

TRABAJO ESPECIAL DE GRADO

DESARROLLO DE UN SISTEMA DE PROCESAMIENTO DE IMÁGENES PARA MICROSATÉLITES

Profesor Guía: Tremante Panayotis
Tutor Industrial: Claudio Passerone

Presentado ante la Ilustre
Universidad Central de Venezuela
por el Br. Alejandro G. González E.
para optar al Título de
Ingeniero Electricista

Caracas, 2009

CONSTANCIA DE APROBACIÓN

Caracas, 2009

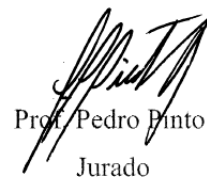
Los abajo firmantes, miembros del Jurado designado por el Consejo de Escuela de Ingeniería Eléctrica, para evaluar el Trabajo Especial de Grado presentado por el Bachiller Alejandro Gabriel González Esculpi, titulado:

“DISEÑO DE UN SISTEMA DE PROCESAMIENTO DE IMÁGENES PARA MICROSATÉLITES”

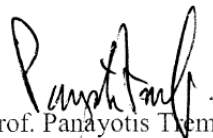
Consideran que el mismo cumple con los requisitos exigidos por el plan de estudios conducente al Título de Ingeniero Electricista en la mención de Electrónica, y sin que ello signifique que se hacen solidarios con las ideas expuestas por el autor, lo declaran APROBADO.



Jurado



Jurado



Prof. Panayotis Tremante
Profesor Guía

González E., Alejandro G.

**DESARROLLO DE UN SISTEMA DE PROCESAMIENTO DE
IMÁGENES PARA MICROSATELITES**

Prof. Guía: Tremante Panayotis. Tutor Industrial: Claudio Passerone. Tesis. Caracas. U.C.V. Facultad de Ingeniería. Escuela de Ingeniería Eléctrica. Ingeniero Electricista. Opción: Electrónica. Institución: Politécnico de Turín. Turín, Italia. 2009. 95 h. + anexos.

Palabras Claves: Satélite; Procesamiento de imágenes; Procesador; VHDL; FPGA; Sistema Operativo; Programas.

Resumen. Se plantea el estudio y desarrollo de una propuesta para un sistema de procesamiento de imágenes como eventual Payload de un satélite universitario basado en el estándar CUBESAT. Se toma como punto de partida el modelo VHDL sintetizable de un procesador el cual puede servir de base a un sistema operativo compilable, capaz de sostener programas para el procesamiento de imágenes. Para el desarrollo del proyecto fue provista una tarjeta de bajo costo basada en un chip FPGA capaz de sostener el procesador mencionado así como varios periféricos útiles para la realización de diversas pruebas sobre el mismo.

Sumario

La simplicidad del estándar CubeSat para el diseño de picosatelites, definido en el año 2001 por el Profesor Robert Twiggs de la Universidad de Stanford en EE.UU., ha motivado varias universidades a nivel mundial a desarrollar sistemas de esta categoría como actividad didáctica, capturando además el interés de diversos sectores a invertir en el desarrollo de investigaciones en el campo de las telecomunicaciones.

El satélite PICPOT fue desarrollado en el Politécnico di Torino desde el 2003, sirviendo como primera experiencia para la institución y motivando futuros proyectos en el mismo campo. Así en el año 2006 nace el proyecto ARAMIS, segundo satélite del Politécnico di Torino, actualmente en pleno desarrollo.

La peculiaridad del proyecto ARAMIS es la “modularidad”, es decir, el desarrollo independiente y simultaneo de diversos módulos estándar compatibles entre sí, en lugar de ser orientado al cumplimiento de una misión particular. Los principales módulos o etapas que constituyen el proyecto ARAMIS son: Etapa de Potencia, Etapa de transmisión y recepción, Computador de Bordo, Estación de Tierra y Payload.

Este trabajo está orientado al estudio y desarrollo de una propuesta para un eventual Payload dedicado a la adquisición de imágenes. Se toma como punto de partida el modelo VHDL sintetizable de un procesador el cual puede servir de base a un sistema operativo compilable, capaz de sostener programas para el procesamiento de imágenes. Para el desarrollo del proyecto fue provista una tarjeta de bajo costo basada en un chip FPGA capaz de sostener el procesador mencionado así como varios periféricos útiles para la realización de diversas pruebas sobre el mismo.

La tarjeta utilizada es la GR-XC3S-1500 construida por Pender en colaboración con Gaisler Reseach, basada en una FPGA Spartan 3, conteniendo como principales periféricos memorias PROM y puertos: seriales, Ethernet, USB, JTAG, y VGA.

El modelo VHDL utilizado corresponde al procesador LEON3, desarrollado por Gaisler Research basado en la arquitectura SPARC V8, especialmente diseñado para aplicaciones satelitales. La librería GRLIB IP esta también incluida en el modelo, conteniendo dispositivos útiles y en algunos casos indispensables para el procesador. También son provistas por la Gaisler varias herramientas que simplifican la configuración del modelo VHDL antes de la síntesis y para la compilación de aplicaciones a ser ejecutadas en el procesador.

Para la síntesis del modelo VHDL y su instalación en la FPGA (utilizando un cable JTAG) son utilizadas aplicaciones contenidas en el paquete ISE Foundation de Xilinx. Para probar el procesador instalado se utilizó una versión de prueba de GRMON, un programa que sirve de interfaz para procesadores LEON a través de los puertos JTAG, serial o USB; permitiendo controlar los dispositivos instalados junto al procesador, la instalación y ejecución de aplicaciones, así como otros aspectos importantes del sistema.

Varios fallos fueron observados en el procesador obtenido del modelo VHDL sintetizado con las herramientas disponibles (inclusive utilizando la configuración default para el modelo) el cual hizo imposible su utilización. Por ende se decidió utilizar una versión ya sintetizada del procesador también provisto por la Gaisler, la cual funcionó en la tarjeta de manera óptima, permitiendo descartar posibles daños físicos como causa de los fallos previamente obtenidos. Se decidió continuar el proyecto utilizando la versión operativa del procesador en lugar de intentar corregir la versión configurable, permitiendo probar la capacidad del procesador más allá de no poder demostrar la flexibilidad del modelo VHDL.

Se utilizó SnapGear Linux como sistema operativo a ser instalado en el procesador, el paquete escogido incluye el kernel correspondiente, herramientas para simplificar su configuración y diversos templates entre los cuales uno corresponde al procesador LEON3 en la tarjeta GR-XC3S-1500, las modificaciones realizadas al kernel fueron limitadas a la asignación de la dirección IP y la instalación de programas a ser ejecutados de forma automática en la tarjeta.

Para probar la adquisición y envío de datos se utilizó el puerto Ethernet, aprovechando el protocolo TCP/IP y desarrollando programas basados en el modelo Cliente-Servidor para el intercambio de datos entre la tarjeta y bloques externos del sistema emulados por diversos procesos en un servidor externo. Para probar la capacidad del procesador para el procesamiento de imágenes se utilizó la librería JPEG elaborada por IJG (Independent JPEG Group), la cual incluye archivos en lenguaje C en formato open-source, open-license, conteniendo códigos fuente de programas que hacen la compresión basada en el algoritmo correspondiente. Se decidió trabajar con archivos en formato PPM para ser comprimidos y obtener los archivos correspondientes en formato JPEG.

La última parte del proyecto consistió en el desarrollo de programas que incluyen en el modelo Cliente-Servidor, los tipos de archivo utilizados y las funciones para la compresión. Los programas desarrollados fueron los siguientes:

- sendPPM: envía un archivo en formato PPM del servidor a la tarjeta.
- INGRB: corre en la tarjeta de forma constante, esperando por un archivo PPM a ser comprimido, ejecuta el programa para la compresión al recibirlo, envía el archivo JPEG obtenido y espera por el siguiente archivo PPM.
- recJPG: corre en el servidor de manera constante, recibe el archivo JPEG enviado desde la tarjeta y lo almacena en memoria.

La compresión es realizada por el programa cjpeg3, el cual es el mismo cjpeg provisto por la librería de IJG, con ligeras modificaciones en su código fuente. Los resultados obtenidos fueron satisfactorios, se utilizaron varios archivos PPM y fueron medidos los tiempos obtenidos para el envío, la compresión y la recepción de archivos correspondientes para cada caso.

El trabajo presentado se organiza de la manera siguiente:

- Capitulo 1: introducción sobre los satélites universitarios, descripción de aspectos del proyecto ARAMIS.
- Capitulo 2: describe el desarrollo del hardware (posicionamiento del procesador sobre la tarjeta).
- Capitulo 3: describe el desarrollo del sistema operativo (compilación del kernel y cambios realizados)
- Capitulo 4: describe el desarrollo de programas para el envío y recepción de datos, y el procesamiento de imágenes.
- Capitulo 5: conclusiones.

Contents

Constancia de aprobación	iii
Resumen	iv
Sumario (en castellano)	v
Contents	ix
List of Figures and Tables	xi
1. Introduction	1
1.1. The PICPOT project	2
1.2. The ARAMIS project	4
2. Hardware Implementation	6
2.1. The LEON3 Processor Core	6
2.1.1. LEON3 Architecture and Features	7
2.1.2. The GRLIB IP Library	9
2.1.3. LEON3 Applications development tools	10
2.2. The GR-XC3S-1500 Development Board	11
2.2.1. Board technical description	13
2.2.2. Board parts to be used	16
2.3. LEON3 Design Configuration	20
2.4. Procedure to place the processor on the FPGA	22
2.5. The GRMON LEON Monitor	24
2.6. Tests and Results	25
3. Firmware Implementation	29
3.1. The SnapGear Linux	29
3.2. SnapGear Linux Compiling	30
3.2.1. Installing the Toolchain	31
3.2.2. Installing the SnapGear Distribution	32
3.2.3. Configuring Linux	32
3.2.4. Building SnapGear	37
3.3. Modifications over the SnapGear Linux Kernel	37

3.3.1. Board IP address	37
3.3.2. Adding programs to be executed on the board	38
3.3.3. Automatic Program Startup	39
3.4. SnapGear Linux Installing Procedure	40
3.5. Tests and Results	42
4. Software Implementation	48
4.1. Used File Formats	48
4.1.1. PPM file format	49
4.1.2. JPEG file format	51
4.2. An Overview of the JPEG Compression Algorithm	53
4.3. The IJG JPEG Library	57
4.4. Communication Channel Development	58
4.4.1. The Client-Servr model	58
4.4.2. Socket Types	58
4.4.3. Socket implementation on the Internet domain	59
4.5. Developed Program Description	67
4.5.1. To run outside the Board	67
4.5.2. To run Inside the Board	69
4.6. Developed Programs Usage	77
4.7. Tests and Results	79
5. Conclusions	81
Bibliography	84
Appendixes	86
A.Developed Programs Source Codes	86

List of Figures and Tables

Figure 1.1. The PICPOT Satellite	3
Figure 2.1. LEON3 processor core block diagram	7
Figure 2.2. The GR-XC3S-1500 Development Board	11
Figure 2.3. GR-XC3S-1500 development board block diagram	13
Figure 2.4. Board Memory Interface	16
Figure 2.5. Configuration of Oscillators	18
Figure 2.6. LEON3 GUI configuration tool	20
Figure 2.7. Processor configuration window	21
Figure 2.8. Peripherals configuration window	21
Figure 2.9. Built LEON3 processor on GRMON monitor	25
Figure 2.10. Failures on the built processor	26
Figure 2.11. Precompiled LEON3 processor on GRMON monitor	27
Figure 3.1. SnapGear main configuration GUI	32
Figure 3.2. Template Configuration selection	33
Figure 3.3. Vendor/Product section	34
Figure 3.4. kernel/library/defaults selection menu	34
Figure 3.5. Linux 2.6.x kernel GUI configuration utilit	35
Figure 3.6. Board IP address setting	38
Figure 3.7. Example of the inittab file	39
Figure 3.8. Flash erasing and SnapGear image download on GRMON	43
Figure 3.9. Programs and files added to the root file system	44
Figure 3.10. Board IP address verification from a remote host	44
Figure 3.11. Board IP address verification on SnapGear shell	45
Table 4.1. PPM formats comparison	50
Table 4.2. JPEG markers	52
Figure 4.1 Server side socket building and usage	60
Figure 4.2. Client side socket building and usage	64

Figure 4.3. sendPPM program flow chart	72
Figure 4.4. recJPG program flow chart	73
Figure 4.5. INGRB program flow chart	74
Figure 4.6. receiveP6 function flow chart	75
Figure 4.7. sendJPG function flow chart	76
Figure 4.8. Programs test environment	79
Table 4.3. Programs performance	80

Chapter 1

Introduction

Issued on the year 2001, the CUBESAT (cube satellite) [1] standard has inspired many universities worldwide to invest several research efforts into the development of aerospace applications based on that model. The standard has been developed under the guidance of Professor Robert Twiggs (from Stanford University, USA) in association with the Space Systems Development Laboratory (SSDL) from Stanford University and California Polytechnic State University.

The CUBESAT standard is basically defined by the following characteristics:

- Volume: 10x10x10cm
- Weight: < 1kg

Compared to traditional multi-million-dollar satellite missions, CubeSat projects have the potential to educate the participants and implement successful and useful missions in science and industry at much lower costs.

Inspired on the CUBESAT model, Politecnico di Torino has recently developed the PICPOT satellite [2], and now the ARAMIS project is under development.

1.1 The PICPOT Project

PICPOT [2] was the first satellite developed at Politecnico di Torino. Its development began on 2003 and its launch failed on 2006[1]. The project was based on the following requirements:

- Cube shape, 13x13x13cm
- Weight: < 5kg
- Medium Power < 1.5 W
- Minimum lifetime: 90 days
- COTS components usage in space
- LEO Orbit (altitude between 600 and 800km)
- Compatibility with the POD launcher

The functions of the satellite regarded the following parameters:

- Temperature and luminescence measures acquisition
- Photographies acquisition
- Data transfer to the ground station

A PICPOT picture is presented in figure 1.1

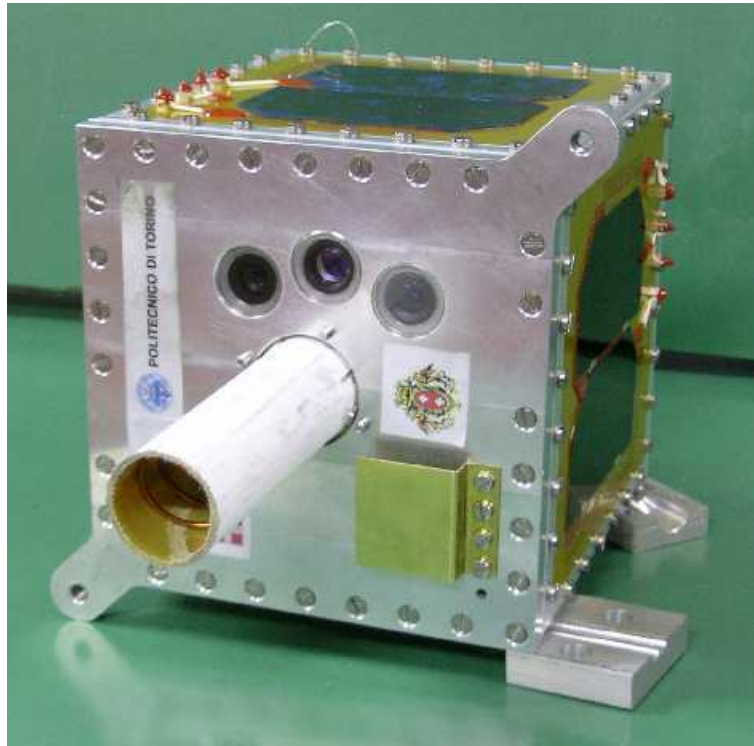


Figure 1.1. The PICPOT satellite

Five of the six satellite faces were used to place the solar panels. On the remaining face two antennas (437MHz and 2.4GHz), 3 cameras, 2 kill switches and a test connector to verify the on-board electronics correct functioning were placed. Internal power supply was provided by 6 rechargeable batteries placed between the solar cells and the electronic boards. Three processors were placed on the satellite:

- ProcA: on-board management, associated to the 437MHz communication channel, 11MHz clock frequency and power supply of 3.3V.
- ProcB: on-board management associated to the 2.4GHz communication channel, 4MHz clock frequency and power supply of 3.3V
- Payload: for the image acquisition from the three photocameras, send to ground station by ProcA and ProcB.

ProcA and ProcB were operationally independent. The communication with the ground station was designed to be performed through two channels, both of them in amateur band with APRS protocol.

1.2 The ARAMIS Project

The ARAMIS project has assumed the PICPOT project evolution. The ARAMIS target is to define a low cost standard modular architecture for small size satellites. The applied “modular architecture” [3] method aims to develop some standard modules connectable between them and with specific modules built according to the mission, rather than develop the entire system based on the mission. The standard modules to be developed are listed as follows:

- Power Supply (Power Management tile)
- Tx-Rx (Telecommunication tile)

The modules which complete the satellite system and are developed according to the required mission are:

- Ground Station
- On Board Computer
- Payload

This procedure allows also fix the number of standard modules according of each kind to the satellite mission. The main feature of the modular architecture model is the re-usability of the designs involved in the standard modules; this factor is translated as saving of a high amount of resources and time in future projects.

The main target of the presented project in this thesis is to develop a proposal for an eventual Payload processor oriented to image acquisition. The starting point is a processor defined by a provided VHDL model able to support an operating system,

which should be able to support applications that manage the activities related to procedures in image processing. It is provided a development board specially designed for the mentioned processor support.

Chapter 2

Hardware Implementation

This chapter explains the procedure to compile a processor open-source VHDL model and place it over a based-on FPGA development board. After the compiling is completed and the processor is placed on the board, are discussed the results of some tests performed over it to prove its operation, the flexibility of the used model and the reliability of the development board itself. The built processor is also compared with a precompiled version provided by the manufacturer.

2.1 The LEON3 Processor Core

The LEON3 [4,6] processor core is a synthesizable VHDL model of a 32-bit processor with the SPARC V8 architecture. The core is highly configurable and suitable for system-on-a-chip (SOC) designs. The configurability allows designers to optimize the processor for performance, power consumption, I/O throughput, silicon area and cost. The core is interfaced using AMBA-2.0 AHB bus, and supports the IP plug and play method provided in the Gaisler Research IP Library (GRLIB). The Processor can be efficiently on both FPGA and ASIC technologies and uses standard synchronous RAM cells for both caches and register file. To promote the SPARC architecture and simplify early evaluation and prototyping, the processor and associated IP library is provided in full source code under open-source license. The LEON3 processor core block diagram is shown in figure 2.1.

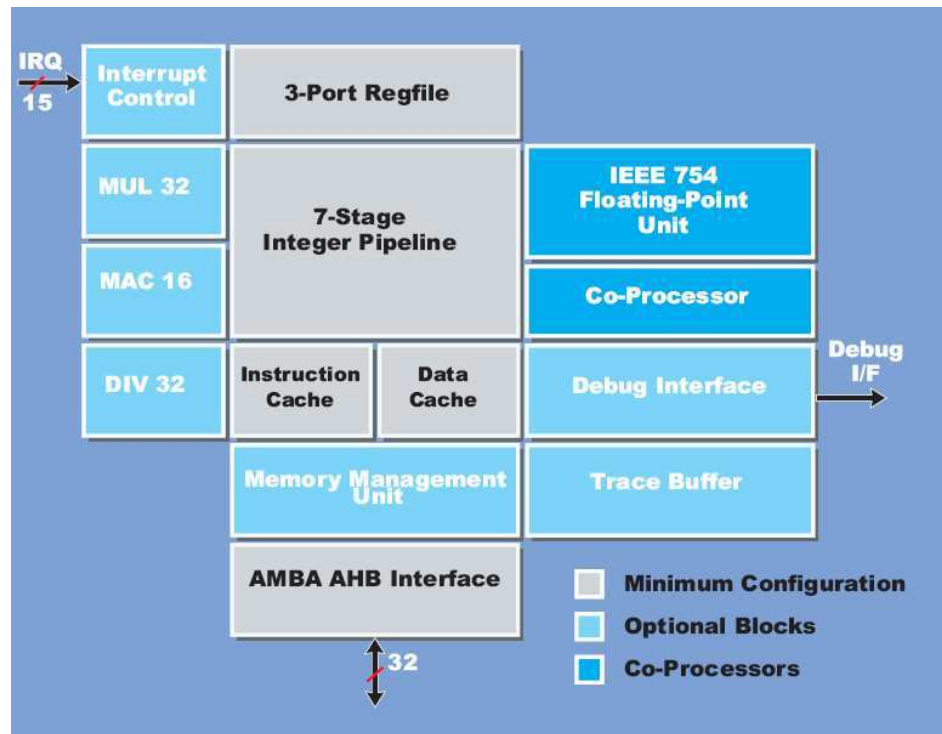


Figure 2.1. LEON3 processor core block diagram

The LEON3 core is also available in a fault-tolerant version (LEON3FT) immune to single event upsets (SEUs), for space and other high-rel applications. The fault tolerant version is not provided under open-source license, but it supports most of the functionality in the standard LEON3 processor.

2.1.1 LEON3 Architecture and features

LEON3 [4,6] is implemented using an advanced 7-stage pipeline with separate instruction and data cache buses (Harvard architecture). The processor supports the full SPARC V8 instruction set, including the MUC, MAC and DIV instructions. An optional IEEE-754 floating-point unit (FPU) provides support for both single- and double-precision floating point operations. The cache system supports multi-set caches with up to 4 sets per cache, 256 kbyte per set and a choice of LRU, LRR or random replacement policy.

LEON3 can be utilized in synchronous multiprocessor configurations (SMP), and provides hardware support for cache coherency, processor enumeration and SMP interrupt steering. A unique debug interface allows non-intrusive hardware debugging of both single- and multi-processor systems, and provides access to all on-chip registers and memory. Trace buffers for both instructions and AMBA bus traffic are also available.

The basic processor core (pipeline, cache controllers and AHB interface) consumes around 20,000 gates and can be implemented on both ASIC and FPGA technologies. On a typical 0.13 μm standard-cell technology, over 400 MHz can be reached.

The LEON3 processor features are resumed in the following list:

- SPARC V8 integer unit with 7-stage pipeline
- Hardware multiply, divide and MAC units
- Interface to the Meiko FPU and custom co-processors
- Interface to high performance IEEE-754 FPU
- Separate instruction and data cache
- Set-associative caches: 1 – 4 sets, 1 – 256 kbytes/set. Random, LRR or LRU replacement
- Data cache snooping
- On-chip 0-waitstate scratch pad data RAM
- SPARC V8 Reference Memory management unit (MMU)
- Power-down mode
- Advanced on-chip debug support unit and instruction trace buffer
- AMBA-2.0 AHB and APB on-chip buses

2.1.2 The GRLIB IP Library

To achieve optimum performance and minimum cost for a SOC design, it is important to reuse existing IP cores and be able to configure these cores for the specific application. The GRLIB IP Library [6] provides a standardized and vendor-independent infrastructure to deliver reusable IP cores.

Integrating third party IP cores from different suppliers can require significant adaptation and harmonization of both functional and logistical interfaces. The GRLIB IP library enhances the development of SOC devices by providing reusable IP cores with common functional and logistical interfaces.

The library is designed to be easy portable to different CAD tools and target technologies. It does not depend on any vendor specific interface or technology which needs to be licensed or procured. The library is designed to allow contributions or extensions from other parties. The GRLIB is designed to be “bus-centric”, i.e., under the assumption that most of the IP cores will be connected through an on-chip bus (such as AMBA-2.0, AHB/APB).

The GRLIB library contains the following IP cores:

- AHB arbiter/multiplexer with plug&play support
- AHB/APB bridge
- 8/16/32-bits PROM and SRAM controller
- 32-bits PC133 SDRAM controller
- UART, timer unit, interrupt controller and GPIO port
- AHB trace buffer
- 32-bit Initiator/Target PCI interface (FIFO/DMA)
- PCI trace buffer
- 10/100 Mbit Ethernet MAC

- Fully pipelined single- and double- precision IEEE-754 FPU
- Technology-independent memory and pad wrappers

2.1.3 LEON3 Applications development tools

Many open source software tools are available for LEON3 applications development [3,6]. A package based on a GNU cross-compilation system is provided by Gaisler Research, including the following tools:

- GNU C/C++ compiler
- Linker, assembler, archiver etc.
- Standalone C-library
- RTEMS real-time kernel with network support
- Boot-prom utility (mkprom)
- Remote debugger monitor for gdb
- GNU debugger with Tk front-end
- DDD graphical user interface for gdb

2.2 The GR-XC3S-1500 Development Board

The GR-XC3S board [5] is a compact, low-cost development board which has been developed by Pender Electronic Design in cooperation with Gaisler Research to enable the evaluation of the LEON2 and LEON3/GRLIB processor systems. The board incorporates a 1.5 million gate XC3S1500 FPGA device from the Xilinx Spartan3 family, which is supported by the free Xilinx web-pack synthesis and place and route tools.

On-board Flash memory and SDRAM are provided together with Ethernet, JTAG, Serial, Video, USB and PS2 interfaces for off-board communication. The incorporation of the onboard volatile and non-volatile memory, together with the communication interfaces makes the board ideal for fast prototyping, evaluation and development of software for Leon microprocessor applications. An actual picture of the board is provided in figure 2.2.

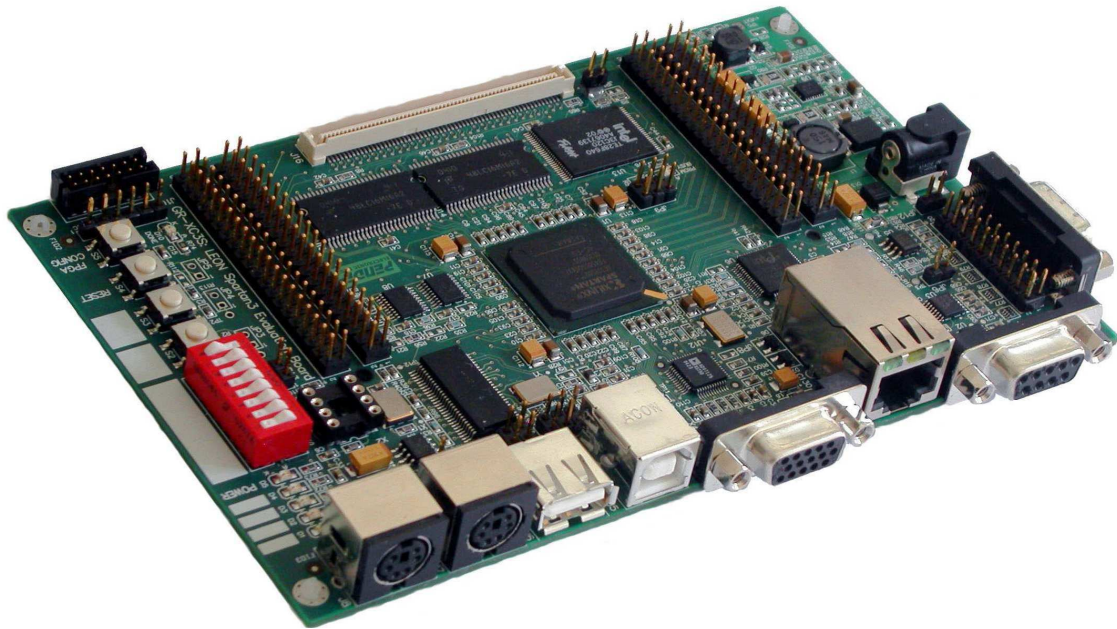


Figure 2.2. The GR-XC3S-1500 Development Board

Expansion to user's peripherals and circuits can be implemented using the expansion connectors, either to implement a user defined mezzanine board, or via ribbon cable connections. A specific connector is provided to allow connection to the standard memory bus signals.

Although targeted for the development of small Leon based systems and computer peripherals, this board can easily be used as a general purpose FPGA development environment for any Xilinx Spartan-3 design.

The board technical description and selection of parts to be used are described as follows.

2.2.1 Board technical description

As previously said, the GR-XC3S-1500 development board incorporates a large capacity Xilinx Spartan-3 FPGA, with on-board memory and interfaces. The board provides a platform which enables the implementation of complex FPGA designs, specially to LEON processors based systems. The development board block diagram is presented in figure 2.3.

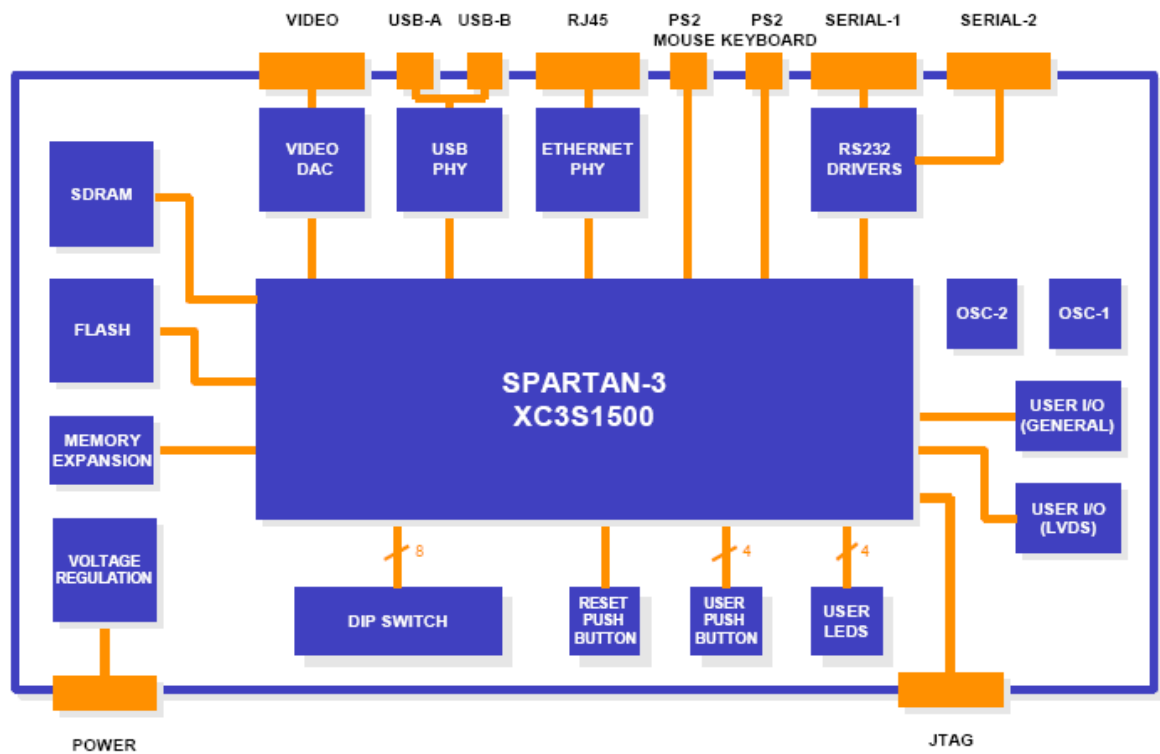


Figure 2.3. GR-XC3S-1500 development board block diagram

The features of the GR-XC3S-1500 Development Board are as follows [ref2]:

- Compact Eurocard (100x160mm) size stand-alone operation with +5V power input
- Xilinx XC3S1500-4FG456 FGPA
 - 1.5million gate Xilinx Spartan 3 device in 456 BGA package

- 1 x 4Mbit (XCF04S) and 1 x 1Mbit (XCF01S) platform Flash Proms for non-volatile storage of FPGA configuration

- On-Board Power Regulators
Texas Instruments TPS75003 Triple-Supply Power Management IC providing
 - +3.3V i/o voltage
 - +2.5V auxiliary voltage
 - +1.2V core voltage

- On-Board Memory
 - PROM 64 Mbit (8 Mbyte) FLASH (organised x8 or x16 bit)
 - SDRAM 512 Mbit (64 Mbyte) PC-133 SDRAM on board (32 bits wide interface)
 - SRAM memory can be added via Mezzanine board, using the memory expansion connector.

- On-Board Oscillators
 - Main oscillator 50MHz
 - User fitted oscillator (DIL-8 socketed)
 - Ethernet oscillator 25MHz

- Interfaces
 - Serial interfaces: Texas Instruments SN75C3232, 3-V To 5.5-V Multichannel RS-232 Compatible Line Driver/Receivers providing high speed serial interfaces (1 Mbaud RS232) with two standard SUB-D9 female connector interfaces. Can easily be configured to support LEON serial DSU for processor debug and program download
 - Ethernet PHY: Intel LXT971A 3.3V Dual-Speed Fast Ethernet PHY Transceiver device providing 10/100Mbit/s Ethernet interface, with RJ45 10/100Mbit Ethernet connector

- Video DAC: Analog Devices ADV7125-50 Triple 8 bit High-Speed Video DAC device, providing 50MHz, 24 bit Video DAC interface for driving a standard 15 pin VGA type connector interface
- USB: Cypress CY7C68000 USB 2.0 UTMI Transceiver connecting to either USB-A (Host) or USB-B (Peripheral) style connectors for USB 2.0 interfaces
- PS2 Mouse and Keyboard: Two PS2 style connectors providing standard PS2 style interfaces (e.g. Mouse and Keyboard)
- JTAG: Connectors supporting both Parallel Cable III (Flying leads) and Parallel cable IV (2x7pin 2mm header) for JTAG programming and configuration download to FPGA
- Memory expansion connector: 120 pin expansion connector: AMP 177-984-5, allowing connection to mezzanine board or to a logic analyser with appropriate adapter
- User I/O's
 - LVDSIO: One 2 x 20 pin 0.1" header providing 12 LVDS signal pairs for User defined signals. Can also be configured and used as 24 standard LVTTTL/LVCMOS single ended I/O signals for user defined signals
 - GENIO: Three 2 x 20 pin 0.1" headers, each providing 20 user definable I/O signals (total 60 user defined signals)
 - PIO: 16 bit PIO port accessible via expansion connector (20 pin 0.1" header), compatible with existing GR-PCI accessory products to provide easy expansion of optional 2 x RS232 / 2 x RS422 / 2 x LVDS or 2 x CANBUS interfaces. Can also be configured to provide 16 user defined I/O signals
- Switches
 - 8 pole DIP switch for User Definable functions
 - Two user definable push-button switches

- One push-button switch for system reset and one push-button switch for FPGA (re)configuration
- Indicators
 - Four User definable LED indicators.
 - One indicating power on board
 - One indicating FPGA programming status
 - One indicating Prom-busy
 - One indicating USB current fault

2.2.2 Board parts to be used

The parts of the development board used in the project beside the FPGA are explained with some detail as follows.

- Memory Organization

The *GR-XC3S-1500* provides FLASH (PROM) memory and SDRAM on board, providing the necessary memory control and Address/Data signals. The memory organisation on the *GR-XC3S-1500* board is represented in Figure 2.4.

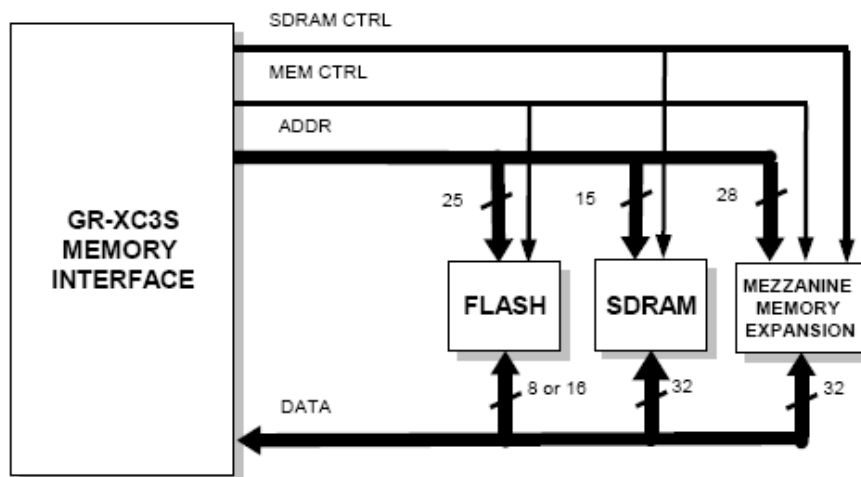


Figure 2.4. Board Memory Interface

- PROM 64 Mbit (8 Mbyte) FLASH (organised x8 or x16 bit)
- SDRAM 512 Mbit (64 Mbyte) PC-133 SDRAM on board (32 bits wide interface)

The provided mezzanine memory expansion is not used in this project. The address, data and standard memory control signals (MEM CTRL) are made available for external use on the J9 expansion connector.

The FLASH memory is normally not write protected. However, if a zero-ohm resistor is installed for JP1, the FLASH memory can be configured to protected it from Write/Erase operations. When writing or block-erasing the FLASH memory, LED D12 will illuminate. The FLASH memory normally operates in 8bit wide memory mode. However, if a zero-ohm resistor is installed for JP2, the FLASH memory can be configured to instead operate in 16 bit data mode.

- Oscillators

A number of oscillators are provided on the *GR-XC3S-1500* board as represented in Figure 2.5.

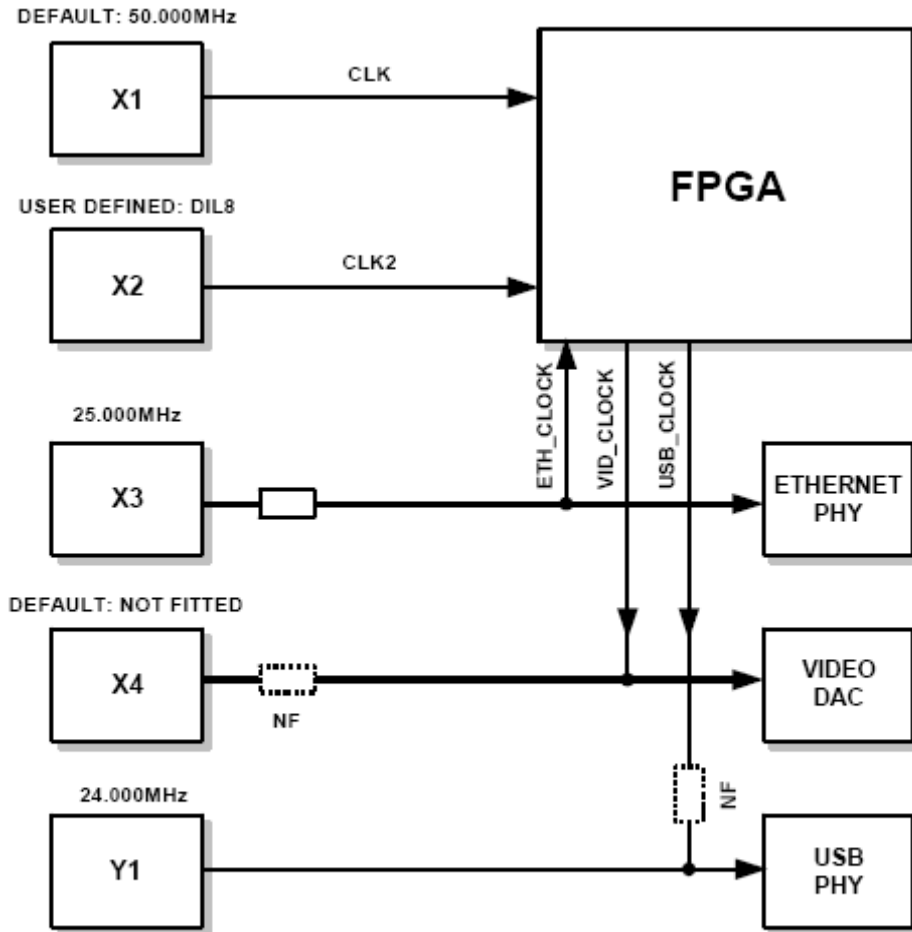


Figure 2.5. Configuration of Oscillators

The main oscillator (*CLK*) for the GR-XC3S-1500 device is 50MHz precision oscillator, X1. A DIL8 socket is provided in order to allow the user to install their own user defined oscillator to provide *CLK2* if required.

The Ethernet PHY requires a 25.000MHz clock which is provided by X3. This clock is also an input to the FPGA (*ETH_CLK*).

In the default configuration X4 is not fitted, and the clock for the Video DAC is intended to be generated by the logic inside the FPGA.

The clock for the USB PHY controller is provided by Y1, it is not used in this project.

- Serial interfaces

The *GR-XC3S-1500* provides two serial interfaces with standard SUB-D 9 pin female connectors, and RS232 line driver/receiver chips. The IC's implemented on board are capable of supporting data rates up to 1 M baud.

- Ethernet Interface

The *GR-XC3S-1500* board incorporates a Intel LXT971A 3.3V Dual-Speed Fast Ethernet PHY Transceiver device providing 10/100Mbit/s Ethernet interface, with RJ45 10/100Mbit Ethernet connector. To use this feature it is necessary to implement the Ethernet MAC function in the logic of the FPGA. Communication and data transfer between the MAC and PHY occurs over a standard MII interface. To utilize the Ethernet interface in a Leon system, appropriate driver software will be required depending on the features and operating system which the user wishes to implement.

- JTAG

Connector J9 allows a Xilinx Parallel Cable IV type ribbon cable (2x7pin 2mm connector) to be connected to the board. Alternatively connector J10 allows either flying leads or a low-cost JTAG programming cable such as the Digilent JTAG3 or USB type cables to be connected to the board. Both types of cable can be used with the Xilinx iMPACT software for programming of the Platform Configuration proms and for configuration of the FPGA.

2.3 LEON3 Design Configuration

The provided files by Gaisler Research which contains the VHDL model to be used could be “manually” modified in order to choose the features to be placed on the FPGA. However, a GUI xconfig tool which allows to configure the model is provided. The main window of the GUI configuration tool can be seen in figure 2.6 [3,6].

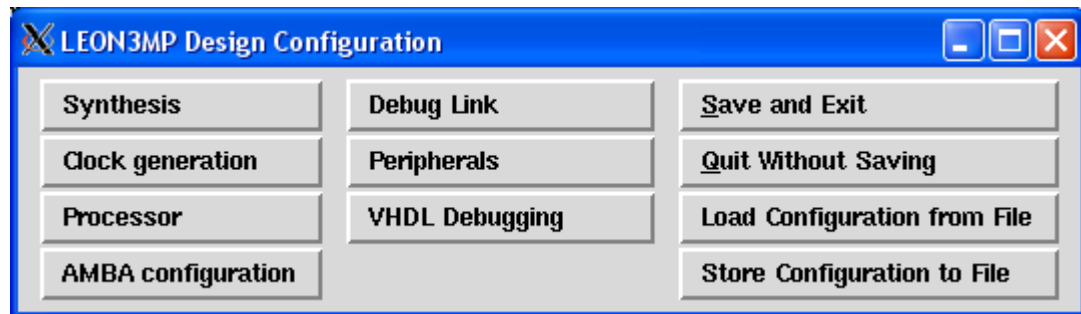


Figure 2.6. LEON3 GUI configuration tool

The GUI configuration tool can be accessed by executing the following command line on a cygwin (or Unix) bash shell, on the directory corresponding to the VHDL model template for the board (Leon_VHDL_model\Grlib-Leon3\grlib-gpl-1.0.17 -b2710\designs\leon3-gr-xc3s-1500):

```
$ make xconfig
```

The GUI configuration tool allows to add or delete features of the VHDL model and set parameters for the synthesis and VHDL debugging. The set configuration is saved at a “config.vhd” file by clicking the “save and exit” button. The figures 2.7 and 2.8 shows the configuration windows for the processor and its peripherals, respectively.

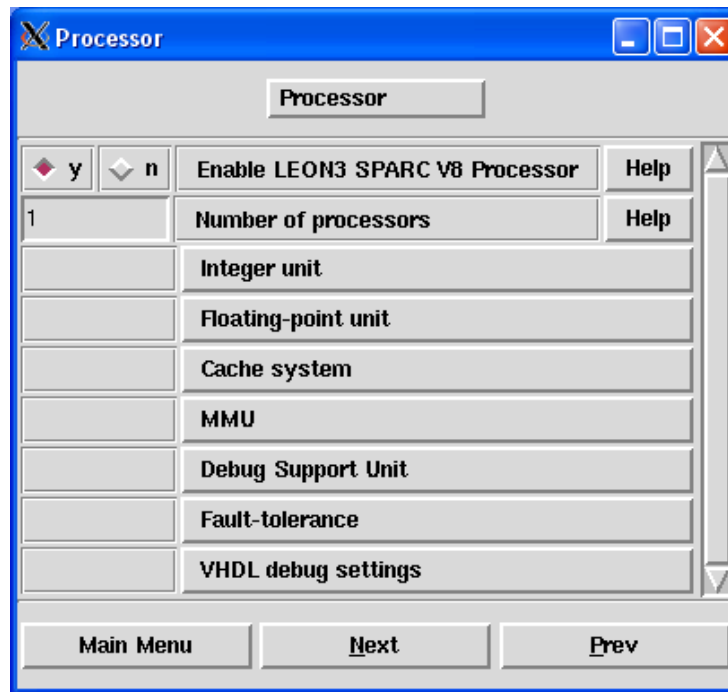


Figure 2.7. Processor configuration window

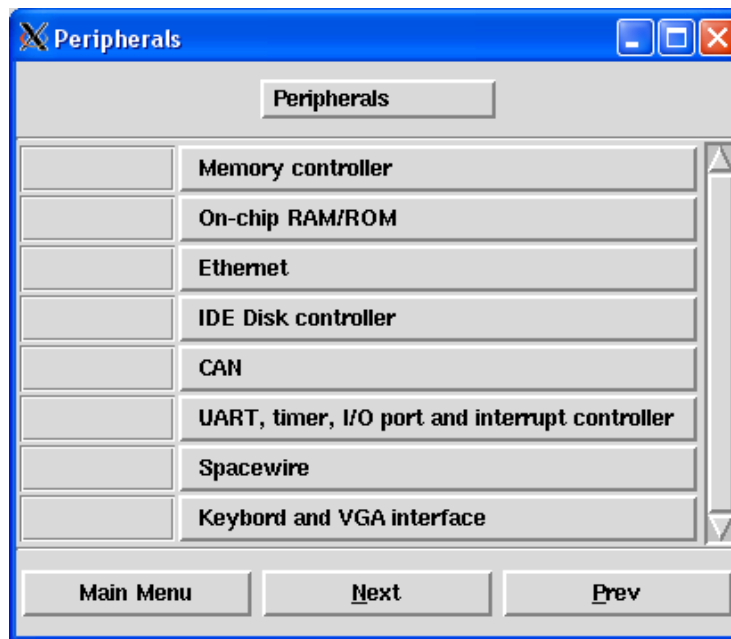


Figure 2.8. Peripherals configuration window

2.4 Procedure to place the processor on the FPGA

A design model containing the VHDL files of LEON3 for the GR-XC3S-1500 development board is provided by Gaisler. The available software tool to synthesize the provided files was the ISE® Foundation™ Package from Xilinx (version 9.2i) [7]. The ISE Foundation package is in the list of recommended software by the provider to perform the synthesis operation of the provided files.

The empirically “successful” procedure to place the correspondent .bit file on the FPGA is described by the following steps:

- 1) Synthesize XST with the ISE 9.2i Project Navigator. The project is opened at the directory at the package provided by Gaisler Research:

```
Leon_VHDL_model\Grlib-Leon3\grlib-gpl-1.0.17-b2710\designs\leon3-gr-  
xc3s-1500
```

- 2) On a cygwin bash shell and at the previously indicated directory, the following command line is executed in order to perform the Place&Route in bash mode:

```
$ make ise-map
```

- 3) After the Place&Route is successfully completed, the *.bit file is generated by executing the following command line (on the same cygwin bash shell):

```
$ make ise
```

After the .bit file is generated, it can be download to the FPGA with iMPACT (another program in the ISE Package) via JTAG cable. The *.bit file download over the FPGA can be done from the iMPACT GUI. By this procedure the downloaded

FPGA configuration is lost every time the board is turned off. The same procedure can be done from the cygwin bash shell by executing the following command line:

```
$ make ise-prog-fpga
```

The FPGA configuration PROMs can be programmed using the obtained *.bit file by executing the following command on the cygwin shell window (the file is also downloaded via JTAG by this way):

```
$ make ise-prog-prom
```

Downloading the .bit file into the PROMs configures the board to download automatically the file into the FPGA the *.bit file configuration every time that it is turned on, making stand alone operation of the board possible.

A precompiled *.bit file is also provided by Gaisler Research, it can be downloaded into the PROMs by ISE executing the following command in the cygwin bash shell (also via JTAG):

```
$ make ise-prog-prom-ref
```

The results obtained with both *.bit files are compared at the “test and results” section of this chapter.

2.5 The GRMON LEON Monitor

GRMON [4,5,6] is a debug monitor developed by Gaisler Research for the LEON Debug Support Unit (DSU), providing a non-intrusive debug environment on real target hardware. The LEON DSU can be controlled through any AHB master, and GRMON supports communications through the dedicated DSU UART or a PCI interface if available. GRMON can operate in two modes: stand alone or attached to gdb. Numerous commands are available to examine data, insert breakpoints and advance execution.

GRMON is used in stand alone mode during this project. LEON applications can be loaded and debugged using a command line interface in stand alone mode.

Some of the operations that are simplified using GRMON are listed as follows:

- Read/write access to all registers and memory
- Dissassembler and trace buffer management
- Downloading and execution of LEON applications
- Breakpoint and watchpoint management
- Auto-probing and initialization of LEON peripherals and memory settings

GRMON allows to access the LEON processor by several ways, in this project JTAG cable and parallel port (as set by default) were the only used. The following command line (in the GRMON installed directory) in a cygwin or Unix (DOS is also supported) bash shell is executed to initialize GRMON operation with JTAG cable:

```
$ grmon -jtag
```

Several options can be used for GRMON initialization, as will be seen in further sections and chapters. An evaluation version of GRMON is included in the LEON software tools provided by Gaisler Research.

2.6 Tests and Results

The bit file obtained following the procedure described at the section 2.3 was by the methods also specified downloaded into the FPGA. The processor placed on the board was accessed with GRMON, despite some communication errors and a stack pointer warning, the GRLIB library components seemed to be correctly installed, as can be seen in figure 2.9.

```

C:\Documents and Settings\agonzalez\Desktop\grmon-eval-1.1.32\grmon-eval\cygwin\grmo...
DCOM communication error, retrying ...
warning: stack pointer not set
DCOM communication error, retrying ...
DCOM communication error, retrying ...
Component                               Vendor
LEON3 SPARC U8 Processor                 Gaisler Research
AHB Debug UART                          Gaisler Research
AHB Debug JTAG TAP                       Gaisler Research
SUGA frame buffer                        Gaisler Research
GR Ethernet MAC                          Gaisler Research
ATA Controller                           Gaisler Research
LEON2 Memory Controller                  European Space Agency
AHB/APB Bridge                           Gaisler Research
LEON3 Debug Support Unit                 Gaisler Research
Generic APB UART                         Gaisler Research
Multi-processor Interrupt Ctrl           Gaisler Research
Modular Timer Unit                       Gaisler Research
Keyboard PS/2 interface                  Gaisler Research
Keyboard PS/2 interface                  Gaisler Research
General purpose I/O port                 Gaisler Research

Use command 'info sys' to print a detailed report of attached cores
grlib>
    
```

Figure 2.9. Built LEON3 processor on GRMON monitor

Also some applications were successfully downloaded over RAM. However, it was not possible to execute the downloaded applications, not even a “hello world” test program. The obtained error message was “processor not in debug mode”, making the board reset necessary to continue. The Flash memory related instructions were not possible to execute under these conditions. A failed attempt to access the registers is reported in figure 2.10.

```

C:\Documents and Settings\agonzalez\Desktop\grmon-eval-1.1.32\grmon-eval\cygwin\grmo...
    apb: 80000600 - 80000700
DCOM communication error, retrying ...
04.01:01d Gaisler Research GR Ethernet MAC (ver 0x0)
    ahb master 4, irq 12
    apb: 80000b00 - 80000c00
    edcl ip 192.168.0.51, buffer 2 kbyte
05.01:024 Gaisler Research ATA Controller (ver 0x0)
    ahb master 5, irq 10
    ahb: fffa0000 - fffa0100
    Device 0: <None>
    Device 1: <None>
00.04:00f European Space Agency LEON2 Memory Controller (ver 0x1)
    ahb: 00000000 - 20000000
    ahb: 20000000 - 40000000
    ahb: 40000000 - 80000000
    apb: 80000000 - 80000100
    8-bit prom @ 0x00000000
01.01:006 Gaisler Research AHB/APB Bridge (ver 0x0)
    ahb: 80000000 - 80100000
02.01:004 Gaisler Research LEON3 Debug Support Unit (ver 0x1)
    ahb: 90000000 - a0000000
01.01:00c Gaisler Research Generic APB UART (ver 0x1)
    irq 2
    apb: 80000100 - 80000200
DCOM communication error, retrying ...
    baud rate 5000000
02.01:00d Gaisler Research Multi-processor Interrupt Ctrl (ver 0x3)
    apb: 80000200 - 80000300
03.01:011 Gaisler Research Modular Timer Unit (ver 0x0)
    irq 8
    apb: 80000300 - 80000400
    8-bit scaler, 2 * 32-bit timers, divisor 40
04.01:060 Gaisler Research Keyboard PS/2 interface (ver 0x1)
    irq 4
    apb: 80000400 - 80000500
05.01:060 Gaisler Research Keyboard PS/2 interface (ver 0x1)
    irq 5
    apb: 80000500 - 80000600
08.01:01a Gaisler Research General purpose I/O port (ver 0x0)
    apb: 80000800 - 80000900
grlib> reg
DCOM communication error, retrying ...
DCOM communication error, retrying ...
Cannot continue, processor not in debug mode
DCOM communication error, retrying ...
Cannot continue, processor not in debug mode

psr: 00000000  wim: 00000000  tbr: 00000000  y: 00000000
DCOM communication error, retrying ...

pc: 00000000  unimp                                DCOM communication error, retrying
...

npc: 00000000  unimp                                DCOM communication error, retrying
...

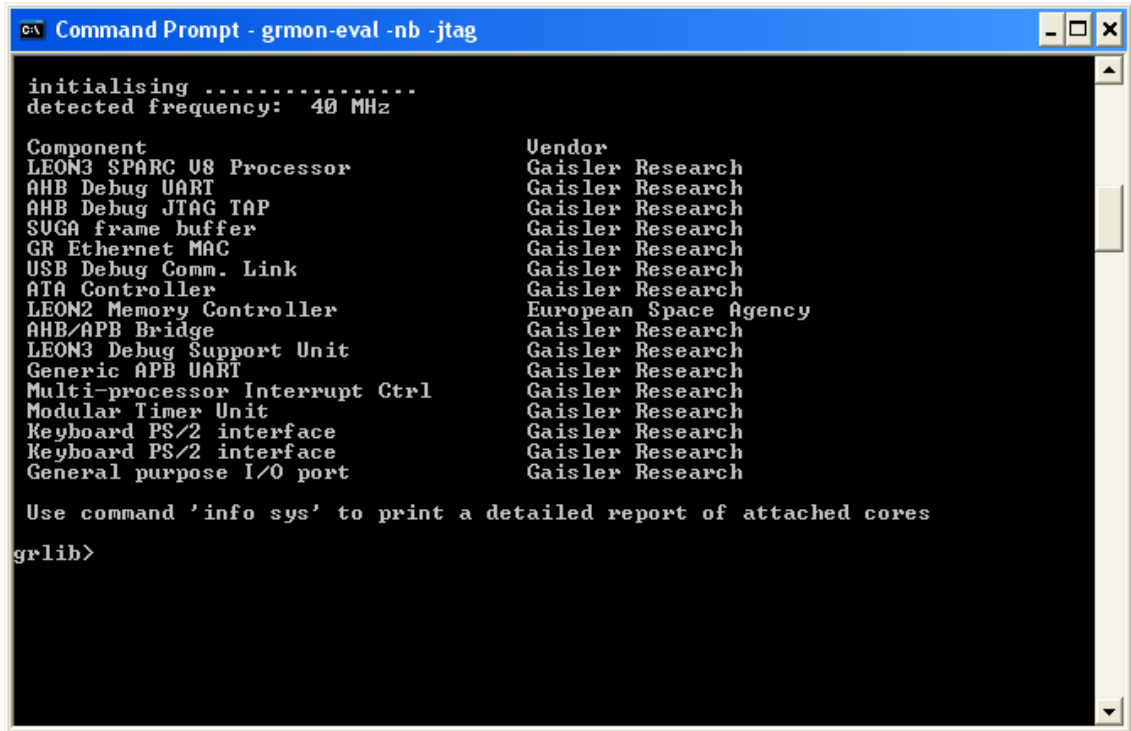
grlib>

```

Figure 2.10. Failures on the built processor

The bitgen report and Place&Route report files presented several warnings but almost all of them were related to the USB port devices.

After the reported results, the precompiled bitfile provided by Gaisler Research was downloaded on the FPGA. No communications errors or stack pointer warnings appeared in that case, as can be seen in figure 2.11.



```
Command Prompt - grmon-eval -nb -jtag
initialising .....
detected frequency: 40 MHz

Component                               Vendor
LEON3 SPARC V8 Processor                 Gaisler Research
AHB Debug UART                           Gaisler Research
AHB Debug JTAG TAP                       Gaisler Research
SUGA frame buffer                        Gaisler Research
GR Ethernet MAC                          Gaisler Research
USB Debug Comm. Link                    Gaisler Research
ATA Controller                           Gaisler Research
LEON2 Memory Controller                  European Space Agency
AHB/APB Bridge                           Gaisler Research
LEON3 Debug Support Unit                 Gaisler Research
Generic APB UART                         Gaisler Research
Multi-processor Interrupt Ctrl           Gaisler Research
Modular Timer Unit                       Gaisler Research
Keyboard PS/2 interface                  Gaisler Research
Keyboard PS/2 interface                  Gaisler Research
General purpose I/O port                 Gaisler Research

Use command 'info sys' to print a detailed report of attached cores
grlib>
```

Figure 2.11. Precompiled LEON3 processor on GRMON monitor

Operations over the registers, and applications download (into RAM and Flash) and execution were successfully completed under these conditions.

After the obtained results and a relatively high time investment to solve the problems of the “manually” generated processor, and after analyzing the possible problem causes such as the ISE version used, it was decided to continue the board testing using the precompiled processor. The possibility of physical damages on the board was dismissed since it was possible to run on it the precompiled processor.

The LEON3 VHDL model flexibility cannot be discarded by the obtained results, but it neither cannot be proved due to the choice of continue the project using the precompiled version. However, the LEON3 processor reliability for the support of an embedded operating system was possible to test by the procedures performed in the following chapter. The fully-operative processor version allowed to continue with the project's next step.

Chapter 3

Firmware Implementation

This chapter explains the procedure to place an Operating System on the Board, after the processor has been installed.

3.1 The SnapGear Linux

SnapGear Linux is a full source package provided by Gaisler. It contains kernel, libraries and application code for rapid development of embedded Linux systems. MMU and non-MMU Leon configurations are supported. A single cross compilation toolchain is provided, which is capable of compiling the kernel and applications for any configuration.

The Linux kernel can be configured using a graphical interface. Drivers and features can be removed in order to save space. On LEON3 systems the AMBA plug&play information is used to detect devices and load their respective software drivers.

A small boot loader is incorporated into the SnapGear Linux software distribution, it is specially designed for LEON processors. Its main propose is initialize basic hardware, such as memory controllers and console output for debugging, before launching LEON Linux.. The boot loader is stored in a non-volatile memory at the address where the LEON processor reads first its instructions to be executed, usually stored in flash at address 0.

Supported hardware on the latest version is presented in the following list.

New hardware is being added constantly:

- LEON2, with or without MMU, FPU, MUL/DIV.
- LEON3, with or without MMU, FPU, MUL/DIV.
- LEON3 multi processor systems, SMP
- APBUART
- GPTIMER
- GRETH 10/100 and Gbit
- OpenCores 10/100 Ethernet MAC
- SMC91x 10/100 Ethernet MAC
- APBPS2
- APBVGA
- GRUSBHC
- GRVGA
- ATACTRL
- GRPCI
- GRETH over PCI
- GR/OpenCores

3.2 SnapGear Linux Compiling

The SnapGear Linux compiling processes was completed successfully in a Unix host. Several problems appeared using cygwing, so it was discarded.

The procedure to compile SnapGear Linux is composed by the following steps:

- 1) Installing the toolchain
- 2) Installing the SnapGear distribution
- 3) Configuring Linux
- 4) Building SnapGear

The indicated steps are described as follows:

3.2.1 Installing the Toolchain

Before compile SnapGear Linux, a toolchain able to compile LEON SPARC Linux binaries must be chosen, it was selected the sparc-linux-3.4.4 toolchain. The chosen toolchain was installed in the host's /opt directory. After being installed, the toolchain was added to the PATH variable using the following command line:

```
$ export PATH=$PATH:/opt/sparc-linux-3.4.4/bin
```

After installing the toolchain it is possible to cross compile applications for SPARCC LEON Linux, as indicated by the following command line (the brackets are representative):

```
$ sparc-linux-gcc -o [executable name] [source file name .c]
```

The following command line demonstrates that the output binary is a SPARC binary:

```
$ file [executable name]
```

Obtaining the following output message (for an executable file named “args”):

```
args: ELF 32-bit MSB executable, SPARC, version 1 (SYSV), dynamically  
linked (uses shared libs), not stripped
```

3.2.2 Installing the SnapGear Distribution

The provided SnapGear distribution is compressed with tar and bunzip2, its installation on the Linux host in use can be done by executing the following command line (xx and yy values depends on the version).

```
$ tar -xjf snapgear-xx-yy.tar.bz2
```

3.2.3 Configuring Linux

The SnapGear distribution includes an easy to use graphical interface (GUI). From the GUI is possible to select processor, Linux version, C library and what applications will be include into the root file system (ROMFS image) accessed by Linux during run time. Is also possible to configure the boot loader parameters and configure the Linux kernel.

The configuration GUI can be invoked by executing the following command line in the SnapGear distribution installed directory:

```
$ make xconfig
```

The GUI main window can be seen in figure 3.1.

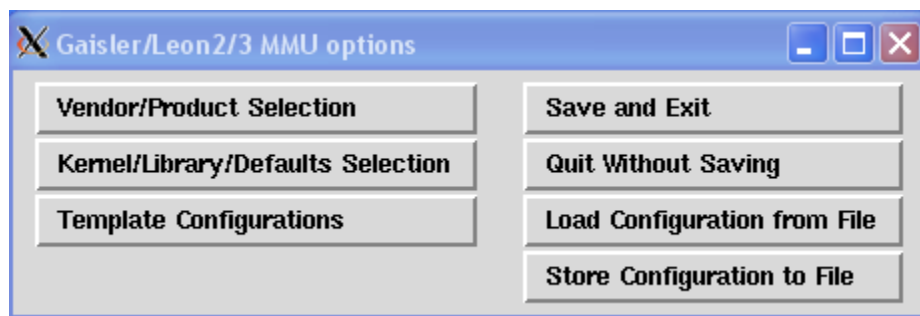


Figure 3.1. SnapGear main configuration GUI

A template configuration for the board is also included, this can be set by clicking the “Template Configuration” button on the GUI, then choosing it (GR-XC3s-1500) in the window that appears and activating the update configurations option, as shown in figure 3.2.

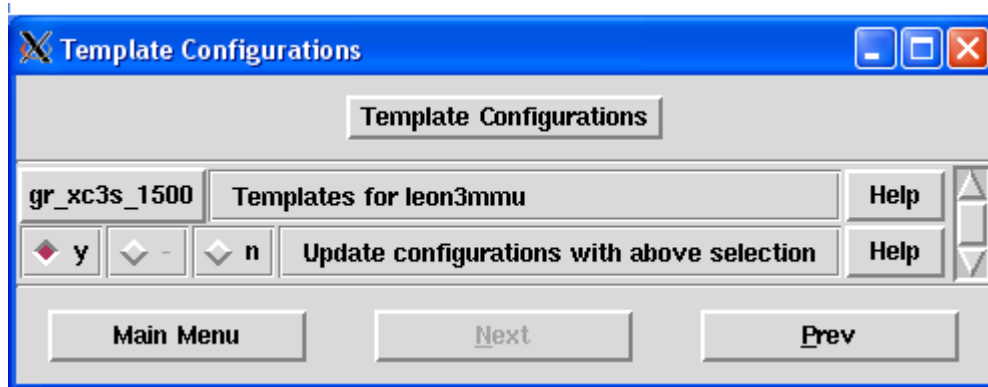


Figure 3.2. Template Configuration selection

By activating the “Update configurations with above selection” option, the launch of the GUIs for the configuration of the kernel and other settings are suspended, and all the settings are configured as indicated in the chosen template. However, as it can be seen in figure 3.1, there is also an option for load a configuration from a file, so the board template configuration and customized settings can be mixed.

In this project, the activation of the template for the GR-xc3s-1500 Board was considered enough. This selection was compatible with the modifications over the Linux kernel indicated at the section 3.3 (Modifications over the SnapGear Kernel).

However, desired changes can be performed by setting the provided options by clicking other two buttons on the GUI, Vendor/Product Selection and

Kernel/Library/Defaults Selection. The application selection menus obtained for each buttons are shown on figures 3.3 and 3.4.

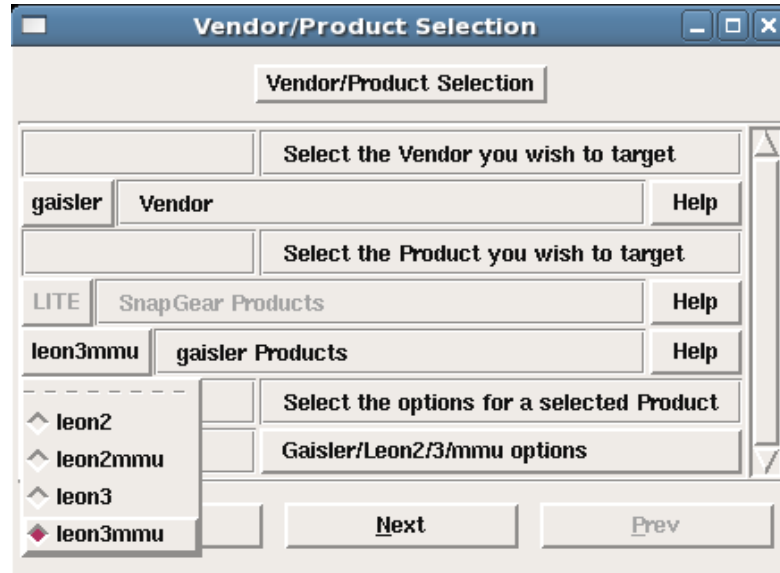


Figure 3.3. Vendor/Product section

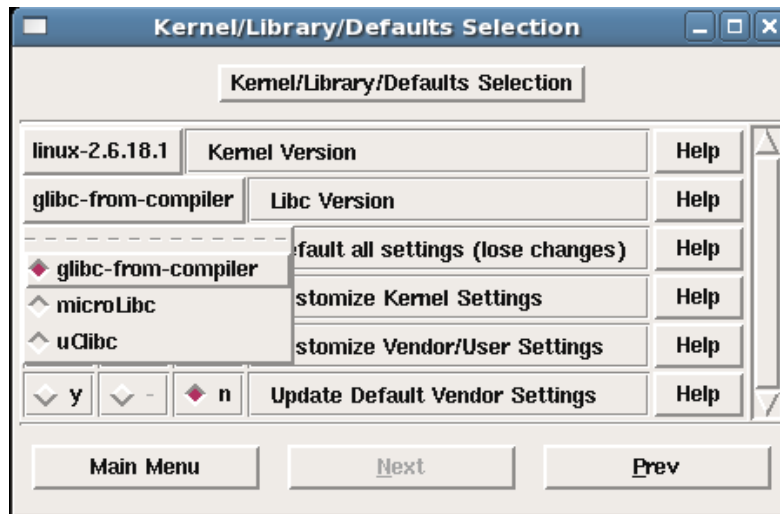


Figure 3.4: kernel/library/defaults selection menu

Changes in this options are not saved if a template has been selected as previously indicated, but a template can be used if the button “load configuration from file” is used. Some of the allowed features to modify are presented as follows:

- Processor Type and MMU
- C library
- Kernel version
- Configuring the boot loader
- Configuring the Linux 2.6.x kernel (MMU, i.e. only LEON3)
- Configuring the Linux 2.0.x kernel (no MMU)

Special GUI configurations utilities are provided for the both kernels, however, they do not perform automatically changes needed as a consequence of eventual modifications made by the user, leading to possible errors on the compiling process. The GUI configuration utility for the Linux 2.6.x case is shown on figure 3.5.

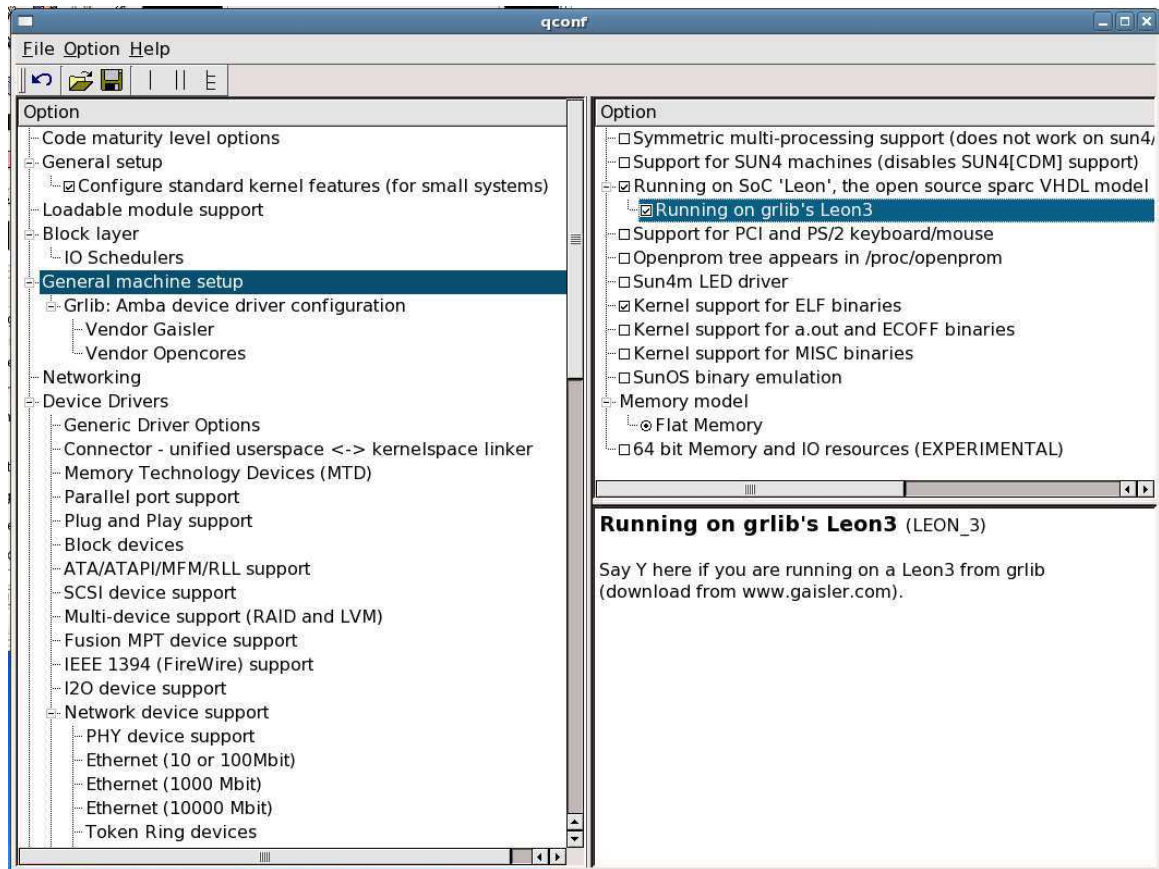


Figure 3.5. Linux 2.6.x kernel GUI configuration utility

It was considered to use a JFFS2 file system on flash memory but many compilation problems appeared and many files seemed to be necessarily (without help of the GUI) modified. Similar problems appeared trying to configure the USB port in order to manage an external memory device. The kernel GUI configuration was not very useful in those cases. However the data storage on the board was not considered a critical issue, since it is also possible to transmit data immediately after it is processed.

Actual changes on the kernel were performed on some files without the help of the previously described GUI, as it can be seen on section 3.3.

3.2.4 Building SnapGear

The Linux SnapGear compilation is executed by the “make” command, as indicated in the following line in the SnapGear distribution installed directory:

```
$ make
```

The generated images can be found at the /images subdirectory. Information about the compiling times can be found at the section 3.5 (Tests and Results).

3.3 Modifications over the SnapGear Linux Kernel

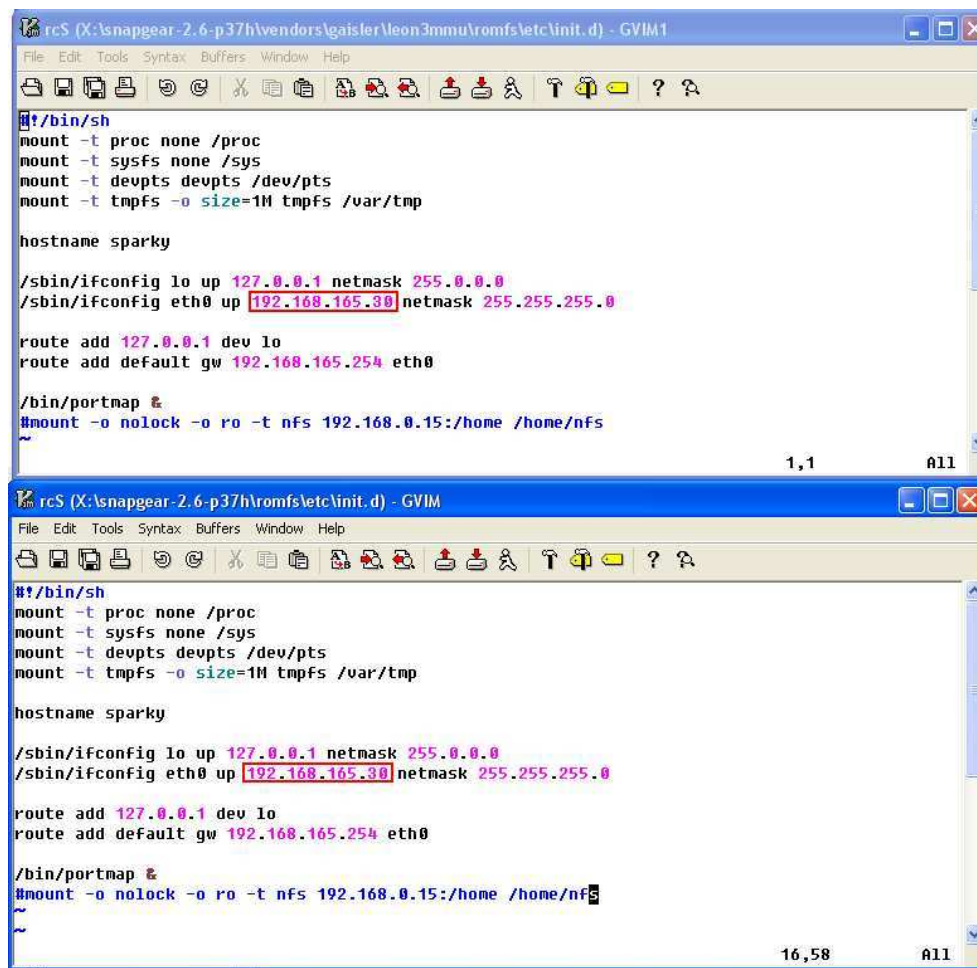
Some changes were performed over the Kernel in order to set the following parameters:

3.3.1 Board IP address

As previously said, the Board’s Ethernet port was chosen to test the communication with external stages. The following files were modified indicating the Board new IP address:

- vendors/gaisler/leon3mmu/romfs/etc/init.d/rcS
- romfs/etc/init.d/rcS

Both files contain the same information, as shown on figure 3.6. The assigned IP address for the Board was 192.168.165.30 (as also indicated on figure 3.6).



The image shows two screenshots of a terminal window running rcS. The top screenshot shows the initial configuration steps, including mounting filesystems, setting the hostname to 'sparky', and configuring network interfaces. The bottom screenshot shows the same configuration steps, but with the IP address for eth0 set to 192.168.165.30. The terminal output is as follows:

```
rcS (X:\snapgear-2.6-p37h\vendors\gaisler\leon3mmu\romfs\etc\init.d) - GVIM1
#?/bin/sh
mount -t proc none /proc
mount -t sysfs none /sys
mount -t devpts devpts /dev/pts
mount -t tmpfs -o size=1M tmpfs /var/tmp

hostname sparky

/sbin/ifconfig lo up 127.0.0.1 netmask 255.0.0.0
/sbin/ifconfig eth0 up 192.168.165.30 netmask 255.255.255.0

route add 127.0.0.1 dev lo
route add default gw 192.168.165.254 eth0

/bin/portmap &
#mount -o nolock -o ro -t nfs 192.168.0.15:/home /home/nfs
~
1,1 011
```

```
rcS (X:\snapgear-2.6-p37h\romfs\etc\init.d) - GVIM
#?/bin/sh
mount -t proc none /proc
mount -t sysfs none /sys
mount -t devpts devpts /dev/pts
mount -t tmpfs -o size=1M tmpfs /var/tmp

hostname sparky

/sbin/ifconfig lo up 127.0.0.1 netmask 255.0.0.0
/sbin/ifconfig eth0 up 192.168.165.30 netmask 255.255.255.0

route add 127.0.0.1 dev lo
route add default gw 192.168.165.254 eth0

/bin/portmap &
#mount -o nolock -o ro -t nfs 192.168.0.15:/home /home/nfs
~
16,58 011
```

Figure 3.6. Board IP address setting

It is important to notice that the directory /romfs is created during the kernel first compilation. However, if /romfs and some subdirectories are manually created they will not be overwritten. Only missing directories and default settings will be added during the compilation process.

3.3.2 Adding programs to be executed on the board

The executable files of the programs to be executed on the Board must be added to the /romfs directory. By doing this, the programs will be found at the SnapGear

Other actions can be called besides `sysinit` and `respawn`, but are not needed in this project. As can be presumed, a wrong configuration of the `inittab` file can make totally inoperative the operating system.

3.4 SnapGear Linux Installing Procedure

After the Kernel compilation has been completed, the obtained images can be found at the `/images` directory. The `image.flashbz` file is the appropriate one to use, since it can be stored in Flash memory and contains a complete boot loader.

The chosen procedure to place `image.flashbz` in the processor's Flash memory was using the GRMON monitor via JTAG cable. GRMON is invoked on the server as indicated in the following command line:

```
$ grmon-eval -nb -jtag
```

The `-nb` option allows Linux to take care of traps instead of having GRMON to stop the execution. After verify the processor operation the flash memory shall be erased, as indicated by the following execution line on `grmon`:

```
GRMON> flash erase all
```

Then, the image can be loaded into Flash memory:

```
GRMON> flash load image.flashbz
```

After the download is completed, SnapGear can be executed by pressing the board's reset button or executing the following line:

GRMON> run 0

As previously said, for the chosen Kernel configuration SnapGear will run on the serial port 1 of the board, at 38400 baud. By the described procedure SnapGear will run in stand alone operation each time the Board is turned on or reset. The next section explains the details the about SnapGear usage.

3.5 Tests and Results

The compiling times for SnapGear were measured for the previously indicated configuration (immediately after execute the make command). If the SnapGear distribution in use is compiled by first time, it takes around 5 minutes. If it is recompiled it takes around 40 seconds, since are used almost all the files built for the first compilation, despite eventual minor changes on the kernel (such as the indicated in the section 3.3).

The downloading time for the image on the Board Flash memory was also measured, registering around 15 minutes, the download speed was 24.8 Kbit/s as can be seen in figure 3.8. The erasing of the flash memories (a step previous to the image download) takes around 2 minutes.

```

Command Prompt - grmon-eval -jtag -nb
grlib> flash load image25062.flashbz
section: .text at 0x0, size 2736244 bytes
total size: 2736244 bytes <24.8 kbit/s>
read 37 symbols
entry point: 0x00000000
grlib> flash erase all
Erase in progress
Block @ 0x00000000 : code = 0x80 OK
Block @ 0x00020000 : code = 0x80 OK
Block @ 0x00040000 : code = 0x80 OK
Block @ 0x00060000 : code = 0x80 OK
Block @ 0x00080000 : code = 0x80 OK
Block @ 0x000a0000 : code = 0x80 OK
Block @ 0x000c0000 : code = 0x80 OK
Block @ 0x000e0000 : code = 0x80 OK
Block @ 0x00100000 : code = 0x80 OK
Block @ 0x00120000 : code = 0x80 OK
Block @ 0x00140000 : code = 0x80 OK
Block @ 0x00160000 : code = 0x80 OK
Block @ 0x00180000 : code = 0x80 OK
Block @ 0x001a0000 : code = 0x80 OK
Block @ 0x001c0000 : code = 0x80 OK
Block @ 0x001e0000 : code = 0x80 OK
Block @ 0x00200000 : code = 0x80 OK
Block @ 0x00220000 : code = 0x80 OK
Block @ 0x00240000 : code = 0x80 OK
Block @ 0x00260000 : code = 0x80 OK
Block @ 0x00280000 : code = 0x80 OK
Block @ 0x002a0000 : code = 0x80 OK
Block @ 0x002c0000 : code = 0x80 OK
Block @ 0x002e0000 : code = 0x80 OK
Block @ 0x00300000 : code = 0x80 OK
Block @ 0x00320000 : code = 0x80 OK
Block @ 0x00340000 : code = 0x80 OK
Block @ 0x00360000 : code = 0x80 OK
Block @ 0x00380000 : code = 0x80 OK
Block @ 0x003a0000 : code = 0x80 OK
Block @ 0x003c0000 : code = 0x80 OK
Block @ 0x003e0000 : code = 0x80 OK
Block @ 0x00400000 : code = 0x80 OK
Block @ 0x00420000 : code = 0x80 OK
Block @ 0x00440000 : code = 0x80 OK
Block @ 0x00460000 : code = 0x80 OK
Block @ 0x00480000 : code = 0x80 OK
Block @ 0x004a0000 : code = 0x80 OK
Block @ 0x004c0000 : code = 0x80 OK
Block @ 0x004e0000 : code = 0x80 OK
Block @ 0x00500000 : code = 0x80 OK
Block @ 0x00520000 : code = 0x80 OK
Block @ 0x00540000 : code = 0x80 OK
Block @ 0x00560000 : code = 0x80 OK
Block @ 0x00580000 : code = 0x80 OK
Block @ 0x005a0000 : code = 0x80 OK
Block @ 0x005c0000 : code = 0x80 OK
Block @ 0x005e0000 : code = 0x80 OK
Block @ 0x00600000 : code = 0x80 OK
Block @ 0x00620000 : code = 0x80 OK
Block @ 0x00640000 : code = 0x80 OK
Block @ 0x00660000 : code = 0x80 OK
Block @ 0x00680000 : code = 0x80 OK
Block @ 0x006a0000 : code = 0x80 OK
Block @ 0x006c0000 : code = 0x80 OK
Block @ 0x006e0000 : code = 0x80 OK
Block @ 0x00700000 : code = 0x80 OK
Block @ 0x00720000 : code = 0x80 OK
Block @ 0x00740000 : code = 0x80 OK
Block @ 0x00760000 : code = 0x80 OK
Block @ 0x00780000 : code = 0x80 OK
Block @ 0x007a0000 : code = 0x80 OK
Block @ 0x007c0000 : code = 0x80 OK
Block @ 0x007e0000 : code = 0x80 OK
Erase complete
grlib> flash load image25063.flashbz
section: .text at 0x0, size 2736512 bytes
total size: 2736512 bytes <24.8 kbit/s>
read 37 symbols
entry point: 0x00000000
grlib>

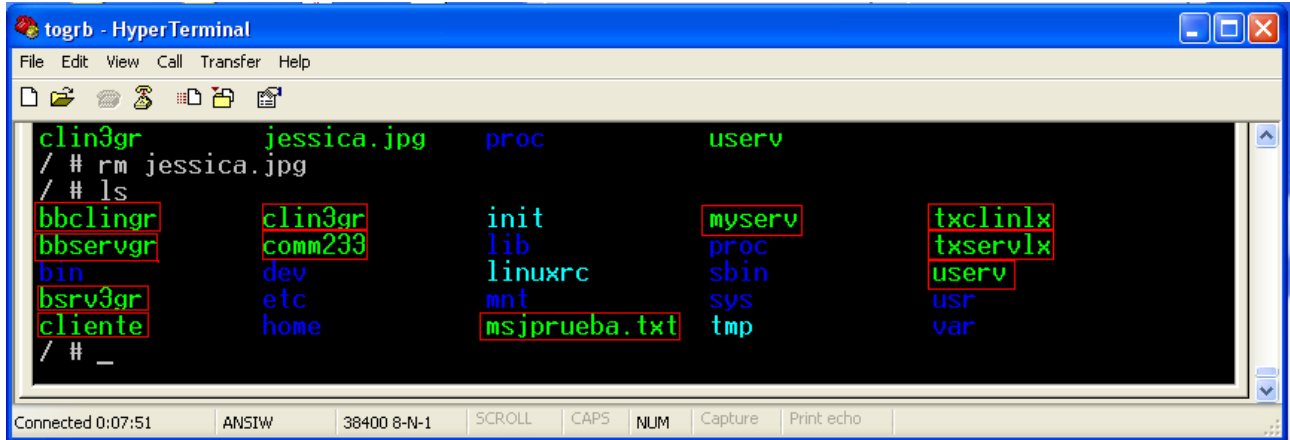
```

Figure 3.8. Flash erasing and SnapGear image download on GRMON

As set in the selected template, SnapGear Linux is executed at the serial port 1 of the board at 38400 baud. The board serial port is connected to a Windows host, where is used the HyperTerminal program to view the SnapGear execution. The SnapGear boot sequence read at HyperTerminal can be seen at the end of the section.

The presence on the root file system of the programs and files placed on the /romfs directory before the SnapGear compiling, was verified using the “ls”

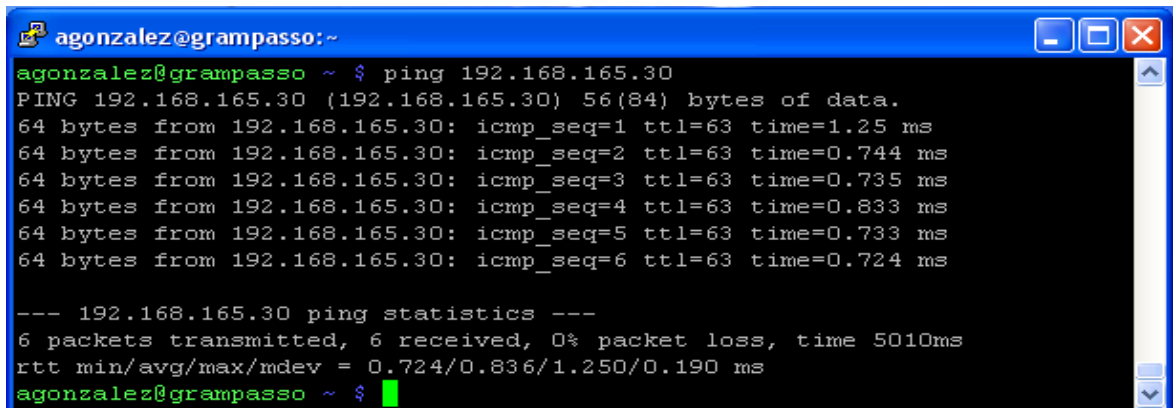
command on the SnapGear bash shell, as can be seen in the figure 3.9. This proof also demonstrates the SnapGear correct operation.



```
togrb - HyperTerminal
File Edit View Call Transfer Help
bbclngr  jessica.jpg  proc      userv
/ # rm jessica.jpg
/ # ls
bbclngr  clin3gr  init      myserv    txclinlx
bbservgr comm233  lib       proc      txservlx
bin      dev      linuxrc   sbin      userv
bsrv3gr  etc      mnt       sys       usr
cliente  home     msjprueba.txt tmp       var
/ # _
Connected 0:07:51  ANSIW  38400 8-N-1  SCROLL  CAPS  NUM  Capture  Print echo
```

Figure 3.9. Programs and files added to the root file system

The configured IP address of the Board was verified by two different methods. At figure 3.10 is shown the result of the execution of the ping command in a remote Unix host indicating the IP address set for the board. At figure 3.11, the IP address is read on the SnapGear bash shell by executing the “ipconfig” command.



```
agonzalez@grampasso:~
agonzalez@grampasso ~ $ ping 192.168.165.30
PING 192.168.165.30 (192.168.165.30) 56(84) bytes of data:
64 bytes from 192.168.165.30: icmp_seq=1 ttl=63 time=1.25 ms
64 bytes from 192.168.165.30: icmp_seq=2 ttl=63 time=0.744 ms
64 bytes from 192.168.165.30: icmp_seq=3 ttl=63 time=0.735 ms
64 bytes from 192.168.165.30: icmp_seq=4 ttl=63 time=0.833 ms
64 bytes from 192.168.165.30: icmp_seq=5 ttl=63 time=0.733 ms
64 bytes from 192.168.165.30: icmp_seq=6 ttl=63 time=0.724 ms

--- 192.168.165.30 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5010ms
rtt min/avg/max/mdev = 0.724/0.836/1.250/0.190 ms
agonzalez@grampasso ~ $
```

Figure 3.10. Board IP address verification from a remote host

```

togrb - HyperTerminal
File Edit View Call Transfer Help
starting pid 14, tty '': '/etc/init.d/rcS'
mount: mounting tmpfs on /var/tmp failed: Invalid argument
starting pid 26, tty '': '/bin/sh'
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 00:00:7A:CC:00:12
          inet addr:192.168.165.30  Bcast:192.168.165.255  Mask:255.255.255.0
          inet6 addr: fe80::200:7aff:fecc:12/64 Scope:Link
          UP BROADCAST RUNNING MTU:1500 Metric:1
          RX packets:11 errors:0 dropped:0 overruns:0 frame:0
          TX packets:5 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Base address:0xb00

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING MTU:16436 Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

/ # _
Connected 0:01:07  Auto detect  38400 8-N-1  SCROLL  CAPS  NUM  Capture  Print echo

```

Figure 3.11. Board IP address verification on SnapGear shell

Also the configured automatic program run at startup configuration was successfully verified, as can be seen in the SnapGear boot sequence presented as follows (The line “waiting for PPM file” corresponds to the program chosen to be run at startup):

```

decompress_kernel(to: 40000000, freemem:40397538, freemem_end:43ffdf38)
output_data:40000000, free_mem_ptr:40397538, free_mem_ptr_end:43ffdf38
- Inputbuf
fer [ptr: 3168, sz: 299017]
.....
done [sz:0x38f100], booting the kernel.

Booting Linux
Booting Linux...
PROMLIB: Sun Boot Prom Version 0 Revision 0
Linux version 2.6.21.1 (agonzalez@grampasso) (gcc version 3.4.4) #20 Thu Jun 25
12:23:10 CEST 2009
ARCH: LEON
      Vendors      Slaves
Ahb masters:
0( 1: 3| 0):  VENDOR_GAISLER  GAISLER_LEON3
1( 1: 7| 0):  VENDOR_GAISLER  GAISLER_AHBUART
2( 1: 1c| 0): VENDOR_GAISLER  GAISLER_AHBJTAG
3( 1: 63| 0): VENDOR_GAISLER  GAISLER_SVGA
4( 1: 1d| 0): VENDOR_GAISLER  GAISLER_ETHMAC
5( 1: 22| 0): VENDOR_GAISLER  Unknown device 22
6( 1: 24| 0): VENDOR_GAISLER  GAISLER_ATACTRL
Ahb slaves:
0( 4: f| 0):  VENDOR_ESA    ESA_MCTRL
+0: 0x0 (raw:0x3e002)
+1: 0x20000000 (raw:0x2000e002)
+2: 0x40000000 (raw:0x4003c002)

```


3 – Firmware Implementation

1(1: 6|0): VENDOR_GAISLER GAISLER_APB MST
+0: 0x80000000 (raw:0x8000ff2)
2(1: 4|0): VENDOR_GAISLER GAISLER_LEON3DSU
+0: 0x90000000 (raw:0x9000f002)
3(1: 24|10): VENDOR_GAISLER GAISLER_ATACTRL
+0: 0xffffa0000 (raw:0xa000ff3)
Apb slaves:
0(4: f|0): VENDOR_ESA ESA_MCTRL
+ 0: 0x80000000 (raw:0xff1)
1(1: c|2): VENDOR_GAISLER GAISLER_APB
+ 0: 0x80000100 (raw:0x10ff1)
2(1: d|0): VENDOR_GAISLER GAISLER_IRQMP
+ 0: 0x80000200 (raw:0x20ff1)
3(1: 11|8): VENDOR_GAISLER GAISLER_GPTIMER
+ 0: 0x80000300 (raw:0x30ff1)
4(1: 60|4): VENDOR_GAISLER GAISLER_KBD
+ 0: 0x80000400 (raw:0x40ff1)
5(1: 60|5): VENDOR_GAISLER GAISLER_KBD
+ 0: 0x80000500 (raw:0x50ff1)
6(1: 63|0): VENDOR_GAISLER GAISLER_SVGA
+ 0: 0x80000600 (raw:0x60ff1)
7(1: 7|0): VENDOR_GAISLER GAISLER_AHBUART
+ 0: 0x80000700 (raw:0x70ff1)
8(1: 1a|0): VENDOR_GAISLER GAISLER_PIOPORT
+ 0: 0x80000800 (raw:0x80ff1)
9(1: 1d|12): VENDOR_GAISLER GAISLER_ETHMAC
+ 0: 0x80000b00 (raw:0xb0ff1)
TYPE: Leon2/3 System-on-a-Chip
Ethernet address: 0:0:0:0:0
CACHE: direct mapped cache, set size 4k
CACHE: not flushing on every context switch
Boot time fixup v1.6. 4/Mar/98 Jakub Jelinek (jj@ultra.linux.cz). Patching kernel for srmmu[Leon2]/iommu
node 2: /cpu00 (type:cpu) (props:.node device_type mid mmu-nctx clock-frequency uart1_baud uart2_baud)
PROM: Built device tree from rootnode 1 with 918 bytes of memory.
DEBUG: psr.impl = 0xf fsr.ver = 0x7
Built 1 zonelists. Total pages: 15315
Kernel command line: console=ttyS0,38400 rdinit=/sbin/init
PID hash table entries: 256 (order: 8, 1024 bytes)
Todo: init master_i110_counter
Attaching grlib apbuart serial drivers (clk:40hz):
Console: colour dummy device 80x25
Dentry cache hash table entries: 8192 (order: 3, 32768 bytes)
Inode-cache hash table entries: 4096 (order: 2, 16384 bytes)
Memory: 60616k/65532k available (1536k kernel code, 4848k reserved, 180k data, 1904k init, 0k highmem)
Mount-cache hash table entries: 512
NET: Registered protocol family 16
NET: Registered protocol family 2
IP route cache hash table entries: 1024 (order: 0, 4096 bytes)
TCP established hash table entries: 2048 (order: 2, 16384 bytes)
TCP bind hash table entries: 2048 (order: 1, 8192 bytes)
TCP: Hash tables configured (established 2048 bind 2048)
TCP reno registered
io scheduler noop registered
io scheduler cfq registered (default)
grlib apbuart: 1 serial driver(s) at [0x80000100(irq 2)]
grlib apbuart: system frequency: 40000 khz, baud rates: 38400 38400
ttyS0 at MMIO 0x80000100 (irq = 2) is a Leon
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
loop: loaded (max 8 devices)
Probing GRETH Ethernet Core at 0x80000b00
PHY: Vendor 4de Device e Revision 2
10/100 GRETH Ethernet at [0x80000b00] irq 12. Running 100 Mbps full duplex
TCP cubic registered
NET: Registered protocol family 1

```
NET: Registered protocol family 10
IPv6 over IPv4 tunneling driver
Freeing unused kernel memory: 1904k freed
init started: BusyBox v1.8.2 (2009-03-25 12:04:03 CET)
starting pid 14, tty ": /etc/init.d/rcS'
mount: mounting tmpfs on /var/tmp failed: Invalid argument
starting pid 25, tty ": /etc/init.d/ingr'

waiting for PPM file...
```

The obtained results allow to use the SnapGear operation system to support complex programs able to manage the board communications and data processing, as it is reported in the following chapter.

Chapter 4

Software Implementation

This chapter explains how the software to be run on the Board was implemented. First, the following topics are overviewed:

- Type of files to be used for images before and after the compression
- Chosen compression algorithm
- Used tools for the compression
- Implementation of a communication channel between processes

Then, is exposed a description of the developed programs for the image compression, and building of communication links between the board and external stages (in order to exchange data, the compressed and to be compressed files).

The final section of the chapter provides an analysis of the tests over the developed programs and their respective results.

4.1 Used File Formats

PPM files [10] have been chosen to test the board capacities for image processing, using the JPEG algorithm over them and obtaining JPEG files; both formats are explained as follows:

4.1.1 PPM file format

The PPM format [10] was developed in the latter 1980s by Jef Poskanzer as a part of the Portable Bitmap Utilities (PBM); the name of the format is an acronym for “Portable Pixel Map”.

The format is a lowest common denominator color image file format, in which each pixel is defined by three ASCII decimal values between 0 and a specified maximum value. Each three values for each pixel represent an intensity scale of red, green and blue, respectively.

The PPM format is highly inefficient and redundant, since it contains a lot of information that can't be discerned by the human eye; however, the structure of the format makes very easy the development of programs for its reading and analysis.

There are two types of PPM files, P3 and P6; P3 files are entirely written in ASCII format making them easy to read such as any text file. P6 files have only their header written in ASCII format. P6 image values for each pixel are assigned entirely in binary, making them much less heavy than P3 files (even if corresponding to the same image). The following table compares two examples for the same image written in both formats:

P3	P6
P3 #any comment string 3 2 255 255 0 0 0 255 0 0 0 255 255 255 0 255 255 255 0 0 0	P6 #any comment string 3 2 255 !@#\$\$%^&*()_+{ }: "<

Table 4.1. PPM formats comparison

As it can be seen, the header structure is the same in the both cases:

- First line: Type of file, P3 or P6
- Second line: comment string
- Third line: columns and rows of the image
- Fourth line: maximum color value

In the case of P6 files, the value for each color of the pixel is usually represented by a byte, so in that case the maximum color value possible is 255. It was decided to use P6 files rather than P3 in this project.

4.1.2 JPEG file format

JPEG [12,13] is a standardized image compression mechanism. Its name stands for “Joint Photographic Experts Group”, the committee that created the standard and issued it in 1992, being approved in 1994 as ISO 10918-1. The name JPEG is also used to refer to the format of the files obtained using the compression standard.

The standard is designed to compress either full-color or gray-scale images, working at its best on photographs, naturalistic artwork, and similar material; unlike text, cartoons or line drawings.

Some information from the original image is lost during the compression process, which can't be recovered applying the decompression algorithm. JPEG takes advantage of known limitations of the human eye visual resolution; the degree of lossiness can be varied by adjusting compression parameters. Usually JPEG achieves 10:1 compression with little perceptible loss in image quality.

A JPEG image is composed by a sequence of segments, which are identifiable by markers at their beginning; each of them begins with a 0xFF byte, followed by another byte indicating what kind of marker it is. Beside the data containing the image itself, some segments contain information about applied coding methods and parameters values used for the compression. The markers used by the JPEG standard are indicated in the Table 4.2.

Marker	Code Assignment	Symbol	Description
Start Of Frame markers, non-differential, Huffman coding	0xFFC0	SOF0	Baseline DCT
	0xFFC1	SOF1	Extended sequential DCT
	0xFFC2	SOF2	Progressive DCT
	0xFFC3	SOF3	Lossless (sequential)
Start Of Frame markers, differential, Huffman coding	0xFFC5	SOF5	Differential sequential DCT
	0xFFC6	SOF6	Differential progressive DCT
	0xFFC7	SOF7	Differential lossless (sequential)
Start Of Frame markers, non-differential, arithmetic coding	0xFFC8	JPG	Reserved for JPEG extensions
	0xFFC9	SOF9	Extended sequential DCT
	0xFFCA	SOF10	Progressive DCT
	0xFFCB	SOF11	Lossless (sequential)
Start Of Frame markers, differential, arithmetic coding	0xFFCD	SOF13	Differential sequential DCT
	0xFFCE	SOF14	Differential progressive DCT
	0xFFCF	SOF15	Differential lossless (sequential)
Huffman table specification	0xFFC4	DHT	Define Huffman table(s)
Arithmetic coding conditioning specification	0xFFCC	DAC	Define arithmetic coding conditioning(s)
Restart interval termination	0xFFD0 through 0xFFD7	RST _m *	Restart with modulo 8 count “m”
Other markers	0xFFD8	SOI*	Start of image
	0xFFD9	EOI*	End of image
	0xFFDA	SOS	Start of scan
	0xFFDB	DQT	Define quantization table(s)
	0xFFDC	DNL	Define number of lines
	0xFFDD	DRI	Define restart interval
	0xFFDE	DHP	Define hierarchical progression
	0xFFDF	EXP	Expand reference component(s)
	0xFFE0 through	APP _n	Reserved for application segments
	0xFFEF	JPG _n	Reserved for JPEG extensions

	0xFFFF0 through 0xFFFFD 0xFFFFE	COM	Comment
Reserved markers	0xFF01 0xFF02 through 0xFFBF	TEM* RES	For temporary private use in arithmetic coding Reserved

Table 4.2. JPEG markers[13]

4.2 An Overview of the JPEG Compression Algorithm

The JPEG algorithm [12,13] is a very complex process, it works on either full-color or gray-scale images; it does not handle so well bilevel (black and white) images; and it doesn't handle colormapped images either, those have to be to pre-expanded into an unmapped full-color representation. The algorithm works best on "continuous tone" images, unlike images with many sudden jumps in color values.

There are a lot of parameters to the JPEG compression process; by adjusting the parameters, is possible to trade off compressed image size against reconstructed image quality over a "very" wide range. Also is possible to get image quality ranging from op-art (at 100x smaller than the original 24-bit image) to quite indistinguishable from the source (at about 3x smaller). Usually the threshold of visible difference from the source image is somewhere around 10x to 20x smaller than the original (i.e., 1 to 2 bits per pixel for color images). Grayscale images do not compress as much. In fact, for comparable visual quality, a grayscale image needs perhaps 25% less space than a color image.

JPEG defines a "baseline" lossy algorithm, plus optional extensions for progressive and hierarchical coding. There is also a separate lossless compression mode; this typically gives about 2:1 compression (about 12 bits per color pixel). Most currently available JPEG hardware and software handles only the baseline mode.

The outline of the baseline compression algorithm is described as follows:

1) Transform the image into a suitable color space.

This isn't needed for grayscale, but for color images the usual procedure is about transform RGB into a luminance/chrominance color space such as YCbCr, YUV. The luminance component is grayscale and the other two axes are color information. This is done in order to afford to lose a lot more information in the chrominance components rather than in the luminance component, since the human eye is not as sensitive to high-frequency chromatic info as it is to high-frequency luminance.

This step isn't indispensable since the remainder of the algorithm works on each color component independently, and doesn't care just what the data is; however, compression will be less since all the components at luminance quality will be coded. As can be noticed, colorspace transformation is slightly lossy due to roundoff error, but the amount of error is much smaller than the one typically introduced in further steps.

2) Downsample each component by averaging together groups of pixels.

The luminance component is left at full resolution, while the chroma components are often reduced 2:1 horizontally and either 2:1 or 1:1 (no change) vertically. In the JPEG environment these alternatives are usually called 2h2v and 2h1v sampling. This step immediately reduces the data volume by one-half or one-third. In numerical terms it is highly lossy, but for most images it has almost no impact on perceived quality, because, as previously said, of the eye's poorer resolution for chroma info. As can be

noticed, downsampling is not applicable to grayscale data; this is one reason color images are more compressible than grayscale.

3) Group the pixel values for each component into 8x8 blocks.

Transform each 8x8 block through a discrete cosine transform (DCT). The DCT is a relative of the Fourier transform and likewise gives a frequency map, with 8x8 components. Now numbers representing the average value in each block and successively higher-frequency changes within the block are obtained; allowing to throw away high-frequency information without affecting low-frequency information (The DCT transform itself is reversible except for roundoff error).

4) In each block, divide each of the 64 frequency components by a separate "quantization coefficient", and round the results to integers.

This is the fundamental information-losing step, since more data gets discarded for larger quantization coefficients. Even the minimum possible quantization coefficient, 1, some information is lost, because the exact DCT outputs are typically not integers. Higher frequencies are always quantized less accurately (given larger coefficients) than lower, since they are less visible to the eye. Also, the luminance data is typically quantized more accurately than the chroma data, by using separate 64-element quantization tables. Tuning the quantization tables for best results is a very difficult procedure and is an active research area. Most existing encoders use simple linear scaling of the example tables given in the JPEG standard, using a single user-specified "quality" setting to determine the scaling multiplier. This works fairly well for midrange qualities but is quite non-optimal at very high or low quality settings.

5) Encode the reduced coefficients using either Huffman or arithmetic coding.

This step is lossless, so it doesn't affect image quality. The arithmetic coding option uses Q-coding, which is patented. Most existing implementations support only the Huffman mode, so as to avoid license fees. The arithmetic mode offers maybe 5 or 10% better compression, which isn't enough to justify paying fees.

6) Tack on appropriate headers and output the result.

In a normal "interchange" JPEG file, all of the compression parameters are included in the headers so that the decompressor can reverse the process. These parameters include the quantization tables and the Huffman coding tables. For specialized applications, the spec permits those tables to be omitted from the file; this saves several hundred bytes of overhead, but it means that the decompressor must know a-priori what tables the compressor used. Omitting the tables is safe only in closed systems.

The decompression algorithm reverses this process. The decompressor multiplies the reduced coefficients by the quantization table entries to produce approximate DCT coefficients. Since these are only approximate, the reconstructed pixel values are also approximate, but if the design has done what it is supposed to do, the errors won't be highly visible. A high-quality decompressor will typically add some smoothing steps to reduce pixel-to-pixel discontinuities.

The JPEG standard does not specify the exact behavior of compressors and decompressors.

4.3 The IJG JPEG Library

The IJG JPEG library [12] is a free software package developed by the Independent JPEG Group (IJG, not related to the JPEG committee that developed the standard); it provides C code to read and write JPEG-compressed image files. The surrounding application program receives or supplies image data a scanline at a time, using a straightforward uncompressed image format. All details of color conversion and other preprocessing/postprocessing can be handled by the library.

The library includes a substantial amount of code that is not covered by the JPEG standard but is necessary for typical applications of JPEG. These functions preprocess the image before JPEG compression or postprocess it after decompression. They include colorspace conversion, downsampling/upsampling, and color quantization. The application indirectly selects use of this code by specifying the format in which it wishes to supply or receive image data. For example, if colormapped output is requested, then the decompression library automatically invokes color quantization.

A wide range of quality vs. speed tradeoffs are possible in JPEG processing, and even more so in decompression post-processing. The decompression library provides multiple implementations that cover most of the useful tradeoffs, ranging from very-high-quality down to fast-preview operation. On the compression side low-quality choices are not provided, since compression is normally less time-critical. It should be understood that the low-quality modes may not meet the JPEG standard's accuracy requirements; nonetheless, they are useful for viewers.

It is handled a subset of the ISO JPEG standard; most baseline, extended-sequential, and progressive JPEG processes are supported.

4.4 Communication Channel Development

This section explains the procedure to build a communication channel between two processes, that will be used to communicate the board with external stages, all based on the Client-Server model.

4.4.1 The Client-Server model

It is a model for communication between two processes, where one of them, the “server”, waits for a connection from other process, the “client”, and after a communication channel has been established, data transfer can be done in any of the both senses, server to client or client to server [9].

Considering that the server is the process that waits to be contacted, it does not have to know the address of the client before the connection is established.

The connection building procedure is not the same for both processes, despite each of them build a “socket” for its respective side of the communication channel, where the data to receive and transmit is read and written respectively.

4.4.2 Socket Types

The communication between two processes can be achieved only if their sockets are of the same type and use the same address domain.

The most used address domains are the Unix domain, in which the processes which share a common file system communicate; and the Internet domain, in which the involved processes run on any two hosts on the internet.

In the Unix domain, the address of a socket is a character string related to the file system

In the Internet domain the socket address consists of the internet address of the host machine (a 32 bit address, often referred to as its IP address) and also a port number on that host, which are 16 bit unsigned integers.

The most used socket types are “stream sockets” and “datagram sockets”. Stream sockets use continuous streams of characters, while datagram sockets read the entire message at once.

Stream sockets use TCP (Transmission Control Protocol), while datagram sockets use UDP (Unix Datagram Protocol)

4.4.3 Socket implementation on the internet domain

The steps involved on the socket building and use procedure on the server side are indicated in the following flow chart:

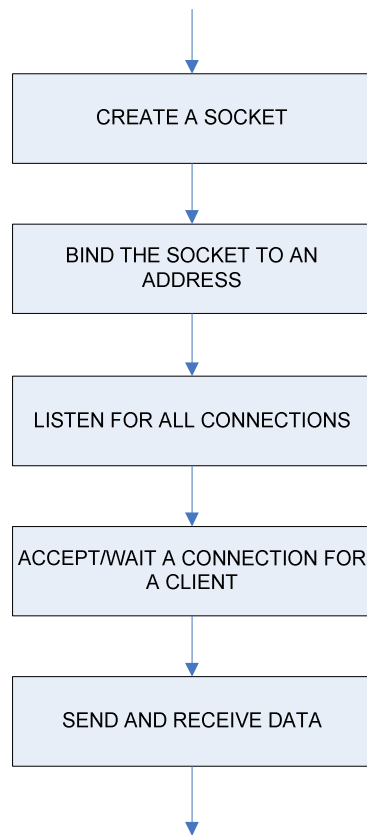


Figure 4.1. Server side socket building and usage

The steps indicated on the chart are explained as follows, with references to their respective implementation on C language:

1) Create a socket

This is done using the `socket()` system call, as indicated in the following code line:

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

The first argument is the address domain (the symbol constant `AF_INET` makes reference to the internet domain), the second argument is the type of

socket (SOCK_STREAM makes reference to an stream socket) and the third argument is the protocol, if zero (0) is used, the operating system will choose the most appropriate protocol (TCP for stream sockets, as said previously).

The socket() system call also returns an integer value which is an entry into the file descriptor table, this value is used for further references to the socket. If the socket() call falls, it returns -1; in the example, this value is assigned to the variable sockfd.

2) Bind the socket to an address

The socket is bound to an address using the bind() system call, as indicated in the following code lines:

```
if (bind(sockfd, (struct sockaddr *) &serv_addr,      sizeof(serv_addr)) < 0)
    error("ERROR on binding");
```

In the shown example, the address is referred to the current host and port number on which runs the server. It uses three arguments, the socket file descriptor, the address which it is bound, and the size of the address which it is bound. The second argument is a pointer to a structure of type sockaddr, but what was passed in is a structure of type sockaddr_in, which will be explained as follows.

A structure of the type struct sockaddr_in, has four fields, as can be seen in its definition:

```
struct sockaddr_in
{
    short  sin_family;
    u_short sin_port;
    struct in_addr sin_addr;
    char  sin_zero[8];
};
```


The first field (short `sin_family`) contains a code for the address family, so it must be set to the symbolic constant `AF_INET`; the second field (`u_short sin_port`) contains the port number (converted from “host byte order” to “network byte order” with the function `htons()`); the third field (`sin_addr`) is a structure of type `struct in_addr` which contains a single field unsigned long `s_addr`, that contains the IP address of the host machine, which is get using the symbolic constant `INADDR_ANY`; the fourth field must be zero and isn't used.

3) Listen for all connections

This is done using the `listen()` system call, as shown in the following code line:

```
listen(sockfd,5);
```

This system call allows the process to listen on the socket for connections. The first argument is the socket file descriptor, the second is the number of connections that can be waiting while the process is handling a particular connection, which should be 5, the maximum size permitted by most systems.

4) Accept a connection from a client

This is done using the `accept()` system call as indicated in the following code lines:

```
clilen = sizeof(cli_addr); //size of client address
newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
if (newsockfd < 0)
    error("ERROR on accept");
```

This system call causes the process to block until a client connects to the server; the process is awoken when a connection from a client is successfully established; its first argument is the socket file descriptor, the second argument is a reference to the pointer to the address of the client (also a structure of the type `struct sockaddr_in`, which has been explained beside the binding process) on the other end of the connection, and the third argument is the size of this structure; `accept()` also returns a new file descriptor, which shall be use for communication on the built connection.

5) Send and receive data

These operations can be done easily with the `write()` and `read()` system calls using as arguments the socket file descriptor, the buffer which contains the data to write or will contain the read data, and the size of that buffer, as shown in the following code lines:

Read socket (receive data):

```
n = read(newsockfd,buffer,255);
```

Write on socket (send data):

```
n = write(newsockfd,"I got your message",18);
```

The procedure for the socket building on the client side is described by the following flow chart:

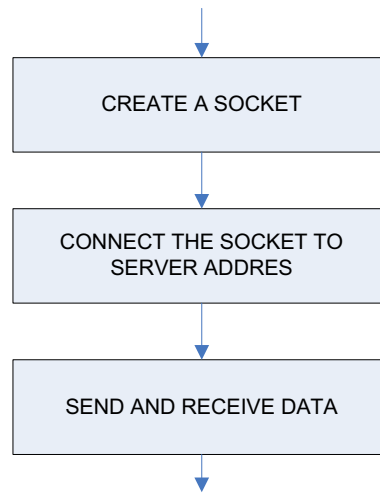


Figure 4.2. Client side socket building and usage

The steps of the procedure shown on the chart, with examples in C language, are explained as follows:

1) Create a socket

This operation is done with the `socket()` system call, exactly as previously explained for the server side.

2) Connect the socket to the address of the server

Before do the connection is important to read some information about the server, this is done using the function `gethostbyname()` which has as only argument the name of the host on the Internet domain, this function is defined as follows:

```
struct hostent *gethostbyname(char *name)
```

As can be seen, it returns a structure of type struct hostent, which is defined as follows:

```
struct hostent
{
    char  *h_name;      /* official name of host */
    char  **h_aliases; /* alias list */
    int   h_addrtype;  /* host address type */
    int   h_length;    /* length of address */
    char  **h_addr_list; /* list of addresses from name server */
    #define h_addr h_addr_list[0] /* address, for backward compatibility */
};
```

The indicated members of this structure are:

`h_name` Official name of the host.
`h_aliases` A zero terminated array of alternate names for the host.
`h_addrtype` The type of address being returned; currently always AF_INET.
`h_length` The length, in bytes, of the address.
`h_addr_list` A pointer to a list of network addresses for the named host. Host addresses are returned in network byte order.

Using `gethostbyname()` and storing its returned value in `struct hostent *server`, the fields of `serv_addr` are set as follows:

```
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
bcopy((char *)server->h_addr,
      (char *)&serv_addr.sin_addr.s_addr,
      server->h_length);
serv_addr.sin_port = htons(portno);
```

Considering that `serv_addr` is a structure of type `struct sockaddr_in`, which has been explained for the server side binding procedure.

Then, the connection is done using the `connect()` system call as follows:

```
if (connect(sockfd,&serv_addr,sizeof(serv_addr)) < 0)
    error("ERROR connecting");
```

If no server is described by `serv_addr`, the returned value is -1 and the process is interrupted.

3) Send and receive data

After the connection has been established, data can be received and sent using the `read()` and `write()` system calls, as shown for the server connection side.

4.5 Developed Programs Description

A system has been built based on the server-client model for interprocess communications over the Internet domain using the TCP protocol; involving two programs on the board and another two outside the board.

The first program on the board is in charge of, in stand alone operation, receive from the previous stage the file to be compressed, execute the second one to perform the compression, and send the resulting file to the following stage; the related stages outside the board are emulated by an external PC in an Unix-OS environment in which the other two programs run contemporaneously.

The mentioned programs are described as follows:

4.5.1 To run outside the Board

sendPPM

It works as a client that sends to the board the image to be compressed; its input parameters (manually placed by the user) are the server (board) IP address, the number of the port used by the server for the image reception and the name of the file which contains the image.

As can be seen in the flow chart at figure 4.3, the program operates by opening the file to send for its reading, then the transmission cycle begins building a socket and sequentially sending through it a part of the file on each loop (using a 2500 characters length buffer); a counter (named count) is used to quantify the already sent characters and is compared with the size of the file at the end of each loop and repeating the cycle again for the following part of the file; after “count” reaches the file size, the transmission ends, the success is notified to the user and the

used file is closed (the file size is computed with the data present on the file header, included in the buffer during the first loop after read the socket).

The program ends its execution after close the used file, or is terminated if an error occurs during the socket building or using as indicated in the correspondent blocks of the flow chart.

The command line on Unix corresponding to the program execution has the following structure:

```
$ ./sendPPM [server address] [port number] [PPM file name]
```

The source code of the program can be found in Appendix A.

recJPG

It works as a server that receives from the board the JPEG file obtained from the compression; its only input parameter is the number of the port to be used.

As shown in the flow chart in figure 4.4, the program loads an index value to assign for the name of the destination file for the next received JPEG image (assigns 0 if can't load value), builds the socket at the specified port number (only once) and then starts a cycle receiving, or waiting for, each part of the JPEG image through the socket sent by a client, storing the received parts at the destination file (a 256 characters buffer is used in this operation) the operation is repeated until the "end of image" sequence of characters is recognized inside the part of the file being processed or if a communication error occurs as indicated in the respective blocks of the chart.

After a JPEG image is completely received the destination file is closed, the index value stored and incremented, a new destination file with the new index value is opened and the program connect to the socket waiting for the next image.

The program ends its execution only if it is externally interrupted or, as said previously, a communication error occurs during an image reception.

The command line on Unix corresponding to the program execution has the following structure:

```
$ ./recJPG [port number]
```

The source code of the program can be found in Appendix A.

4.5.2 To run inside the Board

Ingrb

It is the program in charge of the management of the communications with the external stages and orders the execution of the program for the image compression (cjpeg3).

As indicated in the flow chart at figure 4.5, the communications management in the main process is done by calling two functions, one to receive the PPM files through the port 50900 (receiveP6), and another called after the compression is done to send the resulting JPEG file (out_img.jpg) through the port 50800 (sendJPG). At the main process a flag is also used in order to indicate to the function receiveP6 if a new socket must be built for the PPM files reception.

After the compressed file is sent to the next stage, the cycle begins again and the program calls `receiveP6`, which waits for the next PPM file to be compressed. The source code of the program and its functions can be found in Appendix A.

The functions used by the program are explained as follows:

receiveP6

It is the function that plays the server role for the reception of a file to be compressed; its input parameters are: port number, name of the file where the PPM image will be stored, and a flag that indicates if a new socket is needed.

As can be noticed at the flow chart on figure 4.6, the function opens a destination file with the specified name (if a file with the same name already exists, it is overwritten) builds a socket (if indicated by the respective flag) and then waits for a connection of an external client to the socket, after that happens it begins to (using a 2500 character buffer) receive sequentially the parts of the file sent by the connected client until the entire file is received (the size of the file is computed with the information on its header); then the destination file is closed and the program returns to the process that has called the function.

As shown in the respective blocks of the chart, the program is terminated by the function itself if a communication error occurs during the file reception process.

sendJPG

It is the function that acts as a client to send a JPEG image to a server according to its input parameters: server address, number of the port to be used and name of the file to be sent.

As shown in the flow chart in figure 4.7, its operation begins opening the file to be sent, then the sending process starts building and connecting to the socket, and writing on it the respective part of the file to be sent (using a 256 character buffer); the process is repeated until the “End of Image” sequence of characters is recognized inside the part of the file that is being sent.

After all the file is sent, the used file is closed and the programs returns to the process that has called the function.

As receiveP6, the function sendJPG is able to terminate the program if a communication error occurs during the sending process.

cjpeg3

It is the program that does the JPEG compression, its source file is part of the package of the libjpeg library[10], only minor modifications were made on it.

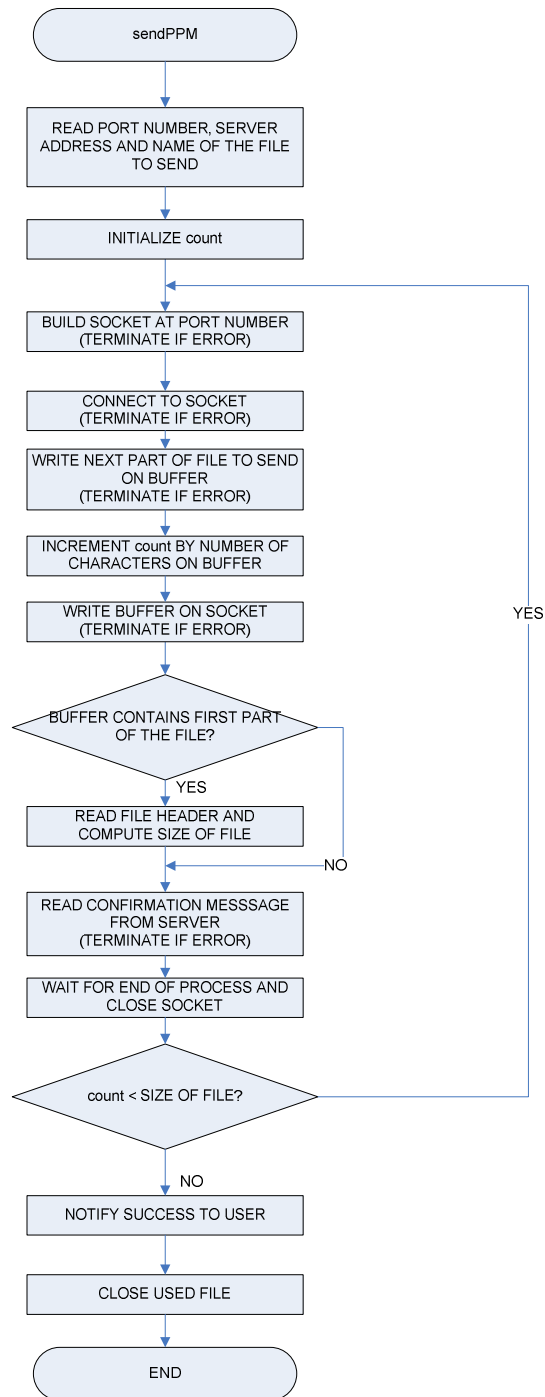


Figure 4.3. sendPPM program flow chart

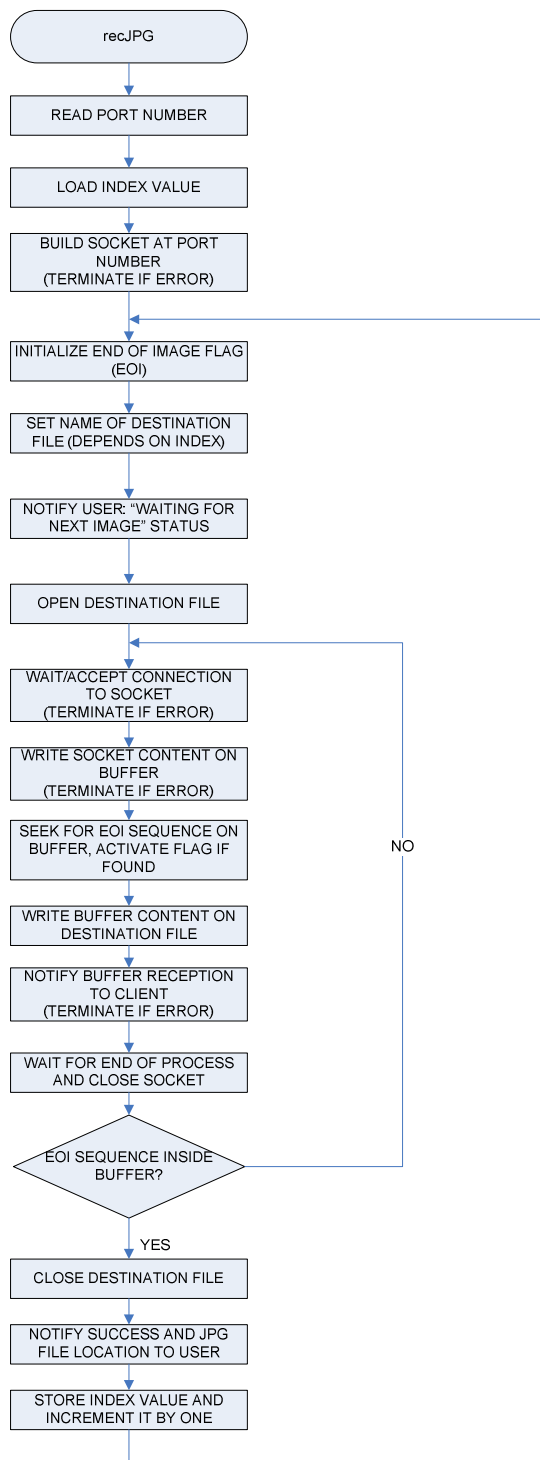


Figure 4.4. recJPG program flow chart

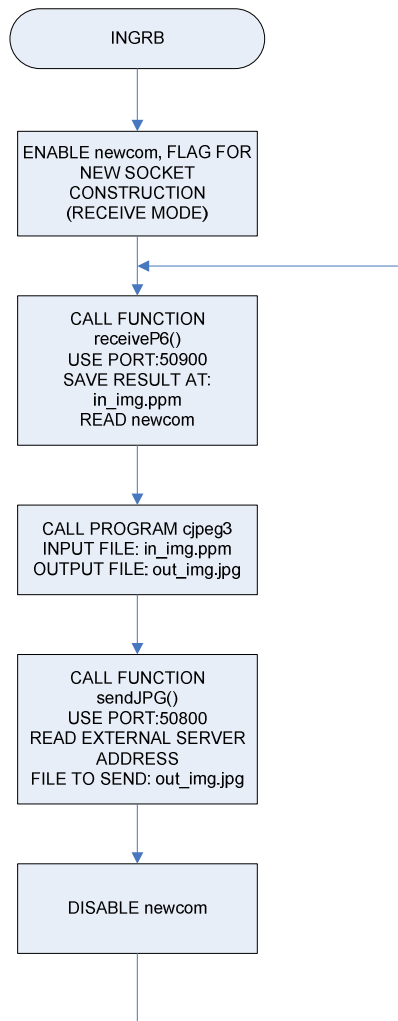


Figure 4.5. INGRB program flow chart

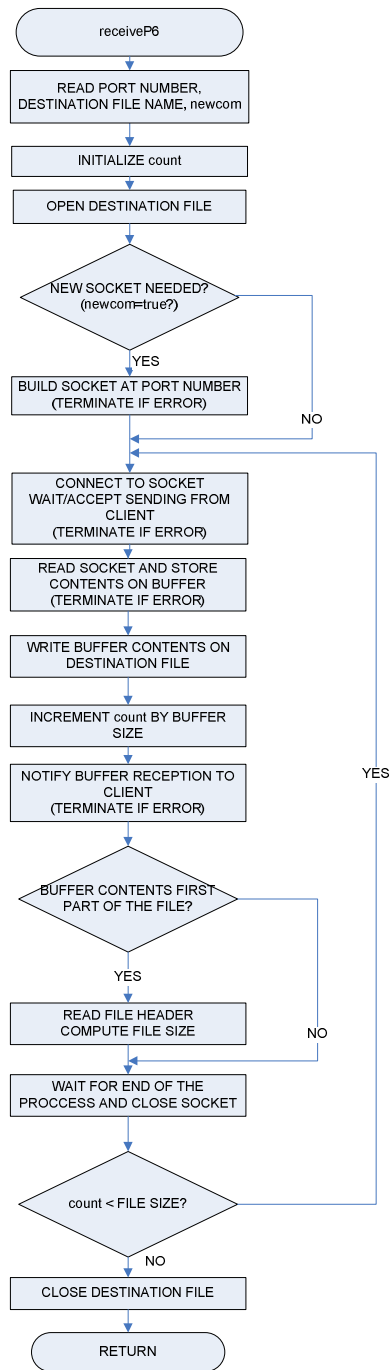


Figure 4.6. receiveP6 function flow chart

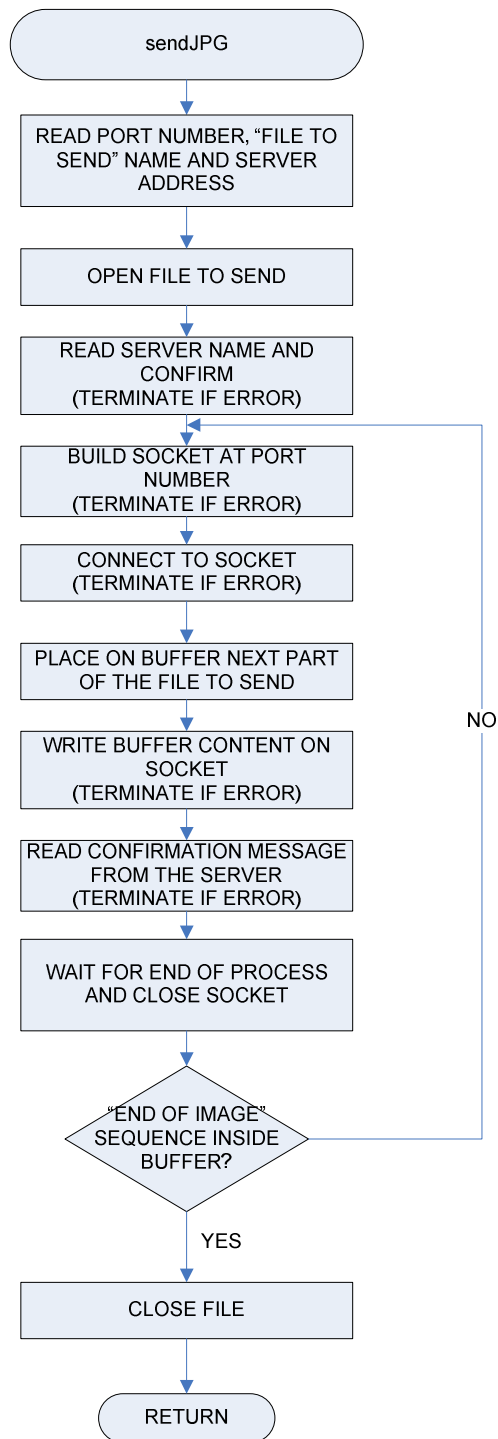


Figure 4.7. sendJPG function flow chart

4.6 Developed Programs Usage

The way in which the previously explained programs are used to test the board is explained in this section; as said before, the stages outside the board are emulated by running two programs on an external PC. The programs that run outside the board are compiled with gcc, and those on the board are compiled with sparc-linux-gcc.

The programs operation sequence is enumerated as follows, the execution order of the first two is irrelevant since the server processes wait for a client connection:

- 1) The user executes recJPG at the external PC indicating the port 50800 for the JPEG image reception, as indicated in the following command line:

```
$ ./recJPG 50800
```

- 2) INGRB runs on the board since it is turned on or reset, waiting for a file to be compressed (as indicated on the receiveP6 function) on the port 50900.

- 3) At the external PC the user executes sendPPM indicating the port 50900 and the name of the PPM file to be compressed, as indicated in the following command line:

```
$ ./sendPPM [board IP address] 50900 [.ppm file name]
```

- 4) After the PPM file is received by INGRB and saved at the board as “in_img.ppm”, cjpeg3 is executed and the resulting file is saved as “out_img.jpg”.

- 5) The compressed image is sent through the port 50800 to the external PC, where it is stored by recJPG as aramisXXXX.jpg, where XXXX symbolizes a

4 digit number related to the index value (as explained on the previous section).

- 6) INGRB and recJPG wait for the next PPM and JPEG files respectively, then the process is repeated until an error occurs or one of the programs is interrupted.

4.7 Tests and Results

The developed programs were used as described by the previous section; the environment for the performed tests can be seen in the figure 4.8 (the command lines and the messages from the programs were magnified in order to make them readable).

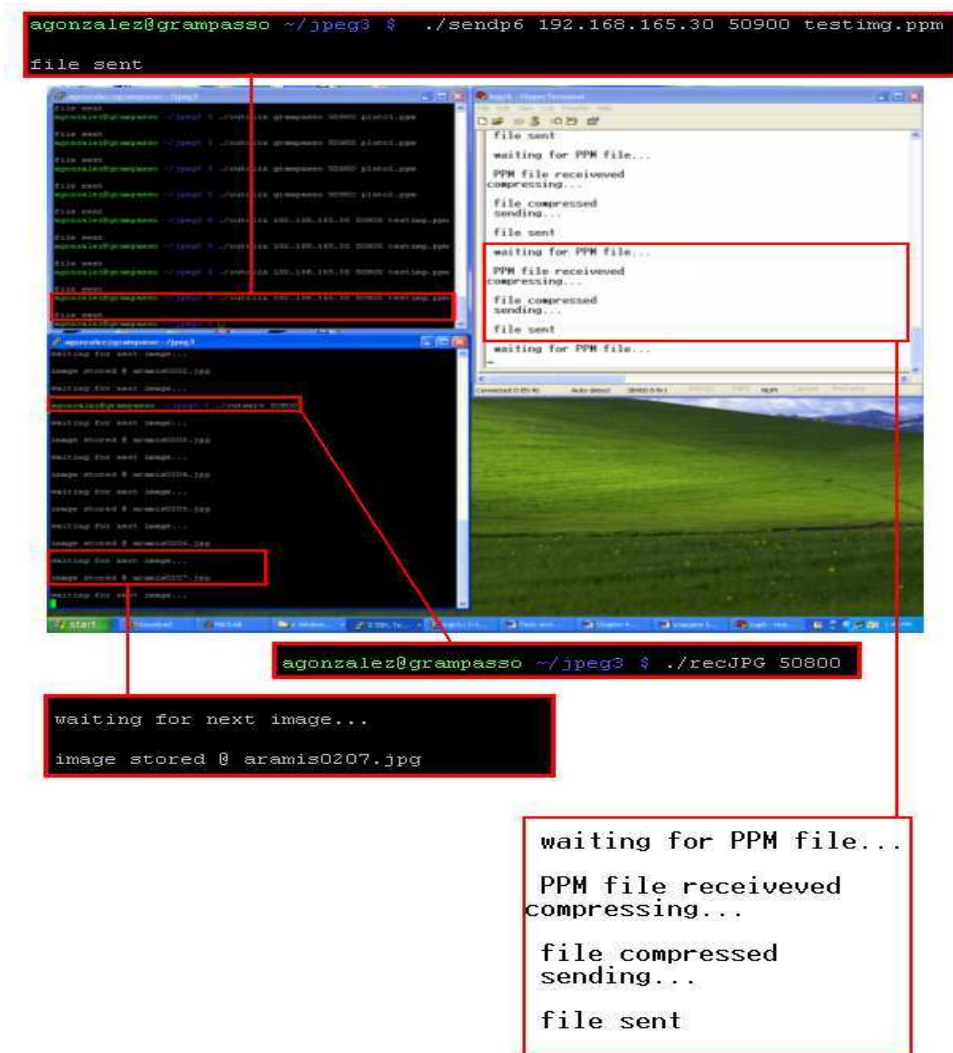


Figure 4.8. Programs test environment

In order to test the reliability of the programs, and the capacities of the board itself, four different PPM files were used; the times for sending the image (to the board), the compression and the receiving of the resulting file were measured, the

obtained results are shown on the table 4.3 (times under 0.50s were considered negligible).

File name	Original Size (KB)	Sending Time (s)	Compression Time (s)	Receiving Time (s)	Result Size (KB)
aqui4.ppm (color image)	533	1.7	3.4	<0.50	28.0
testing.ppm (color image)	99.1	<0.50	<0.50	<0.50	5.85
pippo.ppm (grayscale image)	1216	3.6	7.1	<0.50	14.9
pluto1.ppm (color image)	1216	3.6	7.1	<0.50	28.5

Table 4.3. Programs performance

It is important to notice that the sending and receiving speeds are related to many factors, some of them produced by the methods applied on the syntax of each program (such as the size of the used buffers to read and write on the respective sockets); others related to the hardware, like the processing times of the board (Leon3) and the external PC (Pentium D) and the chosen media for the communication (Ethernet).

The compression speed depends on the settings of the program for the compression (set as default), which are related to the parameters of the methods involved in the compression algorithm, also to the syntax of the program itself and, of course, the processing time of the board.

From the results can also be noticed, as expected, the influence of the type of the image to be compressed and the size of the obtained file, as can be seen with “pippo.ppm” and “pluto1.ppm” which have the same size; the smaller size obtained for the grayscale image after the compression is related to the redundant data of the original PPM size (which contain the information in scales of red, green and blue), which is eliminated during the compression itself.

Chapter 5

Conclusions

During this project an option for an image processor as an eventual part of the Payload of the ARAMIS satellite has been analyzed. Different results were obtained for each stage of the development process.

The LEON3 processor and GRLIB IP library VHDL model reliability was partially proved. The used LEON3 precompiled version allowed to evaluate the capacities of the processor and features of the GRLIB IP library with successful results. The needed parts for the design process following stages were present on the precompiled version.

The flexibility of the used VHDL model can be studied in future works using different synthesis tools (including a faster host) and/or studying the model itself, in order to find needed modifications for its usage. The reported results in this project should be considered before the eventual acquisition of a fault tolerant version of the LEON3 processor, which is able to be used on a satellite. It is also important to note that the manufacturer (Gaisler Research) does constant improvements and changes to the existent VHDL model.

In addition, the GR-XC3S-1500 development board was successfully used as an evaluation and prototyping tool for LEON3 processor and GRLIB IP library applications. Most of the performed procedures over the board are applicable to similar FPGA based-on devices specially designed to work under aerospace conditions.

The SnapGear package features were useful for the development of the operation system placed on the board. The performed changes over the provided kernel did not require difficult procedures. However, the usage of some of the included tools to modify the SnapGear Linux kernel could be quite complex in order to perform certain modifications to manage some devices on the board.

The development of programs to build communication links taking advantage of the Ethernet port was relatively easy. The obtained results of the performed data transmission tests based on the Client-Server model were successful. Similar procedures could be applied over similar communication protocols that are often used for data transmission on satellites, such as spacewire.

The JPEG library developed by IJG was a really useful tool on the design. The library was easily installed over an Unix host. Some errors appeared on the first times that the library was used, however, them were easily solved using basic knowledge of the C language. The provided documentation and examples were very helpful. The integration of the image compression and communications management programs was easily completed taking advantage of the library usage simplicity.

Future projects should complete the satellite image processing system by selecting an appropriate image capture device and developing the correspondent data transmission to the image compression stage. A webcam could be used taking advantage of the USB management tools provided by the GRLIB IP library and SnapGear, which could be used on a similar development board designed for aerospace conditions, as previously indicated. Other pending issues are the develop a data storage and/or sending stages for the compressed images and the selection of an appropriate data transmission protocol. A discussion topic could be if store the compressed images in the satellite in case of a communication failure with the ground segment. Another options such as an DSP device can be also considered for the image

compression stage itself, analyzing the reported advantages and disadvantages of using a based on FPGA device in this project.

Bibliography

1. Twiggs B., Puig-Suari J. CUBESAT Design Specification Document, Stanford University and Polytechnical Institute. www.cubesat.org
2. Reyneri L., “PICPOT Satellite Universitario del Politecnico di Torino, documento di specifiche di sottosistemi elettronici”, 6th edition
3. S. Spereta, L. Reynery, C. Sansoè, M. Tranchero, C. Passerone and Dante Del Corso, “Modular Architecture for satellites” 58th International Astronautical Congress, September 2007
4. LEON3 Product sheet
5. GR-XC3S-1500 Development Board User’s Manual
6. GRLIB IP User’s Manual
7. Xilinx ISE9 Foundation Tutorial
8. SnapGear Linux Manual
9. Linux Howto Socket Construction
http://www.linuxhowtos.org/C_C++/socket.htm
10. NETBPM Homepage
<http://netpbm.sourceforge.net/>
11. Kernighan & Ritchie - The C Programming Language

12. IJG Homepage

<http://www.ijg.org/>

13. JPEG standard

<http://www.w3.org/Graphics/JPEG/itu-t81.pdf>

Appendixes

Appendix A

Developed Programs Source Codes

Appendix A1. File INGRB.c

```
/* File: INGRB.C
Program: INGRB
Developed by:
Alejandro Gabbriel Gonzalez Esculpi
Tesi di Laurea Magistrale - Politecnico di Torino
Program to be executed to process images
A PPM File is received by port 50900 (acts as a server)
A program to compress the file is called
The obtained JPEG file is sent through (acts as a client)
the port 50800 (if server found)
*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

//Global Variables
struct sockaddr_in serv_addr;
int sockfd,clilen;
int newcom;

void error(char *msg)
{
    perror(msg);
    exit(1);
}

void receiveP6(int portno, char *fn, int newcon);
void sendJPG(int portno, char *servername, char *fn);

//main process
int main(void)
{

//enable flag for new socket construction
newcom=1;
```

```

//loop start
while (1)
{
    //call the function to receive the .PPM file
    //through the port 50900
    //and store it at "in_img.ppm"
    //if indicated by "newcom", create a new socket
    receiveP6(50900, "in_img.ppm",newcom);

    //call the program to do the jpeg compression
    //and save the result at out_img.jpg
    (void)system("./cjpeg3 -outfile out_img.jpg in_img.ppm");

    //send the compressed image to the server
    //(identified by its IP address)
    //through the port 50800
    sendJPG(50800,"130.192.165.79","out_img.jpg");

    //disable newcom (a new socket isn't needed
    //for the reception of the next .PPM file)
    newcom=0;
}
}

//Function for the reception of a .ppm file
void receiveP6(int portno, char *fn, int newcom)
{
    int newsockfd;
    char buffer[2500], filetype[2];
    struct sockaddr_in cli_addr;
    FILE *f;
    int n, i, r, ini=1, cols, rows, maxcolor, count=0, maxcount=50000;

    //open file to write on
    f = fopen(fn,"wb");

    //if indicated by newcom, build a socket
    if (newcom)
    {
        sockfd = socket(AF_INET, SOCK_STREAM, 0);

        if (sockfd < 0)
            error("ERROR opening socket");
        bzero((char *) &serv_addr, sizeof(serv_addr));
        serv_addr.sin_family = AF_INET;
        serv_addr.sin_addr.s_addr = INADDR_ANY;
        serv_addr.sin_port = htons(portno);

        if (bind(sockfd, (struct sockaddr *) &serv_addr,
        sizeof(serv_addr)) < 0)

```

```

        error("ERROR on binding");
    }

    //Reception sequennce
do
{
    listen(sockfd,5);
    clilen = sizeof(cli_addr);

    //connect to socket
    newsockfd = accept(sockfd,
        (struct sockaddr *) &cli_addr,
        &clilen);
    if (newsockfd < 0)
        error("ERROR on accept");
    bzero(buffer,2500);

    //read socket and store its content on buffer
    n = read(newsockfd,buffer,2499);
    if (n < 0)
        error("ERROR reading from socket");

    //write buffer content on the destination file
    fwrite(buffer,1,2499,f);

    //increment character count by buffer's size
    count+=sizeof(buffer);

    //notify buffer reception to the client
    n = write(newsockfd,"I got your message",18);
    if (n < 0)
        error("ERROR writing to socket");

    //read file header and define the size of the file to be
generated
    if (ini)
    {
        ini=0;
        sscanf(buffer,"%s                %d                %d"
            %d",filetype,&cols,&rows,&maxcolor);
        maxcount=cols*rows*3;
    }

    //wait for the end of the process and close socket
    wait(0);
    close(newsockfd);

    //repeat loop until all the file is received
    //or an error occurs
}
while ((n>0)&&(count<maxcount));

//close file

```

```

fclose(f);
}

void sendJPG(int portno, char *servername, char *fn)
{
int n, sockfd1;
struct sockaddr_in serv_addr1;
struct hostent *server;
char buffer[256], buf[256];
FILE *f;
int i, band=1, EOI=1,r;

//open file to be sent
f = fopen(fn,"rb");
r=0;

//read server name and confirm
server = gethostbyname(servername);
if (server == NULL)
{
    fprintf(stderr,"ERROR, no such host\n");
    exit(0);
}

//start loop
do
{
    //build socket
    sockfd1 = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd1 < 0)
        error("ERROR opening socket");
    bzero((char *) &serv_addr1, sizeof(serv_addr1));
    serv_addr1.sin_family = AF_INET;
    bcopy((char *)server->h_addr,
    (char *)&serv_addr1.sin_addr.s_addr,
    server->h_length);
    serv_addr1.sin_port = htons(portno);

    //conect to socket
    if (connect(sockfd1,&serv_addr1,sizeof(serv_addr1)) < 0)
        error("ERROR connecting");

    //place part to be sent of the file on buffer
    bzero(buf,256);
    n= fread(buf,255,1,f);
    if (n < 0)
        error("ERROR reading file");

    //seek for "end of image" sequence
    //of characters in the buffer (0xffff 0xffd9)
    //EOI is changed to false if the sequence
    //is found
    for(i=0;((i<256)&&(EOI));i++)

```

```

    {
        r=(unsigned short)buf[i];
        if ((r==0xffd9)&&(band))
            EOI=0;
        if (r==0xffff)
            band=1;
        else
            band=0;
    }

    //write buffer content on socket
    n = write(sockfd1,buf,255);
    if (n < 0)
        error("ERROR writing to socket");

    //read confirmation message from the server
    bzero(buffer,256);
    n = read(sockfd1,buffer,255);
    if (n < 0)
        error("ERROR reading from socket");

    //wait for end of process and close socket
    wait(0);
    close(sockfd1);

    //repeat loop until an error occurs or End Of Image (EOI false)
    //sequence is detected
    }
    while((n>0)&&(EOI));

    //close file
    fclose(f);
}

```

Appendix A2. File sendPPM.c

```
/* File: sendPPM.c
Program: sendPPM
Developed by:
Alejandro Gabbriel Gonzalez Esculpi
Tesi di Laurea Magistrale - Politecnico di Torino
Program for a client in the internet domain
using TCP that sends
a .ppm file to a server (indicated by its
IP address) through a selected port*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
void error(char *msg)
{
    perror(msg);
    exit(0);
}

int main(int argc, char *argv[])
{
    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;
    char buffer[2500], filename[20], *fn, *testo;
    FILE *f;
    char buf[2500], r, filetype[2];
    int i, ini=1, cols, rows, maxcolor, count=0,
    maxcount=50000, headsize, eolcount=0;
    //the file's size in characters is set by default in 50000
    if (argc < 4) {
        fprintf(stderr, "usage %s hostname port file\n", argv[0]);
        exit(0);
    }

    //read the name of the file to be sent
    fn=argv[3];
    //read port number
    portno = atoi(argv[2]);
    //read server's name and confirm
    server = gethostbyname(argv[1]);
    if (server == NULL)
    {
        fprintf(stderr, "ERROR, no such host\n");
        exit(0);
    }
    //open (to read) file to be sent
    f = fopen(fn, "rb");
    //start loop
    do
    {
        //build socket
        bzero((char *) &serv_addr, sizeof(serv_addr));
        serv_addr.sin_family = AF_INET;
```

```

bcopy((char *)server->h_addr,
(char *)&serv_addr.sin_addr.s_addr,
server->h_length);
serv_addr.sin_port = htons(portno);
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
    error("ERROR opening socket");
//connect to socket
if (connect(sockfd,&serv_addr,sizeof(serv_addr)) < 0)
    error("ERROR connecting");
//clean buffer
bzero(buf,2500);
//write on buffer the part of the
//file to be sent, until fill the buffer or
//at arrive at the number of characters of the file
for(i=0;((i<2499)&&(count<maxcount));i++)
{
    //read character and place it on buffer
    buf[i]=fgetc(f);
    r=buf[i];
    //increment character counter
    count++;
    //recognize end of line character (10)
    //(the file's header should be placed in the
    //first 3 lines)
    if ((r==10)&&(eolcount<3)&&(ini))
    {
        eolcount++;
        headsize=count;
    }
}
//write buffer on socket
n = write(sockfd,buf,i);
//if the buffer contains the first part of the file,
//read the files specs from the header
if (ini)
{
    //operation to be done only for the first part of the file
    ini=0;
    //read image parameters (columns, rows and maximum color number)
    r = sscanf(buf,"%s %d %d
%d",filetype,&cols,&rows,&maxcolor);
    //compute file's size (each 3 bytes define a pixel and
    //the header's size must be included)
    maxcount=cols*rows*3+headsize;
}
if (n < 0)
error("ERROR writing to socket");
bzero(buffer,2500);
//read confirmation message from the server
n = read(sockfd,buffer,2499);
if (n < 0)
    error("ERROR reading from socket");
//wait for end of the process and close socket
wait(0);
close(sockfd);
//repeat again until all the file is sent
}
while((count<maxcount)&&(n>0));
//close file
fclose(f);

```

```
    //notify success
    if(n>0)
    printf("\nfile sent \n");
    return 0;
}
```


Appendix A3. File recJPG.c

```
/* File: recJPG.c
Program: recJPG
Developed by:
Alejandro Gabbriel Gonzalez Esculpi
Tesi di Laurea Magistrale - Politecnico di Torino
Server in the internet domain using TCP
the port number is passed as an argument
the program notifies the client after each
part of the file is received and asks it for
the next part, until the recognition of the
end of image sequece corresponding to a
JPEG file*/

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void error(char *msg)
{
    perror(msg);
    exit(1);
}

int main(int argc, char *argv[])
{
    int sockfd, newsockfd, portno, clilen;
    char buffer[256], filename[50], *fn, idch[5];
    struct sockaddr_in serv_addr, cli_addr;
    int n,i,r,idx=0,count;
    FILE *f,*f1;
    int buf, band=0, EOI=1, newcon=1;

    //open file where index value is stored
    //and assign value to index.
    //if it does not exist, create it
    //and assign 0 as index value.
    if (f1=fopen("index.txt","r"))
    {
        fscanf(f1,"%i",&idx);
        fclose(f1);
        idx++;
    }
    else
        idx=0;

    //Build socket (only once)
    if (argc < 2)
    {
        fprintf(stderr,"ERROR, no port provided\n");
        exit(1);
    }
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");
    bzero((char *) &serv_addr, sizeof(serv_addr));
    portno = atoi(argv[1]);
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(portno);
    if (bind(sockfd, (struct sockaddr *) &serv_addr,
```

```

sizeof(serv_addr) < 0)
    error("ERROR on binding");
newcon=0;

//start loop
while (1)
{
    //initiate EOI (end of image) flag
    //Its value is non zero until the
    //end on image sequence is found on the file
    EOI=1;

    //set name of the file to store the received image
    //according to the index value, if it is equal to
    //10000, restart from 0
    if (idx<10)
        sprintf(filename,"aramis000%i.jpg",idx);
    else if (idx<100)
        sprintf(filename,"aramis00%i.jpg",idx);
    else if (idx<1000)
        sprintf(filename,"aramis0%i.jpg",idx);
    else if (idx<10000)
        sprintf(filename,"%i",idx);
    else
    {
        idx=0;
        printf("\nWARNING: MAXIMUM INDEX REACHED, OVERWRITING");
    }

    fn=filename;

    //notify "waiting for next image" status
    printf("\nwaiting for next image...\n");

    //open file to write image on
    f=fopen(fn,"wb");

    //initialize character counter
    count=0;

    //Receivng sequennce
    //start loop
    do
    {
        listen(sockfd,5);
        clilen = sizeof(cli_addr);

        //accept connection to socket
        newsockfd = accept(sockfd,
            (struct sockaddr *) &cli_addr,
            &clilen);
        if (newsockfd < 0)
            error("ERROR on accept");

        bzero(buffer,256);

        //read socket content and assign it to buffer
        n = read(newsockfd,buffer,255);
        if (n < 0)
            error("ERROR reading from socket");

        //seek for end of image sequence (ffff ffd9)
        //inside the buffer
        for(i=0;((i<256)&&(EOI));i++)
        {
            r=(unsigned short)buffer[i];
            if ((r==0xffd9)&&(band))

```

```

        EOI=0;
        if (r==0xffff)
            band=1;
        else
            band=0;
    }

    //write buffer content on destination file
    n=fwrite(buffer,i-1,1,f);
    if (n < 0)
        error("ERROR writing to file");

    //notify success to client
    n = write(newsockfd,"I got your message",18);
    if (n < 0)
        error("ERROR writing to socket");

    //wait for end of process and close socket
    wait(0);
    close(newsockfd);

    //increment character counter by buffer size
    count+=256;

    //notify error if jpeg file is "too big"
    if (count>9e6)
        error("ERROR incorrect file format or it is too big");

    //repeat loop until an error occurs or end of image
    //sequence is detected
    }
    while ((n>0)&&(EOI));

    //close file with the received image
    fclose(f);

    //notify success and image location
    //to user
    printf("\nimage stored @ %s\n",filename);

    //store index value and increment
    fl=fopen("index.txt","w");
    fprintf(fl,"%i\n",idx);
    fclose(fl);
    idx++;

    //wait for next image (.jpg file), repeat loop
    }
    return 0;
}

```