngr00_swb16_cs ) = '1' else (others
=> '0');
```

And finally the ACK signal, and read data for Master (wrapper device) are generated.

```
    wrapper_wbm_readdata <=
            irq_mngr00_wbs_s1_readdata when irq_mngr00_swb16_cs='1' else
             control_wbs_readdata when control_swb16_cs='1' else
             status_wbs_readdata when status_swb16_cs='1' else
            mem_xn_wbs_readdata when mem_xn_swb16_cs='1' else
            mem_xk_r_wbs_readdata when mem_xk_r_swb16_cs='1' else
            mem_xk_i_wbs_readdata when mem_xk_i_swb16_cs='1' else
            mem_xs_wbs_readdata when mem_xs_swb16_cs='1' else
             (others => '0');
    wrapper_wbm_ack <=  (irq_mngr00_wbs_s1_ack and irq_mngr00_swb16_cs)
or
                                        (control_wbs_ack and
control_swb16_cs) or
                                        (status_wbs_ack and
status_swb16_cs) or
                      (mem_xn_wbs_ack and mem_xn_swb16_cs) or
                      (mem_xk_r_wbs_ack and mem_xk_r_swb16_cs) or
                      (mem_xk_i_wbs_ack and mem_xk_i_swb16_cs) or
                      (mem_xs_wbs_ack and mem_xs_swb16_cs);
```

4.2.5.3 The Irq_manager Circuit

This IP functionality is to manage the interrupts. There will be two possible interrupt sources:



1- when the 128 samples adquisition is finished (irq_rq bit of Ctl_Adq circuit).
2- when the FFT IP-Core has finished (FFT IP-Core DV bit).

These two signals will be wired to irq_0 and irq_1 lines so will have assigned the bit 0 and 1 respectively in each register.

Bit Number	IRQ Source
0	Adquisition Completed
1	FFT Finished

In case other application requires more interrupts to be used is easily expanded without need to modify firmware code.

There are 3 registers , so the addres decodiing is two bits wide
The addres , assigned is (the bit 0 is not considered):

0x00	IRQ_MASK : Bit = 1 IRQ Enable d/ 0 = Disabled (R/W)
0x01	IRQ_PENDING : Bit =1 Pending / Write a '1' to ack R/W)
0x10	ID (=1) Read Only

The circuit asserts the same irq pin, so the driver (Irq_manager driver) must check the IRQ_PENDING register in order to know the actual interrupt cause.

The VHDL code for address decodigng is :

```
    if(wbs_s1_strobe = '1' and wbs_s1_write = '0' and wbs_s1_cycle = '1') then
      rd_ack  <= '1';
      if(wbs_s1_address = "00") then
        readdata(irq_count-1 downto 0) <= irq_mask;
      elsif(wbs_s1_address="01") then
        readdata(irq_count-1 downto 0) <= irq_pend;
      elsif(wbs_s1_address="10") then
        readdata <= std_logic_vector(to_unsigned(id,16));
      else
        readdata <= (others => '0');
      end if;
    end if;
```

The two first registers can be read and write , but the ID register is readonly and can be used to check if the device is present. In both registers only Two bits are used , for the two posible sources :

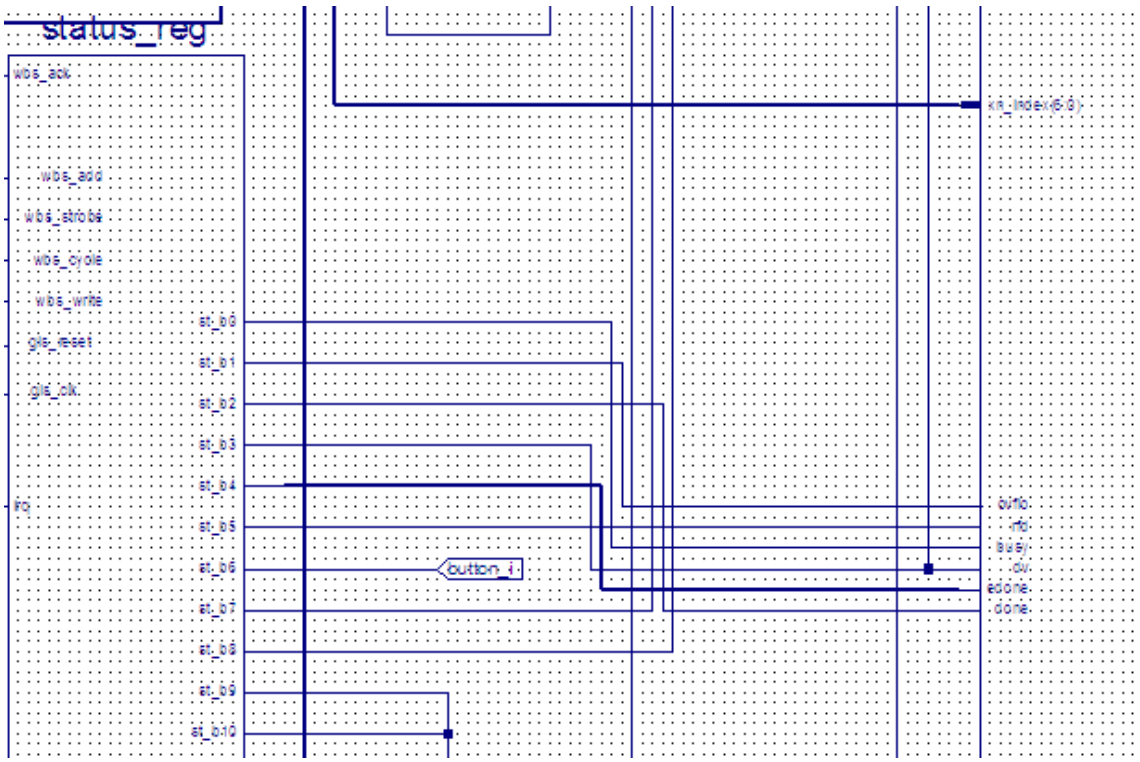
```
 if(gls_reset='1') then
    irq_pend <= (others => '0');
  elsif(rising_edge(gls_clk)) then
    irq_pend <= (irq_pend or ((irq_r and (not irq_old))and irq_mask)) and (not
irq_ack);
```

The operation is as follows
when any bit in IRQ_PENDIG register is set, irq_ gls_signal is assertted causing a system interruption
a IRQ_PENDING bit is set when a rising edge is detected in the corresponding irq_port input, if the correspong bit in IRQ_MASK is clear
a IRQ_PENDING bit is cleared when the ACK when is writen the IRQ_PENDING register with the correspong bit set.

As will be discused in the driver implementation section, this circuit will be acessed by the irq_manager kernel driver. The number of irq lines must be changed dependig on the application needs, and change the code. There is a maximum of 16, In this case only two interrupts are needed.

4.2.5.4 The Status Register

Status Register uses the word's first byte, where the FFT circuit status bits are connected.



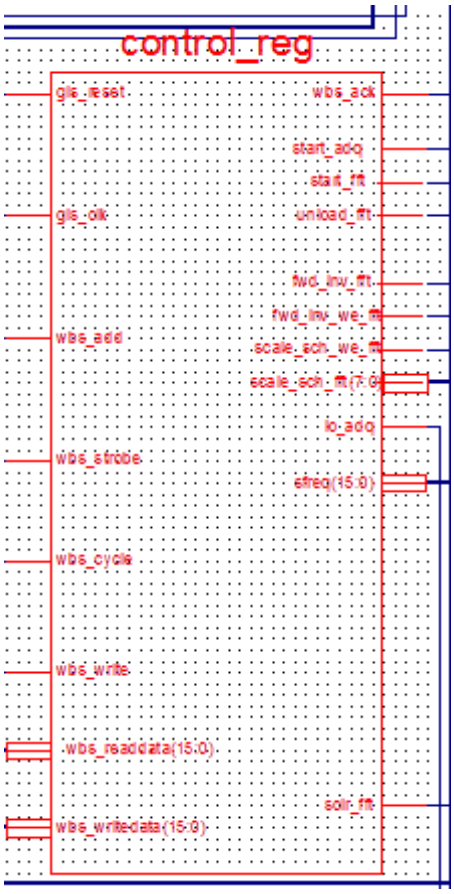
Bit 0	Bit 1	Bit2	Bit3	Bit 4	Bit 5	Bi6	Bit 7
DV_ADQ	FFT_DV	FFT_OVF	FFT_RFD			BTN	

The bit functionality is:
BUSY : This pin is going hish while the core is computing. This signal is also connected to the Board led0.
OVF: overflow indicator (Active High): OVFLO is High during result unloading if any value in the data frame overflowed. The OVFLO signal is reset at the beginning of a new frame of data. This port is optional and only available with scaled arithmetic or single precision floating-point I/O.
DONE:FFT complete strobe (Active High): DONE transitions High for one clock cycle when the transform calculation has completed.
DV: Data valid (Active High): This signal is High when valid data is presented at the output.

EDONE: Early done strobe (Active High): EDONE goes High one clock cycle immediately prior to DONE going active.
RFD: Ready for data (Active High): RFD is High during the load operation.
BTN: connected to the board Push Button
This Module , is transparent to the bit assignement , so can be used for any aplication , conecting the signals to be monitorized.
The circuit also can generate an Interrupt , when a non masked bit state changes . Although this feature is not used in this project.
This module VHDL Code is shown below.

```
      creg : process(gls_clk,gls_reset)
      begin
            if gls_reset = '1' then
            elsif rising_edge(gls_clk) then
            end if;
      end process creg;

      -- rise interruption
       preg : process(gls_clk,gls_reset)
       begin
      if gls_reset = '1' then
            irq <= '0';
            reg <= (others => '0');
            reg_old <=(others => '0');

      elsif rising_edge(gls_clk) then
            reg <=
st_b0&st_b1&st_b2&st_b3&st_b4&st_b5&st_b6&st_b7&st_b8&st_b9&st_b10&st_b11&st_b12
&st_b13&st_b14&st_b15;
            if reg /= reg_old then

                  irq <= (reg(0) and mask(0))or(reg(1) and mask(1)) or
                         (reg(2) and mask(2)) or (reg(3) and mask(3))
                          or (reg(4) and mask(5)) or (reg(6) and mask(6))
                          or (reg(7) and mask(7)) or (reg(8) and mask(8))
                          or (reg(9) and mask(9)) or (reg(10) and mask(10))
                          or (reg(11) and mask(11))or(reg(12) and mask(12))
                         or (reg(13) and mask(13))or(reg(14) and mask(14))
                         or (reg(15) and mask(15));
            else
                  irq <= '0';
            end if;
            reg_old <= reg;
      end if;
      end process preg;

      -- register reading process
      pread : process(gls_clk,gls_reset)
      begin
            if(gls_reset = '1') then
                  wbs_ack <= '0';
                  wbs_readdata <= (others => '0');
            elsif(rising_edge(gls_clk)) then
                  if(wbs_strobe = '1' and wbs_write = '0'
                        and wbs_cycle = '1')then
                  wbs_ack <= '1';
               if wbs_add = REG_MASK then
                        --wbs_readdata <=
      std_logic_vector(to_unsigned(id,16));
                                    wbs_readdata <= mask;

               else
                   wbs_readdata <= reg;
               end if;
            else
                wbs_readdata <= (others => '0');
                wbs_ack <= '0';
            end if;
             end if;
      end process pread;
```

4.2.5.5 Control Register



Control Register bits are connected to Equalizer and FFT inputs to control its operation.
The outputs functionality is :
START_ADQ: start the adquisition process.
START: FFT start signal (Active High): START is asserted to begin the data loading and transform calculation (for the Burst I/O architectures). For Streaming I/O, START begins data loading, which proceeds directly to transform calculation and then data unloading.
IO_ADQ: Controls whether an Input Analog to Digital or output Digital to Analog process is taking place.
UNLOAD_FFT : Unloads results (Active High)
SCLR_FFT : IP Reset
F/I_WE: Write enable for FWD_INV (Active High).
F/I: Fordward FFT or Inverse FFT select
indicates if a forward FFT or an inverse FFT is performed. When FWD_INV=1, a forward transform is computed. If FWD_INV=0, an inverse transform is computed.
SFREQ : Sample Frequency for adquisition.
SCL_WE:Write enable for SCALE_SCH (Active High): This port is available only with scaled arithmetic.
SCLR: Master synchronous reset (Active High): Optional port. The synchronous reset overrides clock enable when both are present on the core.
ULD: Result unloading (Active High). Starts the unloadig of the results in natural order.
SCALE0-SCALE7: For Burst I/O architectures, the scaling schedule is specified with two bits for each stage, starting at the two LSBs. The scaling can be specified as 3, 2, 1, or 0, which represents the number of bits to be shifted. An example scaling schedule for *N* =1024, Radix-4, Burst I/O is [1 0 2 3 2]. For N=128, Radix-2, Burst I/O or Radix-2 Lite, Burst I/O, one possible

scaling schedule is [1 1 1 1 0 1 2]. For Pipelined, Streaming I/O architecture, the scaling schedule is specified with two bits for every pair of Radix-2 stages, starting at the two LSBs. For example, a scaling schedule for N=256 could be [2 2 2 3]. When N is not a power of 4, the maximum bit growth for the last stage is one bit. For instance, [0 2 2 2 2] or [1 2 2 2 2] are valid scaling schedules for N=512, but [2 2 2 2 2] is invalid. The two MSBs of SCALE_SCH can only be 00 or 01. This port is only available with scaled arithmetic (not unscaled, block floating-point or single precision floating-point).

This module VHDL Code is shown below. Two registers are defined. Reg_2 is used for the sfreq parameter. Two flags are defined read_ack and write_ack , the final wishbone ack is the or of them.

```
    signal reg_1 : std_logic_vector( wb_size-1 downto 0);
    signal reg_2 : std_logic_vector( wb_size-1 downto 0);

    signal read_ack : std_logic ;
    signal write_ack : std_logic ;
```

both registers are set to '0' upon reset. The circuit logic is active when cycle and strobe signals are active on the clk rising edge.

As can be seen in the VHDL code the registers can be rad or write, and selection in done depending on the wbs_add.

```
begin


wbs_ack <= read_ack or write_ack;

-- manage register
write_bloc : process(gls_clk,gls_reset)
begin
    if gls_reset = '1' then
        reg_1 <= (others => '0');
        reg_2 <= (others => '0');

        write_ack <= '0';
    elsif rising_edge(gls_clk) then
        if ((wbs_strobe and wbs_write and wbs_cycle) = '1' ) then
             if wbs_add = '0' then
               reg_1 <= wbs_writedata;
                        else
               reg_2 <= wbs_writedata;
             end if;
             write_ack <= '1';
        else
             write_ack <= '0';
        end if;
    end if;

end process write_bloc;

read_bloc : process(gls_clk, gls_reset)
begin
    if gls_reset = '1' then
        wbs_readdata <= (others => '0');
    elsif rising_edge(gls_clk) then
        if (wbs_strobe = '1' and wbs_write = '0'  and wbs_cycle = '1' ) then
            read_ack <= '1';
            if wbs_add = '0' then
                wbs_readdata <= reg_1;
```

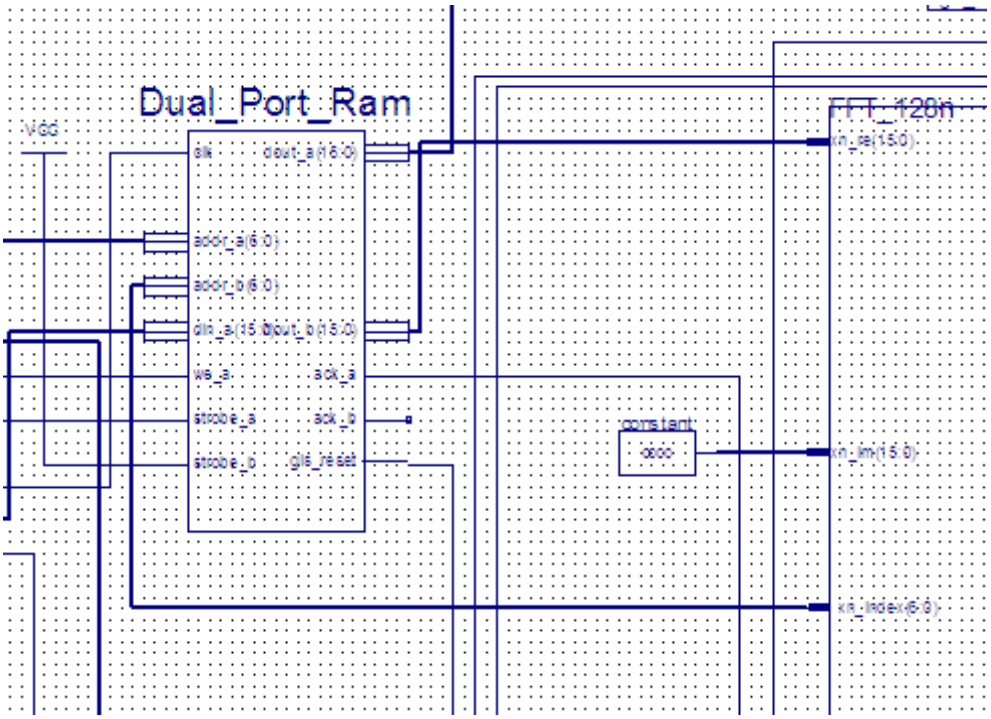
```
            else
                wbs_readdata <= reg_2;
            end if;
        else
            wbs_readdata <= (others => '0');
            read_ack <= '0';
        end if;
    end if;
end process read_bloc;
```

Finally the module outputs are connected to the registers bits:

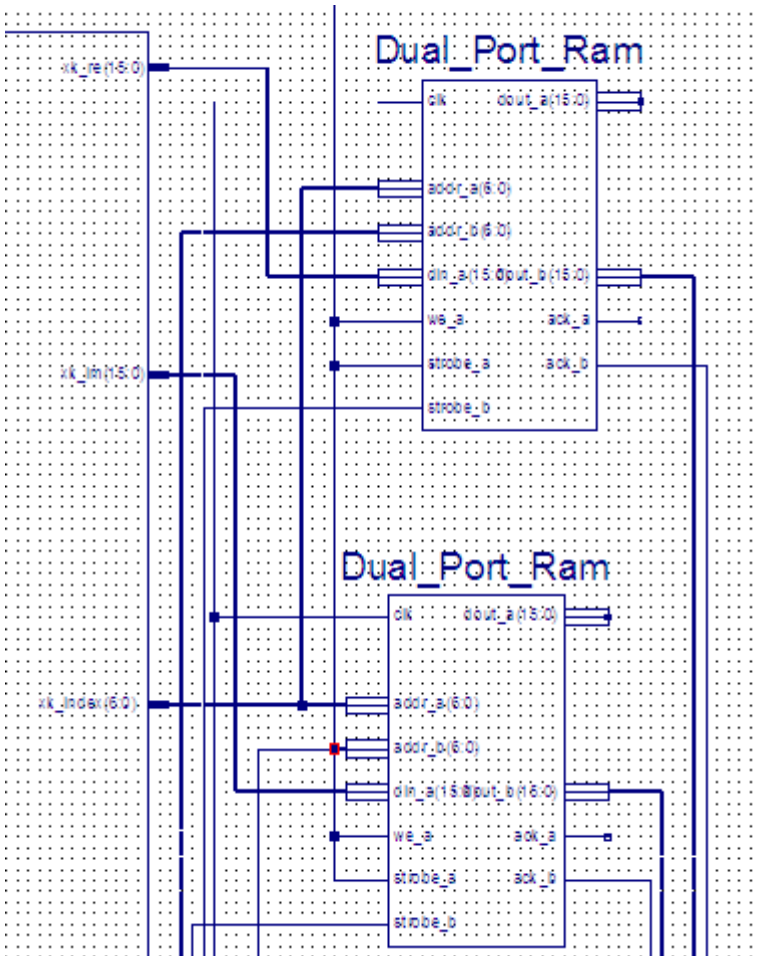
```
start_fft <= reg_1 (0);
start_adq <= reg_1 (1);
unload_fft <= reg_1 (2);
sclr_fft <= reg_1 (3);
fwd_inv_fft <= reg_1 (4);
fwd_inv_we_fft <= reg_1 (5);
scale_sch_we_fft <=reg_1(6);
io_adq <= reg_1(7);
scale_sch_fft <= reg_1(15 downto 8);

sfreq <= reg_2;
```

4.2.5.6 Dual-Port Memorys

This memorys are dual port RAM with wishbone interface.
This circuit is applied for the input and output memories. It has two ports. The port A is a Read/Write port and de port B is a Read only port. It has been dimesioned to 128 kwords , however it can be easily changed.
The Input data dual port RAM is used to store the signal X(n) sampled values through the A port . The FFT IP-Core reads asyncronously through the port B the data stored by using the XN_INDEX lines to address the memory (see datasheet).



Output Memory
while the two output dual-port RAM port is for FFT writing and processor read access , and is used to srore results for the real and imaginary part.
This is a dual-port 128 kwords memory, used for the FFT IP-Core to store the signal frequency-samples in the transform domain , thru the A port , using the XT_INDEX lines to address the memory. The iMX Processor reads asyncronously the data stored using the B port.



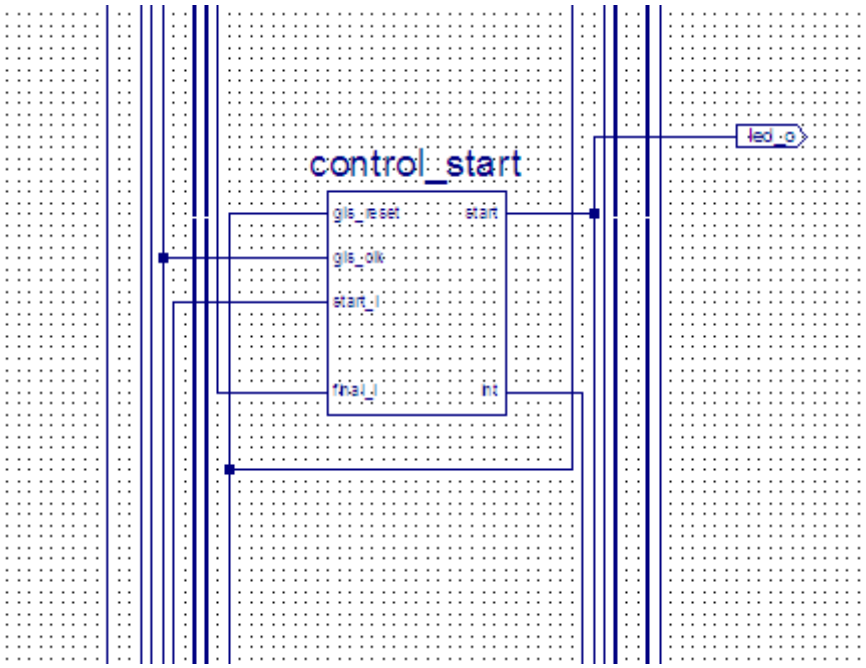
The access is syncronous and a ACK signal is generated for the wishbone interface protocol . See the VHDL code.

```
begin
      process (clk)
      begin
      if (clk'event and clk ='1') then
          if (we_a = '1') and (strobe_a = '1') then
                ram (to_integer(unsigned (addr_a))) <= din_a;
                    write_ack_a <='1';
              else
                    write_ack_a <='0';
              end if;
              if (we_a = '0') and (strobe_a ='1') then
                 read_ack_a <= '1';
                    addr_a_reg <= addr_a;
              else
                 read_ack_a <= '0';
              end if;
              if (strobe_b ='1') then
                 addr_b_reg <= addr_b;
                 read_ack_b <= '1';
              else
                 read_ack_b <= '0';
              end if;
      end if;

      dout_a <= ram (to_integer(unsigned(addr_a_reg)));
      dout_b <= ram (to_integer(unsigned(addr_b_reg)));
      ack_a <= read_ack_a or write_ack_a;
      ack_b <= read_ack_b;
      end process;
```

```
end Behavioral;
```

4.2.5.7 Control_start circuit

This circuit control the FFT and Adquisition execution's process , detects a rising edge in the start_fft control bit, to start the FFT process, and waits the rising edge in the start_i signal, to activate the start output , when final_i rises the circuit set to '0' the start signal and generates a pulse in the int line connected to an interrupt input of irq_manager circuit.
There is two control_start circuit , one for the sampling process and other for the FFT process

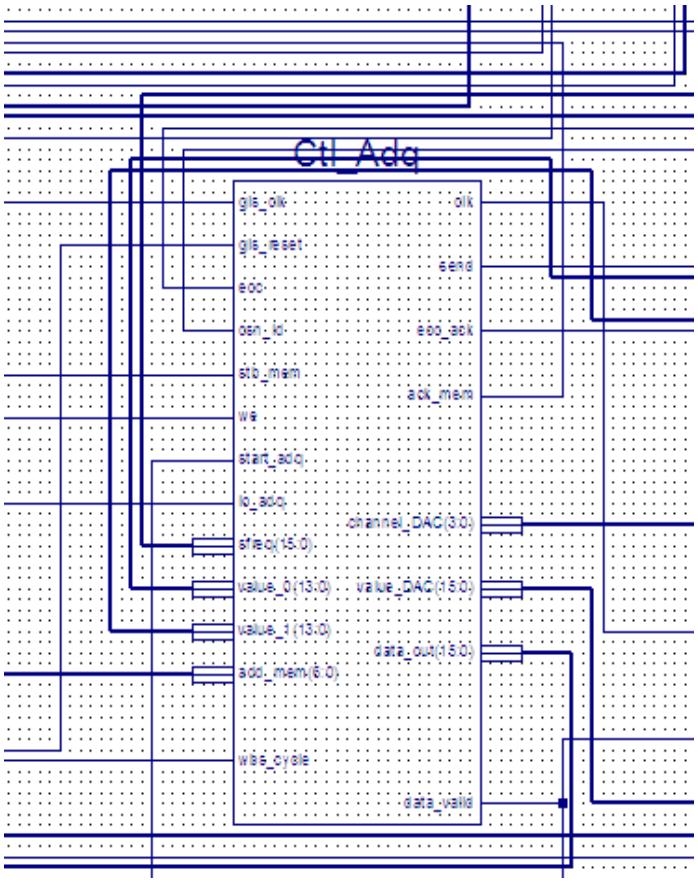


This is the VHDL code

```
Entity control_start is
      port
      (
            -- global signals
            gls_reset : in std_logic ;
            gls_clk      : in std_logic ;
            -- output
            start : out std_logic ;
            int : out std_logic;
            --input
            start_i             : in std_logic;
            final_i         : in std_logic
      );
end entity;
--------------------------------------------------------------------------------
Architecture control_start_1 of control_start is
--------------------------------------------------------------------------------
      signal start_r : std_logic ;
      signal final_r : std_logic;

begin
      - start control
      cstart : process(gls_clk,gls_reset)
      begin
```

```
            if gls_reset = '1' then
                  start <= '0';
                  start_r <= '0';
            elsif rising_edge(gls_clk) then
                  if start_r /= start_i then
                     if ( start_i = '1') then
                           start <= '1';
                     else
                            start <= '0';
                      end if;
                  else
                        start <= '0';
                  end if;
                  start_r <= start_i;
            end if;
      end process cstart;

      -- generate interruption when process  ends
      cint : process(gls_clk,gls_reset)
      begin
            if gls_reset = '1' then
                  int <= '0';
                  final_r <= '0';
            elsif rising_edge(gls_clk) then
                  if final_r /= final_i then
                     if ( final_i = '1') then
                           int <= '1';
                     else
                                 int <= '0';
                            end if;
                  else
                        int <= '0';
                  end if;
                  final_r <= final_i;
            end if;
      end process cint;
```

4.2.5.8 Ctl_Adq Module



The Ctl_Adq Module controls the Adquisition Process.

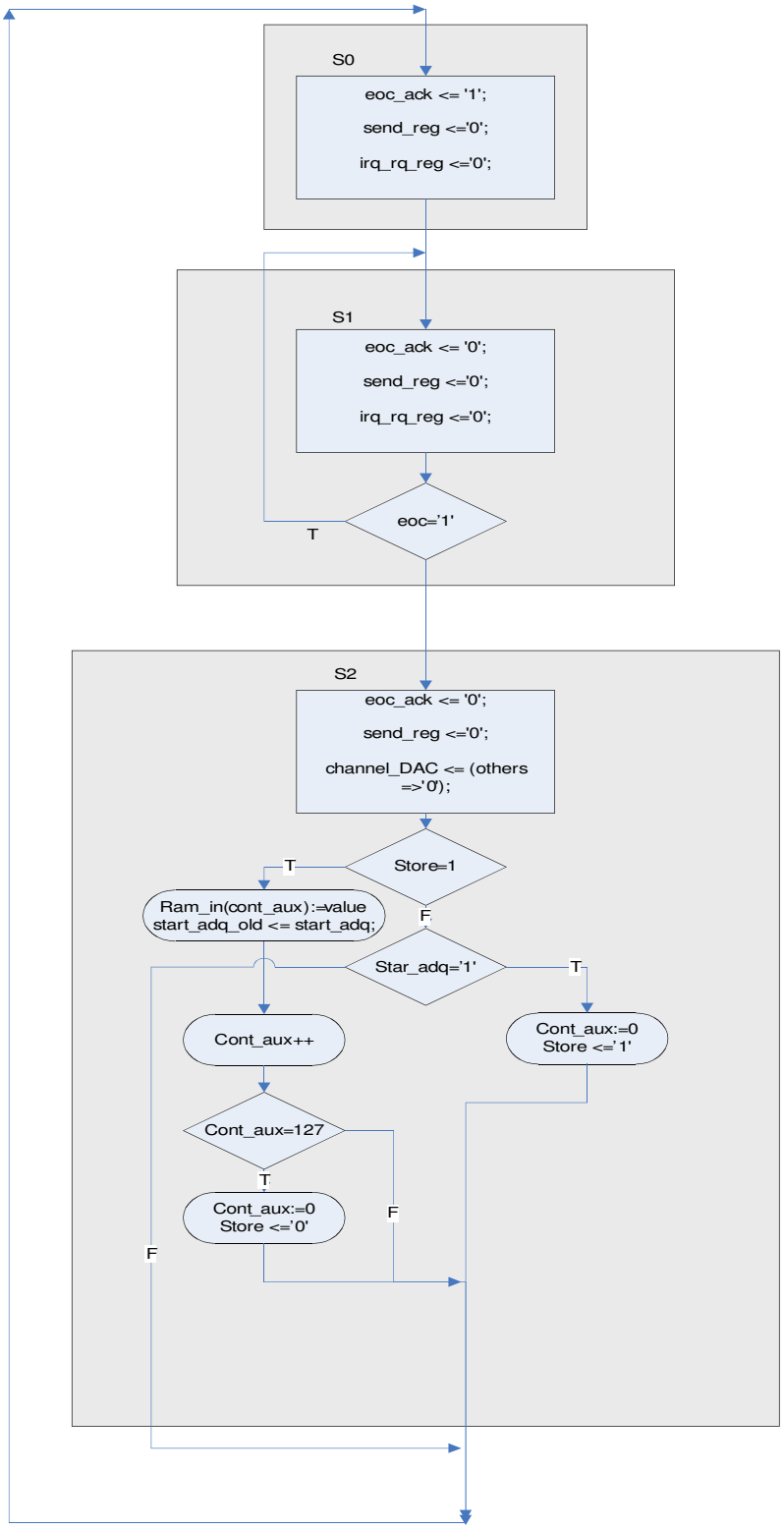
The clk_ext frequency has been divided by four through the div_clk circuit , in order to limit the frequency to the maximum frequency that DAC and ADC are able to handle. It receives the order to sample from the Control register Start_Adq signal. Sets the IO_Adq signsl '1' to store Analog input samplig in the imput memory. It also interfaces with ADC y DAC, and has an irq_rq line to connect to connect yo control_start module.

This are Port definitions

```
entity Ctl_Adq is
    Port ( gls_clk : in STD_LOGIC;
        gls_reset : in STD_LOGIC;
         wbs_cycle: in STD_LOGIC;
         clk : out  STD_LOGIC;
         sfreq : in STD_LOGIC_VECTOR (15 downto 0);
                send : out  STD_LOGIC;
                channel_DAC : out  STD_LOGIC_VECTOR (3 downto 0);
                 value_0 : in  STD_LOGIC_VECTOR (13 downto 0);
                value_1 : in  STD_LOGIC_VECTOR (13 downto 0);
         value_DAC :out STD_LOGIC_VECTOR (15 downto 0);
               eoc : in  STD_LOGIC;
         csn_ld : in STD_LOGIC;
```

```
        eoc_ack : out  STD_LOGIC;
               add_mem : in  STD_LOGIC_VECTOR (6 downto 0);
              data_in : in  STD_LOGIC_VECTOR (15 downto 0);
        data_out: out STD_LOGIC_VECTOR (15 downto 0);
                 stb_mem : in  STD_LOGIC;
         we : in STD_LOGIC;
         ack_mem: out STD_LOGIC;
         start_adq : in STD_LOGIC;
         io_adq: in STD_LOGIC;                    -- 1= Input ADC / 0= Output
DAC
         irq_rq: out STD_LOGIC  );
end Ctl_adq;
```

The adquisition process is implemeted through an FSM , whose ASM is shown below.


S0
eoc_ack <= '1';
send_reg <='0';
irq_rq_reg <='0';
S1
eoc_ack <= '0';
send_reg <='0';
irq_rq_reg <='0';
eoc='1'
T
S2
eoc_ack <= '0';
send_reg <='0';
channel_DAC <= (others =>'0');
Store=1
T
Ram_in(cont_aux):=value
start_adq_old <= start_adq;
F
Star_adq='1'
T
Cont_aux++
Cont_aux:=0
Store <='1'
Cont_aux=127
T
Cont_aux:=0
Store <='0'
F
F
T

As can be seen the FSM has threre states :

S0 : Asserts eoc_act to start the ADC conversion

S1 : Waits for the conversion to complete

S2: Get value and store in memory see ASM flowchart to understand sampling management

The process to control the statesequence source code in VHDL is shown below:

```
process (state_reg, eoc, csn_ld)
begin
      case state_reg is
            when s0 =>          -- Assert eoc_ack=1 to start ADC conversion
              state_next <= s1;
            when s1 =>         -- waits for eoc of ADC
                  if (eoc = '1') then
                        state_next <= s2;
                  else
                       state_next <= s1;
                  end if;
             when s2 =>  -- gets values from control_ADC and store in memory
                  state_next <= s0;
      end case;
end process;
```

Each state has actions asociated, this is controled by other process shown below , that basically does diffrentes actions or logics depending on the current state, that is the Moore logic.

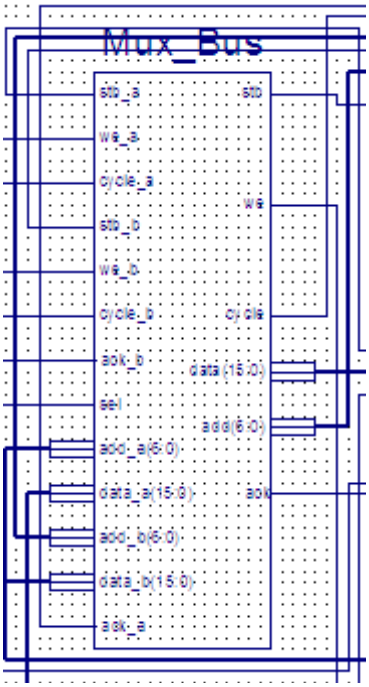
```
process (state_reg)
    variable cont_aux : integer range 0 to 127 ;
        variable cont_vect: std_logic_vector (6 downto 0);

begin
      case state_reg is
            when s0 =>
              eoc_ack <= '1';
              send_reg <='0';
              irq_rq_reg <='0';

         when s1 =>
              eoc_ack <= '0';
        send_reg <='0';
              irq_rq_reg <='0';

         when s2 =>
              eoc_ack <= '0';
```

```
                send_reg <= '0';
                      channel_DAC <= (others =>'0');
                if (store = '1' )) then
                     ram_in (cont_aux) <= std_logic_vector(to_unsigned
(cont_aux,16));                          cont_aux := cont_aux+1;
                     if ( cont_aux = 127 ) then
                             cont_aux := 0;
                            irq_rq_reg <='1';
                      end if;
                else
                            if ( start_adq = '1')
                            cont_aux = 0;
                            end if;
                end if;
                start_adq_old <= start_adq;
          when s3 =>
                  send_reg <= '1';
                  eoc_ack <= '0';
          end case;
end process;
```

The ack line is controled with an FSM in the same way as in other circuits already explaines, so I will not repeat here the same discusion. Finally the output.conections, remenbering that Th memry must be read (from ADC) if io_adq = 1. The VHDL code is.

```
data_out <= ram_in (to_integer(unsigned(addr_reg))) when (io_adq = '1') else
ram_out (to_integer(unsigned(addr_reg))) ;
ack_mem <= read_ack or write_ack;
clk <= gls_clk;
send <= send_reg;
value_0_aux ( 15 downto 0) <=  value_0 (13 downto 0)&"00";
value_DAC <= value_0_aux;
irq_rq <= irq_rq_reg;
```

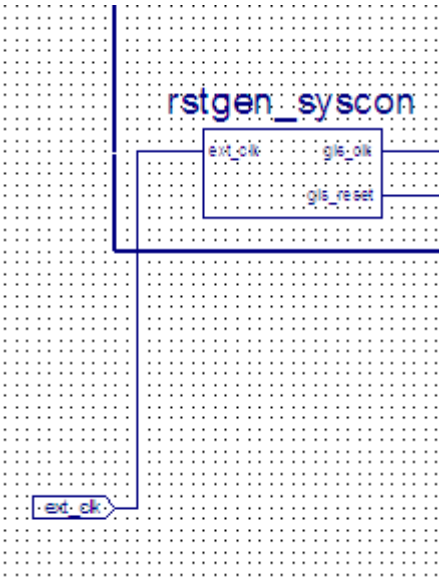
4.2.5.9 Mux_Bus circuit



Mux_bus circuit performs the input memory addres, data and some control lines bus access switch depending on the io_adq signal , in order to connect the input memory to the Ctl Adq to write sample values circuit or processor to read them (see block diagram).
Thi s is done with the following code:

```
add <= add_a when sel = '1' else add_b ;
data <= data_a when sel = '1' else data_b;
stb <= stb_a when sel = '1' else stb_b;
we <= we_a when sel = '1' else we_b;
cycle <= cycle_a when sel='1' else cycle_b;
ack_a <= ack when sel = '1' else '0';
ack_b <= ack when sel = '0' else '0';
```

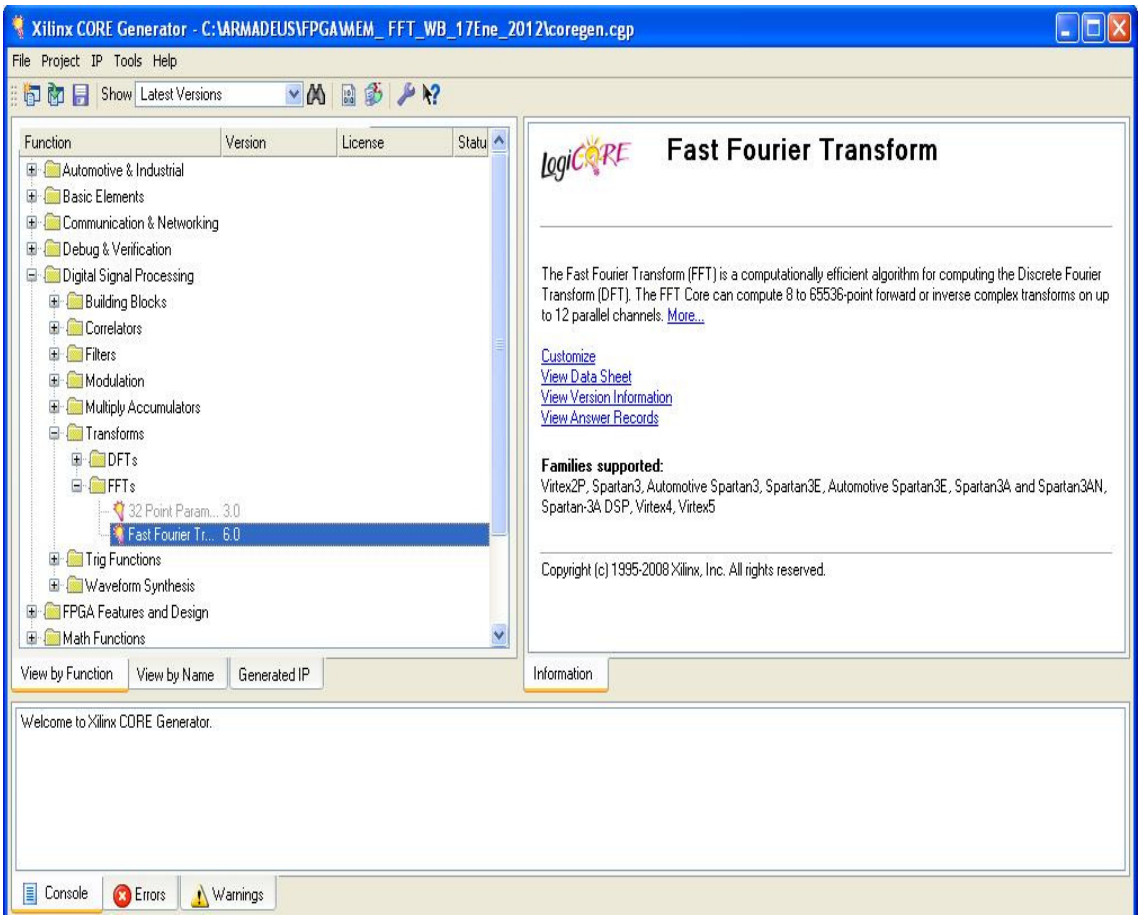
4.2.5.10 Rtsgen_syscon circuit



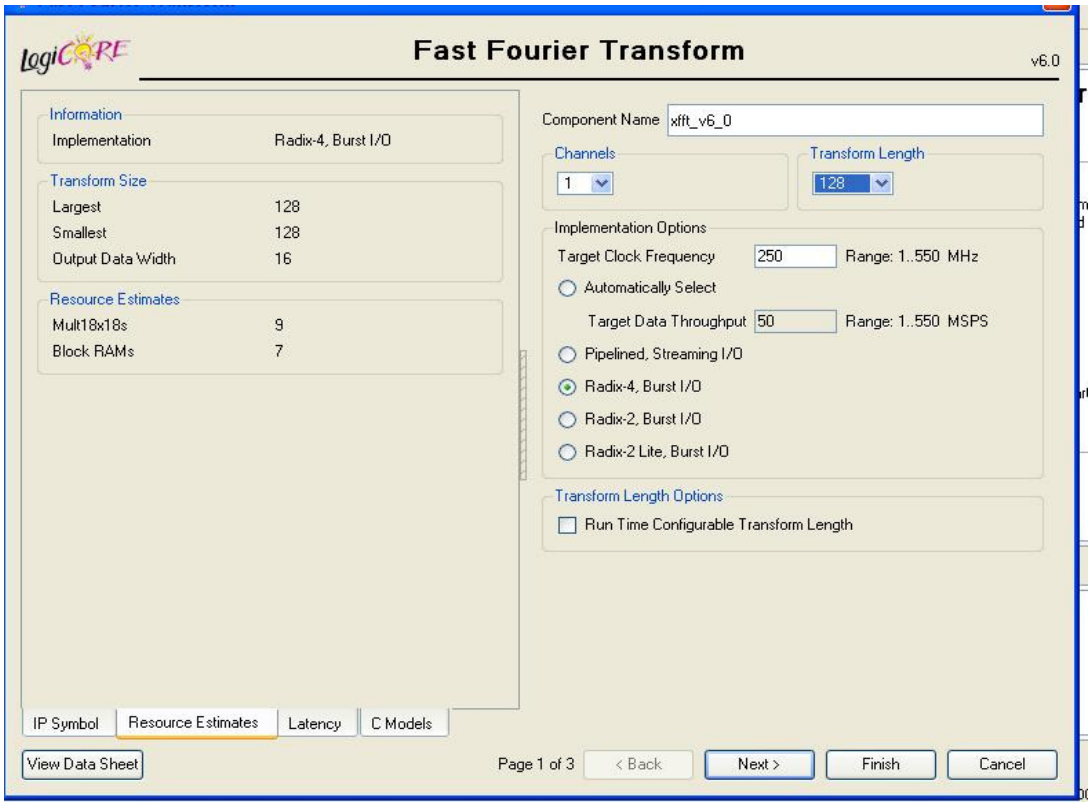
This is a circuit whose functionality is simply generate the initial pulse in the line gls_reset one clock wide so as to reset all system circuitry.

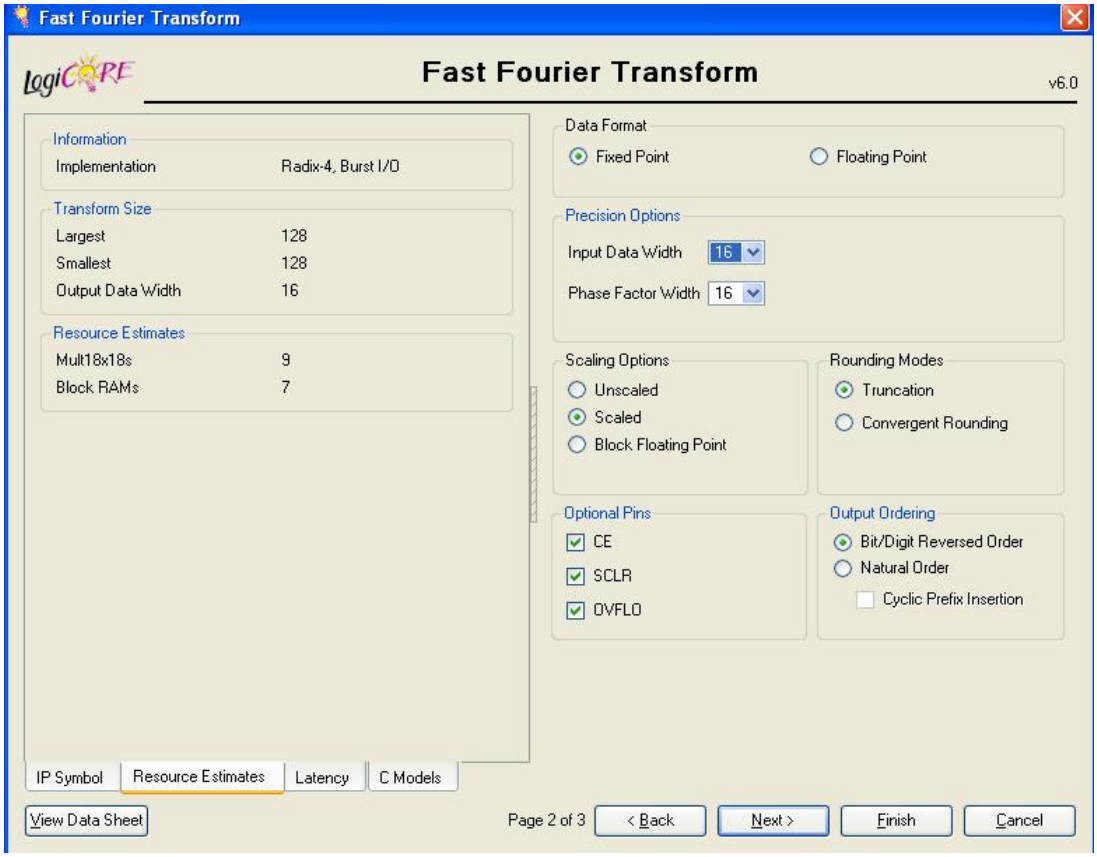
4.2.5.11 FFT IP-Core

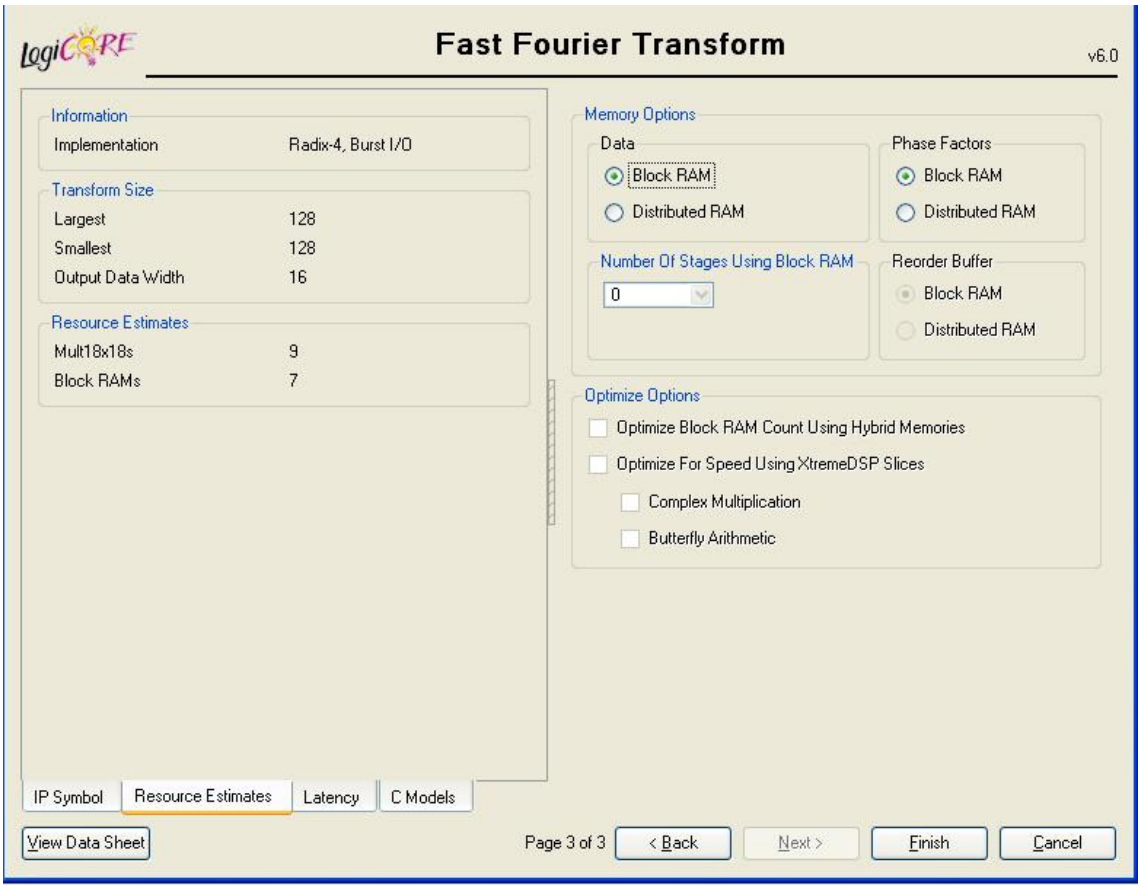
For FFT computation is used an free available Xilinx IP-Core. The IP-Cores are managed from within the ISE enviroment . To include an IP-Core into the project design , from Project menu option , select new source and finally select IP (CORE Generator & Architecture Wizard), then the CoreGenerator application is lauched. Some librarys are available, with some components inside. In this case we select the FFT as shown bellow.



The wizard allows to configure the parameters to define the FFT algoritm , points, and more the parameters entered in the screenshoots:


Fast Fourier Transform
v6.0
Information
Implementation Radix-4, Burst I/O
Transform Size
Largest 128
Smallest 128
Output Data Width 16
Resource Estimates
Mult18x18s 9
Block RAMs 7
Component Name xfft_v6_0
Channels 1
Transform Length 128
Implementation Options
Target Clock Frequency 250 Range: 1..550 MHz
Automatically Select
Target Data Throughput 50 Range: 1..550 MSPS
Pipelined, Streaming I/O
Radix-4, Burst I/O
Radix-2, Burst I/O
Radix-2 Lite, Burst I/O
Transform Length Options
Run Time Configurable Transform Length
IP Symbol
Resource Estimates
Latency
C Models
View Data Sheet
Page 1 of 3
< Back
Next >
Finish
Cancel


Fast Fourier Transform
Fast Fourier Transform
v6.0
Information
Implementation Radix-4, Burst I/O
Transform Size
Largest 128
Smallest 128
Output Data Width 16
Resource Estimates
Mult18x18s 9
Block RAMs 7
Data Format
Fixed Point
Floating Point
Precision Options
Input Data Width 16
Phase Factor Width 16
Scaling Options
Unscaled
Scaled
Block Floating Point
Rounding Modes
Truncation
Convergent Rounding
Optional Pins
CE
SCLR
OVFLO
Output Ordering
Bit/Digit Reversed Order
Natural Order
Cyclic Prefix Insertion
IP Symbol
Resource Estimates
Latency
C Models
View Data Sheet
Page 2 of 3
< Back
Next >
Finish
Cancel



For this application the option and parameters selected are:

- Number of Points: 32
- Architecture: Radix-2 Burst I/O
- Input Data Width: 16 bits data bus. Fixed Point , real data signal. So the imaginary part must be set to '0' this is done by VHDL code, connecting all the XN_IM data lines to '0'.
- Scaling: None. The data samples will be 8 bits so in the worst cas then output samples wil be of 8 +5 +1 = 14 bits (see datasheet Finite Word Considerations)
- Output natural Order
- Optional Pins :CE,SCLR,OVFL
- Tipe of RAM used : Bloq RAM

(see datasheet).

As we see 128 points is the lenght selected, but the IP suports up to 2**16 points.

Once the parameters are selected the IP-Core is generated , so that it can be included in the project design.

4.2.5.12 FFT IP-Core Description and Configuration

FFT IP-Core computes an N-point forward DFT or inverse DFT (IDFT) where N can be 2m, m = 3–16. For fixed-point inputs, the input data is a vector of N complex values represented as dual bx-bit two's-com-plement numbers, that is, bx bits for each of the real and imaginary components of the data sample, where bx is in the range 8 to 34 bits inclusive. Similarly, the phase factors bw can be 8 to 34 bits wide. For single-precision floating-point inputs, the input data is a vector of N complex values represented as dual 32-bit floating-point numbers with the phase factors represented as 24- or 25-bit fixed-point numbers. All memory is on-chip using either block RAM

or dis-tributed RAM. The N element output vector is repre-sented using by bits for each of the real and imaginary components of the output data. Input data is presented in natural order and the output data can be in either natural or bit/digit reversed order. The complex nature of data input and output is intrinsic to the FFT algo-rithm, not the implementation. Three arithmetic options are available for computing the FFT: • Full-precision unscaled arithmetic • Scaled fixed-point, where the user provides the scaling schedule • Block floating-point (run-time adjusted scaling) The point size N, the choice of forward or inverse trans-form, the scaling schedule and the cyclic prefix length are run-time configurable. Transform type (forward or inverse), scaling schedule and cyclic prefix length can be changed on a frame by frame basis. Changing the point size resets the core. Four architecture options are available: Pipelined, Streaming I/O, Radix-4, Burst I/O, Radix-2, Burst I/O, and Radix-2 Lite, Burst I/O. For detailed information about each architecture, see the IPCore Datasheet.

4.3 Linux Driver

To make easier the design procedure and use available code will break down the functionality in two parts:

- Interruption manager driver.
- Specific functionality driver.

Both linux drivers are implemented as a platform device driver , known as a driver for a device that is located at the same system.board, not connected through and extension bus.

There is a defined structure and an API, very used in the LINUX comunity to implement this type of drivers on which I have based this implementation .

So I will refer to this API, explaining the driver code.

The implemenation has bee made in two modules in files :

- board_*.c : Initialization and registration of plataform device for the specific board (pins , interrupts and memory resurces) the module will be called fft_ocore
- *.c : Contains de driver actual code , for the virtual device.

A platform device is represented by the struct platform_device, which, like the rest of the relevant declarations, can be found in <linux/platform_device.h>. These devices are deemed to be connected to a virtual "platform bus"; drivers of platform devices must thus register themselves as such with the platform bus code. This registration is done by way of a platform_driver structure:

```
struct platform_driver {
    int (*probe)(struct platform_device *);
    int (*remove)(struct platform_device *);
    void (*shutdown)(struct platform_device *);
    int (*suspend)(struct platform_device *, pm_message_t state);
    int (*resume)(struct platform_device *);
    struct device_driver driver;
    const struct platform_device_id *id_table;
  };
```

At a minimum, the probe() and remove() callbacks must be supplied; the other callbacks have to do with power management and should be provided if they are relevant.

4.3.1 Interruption manager driver

This driver is who receive the interruption signal that generates the IRQ Manager circuit, that is defined in the board file as `ARMADEUS_FPGA_IRQ`, and process it.

The interruption processing is generic for any application and does the followin g things (see ocore_irq_mng_interrupt function):

1- Acknowledges The Interruption

2- Find out the interruption cause , as per the bit number set in IRQ_PENDING register and calls the corresponding handler for the interruption number , that is

IRQ_FPGA+nbit.

So this driver acts as an interrupt multiplexer.

When a interrupt is registered the kernel stores a descritor including the handler. This information can be retrieved by the calling Irq_to_desc kernel function.

So, To simulate an interrupt by calling the handler is done calling the handle_irq member and sending the irq number .

```
desc = irq_to_desc(irq);
desc->handle_irq(irq, desc);
```

The registration is done by the specific driver probe() function.

The driver code has to be modified only in regard with the number of interrupts covered.

4.3.2 Specific Functionality Driver

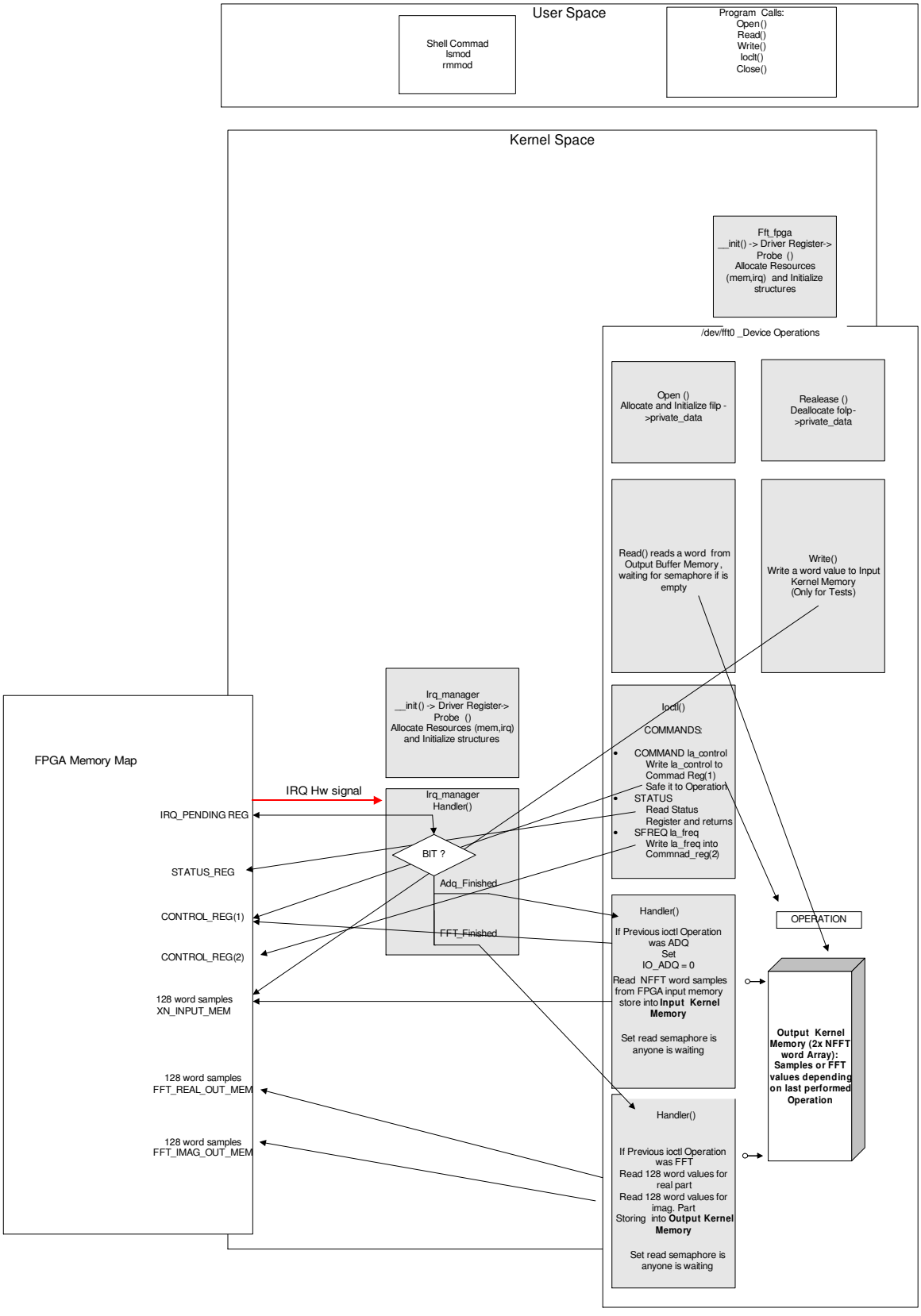
The implemenation has bee made in two modules in files :

- board_fpga.c : Initialization and registration of plataform device for the specific board (pins , interrupts and memory resurces) the module will be called fft_ocore
- gfft.c : Contains de driver actual code , for the virtual device. The module will be called fwb_example_fft

As every device driver the one developed in this project has three sides: one side talks to the the kernel, other talks to the hardware, and the user one talks to the user throught the device file:
So The /dev/fft0 device file is the only interface from the user ponit of view to talk to the driver.
When the user opens /dev/fft0, the kernel calls the fft driver open routine
When the user closes /dev/fft, the kernel calls fft's release routine . When the user reads or writes from or to /dev/fft0 , kernel calls fft's read and write functions, and finally when the user calls ioctl () calls the ioctl device kernel driver.

4.3.3 Kernel Drivers Block Diagram

Next page is a simplified diagram of both drivers irq_manager and fft_fpga together with the FPGA and User space intercations.
In the next section each function of the specific driver is dicused in detail.


User Space
Shell Commad
lsmod
rmmod
Program Calls:
Open()
Read()
Write()
Ioclt()
Close()
Kernel Space
Fft_fpga
__init() -> Driver Register-> Probe ()
Allocate Resources (mem,irq) and Initialize structures
/dev/fft0 _Device Operations
Open ()
Allocate and Initialize filp ->private_data
Realease ()
Deallocate folp ->private_data
Read() reads a word from Output Buffer Memory , waiting for semaphore if is empty
Write()
Write a word value to Input Kernel Memory
(Only for Tests)
Irq_manager
__init() -> Driver Register-> Probe ()
Allocate Resources (mem,irq) and Initialize structures
Ioctl()
COMMANDS:
COMMAND la_control
Write la_control to Commad Reg(1)
Safe it to Operation
STATUS
Read Status Register and returns
SFREQ la_freq
Write la_freq into Commnad_reg(2)
FPGA Memory Map
IRQ Hw signal
Irq_manager
Handler()
IRQ_PENDING REG
BIT ?
STATUS_REG
Adq_Finished
CONTROL_REG(1)
FFT_Finished
CONTROL_REG(2)
Handler()
If Previous ioctl Operation was ADQ
Set
IO_ADQ = 0
Read NFFT word samples from FPGA input memory store into Input Kernel Memory
Set read semaphore is anyone is waiting
OPERATION
128 word samples
XN_INPUT_MEM
Output Kernel Memory (2x NFFT word Array): Samples or FFT values depending on last performed Operation
128 word samples
FFT_REAL_OUT_MEM
128 word samples
FFT_IMAG_OUT_MEM
Handler()
If Previous ioctl Operation was FFT
Read 128 word values for real part
Read 128 word values for imag. Part
Storing into Output Kernel Memory
Set read semaphore is anyone is waiting

4.3.4 Platform Device Framework implementation

The Platform Device Bus framework consist of an API and a defined funtions set to be implemented.
fft driver talks to the kernel in its initialization function, through register_chrdev.
Also, the driver must provide a way for the bus code to bind actual devices to the driver. To do that , is defined this static structure in board_fft.c file:

```
static struct platform_device plat_fft0_device = {
      .name       = "fft",
      .id         = 0,
      .dev        = {
            .release    = plat_fft_release,
            .platform_data    = &plat_fft0_data
      },
      .num_resources    = ARRAY_SIZE(fft0_resources),
      .resource   = fft0_resources,
};
```

The above Platform_data structure field contains some driver data This is the inicialitation data for the fft driver :

```
static struct plat_fft_port plat_fft0_data = {
      .name       = "FFT0",
      .num        = 0,
      .idnum            = 3,
      .idoffset   = 0x00 * (16 / 8)
};
```

With this setup, any device identifying itself as "fft" will be bound to this driver. At a minimum, the probe() and remove() callbacks must be supplied; the
other callbacks have to do with power management and should be provided if they are relevant.
Platform drivers make themselves known to the kernel with:

```
   int platform_driver_register(struct platform_driver *driver);
```

As soon as this call succeeds, the driver's probe() function can be called with new devices. That function gets as an argument a platform_device pointer describing the device to be instantiated:

```
    struct platform_device {
      const char  *name;
      int         id;
      struct device     dev;
      u32         num_resources;
      struct resource   *resource;
      const struct platform_device_id     *id_entry;
      /* Others omitted */
    };
```

The resource array can be used to learn where various resources, including memory-mapped I/O registers and interrupt lines, can be found. There are a number of helper functions for getting data out of the resource array; these include:

```
   struct resource *platform_get_resource(struct platform_device *pdev,
                                unsigned int type, unsigned int n);
    struct resource *platform_get_resource_byname(struct platform_device
*pdev,
                                unsigned int type, const char *name);
```

```
    int platform_get_irq(struct platform_device *pdev, unsigned int n);
```

The "n" parameter says which resource of that type is desired, with zero indicating the first one.
Assuming the probe() function finds the information it needs, it should verify the device's existence to the extent possible, register the "real" devices associated with the platform device, and return zero.
So now we have a driver for a platform device, but no actual devices yet. As was noted at the beginning, platform devices are inherently not discoverable, so there must be another way to tell the kernel about their existence. That is typically done with the creation of a static platform_device structure providing, at a minimum, a name which is used. This is the definition for the fft driver in board_fft.c file.

```
#define FFT0_IRQ   IRQ_FPGA(0)

static struct resource fft0_resources[] = {
      [0] = {
            .start      = ARMADEUS_FPGA_BASE_ADDR ,
            .end  = ARMADEUS_FPGA_BASE_ADDR  + 0x400 + 8 ,   /* 128
points FFT */
            .flags      = IORESOURCE_MEM,
      },
      [1] = {
            .start      = FFT0_IRQ,
            .end  = FFT0_IRQ,
            .flags      = IORESOURCE_IRQ,
      }
};
static struct platform_device plat_fft0_device = {
      .name       = "fft",
      .id         = 0,
      .dev        = {
            .release    = plat_fft_release,
            .platform_data    = &plat_fft0_data
      },
      .num_resources    = ARRAY_SIZE(fft0_resources),
      .resource   = fft0_resources,
};
```

These declarations describe a "fft" device with a 0x408 bytes long region starting at ARMADEUS_BASE_ADDRESS and using FFT0_IRQ interrupt number .
The device is made known to the system with:

```
  int platform_device_register(struct platform_device *pdev);
```

Once both a platform device and an associated driver have been registered, the driver's probe() function will be called and the device will be instantiated. Registration of device and driver are usually done in different places and can happen in either order. A call to platform_device_unregister() can be used to remove a platform device.
The driver also needs aditional information. The mechanism used to pass this information is called "platform data"; in short, one defines a structure containing the specific information needed and passes it in the platform device's dev.platform_data field.
With the fft driver the data is passsed , in the above platform_device, in the field .platform_data that is defined as

```
static struct plat_fft_port plat_fft0_data = {
      .name       = "FFT0",
```

```
      .num        = 0,
      .idnum            = 3,
      .idoffset   = 0x00 * (16 / 8)
};
```

When the driver's probe() function is called, it can fetch the platform_data pointer and use it to obtain the rest of the information it needs.
All functionality implemenation are located at the gfft.c file and discused in the next sections.

4.3.4.1 Init and exit functions

Init funtion is the first called when the module is installed , it only registers the driver. Exit is colled upon the uninstallation and unregisters the driver.

```
static int __init board_fft_init(void)
{
      return platform_device_register(&plat_fft0_device);
}
static void __exit board_fft_exit(void)
{
      platform_device_unregister(&plat_fft0_device);
}
module_init(board_fft_init);
module_exit(board_fft_exit);
```

4.3.4.2 Probe function

This model includes the device_driver struct which defines methods to implement operations that the core calls to perform device specific actions.

This function is called from device_register funtcion.It should verify that hardware is correct and can sucessfully initiated, it also makes all initialization and device and interrupts registration.

It initializes the device platform structure , defined in fpga_fft.c recived as argument.

```
/* platform device */
struct plat_fft_port {
      const char *name; /* instance name */
      int num;          /* instance number */
      int idnum;        /* identity number */
      int idoffset;           /* identity relative address */
      struct fft_dev *sdev;  /* struct for main device structure */
};
```

The actions are :

- Check platform data
- Get resorces information (Irq,mem)
- Allocate virtual memory for I/O (request_mem_region()) and make it visible to kernel (ioremap()), in order to be accesible through readw() and writew() functions.
- Allocate memory for *sdev structure that point to the structure fft_dev defined in gfft.c

```
      struct fft_dev {
      char *name;       /* name of the instance */
      struct cdev cdev; /* char device structure */
      struct semaphore sem;   /* mutex */
      void *membase;          /* base address for instance  */
      dev_t devno;            /* to store Major and minor numbers */
```

```
      u8 read_in_wait;  /* user is waiting for value to read */
      struct resource *mem_res;
      struct resource *irq_res;
        u16 buf[FFT_POINTS*2];
        u16 rp;
        u16 wp;
      u16 nsample;           /* sample number to be writed in fpga */
      u16 operation;        /* operation type */
};
```

– Initialization of cdev char device structure with the operations suported, and their callback funtions.

```
      cdev_init(&sdev->cdev, &fft_fops);
      sdev->cdev.owner = THIS_MODULE;
      sdev->cdev.ops = &fft_fops;

      pr_debug("%s: Add the device to the kernel, "
             "connecting cdev to major/minor number \n", pdata->name);
      ret = cdev_add(&sdev->cdev, sdev->devno, 1);
      if (ret < 0) {
            dev_warn(&pdev->dev, "%s: can't add cdev\n", pdata->name);
            goto out_cdev_free;
      }
```

– Create and initalize other particular fields (mutex...)

– Obtains device file numbers and connect them to the char device

– Register interrupt handler

The ccomplete code is shown below.

```
static int fft_probe(struct platform_device *pdev)
{
      struct plat_fft_port *pdata = pdev->dev.platform_data;
      int ret = 0;
      int fft_major, fft_minor;
      u16 ip_id;
      struct fft_dev *sdev;
      struct resource *mem_res;
      struct resource *irq_res;

      pr_debug("fft probing\n");
      pr_debug("Register %s num %d\n", pdata->name, pdata->num);

      if (!pdata) {
            dev_err(&pdev->dev, "Platform data required !\n");
            return -ENODEV;
      }


      /* get resources */
      mem_res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
      if (!mem_res) {
            dev_err(&pdev->dev, "can't find mem resource\n");
            return -EINVAL;
      }
      irq_res = platform_get_resource(pdev, IORESOURCE_IRQ, 0);
      if (!irq_res) {
            dev_err(&pdev->dev, "can't find irq resource\n");
            return -EINVAL;
```

```
      }

      mem_res =
          request_mem_region(mem_res->start, resource_size(mem_res),
                             pdev->name);
      if (!mem_res) {
            dev_err(&pdev->dev, "iomem already in use\n");
            return -EBUSY;
      }

      /* allocate memory for private structure */
      sdev = kmalloc(sizeof(struct fft_dev), GFP_KERNEL);
      if (!sdev) {
            ret = -ENOMEM;
            goto out_release_mem;
      }

      sdev->membase = ioremap(mem_res->start, resource_size(mem_res));
      if (!sdev->membase) {
            dev_err(&pdev->dev, "ioremap failed\n");
            ret = -ENOMEM;
            goto out_dev_free;
      }
      sdev->mem_res = mem_res;
      sdev->irq_res = irq_res;

      /* check if ID is correct */
      /*
      ip_id = readw(sdev->membase + fft_ID_OFFSET); */
        ip_id = pdata->idnum;
      if (ip_id != pdata->idnum) {
            ret = -ENODEV;
            dev_warn(&pdev->dev, "For %s id:%d doesn't match "
                   "with id read %d,\n is device present ?\n",
                   pdata->name, pdata->idnum, ip_id);
            goto out_iounmap;
      }

      pdata->sdev = sdev;

      sdev->name = (char *)kmalloc((1 + strlen(pdata->name)) *
sizeof(char),
                             GFP_KERNEL);
      if (sdev->name == NULL) {
            dev_err(&pdev->dev, "Kmalloc name space error\n");
            goto out_iounmap;
      }
      if (strncpy(sdev->name, pdata->name, 1 + strlen(pdata->name)) < 0)
{
            printk("copy error");
            goto out_name_free;
      }

      /* Get the major and minor device numbers */
      fft_major = 251;
      fft_minor = pdata->num;

      sdev->devno = MKDEV(fft_major, fft_minor);
      ret = alloc_chrdev_region(&(sdev->devno), fft_minor, 1, pdata-
>name);
      if (ret < 0) {
            dev_warn(&pdev->dev, "%s: can't get major %d\n",
```

```
                  pdata->name, fft_major);
            goto out_name_free;
      }
      dev_info(&pdev->dev, "%s: MAJOR: %d MINOR: %d\n",
             pdata->name, MAJOR(sdev->devno), MINOR(sdev->devno));

      /* initiate mutex locked */
      sdev->read_in_wait = 0;
      sema_init(&sdev->sem, 0);

      /* Init the cdev structure  */
      cdev_init(&sdev->cdev, &fft_fops);
      sdev->cdev.owner = THIS_MODULE;
      sdev->cdev.ops = &fft_fops;

      pr_debug("%s: Add the device to the kernel, "
             "connecting cdev to major/minor number \n", pdata->name);
      ret = cdev_add(&sdev->cdev, sdev->devno, 1);
      if (ret < 0) {
            dev_warn(&pdev->dev, "%s: can't add cdev\n", pdata->name);
            goto out_cdev_free;
      }

      /* irq registering */
      ret = request_irq(sdev->irq_res->start,
                  fft_interrupt,
                  IRQF_SAMPLE_RANDOM, sdev->name, sdev);
      if (ret < 0) {
            printk(KERN_ERR "Can't register irq %d\n",
                   sdev->irq_res->start);
            goto request_irq_error;
      }

      /* OK driver ready ! */
      printk(KERN_INFO "%s loaded\n", pdata->name);
      return 0;

      free_irq(sdev->irq_res->start, sdev);
request_irq_error:
      cdev_del(&sdev->cdev);
      pr_debug("%s:cdev deleted\n", pdata->name);
out_cdev_free:
      unregister_chrdev_region(sdev->devno, 1);
      printk(KERN_INFO "%s: fft deleted\n", pdata->name);
out_name_free:
      kfree(sdev->name);
out_iounmap:
      iounmap(sdev->membase);
out_dev_free:
      kfree(sdev);
out_release_mem:
      release_mem_region(mem_res->start, resource_size(mem_res));

      return ret;
}
```

4.3.4.3 Open funtion

The open function is called by the kernel when the device file is open from the user space.

Initializes the filp->private_data of the file descriptor pointer with a fft_dev structure.

```
int fft_open(struct inode *inode, struct file *filp)
{
        struct fft_dev *dev;

      /* Allocate and fill any data structure to be put in filp-
>private_data */
      dev = filp->private_data =
          container_of(inode->i_cdev, struct fft_dev, cdev);
        dev->rp = 0;
        dev->wp = 0;
        dev->nsample = 0;
      return 0;
}
```

4.3.4.4 Relase funtion

The release function is called by the kernel when the device file is closed from the user space.

It realises the filp->private_data of the file descriptor pointer.

```
int fft_release(struct inode *inode, struct file *filp)
{
      struct fft_dev *dev;
      dev = container_of(inode->i_cdev, struct fft_dev, cdev);
      filp->private_data = NULL;
      return 0;}
```

4.3.4.5 Read funtion

The read function is called by the kernel when device file is read from user space. It takes data from the buffer in the fft_dev structure storing values and copy them to the user space memory. The in the read buffer was stored by the interrupt handler function.

```
ssize_t fft_read(struct file *fildes, char __user * buff,
                 size_t count, loff_t * offp)
{
      struct fft_dev *ldev = fildes->private_data;
      u16 data = 0;
      /* char __user * la_addr; */
      ssize_t retval = 0;
      /* u16 la_indx; */
      if (*offp + count >= 1) count = 1 - *offp;
                 if ( ldev->rp >= ldev->wp){
         ldev->read_in_wait = 1;
         if ((retval = down_interruptible(&ldev->sem)) < 0) {
            goto out;
         }
        }
        printk("Acceso Read pos: %d \n ", ldev->rp);
        data = ldev->buf[ldev->rp]; /* read data from read buffer */
        if ( ldev->rp >= FFT_POINTS*2 ) /* Only FFT_POINTS*2 can be read
*/
        {    retval = 0;
                goto out;
        }
      /* return data for user */
      if (copy_to_user(buff, &data, 2)) {
            printk(KERN_WARNING "read : copy to user data error\n");
            retval = -EFAULT;
            goto out;
      }
```

```
      ldev->rp = ldev->rp+1;
      retval = 1;
out:
      ldev->read_in_wait = 0;
      return retval;
}
```

4.3.4.6 Ioctl funtion

The ioctl function is called by the kernel when ioctl function is called with the device file as the first argument from user space.

Depending on the seccond argument , it does different things. The two implemented functions are:

FFT_IOCCOMMAND :

reads the third argument and writes it to the FPGA control register.

Stores the value en operation variable. In order to take it into account when an interrupt arrives.

FFT_IOCSTATUS:

reads status register and returns it to the user space.

See below the source code.

```
int fft_ioctl(struct inode *inode, struct file *fildes,
                 unsigned int cmd, unsigned long arg)
{
      int retval = 0;
      u16 la_control,la_status;
      struct fft_dev *ldev = fildes->private_data;
      /* extract the type and number bitfields, and don't decode
       * wrong cmds: return ENOTTY (inappropriate ioctl) before
access_ok() */
      if (_IOC_TYPE(cmd) != FFT_IOC_MAGIC) return -ENOTTY;
      if (_IOC_NR(cmd) > FFT_IOC_MAXNR) return -ENOTTY;
      switch(cmd) {
        case FFT_IOCCOMMAND:
            retval = __get_user ( la_control,( u16 __user *)arg);
                    writew(la_control ,ldev->membase + FFT_COMMAND_REG);
                  ldev -> operation = la_control;
            break;
        case FFT_IOCSTATUS: /* Set: arg points to the value */
            la_status = readw(ldev->membase + FFT_STATUS_REG);
                  retval = __put_user (la_status,(u16 __user *)arg);
            break;
        default:  /* redundant, as cmd was checked against MAXNR */
            return -ENOTTY;
      }
      return retval;
}
```

4.3.4.7 fft_interrupt handler function

This funtion is called from kernel when the configured interrupt line is asserted from the FPGA circuit. The interruption is first intercepted by the irq_mngr interrupt handler which examines the interrupt register ISR in the FPGA irq_mngr module , to find out the corresponding bit. Once is determined the bit it send the interruption to this handler, with the argument irq , from which the bit can be found, (i.e. Bit 1 with irq= 256 and bit2 with irq = 257).

As can be seen, it does different things depending on the value stored in the operation variable. it reads from corresponding FPGA memory space FFT nesults (real and imag parts) if operation is OP_CALC, or Adquistion valuies if operation is OP_ADQ , them stores data in the buffer in order to be availabe to read from the user space , through a read call.

```
static irqreturn_t fft_interrupt(int irq, void *dev_id)
{
      u16 la_indx;
      struct fft_dev *ldev = dev_id;
/* 1 Dic 2013 */
        unsigned int mask;
        unsigned int lbit;
      struct irq_desc *desc;
        mask = readw(ldev->membase + FPGA_ISR);
      /*irq = IRQ_FPGA_START*/;

#if LINUX_VERSION_CODE > KERNEL_VERSION(2,6,29)
      desc = irq_to_desc(irq);
#else
      desc = irq_desc + irq;
#endif
      /* handle irqs */
      printk ("handling irq %d 0x%08x\n", irq,
                              (unsigned int)desc);
        lbit = 0;
      while (mask) {
            if (mask & 1) {

                  if ( lbit == 0) {
                       printk("fft driver : Interruption bit %d
\n",lbit);
                           if ( ldev-> operation == FFT_OP_CALC ){
                              printk("fft driver : Cal_Op_Results \n");
                              ldev->wp = 0;
                              ldev->rp = 0;
                        for ( la_indx = 0; la_indx < FFT_POINTS; la_indx
=la_indx+1){
                             ldev->buf[la_indx] = readw(ldev->membase +
FFT_READDATA_OFFSET+ ldev->wp);
                           ldev->wp = ldev->wp+1;
                           ldev->buf[la_indx+1] = readw(ldev->membase +
FFT_READDATA_OFFSET+0x100+ldev->wp);
                           ldev->wp = ldev->wp+1;
                          }

                        if (ldev->read_in_wait)
                        up(&ldev->sem);
                           }
                  }
                        else if ( lbit == 1){
                       printk("fft driver : Interruption bit %d
\n",lbit);
                           if ( ldev-> operation == FFT_OP_ADQ ){
                              printk("fft driver : Signal_Adq \n");
                              ldev->wp = 0;
                              ldev->rp = 0;
                        for ( la_indx = 0; la_indx < FFT_POINTS; la_indx
=la_indx+1){
```

```
                            ldev->buf[la_indx+1] = readw(ldev->membase +
FFT_READDATA_OFFSET+0x200+ldev->wp);
                            ldev->wp = ldev->wp+1;
                           }

                        if (ldev->read_in_wait)
                        up(&ldev->sem);
                            }

                    }
                  }
              lbit++;
              mask >>= 1;
        }
          /* Clear interrupts */
        writew(0xffff, ldev->membase + FPGA_ISR);
        return IRQ_HANDLED;
}
```

4.3.4.8 write function

The write function is called by kernel when write is called from user space. It copies the data from user space and write them to the FPGA memory space.

```
ssize_t fft_write(struct file *fildes, const char __user * buff,
                  size_t count, loff_t * offp)
{
           u16 data =0;
           ssize_t retval = 0;
           struct fft_dev *ldev = fildes->private_data;
           if ( copy_from_user (&data, buff,2)) {
           retval = -EFAULT;
           printk (" Error\n");
           goto out;
        }
        printk (" Write sample %d data %d\n",ldev->nsample,data);
        writew(data,ldev->membase + FFT_WRITEDATA_OFFSET+ldev->nsample);
        ldev->nsample = ldev->nsample + 2;
        retval = 2;
out:
        return retval;
}
```

4.3.5 Driver compilation

1) variables de entorno

>cd armadeus-4.1

>sudo sh ./armadeus_env.sh

2) Configuración kernel : Hay que incluir los drivers

>make linux-menuconfig

Device Drivers-> Armadeusspecific drivers->FPGA Drivers

**** Specific Designs ****

* Virtual components *

marcar fft como módulo (M)

en seccion

- Board Designs *

  Marcar fpga fft como módulo (M)

Para compilar los modulos

>Make -C target/linux/modules

o bien

>make linux26

esto ultimo tambien instala los modulos en rootfs

y por ultimo

>make

copy images from directory buildroot/output/images to /tftpboot

From a terminal attached to Apf27

BIOS> run update_kernel

BIOS> run update_rootfs

Boot the system

BIOS>boot

4.3.6 Driver Installation

Once started Linux , to install module

$ modprobe fft_ocore

$ modprobe wb_example_fft

several messages are shown , between them the important is

fft fft.0: MAJOR: 249 MINOR 0

to check if the driver is loaded
$lsmod

Module	Size	Used by	Not tainted
wb_example_fft	1272	0	
wb_example_fft	6580	0	

Finally create the device file with the mayor and minor number shown in the modprobe command.

mknod /dev/fft0 c 249 0

At this point the driver is installed and ready to use.

4.3.7 Driver Debuging

To see interrupts installed and the number of interrupts :

cat /proc/interrupts

To see the printk messages , first enable kernel messages
$echo 8 > /proc/sys/kernel/printk

then see the system messages
$dmesg

4.3.8 Driver operation and programming

This driver will create one device file /dev/fft0 when installed.

Fft0 device can be write or read.

This file supports ioctl, read and write operations.

The ioctl commands suported are:

FFT_IOCOMMAND: This ioctl command will take the second argument that should be and integer number and writes it to the FPGA command register.

FFT_IOSTATUS : This ioctl command will read the FPGA status register and return the value to the program.

As was said above The FFT_IOCOMMAD will write the the second argument value to the control register, this output register is conected to other circuits. The pin asignements are as follows:

Bit 0 = start_fft starts FFT IP Core calculations

Bit 1 = start_adq starts adquisition process

Bit 2 = unloadfft tells the IP Core to unload results when available.

Bit 3 = sclr_fft Reset IPCore

Bit 4 = Fwd_Inv_FFT Selects Fordward or invers FFT

Bit 5 = Fwd_Inv_we Writes Fwd_Inv_FFT

Basically Two Operations Can be done:

1) Signal Adquisition
2) FFT Calculation

Signal Adquition

To Start The Signal adquisition process the following ioctl command must be assserted

ioctl FFT_IOCOMMAND 2

So the bit 1 is set

when this command is asserted , the FPGA Equalizer IPCore logic starts the adquisition process , storing the time domain samples , in the internal memory , When 128 samples have been stored an interruption is asserted for the driver to read the results and stores at fft0 device , then the semaphor is set , for the processor to read the results.

-FFT Calculation

To Start The FFT calculation following ioctl commands must be assserted

ioctl FFT_IOCOMMAND 48

ioctl FFT_IOCOMMAND 5

In the first ioctl commad the fordware FFT is selected

The second command starts the FFT calculation and unload results , when this is completed the fft values are stored in the FPGA internal memory, Then the Equalizer module asserts an interrupt , so the driver stores at fft0 device the sample values, and the semphor is set, so that the aplication can read the results.

So , as is said above the read operation , obtains time domain samples or fft results , depending on the previos ioctl operation.

The write operation, stores the values in input time domain signal samples for the FFT IPCore to compute the fft transform.

4.4 User Space Data Server Program

In order to demostrate the full device functionality , has been developed a user space test program to be run in the APF board, that implementes a dataserver . This program basically establishes a server socket, and responds to client messages , accessing the device file /dev/fft0 to get the adquisition and FFT signal data and send them to a client program throught a TCP/IP connection, when requested.

On the other hand a client has been implemented as a Java application, that will be discused in the next section..

An aplication communication protocol has been also defined and implemented for the server-client communiication. .The messages supoorted as well as the program logic is described here.The message format is also specified

The main funtion does the following actions:

It first stablishes the SIGINT signal handler to control the ctrl-c to correctly close socket and device file.

Then calls the function Inicializacion_Server_Socket , passing the port number as argument.. This fuction creates a TCP socket,initializes IP adreess and port number, then it call to bind , and eventually to listen function.

Next creates a thread calling pthread with the socket just creetd, and the pointer to Process _Socket funtion..

Finally the main function get into a infinite loop.

The Process _Socket is a process that waits for a connection calling the accept function.

When a connection is recibed , another thread is created to attend that connection.

This new created process gets into a loop in which is receiving and processing messages untill a MSG_ENDCNX end connection message is recived to exit the loop.

The aplication message format sent from the client is :

```
        short cab; /* 02 */
        short type;
        short period;  /* this fied is used for the sample period */
```

The aplication message format sent from the the server, where FFT_POINTS is defined to be 128 is :

```
        short cab;   /* 02 */
        short type;
        short status;
        short signal_samples[FFT_POINTS];
        short fft_r[FFT_POINTS]; /* fft real part */
        short fft_i[FFT_POINTS]; /* fft imag part */
```

The different message types supported are:

```
#define MSG_ADQ_FFT_REQ 0x01
#define MSG_ADQ_FFT_REP 0x02
#define MSG_ADQ_FFT_ACK 0x03
#define MSG_ENDCNX      0x04
#define MSG_ADQ_REQ     0x05
#define MSG_FFT_REQ     0x06
```

4.4.1 Message processing

the message processing are descrived next.

4.4.1.1 Message MSG_ADQ_FFT_REQ

this command for the system to send signal adquisition values and the corresponding FFT analisys

- Open device file /dev/fft0

```
ffft = open("/dev/fft0", O_RDWR);
```

- Ioctl command to make to init device (la_control = 0)

```
la_control = 0;
ioctl ( ffft, FFT_IOCCOMMAND , &la_control);
```

- Ioctl command to set the sample period using the received value

```
ioctl ( ffft, FFT_SFREQ, &la_period);
```

- Tells the device to start signal adquisition , seting the multiplexer to select memory to Ctl_Adq module conection

```
la_control=MSK_ADQ_START|MSK_ADQ_IO;
ioctl ( ffft, FFT_IOCCOMMAND , &la_control);
```

this command makes Ctl_Adq module to store in the signal memory 128 sample values , and assert an ADQ interruption so that the driver interrupt handler to take the data and copy them into kernel space memory available to read using the device file.

- read samples values from device file and stores them in the answer message structure.

```
for ( i=0; i<FFT_POINTS;i++) {
    read(ffft, &data, 1);
    msg_tx.message.signal_samples[i]= htons(data);

}
```

- assets a CLR signal to the FFT IP-Core

```
la_control = MSK_FFT_CLR;
ioctl ( ffft, FFT_IOCCOMMAND , &la_control);
```

- Sets FFT IP-Core scale factor to 0.

```
la_control = MSK_FFT_SCALE_WE;
ioctl ( ffft, FFT_IOCCOMMAND , &la_control);
```

- Ioctl command to select IP-Core to compute fordward FFT to device file, by asserting a pulse in FFT_FWD_INV and FWD_INV_WE

```
la_control = MSK_FFT_FWD_INV|MSK_FFT_FWD_INV_WE;
ioctl ( ffft, FFT_IOCCOMMAND , &la_control);
la_control = 0;
ioctl ( ffft, FFT_IOCCOMMAND , &la_control);
```

- waits for the FFT IP-Core to be ready

```
do {
    ioctl ( ffft, FFT_IOCSTATUS , &la_status);
    if (la_status & MSK_FFT_RFD == 0){
```

```
            usleep (1);
        }
    } while (( la_status & MSK_FFT_RFD)!=0);
```

- Ioctl to start FFT

```
la_control = MSK_FFT_START|MSK_FFT_UNLOAD;
ioctl ( ffft, FFT_IOCCOMMAND , &la_control);
```

this signal makes FFT IP-Core to compute the FFT and store results in output memory. Upon completion the IP Core raises the DV signal, causing a FFT interrupt to the driver, the driver reads the two output memories (for real and imag parts) and stores them in kernel memory, in such a way that real and imag parts are stored in consecutive positions, available to read using the device file

- waits for the results, testing that DV bit is set.

```
do {
      ioctl ( ffft, FFT_IOCSTATUS , &la_status);
      if ( la_status&MSK_FFT_DV==0) usleep (1);
} while (( la_status & MSK_FFT_DV)!=0);
```

- get FFT results reading values from device, taking into account the store order for real and imag parts explained above, and stores them in the answer message structure.

```
for ( i=0; i<FFT_POINTS;i=i+1) {
/* read value */
      ret = read(ffft, &j, 1);
      if ( ret < 0) {
            perror("read error \n");
            exit(EXIT_FAILURE);
      }
      else if (ret==0) {
            printf ( "fin fichero \n");
            break;
      }
      msg_tx.message.fft_r[i]= htons(j);

      ret = read(ffft, &j, 1);
      if ( ret < 0) {
            perror("read error \n");
            exit(EXIT_FAILURE);
      }
      else if (ret==0) {
            printf ( "fin fichero \n");
            break;
      }
      msg_tx.message.fft_i[i]= htons(j);
}
```

- Resets control FFT signals.

```
la_control = 0x0;
ioctl ( ffft, FFT_IOCCOMMAND , &la_control);
```

- obtains the STATUS word to send it in the answer message

```
ioctl ( ffft, FFT_IOCSTATUS , &la_status);
msg_tx.message.status = htons(la_status);
if (la_status & MSK_FFT_OVF ) printf ("FFT Overflow !\n");
```

because this IP operates in fixed point arithmethic , note that the OVF bit is set when signal values are too large , in which case results are not valid

- Close device file

```
close(ffft);
```

- Send answer message setting type MSG_ADQ_FFT_REP

```
msg_tx.message.cab = htons(STX);   /* STX */
msg_tx.message.type = htons(MSG_ADQ_FFT_REP);
write(client_sock,&msg_tx.b[0],sizeof(msg_tx));
```

- Finally it waits for the ACK message, and check is correct.

```
la_offset = 0;
la_wanted = sizeof(msg_rx);
while (la_wanted >0){
  la_bytes = read(client_sock,&msg_rx.b[la_offset],la_wanted);
  la_wanted -=la_bytes;
  la_offset +=la_bytes;
}
la_cab = ntohs (msg_rx.message.cab);
la_type= ntohs (msg_rx.message.type);
la_period = ntohs (msg_rx.message.period);
if ((la_cab == STX) &(la_type == MSG_ADQ_FFT_ACK)){
   printf ("Message ACK Received: %d, %d,
%d\n",la_cab,la_type,la_period);
}
else {
  printf ("Wrong Message Received: %d, %d,
%d\n",la_cab,la_type,la_period);

}
```

4.4.1.2 Message MSG_ADQ_FFT_REP

As we saw in the above point this message is sended from server to client in reply to a MSG_ADQ_FFT_REQ.

4.4.1.3 Message MSG_ADQ_FFT_ACK

This message is sended from client to server in reply to a MSG_ADQ_FFT_REP, to acknowledge the reception. After this message the server is ready to receive another request .

4.4.1.4 Message MSG_ADQ_REQ

This message is analogous to MSG_ADQ_FFT_REQ but it does only the part corresponding to adquisition. The reply has the same message but only the signal adquisition part is stuffed.

4.4.1.5 Message MSG_FFT_REQ

This message is analogous to MSG_ADQ_FFT_REQ but it does only the part corresponding to adquisition.The reply has the same message but only the signal FFT part is stuffed.

4.5 Java Client Application Program

The client application has been implemented in JAVA using Netbeans IDE. The project includes the following classes:

- The Client_ADQ_FFT Class is the main Class, extending JFrame swing class.
- Only this class will be analized here, as implements all functionalities but the machine-human interface that is obbious, however the code can seen in anexes.This class contains the main() procedure , that creates the Client_ADQ_FFT instance , configure the window frame and executes the runClient() function :

```
public static void main(String[] args) {
  Client_ADQ_FFT application;
  if (args.length  == 0){
      System.out.println("usage: ClientAdqFft host");
  } else{
      application = new Client_ADQ_FFT ( args[0]);
      application.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
      application.setSize(800, 700);
      application.setVisible(true);
      application.runClient();
  }
```

In the Client_ADQ_FFT construcctor, 5 Panels are created, so besides the base panel are created, this classes implement the machine-himan interface functionalities :

panel_signal, to represent de adquisition signals samples , that is, in the time domain.

Panel_fft, to represent the signal FFT

panel info , to represent information about de process

panel_buttons for the operation buttons.

```
public Client_ADQ_FFT (String arg)
{
       super ("Adquisition & FFT Analisys " );
 host = arg;
 panel = new JPanel();
 panel_signal = new Panel_ADQ();
 panel_fft = new Panel_FFT();
 panel_info = new Panel_Info();
 panel_buttons = new Panel_Buttons(this);

 panel.setLayout (new GridLayout (2,2));
```

```
        panel.add(panel_signal);
        panel.add(panel_buttons);
        panel.add(panel_fft);
        panel.add(panel_info);

        add (panel,BorderLayout.CENTER);
    }
```

The runClient () function is responsible to create the Client communications thread, that connects to the server and enters into a loop sending messages and reading/processing answers.

- memory buffers allocation and Initalization
- conection socket
- input and output streams creation

Then enters into a while loop until exit button is pressed, In which case the conection socket and input/output streams are closed.

The panel_buttons , mantains a flags to controp operation. This flags are:

- state_run : tells if messages request must be sended to the server. This flags is set when START button is presssed and reset on the reply message reception only on single mode operation, so otherwise is continuosly sendidng messages until STOP button is pressed.
- Type : select what information must be requested :

  1: Signal adquisition.

  2: last adquisition FFT values.

  3: Both Segnal adquisition and FFT values.

So if state_run is active, in base of above flags, and the sample frequency value, a request message is built.

Then the message is sent to the output socket.

Next the answer message is received from the input socket.

Finally is processed the answer message ,to populate the memory structures that are displayed at the adquisition and FFT panels.

```
public void runClient()
{
        int la_indx;
        double la_mod;
        System.out.println("Atempting Connection..");
        buf_tx = new short[3];
        buf_rx = new short[FFT_POINTS*3+3];
try {
client = new Socket (InetAddress.getByName(host),PORT);
input = new DataInputStream (client.getInputStream());
```

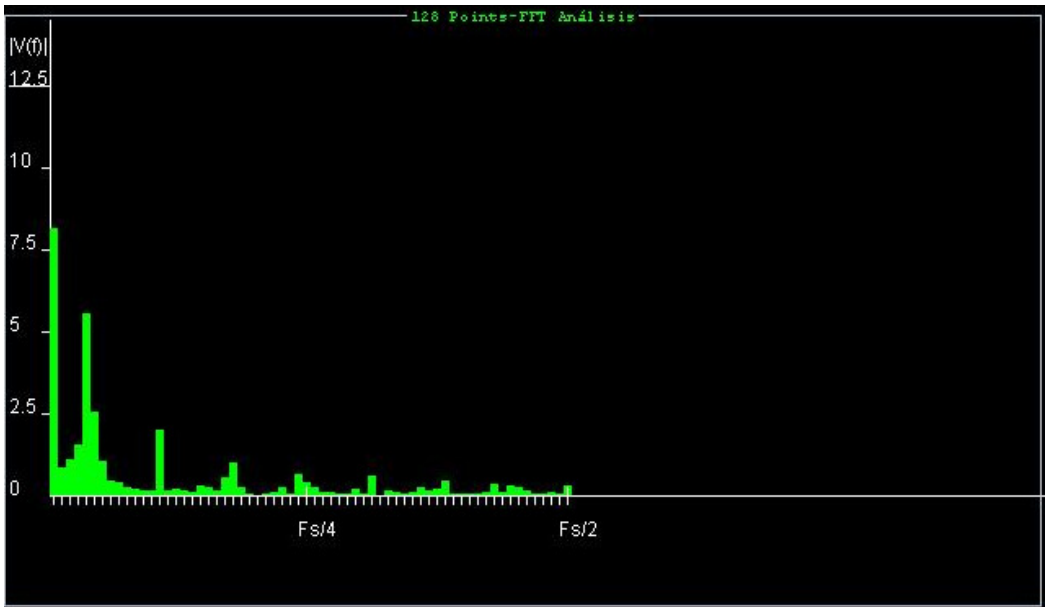
```
output = new DataOutputStream(client.getOutputStream());
System.out.println(" Connection Stablished");
while (!panel_buttons.exit){
   if ( panel_buttons.state_run){
      buf_tx [0]= STX;
      if (panel_buttons.type_op==1){
          buf_tx[1]= MSG_ADQ_REQ;
      }
      else if (panel_buttons.type_op == 2){
         buf_tx[1]= MSG_FFT_REQ;
          }
      else{
         buf_tx[1]= MSG_ADQ_FFT_REQ;
      }
      buf_tx[2]= sample_period;  /* Period = la_period*2.56uS)*/

      for (offset=0 ;offset < buf_tx.length;offset++){
         output.writeShort(buf_tx[offset]);
      }
      System.out.println(" Sent "+ buf_tx.length+" shorts ");
      wanted = buf_rx.length;
      offset =0;
      while (wanted >0){
         buf_rx[offset] = input.readShort();
         wanted -=1;
         offset +=1;
      }
      /* Answer processing */
      /* Status is the 3 word first bit  */
      if ( (buf_rx[2]& 0x0002) !=  0){
      fft_overflow = true;
      }
      else
      {
          fft_overflow =false;
      }
      for (la_indx=0; la_indx <FFT_POINTS ;la_indx++){
          panel_signal.signal_x[la_indx] = 30+la_indx;
          panel_signal.signal_y[la_indx] = 200-buf_rx[3+la_indx];
          if ( panel_buttons.log_info){
              panel_info.addInfo("Signal["+la_indx+"]="+buf_rx[3+l
```

```
a_indx]+"\n");


               }
            }
            panel_signal.sample_time
=((float)sample_period)*(float)FFT_POINTS*(float)2.56;
            boolean flip = true;


            for (la_indx=0; la_indx <FFT_POINTS/2 ;la_indx++){
               la_mod =
            (double)(Math.pow((double)buf_rx[la_indx+3+FFT_POINTS],2)+
            Math.pow((double)buf_rx[la_indx+3+2*FFT_POINTS],2) );
               la_mod = Math.sqrt(la_mod)/10;
               if ( panel_buttons.log_info){
                   if (flip){
                       panel_info.addInfo("|X|[" +la_indx+"]=" +la_mod+
" "+buf_rx[la_indx+3+FFT_POINTS]+" "+buf_rx[la_indx+3+2*FFT_POINTS]
+"\t");
                       }
                       else{
                           panel_info.addInfo("|X|[" +la_indx+"]="
+la_mod+ " "+
                           buf_rx[la_indx+3+FFT_POINTS]+"
"+buf_rx[la_indx+3+2*FFT_POINTS]+"\n");
                       }
                       flip  = !flip;
                    }
                    panel_fft.fft_x[la_indx] = 32+la_indx*5;
                    panel_fft.fft_y[la_indx] = 300-(int)(la_mod)/2;
                }
                repaint();



                /*Send ACK message */
                buf_tx [0]= STX;
                buf_tx[1] =MSG_ADQ_FFT_ACK ;

                for (offset=0 ;offset < buf_tx.length;offset++){
                   output.writeShort(buf_tx[offset]);
                }

                if (panel_buttons.single_adq){
```

```
                    panel_buttons.state_run = false;
                }
            }

        }
        /*Send END CONECTION message */
            buf_tx [0]= STX;
            buf_tx[1] = MSG_ENDCNX ;

            for (offset=0 ;offset < buf_tx.length;offset++){
               output.writeShort(buf_tx[offset]);
            }

        /* Close Connection */
            output.close();
            input.close();
            client.close();
            System.exit(0);
        }
        catch (IOException e){
            System.out.println(e);
        }
}
```

– The Panel_FFT Class implements the FFT bar graphic drawing. Is interesant to note that in order to give better visibility , only DC component and the positive half FFT results are plotted, as the other half should be simetric.

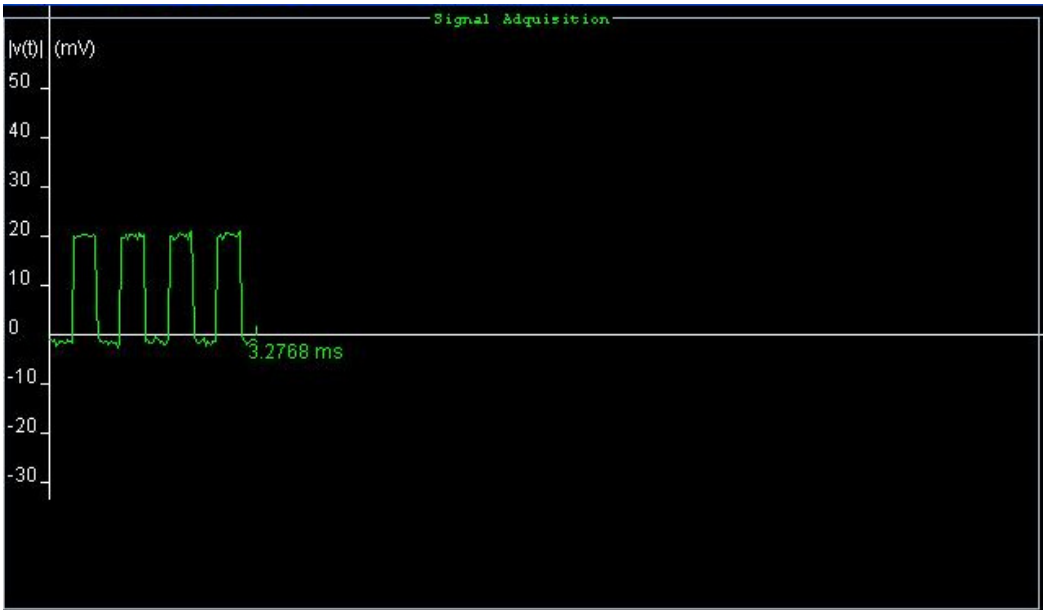
```
        for ( la_indx=0; la_indx <FFT_POINTS/2; la_indx++){
            g.setColor(Color.green);
            g.fillRect(fft_x[la_indx]-2, fft_y[la_indx], 5, 300-
fft_y[la_indx]);
            g.setColor(Color.white);
            g.drawLine(fft_x[la_indx],305,fft_x[la_indx] , 300);
}
```


128 Points-FFT Análisis
|V(f)|
12.5
10
7.5
5
2.5
0
Fs/4
Fs/2

–

– The Panel_ADQ Class implements the adquisitio graphic drawing. The 128 sample values are plotted , scale has been assigned based on an initial calibration. The time plotted corresponds to 128 sample periods.

  The code is clear and straighfordward , so no comments will be done.



– The Panel_Info , represts the parameters and operation state:
  - Adquisition period
  - Adquisition frequency
  - Operation : ADQ, FFT or ADQ&FFT
  - Overflow flag : this is the FFT IP-Core bit status
  - Operation Mode: Single or Continuous

  This panel also contais the butons to change this settings and control operation.

  - Operation type radiobutton group
  - Operation Mode radiobutton group
  - Log Debug Info checkbox
  - Start/Sop/Exit buttons
  - Sampling frequency select button

– If“Samplig Frequency” button is clicked a popup window lets you enter a value to select the desired sample frequency. As each adquisition cicle last 2.56 us, the sample period must be multiple of that time. The value entered corresponds the number of cycles one sample period.

```
int sFreq  = Integer.parseInt(input);
```

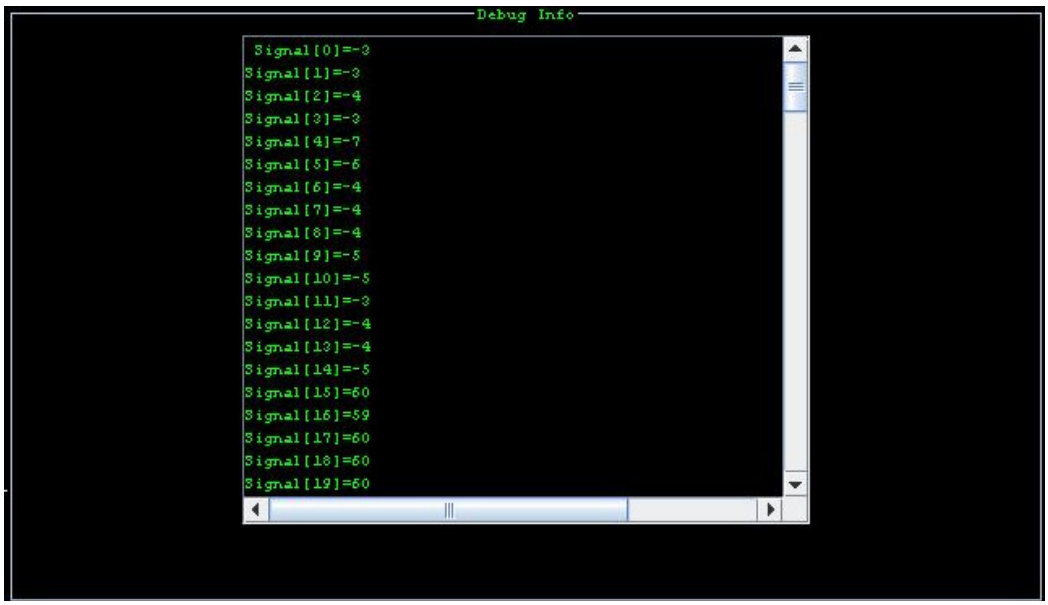
```
client.sample_period = (short)(1000000.0/((float)(sFreq)*2.56));
```

This screenshoot represents the final Panel_Info HMI :

The "Log Debug Info" checkbox tells whether to display or not data received from the server for debug purposes.

– Panel_Info Class

This class creates a scrollable Text area object to display debug information. This information will be based on the received message data. from Client_ADQ_FFT this information will be writen in case the Log_Debug_Info flag is set.



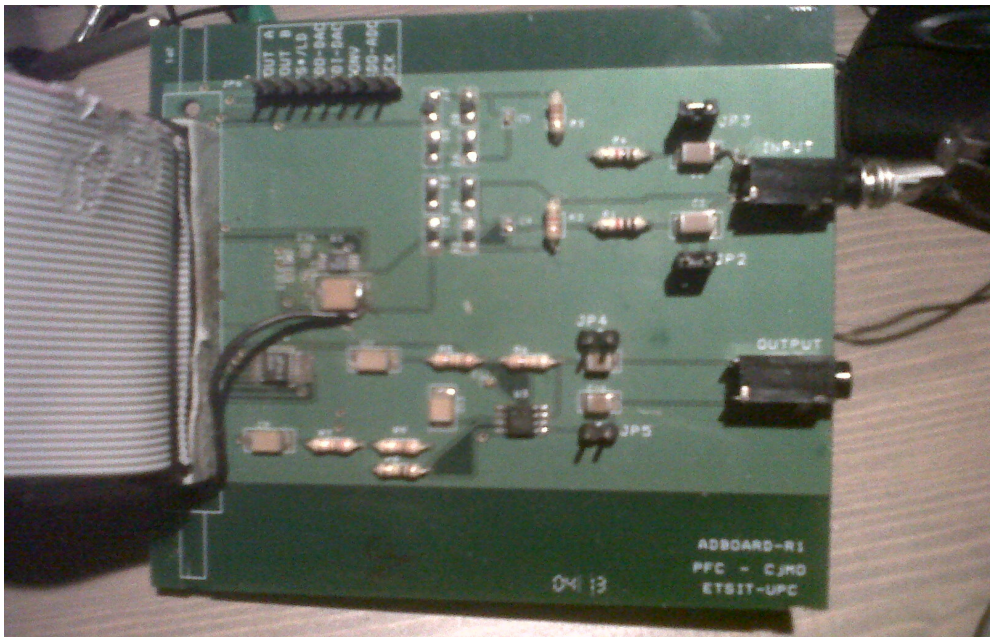
5 RESULTS

5.1 Analog I/O Board

The analog I/O Board has been designed in two layers using Orcad, the board has been also manufactured using the gerver output files.

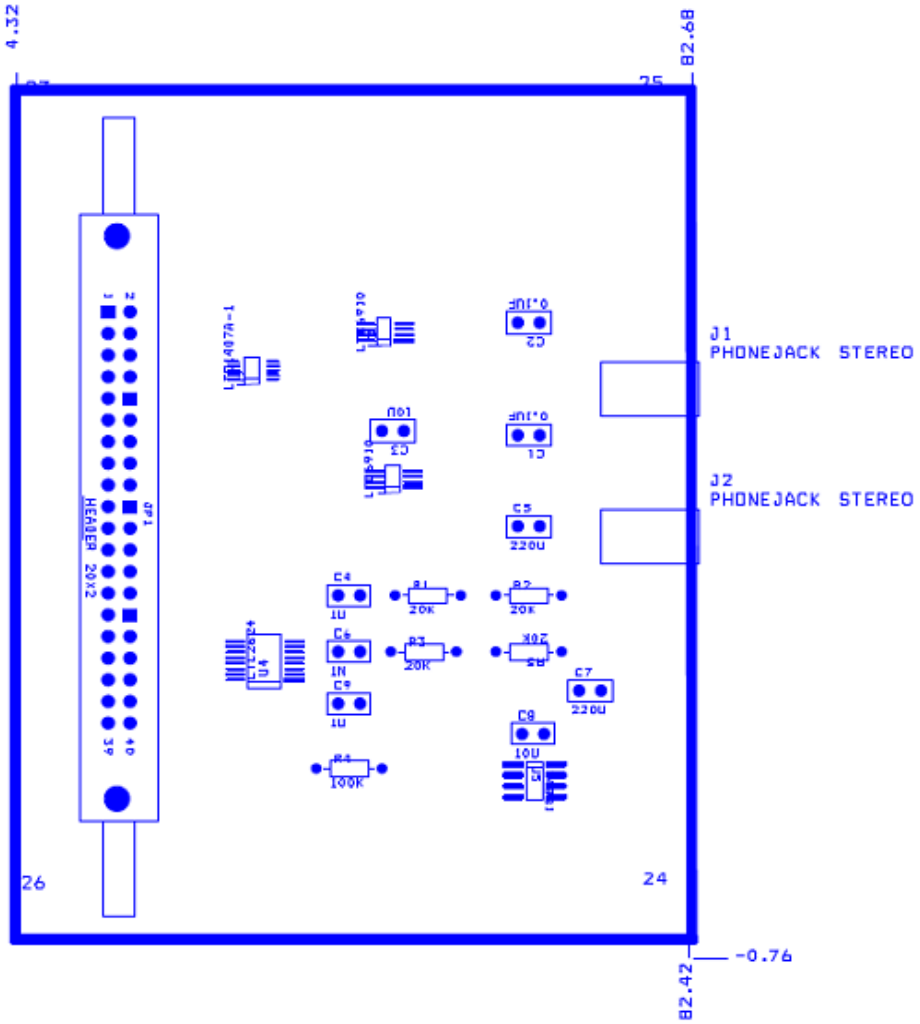
As can be seen the probe pins has been allocated in a pin row , in order to easily connect the instrumentation. The components are mostly SMD. At the left side is the connector to the J20 APF_Dev Board. The INPUT and OUTPUT connections to the external signals are the coaxial connetors at the left side.

The following sections show the photolites.

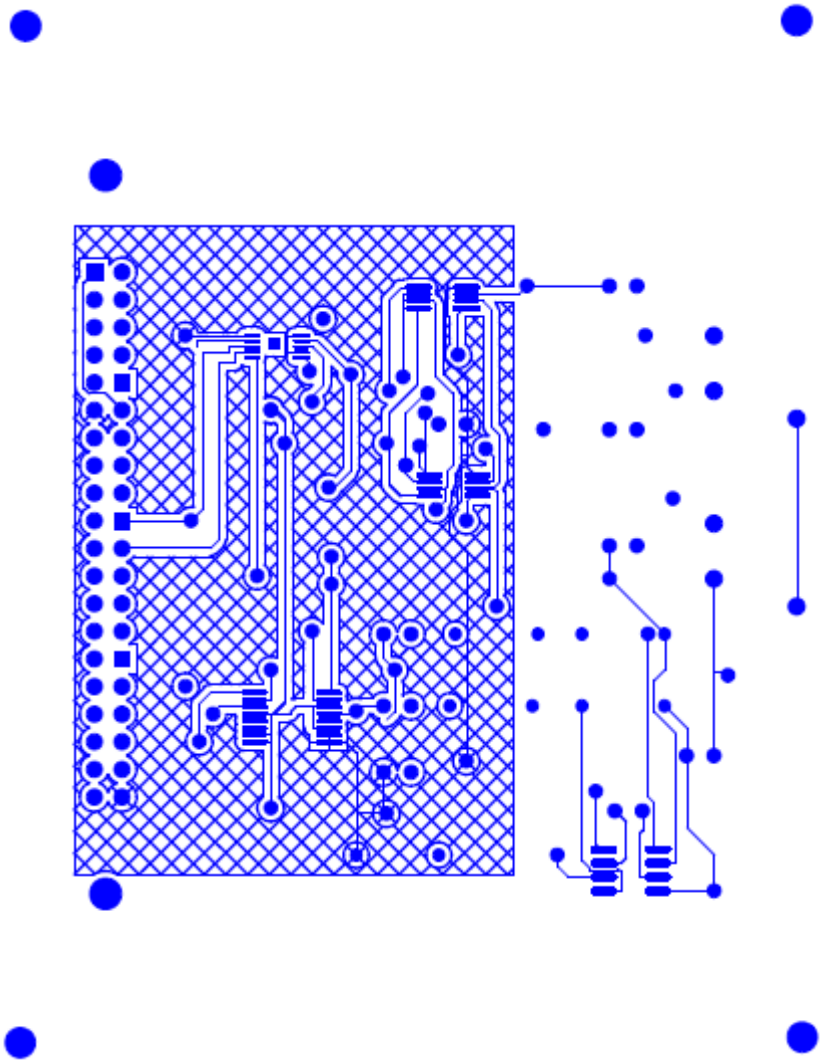


In the next section are shown the intermediate results for the board design process.

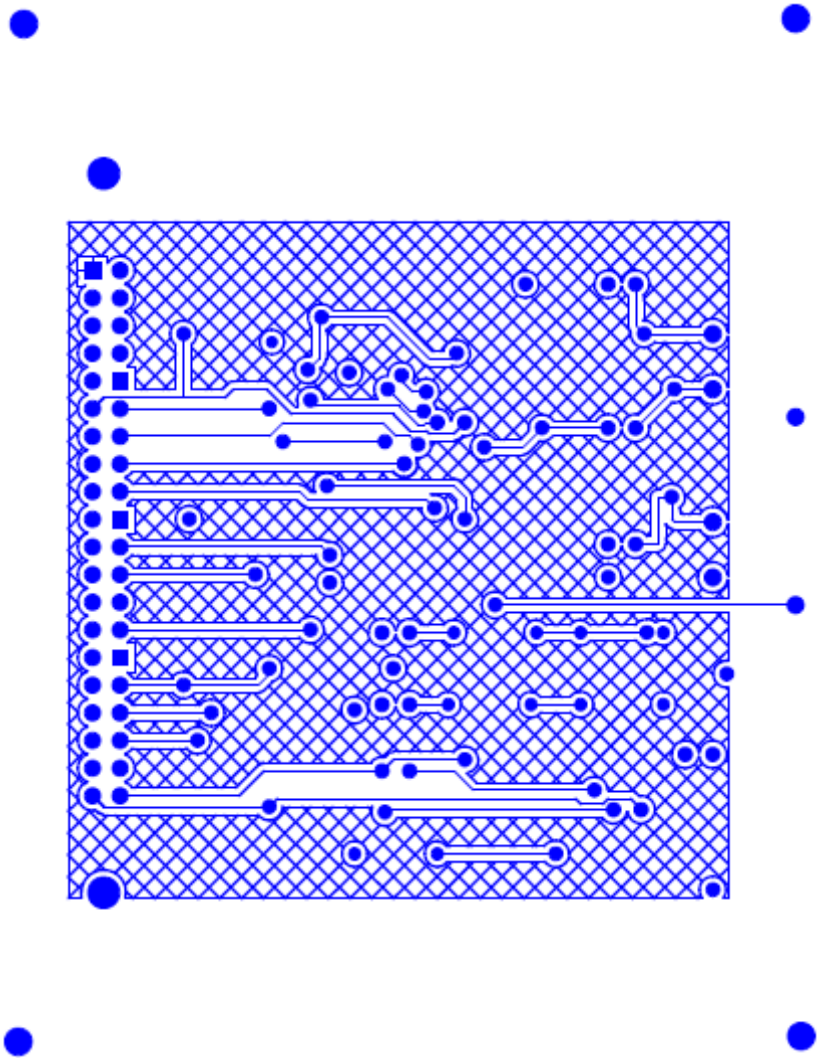
5.1.1 Assembly Layer



5.1.2 Top Layer

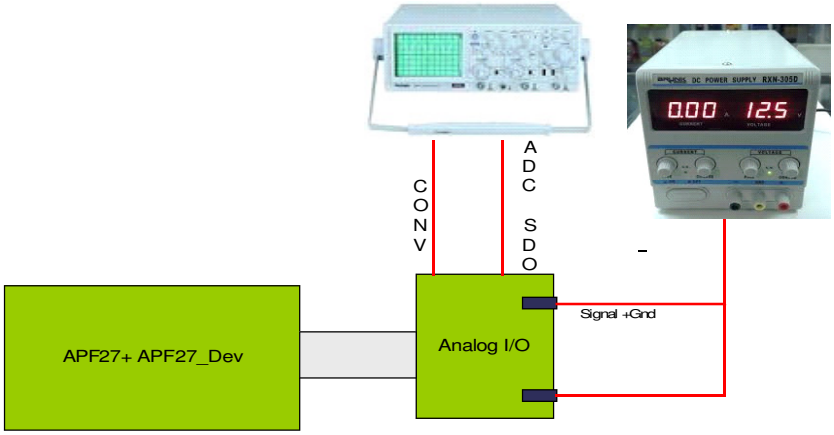


5.1.3 Bottom Layer



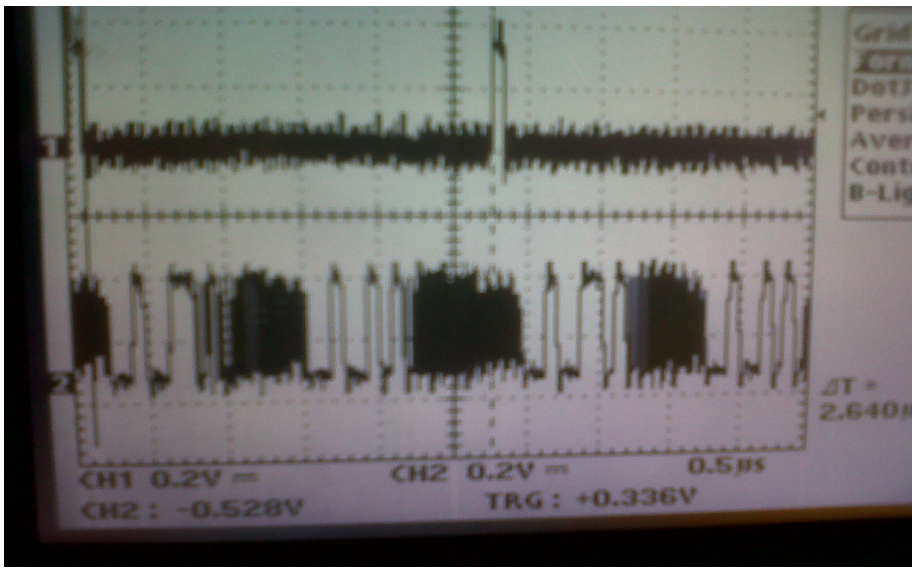
5.2 Hardware and Firmware Functionality Test

A test was designed to verify the hardware and firmware handshake for the adquisition functionality. Basically consist in applying a DC signal to both analogs input , connecting the scope to see the signals betwen ADC and FPGA. The schematic corresponding to the installation is shown bellow



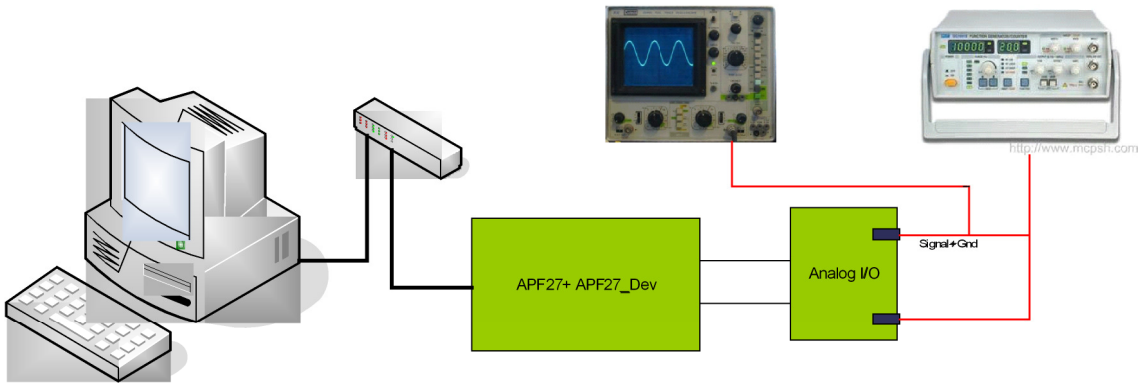
As can be seen in the above picture the scope channel 1 was connected to the CONV signal, . This signal output from FPGA to a ADC input , on low to high transition tells the ADC to initiate a conversion. The scope channel 2 was connected to the ADC SDO signal. This signal goes from ADC to FPGA with the conversions results. The picture below , shows the Scope The Channel 1 at the top and Cahnnel 2 at the bootom, shows that the ADC is sendig two 14 bit words, for the two analog inputs, as per the datasheet information, after the CONV signal.

The data captured for channel 1 was 0x26b0, this is 0010 0110 1011 00 , it can also be appreciated from the image that the nine most significant bits are clear, while the five less significant are blurry, showing a signal small noise pressence .



5.3 Full Functionality Test

A test was designed to verify the full system functionality. Basically consist in applying a signal generator to the channel 1 analog , connecting the scope to see the inyected signal .The APF Armadeus system is connectedthrough a switch to a PC with Windo OS to run the Java Client. The schematic corresponding to the installation is shown bellow



5.3.1 Sinoidal signal Test

As first test a sinusoidal input signal, with F=60 Khz is applied , this signal has a 10 mV amplitude and 10 mV DC component , so the explession (t in secconds)would be

$$x(t)=10+10\cos(2\pi \dot{F} \dot{t})$$

The sample period is 10 us , so the 128 samples would be 1.280 ms , as we can see the 3 complete periods , the signal period is :

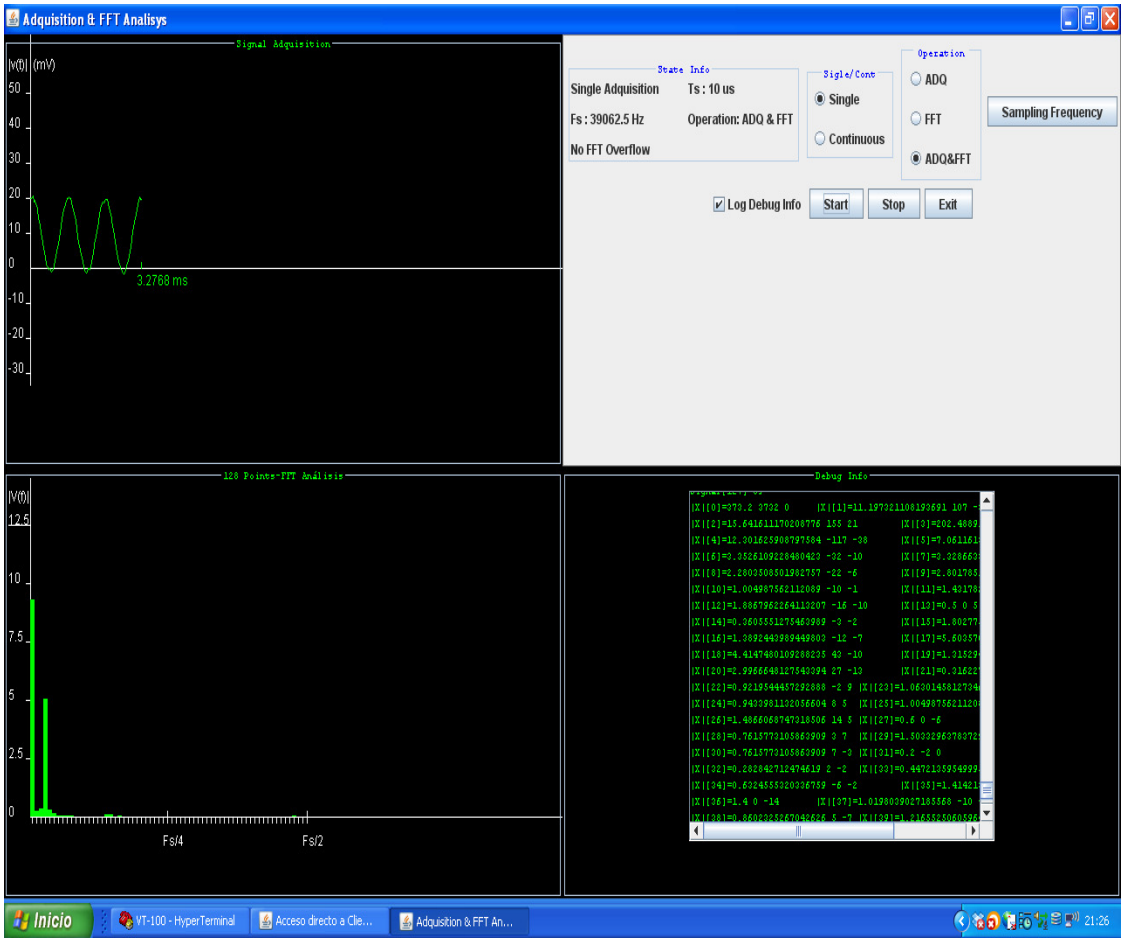
$$T=\frac{1.280}{3}=0.4267\,ms$$

So, the frequency is

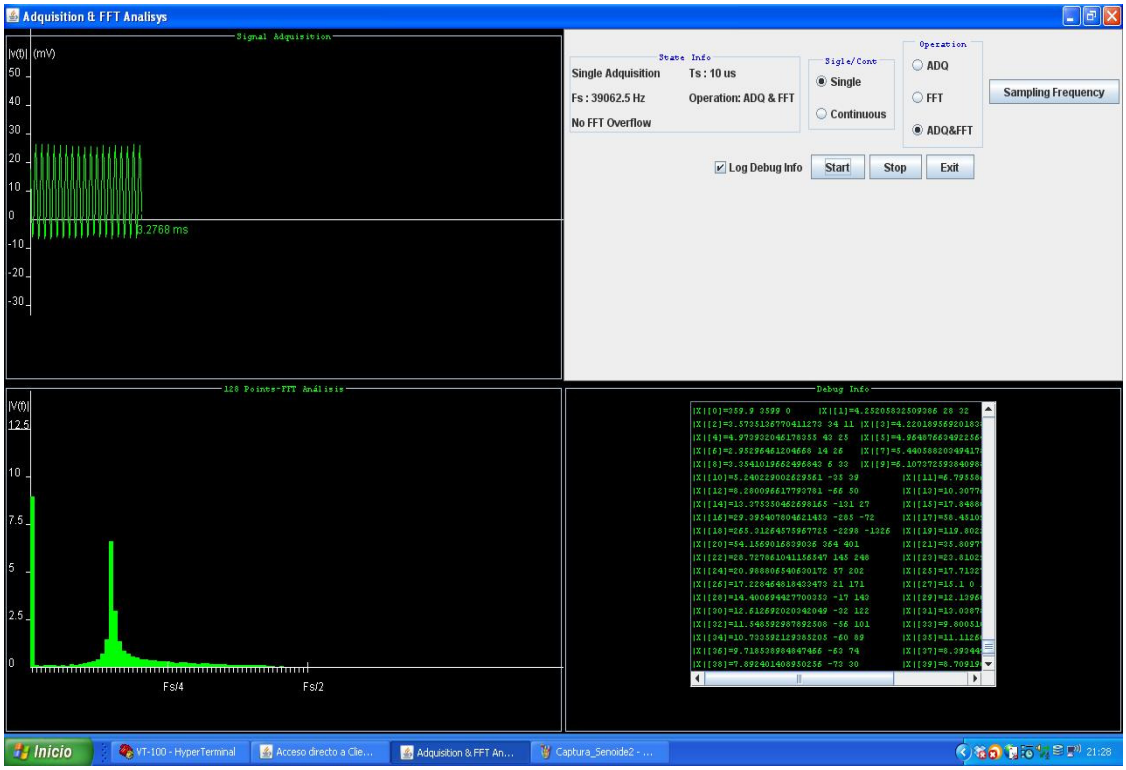
$$F=\frac{1}{0.4267}=2.34\,kHz$$

On the other hand , in the lower panel , is seen the FFT where two delta are clearly distinguished the DC component, and the the frequency term coorespondent to the AC component with half of amplitude , negative frequencies are not ploted as is simmetrical, and only adds phase information.

The delta is the third term X(3). The maximun frequency ($\frac{1}{T_s}=50\,KHz$) would be X(64), that would coorspond to 64 cycles captured seen in scope. The frequency resolution is 0.781 Khz , that is because is the third term.

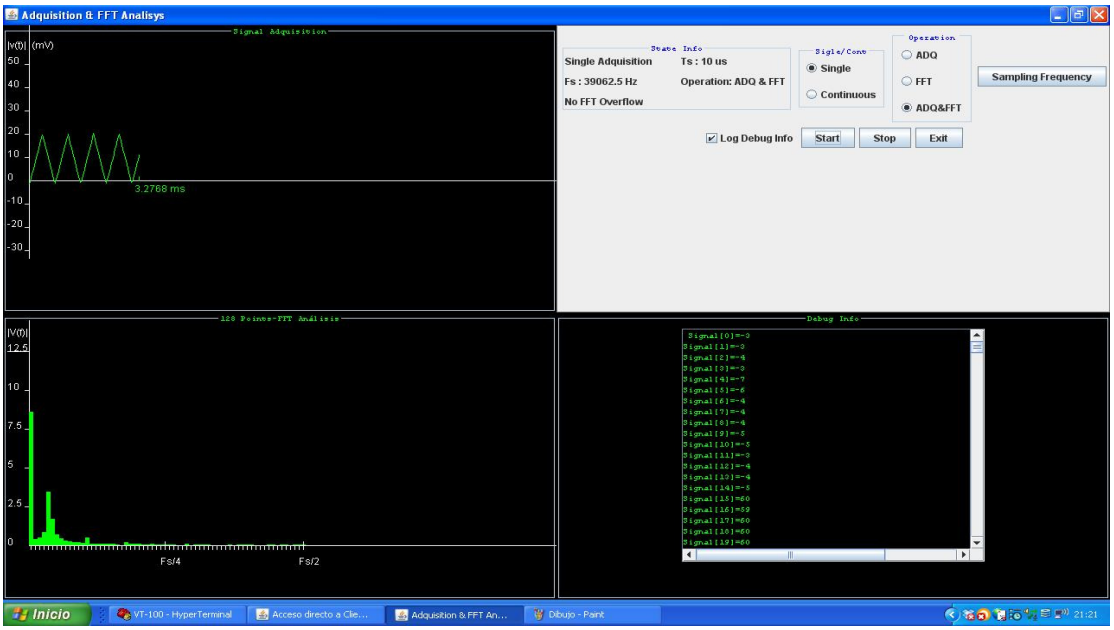


If we rise the frequecy up to 14 Khz we can see the following screenshoot, where the delta is at X(18) position. (14*0,781 = 18)



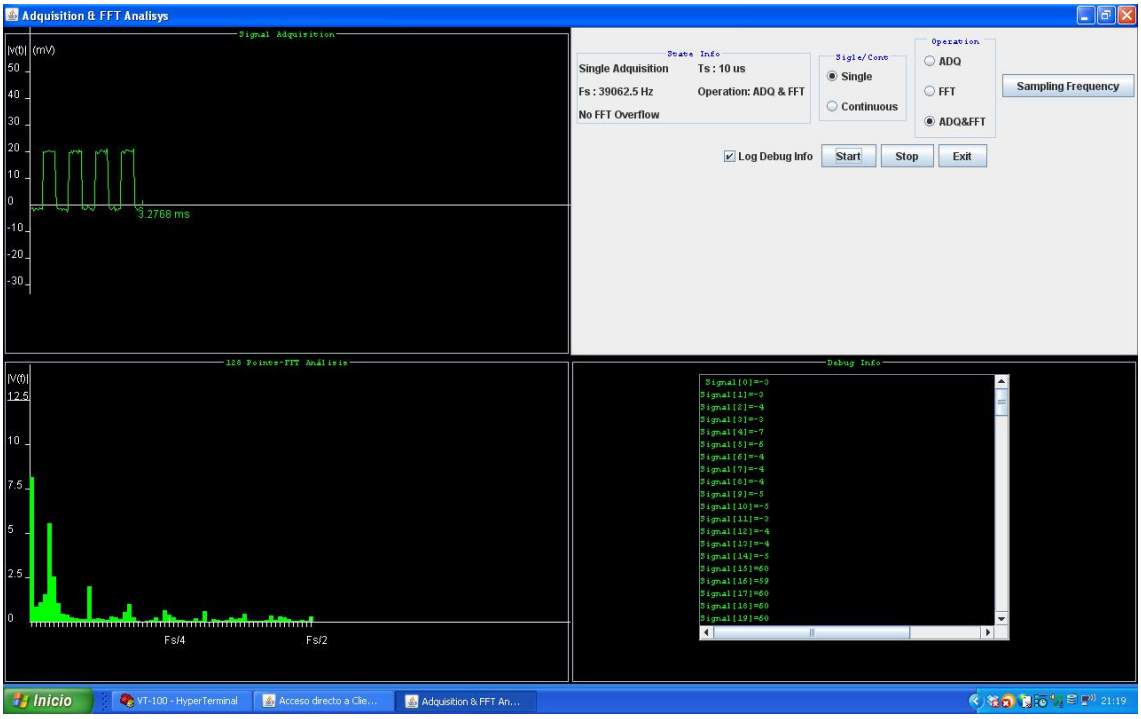
5.3.2 Triangular signal Test

Here we can see a triangular spectrum, where the third and fith armonics can be distingished.



5.3.3 Square signal Test

Finally we can see the square signal spectrum , the screeshoot reflects a FFT sinc like distribution.



5.4 Digital System and Linux Driver Design procedure

Finally, using the developed system as stating point can be stablished a procedure to implement FPGA digital systems for Linux embeded systems as was pointed out at the introduction as one of the projects target.

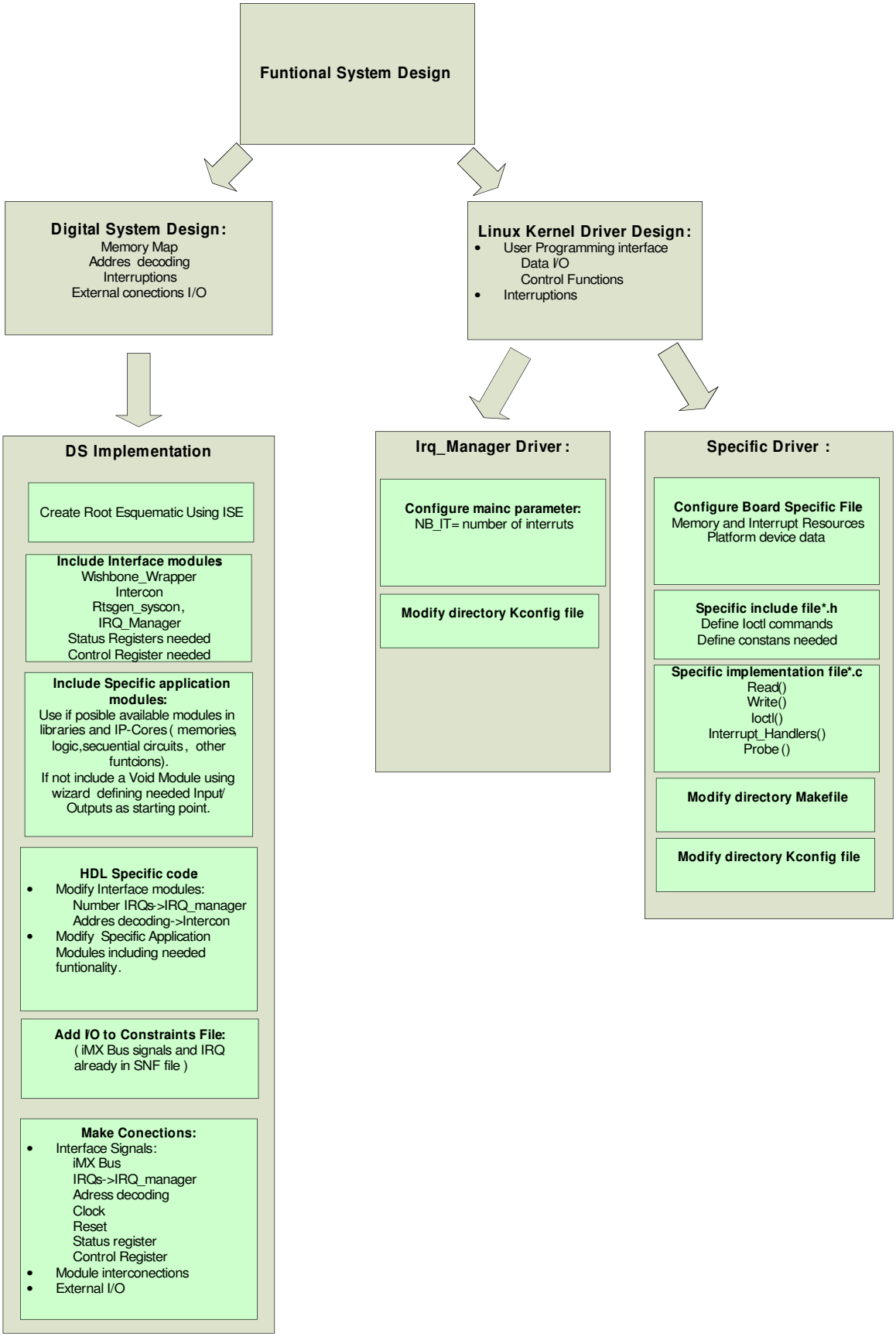
The following picture summarizes the procedure. This procedure should start with a functional análisis to define the high level functionalities. From the user point of view the system wll have a device file interface on the Linux side and external input and outputs on the FPGA external interface.
The next step is is to a separate design for the linux driver and the digital design funcionalities and its interface . This análisis should take into account the best solution , in order to use the right resurces in each side to meet the system requirements.

Once finished the designed phase , follows the implementation phase. This phase is suposed to start from these project files.
The digital system is recomended to wholly be done by using the Xilinx ISE Webpack development Tool, as it has a many features that simplifies implementation tasks.
Follow the implementation steps indicated in the diagram , once finished, the project must be compiled whose processs end up with the FPGA program file *.bit generation.
To download to the APF board and fplash the FPGA see Annexes or the Armadeus Wiki webpage.
The kernel driver part is divided in two that actually are two diferent drivers.
The irq_manager driver is the part engaged of the IRQ_manager circuit, so it receives the interruption when this circuit rises the irq line , executing the interrupt handler that depending on the state of the IRQ_PENDING register asserts the corresponding specific driver interrupt.
The only thing that should be defined in this driver is the number of bits used for interrupts, that is the number of input interrupt lines that the IRQ_Circuit has.

The specific driver, requires to configure the specific device platform board file, to indicate resources , that is , memory and interruts to be used, as well as other data as driver name , identification , etc...
the main task is to do the specific implementation file , in which must be coded the driver functionality. Using this project file as starting point should be modified read,write,ioctl,interrupt handlers and probe funtions.

It should be note that is needed the probe function to register an interrupt hander for each irq_manager driver output interrupt , however this interrupt hander may be the same function as the interrupt number is received , so as let the handler know about it.

A specific include file like fft_fpga.h in this project may be created to include the project constans like memory offsets , parameters, messages numbers,etc.


Funtional System Design
Digital System Design:
Memory Map
Addres decoding
Interruptions
External conections I/O
Linux Kernel Driver Design:
User Programming interface
Data I/O
Control Functions
Interruptions
DS Implementation
Create Root Esquematic Using ISE
Include Interface modules
Wishbone_Wrapper
Intercon
Rtsgen_syscon,
IRQ_Manager
Status Registers needed
Control Register needed
Include Specific application modules:
Use if posible available modules in libraries and IP-Cores (memories, logic,secuential circuits, other funtcions).
If not include a Void Module using wizard defining needed Input/ Outputs as starting point.
HDL Specific code
Modify Interface modules:
Number IRQs->IRQ_manager
Addres decoding->Intercon
Modify Specific Application Modules including needed funtionality.
Add I/O to Constraints File:
(iMX Bus signals and IRQ already in SNF file)
Make Conections:
Interface Signals:
iMX Bus
IRQs->IRQ_manager
Adress decoding
Clock
Reset
Status register
Control Register
Module interconections
External I/O
Irq_Manager Driver:
Configure mainc parameter:
NB_IT= number of interruts
Modify directory Kconfig file
Specific Driver :
Configure Board Specific File
Memory and Interrupt Resources
Platform device data
Specific include file*.h
Define Ioctl commands
Define constans needed
Specific implementation file*.c
Read()
Write()
Ioctl()
Interrupt_Handlers()
Probe ()
Modify directory Makefile
Modify directory Kconfig file

6 CONCLUSIONS

As show results, the project scope has been successfully covered, in particular, the following task

- Analog I/O Board design and implementation , to interface ADC with external analog signal and existing FPGA through SPI in APF27 board. This board serves not only for this project specific application but also for other projects including analog outputs.
- Design and implemetation in HDL languaje the needed firmware for interfacing with the APF27 system bus and make the specific application functionality , in this project case the signal adquisition (throught the ADC), and the 128-FFT signal análisys.
- Interrupt driven Linux device driver dessign and implemetation, including specific functionality. The design procedure has been break down into two parts , Irq_manager for interruption managing and Character device driver for the specific funcionality.
- User space application design and implemenation , in the particular application a tcp server, to send the adquisition samples and 128-point FFT to a remote system.
- Demostration Program , for this project a JAVA Client application, that connects with the tcp server and make a graphic plot representing the 128 signal adquisition sample points and the 128-FFT ponts , to demostrate the final system performance , displaying results. It also have others functionality for debuging and configuration.

Finally , the project stablishes a platform that serves as start point for a design procedure to properly interface a specific digital system with a linux embeded system, and to develop the kernel driver.

This project resuts can be used for an undergraduate optative subject oriented to digital systems design for embeded systems, to fill the gap between both Digital Systems and Embeded Linux programming subjects.

Specifiacally the academic value that can be taken from this project is

for the digital system design for FPGA:

- the microprocessor interface (wraper, intercon), and memory map
- wishbone bus
- interrupts handling (irq_manager)
- internal memory implementation
- registers (status , control)
- FMS for application control (Ctl_Adq)

For Linux software developing

- Kernel Drivers developping
- User space using device drivers and TCP/IP communications

For signals analisys.

- Signal adquisition
- FFT analisys

For JAVA applications developing

- TCP/IP communications
- HMI programming.

This project results can be applied to make applications for a wide range of fields, like electromedical, mechanical, power electronics , automation etc...

The specific application for the project has been sucessfully implemented and tested and a general procedure based on project results has been designed. So as final conclusion can be said that the project targets marked at the introduction has been sucessfully accomplished.