Master of Engineering Thesis

Fachhochschule Augsburg
University of Applied Sciences

UNIVERSITY of ULSTER

# Evaluation and Implementation of the RTOS *eCos*

Author: Michael Labus
15512204
famlabus@gmx.de

First Supervisor: Prof Dr H Hoegl

Second Supervisor: Dr C Turner

Submission date: May 19, 2006

**Master of Engineering Thesis**

**University of Applied Sciences, Augsburg, Germany**
Department of Electrical Engineering

**University of Ulster Jordanstown, Newtownabbey, Northern Ireland**
Faculty of Engineering
School of Electrical and Mechanical Engineering

I affirm that the Master of Engineering Thesis is my own work, and that it has never been submitted for examination purposes before. All sources and citations used have been quoted as such, and all utilized tools have been mentioned.

Michael Labus

First edition: 19 May 2006

this page intentionally left blank

## Abstract

The main topic of this project is the evaluation and implementation of the real-time based operating system *eCos*.
*eCos* is a royalty free real time operating system which is highly configurable and has a very small footprint. Within the scope of this thesis, the system was installed on an ARM based microcontroller, using the Olimex LPC-E2294 development board. Therefore the *eCos* hardware abstraction layer was ported to the custom hardware. Drivers were partly designed to provide access to certain hardware interfaces and peripherals including external flash, Ethernet, serial, and CAN. Furthermore an application was implemented to demonstrate the threading capabilities of *eCos* and the control of the serial and GPIO ports.

# Acknowledgements

I would like to thank my first supervisor Prof. Dr. Hubert Hoegl at the University of Applied Sciences, Augsburg, for many ideas and suggestions, for great encouragement, and for giving me the opportunity to conduct this thesis.

I would like to thank my second supervisor Dr. Colin Turner for his support during my stay at the University of Ulster, Newtownabbey.

Furthermore I would like to thank Prof. Dr. Frank Owens at the University of Ulster, Newtownabbey, for his help and accommodation in many organisational matters. Among other favours together with Mr. Gilmore Wilf at the University of Ulster a laboratory was established to offer sufficient working conditions.

Last but not least I would like to thank Timo Bruderek for his advices and hints concerning diverse embedded system problems.

# Contents

# Abbreviations

| | |
|---|---|
| CAN | Controller Area Network |
| CDL | Configuration Definition Language |
| CPU | Central Processing Unit |
| CVS | Concurrent Versions System |
| *eCos* | embedded Configurable operating system |
| ELF | Executable and Linking Format |
| EMC | External Memory Controller |
| GNU | GNU's Not Unix |
| GUI | Graphical User Interface |
| HAL | Hardware Abstraction Layer |
| IDE | Intergarted Development Environment |
| JTAG | Joint Test Action Group |
| LCD | Liquid Crystal Display |
| MCU | Microcontroller |
| RAM | Random Access Memory |
| ROM | Read Only Memory |
| RTOS | Real Time Operating System |

# List of Figures

# List of Tables

# Executive Summary

*eCos* is an open source real-time operating system developed by Redhat and its user community. It, 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]. *eCos* 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. *eCos* 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 build a cheap device with a network interface and software support for the TCP/IP protocol suit. 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 *eCos*. 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 port of *eCos* to a custom hardware platform.
During the conduction of this project several major aims were strived. The following list gives a summerised overview about the aims and objectives.

- Evaluation of the *eCos* operating system regarding architecture, capabilities, and footprint.
- Investigation of third party support like web server and GUIs
- Setup of the hardware and software development system
- Port of *eCos* to the target hardware
- Implementation of required drivers
- Integration of the HAL port and new drivers into the *eCos* component framework
- Application development

The initial step of the project was a brief reading of relevant literature. All required software components for host and target development has been obtained from the *eCos* distributor's internet resources. The hardware, besides the host PC, was provided by Prof. Hoegl, the first supervisor. The basic hardware for this project was

a Debian Sarge operating PC and the Olimex LPC-E2294 evaluation board. The interface between the host and the target was established using the Chameleon POD from Amontec [1]. The required software connection between host and target was established using OpenOCD [25] which is a free JTAG software debugger for ARM7 and ARM9 CPUs. Several guides, all referenced in the bibliography, were considered to port the system to the custom hardware. Furthermore the *eCos* mailing list gave supporting answers to upcoming questions.

The first major task of the thesis was the port of the *eCos* HAL to the custom hardware. This goal has been accomplished successfully. Out of the four aimed interface drivers the serial and the flash memory driver have been implemented and tested successfully. The Ethernet and the CAN driver need some further work to be entirely integrated into the *eCos* system. Thereby the developed routines of the Ethernet driver have to be debugged and tested to provide the demanded functionality. The CAN driver lacks of the proper implementation of the read routines and some accomplishing works to provide a write access to the CAN bus. Nevertheless all drivers are selectable and configurable using the *eCos* Configuration Tool. The conducted HAL port and the drivers have been entirely adapted to the *eCos* component framework.

The implemented applications RedBoot and the LCD control example illustrate the functionality of the *eCos* HAL, the serial driver and the usage of the GPIO ports of the LPC2294 MCU.

Regarding further works the GoAhead web server and the graphical libraries MiniGUI and Nano-X have been investigated and described.

# 1 Introduction

## 1.1 Motivation

For embedded systems some of the tasks of an implemented program might be managing threads, controlling I/O interfaces, and reacting appropriate on exceptions or interrupts. Since these functionalities can be usually offered by a common real-time operating system, the design of own low-layer applications to configure the used hardware and run tasks is not always required. Although self-designed routines can be more efficient to resource management, available operating systems are a reasonable alternative by offering higher portability, a generalized interface to upper applications and the usage of already implemented routines.

Once ported to a specific target own applications can use the provided interfaces and routines to access the hardware functionality. Tasks can run in threads which are automatically administrated and synchronised by the kernel and its scheduler.

The motivation for this master thesis arose from the need to build a cheap device with a network interface and software support for the TCP/IP protocol suit. Since the computing power and memory resources were below the limits to run Embedded Linux, the natural decision was to run the much more economical real-time operating system *eCos*

## 1.2 Aims and Objectives

The major aim and objectives are listed below.

- Evaluation of the *eCos* operating system, regarding architecture, capabilities, and footprint
- Investigation of third party support as webserver or GUIs[1]
- Setup of the hardware and software development system
- Port of *eCos* to the target hardware

[1]Grafical User Interface

- Implementation of required drivers (Ethernet, CAN, Serial, Flash memory)
- Integration of the HAL port and new drivers in the *eCos* component framework
- Application development
- Within the scope of this master thesis only free, open-source software packages and tools have to be used.

Furthermore the thesis has to be available to the public by choosing an open license, e.g. the Creative Commons License.

## 1.3 Structure of this Document

The structure of this document is briefly described in the following. The next chapter describes the requirements for this thesis due to the aims and objectives. The necessary outcome is described in more detail and divided into subtasks. Chapter 3 provides information about the *eCos* system, its architecture, behaviour, and components, down to the *eCos* Kernel features. Additionally crucial information about linking and building of *eCos* and its applications are listed.

Since one of the objectives considers the design of a CAN driver for *eCos*, Chapter 4 deals with the most important fundamentals of this network protocol. In the 5th chapter the required host and target side software and hardware components are described. Since all utilised software has to be open source the setup of these tools needs more attention than for commercial software packages.

Finally Chapter 6 covers the implementation of the *eCos* system to the development platform containing the port itself and the implementation of drivers and applications. The last chapter contains a critical conclusion of the progressed works and an outlook on further work.

# 2 Requirements Specification

The following requirements specification is derived from the main objectives of the Final Year Project mentioned in Section 1.2.

## 2.1 Comprehension of eCos

A detailed comprehension of the *eCos* operating system is a fundamental requirement for the process of the whole implementation. The architecture of *eCos* has to be analysed and the crucial components for the single steps of the *eCos* implementation have to be mentioned. One of the core components of the *eCos* operating system is the Hardware Abstraction Layer (HAL). This layer builds together with the driver components the only hardware dependent part of eCos. Superior 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 realisation process of own tasks has to be described. Applications running on *eCos* 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 *eCos* scheduler have to listed and explained.

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

## 2.2 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 development components have to be mentioned and in addition nontrivial installation and configuration steps have to be explained.

### 2.2.1 Software

As already mentioned *eCos* 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 ought to be a PC running Debian Linux. Concrete installation steps and configuration settings of the host operating system have not to be 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 *eCos* port, the driver development and the design of applications has to described in detail but within the scope of a final report. If the installation of certain software excesses the scope of the thesis further literature has to be offered.

### 2.2.2 Hardware

Unless the host hardware guarantees proper interfaces for the communication to the target hardware no other specifications are necessary for this component.

The target hardware needs to provide a sufficient amount of memory. It has to be investigated which footprints are typical for *eCos*. Third party components and the required memory demands have to be considered and listed.

Since an ARM7TDI-S CPU will be used in future developments this type of processor has to be regarded. It is mandatory that the platform offers the general interfaces as ethernet, SPI, CAN, Serial, and I2C. For debugging purposes a JTAG interface has to be provided.

Debugging has to be conducted using a raven dongle. A suggestion is the Amontec Chameleon POD which is a programmable JTAG interface able to emulate specific parallel to serial interfaces.

For the on going 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.

### 2.2.3 Implementation of eCos

The first major requirement is the port of the *eCos* 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 initialise the target hardware containing CPU, memory, serial interface and GPIO setup. Furthermore the new port has to be fitted into the *eCos* component framework. This makes the new board selectable and configurable using the *eCos* Configuration Tool.

Drivers for the Ethernet and serial communication, the external flash access and CAN messaging control have to be implemented. In particular the CAN driver has to allow access to a bus sending and receiving messages. FullCAN 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.

### 2.2.4 Further miscellaneous Requirements

A possible interface for MCUs is a graphical LCD and/or a touch panel. The thesis has to evaluate which Graphical User Interface libraries are suitable and feasible for *eCos*.

The generated source code and the Final Thesis have to be contributed under appropriate open source licenses.

# 3 *eCos*

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. 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.

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 $\mu$Clinux. It was written by Cygnus Solutions [3] 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 *eCos* belongs to Redhat and the development is due to the huge *eCos* community still in progress. A wide range of ports to general platforms is already available [21]. *eCos*, like other conventional

***Figure 3.1:*** *eCos logo [5]*

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 constructed. 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. *eCos* 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. Thus, the small footprint size, configurability and portability of *eCos* make it the RTOS of choice for this implementation.

## 3.1 eCos Source Tree *Roadmap*

In order to ease the usage of this document Figure 3.2 gives an overview which components can be found in which part of the eCos source tree. Due to clarity reasons only directories are listed which actually contain relevant files.

## 3.2 The *eCos* Architecture

*eCos* 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 3.3 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 *eCos* and a common PC operation system. *eCos* 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.

## 3.3 Hardware Abstraction Layer

The HAL is a key component for the portability of the *eCos* system. Regardless to the specific hardware the higher software layers can use the HAL interfaces to access the

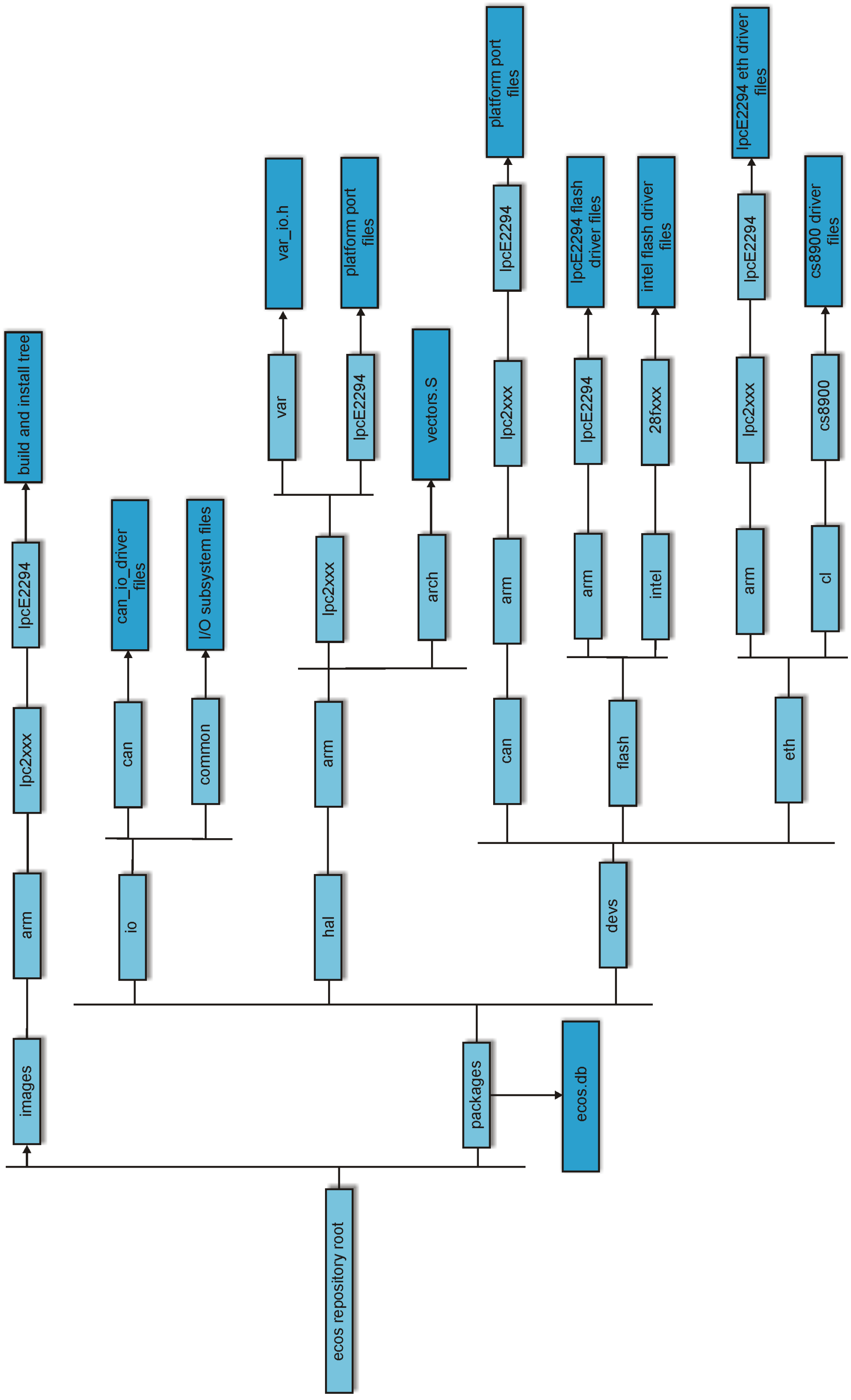


*Figure 3.2:* eCos Roadmap

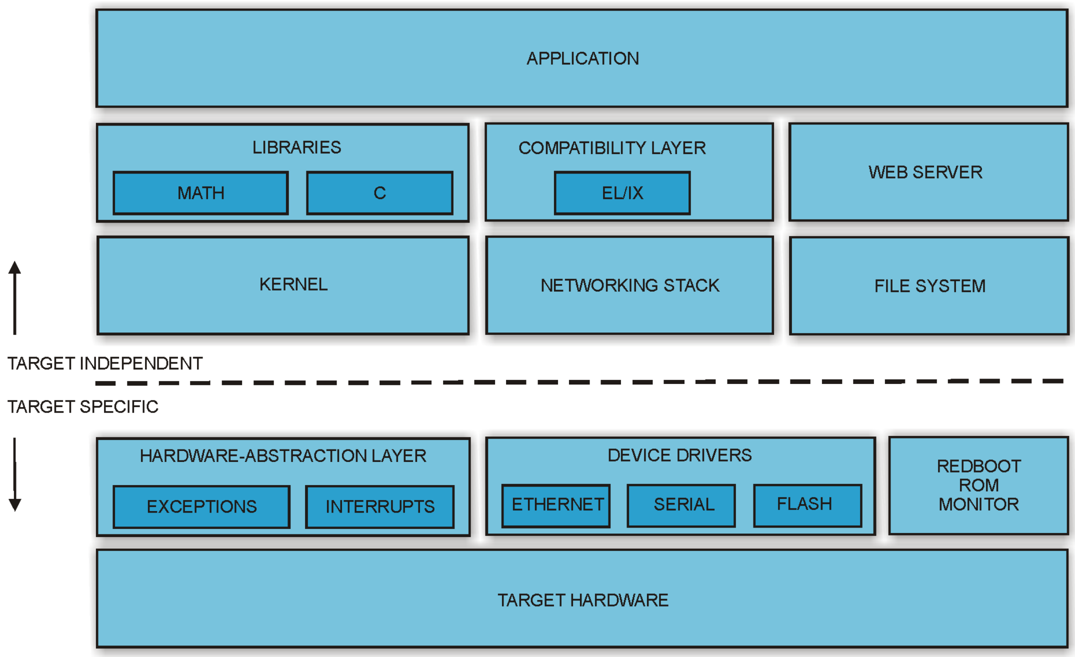


***Figure 3.3:** The overall system architecture of eCos*

full functionality of the platform. The encapsulation allows portability of all others infrastructure components and eases the port process.

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 ARM7 family. For each family the sub-module includes, among others, the code for CPU start-up, interrupt delivery and context switching.

A variant is a special processor within a family and is contained by the variant sub-module, e.g. the ARM7TDMI-S variant. The variant sub-module activates CPU features like caches, MMU or the floating point unit.

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. Listing 3.1 shows an example 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.

```
#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
```

***Listing 3.1:*** *Architecture dependent source code*

### 3.3.1 HAL Start-up

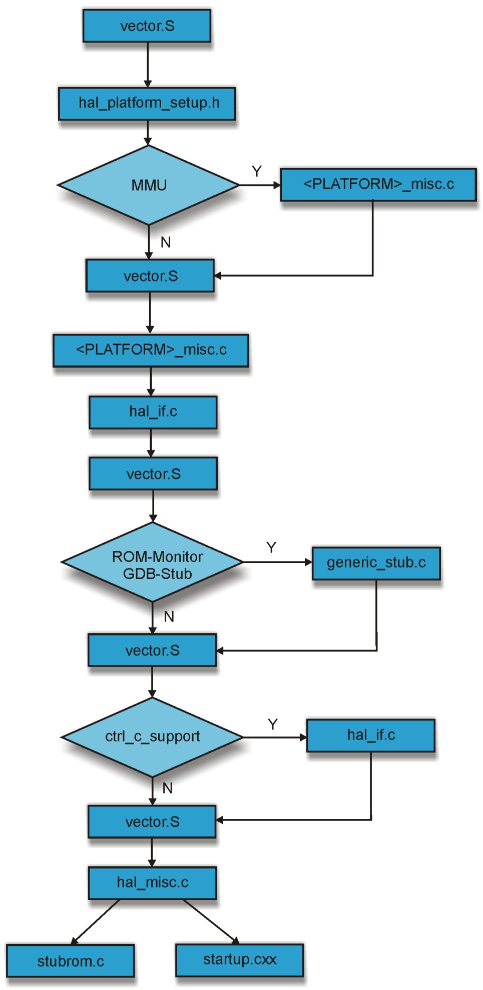


***Figure 3.4:*** *eCos startup procedure*

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 CPUs of an architecture use the same vectors.S file. The start-up process is chronologically shown in Figure 3.4 and described as following[1]:

1. vectors.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_SETUP1.

2. hal_platform_setup.h:
   General hardware initialisation (refer to Section 6.1.2)

3. <PLATFORM>_misc.c:
   This module includes a function, if a MMU is present, which setups the MMU to the final memory layout.

4. vector.S:
   In the next step CPU depending code is executed. The routines depend on the type of the startup respectively whether a ROM-Monitor or an application is started.

5. <PLATFORM>_misc.c:
   In this file the hardware specific initialisation functions 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. vectors.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.c:
    hal_ctrlc_isr_init activates an interrupt handler which stops the application received a break signal.

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

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 breakpoint function on each pass.

If an application is built the further start-up process can be seen in section 3.5.1. Further information about the *eCos* startup on ARM devices can be found in "*eCos* Portierung unter besonderer Beruecksichtigung der ARM-Architektur" from Andreas Buergel [14].

## 3.4 The Redboot ROM Monitor

Redboot is an acronym for "Red Hat Embedded Debug and Bootstrap" [21]. 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. Some of the features provided by *eCos* are:

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

Redboot consists of a slimed version of *eCos* without Kernel and without any applications. However it includes the HAL and is therefore portable to all platforms which are supported by eCos. Figure 3.5 shows the block diagram architecture of some of the features included with the RedBoot ROM monitor. The interaction between Redboot and the *eCos* application varies depending on the configuration option settings in both images.

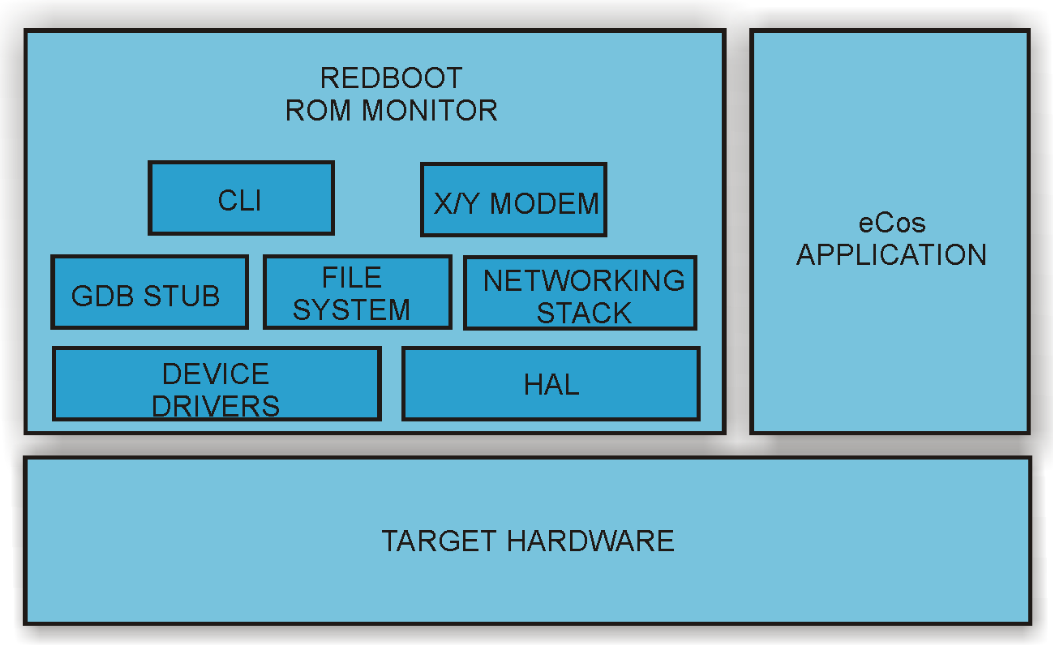


***Figure 3.5:*** *Redboot ROM monitor architecture*

## 3.5 Kernel

The main feature of any RTOS is the Kernel. The *eCos* Kernel provides, among other properties, selectable scheduling policies, mechanisms for thread synchronisation, interrupt and exception handling, counters and clocks. These standard functionalities of the *eCos* 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 *eCos* 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 *eCos* Kernel provides assertions that can be enabled or disabled within the *eCos* package. Enabled assertions during debugging allow the performance of certain error checking and ease the development process.

In the following the sub-sections 3.5.3 and 3.5.4 describe the interrupt and exception handling of eCos. Corresponding systems calls for the interrupt and exception

[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

management will not be listed. All required information about these can be found in the *eCos* Reference [16] and in Anthony Massa's *eCos* book [21].

### 3.5.1 Kernel Startup

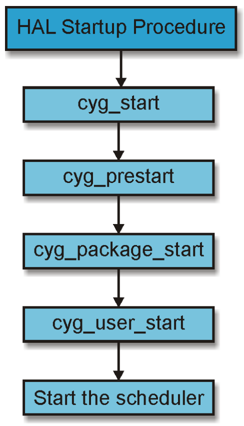


***Figure 3.6:*** *Kernel startup procedure*

After all hardware initialisation is complete, the Kernel start-up procedure is invoked 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 adapted 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 $\mu$TRON or ISO C library before the cyg_user_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. The scheduler is started when the $cyg\_user\_start$ functions returns.

### 3.5.2 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 nonzero. 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. *eCos* 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.

### 3.5.3 Interrupt Handling

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 *eCos* employs a two step interrupt handling scheme. 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.

### 3.5.4 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.
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. *eCos* 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 defined 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 *eCos* 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 3.7). 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 eCos. This method allows the application to take over all or some control over the exception handling. After an exception occurs the processor directly vectors to an application

[5] *hal_misc.c* under the HAL arch subdirectory

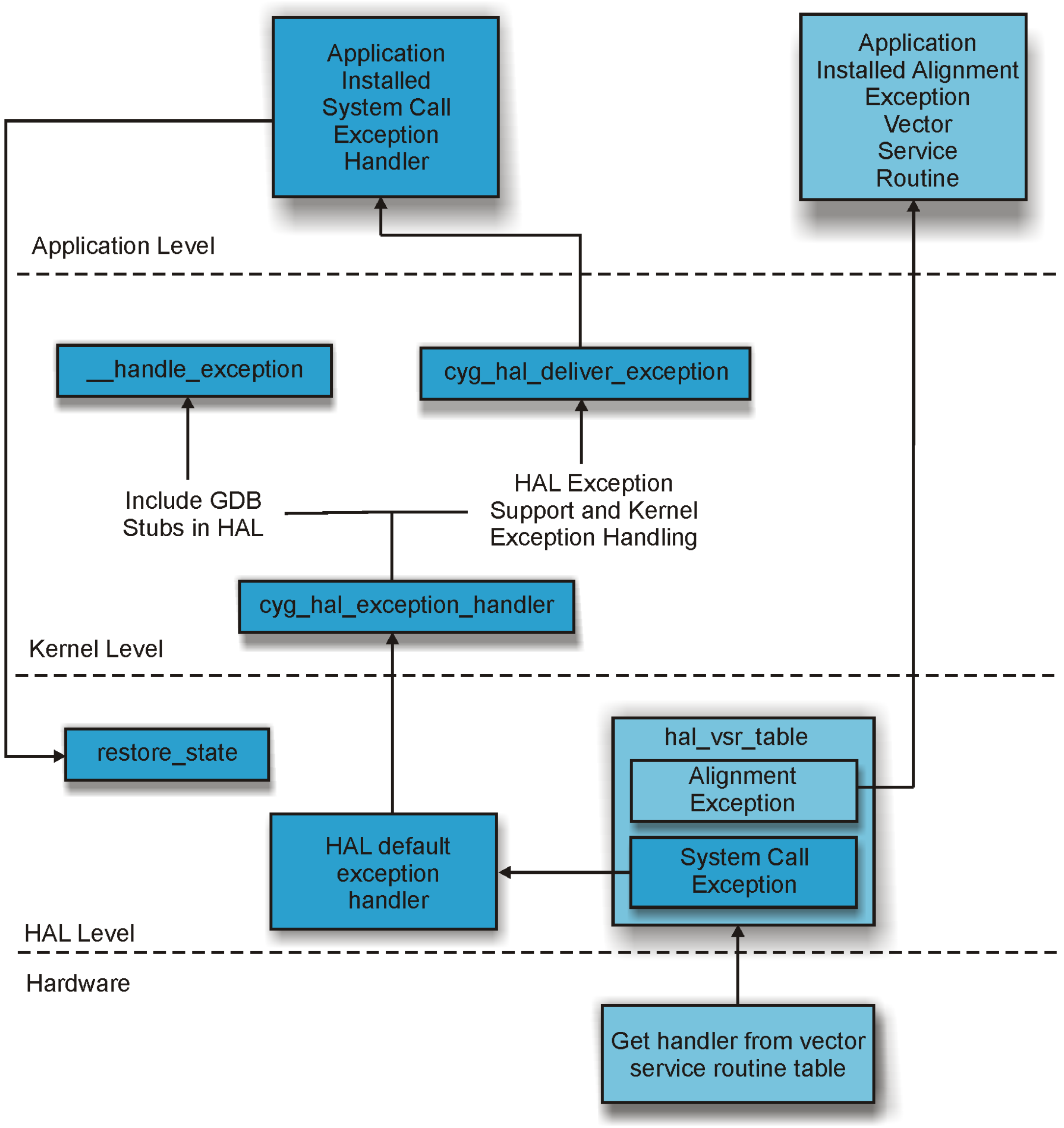

Application Installed System Call Exception Handler
Application Installed Alignment Exception Vector Service Routine
Application Level
__handle_exception
cyg_hal_deliver_exception
Include GDB Stubs in HAL
HAL Exception Support and Kernel Exception Handling
cyg_hal_exception_handler
Kernel Level
restore_state
hal_vsr_table
Alignment Exception
System Call Exception
HAL default exception handler
HAL Level
Hardware
Get handler from vector service routine table

*Figure 3.7: eCos exception handling execution flow*

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

## 3.6 Thread Synchronisation

*eCos* provides several mechanisms to coordinate the access to shared resources and for synchronisation of threads. Corresponding systems calls for the interrupt and exception management will not be listed. All required information about these can be found in the *eCos* Reference [16] and in Anthony Massa's *eCos* book [21].

### 3.6.1 Mutexes

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.

### 3.6.2 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.

### 3.6.3 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.

### 3.6.4 Spinlocks

Spinlocks are flags used to lock a piece of code therewith 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.[21]

### 3.6.5 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.

### 3.6.6 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.

## 3.7 I/O Control System

The *eCos* 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 3.8 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.

### 3.7.1 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 3.1. More information, especially about the function

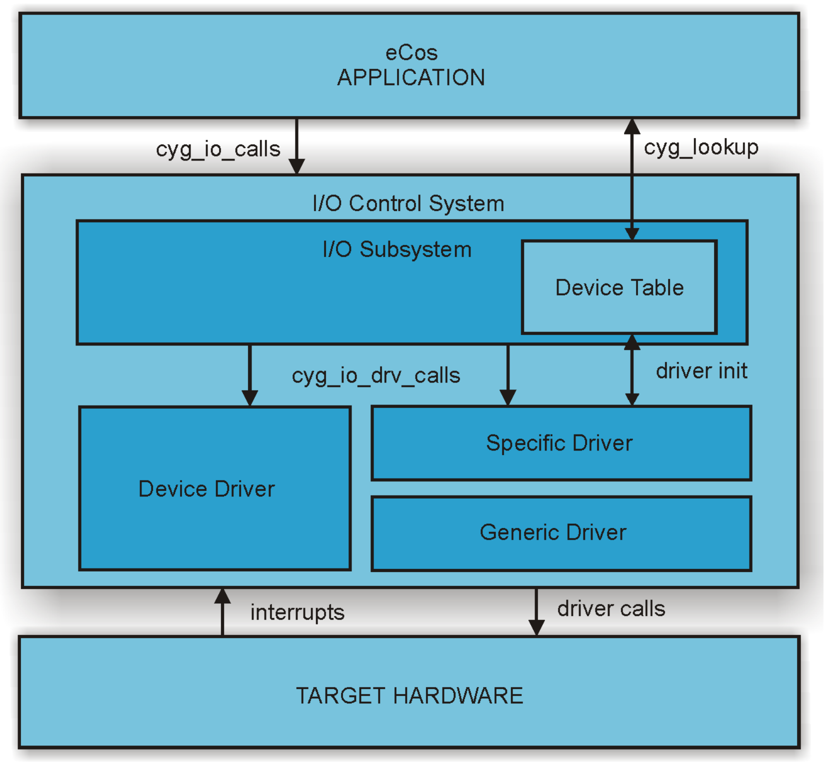


***Figure 3.8:*** *The eCos I/O Subsystem Architecture*

parameters, can be found in the *eCos* Reference [16].

### 3.7.2 Device Drivers

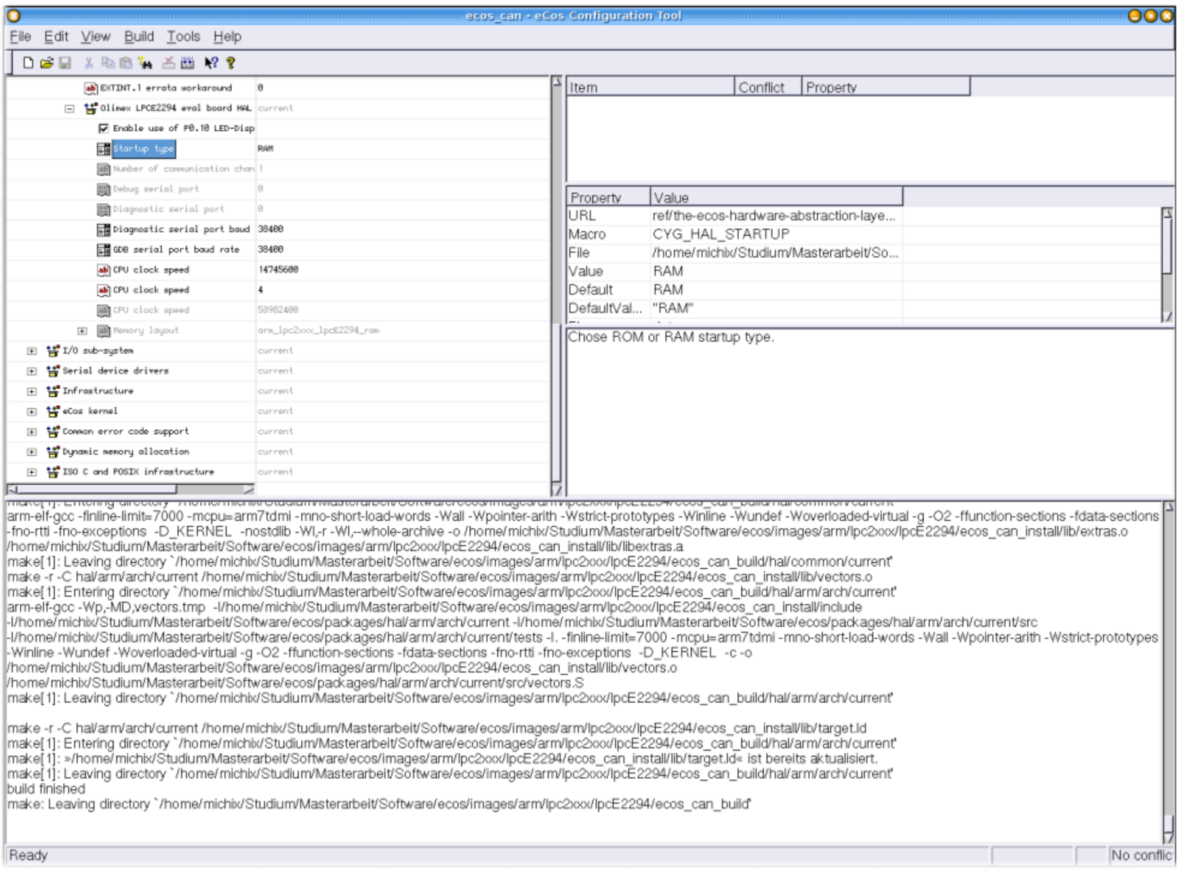
A device driver is the entity which provides the implementation of the I/O function to a specific piece of hardware. It also covers the control of the interrupt handling. *eCos* supports, but does not require, a layered device driver architecture to enlarge portability. Often a generic reusable driver 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.

## 3.8 Configuration Tool

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

| I/O function | Description |
|---|---|
| $Cyg\_ErrNo\ cyg\_io\_lookup(...)$ | 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 3.1:*** *I/O API calls*



| | |
|---|---|
| EXTINT.1 errata workaround | 0 |
| Olimex LPCE2294 eval board HAL | current |
| ☑ Enable use of P0.10 LED-Disp | |
| Startup type | RAM |
| Number of communication chan | 1 |
| Debug serial port | 0 |
| Diagnostic serial port | 0 |
| Diagnostic serial port baud | 38400 |
| GDB serial port baud rate | 38400 |
| CPU clock speed | 14745600 |
| CPU clock speed | 4 |
| CPU clock speed | 58982400 |
| Memory layout | arm_lpc2xxx_lpcE2294_ram |
| I/O sub-system | current |
| Serial device drivers | current |
| Infrastructure | current |
| eCos kernel | current |
| Common error code support | current |
| Dynamic memory allocation | current |
| ISO C and POSIX infrastructure | current |

| Item | Conflict | Property |
|---|---|---|

| Property | Value |
|---|---|
| URL | ref/the-ecos-hardware-abstraction-laye... |
| Macro | CYG_HAL_STARTUP |
| File | /home/michix/Studium/Masterarbeit/So... |
| Value | RAM |
| Default | RAM |
| DefaultVal... | "RAM" |

Chose ROM or RAM startup type.

```
arm-elf-gcc -finline-limit=7000 -mcpu=arm7tdmi -mno-short-load-words -Wall -Wpointer-arith -Wstrict-prototypes -Winline -Wundef -Woverloaded-virtual -g -O2 -ffunction-sections -fdata-sections
-fno-rtti -fno-exceptions  -D_KERNEL  -nostdlib -Wl,-r -Wl,--whole-archive -o /home/michix/Studium/Masterarbeit/Software/ecos/images/arm/lpc2xxx/lpcE2294/ecos_can_install/lib/extras.o
/home/michix/Studium/Masterarbeit/Software/ecos/images/arm/lpc2xxx/lpcE2294/ecos_can_install/lib/libextras.a
make[1]: Leaving directory `/home/michix/Studium/Masterarbeit/Software/ecos/images/arm/lpc2xxx/lpcE2294/ecos_can_build/hal/common/current'
make -r -C hal/arm/arch/current /home/michix/Studium/Masterarbeit/Software/ecos/images/arm/lpc2xxx/lpcE2294/ecos_can_install/lib/vectors.o
make[1]: Entering directory `/home/michix/Studium/Masterarbeit/Software/ecos/images/arm/lpc2xxx/lpcE2294/ecos_can_build/hal/arm/arch/current'
arm-elf-gcc -Wp,-MD,vectors.tmp  -I/home/michix/Studium/Masterarbeit/Software/ecos/images/arm/lpc2xxx/lpcE2294/ecos_can_install/include
-I/home/michix/Studium/Masterarbeit/Software/ecos/packages/hal/arm/arch/current -I/home/michix/Studium/Masterarbeit/Software/ecos/packages/hal/arm/arch/current/src
-I/home/michix/Studium/Masterarbeit/Software/ecos/packages/hal/arm/arch/current/tests -I. -finline-limit=7000 -mcpu=arm7tdmi -mno-short-load-words -Wall -Wpointer-arith -Wstrict-prototypes
-Winline -Wundef -Woverloaded-virtual -g -O2 -ffunction-sections -fdata-sections -fno-rtti -fno-exceptions  -D_KERNEL  -c -o
/home/michix/Studium/Masterarbeit/Software/ecos/images/arm/lpc2xxx/lpcE2294/ecos_can_install/lib/vectors.o
/home/michix/Studium/Masterarbeit/Software/ecos/packages/hal/arm/arch/current/src/vectors.S
make[1]: Leaving directory `/home/michix/Studium/Masterarbeit/Software/ecos/images/arm/lpc2xxx/lpcE2294/ecos_can_build/hal/arm/arch/current'

make -r -C hal/arm/arch/current /home/michix/Studium/Masterarbeit/Software/ecos/images/arm/lpc2xxx/lpcE2294/ecos_can_install/lib/target.ld
make[1]: Entering directory `/home/michix/Studium/Masterarbeit/Software/ecos/images/arm/lpc2xxx/lpcE2294/ecos_can_build/hal/arm/arch/current'
make[1]: »/home/michix/Studium/Masterarbeit/Software/ecos/images/arm/lpc2xxx/lpcE2294/ecos_can_install/lib/target.ld« ist bereits aktualisiert.
make[1]: Leaving directory `/home/michix/Studium/Masterarbeit/Software/ecos/images/arm/lpc2xxx/lpcE2294/ecos_can_build/hal/arm/arch/current'
build finished
make: Leaving directory `/home/michix/Studium/Masterarbeit/Software/ecos/images/arm/lpc2xxx/lpcE2294/ecos_can_build'
```

***Figure 3.9:*** *eCos Configtool Interface*

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 [21]. *eCos* 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 eCos Component Writer's Guide* [10].

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

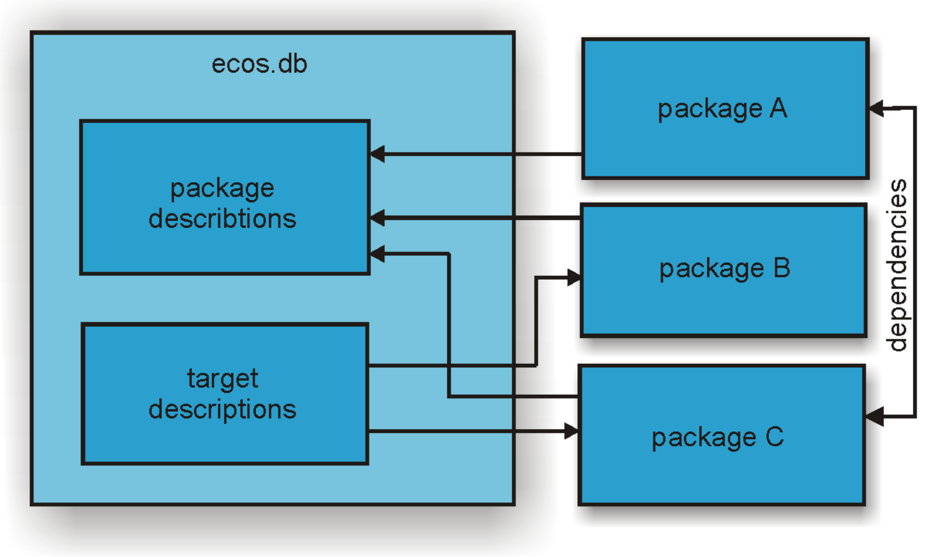


***Figure 3.10:*** *eCos Package Database Structure*

the option to use predefined templates for various platforms and interfaces. These templates are defined in the ecos.db database which can be found in the *eCos* 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 3.10 the target descriptions are linked to actual hardware packages. Furthermore the ecos.db lists all packages which are include in the *eCos* 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 Section 6.

## 3.9 *eCos* building process

The building process for an *eCos* application consists of two major steps. At first an *eCos* configuration is setup including all required components which will be used by the application. The configuration is usually generated using the *eCos* Configuration tool and saved as an *eCos* configuration file (.ecc). By this the Configuration tool generates appropriate files for the build.

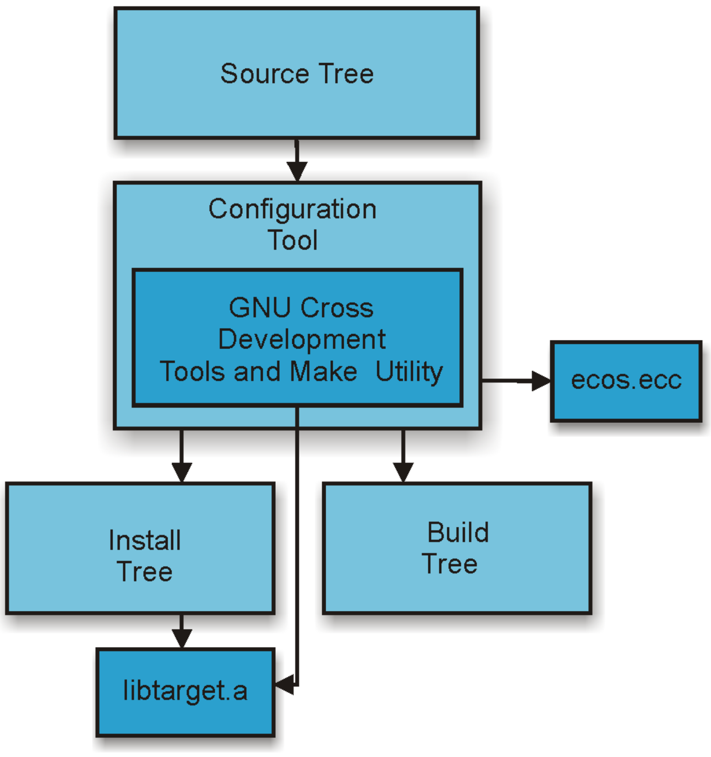


*Figure 3.11:* *eCos* Build Process

- ***filename*_install** contains libtarget.a and target.ld. The libtarget.a is the archived *eCos* 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 *eCos* Configtool 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.
The second major step is to link the specific application with the *eCos* 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 A. Using the make file the INSTALL_DIR variable must be set to the install tree build by the *eCos* Configtool. 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 5.2.1).

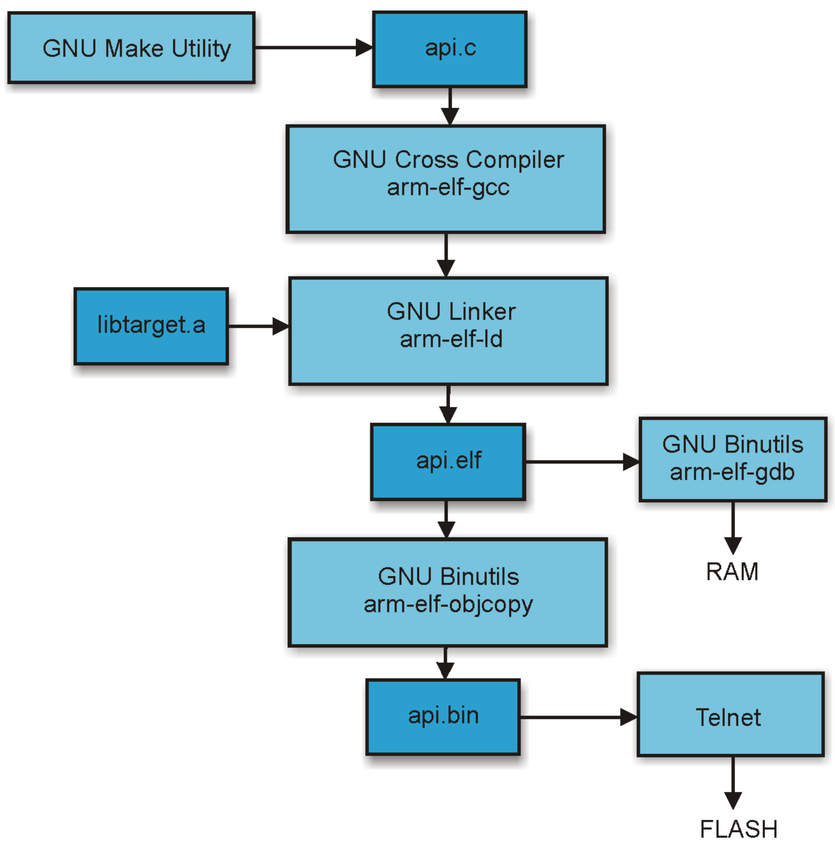


*Figure 3.12: Linking an Application*

## 3.10 Third Party Support

Additionally to the generic *eCos* components third party distributors have ported their software packages to *eCos*. Three of these are the GoAhead Webserver from GoAhead Software and the open source GUI libraries - MiniGui and Nano-X.

### 3.10.1 GoAhead Webserver

The GoAhead Webserver is specifically designed for the use in embedded systems. It is highly portable, has a small footprint and is, of course, open-source. All common internet technologies are supported including ASP, CGI, HTTP 1.0, SSL 3.0, and DAA.

To run the GoAhead Webserver a TCP/IP stack, an event timer, and approximately 60KB of RAM are required. Since the GoAhead Webserver has been already ported to *eCos* the main effort is to supply the requirements of the web server [20].

### 3.10.2 Grafical User Interface

Since no grafical interface has been realized in this project only a brief summary about GUI libraries for embedded systems is given. Currently only two GUI libraries have been ported to eCos. The MiniGUI library from Feynman Software [8] and the Nano-X library formerly known as Microwindows [9], but except of the name the same system. A direct comparison of MinGUI and Nano-X can be found in Appendix B

### MiniGUI

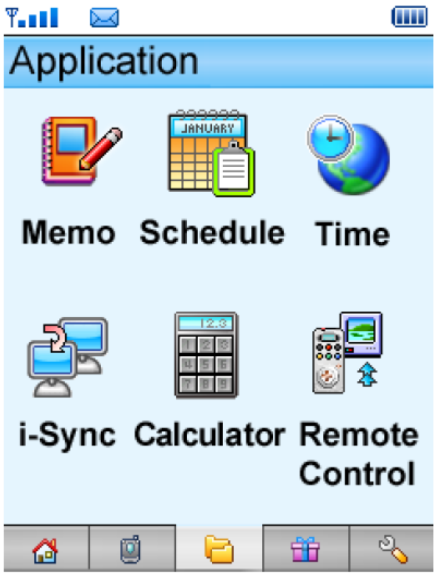


*Figure 3.13: MiniGui example application [13]*

MiniGUI is a free software project, led by Beijing Feynman Software Co., Ltd.. It aims to provide a fast, stable, lightweight, and cross-platform Graphics User Interface support system, which is especially fit for real-time embedded systems based-on Linux/uClinux, eCos, and uC/OS-II. The minimal system resources needed by MiniGUI itself are 700KB of static memory and 1MB of dynamic memory. The MiniGUI datasheet V2.0 [11] specifies that MiniGUI ported to *eCos* requires 2 MB of Flash and 2 MB of RAM when used with applications. For a recommended platform configuration the values have to be doubled.

MiniGUI is capable of running on a system with 30 MHz CPU and 4MB RAM, which can not be reached by Nano-X or Qt/Embedded according to Feynman Software [8]. The relationship between MiniGUI and *eCos* or any other RTOS can be seen from FIGURE 3.14. MiniGUI and the ANSI C library provide functions to a superior application. Similar to the HAL of *eCos* the Portable layer of miniGUI encapsulates the graphical library from specific hardware and operating systems. Hence the running application need not to take care of the output and input devices.[12]

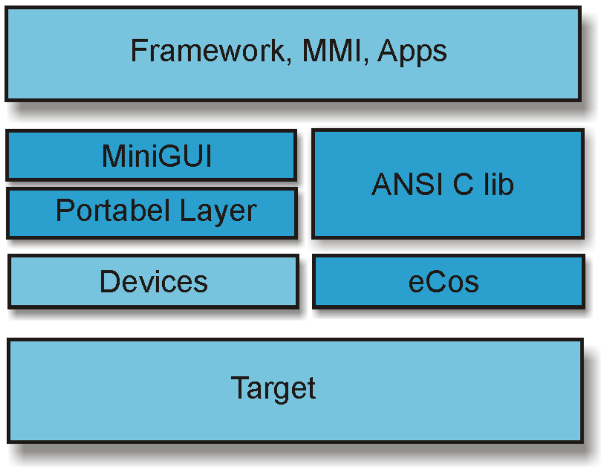


***Figure 3.14:*** *MiniGui archictecture [13]*

### Microwindows/Nano-X

Nano-X is an open source project developed by Century Software [2]. It is aimed at bringing the features of modern graphical windowing environments to smaller devices. Nano-X is a GUI system based on a three layer architecture. The lowest layer handles the access to graphics output, keyboard and touch screen. The intermediate layer provides an common interface to the hardware close layer. The top layer provides APIs compatible with X Window and Win32 subset.

Designed as a replacement for the X Window System, Microwindows provides similar functionality using much less RAM and file storage space: from 100K to 600K [7]. On 16 bit systems, the entire system, including screen, mouse and keyboard drivers runs in less than 64k [9].

## 3.11 *eCos* Support

There are six different mailing lists available for the *eCos* project:

- **Discussion List** - Contains support and technical assistance on various topic about the *eCos* project from developers. Most of the discussions are hold at this place.
- **Patches List** - Used for submitting *eCos* patches for approval by the maintainers before they are committed to the repository.
- **Development List** - Includes discussion about current enhancements being developed, such as new ports and new features.
- **Announcement List** - A low-volume list for significant news about *eCos* that is also used to announce new *eCos* releases or major feature enhancements.

- **CVS Web Pages List** - Contains notifications of changes to the *eCos* web pages that are maintained in the CVS.
- **CVS List** - A read-only list that gives notifications of changes made to the *eCos* source code repository.

These lists and further information can be found at [6]

# 4 CAN-Bus Fundamentals

The Controller Area Network (CAN) is a serial communications protocol which efficiently supports distributed real-time control offering a very high level of security. Its field of application ranges from high speed networks to low cost multiplex wiring.

## 4.1 CAN Topology

The CAN network has a line topology as shown in Figure 4.1. Since all control devices are connected in parallel to a central transmission line this structure offers the advantage that the communication is only intercepted when the line itself fails. The

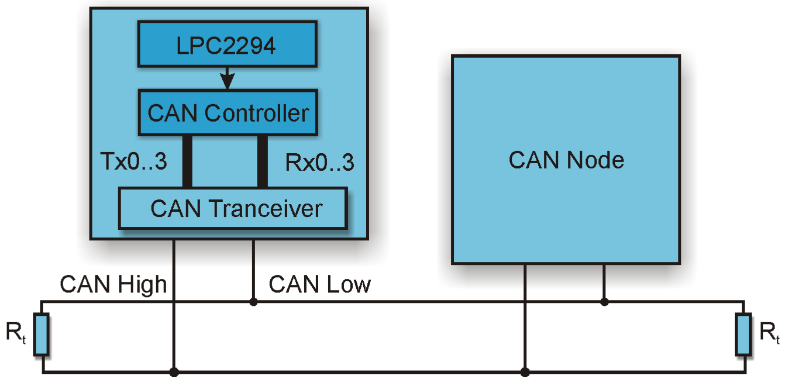


***Figure 4.1:*** *CAN topology with the LPC2294 CAN Controller*

amount of devices connected to the bus is not specified and only restricted by the bus drivers. The maximal data rate of a CAN bus is 1000 Kbit/s.

## 4.2 Object Identifier

The object identifier indicates the content of a CAN messages and not the transmitter or Receiver. E.g. different identifiers can be defined for the temperature, the voltage or the pressure in a control or measurement system. The receiver decides using the identifier whether a particular message is relevant and will be received or not.

The CAN specification defines two different formats of identifiers:

- 11-bit identifier (Base/standard frame format)
- 29-bit identifier (Extended frame format)

Furthermore the identifier is used for prioritisation of messages. A high priority guarantees that the message is send with the shortest possible latency. Due to the specification to avoid arbitrary conflicts two or more transmitters are not allowed to use the same identifier.

## 4.3 CAN-Messages

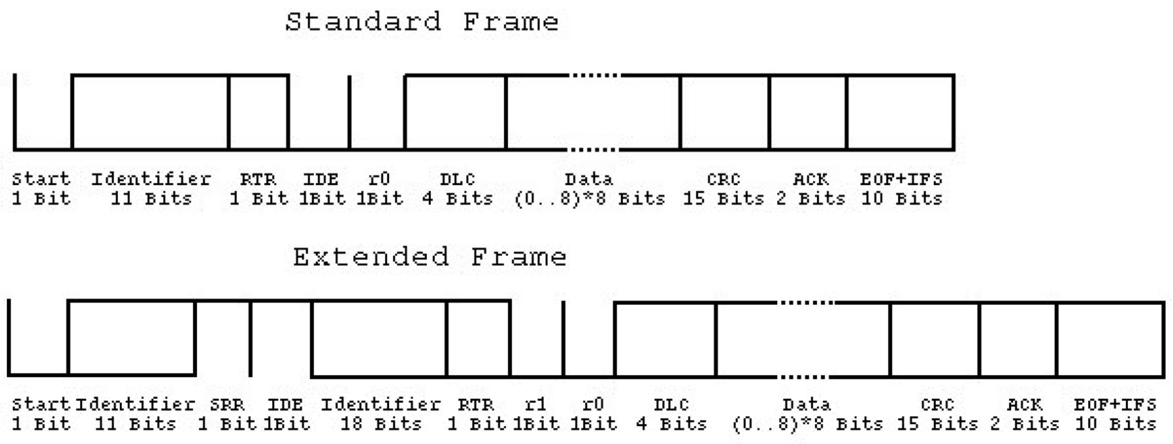


***Figure 4.2:*** *Structure of a CAN message [17]*

Generally the CAN communication is established using four different types of messages:

| | |
|---|---|
| **Data frames** | which transmit up to eight bytes of data. |
| **Remote frames** | which request data frames from another device connected to the bus. |
| **Error frames** | which indicate an error via broadcast to all network participants. |
| **Overload frames** | which provide a timeout between data and remote frames. |

A data frame has the following structure:

| | |
|---|---|
| **Start of Frame** (SOF) | 1 bit, always low, falling edge synchronizes the nodes of the bus |
| **Identifier** | 11 bits for standard frames and 29 bits + 2 bits for extended frames |
| **Remote Transmission Request** (RTR) | for standard frames or Substitute Remote Request (SSR) for extended frames |
| **Control** (CTRL) | 6 bits (2 bit reserved + 4 bit length of data) |
| **Data** | 0-64 bits (8 x 8 bits) |
| **Checksum** (CRC) | 16 bits (15 bit CRC + recessive delimiter bit) |
| **Acknowledge** (ACK) | 2 bits (high) |
| **End of Frame** (EOF) | 7 bits (high) |
| **Intermission Frame Space** (IFS) | 3 bits (high) to separate following messages |

# 5 Development Setup

## 5.1 Hardware Setup

### 5.1.1 Development Host Platform

For the evaluation and implementation of *eCos* a Maxdata Pro 8000X notebook with the following components was used.

| | |
|---|---|
| CPU: | Pentium M, 1500MHz |
| operating System: | Debian GNU/Linux 3.1 "Sarge" and later "Etch" |
| Linux Kernel: | 2.6.8 - 2.6.16 |
| gcc: | Version 4.0.3 (Debian 4.0.3-1) |
| arm-elf GNU toolchain: | Version 3.2.1 |
| openocd: | Version preview060213 |
| Eclipse: | Version 3.2 |
| Kdevelop: | Version 3.2.2 |

Section 5.2 describes the setup of certain listed software packages.

### 5.1.2 Development Target Platform

#### Olimex LPC-E2294 Board

In order to evaluate the *eCos* operating system the Olimex LPCE2294 development platform was chosen. This Platform uses the aimed processor architecture for further developments and offers a generous amount of interfaces. The most important features are given in the following list.

- **MCU**: LPC2294 16/32 bit ARM7TDMI-S with 256K Bytes Program Flash, 16K Bytes RAM, RTC, 4x 10 bit ADC, 2x 32bit TIMERS, 7x CCR, 6x PWM, WDT, 5V tolerant I/O, up to 60MHz operation,
- **Interfaces**: 2x UARTs, 4x CAN, I2C, SPI, JTAG, USB to RS232 converter, RS232, Dallas i-Button

- **External Memory**: 1MB (256Kx32bit) 12 ns 71V416 SRAM; 4MB (512Kx16bit) 70ns TE28F320C3BD70 C3 Intel flash
- Ethernet controller with CS8900A and RJ45 connector
- LCD 16x2 display with backlight
- 2 buttons

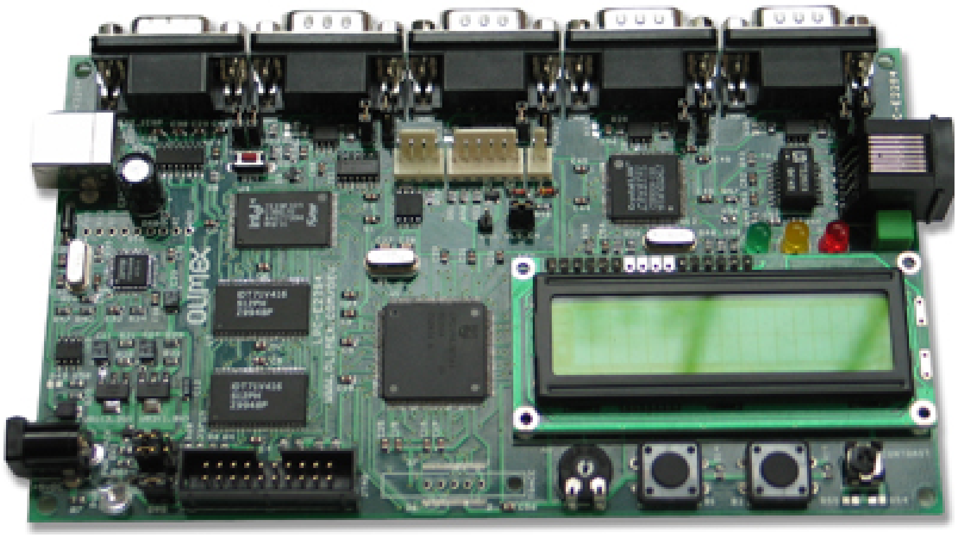


***Figure 5.1:*** *The LPC-E2294 board of Olimex [18]*

### LPC2294

One of the newest members of the Philips LPC family is the LPC2294 chip implementing the ARM7TDMI-S processor with real-time emulation and embedded trace support. The CPU offers a maximum clock of 60MHz and is set to little endian mode by the LPC2294 configuration. Two modes are selectable, the 32-bit ARM mode and the memory saving 16-bit Thumb mode. The LPC2294 chip is provided with 256 kB internal Flash and 16 kB internal static RAM. The detailed memory setup is described detailed below. Depending on the configuration 76 (with external memory, this board) to 112 (single chip) GPIOs are available.

Additional features are various 32-bit timers, 8-channel 10-bit ADC, 4 advanced CAN channels with acceptance filter, PWM channels and up to 9 external interrupt pins [22].

### LCD

One of the most common human interfaces for a MCU is a Liquid Crystal Display (LCD). Furthermore such an interface can be used for testing purpose e.g. to test the

GPIO ports of a LPC2xxx device.

Common LCDs are 16x2 or 20x2 displays i.e. 16 characters per line by 2 lines and 20 characters per line by 2 lines, respectively. Fortunately, a very popular standard exists which allows to communicate with the vast majority of LCDs regardless of their manufacturer. The standard is referred to as HD44780U, which refers to the controller chip which receives data from an external source and communicates directly with the LCD. The standard requires 3 control signals (E, RS, and RW) and 4 or 8 I/O data

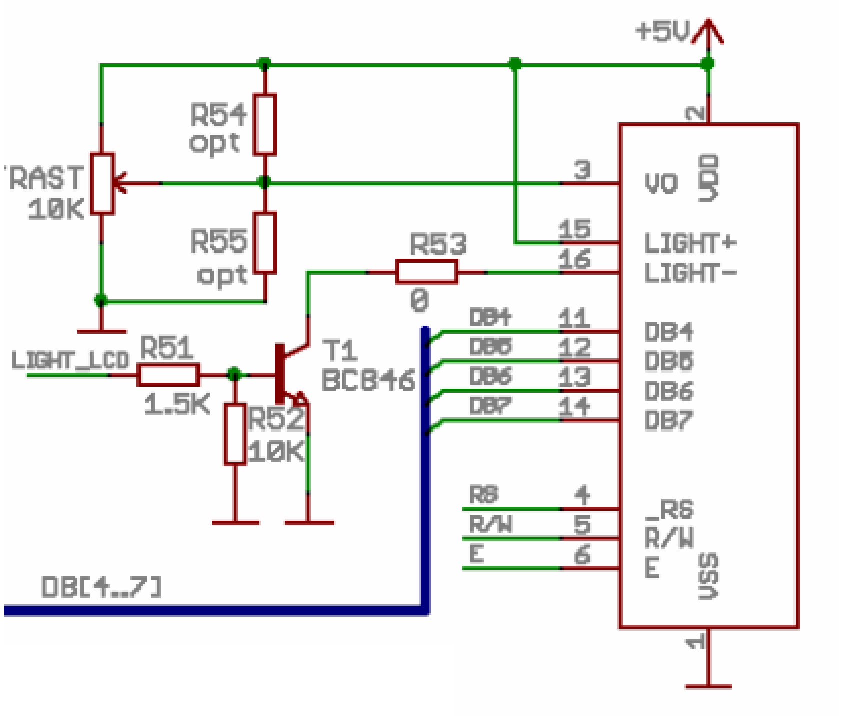


*Figure 5.2: Connection layout of the LCD [18]*

lines. The usage of only upper 4 data lines has the advantage that less GPIO ports of the MCU are used. But the 8 bit messages between the MCU and the LCD controller have to be send in two write accesses. Thereby the higher nibble is transmitted first followed by the lower nibble. Some LCDs support the usage of a backlight. Therefore two additional control lines are used (light+ and light+ , see table 5.1).

The Olimex LPC-E2294 board uses a 16x2 display and the 4 bit data transmission mode. The connections of the LCD can be seen in Figure 5.2. When the board is supplied with power the LCD backlight is, due to the light+ pin directly connected to Vcc, automatically switched on. The light- signal is pulled down by a set LIGHT_LCD (Pin 0.10 of the LPC2294) signal.

The Enable line E is used to tell the LCD that data will be send. Therefore the line is set to low until data was written on the bus or the two other control lines are manipulated. Then the enable signal is set high for a LCD dependent amount of time until it is again brought to low.

| PIN# LCD | PIN Name | Description | LPC2294 |
|---|---|---|---|
| 1 | Vss | GND | |
| 2 | VDD | Power supply | 5V |
| 3 | Vo | contrast adjustment voltage | Potentiometer |
| 4 | RS | Register Select Signal 0 = instruction register (then writing) busy flag and address counter (then reading) 1= data register then writing and reading) | P0.28 |
| 5 | RW | Read/Write 0 = writing 1 = reading | P0.30 |
| 6 | E | Enable | P0.29 |
| 7:10 | DB0:3 | Databit0:3 | not connected |
| 11:14 | DB4:7 | Databit4:7 | P0.4 to P0.7 |
| 15 | light+ | Backlight | 5V |
| 16 | light- | Backlight | P0.10 |

***Table 5.1:*** *LCD Pin Connections*

A low signal on the Register Select (RS) line indicates that data send to the device has to be treated as a command or instruction. When the RS signal is set high the send data consists of text information.

The RW line is the Read/Write control line. When RW is low, the information on the data bus is being written to the LCD. When RW is high the LCD status can be read.

Section 6.4 describes amongst other the implementation of the code required for the control of the LCD.

### CAN Controller and Acceptance Filter

The LPC2294 contains four CAN controllers which provide data rates up to 1 Mbit/s on each bus and a 32-bit register. The Global Acceptance Filter offers FullCAN-style automatic reception for selected Standard identifiers. Additionally it can receive extended 29-bit identifiers [23]. The hardware architecture of the LPC2294 CAN module is shown in Figure 4.1.

### CS8900 Ethernet Controller

The ethernet interface of the Olimex LPC-E2294 board is provided by the Crystal LAN CS8900A Ethernet controller. The hardware setup of the chip is shown in Figure C.1. The controller is connected to operate in the default I/O Mode using the I/O Mode Registers (Table 5.2) for communication with the LPC2294 controller. The interrupt pins (INTRQ0-3) are not used. Since the AEN (Address Enable) and ChipSel pins are connected to ground, read and write operations of the I/O registers are enabled by the /IOR and /IOW signals. These signals reference the /OE and /WE signals of the LPC2294. The CS2 signal qualifies these values as it can be seen in figure 5.3. Since

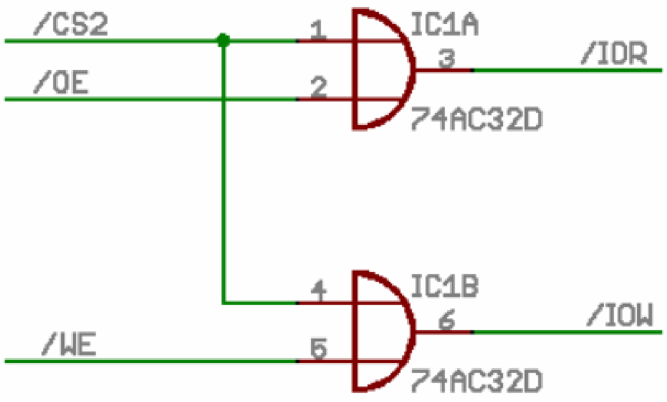


***Figure 5.3:*** *Signal Qualifier*

the SBHE (Set Byte High Enabled) Pin is connected to Vcc only the default 8-bit mode can be used, accidentally all 16 data pins have been connected to the LPC2294 data pins by Olimex.

The address pins SA0 to SA3 of the CS8900 controller are connected to the pins A0 to A3 of the LPC2294 chip. The SA8 and SA9 pins are connected to Vcc which sets the base address to 0x300 (typical IO base address for LAN peripherals [26]) since all other address pins of the CS8900 are connected to ground. I.e. the LPC2294 MCU addresses the CS8900 by writing to its memory space between 82xxxx00 and 82xxxx0F (8-bit addressing).

### Memory Map

In addition to the internal memories the Olimex board offers external Flash and Ram memory. The memory is controlled by the LPC External Memory Controller (EMC), which is is only available for LPC2219 and LPC2294 devices. Altogether four banks of external memory can be addressed. Each memory bank may be 8, 16, or 32 bits wide. The decoding among the four banks uses address bits A[25:24]. The native location

| Offset | Type | Description |
|---|---|---|
| 0000h | Read/Write | Receive/Transmit Data (Port 0) |
| 0002h | Read/Write | Receive/Transmit Data (Port 1) |
| 0004h | Write only | TxCMD (Transmit Command) |
| 0006h | Write only | TxLength (Transmit Length) |
| 0008h | Read-only | Interrup Status Queue |
| 000Ah | Read/Write | PacketPage Pointer |
| 000Ch | Read/Write | PacketPage (Port 0) |
| 000Eh | Read/Write | PacketPage (Port 1) |

***Table 5.2:*** *I/O Mode Mapping [26]*

of the four banks is at the start of the External Memory area identified in Figure 5.4.

The external Ram has a size of 1MB and is connected to the CS1 of the LPC2294. CS0 is used to call the external 4MB flash memory. It can also be used for initial booting under control of the BOOT[1:0] pins. The Crystal LAN Ethernet controller is addressed using the CS2 signal (see Section 5.1.2). The resulting memory map can be found in Figure 5.4.

## 5.2 Software Setup

### 5.2.1 OpenOCD - Debugger

Amontec [1] offers the programmable JTAG Debugger Chameleon POD which can operate as the Amontec Accelerator together with openOCD. OpenOCD is an open source ARM JTAG debugger for targets based on the ARM7 and ARM9 family. OpenOCD was developed by Dominic Rath as his diploma thesis [24] at the University of Applied Sciences Augsburg in Germany.
OpenOCD establishes a telnet port and a GDB remote target port which can be used by the GNU debugger gdb compiled for the ARM architecture. In the direction to the target a JTAG channel is opened. Besides debugging, OpenOCD can control any JTAG-based operation, e.g. programming the FPGA based Chameleon POD by an integrated XSVF player. In addition internal and external FLASH memory programming is supported. OpenOCD was originally developed for Linux but can be used

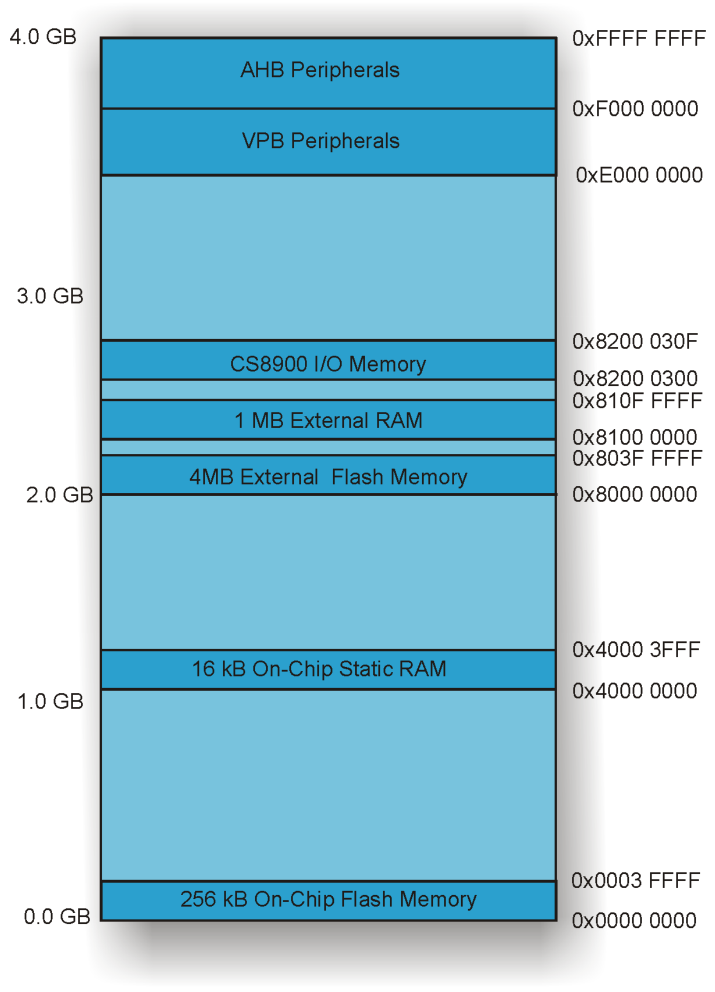


***Figure 5.4:*** *Memory Map of the Olimex LPCE2994 Board*

on Windows systems running Cygwin [4].
In order to enable access to the flash memory of the Olimex LPC-E2295 board the openOCD configuration file needs to be completed with the code listed in listing 5.1.

```
#flash bank lpc2000 base size chipwidth buswidth /
#lpc_variant target# cclk calc_checksum
 flash bank lpc2000 0x0 0x40000 0 0 lpc2000_v1 0 14765 calc_checksum
 flash bank cfi 0x80000000 0x400000 2 2 0 0x40000000 0x4000
```

***Listing 5.1:*** *ARM Wiggler Configuration*

The second line enables the access to the internal 256 MB flash bank of the board. The third line allows the usage of the 4MB external flash memory. lpc2000 and cfi describe the memory, whereas cfi means a Intel compatible memory. In both cases the next two parameters describe the memory starting addresses and its size. Internal memory needs no specified chip- and bus width (0 0). The flash writing algorithm uses the internal 16kb on-chip SRAM located at 0x40000000 to accelerate flash writing [25]. It is important that all parameters are only separated by one space for proper openOCD

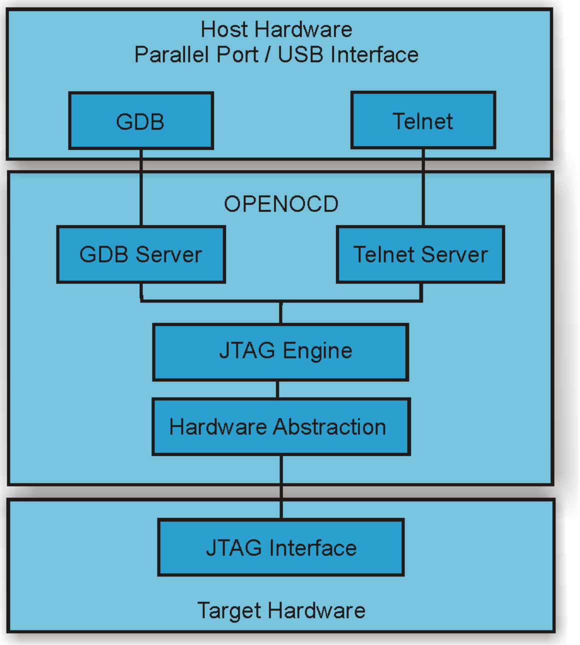


***Figure* 5.5:** *OpenOCD layers and interfaces*

functionality. Further information about openOCD and the command reference can be found in [25].

### 5.2.2 *eCos* CVS Repository

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

```
≫ cvs -d :pserver:anoncvs@ecos.sourceware.org:/cvs/ecos login
```

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

```
≫ 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

```
≫ 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:

```
≫ ECOS_REPOSITORY=/ecoscvs/ecos/packages ;
≫ export ECOS_REPOSITORY
```

(for sh, ksh and bash users)

```
≫ setenv ECOS_REPOSITORY /ecoscvs/ecos/packages
```

(for csh and tcsh users)

### 5.2.3 Eclipse

The Eclipse IDE is a complete Integrated Development Environment platform similar to Microsoft's Visual Studio. Originally developed by IBM, it has been donated to the open source community. By installing the CDT plug-ins, it can be used to edit and debug C/C++ programs. The setup of the Eclipse environment using openOCD and the arm-elf GNU debugger is described by James P. Lynch's tutorial "ARM Cross Development with Eclipse" [19] in detail. For further instructions towards the setup and usage of Eclipse as a debug and development environment consider this document.

### 5.2.4 ARM-ELF GNU Toolchain

The *eCos* 2.0 release features an installation script which simplifies the download and installation of the *eCos* sources, host tools and documentation. Optionally the installation script can download one or more pre-built GNU cross tool chains. The command

```
≫ wget –passive-ftp ftp://ecos.sourceware.org/pub/ecos/ecos-install.tcl
```

downloads the required installation script. The installation tool may then be invoked as follows:

```
≫ sh ecos-install.tcl
```

The used default installation directory is /opt/ecos. Since this install script downloads the official but obsolete *eCos* 2.0 release only the up-to-date CVS sources have been used for the project. Adding the arm-elf executable directory to the system path by calling

```
≫ PATH=$PATH:/opt/ecos/gnutools/arm-elf/bin/ && export PATH
```

eases the further usage of the tool chain.

# 6 Implementation of eCos

Depending on the CPU architecture, the variant and the platform there are three different types of *eCos* 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.

*eCos* was already ported to the lpc2xxx architecture variant hence the HAL has to be ported to the Olimex LPC-E2294 platform.

## 6.1 HAL Port

A platform port is the less complicated variant of an *eCos* HAL port. Five major modifications and implementations have to be fulfilled. 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 Olimex platform the lpcE2294_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-files. Finally the new packages have to be added to the *eCos* component database ecos.db to be available for the *eCos* Configuration Tool.

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

### 6.1.1 Structure of the Port

The LPC-E2294 port is placed in the CVS repository as a subdirectory of the HAL package. Since the lpc2xxx CPU variant is already supported the new platform port is placed under the */arm/lpc2xxx/lpcE2294* subdirectory. The lpcE2294 directory contains the following subfolders:

| | |
|---|---|
| **cdl** | This folder contains the hal_arm_lpc2xxx_lpcE2294.cdl file which describes the port. According to the naming conventions the name of the CDL-file contains the type of the architecture, the variant name, and the described platform. |
| **include** | This folder contains all headers of the port. E.G. the hal_platform_setup.h file for start-up initialisation and the plf_io.h header which contains platform specific register addresses. |
| **include/pkgconf** | All files which determine the memory layout configuration are placed in this sub-folder. |
| **tests** | Simple development tests are placed in this directory, e.g. LCD test and CAN driver test. |
| **misc** | E.g. the minimal configuration files of RedBoot are located in this folders |

### 6.1.2 Platform Initialisation

#### hal_platform_setup.h

This setup file includes several macros written in arm assembler. The central macro is the PLATFORM_SETUP1 macro which is called out of vectors.S[1] during the system start-up. The PLATFORM_SETUP1 macro itself calls, depending on the start-up type, a sequence of further initialisation macros.

- The **_pll_init** macro initialises the programmable PLL of the LPC2294 MCU. Thereby the multiplier value is set to its maximum of 4 and the divider is set to 1. Therefore a resulting CPU frequency of 58 MHz is calculated.
- The **_mem_init macro** copies and maps the interrupt vector table to the internal RAM. Furthermore the CPU memory accelerator module is set fully enabled for highest internal flash performance. The Bank Configuration Registers are configured as well. The values for the BCFG2 register which belongs to the ethernet controller have been adopted from the Olimex example [18]. Since the external RAM is capable to work at 100MHz the fastest possible access time has been chosen.

[1]refer Section 3.3.1

- All values for the pin select registers are set in the **_gpio_init** macro. Besides the direction for the CS pins, the pins for the buttons and the pin for the backlight control are set.
- The **_lcd** macro simply switches the backlight and is called during several times during the start-up (vectors.S).

### lpcE2294_misc.c

Depending on the configuration of the *eCos* HAL component in the *eCos* Configuratio Tool the support of the LCD backlight control routine is defined in the lpcE2294_misc.c file. Furthermore a simple routine is implemented which starts the initialisation of the serial interface.

### var_io.h

In var_io.h[2] one line (refer line 7 in Listing 6.1) has to be added in order to define the addresses of the external memory registers when a LPC2294 MCU is used.

```
1   ...
    //=====================================================
    // External Memory Controller

    #if defined(CYGHWR_HAL_ARM_LPC2XXX_LPC2212) || \
6       defined(CYGHWR_HAL_ARM_LPC2XXX_LPC2214) || \
        defined(CYGHWR_HAL_ARM_LPC2XXX_LPC2294)

    #define CYGARC_HAL_LPC2XXX_REG_BCFG0          0xFFE00000
    #define CYGARC_HAL_LPC2XXX_REG_BCFG1          0xFFE00004
11  ...
```

***Listing 6.1:*** *var_io.h extract*

## 6.1.3 Memory Layout

For each specific platform the memory layout has to be adapted. The Windows version of the *eCos* Configtool contains a graphical editor for the memory setup. This editor generates all required memory layout files. The Linux dirstribution of this tool does not use such a grafical interface. But since the editor is only usefull for less complex layouts that does not influence the workflow notably. Hence the layout files can be maniulated and adapted by hand pretty easy.

[2]located PATH_TO_ECOS/ecos/packages/hal/arm/lpc2xxx/var/current/include/

Three files describe the memory layout:

- **mlt-file** This file is generated by the graphical editor. Since the layout is config 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 files 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 *eCos* has been already ported to the LPC-P2106 platform of Olimex the existing configuration files are manipulated and adapted to the new board. Listing 6.2 shows the code for the ROM start-up version of the ldi-file. Unlike the LPC-P2106 platform an external RAM bank is connected to the MCU. The additional memory is defined as ram0. As the LPC2294 MCU offers only 16KB of internal RAM the support of extra memory is crucial for a proper administration of the system.

```
// \textit{eCos} memory layout

// This is a generated file - do not edit
// external ram (ram) connected to CS1  81000000 - 81ffffff, 256Kx32bit
// external rom connected to CS0        80000000 - 80ffffff, 512Kx16bit

#include <cyg/infra/cyg_type.inc>

MEMORY
{
    ram0   : ORIGIN = 0x40000000, LENGTH = 0x4000
    ram    : ORIGIN = 0x81000000, LENGTH = 0x100000
    rom    : ORIGIN = 0x00000000, LENGTH = 0x40000
}

SECTIONS
{
    SECTIONS_BEGIN
    SECTION_rom_vectors (rom, 0x00000000, LMA_EQ_VMA)
    SECTION_text (rom, ALIGN (0x4), LMA_EQ_VMA)
    SECTION_fini (rom, ALIGN (0x4), LMA_EQ_VMA)
    SECTION_rodata (rom, ALIGN (0x4), LMA_EQ_VMA)
    SECTION_rodata1 (rom, ALIGN (0x4), LMA_EQ_VMA)
    SECTION_fixup (rom, ALIGN (0x4), LMA_EQ_VMA)
    SECTION_gcc_except_table (rom, ALIGN (0x4), LMA_EQ_VMA)
    SECTION_fixed_vectors (ram0, 0x40000400, LMA_EQ_VMA)
    SECTION_data (ram, 0x81000000, FOLLOWING (.gcc_except_table))
    SECTION_bss (ram, ALIGN (0x4), LMA_EQ_VMA)
    CYG_LABEL_DEFN(__heap1) = ALIGN (0x8);
    SECTIONS_END
}
```

***Listing 6.2:*** *ldi-file for ROM startup*

The ldi-file (listing 6.2) 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 costumized accordingly.

### 6.1.4 CDL-File

Each component of the *eCos* framework has to be decribed by a description file (CDL). The CDL file has to offers all required information for the *eCos* Configuration Ttool in order to provide the further process of the module.

Listing 6.3 shows the first part of the implemented CDL file for the LPC-E2294 platform port. The $CYGPKG\_HAL\_ARM\_LPC2XXX\_LPCE2294$ identifier determines the package name which is included in the ecos.db component database. Since the platform port uses the variant port of the LPC2xxx 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 LPC2294. If this option is not given the Configuration Tool will produce a conflict and suggest a solution.

```


cdl_package CYGPKG_HAL_ARM_LPC2XXX_LPCE2294 {
    display       "Olimex LPCE2294 eval board HAL"
    parent        CYGPKG_HAL_ARM_LPC2XXX
    define_header hal_arm_lpc2xxx_lpcE2294.h
    include_dir   cyg/hal
    hardware
    description   "
        The LPCE2294 HAL package provides the support needed to run
        \textit{eCos} on an Olimex LPCE2294 eval board."

    compile       lpcE2294_misc.c

    requires      { CYGHWR_HAL_ARM_LPC2XXX == "LPC2294" }
```

***Listing 6.3:** Port description file*

The following further options have been implemented in the CDL file of the LPC-E2294 Olimex port.

- Selection of the LCD backlight control availability
- Selection of the Start-up type
- Configuration of the serial channel
- Configuartion of the CPU speed
- Global C and LD flags settings
- Rom-Monitor configuration

### 6.1.5 Modification of the *eCos* database

The *eCos* 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 first entry $package\ CYGPKG\_HAL\_ARM\_LPC2XXX\_LPCE2294$ links the CDL file generated for the new port (refer Section 6.1.4) to the database. Furthermore a description text of the package shown in the Configuration Tool is integrated.

The second entry $target\ lpcE2294$ defines the packages included in the template which can be selected with the *eCos* Configtool. The syntax for these entries can be found in the Appendix G, Listing G.1.

## 6.2 Drivers

### 6.2.1 Serial Interface Driver

The implementation of the serial interface is rather simple. Since *eCos* has already been ported to the LPC2xxx family and the UART0 and UART1 serial hardware ports are integrated in the same way for all controllers of the family only one modification has to be done. The serial device drivers for 16x5x compatiple controllers[3] and the serial device driver for the ARM LPC2XXX family [4] have to be included in the target specification of the LPC-E2294 board in the ecos.db database. The two line insertion can be found in Appendix G, Listing G.1.

### 6.2.2 Flash ROM Driver

Compared to the Serial interface driver the effort for the flash memory support is not significantly larger. *eCos* offers a generic support for flash memories which is include in the package CYGPKG_IO_FLASH of the component framework. Hence only an adaption to the given flash memory layout is required. Since the Intel 28Fxxx flash memory series is already contained in the *eCos* repository solely an adaption to the board memory layout has to be made.

Several modifiactions and additions therefore have to be done in the *eCos* repository. First of all a header file has to be genarated which describes the layout of the flash implemention on the Olimex board. The source code can be found extracted in Appendix F, Listing F.1. Additionally this file needs to be described by a corresponding CDL file (refer to Appendix G, Listing G.3). Both files are stored in the devs/flash/arm/lpcE2294 directory of the *eCos* package source tree. The CDL file contains additional information to the related Intel chipset support file.

Furthermore the new support package has to be added to the ecos.db as shown in Appendix G, Listing G.2. Finally the support for the Intel chipset [5] and the board

[3]CDL package: CYGPKG_IO_SERIAL_GENERIC_16X5X
[4]CDL package: CYGPKG_IO_SERIAL_ARM_LPC2XXX
[5]CDL package: CYGPKG_DEVS_FLASH_INTEL_28FXXX

specific support package[6] have to be included in the target specification in the ecos.db data base as shown in Appendix G, Listing G.1.

### 6.2.3 Ethernet

The Olimex board uses the Crystal LAN CS8900A ethernet controller as illustrated in Section 5.1.2. The *eCos* repository offers a driver for this controller, however this driver needs certain modifications and additions. The steps are described using the processed files.

#### devs_eth_arm_lpcE2294.inl

This file has to be generated under the */devs/eth/arm/lpcE2294* directors in the *eCos* packages source tree. It contains the Olimex LPC-E2294 ethernet definition. Amongst other the ethernet driver is initialised in the I/O subsystem and the device table. Other options have been taken over directly from the edb7xxx *eCos* port, which uses the CS8900 chipset as well.

#### lpcE2294_eth_drivers.cdl

The CDL package describtion file is located in the same main directory as the devs_eth_arm_lpcE2294.inl file mentioned above. The implemented code offers the following configuration options in the *eCos* Configuration Tool.

- Definition of the etho interface name, e.g. "/eth0"
- Definition of the ethernet station address (ESA)
- Selection between interrupt and poll mode of the CS8900 driver

Since the Olimex hardware only supports the poll mode, this mode is pre-selected by default.

#### cs8900.h and if_cs8900a.c

The cs8900.h and if_cs8900a.c files are located under the */devs/eth/cl/* directory of the *eCos* repository. Since all access to the ethernet controller has been implemented using the 16 bit mode, two simple additional routines are added to the driver code as demonstrated in Listing 6.4. These routines substitute the 16 bit access calls to the ethernet controller when the driver is used for the Olimex board.

[6]CDL package: CYGPKG_DEVS_FLASH_ARM_LPCE2294

```c
// the olimex E2294 board accesses the cs8900 in 8bit mode;
// writing and reading have to in 8 bit operation (always)

void cs8900a_lpce2294_write_uint16(unsigned int Addr, cyg_uint16 uiData)
{
  HAL_WRITE_UINT8(Addr, uiData);
  HAL_WRITE_UINT8(++Addr, uiData>>8);
}

cyg_uint16 cs8900a_lpce2294_read_uint16(unsigned int Addr)
{
  cyg_uint8 data_low, data_high;
  cyg_uint16 data16;
  HAL_READ_UINT8(Addr, data_low );
  HAL_READ_UINT8(++Addr, data_high);
  data16 = data_low;
  data16 |= (data_high << 8);
  return(data16);
}
```

***Listing 6.4:*** *Implementation of the CS8900 8-bit read and write access*

As already described for the serial and the flash driver the new driver package has to be included in the ecos.db file. The support for the CS8900 ethernet driver [7] and the board specific driver package[8] have to be included in the target specification in the ecos.db data base as shown in Appendix G, Listing G.1.

### Required further work

Up to the time of the generation of this report the described ethernet driver was not completely implemented and tested. Further work has to be done on the basis of the mentioned modification and development. The write and read access using the routines shown in Listing 6.4 to the CS8900 ethernet chip have been successfully tested in a seperate application. Apparently the initialisation of the ethernet package in the I/O subsystem and device table must contain certain malfunctions. This fact needs further investigation.

### 6.2.4 CAN Driver - *lpccan*

The aimed functionality of the developed CAN driver at this point of the project is summerized in the following list.

[7]CDL package: CYGPKG_DEVS_ETH_CL_CS8900A
[8]CDL package: CYGPKG_DEVS_ETH_ARM_LPCE2294

## Attributes of the CAN Driver

- Implementation in the *eCos* component framework
- Configurability and Selection in the *eCos* Configtool
- Provide write access to the CAN bus
- Reading and progress of incoming CAN messages

Since the CAN hardware, except the CAN bus controller, is completly integrated in the LPC2294 MCU the hardware specific driver is called *lpccan*.

## Hardware independent CAN driver

The standard CAN driver supplied with *eCos* is structured hardware independent. To add support for the new CAN port this existing driver and a hardware specific driver (*lpccan*) with an own interface to the actual CAN hardware is implemented. The further interconnection between the hardware independent and dependent drivers is described seperatelly for the initialisation, write and read calls.

## Implementation in the *eCos* component framework

The functionality of the specific hardware CAN driver is implemented in three files located under the */devs/can/arm/lpc2xxx/lpcE2294/* directory in the *eCos* package repository.

| | |
|---|---|
| **lpccan_lpc2294.h** | This file contains a list which links the CAN hardware addresses to usable defines. Additionally some low level lpc2294 interrupt functions are included. |
| **lpccan_lpc2294.c** | All functionallity of the *lpccan* driver is contained in this file. |
| **can_lpc2294.cdl** | All configuration options of the driver and device are implemented in this package file. |

Furthermore the CAN driver package is included in the ecos.db file to make it available to the Configuration Tool. The hardware independant CAN driver [9] and the board specific driver package[10] are included in the target specification in the ecos.db data base as shown in Appendix G, Listing G.1.

[9]CDL package: CYGPKG_IO_CAN
[10]CDL package: CYGPKG_DEVS_CAN_ARM_LPC2XXX_LPC2294

## Configurability and Selection in the *eCos* Configtool

The CDL file of the lpccan driver offers the following options in the Configuration Tool.

- Selection of the debug message output on the debug port
- Selection of up to four supported CAN ports
- For each CAN port the following options can be selected:

  Definition of the device name, e.g. "/can0"

  Selection of the default baud rate

  RX and TX queue size

  Error and Channel interrupt vector number and priority
- Additional driver build options

The defines of the configuration are stored in the install tree of the *eCos* build (refer Section 3.9) in the *devs_can_arm_lpc2xxx_lpc2294.h* automatically generated by the configuration tool.

## Initialisation of the CAN driver

The access to a CAN device is established in *eCos* by two components. The hardware independent *eCos* CAN driver which offers the interface to the application layer and the hardware specific CAN driver *lpccan*. The specific driver has to implement several macros in order to be linked to the upper CAN driver and therefore to the I/O subsystem. For the complete source code and parameter description of these macros please refer to Appendix D Listings D.1, D.2, and D.3.

The CAN_LOWLEVEL_FUNS macro introduces the standard driver routines (please refer Section 3.7 and Table 3.1) of the specific driver to the upper hardware independent driver. Additional functions which start and stop the transmission of the CAN port are included (lpccan_start_xmit and lpccan_start_xmit).

Each CAN device must have a "CAN channel". This is a set of data which describes all operations on the device comparable to the endpoints for USB devices. It also contains buffers, etc., if the device is to be buffered. The CAN channel is created by the macro CAN_CHANNEL_USING_INTERRUPTS for each implemented CAN device.

Finally the DEVTAB_ENTRY generates the device table entry in the I/O subsystem which includes, amongst others, information about the device name, used CAN channel and the function table.

### Initialisation of the CAN hardware

The lpccan_init driver function is called at bootstrap time and fulfilles the following tasks:

- Initialisation of the driver independent driver by calling (chan->callback->can_init)(chan)
- Initialisation of the Vector Interrupt Controller
- Selection of the Pin select register of the LPC2294 chip
- Disabling of the acceptance filter, hence all CAN messages are accepted
- Set of CAN baudrate
- Linking of receive interrupt service routine and Vector table
- Enabling of the receive interrupt

### Implementation of the write access

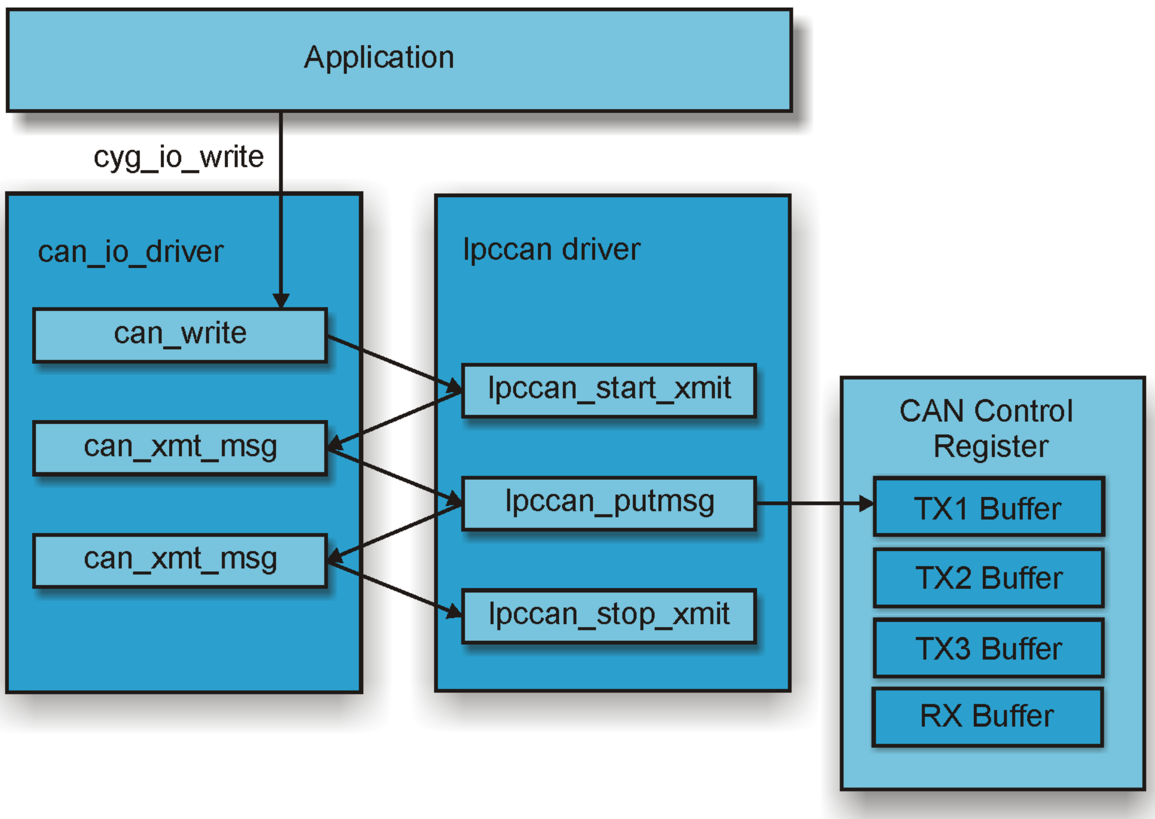


***Figure 6.1:*** *Write process*

A CAN message can be send by an application using the cgy_io_write function of the I/O subsystem interface. Since the CAN access is managed by the cooperation of the hardware independant CAN driver and the platform specific lpccan driver the write call of the application starts the can_write function of the superior CAN driver. This driver first enables the CAN transmitter of the LPC2294 MCU by calling the lpccan_start_xmit function. This function initialises the can_xmt_msg function of the superior can driver. The actual write access to the LPC2294 CAN transmit registers is implemented in the lpccan_putmsg function of the lpccan driver. This function is called out of the can_xmt_msg function as shown in Figure 6.1. Furthermore the lpccan_putmsg routine kicks the transmitter to send the message to the bus. Finally the can_xmit_msg function stops the transmitter.

At the current status of the project the written data on the CAN bus could be measured using an ociloscope. Further investigation and debbugging is required to ascertain why a connected CAN receiver cannot detect the message on the bus.

### Implementation of the read functionality

The implementation of the read access was not finished at the completion of this document. The main stress has to be given to the lpccan_msg_rx_std_isr routine of the lpccan driver. This function reads the receive registers of the LPC2294 MCU and stores the data in a global buffer.

### Further work

- Debugging of the write implementation
- Completition of the read implementation
- Implementation of the driver and device configuration "get" routine during runtime
- Configuration setting call during runtime

## 6.3 Redboot

Once the *eCos* HAL and at least the serial or the ethernet driver have been ported to the target RedBoot can be build. The *eCos* Configuration Tool offers therefore a template which selects all required packages. Furthermore a minimal configuration file is helpful to select the apropriate attributes for the RedBoot application. An example code for a ROM startup minimal configuartion file can be found in Appendix

H, Listing H.1. 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 Olimex board using openOCD and telnet.
If the driver for the external flash is implemented Redboot can be used to give information about or manipulate this memory.

## 6.4 LCD Application

This LCD control example demonstrates several functionalaties and attributes of eCos. The information gained by this demonstration are:

- Genaral structure of an *eCos* application
- Presentation of the GPIO functionality
- Serial device driver functionallity test
- Detemination of the footprint size
- Thread Management Example
  - Thread initialisation
  - Synchronisation with flags and message boxes
- LCD initialisation

The cyg_user_start function is the common entry point for each *eCos* application. In this example the cyg_user_start function implements the following tasks:

- LCD initialisation
- LCD write
- Thread initialisation
- Message box initialisation
- Start of thread execution

After this initialisation process the cyg_user_start function ends and the *eCos* scheduler starts to run, hence both threads are started (cyg_thread_resume). The read thread (rx_thread please refer extract of source code Appendix E Listing E.1) sends a message box (cyg_mbox_put( mbox_handle, (void *)CURSOR_LEFT)) to the write thread (tx_thread) when a buttons was pressed. The message box contains the information whether the left or right button has been used. By then the write thread

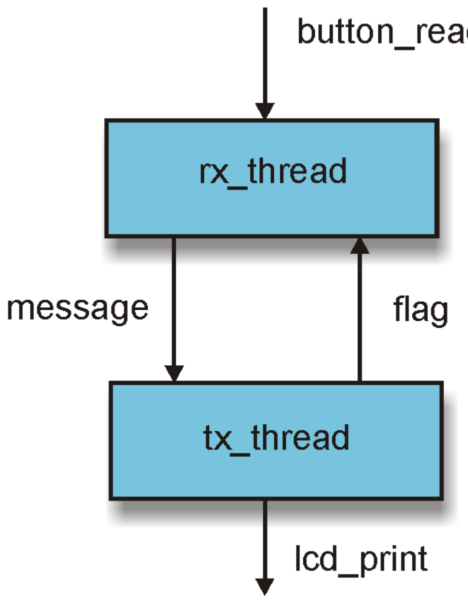


***Figure 6.2:*** *Thread communication and synchronisation*

was stopped to wait for the message box (message = cyg_mbox_get( mbox_handle )). After receiving the mail box the thread processes the information writing to the LCD, sets the information flag for the read thread and starts to wait for the next message box. The set flag indicates to the read thread that he can start his process again.

Furthermore during the process of the system several messages are send to the serial interface i.e. to a terminal. Thereby *eCos* offers twoo different routines to send information to a standard output. The common one is *printf* which requires the stdio library. This library has to be added in the *eCos* configuration to the eCOs system. Additionally several other I/O libraries (e.g. string handling, internationalisation) are required by the stdio package of eCos. Hence the footprint of the *eCos* system has a size using the *printf* routine of 32kB.

However *eCos* provides a more ellegant alternative to avoid the implementation of all memory wasting I/O packages. The *diag_printf* function entirely requires the io_diag library which is include in the I/O subsystem. No further implementations are required. Hence the footprint of the *eCos* binary image shrinks to a size of 28kB. That is an improvment of 8% in this case. Various modifications of this configuration even allow a footprint of 21kB of the binary image. Amongst other exeception handling, interrupt handling, debugging options and the multi-scheduler have been disabled.

# 7 Conclusion and further Work

Within the scope of this Master Thesis the real time operating system *eCos* was ported to the Olimex LPC-E2294 board. A detailed description of the real time operating system is provided by this document. The collected information about *eCos* and the development setup supported the conduction of the project and will help further *eCos* related projects.

The first major task of the thesis was to port the *eCos* HAL to the custom hardware. This goal has been accomplished successfully. Out of the four aimed interface drivers the serial and the flash memory driver have been implemented and tested successfully. The Ethernet and the CAN driver need some further work to be entirely integrated into the *eCos* system. Thereby the developed routines of the Ethernet driver have to be debugged and tested to provide the demanded functionality. The CAN driver lacks of the proper implementation of the read routines and some accomplishing works to provide a write access to the CAN bus. Nevertheless all drivers can be selected and configured using the *eCos* Configuration tool. The conducted HAL port and the drivers have been entirely adapted to the *eCos* component framework.

The implemented applications RedBoot and the LCD control example illustrate the functionality of the *eCos* HAL, the serial driver and the usage of the GPIO ports of the LPC2294 MCU. Additional further works might include the integration of a GUI library and of the GoAhead web server. Since the author of this thesis had no specific experience in the usage of open source development solutions an unexpected hugh amount of time was used to configure the development setup. Compared to commercial tools the open source approach gains the experience in application development significantly more and justifies the additional effort.

# Bibliography

[1] Amontec. http://www.amontec.com.

[2] Century software. http://embedded.centurysoftware.com/.

[3] Cygnus solutions. http://www.cygnus.com.

[4] Cygwin - homepage. http://www.cygwin.com/.

[5] ecos - sourceware. http://ecos.sourceware.org.

[6] The ecos mailing lists. http://ecos.sourceware.org/intouch.html.

[7] Introduction to microwindows programming. http://www.linuxjournal.com/article/4309.

[8] Minigui. http://www.minigui.org.

[9] Nano-x window system. http://www.microwindows.org.

[10] John Dallaway Bart Veer. *The eCos Component Writer's Guide*. 2001. http://ecos.sourceware.org/docs-2.0/.

[11] Ltd Beijing Feynman Software Technology Co. *MiniGUI Datasheet*. 2004. www.minigui.com.

[12] Ltd Beijing Feynman Software Technology Co. *MiniGUI User's Manual*. 2004. www.minigui.com.

[13] Ltd Beijing Feynman Software Technology Co. *MiniGUI White Paper*. 2004. www.minigui.com.

[14] Andreas Buergel. ecos portierung unter besonderer beruecksichtigung der arm-architektur. *http://www.andreas-buergel.de*, April 2003.

[15] eCosCentric Ltd. *eCos User Guide*. 2003. http://ecos.sourceware.org/docs-2.0/.

[16] Nick Garnett et al. *eCos Reference Manual*. 2003. http://ecos.sourceware.org/docs-2.0/.

[17] Wolfhard Lawrenz (Hrsg.). *CAN Controller Area Network; Grundlagen und Praxis*. Huethig, 1994. ISBN 3-7785-2263-7.

[18] Olimex Ltd. *Electronic design and PCB sub-contract assembly*. 2006. http://www.olimex.com.

[19] James P. Lynch. *ARM Cross Development with Eclipse*. 2006. International Business Machines Corp.

[20] Anthony J. Massa. Integrating the goahead webserver and ecos. *Dr. Dobb's Journal*, November 2002. http://www.ddj.com/documents/s=7644/ddj0211e/.

[21] Anthony J. Massa. *Embedded Software Developement with eCos*. Prentice Hall, 2003. http://www.phptr.com/massa.

[22] Philips Semiconductor. *LPC2119/2129/2194/2292/2294 User Manual*, preliminary user manual edition, May 2004.

[23] Philips Semiconductor. *LPC2292/LPC2294 Product Data*, rev. 02 edition, December 2004.

[24] Dominic Rath. *openOCD Debugger*. FH Augsburg, 2005.

[25] Dominic Rath. Open on-chip debugger. *developer.berlios.de*, January 2006.

[26] Cirrus Logic Product Data Sheet. Cs 8900a. *Crystal LAN Ethernet Controller*, September 2004. http://www.cirrus.com.

# A eCos linking and building

```
# Usage:   make INSTALL_DIR=/path/to/ecos/install

INSTALL_DIR:= /home/michix/Studium/Masterarbeit/Software/ecos/images/arm/lpc2xxx/lpcE2294/@
    ecos_lcd_install

include $(INSTALL_DIR)/include/pkgconf/ecos.mak

XCC           = $(ECOS_COMMAND_PREFIX)gcc
XCXX          = $(XCC)
XLD           = $(XCC)

CFLAGS        = -g -O0 -I$(INSTALL_DIR)/include
CXXFLAGS      = $(CFLAGS)
LDFLAGS       = -nostartfiles -L$(INSTALL_DIR)/lib -Ttarget.ld

# RULES

.PHONY: all clean

all: lcd

clean:
        -rm -f  lcd lcd.o

%.o: %.c
        $(XCC) -c -o $*.o $(CFLAGS) $(ECOS_GLOBAL_CFLAGS) $<

%.o: %.cxx
        $(XCXX) -c -o $*.o $(CXXFLAGS) $(ECOS_GLOBAL_CFLAGS) $<

%.o: %.C
        $(XCXX) -c -o $*.o $(CXXFLAGS) $(ECOS_GLOBAL_CFLAGS) $<

%.o: %.cc
        $(XCXX) -c -o $*.o $(CXXFLAGS) $(ECOS_GLOBAL_CFLAGS) $<

lcd: lcd.o
        $(XLD) $(LDFLAGS) $(CFLAGS) $(ECOS_GLOBAL_LDFLAGS) -o $@ $@.o
```

***Listing A.1:*** *Example Makefile for eCos Applications*

# B Comparison of GUI libraries

TableB.1 compares MiniGui, MicroWindows and additionally the QT embedded library. Since this information are directly taken from the MiniGui Whitepaper [13] the outstanding advantages of MiniGui have to be used with caution.

| | **MiniGUI** | **MicroWindows** | **QT/Embedded** |
|---|---|---|---|
| API | Win32 style | X, Win32 subset | QT (C++) |
| API completence | Yes | incomplete with Win32 support | Yes |
| Typical Size of function base | 500K | 600K | 1.5M |
| Portability | Excellent | Better | Good |
| License | GPL/Commercial | MPL | QPL/GPL/ Commercial |
| Multiprocess | Good | Good with X interface, unavailable in Win32 | Good |
| Robustness/ Reliability | Good | Very poor | Good |
| Configurability and Custumability | Good | Fair | Poor |
| Consumption of system Resources | Lowest< Higher (the traditional client/ server architecture based in UNIX socket, featuring frequent in-process communications and higher consumption of system resources) | Highest (C++) | |
| Efficiency | High | Lower | Lowest |
| Operating system support | Linux, uClinux, eCos, uC/OS-II, VxWorks, ... | Linux, eCos | Linux |
| Hardware platform support | x86, ARM MIPS, PowerPC, StrongARM, Minimum CPU frequency 30MHz | x86, ARM MIPS, StrongARM, Minimum CPU frequency 70MHz | x86, ARM, StrongARM , Minimum CPU frequency 300MHz |

***Table B.1:*** *Graphical Library Comparison [13]*

# C CS8900

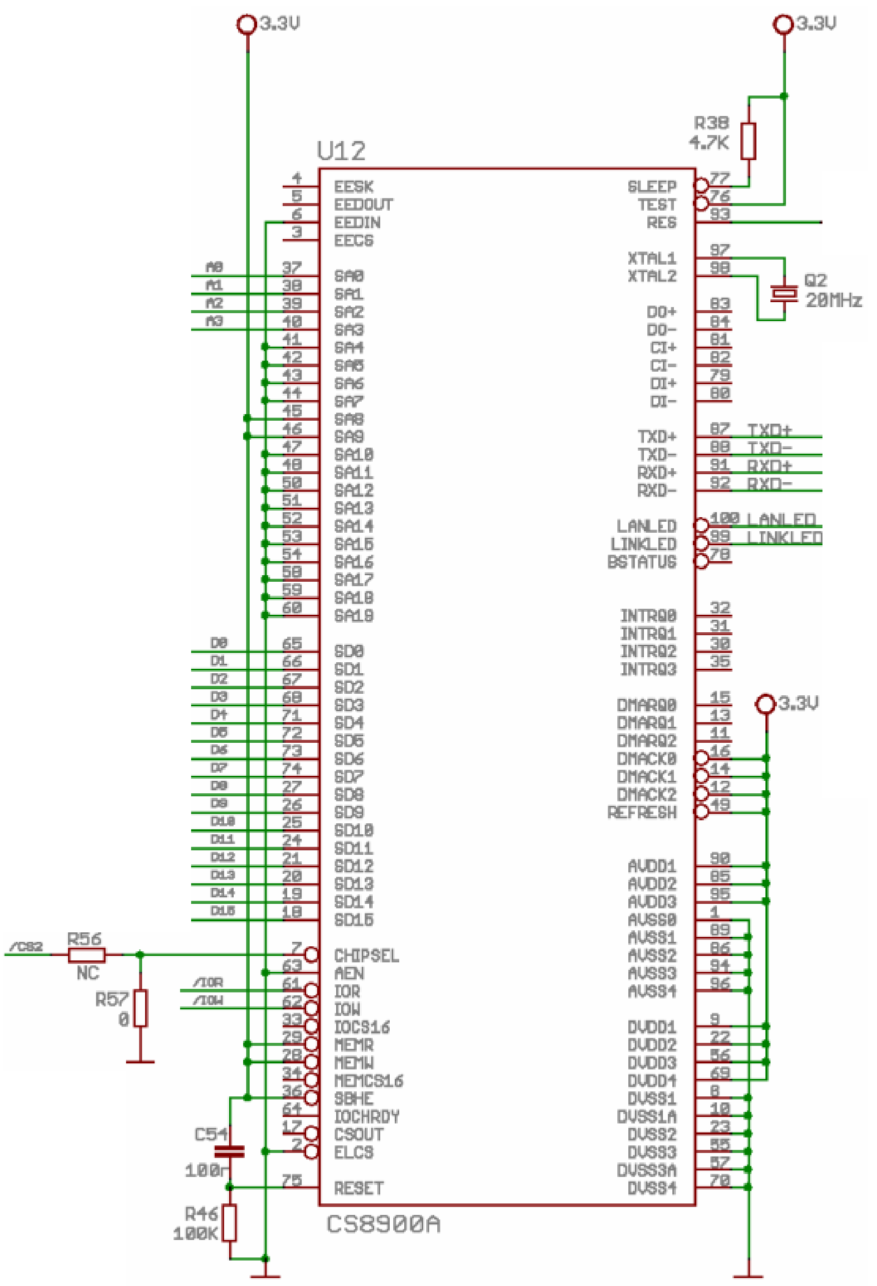


*Figure C.1: cs8900 setup*

# D CAN Driver Macros

```
CAN_LOWLEVEL_FUNS(lpccan_lowlevel_funs,
                  lpccan_putmsg,
                  lpccan_getevent,
                  lpccan_get_config,
                  lpccan_set_config,
                  lpccan_start_xmit,
                  lpccan_stop_xmit
     )
```

***Listing D.1:*** *CAN driver functions*

| Parameter | Description |
|---|---|
| lpccan_lowlevel_funs | The "C" label for this structure. |
| lpccan_putmsg | This function sends one CAN message to the interface. |
| lpccan_getevent | This function fetches one CAN message from the interface. It will be only called in a non-interrupt driven mode, thus it should wait for a character by polling the device until ready. |
| lpccan_set_config | This function is used to configure the CAN port during runtime. |
| lpccan_get_config | This function is used to get the current configuration of the CAN port.configure the port. |
| lpccan_start_xmit | In interrupt mode, turn on the transmitter and allow for transmit interrupts. |
| lpccan_stop_xmit | In interrupt mode, turn off the transmitter. |

***Table D.1:*** *CAN_LOWLEVEL_FUNS Macro parameter description*

```
// buffer for rx can events
cyg_can_event lpccan_can1_rxbuf[CYGNUM_DEVS_CAN_ARM_LPC2XXX_LPC2294_CAN1_QUEUESIZE_RX];
// buffer for tx can messages
cyg_can_message lpccan_can1_txbuf[CYGNUM_DEVS_CAN_ARM_LPC2XXX_LPC2294_CAN1_QUEUESIZE_TX];

CAN_CHANNEL_USING_INTERRUPTS
        (lpccan_can1_chan,
        lpccan_lowlevel_funs,
        lpccan_can1_info,
        CYG_CAN_BAUD_RATE( CYGNUM_DEVS_CAN_ARM_LPC2XXX_LPC2294_CAN1_KBAUD),
        lpccan_can1_txbuf, CYGNUM_DEVS_CAN_ARM_LPC2XXX_LPC2294_CAN1_QUEUESIZE_TX,
        lpccan_can1_rxbuf, CYGNUM_DEVS_CAN_ARM_LPC2XXX_LPC2294_CAN1_QUEUESIZE_RX
    );
```

***Listing D.2:*** *CAN channel initialisation*

| Parameter | Description |
|---|---|
| lpccan_can1_chan | The "C" label for this structure. |
| lpccan_lowlevel_funs | The set of interface functions (see below). |
| lpccan_can1_info | A placeholder for any device specific data for this channel. |
| CYG_CAN_BAUD_RATE | The initial baud rate value. |
| lpccan_can1_txbuf | Pointer to the output buffer. NULL if none required. |
| CYGNUM_DEVS_CAN _ARM_LPC2XXX _LPC2294_CAN1 _QUEUESIZE_TX | The length of the output buffer. |
| lpccan_can1_rxbuf | pointer to the input buffer. NULL if none required. |
| CYGNUM_DEVS_CAN _ARM_LPC2XXX _LPC2294_CAN1 _QUEUESIZE_RX | The length of the input buffer. |

***Table D.2:*** *SERIAL_CHANNEL_USING_INTERRUPTS macro parameter description*

```
DEVTAB_ENTRY(lpccan4_devtab,
            CYGDAT_DEVS_CAN_ARM_LPC2XXX_LPC2294_CAN4_NAME,
             0,              // Does not depend on a lower level interface
             &cyg_io_can_devio,
             lpccan_init,
             lpccan_lookup,  // CAN driver may need initializing
             &lpccan_can4_chan
    );
```

***Listing D.3:*** *Device table insertion macro*

| Parameter | Description |
|---|---|
| lpccan4_devtab | The "C" label for this devtab entry |
| CYGDAT_DEVS _CAN_ARM _LPC2XXX _LPC2294 _CAN4_NAME | The "C" string for the device, e.g. /can1, defined using the Configuration tool |
| 0 | Used lower level interfaces, here none |
| &cyg_io_can_devio | The table of I/O functions. This set is defined in the hardware independent serial driver and should be used. |
| lpccan_init | The module initialization function. |
| lpccan_lookup | The device lookup function. This function typically sets up the device for actual use, turning on interrupts, configuring the port, etc. |
| &lpccan_can4_chan | This table (defined below) contains the interface between the interface module and the can driver proper. |

***Table D.3:*** *DEVTAB_ENTRY macro parameter description*

# E Threads using the LCD Example

```
#ifdef __cplusplus
extern "C"
{
#endif
    int cyg_user_start(void)
    {

        // Set counter GATE input low (0) to halt counter while it's being setup
        lcd_light_out();
        lcd_init();
        lcd_print("Press a BUTTON!!");
        lcd_light_on();
        lcd_cursor_on();
        lcd_cursor_blink();

    cyg_thread_create(
        4,                              // thread priority
        rx_thread,                      // thread routine
        (cyg_addrword_t) "th_read",     // thread specific data
        "th_read",                      // name of thread
        (void *) th_rx_stack,           // pointer to thread stack

        THREAD_STACK_SIZE,              // size of stack
        &th_rx_handle,                  // pointer to thread handle
        &gThread_rx);                   // pointer to thread object


    cyg_mbox_create(&mbox_handle,&mbox);  // mbox allocation
    ...

    cyg_thread_resume ( th_rx_handle);  // start execution
    cyg_thread_resume ( th_tx_handle);

    }
#ifdef __cplusplus
}
#endif



 // read thread
 void rx_thread ( cyg_addrword_t pThreadData)
 {
        ...
        ...
                // start tx thread since button is pressed
                cyg_mbox_put( mbox_handle, (void *)CURSOR_LEFT );
                // Wait for the appropriate bits to be set in the flag by tx_thread
                cyg_flag_wait( &flag_var,
                3,
                CYG_FLAG_WAITMODE_OR |
                CYG_FLAG_WAITMODE_CLR
                );
                lcd_wait_long();
        }
        ...
        ...

  }
 void tx_thread ( cyg_addrword_t pThreadData)
```

```
{
       void *message;
       // Run this thread forever.
       while ( 1 )
       {
       // Wait for the message.
       message = cyg_mbox_get(mbox_handle);
       // Make sure we received the message before attempting
       // to process it.
       if ( message != NULL )
       {
           if((*( int *)message) == CURSOR_RIGHT)
           {
                lcd_cursor_right();
           }
           else
           {
               lcd_cursor_left();
           }
       }
       else
       {
               diag_printf("no message received!!\n");
       }
       // Set the appropriate flag bits to signal rx_thread
       cyg_flag_setbits( &flag_var, 1 );
 }
}
```

***Listing E.1:*** *Extract of Thread example*

# F Flash Driver Extract

```
// The Olimex LPC-E2294 has one Intel 28F320C3 flash memory part.

#define CYGNUM_FLASH_INTERLEAVE (1)
#define CYGNUM_FLASH_SERIES     (1)
#define CYGNUM_FLASH_WIDTH      (16)
#define CYGNUM_FLASH_BASE       (0x80000000)
```

***Listing F.1:*** *Extract of the LPC-E2294 specific flash driver support file*

# G CDL Files

```
package CYGPKG_HAL_ARM_LPC2XXX_LPCE2294 {
        alias           { "Olimex evaluation board LPCE2294 " hal_lpcE2294_arm }
        directory       hal/arm/lpc2xxx/lpcE2294
        script          hal_arm_lpc2xxx_lpcE2294.cdl
        hardware
        description "
        The lpcE2294 HAL package provides the support needed to run eCos on an
        the lpcE2294 evaluation board from Olimex."
}
target lpcE2294 {
        alias { "Olimex evaluation board LPC-E2294" lpcE2294 }
        packages { CYGPKG_HAL_ARM
                   CYGPKG_HAL_ARM_LPC2XXX
                   CYGPKG_HAL_ARM_LPC2XXX_LPCE2294
                   CYGPKG_DEVS_FLASH_INTEL_28FXXX
                   CYGPKG_DEVS_FLASH_ARM_LPCE2294
                   CYGPKG_DEVS_ETH_ARM_LPCE2294
                   CYGPKG_DEVS_ETH_CL_CS8900A
                   CYGPKG_IO_SERIAL_GENERIC_16X5X
                   CYGPKG_IO_SERIAL_ARM_LPC2XXX
                   CYGPKG_DEVICES_WATCHDOG_ARM_LPC2XXX
                   CYGPKG_DEVS_CAN_ARM_LPC2XXX_LPC2294
                   CYGPKG_IO_CAN
        }
        description "
        The e2294 target provides the packages needed to run eCos on the
        LPC-E2294 evaluation board from Olimex."
}
```

***Listing G.1:*** *ecos.db for the Olimex Platform*

```
package CYGPKG_DEVS_FLASH_ARM_LPCE2294 {
       alias           { "Support for the external flash memory on the Olimex LPC-E2294 @
           board" flash_lpcE2294 }
       directory       devs/flash/arm/lpcE2294
       script          flash_lpcE2294.cdl
       hardware
       description "
            This package contains hardware support for flash memory
            on the Olimex LPC-E2294 platform."
}
```

***Listing G.2:*** *ecos.db entry for the flash driver package*

```
cdl_package CYGPKG_DEVS_FLASH_ARM_LPCE2294 {
    display       "ARM LPCE2294 FLASH memory support"

    parent        CYGPKG_IO_FLASH
    active_if     CYGPKG_IO_FLASH
    requires      CYGPKG_HAL_ARM_LPC2XXX_LPCE2294
    implements    CYGHWR_IO_FLASH_DEVICE

    compile       arm_lpcE2294_flash.c

    cdl_interface CYGINT_DEVS_FLASH_INTEL_28FXXX_REQUIRED {
        display   "Generic Intel FlashFile driver required"
    }
    implements    CYGINT_DEVS_FLASH_INTEL_28FXXX_REQUIRED
    requires      CYGHWR_DEVS_FLASH_INTEL_28F320C3
}
```

***Listing G.3:*** *CDL package description for the LPC-E2294 flash Driver*

# H RedBoot Minimal Configuration File

```
cdl_savefile_version 1;
cdl_savefile_command cdl_savefile_version {};
cdl_savefile_command cdl_savefile_command {};
cdl_savefile_command cdl_configuration { description hardware template package };
cdl_savefile_command cdl_package { value_source user_value wizard_value inferred_value };
cdl_savefile_command cdl_component { value_source user_value wizard_value inferred_value };
cdl_savefile_command cdl_option { value_source user_value wizard_value inferred_value };
cdl_savefile_command cdl_interface { value_source user_value wizard_value inferred_value };

cdl_configuration eCos {
    package CYGPKG_IO_FLASH current ;
};

cdl_option CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE {
    user_value 6144
};

cdl_option CYGDBG_HAL_COMMON_INTERRUPTS_SAVE_MINIMUM_CONTEXT {
    user_value 0
};

cdl_option CYGDBG_HAL_COMMON_CONTEXT_SAVE_MINIMUM {
    inferred_value 0
};

cdl_option CYGDBG_HAL_DEBUG_GDB_INCLUDE_STUBS {
    inferred_value 1
};

cdl_option CYGSEM_HAL_ROM_MONITOR {
    inferred_value 1
};

cdl_option CYGSEM_HAL_USE_ROM_MONITOR {
    inferred_value 0 0
};

cdl_component CYG_HAL_STARTUP {
    user_value ROM
};

cdl_component CYGBLD_BUILD_REDBOOT {
    user_value 1
};


cdl_option CYGDAT_REDBOOT_CUSTOM_VERSION {
    set rv {UNKNOW}
    regsub {[$]Revision: (.*?) [$]} {$Revision: 1.1 $} {\1} rv
    user_value 1 $rv
    unset rv
};

cdl_component CYGBLD_BUILD_REDBOOT_WITH_THREADS {
    user_value 1
};
```

```
cdl_option CYGBLD_BUILD_REDBOOT_WITH_IOMEM {
    user_value 1
};

cdl_option CYGBLD_BUILD_REDBOOT_WITH_EXEC {
    user_value 0
};
```

***Listing H.1:*** *RedBoot Minimal Configuration File*

this page intentionally left blank

# Document Revision History

| Rev. | Date | Author | Description |
|---|---|---|---|
| 0.1 | 2006-03-20 | M. Labus | Initial Creation |
| 0.2 | 2006-04-30 | M. Labus | Draft release |
| 1.0 | 2006-05-18 | M. Labus | Final release |

***Table 8.1:*** *Revision history;*

this page intentionally left blank