Intelligent Medical Device Integration with Real Time Operating System

by

© Zaid Jan

A thesis submitted to the
School of Engineering
in partial fulfilment of the
requirements for the degree of
Master of Engineering

Department of Electronics and Computer Syetem Engineering
at Massey University, [Albany],
New Zealand

April 2009

Abstract

Many commercial devices now being produced have the ability to be remotely monitored and controlled. This thesis aims to develop a generic platform that can easily be extended to interface with many different kinds of devices for remote monitoring and control via a TCP/IP connection. The deployment will be concentrated on Medical devices but can be extended to all serial device interfaces.

The hardware to be used in the development of this platform is an ARM Cortex M3 based Micro-Controller board which has to be designed to meet the requirement set by the Precept Health the founder of this platform. The design was conducted at Massey University in collaboration with senior engineer from the company.

The main task in achieving the aim was the development of the necessary software layers to implement remote monitoring and control. The eCosCentric real-time embedded operating system was used to form a generic base for developing applications to monitor and control specific devices. The majority of the work involved in this project was the deployment of the operating system to the Micro-Controller.

During the development process, several hardware issues were discovered with the Ethernet interface and were corrected. Using the generic platform, an application was developed to allow the reading of Bi-Directional pass through a communication protocol from 4 isolated serial input channels, to an Ethernet channel using TCP protocol.

Acknowledgments

The success of this project would not have been possible without the guidance, assistance and dedication of a number of people. I would like to give many thanks to my supervisor, Dr Tom Moir for his advice, feedback and guidance and also for giving me the opportunity to conduct this thesis.

I would like to thank Tony Blomfield for offering the opportunity and funding to develop this platform, and for his on-going support throughout the project. His knowledge and experience has helped me to avoid many pitfalls along the way

Furthermore I would like to thank Nestor and Philip from eWatch for all their help and guidance through-out the project. Special thanks to Philip for explaining some off the most complex parts of the real time operating system and avoid some of the pitfalls which I have encountered during the development cycle

Finally, I would like to thank my family for their ongoing support, and I am certain they are happier than me that it's all over!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Thesis Overview

eCosCentric is an open source real-time operating system developed by Redhat and its user community. It's like other conventional operating systems, seeks to reduce the burden of application development by providing convenient abstractions of physical devices and highly tuned implementations of common functions [15]. eCosCentric has been designed in such a way that a small resource footprint can be constructed. It is extremely configurable and allows developers to select components that satisfy basic application requirements. eCosCentric uses compile-time control methods, along with selective linking, provided by the GNU linker, to give the developer control of its behaviour, allowing the implementation itself to be built for the specific application for which it is intended.

The motivation for this master thesis arose from the need to develop a cheap device with a network interface and software support for the TCP/IP protocol. Since the computing power and memory resources were below the limit to run Embedded Linux, the natural decision is to run the much more economical real-time operating system eCosCentric. This document gives advice for further developments related to the build of the mentioned cheap network device and describes the conducted works related with the porting of eCosCentric to a custom hardware platform. During the conduction of this project several major aims were strived. The following list gives a summarised overview about the aims and objectives.

- Custom hardware design
- Evaluation of the eCosCentric operating system regarding architecture, capabilities, and footprint.

- Setup of the software development system
- Porting of eCosCentric to the target hardware
- Implementation of required drivers
- Integration of the HAL port and new drivers into the eCosCentric component framework
- Application development

The initial step of the project was the reading of relevant literature in the space of embedded system. All required software components for host and target development has been obtained from the eCosCentric distributor's Internet resources. The host PC, was provided by Dr Tom Moir, the project supervisor. The basic hardware for this project was an Ubuntu Sarge operating PC. The interface between the host and the target board was established using the JTAGKEY-Tiny from Amontec [17]. The required software connection between host and target was established using OpenOCD [33] which is a free JTAG software debugger for the ARM Coretex, ARM7 and ARM9 CPUs. Several guides, all referenced in the bibliography, were considered to port the system to the custom hardware. Furthermore the eCosCentric mailing list gave supporting answers to upcoming questions.

The First major task of the thesis was to design the hardware that will meet the requirement set by the sponsor (Precept Health); the 2nd major task was the porting of the eCosCentric HAL (Hardware Abstraction Layer) to the custom hardware. This goal has been accomplished successfully. The three aimed interface drivers the serial [RS232], Ethernet and Micro SD card driver have been implemented and tested successfully. While the port of the Ethernet driver for the eCosCentric operating system was being implemented several hardware issues were discovered with the Ethernet controller and were corrected. Ethernet driver need some further work to be entirely integrated into the eCosCentric system. Thereby the developed routines of the Ethernet driver have to be debugged and tested to provide the demanded functionality. Nevertheless all drivers can be selected and configured using the eCosCentric Configuration Tool. The conducted HAL port and the drivers have been entirely adapted to the eCosCentric component framework.

The implemented applications RedBoot example illustrates the functionality of the eCosCentric HAL, the serial driver and the usage of the GPIO ports of the STM3 MCU.

1.2 Design Specification

The design of this system aims to integrate medical devices such as ECG, ventilator, drugs pumps, etc..., with real-time reconfigurable operating system capabilities. To fulfil future requirements of a medical devices integration platform, commonly used features from existing providers such as Philips, EMG, should also be incorporated for real-time data analysis. The performance throughout the entire system should to be maximized in order to satisfy the needs of future developments.

1.2.1 Requirements

The design of a new platform that will support multi serial interface that will integrate medical devices should ideally consist of the following:

- Design new hardware platform that support medical device integration with medical standards [certification]
- Evaluation of the eCosCentric operating system, regarding architecture, capabilities, and footprint
- Setup of the hardware and software development system
- Porting of eCosCentric to the target hardware
- Implementation of required drivers (Ethernet, Micro SD card, 4 Serial ports)
- Integration of the HAL port and new drivers in the eCosCentric component framework
- Application development
- Within the scope of this master thesis only free, open-source software packages have been used in this project except for the electronics design package (Altium).

1.2.1.1 Understanding of eCosCentric

A detailed comprehension of the eCosCentric operating system is a fundamental requirement for the process of the whole implementation. The architecture of eCosCentric has to be analysed and the crucial components for the single steps of the eCosCentric implementation have to be mentioned. One of the core components of the eCosCentric operating system is the Hardware Abstraction Layer (HAL). This layer builds together with the driver components the only hardware dependent part of eCosCentric. Upper layers can use the

standardised interface of the HAL to access hardware functionality. Hence the attributes of this module have to be described properly.

Interrupt and exception handling are important matters concerning the analysis of an operating system. These functionalities have to be investigated and reflected. Since the final aim of a port is the implementation of own applications the insight process of own tasks has to be described. Applications running on eCosCentric are built in threads, the handling of this software component has to be analysed

The core component of each operating system is the Kernel. This module deals with timing matters and offers scheduling mechanisms for thread administration. The attributes and configuration options of the eCosCentric scheduler have to listed and explained.

Furthermore all necessary attributes of eCosCentric concerning the port and the implementation have to be given.

1.2.1.2 Development Setup

The bases of each development setup are the software packages, the host and target hardware components, and the interface components between host and target. Primarily all used components have to be mentioned and in addition nontrivial installation and configuration steps have to be explained.

Hardware

The target hardware needs to provide a sufficient amount of memory. It has to be investigated which footprints are typical for eCosCentric.

Since an ARM Cortex M3 [STM32F103RE] Micro-controller will be used in future developments this type of processor has to be kept in mind. It is essential that the platform offers the general interfaces as SPI, CAN, Serial, and I2C. For debugging purposes a JTAG interface has to be provided.

Debugging has to be conducted using a hardware debugger. A suggestion is the Amontec JTAGkey-Tiny which is a programmable JTAG interface able to emulate specific USB to JTAG interfaces.

For the ongoing work the target platform design and layout has to be analysed and determining attributes for the implementation progress and the driver development have to be discussed.

Software

As already mentioned eCosCentric is a free, open source operating system. The open source thought has to be kept from the beginning of the master thesis till the final report. This means that all software and tools which are used have to be free. The host computer for development has to be a PC running Ubuntu Linux. Concrete installation steps and configuration settings of the host operating system have not been described since the host hardware is likely to be individual for each future developer. However the used software packages have to be listed together with their current versions. The setup of the development and debug environment for the eCosCentric port, the driver development and the design of applications has to described in detail but within the scope of thesis. If the installation of certain software excesses the scope of the thesis further documentation has to be offered.

Implementation of eCosCentric

The first major requirement is the porting of the eCosCentric HAL to the target hardware. Therefore the porting progress has to be analysed and a decision about the variant of the port has to be made. The HAL must initialize the target hardware containing CPU, memory, serial interface and GPIO setup. Furthermore the implementation has to be fitted into the eCosCentric component framework. This makes the new board selectable and configurable using the eCosCentric Configuration Tool.

Drivers for the Ethernet and serial communication, the external storage access have to be implemented. In particular the RS232 driver has to allow access to a bus sending and receiving messages. Full handshaking abilities, i.e. hardware filtering of messages, are not required.

As a first ROM application Redboot has to be built and run on the target hardware. The GPIO abilities have to be demonstrated using a RAM application using threads.

1.2.2 Constraints

Time and budgetary constraints limit the range of devices that can be incorporated into the hardware platform. Thus the primary goal is to obtain the best possible performance within the given constraints and at minimal cost. These constraints are:

Power Supply: As the device will be safety critical so proper switch mode power supply have to be designed according to the medical standard [it must provide total isolation to all devices], power consumption should be minimized which mean devices that are not in use dose not consume power.

PCB size: As the target device is intended for a stand still platform such as a bed-side device, the size of the board must be kept to a minimum to reduce the cost of PCB board manufacturing. The current dimensions of the board is 100mm x 70mm. A much larger board would become too costly to produce.

PCB Layers: Many PCB manufacturers will manufacture PCB's with up to six layers. Locating a company willing to do small run of prototype is quite difficult and costly in particularly in New Zealand, PCBCART prototype which is located in China are the perfect solution, due to their price and quality of work. The design consists of two layer.

Medical certification: The RoHS (Restriction on Hazardous Substances) directive was enforced in the EU (European Union) on July 1st 2006 [34], with other countries around the world expected to follow suit in the following years. Amongst other things, this directive prohibits the production or import of electronics goods containing lead within the EU. This has caused a variety of production issues in the global electronics industry as all manufacturers require retooling to accommodate lead-free processes.

1.3 Major Contributions

The major contribution of this project are:

- The architectural and schematic design of the Medical Device Adapter;
- Sourcing RoHS-compliant parts for the design;
- The eCosCentric driver support;
    1. ARM Cortex M3 thumb 2 STM32 configuration
    2. STM32 Memory-Mape I/O access
    3. Wiznet Ethernet Driver
    4. RS232 device driver

5. Micro SD card driver [external storage]

- performance and optimization of data transfer from the Ethernet to CPU

1.4 Thesis Outline

Chapter 1 Offers a brief outline of the project, its motivations and its specification.

Chapter 2 Presents background information in the area of embedded systems and also covers the hardware, software requirement of medical device development.

Chapter 3 Documents the approach taken to the hardware design of the Medical Device Adapter.

Chapter 4 Describes the software drivers and interfaces written for the board.

Chapter 5 Describe the approach taken to setting up the software environment.

Chapter 6 Summarizes the project and discusses future work on the system that will follow.

Chapter 2

Background and Literature Review

2.1 Embedded System Overview

Today embedded systems control lots of different tasks implementing more powerful and complex device. They are integrated and plugged in many kinds of machines, but also taking an stand alone application as they have been implemented in devices like PDAs, mobile phones and MP3 players. One major sector of embedded devices from the beginning has been the devices in industry. With an external controller an industry machine with a proper interface can easily be connected to local area network or to internet without making any modifications to the system. The embedded controller acts as a modem like device to the end system. The collected information can then be displayed via a PC in the main controller room and the necessary settings to the end machine can be done from there. More advanced embedded devices can also include analogue or digital inputs and outputs for controlling and process the collected information before it is forwarded on.

The embedded systems today can roughly be divided into high-end embedded systems and deeply embedded systems.

- The high-end embedded system classification is used when a general purpose OS is stripped down and left with specific modules for specific purpose. As the different parts of embedded solutions are becoming less expensive, more and more devices classified to this category. Examples of these devices are router, personal digital assistant; PDA and todays mobile phones.

- Deeply embedded systems are designed for particular application. They need to be very compact, integrated with only few basic functions. These device are designed with a minimal operating systems and hardware layout designed for the specific purpose.

Examples of these devices are small controllers and devices in our everyday life like microwave, where they are embedded in [5].

2.2 Embedded Computing Design

As components like processor and memory has become less expensive are embedded device integrated in several different kinds of device. This also means that we are able to use more processing power and memory on these devices, so they are becoming much more complex, providing more functionality. The different resources running on the device requires to be controlled by more intelligent operating system. In many simple 8-bit controllers, the tasks can be handled in a simple server loop, just by polling the different interfaces on the device. More intelligent device requires more intelligent operating system to manage all the functions that they provide. The operating systems for these devices are many times derivated from a general purpose operating system, so the barrier between embedded and general purpose operating system can be inconstant.

The concept of operating system is not that clear. Usually the operating is supposed to fulfil the tasks of providing an extension to the hardware. This means that it provides layering by providing the lower level drivers for the physical chips and an API for the programmer. Operating system also provides a resource management for different applications running on that device by handling interrupts and allocating memory for applications. Operating system also gives an advantage to the programmer, as it provides some kind pf protection against programming failures for low level devices and gives a foundation where programs can run on [50].

2.2.1 Different operating system solution

The main difference with general purpose operating system and embedded operating system is that the footprint of the embedded version should be only a part of the size that the general purpose OS takes. The difference can be seen when exploring the systems mission. The limitation of memory, processor and interfaces limit the tasks that a real embedded computer can attempt. As the system operates on a narrow, pre-designed area it provides tasks that are useful for that purpose. All tough embedded devices are designed to be beneficial solutions for specific task the barrier between these two systems is nowadays becoming more and more invisible, as the amount of memory and process power is increasing.

Several companies and communities have started to develop operating system specially

designed for these embedded devices demands. Most of these solutions have chosen UNIX like approach to develop the system, as it holds clear solution for small simple device often without almost any external interfaces. As the market of embedded devices are growing a substantial efforts from major companies have been made to enter these markets, also with non Open Source distributions. Still regardless of the embedded OS all embedded systems are an entirety of several different layers of software providing drivers for the hardware and an interface to the end user.

The eCosCentric is the most popular UNIX-like clone used today. The advantages that eCos RTOS provides are that it is royalty free solution with easy configuration and strong networking support. It has also been ported to run on embedded devices and there exists many companies providing commercial and non-commercial embedded Linux solutions. When obtaining an embedded Linux distribution it usually includes a tool chain, some ported applications, ported libraries and of course the actual kernel. The tool chain enables you to build applications or to create the image for an embedded device. Most of these are based on a standard GNU tool chain, but there also exists some packages that require proprietary tools. The applications included to the package are a set of ported applications designed especially for that solution that can be compiled to image [30].

The biggest problem with eCos in commercial usage is that the GPL rules forces companies to keep the source code available to everyone, including the competitors. The license still allows applications and device drivers to be private if they remain separate from the kernel and do not contain any other parts from programs developed under the GPL. Some companies providing Linux distributions have even provided specially designed tools to check which parts of the software have GPL violations. One example of these companies is Lineo's GPL Compliance Tool. More problematic license in embedded devices is LGPL. In embedded device the whole application set is build in a single compact executable. To save space, the package is in many cases linked statically against the libraries. This means that the source code has to be available to all parts [49].

By dropping unnecessary modules and drivers from the general purpose kernel, it can be compiled in the size of 800Kb. Most embedded solutions are using this approach, but when adding some real-time support or supporting MMU-fewer solutions, more patching is required. The open source efforts against embedded Linux systems can be inspected from different angles depending how they are developed. One approach is to start to eliminate the unnecessary functionalities that are not required from the embedded device. One example of this stripped, small footprint operating system is uClinux. As the Linux itself can be

compiled to rather small package as it is, the standard Linux kernel itself can be patched against of the demands of the embedded device. Advantage if this system is that the Linux applications can be ported rather easily to this platform. Examples of these kind of solution are FreeRTOS, eCospro, AXIS and BlueCat Linux [5].

Embedded Linux is also distributed by some bigger commercial companies like Lineo, MontaVista and Red Hat. The main key behind these companies is that they provide support and professional services, a thorough documentation of their software development kits, maybe some specialized tools and systemization services. Some of these vendors, like Monta Vista provides also a real-time extension to the kernel which enables to device to have real-time performance. Two embedded solutions uClinux and eCosCentric are also focused on MMU-less processors. From these the eCosCentric provides an advantage by offering an active developer community and providing free support. It has also been ported to several different processor's and development boards. The eCosCentric kernel and tools are also totally free so for embedded solution it does not create any extra costs.

2.2.2 Managing embedded devices

Most common physical interfaces in embedded devices are serial and Ethernet interface. These are also in many times the only interfaces which allow physical management to be done to the device without any user interfaces like display or keyboard. From these the serial is basically used to create the end connection to the controlled machine, but at the development stage with a proper serial communication program it can also be used to connect to the embedded device. Here the host machine works as a terminal emulator for the embedded device. Serial is also often used to move the systems image to the device or to set the entire basic configuration to the board. The USB has nowadays also taken a place of acting as of connectivity interface to end device, but in embedded devices it is still quite rare.

The physical Ethernet with the connectivity application enables the device to be connected to the network. The TCP/IP's OSI model layer 4 has two protocols UDP and TCP providing the boundary between the user applications and host-layer protocols. Embedded controllers might provide several of different applications that can be used to connect and control the devices. Chapter four introduces the networking protocol provided by the eCosCentric platform.

Client-server programs in embedded devices are usually run as servers or daemons, to enable the connection to be made to the device. Two most known and used program in

this category are Terminal Emulation, Telnet and Secure Shell, SSH. Telnet provides the ability to remotely access to an embedded device. The telnet daemon runs on an embedded device allowing transmitting the keystrokes from the remote host to the target embedded device and displays the resulting screen to back to the remote host. SSH is a more secure version for remote connectivity. The idea is the same as in telnet, but provides secure encrypted communications between the devices. For each new connection a new daemon is created which handles the exchange, encryption, authentication, command execution, and data exchange.

The File transfer protocol, FTP is designed for file up- and downloading. In embedded devices it can often be used in both directions by running both a daemon and a client in an embedded device. This way we can easily place files in an embedded system and also send for example log information back to the host machine.

The Simple Mail Transfer Protocol, SMTP is usually in embedded device implemented as client software. It allows mail to be sent to the remote host when an event triggers it. Usually it is used for sending scheduled information to a known host.

The HTTP can be used as a web configuration, which makes it a powerful tool. The GUI of the HTTP page can be used as a virtual interface to control the device itself and the end system that it is connected to. Usually this enables only the basic set configuration to be done, but still this is extremely handy in embedded Linux as it regularly provides only a console based connection used with the predefined set of commands.

The Simple network message protocol, SNMP is based on asynchronous request / response commands. It is a widely accepted protocol, providing the management of different types of networks with a simple design that causes only a small burden to the network. As it is well known, it also has an extensive range of tool support. The agent of a SNMP protocol is running as a server in each of the monitored or managed device. It provides an interface to each of these nodes by providing a data structure called a management information base, MIB. SNMP provides also an ability to easily manage all the devices distributed to the field by running special management software running on the management station. The agent on the embedded device responds to the control stations query or setting and acts on it. The protocol also enables the card to act as a proxy toward the end system by holding a specially designed for that task. The agent end is also able to send asynchronous traps to the end station when a predefined event occurs [50].

2.3 Hardware

The basic architecture in embedded device is usually the same when it is designed to provide network connectivity to the end system. They usually hold and serial interface to connect the embedded device to the end system, an Ethernet plug for network connectivity and of course a Micro-Controller Unit and some kind of system memory. The different hardware solutions usually concern the processor type, size and type of the memory and the interfaces that the device provides. When designing the hardware solution, the design goals can meet the customer markets by competing with better solution than the existing one or by getting the product to market faster than the competitor.

The best substitute for devices without a hard drive is flash memory which can be designed to emulate a drive. Flash also gives an advantage of being less power consuming, faster and space efficient than standard storages in table PC's [50]. The processors in embedded devices have two major alignments. Ones holding the Memory Management Unit, MMU and others that are designed without it. The advantages gained from the MMU are that it provides memory protection against the applications, but with precautions it is possible to run MMU-fewer versions on a smaller device for compact embedded applications. Of course it also gives a cheaper solution for designing the controller.

The features that are required from the embedded devices are a possibility for long term autonomy, cost efficiency, low power consumption and general reliability. The device is to achieve reliability from the software and hardware. They are used to sense the outside world and control the device in industrial surrounding so a failure in embedded device might mean that the controlled machine is damaged. This requires a thorough testing from the hardware and the software side. The cost efficiency is also a key feature, as the devices are usually distributed to a field, a small change in the price affects significantly to the end purchasing price.

2.4 Programming embedded devices

In spite of the usage of embedded device, the design goal is to try to put more computing power using cheaper CPU's. The amount of memory is often limited to few kilobytes so the decision of what are the different applications integrated with the embedded OS are carefully to be taught through. The embedded device, at least on industrial usage, can stand alone for a long period of time so the software and hardware on the device should

never fail. This undresses the device from mechanical parts that are more sensitive for failures and require more complex drivers and well tested software to be added with the OS.

Unlike software designed to general purpose PC, the embedded software cannot be used in other embedded devices unless it is modified. This is because embedded devices hardware layout of the boards may differ significantly from each other. Different solutions of operating systems often consists a specific tools for compiling applications especially for that processor or board. Still addresses in memory locations and other offsets of the physical interfaces may vary and needs to be modified. Compilers many times have also options for building the code to several specific processor types.

The code generated for the embedded device is compiled in the host machine, where the development tools exist. This process is called cross-compilation. It means that the host machine, regularly a normal table PC, holds the resulting tool chain that will eventually the create binaries for the target embedded machine. The main idea is that the compilation can be done in a machine with more capacity than the target platform. After the compilation the whole package including the operating system kernel and the applications are concatenated to a single package which can be loaded to the embedded machine.

When developing software to embedded devices, thorough testing is required. In worst cases the stand alone device can freeze up totally by badly designed and tested software. This is why generated executable should always be first tested in a host machine, to minimize the existence of errors. Usually some modification might be needed after wards, before it can runs on a target machine. The generated code size is limited by the physical memory of the device. Also processor might do some restrictions by not having enough processing power to run several complex programs at the same time. Programming languages used in embedded devices is usually done with assembly as a low level language and with a C as an application level language. Both of these languages allow programmers to access directly to hardware and not have any really complex data structures.

Usually the tool chain provides also some debugging tools, to easy up developers job. The commercial packages might also include some development studious with GUI included debuggers with multiple options. Example of this is CodeWarriors development studio. Many Linux distributions still relay on GNU tool, and its GNU debugger.

2.5 Medical Device Safety Requirements

There are two types of safety requirements in IEC 61508: safety function requirements and safety integrity requirements. The safety function requirements state that the input/output sequences that perform the safety-critical operation. For example, an ECG could have a blood pressure sensor (input) that can reach a maximum value (algorithm) before the alarm is initiated (output) to buzzer.

The safety integrity requirements of a system are composed of diagnostics and other fail-safe mechanisms used to ensure that failures of the system are detected and that the system goes to a safe state if it's unable to perform a safety function.

The IEC 61508 standard says no; it recognizes that not every element of a system has the same effect on safe operation and therefore allows some modules to be justified as independent. For this reason, the developer, must deploy a modular design method and define clear interfaces and protection mechanisms between those modules, so that you can definitively classify subsystems into critical and noncritical categories [36]. You can do this by using hardware memory protection through a memory-management unit or by using a language that enforces such encapsulation (Ada, C++, Java, Modula, and so forth).

Note that IEC 61508 allows the entire safety system to be implemented in this fashion, including peripheral hardware, communication paths, computing hardware, and software. Each element can be implemented and assigned an SIL (Safety Integrity Levels) that together determines the SIL of the entire system [36].

2.5.1 Software Safety

The Software requirement for medical devices is defined within the IEC 60601-1-4, FDA requirements, which state that medical devices software must meet the corresponding design-input document's requirement which is design by the organisation.

For the most part, software verification testing is a manual process. Software developers maintain spreadsheets that link verification test results to the associated requirements and the internal quality policy. These results become part of the devices design history file [35].

Software development and debugging tools are used to improve the quality of the software. Among other things, these tools identify memory leaks and inefficient processes and perform array boundary checks.

2.5.2 Hardware Safety

The design for medical devices and hardware development must comply with numerous standards. Safety is the main areas of focus during medical device hardware verification. The requirements for the electrical safety of medical devices are much more stringent than those for other electrical devices. The reasons for increased precautions include:

1. Patient may be connected to several medical devices simultaneously (e.g. in intensive care)

2. Patient may be connected conductively to electronic circuitry (e.g. ECG monitoring)

The standard has electrical requirements intended to reduce the electrical hazards both under normal and single fault conditions. Unlike other standards, electrical safety is not considered to be dependent on voltage, but on leakage currents. This is because even a very low voltage, when applied to internal tissue, can cause leakage currents through the body which may be fatal. Clause 19 of IEC 60601-1 gives the requirements for leakage currents [1].

Circuit separation is used as one of the means to implement electrical safety in medical electrical equipment. Certain types of circuits must be physically separated from each other, and must be electrically separated from each other by means of dielectric strength tests, which are specified in the standard.

Chapter 3

Hardware design

3.1 Hardware Architecture and Schematic Design

The goal of this chapter is to propose a functional design as stated in Chapter 1. The functional design consists of three parts. First, the Infrastructure has to be developed. The Infrastructure consists of the basic parts, which are power, clock and reset circuits. Second, the logic presentation of each feature has to be determined which meets requirements stated by the sponsor specification. Third, the Medical device connectivity. Serial to Ethernet terminal server is specially designed for the medical environment. The terminal server allows connection of all RS-232 bedside medical devices to the network, sending data to the hospital information system. The features are all parts of the specification stated in Section 1.2. All the electrical connections between peripheral have to be determined. After designing each part of the Infrastructure, the Protel design tool is used for schematic design entry. The following steps are performed with the Altium 6.7 software in order to create a schematic design:

1. Creation of logic symbols for each components.
2. Annotation of all components as a preparation step for the PCB design.
3. Creation of electrical connections between all the components according to component pcbcart manufacturer guidelines.
4. Verification of the schematic design using the Protel Electrical Rule Checker (ERC). The ERC examines the schematic design for both electrical inconsistencies, such as short circuits, e.g., output pins connected to each other, and drafting inconsistencies, such as unconnected net labels or duplicate designators.

Several months of work were dedicated solely to the design of the hardware platform. The process of selecting parts and ensuring the correctness of the final schematics proved to be quite time consuming. The design and schematics went through several iteration before being transformed to a PCB layout.

3.1.1 Central Processing Unit CPU

The Micro-Controller which has been considered for the design is ARM cortex M3 from STMicroelectronics. This Micro-Controller incorporates the STM32F103RE ARM Thumb 2 processor core, 512KB of flash memory and 64KB of on-chip SRAM, a nested vector interrupt controller, 51 IO lines, 6 times, 4 USART (Universal Synchronous/Asynchronous Receiver/Transmitter) and a 2 watchdog timer. The connection between the peripheral and CPU is shown in Figure 3.1

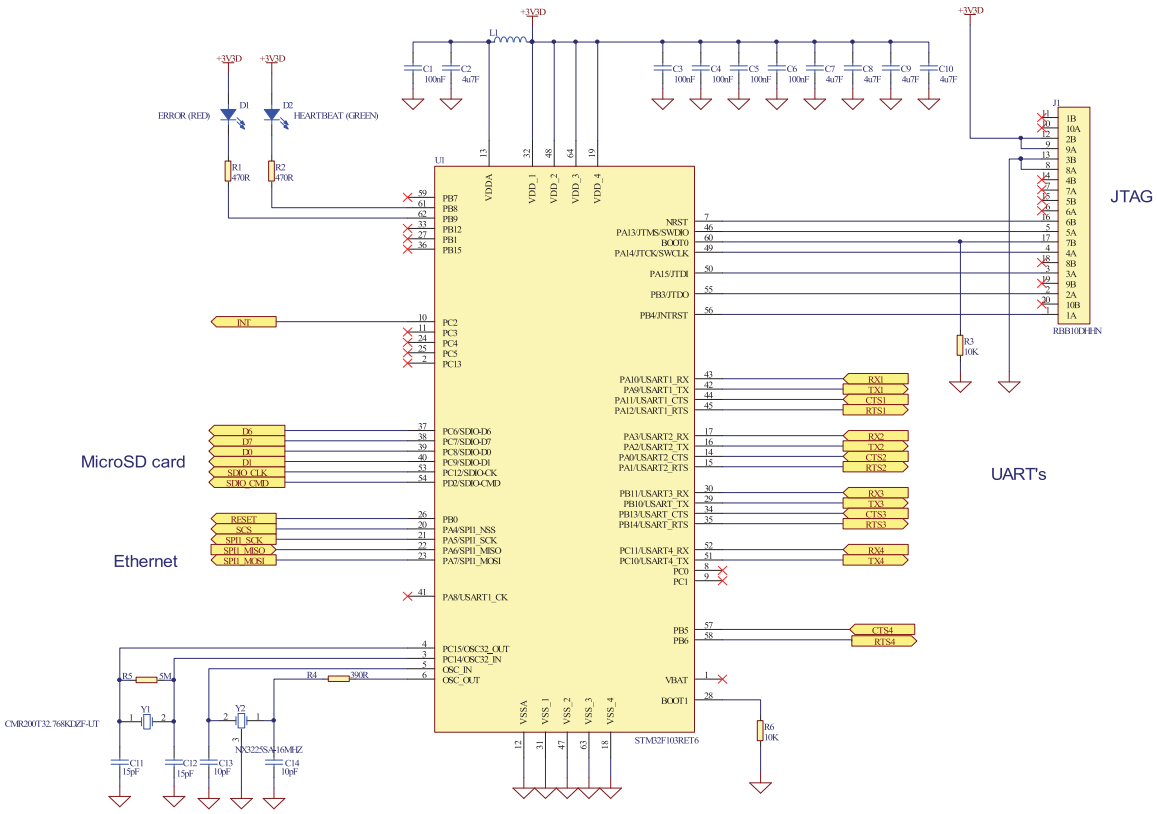

Figure 3.1: *Cortex arch M3*

3.1.1.1 STM32F103 ARM Cortex M3 Thumb2 Processor

The STM32F103RE processor core is an implementation of the ARM7 architecture. This architecture includes both the 32-bit ARM instruction set and the 16-bit Thumb Instruction set. The 16-bit Thumb instructions duplicate the most commonly used ARM instructions and are used to gain better performance when interfacing with 16 bit wide memory.

A three-stage instruction pipeline (fetch, decode, execute) is implemented in the processor core to increase the speed of execution.

The STM32F103RE processor core is capable of accessing memory as single bytes (8-bits), half words (16-bits) or words (32-bits).

An embedded in-circuit emulator is also included for debugging purposes. This can be accessed through a JTAG port.

3.1.1.2 Interrupt Controller

The Interrupt Controller (IC) controls the interrupts generated from the internal peripherals, as well as peripherals connected to the external interrupt lines.

One of the external interrupt lines is the Fast Interrupt Line (FIQ). Interrupts occurring on this line have higher priority than all other interrupts. An 8-level priority encoder allows the user to specify the priority of regular interrupts.

Internal interrupts can be configured to be level- or edge-triggered, while external interrupts can be configured to be high/low level- or positive/negative edge-triggered interrupts.

3.1.1.3 Timer/Counter

The STM32F32RE includes 6 identical 16-bit timers/counters. These may be used independently, or chained together to create 32- or 48-bit counters. Each counter is capable of generating interrupts via an internal connection to the interrupt controller.

The counters may be driven by one of internal clock signals, or one of external clock signals. Each counter also has 2 general-purpose I/O lines that can be used to measure or generate various kinds of waves.

Only internal timer functions may be used on the Micro-Controller board, as all external timer I/O pins are used by other devices for I/O. Thus, these pins are programmed in the PIO for external device use.

3.1.1.4 USART's

The STM32F103RE contains 4 Universal Synchronous/Asynchronous Receiver/Transmitters (USARTs) for serial communication. Each USART operates independently, and may be programmed for different speeds, parity, data and stop bits.

The baud rate clock of the USART can be generated either from internal clock signals, or an external clock signal. The baud rate clock may only be set from the internal clock signals on the Micro-Controller board, as the external clock pins are used for other I/O devices.

Each USART is also capable of generating interrupts via an internal connection to the interrupt controller.

3.1.1.5 On-Chip SRAM

The STM32F103RE incorporates 64KB of on-chip SRAM. This SRAM is internally connected directly to the 32-bit data bus, and allows 32-bit wide single cycle data accesses.

3.1.1.6 Bus Matrix Interface

External memory and peripherals are attached to the CPU via the Bus Matrix. The Bus Matrix has 12 chip selects and a 24-bit address bus with the upper 4 bits of the address bus multiplexed with a chip select. The Micro-Controller board uses a 22-bit address bus. The address bus is controlled by the programmable input output and is used as a chip select. For each chip select, the Bus Matrix allows the user to specify:

1. the data bus width (8 or 16 bits);
2. the number of wait states to be used in accesses;
3. the device page size;
4. the byte access type (byte write or byte select);
5. whether the chip select is active high or active low; and
6. the base address of the device

3.1.1.7 Programmable Input Output `PIO`

The Programmable Input/output controller (PIO) controls the 51 programmable I/O lines. The 51 I/O lines are multiplexed with signals from the on-chip peripherals.

The PIO allows each of the 51 multiplexed lines to be controlled by either the on-chip peripheral or the PIO. All I/O pins are bi-directional, and may be designated as inputs or outputs by the PIO.

The PIO also allows each I/O line to generate an interrupt whenever its level changes

3.1.1.8 Watchdog Timer

The watchdog timer is used to reset the system in the event of a software deadlock. The timer is a 16-bit down counter. If the counter reaches 0, the watchdog will generate an internal reset to restart the system.

3.1.2 Power Supply(Switch Mode)

The STM32 requires a single power supply which must be in the range 2.0V to 3.6V. An internal regulator is used to generate a 1.8V supply for the Cortex core

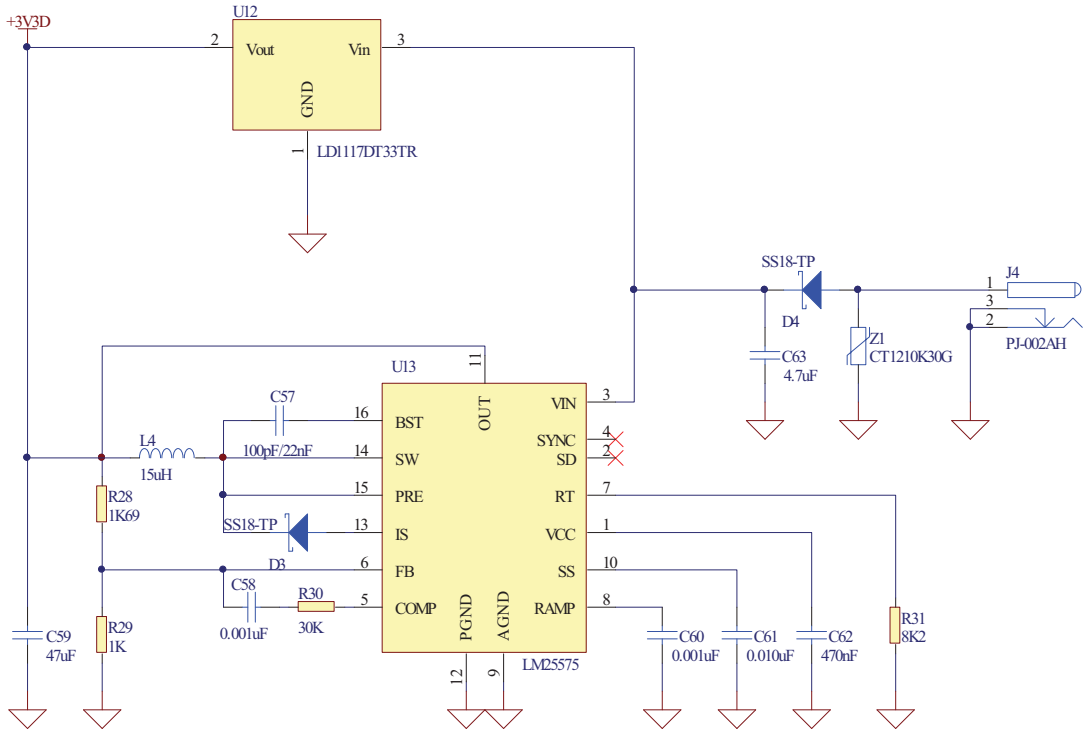

Figure 3.2: *Switch mode power supply*

Power to the main board is supplied by LM25575 a 3.3V regulated switching power supply capable of passing 1.5A of current. A 47uF filtering capacitor is placed on the 3.3V rail, providing filtering and a current reservoir. The PCB uses a 2-layer design with a

ground plane on the solder side in order to reduce noise. The STM32 are powered by the 3.3V rail. The full design of the switch mode power supply is shown in Figure 3.2

3.1.3 Serial port with optical isolation

The device has a 4 USART controller integrated in the Micro-Controller. The USART can be utilized to send/receive data characters from a peripheral device or modem and converts the data signals between serial and parallel and vice versa. In order to use the UART the following is required:

- A component that is able to convert the 3.3V signals originating from the MCU device to RS-232 [20] (+/ - 12V) line voltages and vice versa. This device is also called a line-driver.
- The USART interface should include a RJ45 to RS-232 connector cable, which implies to specific connections to the 4 port RJ45 connectors on the board.
- The line-driver should support speeds up to 238.400 bit per second which is the maximum rate achievable by the MCU device [40].

In order to support a full RS-232 port on the receiver side (RXD and CTS) and the transmit side (TXD and RTS) signals originating from the Micro-Controller device are connected to a MAX3238CPWR line-driver from Texas Instruments Products [20] Figure 3.3 . The line-driver device supports up to 1Mbit per second which is sufficient for the maximum rate required by the devices. Furthermore a decoupling capacitor is used between the power supply and ground connection for filtering purposes.

The opto-isolator is intended to provide electrical isolation between an interface and the equipment connected to its serial port. This required when the system is dealing with completely different voltage levels.

Figure 3.3 shows how the electrical isolation is achieved. Connector J2A is linked to the serial port of the device. IC U2 is used to isolate the I/O lines The other side of the isolator carries TTL levels. This side is powered by the target system power supply. IC U2 is used to buffer the signals for the opto-isolator and also drives the data. The interface has been tested at the baud rate of 38.4k baud.

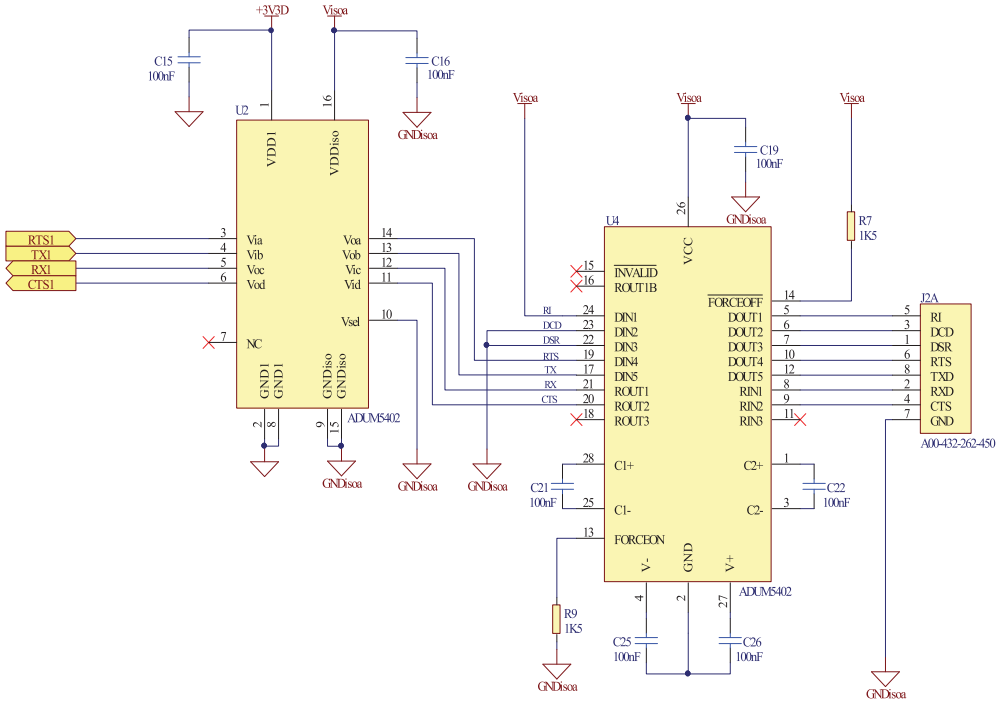

Figure 3.3: *Isolated serial port*

3.1.4 Ethernet Controller (Wiznet W5100)

There are several Ethernet controllers available in the market. Three Ethernet controllers have been investigated. The first Ethernet Controller is RTL8019, which is NE2000 compatible and was made for ISA-bus usage. This Network Interface Controller (NIC) needs 16 lines for operation, that means almost quarter of the I/O-Ports of an STM32 were taken.

Improvements came with the ENC28J60, which has an SPI-Interface and needs only 4 interface lines. Both solutions require a software TCP-IP-Stack which takes about 6kB of program memory.

The next step is the W5100 which needs only 1/3 of the code space and supports the SPI-Interface. The following description in Table 3.1 refers only to the SPI-Interface.

NIC	code size	I/O lines
RTL8019	6kB	16 lines (8 data, 5 addresses, 3 control)
ENC28j60	6kB	(4 SPI-lines)
W5100	2kB	(4 SPI-lines)

Table 3.1: *Code space required by different Ethernet controller using SPI interface*

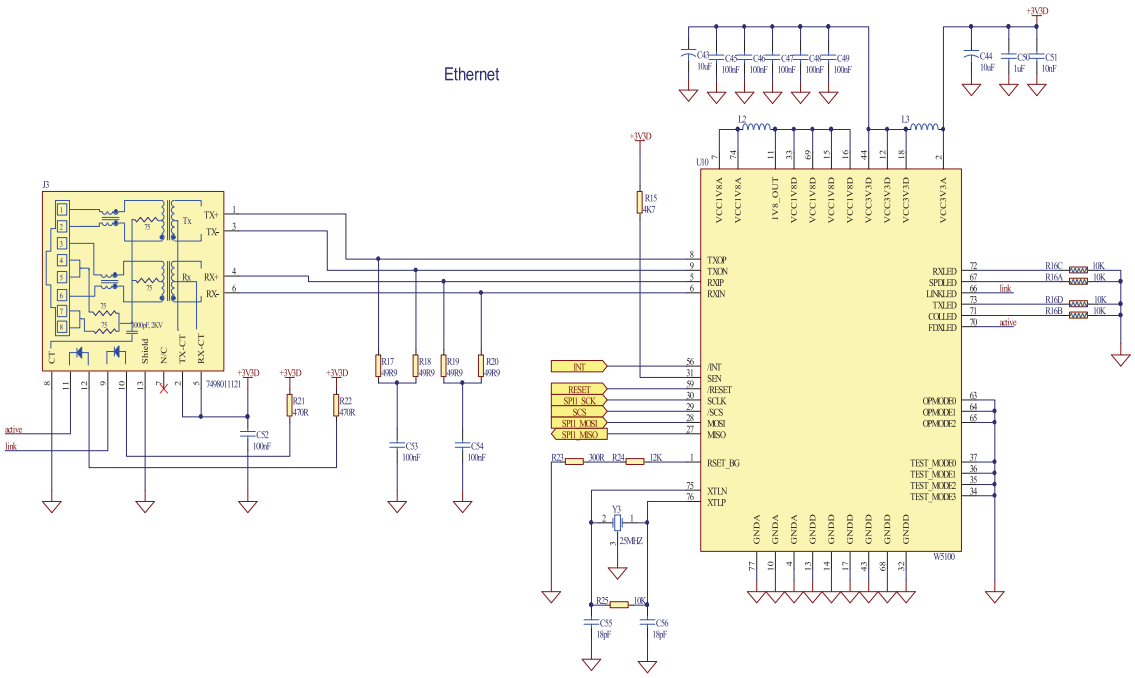

Figure 3.4: *Ethernet controller*

The Wiznet W5100 was chosen as it provides a very easy to use interface to the Internet. It provides 4 sockets which can be used for any number of network connections. To control the Wiznet module, several interface options are offered: SPI, indirect bus, and a full data/address bus, and also the w5100 is a 3.3V IC with 5V tolerant pins. The SPI bus was chosen since its the simplest implementation as shown in Figure 3.4, the required bandwidth was relatively small, and many Micro-Controllers have SPI peripherals on-board

The boot loader support of the STM32 series provides a mechanism for uploading program code by the MCU itself. While the code space for this mechanism is limited, it was not common to upload program code via Ethernet, using the RTL8019 or the ENC28J60.

The W5100 hardwired TCP/IP Stack allows bootloader to use Read-While-Write Self-Programming to the MCU.

3.1.5 Micro SD Card Storage

The Micro SD card is one of the core parts of this project. Its main purpose is to store all four isolated RS232 channel data into an external storage hence the memory card. This project has been tested with 256 MB Nokia SD memory card. The memory card is connected to CPU thru SDIO (Secure Digital Input output) bus. The following cards sizes are supported by this project are 128MB, 256MB. Figure 3.5 shows the interface to the Micro-Controller

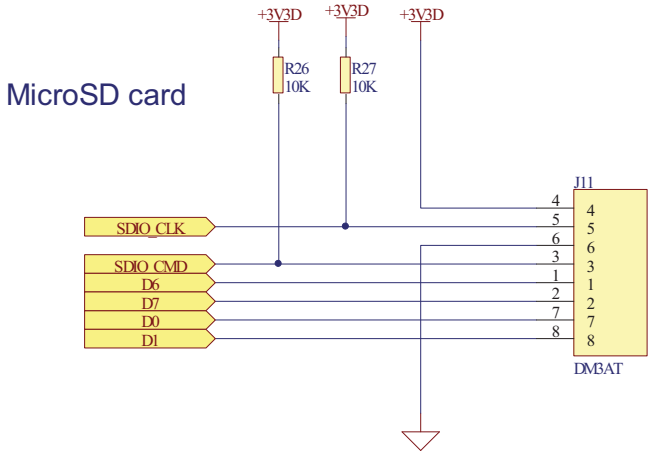

Figure 3.5: *Implementation of the Micro-SD card*

3.1.6 JTAG

The STM32 MCU has its own on-chip debug system. The Cortex M3, ARM7 and ARM9 CPUs have as a minimum a JTAG port which allows a standard debug tool to connect to the CPU and download code into the internal SRAM or FLASH memory. The JTAG port also have the capability to run basic control (single step and setting breakpoints etc) as well as being able to view the contents of memory locations. The full JTAG implementation is shown in Figure 3.6

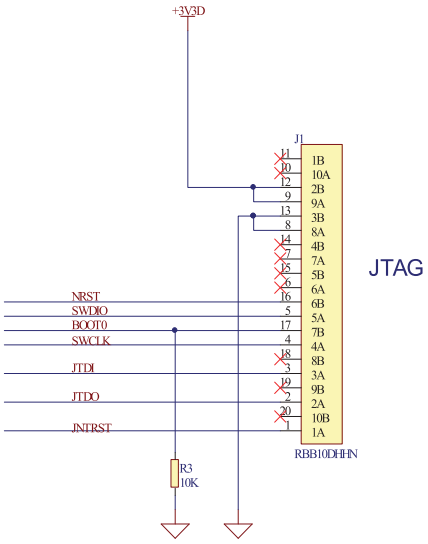

Figure 3.6: *JTAG interface*

For This project, the physical connections for the JTAG interface include 4 wires.

- TCK - Clock. The JTAG clock driven by the MCU
- TDI - Data Input driven by the MCU
- TMS - JTAG mode input to the MCU
- TDO - Data Out. JTAG data output by the MCU

3.1.6.1 JTAG Adapter

JTAG adapter is been designed for the debugging purpose it consists of serial port for the general console debugging and PCI slot for the main board to establishes the connection between the PC and physical hardware. The fallowing Figure 3.7 show the full implementation

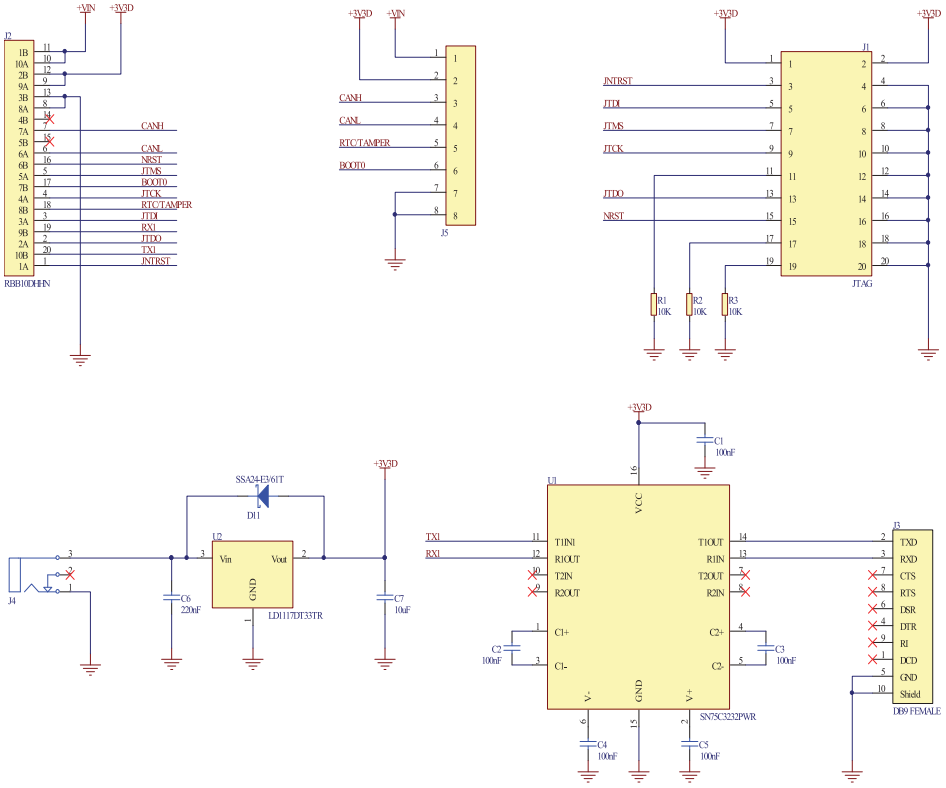

Figure 3.7: *JTAG adapter interface*

of the adapter

3.1.7 Final Implementation of the Prototype Boards

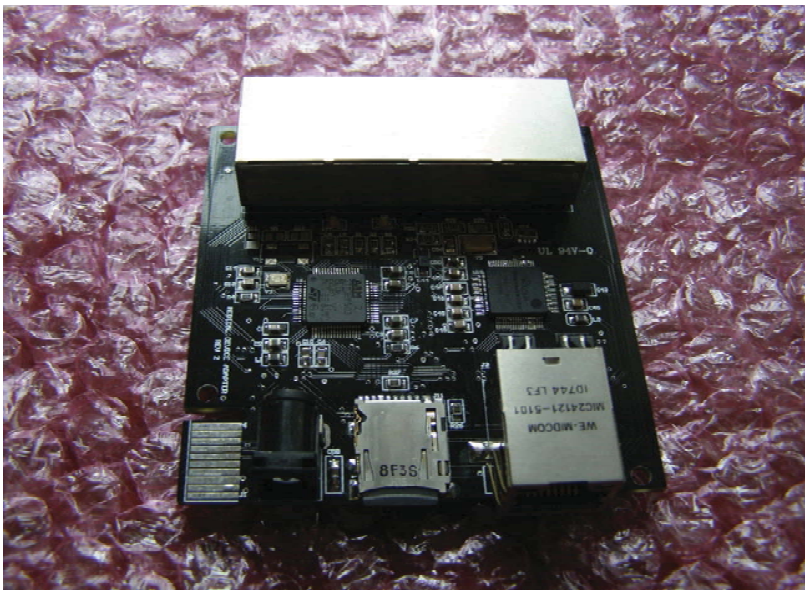

Figure 3.8: *Top layer of the Final Prototype Board*

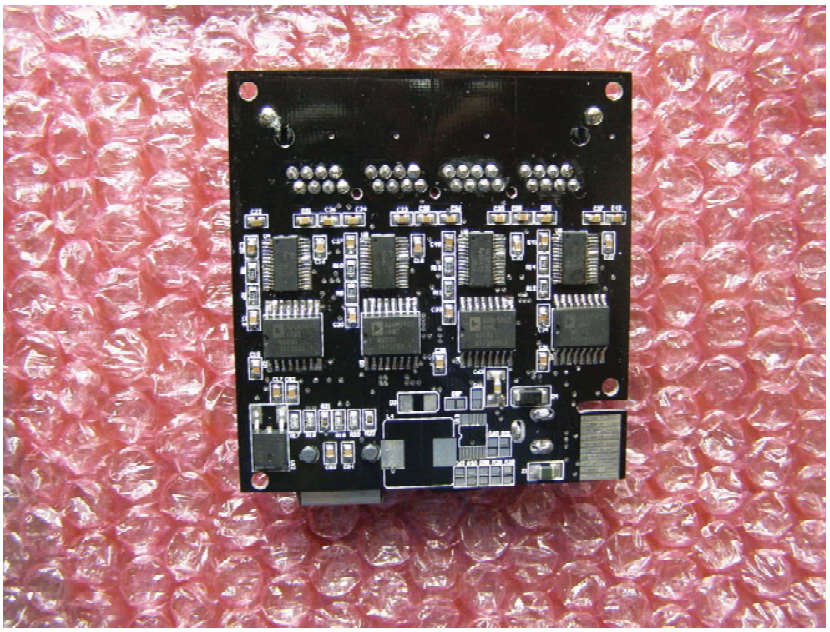

Figure 3.9: *Bottom layer of the Final Prototype Board*

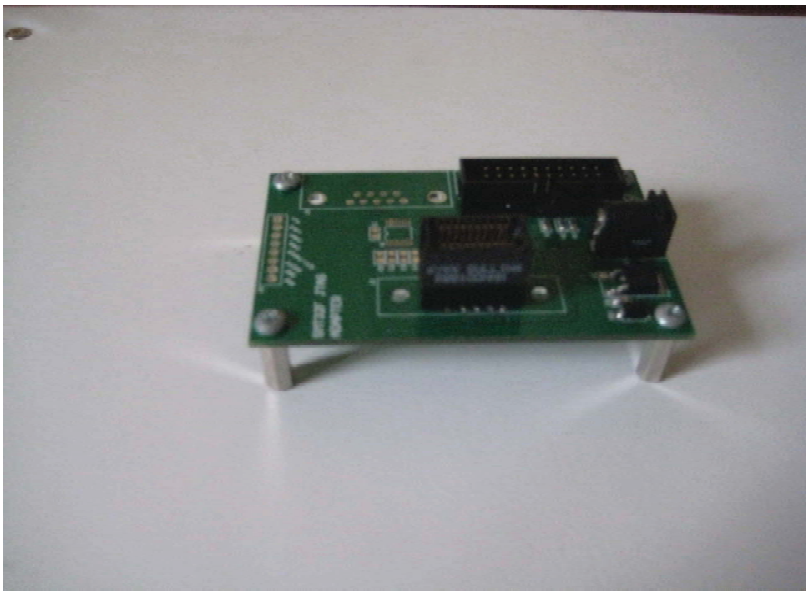

Figure 3.10: *Final implementation of the JTAG Adapter*

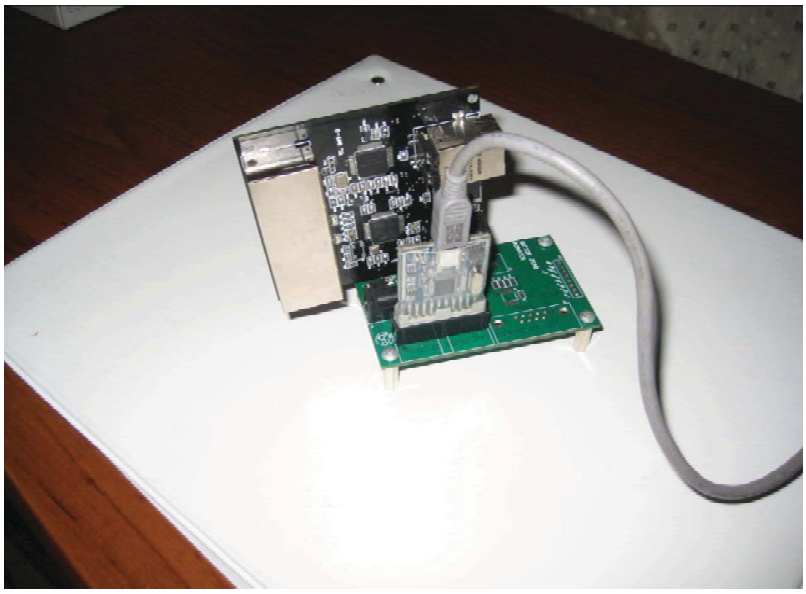

Figure 3.11: *Full picture overview of the system*

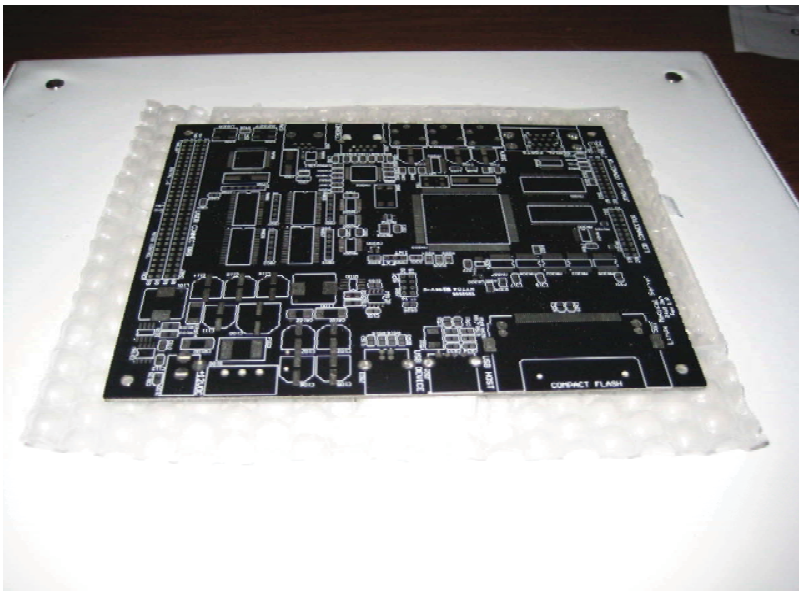

Figure 3.12: *Initial design of this project - top layer [4 layer board]*

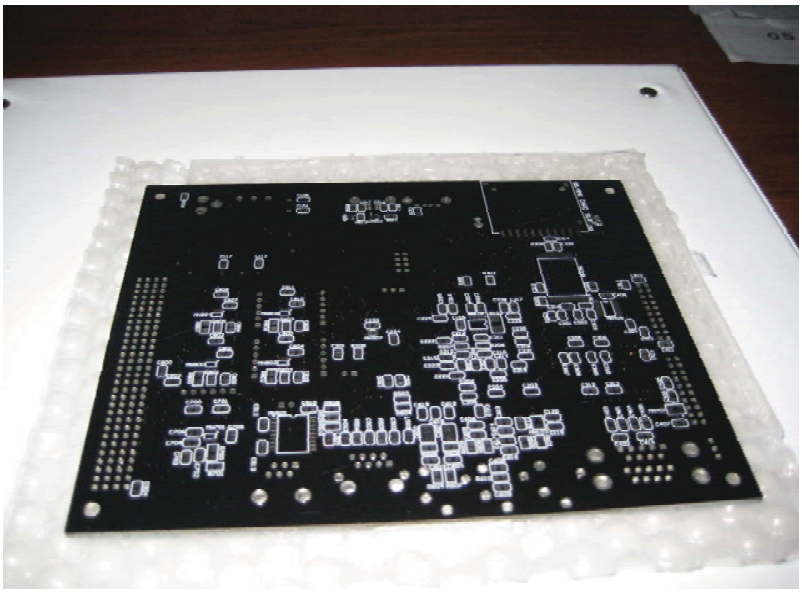

Figure 3.13: *Initial design of this project - Bottom layer [4 layer board]*

3.2 Printed Circuit Board Design (pcb)

A Printed Circuit Board (PCB) contains multiple layers stacked on each other with each layer containing a copper pattern. The function of the copper pattern is to provide electrical connections between the components mounted on the PCB. In order to design a PCB a number of steps are needed. First, the PCB Design Rules are extracted from component manufacturer data sheets. PCB Design Rules are indication which the PCB has to comply with in order to guarantee a functional PCB according to specifications. Second, each component requires a footprint, i.e., a mark on the board which consists of lands pads, with correct mechanical dimensions for the component, and solder mask prints. Third, the PCB requires some floorplanning, i.e., the placement of the components on the PCB which apply to the design rules. Finally, trace routing is performed, i.e., the placing of the electrical connections between components in the form of copper traces. During each design-step a Design Rule Verification is performed [29]. The Design Rule Verification compares the PCB layout with the PCB Design Rules in order to detect design errors which need to be corrected. The tool used for the PCB design is the Protel software from Altium [2]. Figure 3.14 shows the PCB layout of the main board and Figure 3.15 shows the JTAG

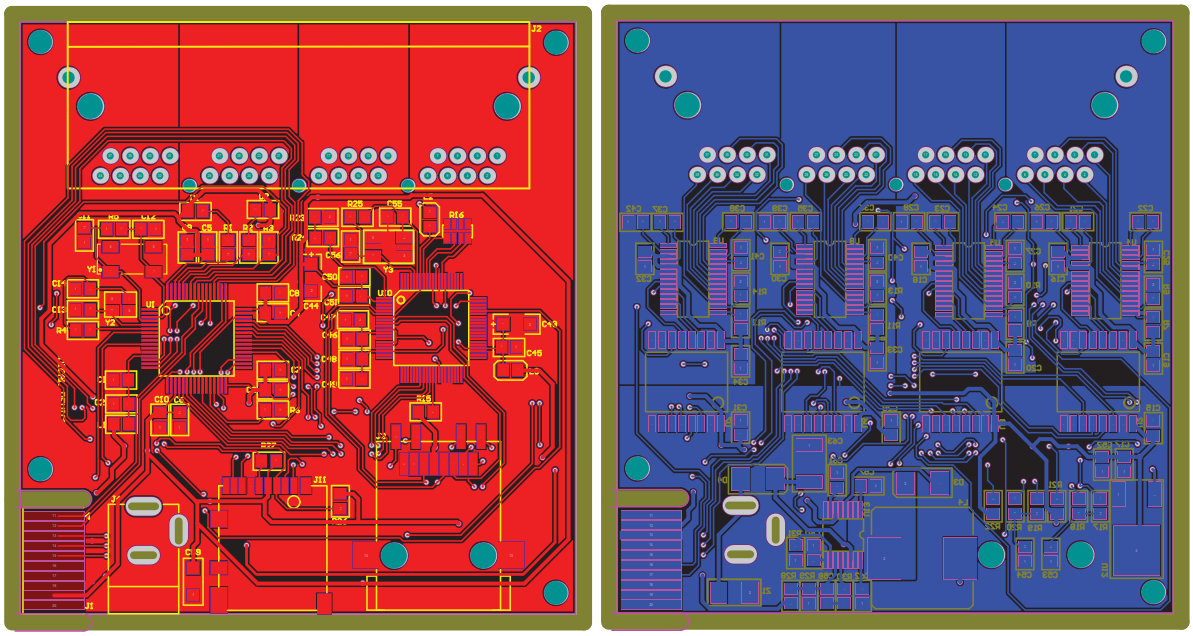

Figure 3.14: *Top and bottom layout of the main board*

adapter.

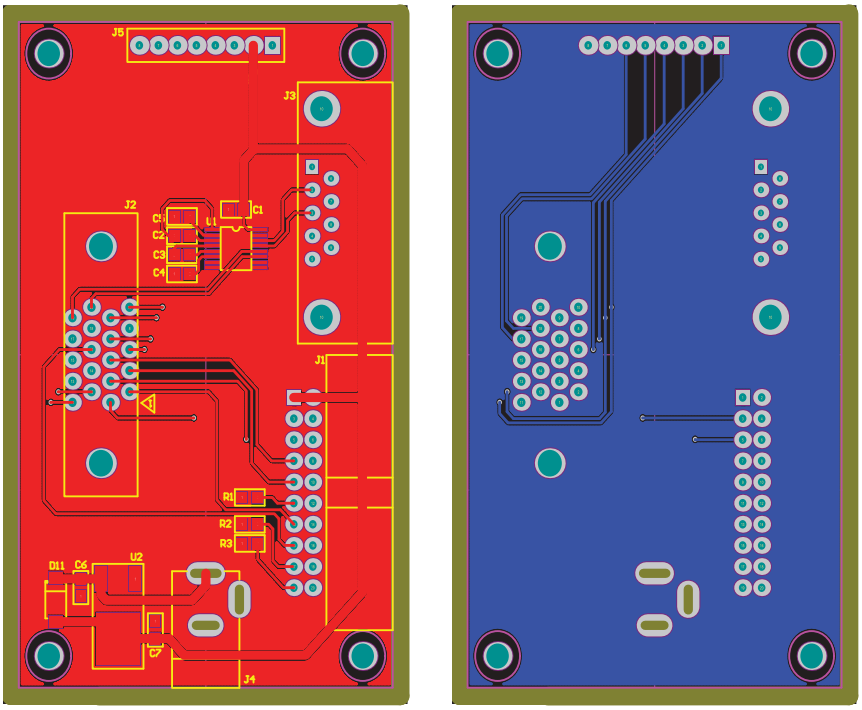

Figure 3.15: *Top and bottom layout of the JTAG adapter*

3.2.1 Overview procedure

A Printed Circuit Board (PCB) is found in almost every electronic device and its purpose is to provide electrical connections between components mounted on it. The electrical connections between the electronic components are implemented with copper traces which carry the signals. Generally there are three types of PCBs:

- Single-sided PCB.
- Double-sided PCB.
- Multi-layer PCB.

The single-sided PCB has only through-hole components at one side of the board while traces (and optionally Surface Mount (SMD) components) are routed on the other side of the board. The double-sided board uses both top and bottom side of the board for components and traces. Finally, the multi-layer board is the most complex, it has besides the top and bottom layer, one or more layers embedded between top and bottom side for electrical routing. As PCBs and PCB component sizes decrease continuously, less room is available for copper traces to be placed and thus more layers on a PCB are required [29].

A typical PCB design has to follow the following steps:

1. Functional design analysis: PCB design rules are extracted from component manufacturer data sheets and PCB guidelines stated in the schematic design. PCB design rules are directives which the PCB has to comply with in order to guarantee a functioning PCB according to specifications. Additional requirements are stated which apply specifically for the PCB, e.g., placing mounting holes.

2. Footprint creation: Each component used must have a PCB footprint, which consists of land pads; solder mask prints, and additionally fiducials. Fiducials are etched features used by the optical system of a pick and place machine as a reference target. The footprint is created according to PCB design rules and component restrictions, e.g., the minimum size of land pads on a footprint.

3. Floorplanning: A floorplanning for the PCB must be performed. Floorplanning is the placement of the components on the PCB which complies with PCB design rules. During floorplanning the layout for power and ground distribution is also considered. Furthermore all additional features must be placed, e.g., placement of PCB mounting holes.

4. Trace routing: After the floorplanning step of the PCB all the connections between the components must be routed. The usual order for routing with high density PCBs is to start with the fan-out routing of all high-density Surface Mount-components, e.g., more than 84 pins in one package. Fan-out routing is routing every land pad out of the footprint. This step is usually done with high- density components in order to be able to connect every component pin. Second is to route all critical traces such as high-frequency and high-voltage signals, because these signals usually have more routing constraints than all remaining signals and thus require maximum flexibility in routing. Finally, the remaining signals are routed.

5. Design rule verification: design rule verification is done during each step of the design-flow in order to prevent errors due to PCB design rules. When errors are detected, proper redesign of the PCB is done.

3.2.2 PCB Design Considerations

In order to specify the PCB requirements the following factors are considered [3]:

1. Trace width and trace clearance requirements: The minimum PCB traces width and clearance on the signal and power and ground plane layers.

2. Vias: The type of vias on the PCB.

3. Floorplanning: The placement of the components on the PCB.

4. PCB Mask Layer: The use of a protective solder mask and paste mask for assembly.

Note that the order different considerations are discussed is from one to four. This is because considerations are dependent on previous decisions, e.g., the type of vias required is among things dependent on the minimal trace width used for dense components.

3.2.2.1 Trace width and trace clearance Requirements

The minimum trace width and trace clearance chosen for the PCB is 5mil in order to provide a maximum possible supply current over the traces. Furthermore most PCB manufacturers require a minimum trace width of 5mil as a manufacturing specification. As a result of this decision the minimum via diameter is 20mil with a hole-size of 10mil. In order to support protection against voltage spikes of up to 1kV a 6mm trace clearance is used for PCB as clearance between low-voltage components and high-voltage traces [29].

3.2.2.2 Type of vias on the PCB

The PCB board only uses through-hole vias in order to reduce PCB design complexity and cost. In addition all vias and traces can be tested when required.

3.2.2.3 Floorplanning

The routing of all components floorplanning issues must be considered. The following factors are important [29]:

- Physical requirements of the PCB
- High-voltage signals
- Stable power distribution
- Heat-dissipating components

After analysing the above requirements a floorplan layout is proposed, which is finalized and checked by senior engineer during the implementation stage.

3.2.2.4 PCB Mask layer design consideration

The PCB includes an additional 4 mil extra spacing around all the pads in the solder mask layer in case of a slight misalignment

3.2.3 PCB Implementation

All the footprints for the PCB are created according to PCB design rules and PCB manufacturers recommendations. There are two different types of footprints, standard footprints available from libraries, e.g., 0805 footprint for resistors, and custom footprints which have to be made manually. An example for a custom footprint is the LQFP Ethernet controller footprint. After footprint creation the PCB floorplanning was done.

As you can see in Figure 3.16, Figure 3.17, Figure 3.18, and Figure 3.19 this board was the initial design, due to the manufacturing, fabrication and component cost the sponsor has decided to put this design on hold, there for this design has not been fully described in this thesis.

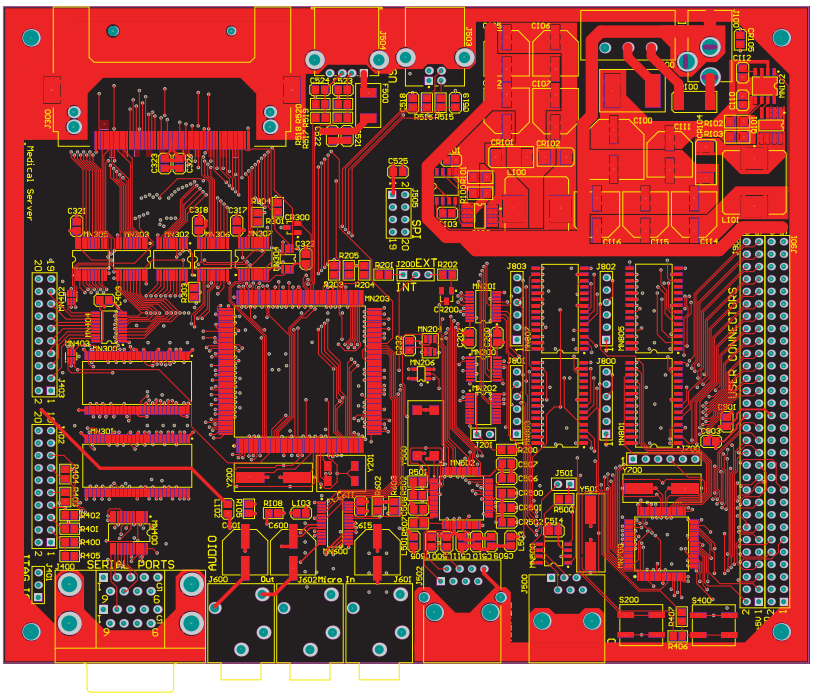

Figure 3.16: *Top layout of the initial board*

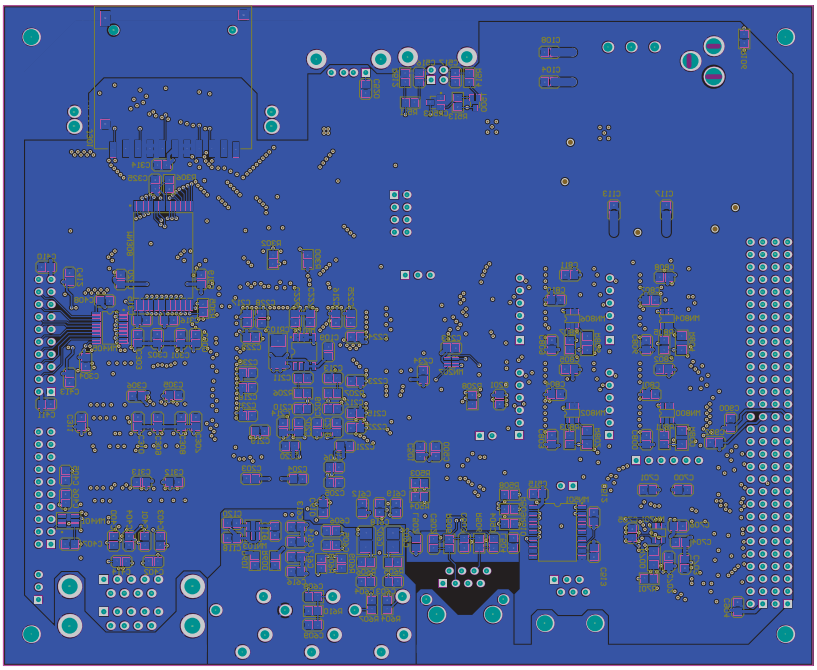

Figure 3.17: *Bottom layout of the initial board*

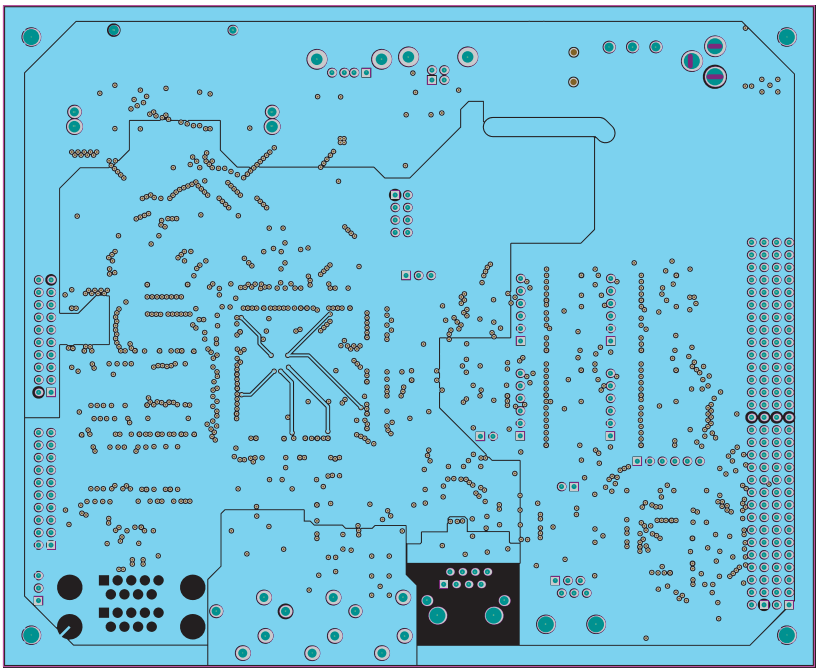

Figure 3.18: *Ground plane of the initial board (inner layer)*

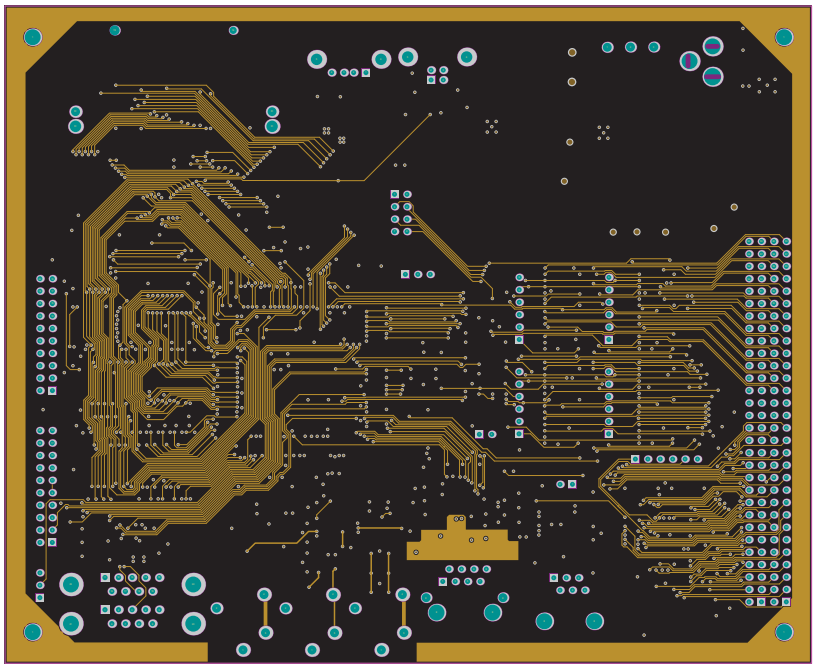

Figure 3.19: *Power plane layout of the initial board (inner layer)*

Chapter 4

Software Design

4.1 Real-time Operating System eCosCentic

Real-time operating systems (RTOS) in general are designed for various kinds of real-time applications and the development of such RTOS is becoming increasingly important. Computers are used in cars, aircraft, manufacturing assembly lines, and other control systems to provide the functionality of these real-time systems. Given random inputs of the peripheral control devices must be processed by the computer and the used operating system in a given amount of time and the resulting output has to be provided within specified deadlines [7]. Frequently and periodically required tasks have to be started and stop at predefined times and have to run during determined time intervals. The behaviour during processing has to be deterministic and undefined states have to be intercepted. A RTOS is valued more for how quickly and predictably it can respond to a particular event than for the given amount of work performed over time. Therefore the key factors in a RTOS are minimal interrupt and thread switching latency. The Kernel of the operating system has to deal with these requirements and implement adequate scheduling mechanisms [7].

The amount of available RTOS is continuously growing, both for commercial distributions and for open source variants. Compared to the commercial RTOS VxWorks, Tornado or Windows CE, eCos is royalty free like Linux or UClinux. It was written by Cygwin Solutions [8] with the motivation to develop a small, configurable real-time operating system. eCos is able to deliver a comparable performance to commercial products although it is totally open source. Nowadays eCosCentric belongs to Redhat and the development is due to the huge eCosCentric community still in progress. A wide range of ports to general platforms is already available [27]. eCosCentric, like other conventional operating systems,

seeks to reduce the burden of application development by providing convenient abstractions of physical devices and highly tuned implementations of common functions [15]. It has been designed in such a way that a small resource footprint can be developed. It is extremely configurable and allows developers to select components that satisfy basic application needs and to configure the OS for the specific implementation requirements for the application. eCosCentric uses compile-time control methods, along with selective linking (provided by the GNU linker) to give the developer control of its behaviour, allowing the implementation itself to be built for the specific application for which it is intended. The small footprint size, configurability and portability of eCosCentric make it the RTOS of choice for this implementation.

4.1.1 eCosCentric Source Tree `Roadmap`

In order to ease the usage of this document Figure 4.1 gives an overview which components can be found in which part of the eCosCentric source tree. Due to clarity reasons only directories are listed which actually contain relevant files.

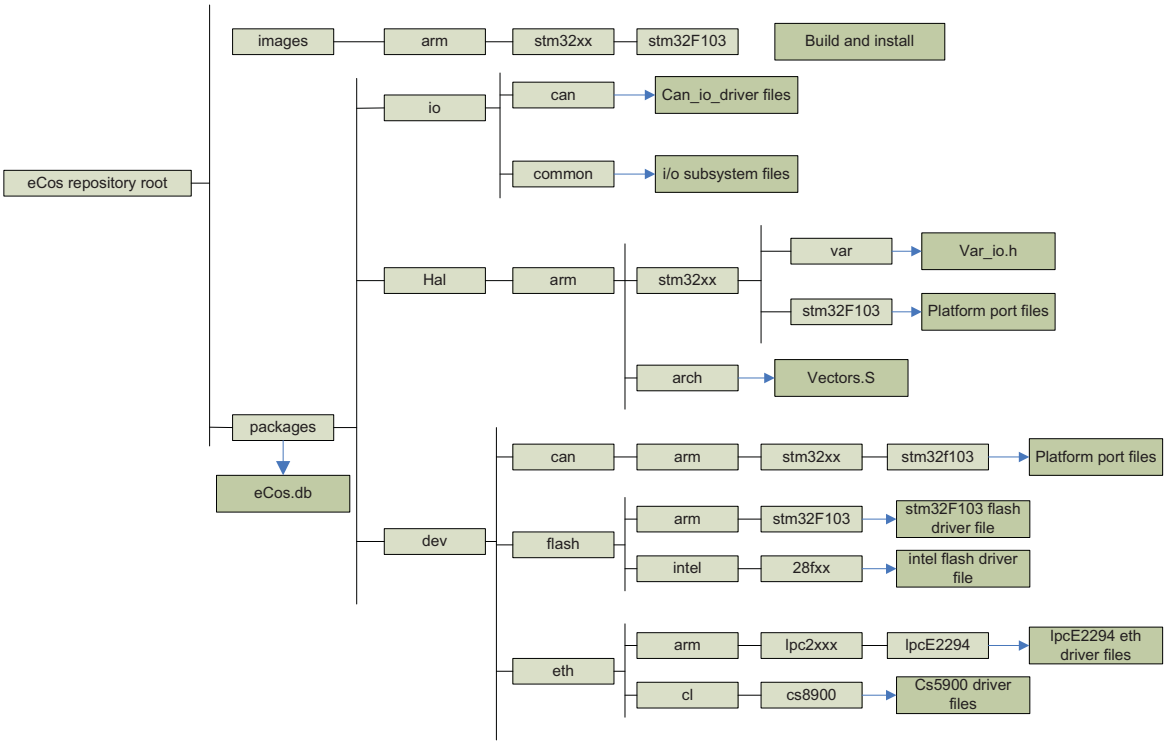

Figure 4.1: *eCosCentric File Directory Roadmap*

4.1.2 The eCosCentric Architecture

eCosCentric is based on a layered software architecture. Application portability and reuse of software is enhanced by encapsulation of target specific hardware from the application. As shown in Figure 4.2 all modules above the dashed line are absolutely hardware independent. The Kernel, the networking stack and the file system build together with the upper compatibility and library layers a consistent platform for the application layer.

The RedBoot, the Hardware Abstraction Layer (HAL) and the device driver layer have to be configured for any specific hardware. The effort for such a port depends on how similar the new target platform is to former ported hardware.

The Redboot module itself uses the HAL to get access to the specific hardware, therefore in some figures Redboot is placed above the HAL. Furthermore the hardware independent modules already belong to the application layer. The web server, networking stack, the Kernel and even the file system do not have to be implemented in a configuration if this is not necessary. This fact shows the difference between eCosCentric and a common PC operation system. eCosCentric does not require a file system using executable or editable directories. Compared to UNIX that fact is a special difference, since UNIX even accesses CPU data (/proc) or hardware (/dev) using files.

4.1.3 Hardware Abstraction Layer

The HAL is a key component for the portability of the eCosCentric system. Regardless to the specific hardware the higher software layers can use the HAL interfaces to access the full functionality of the platform. The encapsulation allows portability of all others infrastructure components and eases the port process.

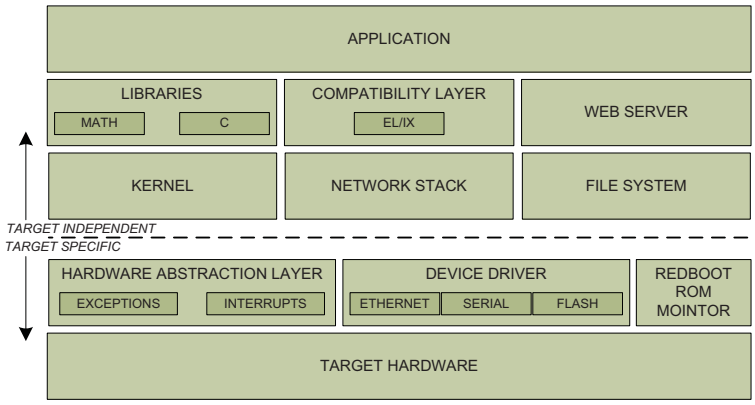

Figure 4.2: *Hardware Abstraction Layer[27]*

The HAL is typically built up with three modules: the architecture, the variant, and the platform module. The architecture sub-module contains all supported processor families, e.g. the ARM Cortex M3, ARM7 and ARM9 family. For each family the sub-module includes, among others, the code for CPU start-up, interrupt delivery and context switching [27].

A variant is a special processor within a family and is contained by the variant sub-module, e.g. the STM32F103RD [ARM Coretx M3] variant. The variant sub-module activates CPU features like caches or the floating point unit.

```
#For the ARM:
    #define HAL_ENABLE_INTERRUPTS()
    asm volatile (
    "mrs r3,cpsr;"
    "bic r3,r3,#0xC0;"
    "msr cpsr,r3"
    :
    :
    : "r3"
    );

#For the PowerPC:
    #define HAL_ENABLE_INTERRUPTS()
    CYG_MACRO_START
        cyg_uint32 tmp1, tmp2;
        asm volatile (
        "mfmsr %0;"
        "ori %1,%1,0x8000;"
        "rlwimi %0,%1,0,16,16;"
        "mtmsr %0;"
        : "=r" (tmp1), "=r" (tmp2));
    CYG_MACRO_END
```

The most specific part of the HAL is the platform sub-module which contains the code for a unique piece of hardware. Typical sources in this module are the platform start-up routines and the chip select configuration. All HAL packages can be found under the HAL subdirectory of the CVS repository. The code above shows an example of how the HAL_ENABLE_INTERRUPTS() function is implemented for two different architectures. The source code for the ARM and the PowerPC architectures is totally different although it causes the same effect on both architectures [27].

4.1.3.1 HAL Start-up

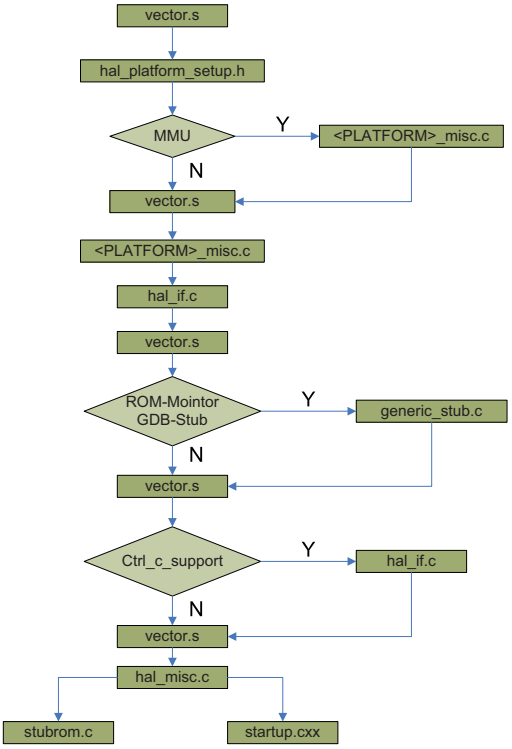

Figure 4.3: *eCosCentric startup procedure[27]*

The most important file of the start-up is vectors.S which controls the whole system initialisation. Most of the initialisation code can be found in different modules since all CPU's of an architecture use the same vectors.S files. The start-up process is shown in Figure 4.3 and described as following[1]:

1. **vector.s:** Jump to the hardware initialisation code at the label reset_vector. The init code can be found in the hal_platform_setup.h in the macro PLATFORM_SETUP.

2. **hal_platform_setup.h:** General hardware initialisation (refer to Section 4.3.2.2)

3. **<PLATFORM>_misc.c:** This module includes the final memory layout.

4. **vector.s**: In the next step CPU depending code is executed. The rourtine depend on the type of startup respectively whether a ROM_Monitor or an application is started.

5. **<PLATFORM>_misc.c:** In this file the hardware specific initialisation functions

[1]The notation <FILE>::<FUNCTION> references the function <FUNCTION> in the file <FILE>

are implemented. The function hal_hardware_init initialises among others the interrupt controller, starts the timer of the system and calls hal_if.c::hal_hal_init.

6. **hal_if.c:** In this module various "deamons" of the operating system are implemented. hal_if_init initialises these "services" and registers them in a data structure.

7. **vector.s:** If a ROM-Monitor or a GDB-Stub is built the function generic-stub.c::initialize_stub will be called.

8. **generic-stub.c:** This module contains the functionality for remote-debugging.

9. **vector.s:** If the support for ctrl-c break-support is activated for the debug mode, the function hal_if.c::hal_ctrlc_isr_init will be called.

10. **hal_if:** hal_ctrlc_isr_init activates an interrupt handler which stops the application received a break signal.

11. **vector.s:** Function hal_misc.c::cyg_hal_invoke_constructors is called.

12. **hal_misc.c:** The function cyg_hal_invoke_constructors calls all constructors which are marked in the constructor table. This one is labeled by the linker (__CTOR_LIST_ and __CTOR_END__).

13. **vector.s:** Depending whether a ROM-Monitor or an application is built the function stubrom.c::cyg_start or the function startup.cxx::cyg_start is called.

14. **stubrom.c:** The function cyg_start consists only of a endless loop which executes the break-point function on each pass.

If an application is built the further start-up process can be seen in section 4.1.5.1. Further information about the eCosCentric startup on ARM devices can be found in "Embedded Software Development with eCos - Book" from Anthony J. Massa [27].

4.1.4 The Redboot ROM Monitor

Redboot is an acronym for "Red Hat Embedded Debug and Bootstrap". This phrase describes already the two most important functions of Redboot. On the one hand it is used for the initialisation of the hardware components and for the start-up of the operating system. On the other hand Redboot supports debugging which is quite important on embedded systems [27]. Some of the features provided by eCosCentric are listed below and described in more detail in section 4.1.4.1:

- Boot scripting support
- Command Line Interface (CLI) for monitor and control support
- Access via serial or Ethernet ports
- GDB suppor
- X/Y modem support
- Network bootstrap support using BOOTP or static IP address configuration

Redboot consists of a slimed version of eCosCentric without Kernel and without any applications. However it includes the HAL and is therefore portable to all platforms which are supported by eCosCentric. Figure 4.4 shows the block diagram architecture of some of the features included with the RedBoot ROM monitor. The interaction between Redboot and the eCosCentric application varies depending on the configuration option settings in both images.

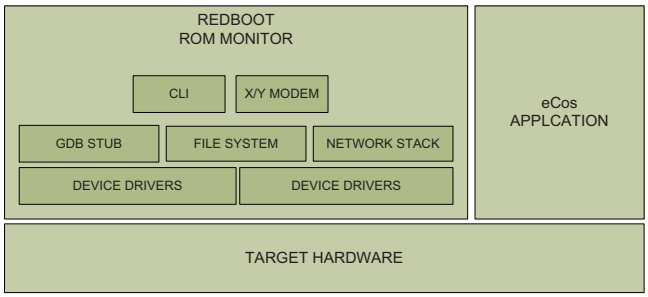

Figure 4.4: *Redboot ROM Monitor Architecture*

4.1.4.1 Redboot feature

RedBoot provides a Command Line Interface (CLI) for monitor and control support. The command line can be accessed via a serial port and also via telnet if networking support is included.

RedBoot provides some basic debugging functionality, allowing the user to examine regions of memory, compare one region of memory with another, calculate the checksum of a region of memory and fill a region of memory with a specified bit pattern.

Using the command line, it is possible to load program images using x/y modem (serial communication protocols) and, if networking support is included, TFTP and HTTP.

RedBoot is also capable of writing images to internal flash, although currently there is no support for multiple sized flash sectors. To overcome this problem, only 64KB sectors may be written with RedBoot.

If compiled with networking support, RedBoot is able to dynamically obtain an IP address using BOOTP. It is also possible to statically assign the IP address to be used by RedBoot.

RedBoot also includes a GDB stub. This allows connection to the target platform from a GDB host for application debugging. The GDB host can connect with the target via a serial interface, or, if networking support is included, via TCP/IP.

4.1.4.2 Virtual vector calling interface

The purpose of a ROM monitor is to load and run other program images and to provide base services to programs being debugged. RedBoot provides these services through the Virtual Vector Table (VVT). The VVT is a table of 64 pointers to service functions. The VVT contains entries for configuring and using console and debug I/O; resetting the board; setting breakpoints; flushing data and instruction caches; delaying; and retrieving any RedBoot configuration data stored in flash. It is placed in a static memory location allowing the functions to be shared between RedBoot and an real time operating system application. For example, by using the VVT, an eCosCentric application is able to print diagnostic messages using the same console port and settings as used by RedBoot [27].

4.1.5 Kernel

The main feature of any RTOS is the Kernel. The eCosCentric Kernel provides, selectable scheduling, mechanisms for thread synchronisation, interrupt and exception handling, counters and clocks, and much more. These standard functionalities of the eCosCentric Kernel are highly configurable. This allows the RTOS to be adapted to any specific needs and furthermore the footprint of the Kernel is kept at the lowest possible level. The eCosCentric Kernel was designed under special consideration of a `low interrupt latency`[2], a `low task switching latency`[3] and a `deterministic behaviour`[4]. Furthermore the eCosCentric Kernel provides assertions that can be enabled or disabled within the eCosCentric package. Enabled assertions during debugging allow the performance of certain error checking

[2] the time an interrupt occurs and the ISR starts
[3] thread activation time
[4] Kernel performance must be predictable and bounded to real-time requirements

and ease the development process. In the following sub-sections 4.1.5.4 and 4.1.5.4 describe the interrupt and exception handling of eCosCentric. Corresponding systems calls for the interrupt and exception management will not be listed. All required information about these can be found in the eCosCentric Reference [16] and in Anthony Massaas eCosCentric book [27].

4.1.5.1 Kernel Boot Procedure

After all hardware initialisation is complete, the Kernel start-up procedure is implemented by the HAL calling the core function cyg_start. This function calls further start-up routines to handle various start-up tasks. All default initialisation functions can be implemented for a specific application by using the same function name in the application code. The cyg-prestart function, which is first called by the cyg_start function, can be used in order to initialise components needed prior to other system initialisation. The cyg_package_start invokes the initialisation functions of other components as the C library before the cyguser_start functions is called. This is the usual application entry point.cyg_user_start can be used in order to perform any application-specific initialisation, create threads or register necessary interrupt handlers [27]. The scheduler is started when the cyg_user_start functions returns.

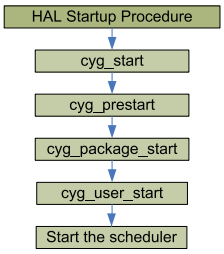

Figure 4.5: *Kernel stratup procedure*

4.1.5.2 Startup Modes

Each port of the eCosCentric operating system generally provides three startup types: ROM, RAM, and ROMRAM. The Micro-Controller port of the eCosCentric operating system only provides support for ROM and RAM, as ROMRAM support is provided by the on-board boot software.

The design goal behind these different startup modes was to simplify application devel-

opment. An application being developed uses the RAM startup mode. A ROM monitor such as RedBoot is then used to initialise the board, load the RAM image and execute the program. The ROM monitor also provides debugging communication between the application and the debugger. Once the software has been debugged, the startup mode is simply changed to ROM and the application is then a standalone unit that can be programmed to flash. The ROM image of the application includes all the code necessary to initialise the board and provides optional debugging support [27].

ROM

The ROM startup mode is used for building the RedBoot ROM monitor as well as final builds of applications suitable for programming to flash.

All read-only sections of code are programmed to internal flash, as is the initial value of the data section. Upon boot, all the system initialisation code is executed, including relocating the data section from ROM to RAM.

RAM

The RAM startup mode is used to test and debug applications that are still under development. The board is first booted using RedBoot or another ROM monitor. The ROM monitor is responsible for the initialisation of the board and loading the program image. Once the program image has been loaded the ROM monitor passes control to the application that may use services provided by the ROM monitor for debugging.

ROMRAM

The ROMRAM startup mode is used for the same purposes as the ROM startup mode. The difference is that the program image is copied from ROM to RAM at startup for faster execution.

The Micro-Controller board does not provide ROMRAM support in eCosCentric as this function is provided by the bootstrap software.

4.1.5.3 Schedulers

The scheduler is used to select the appropriate thread for execution, to provide synchronisation methods and to control the effects of interrupts. To keep interrupt latency low interrupts are not disabled during scheduling. Interrupts increase a lock counter which disables the scheduler if the counter is non-zero. The lock counter values are manipulated on the one hand by the interrupt handlers provided by the HAL and on the other hand by threads. eCosCentric offers two types of schedulers a multilevel queue scheduler and a bitmap scheduler.

Multilevel Queue Scheduler

The multilevel queue scheduler allows threads to be assigned with a priority between 0 (highest priority) and 31 (lowest priority). Yet multiple threads can be assigned to one priority level at the same time. Lower priority level threads are halted as long threads with higher levels are processed. If two or more threads with the same priority level occur time slicing mechanisms ensure that each thread is allowed to process during its predetermined execution time.

Bitmap Scheduler

Like the multilevel queue scheduler the bitmap scheduler allows priority levels between 0 and 31. Although only one thread is allowed to execute at every level. This makes the bitmap scheduler more efficient and simpler hence time slicing is not required.

4.1.5.4 Interrupt Handling Mechanism

An interrupt is an asynchronous external event typically related to some hardware action such as the push of a button or timer expiration. Since an interrupt can occur at any time running Kernel threads have to be stopped in order to process the interrupt. To avoid corrupt data states due to an aborted or paused thread and to reduce interrupt latency in the system eCosCentric employs a two step interrupt handling scheme [27]. The interrupt service routine (ISR) which belongs to the interrupt is processed immediately to guarantee a fast interrupt handling. This routine owns only very restricted rights, e.g. an ISR cannot start a thread avoiding the interruption of other critical threads. If an ISR needs to start

a thread this task is executed by a called Deferred Service Routine (DSR). A DSR has a high priority, is allowed to start threads and is therefore organized by the scheduler.

The occurred interrupt needs to be masked to avoid that it is not called again until the DSR has not finished its processing. Usually the ISR masks the current interrupt and the DSR unmasks it after processing.

4.1.5.5 Exception Handling

An exception is a synchronous event initiated by a process error during the execution of a thread. A proper exception handling is extremely importing to avoid systems failures and to improve the robustness of the software [27]. After an exception occurs, the processor jumps to a defined address (or exception vector) and runs the instructions (exception handling code). Architectures may differ by its implementation of the jump process and the location of the exception handlers. For an embedded system the simplest and most flexible method to handle exception is to call a function. The thread can restart its process after this function has finished successfully. The called function needs some area to operate and some further information like the exception number. eCosCentric provides two methods for exception handling.

HAL and Kernel Exception Handling

The first method (default) is a combination of HAL and Kernel exception handling. The HAL offers a basic hardware level exception handling and passes the control to the Kernel for further operations.

Every exception supported by a processor needs to have a corresponding exception handler. If no handler is installed for a particular exception the processor jumps to an address where no code is present. This can lead to significant problems for the whole system.

The HAL uses a Vector Service Routine (VSR) which is designed in all HAL-packages. The VSR is an array of pointers to exception handler routines. The processor gets the required address of the relevant exception handler out of this table. The eCosCentric HAL offers a default VSR table that guarantees a basic operation of exceptions. These default instructions store the current processor status, call the Kernel handlers for following processes and restore the status of the processor. The only supported architecture which does not use the HAL VSR table is the ARM architecture. The ARM architecture routines

can be found in the vectors.S file, which defines separate handler routines for each exception it supports.

If a ROM monitor was configured the `cyg_hal_exception_handler`[5] calls the `__handle_exception` routine which proceeds with the execution of the exception (Figure 4.6). The `__handle_exception` routine manages breakpoints, single stepping and debug package protocol communication for debugging. The second Kernel-level configuration option allows the application to install its own handler for exceptions to take care of any further processing.

Application Exception Handling

Application Exception Handling is the second handling method provided by eCosCentric. This method allows the application to take over all the control over the exception handling. After an exception occurs the processor directly vectors to an application VSR. Afterwards

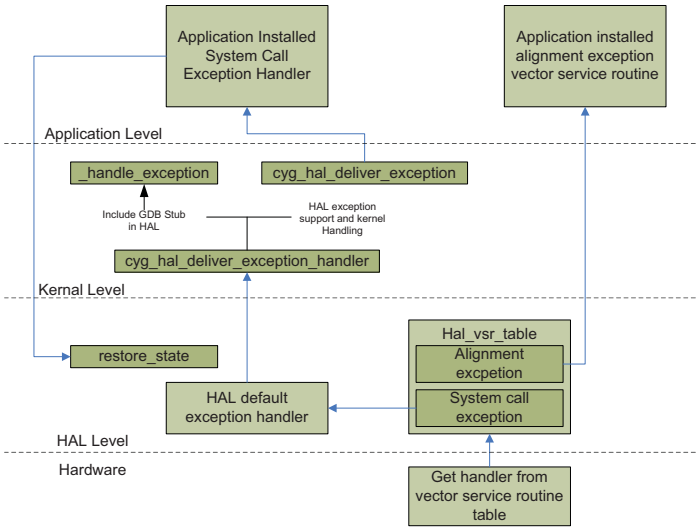

Figure 4.6: *eCosCentric exception handling and execution flow[27]*

the application handler is responsible for storing and restoring the processor's state.

4.1.6 Thread Synchronisation

eCosCentric provides several mechanisms to coordinate the access to shared resources and for synchronisation of threads. Corresponding systems calls for the interrupt and exception

[5] hal_misc.c under the HAL arch subdirectory

management will not be listed. All required information about these can be found in the eCosCentric Reference [16] and in Anthony Massaas eCosCentric book [27].

4.1.6.1 Mutex

Mutexes are used to allow different threads to share resources. Mutexes ensure that only one thread at a time can use a data as long the data is locked. As long as a special thread owns the mutex no other thread can access the data. Only the owning thread is allowed to unlock a mutex. To avoid priority inversion, which can occur if threads with different priorities share resources, the multilevel queue scheduler offers two security mechanisms - the static priority ceiling or the dynamic priority inheritance. Both security functions are based on the principle that a thread, which owns a mutex and might potentially block another thread with a higher priority, gets a higher priority for a short period of time. Using the priority ceiling mechanism the systems checks which threads have ever owned the mutex. The currently owning thread gets the highest registered priority. Although only already registered priorities are noted. The priority inheritance mechanism checks the priority of a thread which tries to become owner of the mutex and increases the priority of the current owner to the same level. This procedure is more efficient since the priority of a thread is only increased if it is necessary [27].

4.1.6.2 Semaphores

A Semaphore is a protected variable which restricts the access to a shared resource. Semaphores can be distinguished between to types - counting and binary semaphores whereas the binary semaphore can be seen as a special case of the counting semaphore. Counting semaphores increment their integer value when a thread posts to it. If the semaphore value is not zero, the thread with the highest priority gets access to the data and the other thread(s) have to wait. The resource is instantly available when the semaphore contains a zero. A binary semaphore can contain only two values whereas ”0” indicates that the resource is available and ”1” locks the resource.

4.1.6.3 Flags

Flags signalize a waiting thread whether a special condition is given and therefore the thread can start its process or if not the thread has wait for its starting condition. Flags are 32 bits words which can be set by one or more threads. The initialisation condition can be a single bit or a combination of several bits.

4.1.6.4 Spinlocks

Spinlocks are flags used to lock a piece of code there with the processor/thread has to wait in a loop until the spinlock is unlocked. When the spinlock is unlocked the processor sets the flag and continues its execution. Since the processor has nothing to do while waiting, it is important that spinlock are not hold for longer than 10 to 12 instructions [27].

4.1.6.5 Condition Variables

Condition variables are used together with mutexes to access shared resources. One thread produces the data and signalizes one or more waiting threads that the data is available. Thereby the message can be send to only one thread or via broadcast.

4.1.6.6 Message Boxes

Message boxes, also called mailboxes, are used by threads to send information towards each other. Thereby one thread will produce a message, which usually consists of more than one byte, and send it to other threads for processing.

4.1.7 i/O Control System

The eCosCentric I/O Control System provides an interface to any application to access the target hardware functions. It consists of two major modules - the I/O subsystem module and the device driver module. Applications use the I/O subsystem as a standardized interface which manages the communication to the hardware specific device drivers. As shown in Figure 4.7 this layered approach splits the I/O Control System into a hardware independent API interface and hardware aligned driver layer. Both modules are configurable and selectable using the Configuration tool.

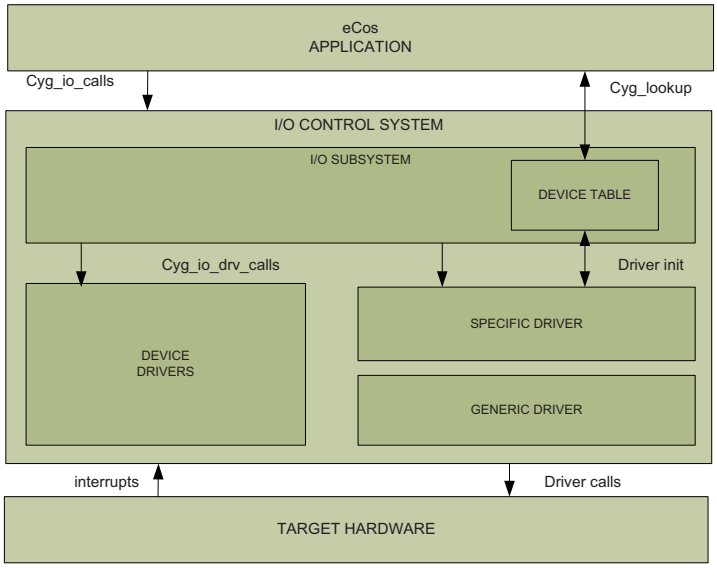

Figure 4.7: *The eCosCentric I/O subsystem Architecture[27]*

4.1.7.1 I/O Subsystem

The I/O Subsystem provides a standard API which uses handlers to access the low level hardware. Each handler points to the device and its driver. The handler is linked to the device in the device I/O table which is implemented in the I/O subsystem. An application gets this handler by calling the cyg_io_lookup function by sending the unique name of the device (e.g. "/dev/eth0") which is stored in the I/O device table. The I/O subsystem offers four functions to any application, which are include in the io/io.h header, listed in Table 4.1. More information, especially about the function parameters, can be found in the eCosCentric Reference [16].

I/O Functions	Describtion
Cyg_ErrNo cyg_io_lookup()	Returns the handler linked to the called device.
Cyg_ErrNo cyg_io_write()	sends data to the device
Cyg_ErrNo cyg_io_read()	receives data from the device
Cyg_ErrNo cyg_io_get_config()	obtains run-time configuration about a device
Cyg_ErrNo cyg_io_set_config()	manipulates or changes the run-time configuration of a device

Table 4.1: *I/O API*

4.1.7.2 Device Drivers

A device driver is the entity which provides the implementation of the I/O function to a specific piece of hardware. It also covers the control of the interrupt handling. eCosCentric supports, but does not require, a layered device driver architecture to enlarge portability. Often a generic reusable driver provides the access to the device and the upper layer then has the flexibility to add features and functions not found in the lower layers. Otherwise a standardised hardware independent driver can be reused and hardware specific routines can be provided by a low level driver.

4.1.8 Network Support

Internet connectivity are becoming a standard requirement for remote control devices, this mean the ability to send and receive data over the Internet or private network. The core connectivity of this network stack are shown in Figure 4.8, which highlights some of the common protocol supported by RTOS

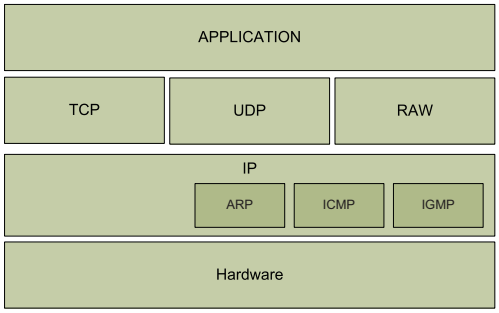

Figure 4.8: *Network Architecture*

4.1.8.1 TCP/IP Stack

Several different TCP/IP stack implementations are supported by eCosCentric. The FreeBSD and OpenBSD stacks have been ported to eCosCentric and are included in the released eCosCentric source repository. There is also a port of the lwIP (lightweight IP) stack for eCosCentric that is not included in the eCosCentric source repository but may be downloaded and installed as an optional eCosCentric package.

The FreeBSD stack is currently the recommended IP stack for eCosCentric; the OpenBSD stack is included more for historical reasons. Both of these stacks only support Ethernet as the underlying hardware - support for PPP or SLIP is not included. An implementation of

PPP is available commercially from eCosCentricCentric (http://www.eCosCentriccentric.com/) but is not included in the freely available code.

The lwIP stack is designed for use in embedded systems and has a significantly smaller memory footprint than the BSD stacks. It also supports PPP and SLIP connections. The lwIP stack has a non-standard implementation of the socket and as a result standard system calls cannot be used on lwIP sockets. lwIP provides its own implementation of the socket system calls to be used on its sockets, however calls such as select that require file descriptors of different types (e.g. a serial port and a socket) will not work.

4.1.8.2 Supported Protocol

The networking protocols supported in the FreeBSD implementation include:

- IPv4 - Internet Protocol version 4
- IPv6 - Internet Protocol version 6
- ARP - Address Resolution Protocol
- RARP - Reverse Address Resolution Protocol
- ICMP - Internet Control Message Protocol
- IGMP - Internet Group Management Protocol
- UDP - User Datagram Protocol
- TCP - Transmission Control Protocol
- DHCP - Dynamic Host Configuration Protocol
- BOOTP - Bootstrap Protocol
- TFTP - Trivial File Transfer Protocol
- Multicast Addressing

Using the BSD stacks, eCosCentric also supports:

- DNS (Domain Name System) client - resolve domain names to IP addresses
- HTTP (Hypertext Transfer Protocol) server - allow web-based monitoring and control

- SNMP (Simple Network Management Protocol) agent - control and monitor the device using SNMP
- SNTP (Simple Network Time Protocol) client - set the system time from a network server
- FTP (File Transfer Protocol) client

The lwIP stack only supports basic IPv4 networking with dynamic address allocation using DHCP.

4.1.9 Configuration Tool

One of the obvious advantages of eCosCentric is its configurability. The developer can adapt eCosCentric very detailed to the given needs. The Configuration Tool, provided with the eCosCentric release, eases the selection and configuration of the software components. Each single module of eCosCentric can be selected or deselected using the Configuration Tool. On the one hand it is even possible to choose single lines of code.

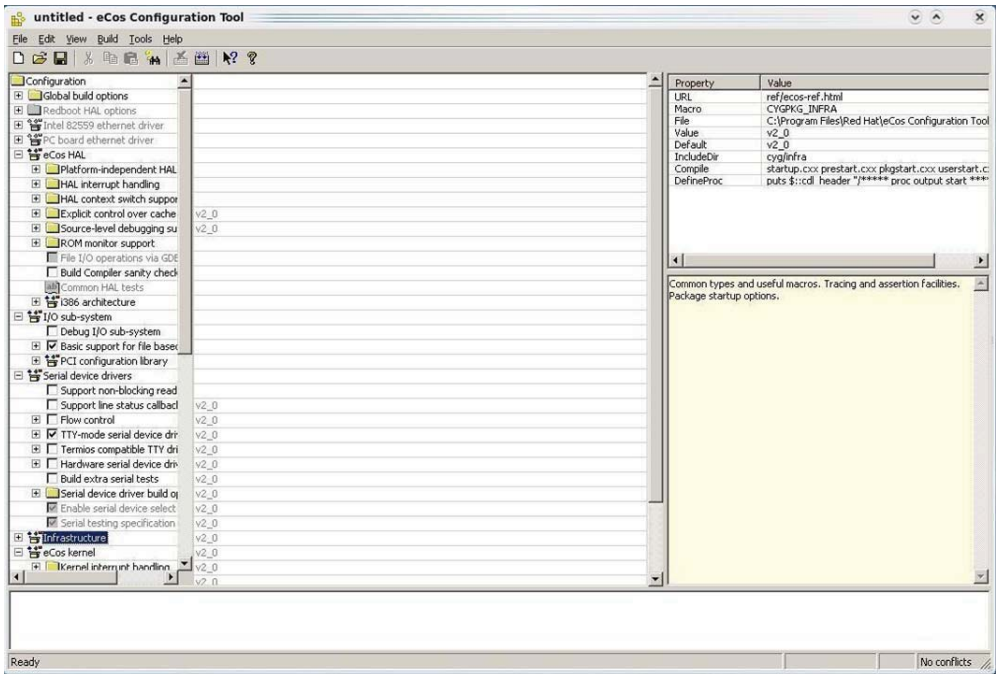

Figure 4.9: *eCosCentric GUI Configuration Tools*

e.g. the TCP/IP stack support can be deselected with only one click. The amount of configuration options increases with each package included to the repository. At the moment

there are over 1000 different options [16]. eCosCentric uses the Configuration Definition Language (CDL) to organize the resulting amount of information about the supported packages. Especially the resulting dependencies of related packages are organized and checked. Since the proper description of the CDL is beyond the scope of this report further information can be found in The eCosCentric Component Writers Guide [9]. If a special package requires the implementation of another package this information is contained in the CDL file of the package. Furthermore the Configuration Tool displays fundamental information for each package and supports the developer to understand the package coherences. Another feature of the Configuration Tool is the option to use predefined templates for various

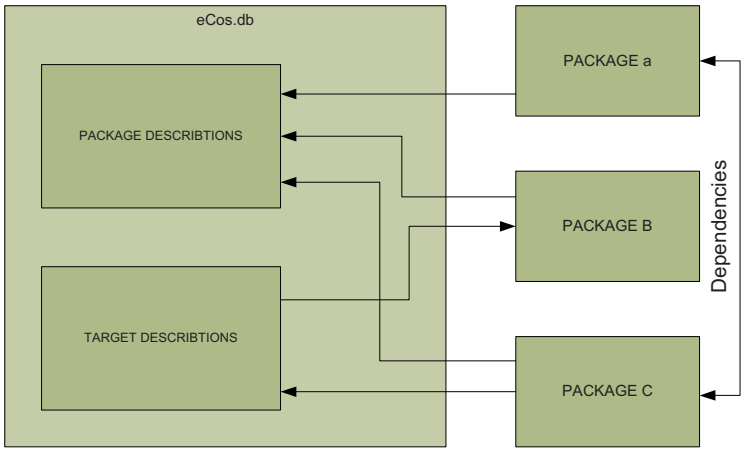

Figure 4.10: *eCosCentric Package Database Structure*

platforms and interfaces. These templates are defined in the eCosCentric.db database which can be found in the eCosCentric CVS repository under /eCos/packages. Each templates describes the target by listing all packages which are included in the configuration of the target. As shown in Figure 4.10 the target descriptions are linked to actual hardware packages. Furthermore the ecos.db lists all packages which are include in the eCosCentric component framework. These packages point to each CDL file which describes the component and its requirements in more detail. An example for an ecos.db entry can be found in the Appendix.

4.1.10 eCosCentric building process

The building process for an eCosCentric application consists of two major steps. At first an eCosCentric configuration is setup including all required components which will be used by the application. The configuration is usually generated using the eCosCentric Configuration

tool and saved as an eCosCentric configuration file (.ecc). By this the Configuration tool generates appropriate files for the build.

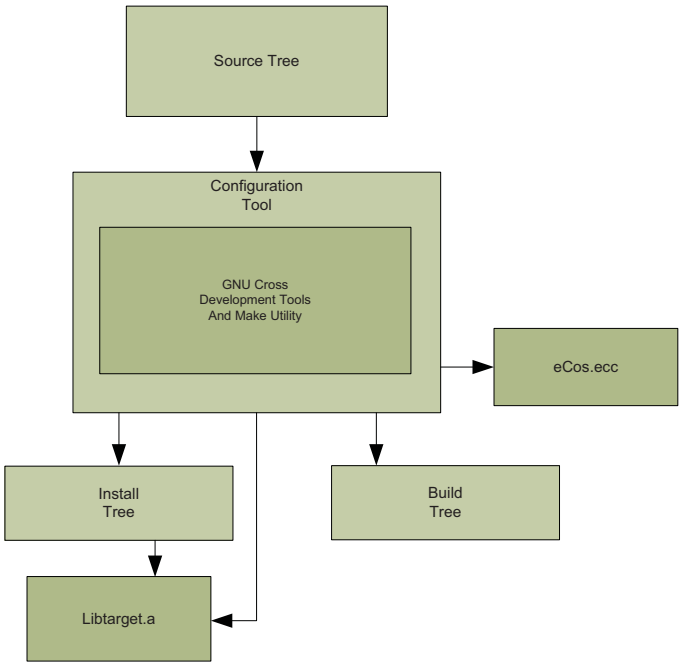

Figure 4.11: *eCosCentric Build Process[27]*

- `filename_install` contains libtarget.a and target.ld. The libtarget.a is the archived eCosCentric Kernel and the target.ld is a linker file specific to the target.
- `filename_built` contains object files and other files specific to the board.
- `filename_mlt` contains memory layout information.

The GNU cross-development tools are used to compile the source code files and produce the final libtarget.a output file. It has to be mentioned that the eCosCentric config tool does not always clean the generated directories properly when a library has to be rebuild. Therefore it is recommended to consider a manual deletion of these files before a rebuild, especially if odd compiler error messages appear [15]. The second major step is to link the specific application with the eCosCentric library. Therefore the application has to be compiled and linked using the matching GNU Cross Compiler and Linker CPU variant. A generic make file can be found in the Appendix. Using the make file the INSTALL_DIR variable must be set to the install tree build by the eCosCentric Config tool. Additionally the name of the application has to be adjusted.

After this the generated ELF file can be uploaded to the target using GNU Debugger (gdb) when a RAM application is designed. The binary image can be loaded on a Flash memory using telnet together with openOCD (see section 5.1.3).

4.1.11 eCosCentric Support

There are six different mailing lists available for the eCosCentric project:

- Development List - Includes discussion about current enhancements being developed, such as new ports and new features.
- Discussion List - Contains support and technical assistance on various topic about the eCosCentric project from developers. Most of the discussions are hold at this place.
- Patches List - Used for submitting eCosCentric patches for approval by the maintainers before they are committed to the repository.
- Announcement List - A low-volume list for significant news about eCosCentric that is also used to announce new eCosCentric releases or major feature enhancements.
- CVS List - A read-only list that gives notifications of changes made to the eCosCentric source code repository.
- CVS Web Pages List - Contains notifications of changes to the eCosCentric web pages that are maintained in the CVS.

These lists and further information can be found at [14]

4.2 Implemtation of Micro-Controller Bootstrap

The bootstrap firmware is located in flash at address 0x00, this is the reset address for the STM32 processor, and therefor the firmware run when the board first boots. The bootstrap software performs the following basic functions:

- Initialises CPU and on-chip peripherals
- Reads boot sector
- Performs memory remap
- Loads program image to RAM

- Sets registers to defined values for program entry
- Runs specified program

4.2.1 Boot Table

The boot table resides at offset 0x4000 in the flash and uses one 8K flash sector. It consists of 16 entries, each 0x100 bytes long. The full boot table can be found in the data sheet [40]

The boot sector is read by the bootstrap software and are used to select one of the boot entries to boot.

4.2.2 Micro_Controller Peripheral Initalisation

The internal CPU peripheral are Initalised by the boot code.

4.2.2.1 PIO

- PIO or peripheral control is set
- PIO direction (input or output) is set
- Interrupts are disabled on all PIO pins

4.2.2.2 Watchdog

- Watchdog timer is disabled

4.2.2.3 USART's

- USART is reset
- Internal clock selected for baud rate generator
- Baud rate is set to 19200 bps
- Mode set to 8 data bits, no parity, 1 stop bit
- Receiver and transmitter timeouts are disabled
- Interrupt generation is disabled

4.2.3 Copy Program Image to SRAM

If the RAM image length in the boot table entry is non-zero, the bootstrap software will copy the image in ROM specified in the boot table entry to the address specified in the internal SRAM before execution begins.

4.2.4 Set Register to Define Values

On entry into the user program stm32 registers are set to contain information about the system. The bootstrap software sets the contents of each of these registers.

Once these registers have been set the CPSR is set and control passes to the program at the address specified in the boot sector.

4.3 Implementation of eCosCentric

Depending on the CPU architecture, the variant and the platform there are three different types of eCosCentric HAL ports. The easiest one is the platform port which requires the implementation of the custom platform attributes. If a port to a special architecture family exist but a special variant of this family is not supported a variant port is required. The most large-scale port is the architecture port which requires a fundamental adaptation of the complete HAL.

eCosCentric was already ported to the ARM Cortex M3 architecture variant hence the HAL has to be ported to the target platform.

4.3.1 Redboot

Once the eCosCentric HAL and at least the serial or the Ethernet driver have been ported to the target RedBoot can be build. The eCosCentric Configuration Tool offers therefore a template which selects all required packages. Furthermore a minimal configuration file is helpful to select the appropriate attributes for the RedBoot application. This file has to be imported by the Configuration Tool and finally the RedBoot image can be build. As a last step the Redboot image can be load to the static memory of the target board using openOCD and telnet. If the driver for the external memory (Micro SD Card) is implemented Redboot can be used to give information about or manipulate this memory.

4.3.2 HAL Port

A platform port is the less complicated variant of an eCosCentric HAL port. Five major modifications and implementations have to be done. The main part of the porting process is the implementation of the hal_platform_setup.h file which contains the major platform initialisation code. For the target platform the stm32fxxx_misc.c file was added with interface initialisation code. Furthermore setup routines for the interrupt controller are usually placed in this file.

The third modification is the customisation of the memory layout. All new port packages have to be described by their associated CDL-file. Finally the new packages have to be added to the eCosCentric component database eCosCentric.db to be available for the eCosCentric Configuration Tool.

Furthermore some common variant and architecture packages need to be slightly modified or supplemented.

4.3.2.1 Structure of the Port

The ARM Cortex M3 port is placed in the CVS repository as a subdirectory of the HAL package. Since the stm32f CPU variant is already supported the new platform port is placed under the /arm/stm32fxxx/stm32f103re subdirectory.

4.3.2.2 Platform Initialisation

hal_platform_setup.h

This setup file includes several macros written in arm assembler. The central macro is the `PLATFORM_SETUP` macro which is called out of vectors.S during the system start-up. The `PLATFORM_SETUP` macro itself calls, depending on the start-up type, a sequence of further initialisation macros, full details can be found in "Embedded Software Development with eCos - Book" from Anthony J. Massa [27].

stm32f103_misc.c

Depending on the configuration of the eCosCentric HAL component in the eCosCentric Configuration Tool the support of the Micro SD card control routine is defined in the stm32f103_misc.c file.

var_io.h

In var_io.h[6] in the code bellow shows the defined addresses of the memory registers when a STM32F Micro-Controller is used.

```
1  ...
2  // internal Memory Controller
3  #if defined(CYGHWR_HAL_ARM_STM32FXXX_STM32F103) || \
4
5      defined(CYGHWR_HAL_ARM_STM32FXXX_STM32F103) || \
6      defined(CYGHWR_HAL_ARM_STM32FXXX_STM32F103)
7
8  #define CYGARC_HAL_STM32Fxxx_REG_BCFG0      0xFFE00000
9  #define CYGARC_HAL_STM32Fxxx_REG_BCFG1      0xFFE00004
10 ...
```

4.3.2.3 Memory Layout

For each specific platform the memory layout has to be adapted. The Windows version of the eCosCentric Configuration Tool contains a graphical editor for the memory setup. This editor generates all required memory layout files. The Linux distribution of this tool does not use such a graphical interface. But since the editor is only useful for less complex layouts that does not influence the work flow. Hence the layout files can be manipulated and adapted by hand pretty easy. Three file describe the memory layout:

- ***mlt-file*** This file is generated by the graphical editor. Since the layout is configured without the graphical editor the mlt-file is unimportant.
- ***ldi-file*** This file is required for the linker scripts.
- ***h-file*** This file contains information about the memory layout (start addresses,length)

As mentioned the file can be found under the /include/pkgconf directory of the port. Each startup-type requires an own set of these files as e.g. for a RAM start-up no information about the ROM mapping are necessary.

Since eCosCentric has been already ported to the STM32Fxxx platform of the ARM Cortex family the existing configuration files are manipulated and adapted to the new board.

[6] located PATH_TO_eCosCentric/eCosCentric/packages/hal/arm/stm32fxxx/var/current/include/

The ldi-file is processed to the target.ld linker script by the C preprocessor. The syntax was adopted from existing ports and manipulated for the specific memory layout.

The other memory layout files have been customised accordingly.

4.3.2.4 CDL-File

Each component of the eCosCentric framework has to be described by a description file (CDL). The CDL file has to offer all required information for the eCosCentric Configuration Tool in order to provide the further process of the module. The code bellow shows the first part of the implemented CDL file for the STM32F103 platform port. The CY GP KG_HAL_ARM _STM32Fxxx_STM32F103 identifier determines the package name which is included in the ecos.db component database. Since the platform port uses the variant port of the STM32FXXX CPU this information has to be given by the "parent" entry. The define_header, include_dir, and compile entries are information for the compiler. Package requirements are listed in the "requires" entry. In this case the processor type has to be STM32F103. If this option is not given the Configuration Tool will produce a conflict and suggest a solution.

```
1 cdl_package CYGPKG_HAL_ARM_STM32FXXX_STM32F103RE {
2     parent      CYGPKG_HAL_ARM_STM32FXXX
3     define_header hal_arm_stm32fxxx_stm32f103.h
4     include_dir     cyg/hal
5     hardware
6     description"The STM32F103RE HAL package provides the support needed to run ecos↩
          on the target platform."
7
8     compile     stm32f103_misc.c
9     requires    { CYGHWR_HAL_ARM_STM32FXXX == "STM32F103RE" }
```

The following further options have been implemented in the CDL file of the target platform.

- Configuration of the Ethernet channel
- Selection of the Start-up type
- Configuration of the serial channels
- Configuration of the CPU speed

4.3.2.5 Modification of the `eCosCentric` database

The eCosCentric Configuration Tool initialises for every start-up its repository using the ecos.db database. In order to support a new port the new package has to be included in this database. Therefore two entries were implemented.

The firstly entry package CY GP KG_HAL_ARM_STM32FXXX_STM32F103RE links the CDL firstly generated for the new port (refer Section 4.3.2.4) to the database. Furthermore a description text of the package shown in the Configuration Tool is integrated.

The second entry target stm32f103 defines the packages included in the template which can be selected with the eCosCentric Config Tool.

4.3.3 Implementation of specific drivers

eCosCentric device drivers in general consists of two components

1. Platform independent driver component; and
2. Platform specific configuration component

This scheme allows drivers to be easily ported among platforms with minimal duplication of source code

Platform independent device drivers are located in the directory
devs/<type>/<manufacturer>/<device>/

- <type>: the device type (e.g. eth, serial)
- <manufacturer>: the device manufacturer (e.g. st, amd, atmel)
- <device>: the product code of the device (e.g. cs8900a, am29xxxxx)

Platform specific device configurations are located in the directory
devs/<type>/<arch>/<platform>/

- <type>: the device type (e.g. eth, serial)
- <arch>: the name of the platform architecture port (e.g. arm)
- <platform>: the name of the platform port (e.g. mu)

As an example, consider the Ethernet driver for the Micro-Controller platform port:

- the platform independent driver is located in the
  `devs/eth/wiznet/w5100`

- the platform specific configuration is located in the directory
  `devs/eth/arm/`

4.3.3.1 Ethernet Controller `Wiznet W5100`

The Wiznet W5100 chip was definitively advantage too implement on this platform. Not only the high level, socket interface of the w5100 chip made the implementation very fast as a TCP stack is not needed the required Micro-Controller specifications are greatly reduced. This project fit in less than 3.8K of flash and less than 1.2K of RAM, thanks to the hardwired TCP/IP stack. The design simplicity can be seen in the schematics which is provided in the Appendix although the system implements a complete Ethernet standalone interface with just two components (the Micro-Controller and the w5100 Ethernet controller).

The following W5100 features were used in this project: three simultaneous sockets, raw MAC mode, TCP mode, UDP mode and SPI interface.

The Wiznet driver was ported to the STM32 and into the operating system driver database architecture (only the SPI interface) and also the Wiznet DNS routines were adapted to use the SPI interface. Several utility routines where extracted from the Wiznet Application Notes and adapted to this project to simplify the implementation.

4.3.3.2 Serial interface Driver

The implementation of the serial interface is rather simple. Since eCosCentric has already been ported to the STM32FXXX family. The four USART's: UART0, UART1, UART2 and UART3 serial hardware ports are integrated in the same way for all controllers of the family's only one modification has to be done. The serial device drivers for the compatible controllers[7] and the serial device driver for the ARM STM32FXXX family[8] have to be included in the target specification of the target board in the eCosCentric.db database.

4.4 Development of Configuration Utility

The application have have been developed to allow communication to W5100 MAC raw mode to capture a special TCP packet directed to the administration port (9000). This way a user can set a new IP without knowing the previous device IP address. This will eliminates the need of implementing a serial interface. The software to change IP can be

[7] CDL package: CYGPKG_IO_SERIAL_GENERIC_UART
[8] CDL package: CYGPKG_IO_SERIAL_ARM_STM32FXXX

developed in Java or in any OS with standard socket interface. In the prototype the interface was developed in Linux as a user command interface as shown in Figure 6.1 .

4.4.1 Board Configuration utility

To use the target platform the user must first setup the IP address, the netmask, the default gateway IP and the DNS server address. The IP setup interface of the system was designed to be set from any computer connected in the same network, even if the previous IP set on the device doesn't belong to the same network. That is, any IP can be set in board from any IP computer. This was implemented using the raw MAC mode of the W5100. To enter the setup mode a soft reset must be done using a jumper cable. The board will start turning on and off the Mode LED every second for about 10 minutes. In this period of time the user must run a setup program from a locally connected computer. The setup software could be developed in any OS that supports TCP, using a Universal language like Java. For my prototype I developed a Unix command line setup program that can be ported easily. The setip command has the following syntax:

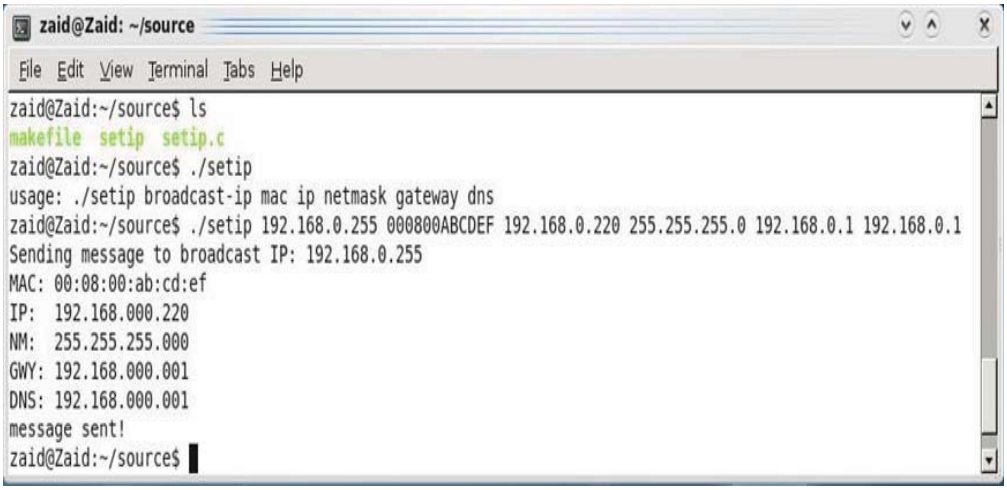

Figure 4.12: *Configuration utility*

The setip command will send a broadcast UDP message to the port 9000, the administration port. The message will be intercepted by application and the contents of the packet will be used to get the IP parameters. The MAC address specified has to match the MAC address set in the device. Once the packet is accepted, the mode LED will turn off to signal that the IP was set.

With the IP set, a telnet to port 9000 will let the user define the rest of the parameters.

The default password is 'password'.

```
if (--argc != 6) {
        fprintf(stderr, "usage: %s broadcast-ip mac ip netmask gateway dns\n", ↩
            progname);
        exit(1);
    }
    broadcast = argv[1];
    mac_s = argv[2];
    ip_s  = argv[3];
    nm_s  = argv[4];
    gwy_s = argv[5];
    dns_s = argv[6];

    /* convert them */

    memset(&adm_message, 0, sizeof(adm_message));
    parse_ip(broadcast, bcast);
    parse_mac(mac_s, adm_message.mac);
    parse_ip(ip_s,   adm_message.ip);
    parse_ip(nm_s,   adm_message.nm);
    parse_ip(gwy_s,  adm_message.gwy);
    parse_ip(dns_s,  adm_message.dns);

    /* dump */
    printf("Sending message to broadcast IP: %s\n", broadcast);
    printf("MAC: ");
    print_mac(adm_message.mac);
    printf("IP:  ");
    print_ip(adm_message.ip);
    printf("NM:  ");
    print_ip(adm_message.nm);
    printf("GWY: ");
    print_ip(adm_message.gwy);
    printf("DNS: ");
    print_ip(adm_message.dns);

    /* build message */
    adm_message.cmd = ADM_SET_IP;

    /* send message */
    send_message();
```

Chapter 5

Development Setup

5.1 Software Environment Setup

This chapter describes the procedure of setting up the environment for getting the eCosCentric source code and target application to work with the custom board

5.1.1 getting eCosCentric Source

The latest version of eCosCentric can be downloaded and updated using the eCosCentric CVS repository. A connection to the CVS server can be established anonymously using the Linux terminal or equal by the command:

```
1 $ mkdir ~/ecos
2 $ cd ~/ecos
3 $ cvs -d :pserver:anoncvs@ecos.sourceware.org:/cvs/ecos login
```

Any password will be accepted. The complete repository can be downloaded using:

```
1 $ cvs -z3 -d :pserver:anoncvs@ecos.sourceware.org:/cvs/ecos co -P ecos
```

For updates of the repository to the latest version use the command

```
1 $ cvs -z3 update -d -P
```

in the base of the repository tree. Once the sources have been checked out the ECOS_REPOSITORY environment variable has to be set to the ecos/packages subdirectory. For example: