Migration from a Real-Time Operating System to Embedded Linux

Håkan Kvist and Maria Larsson

October 24, 2005

Abstract

Many embedded systems today are built around off-the-shelf components which are both fast and inexpensive. With every new generation of components, performance increases and the price/performance ratio decreases. The extra amount of resources available in new systems opens up a door for more general operating systems for embedded devices, Linux being one popular alternative. But replacing a real-time operating system specially developed for use in embedded devices with a general purpose operating system comes at some costs.

This thesis investigates the considerations that have to be made when porting to Linux, and some of the improvements and drawbacks that come with a migration. The investigation was carried out by partly migrating an embedded software product currently running on a real-time operating system (RTOS), and by evaluating the ported system. It could be concluded that the porting difficulties are solvable, and that a total migration to Linux is feasible.

Measurements showed both positive and negative effects of the migration, and it was found that a decrease in real-time performance had taken place. Most notable were the context switching times, which were at least 14 times lower when performed by the RTOS. Network communication was on the other hand 76% faster under Linux. Although a decrease in real-time performance was observed, it could be concluded that Linux is able to provide sufficient real-time properties to satisfy the requirements of the application.

Preface

This master's thesis was performed at TAC in Malmö, in collaboration with the Department of Computer Science at Lund Institute of Technology. The work was carried out between April and September in 2005.

We would like to thank TAC for providing us with an interesting master's thesis proposal and giving us the opportunity to carry out the project with the help and support of many skilful and experienced employees. We greatly appreciate the people at TAC for their encouragement and for making our stay at the company enjoyable. Special thanks goes to our supervisor, Sven Björck, for his practical guidance and endless support, and to Andreas Håkansson, Marcus Skälstad, Anders Davidsson and Allan Björck for answering our questions and helping us out with our problems. We are also grateful to Mårten Berggren from ENEA, for sharing his knowledge with us.

We also wish to acknowledge the Department of Computer Science and especially our supervisor Dr Jonas Skeppstedt, for his valuable input and useful suggestions during our work.

Finally, we would like to express our gratitude to Bo Kvist and Eskil Jakobsson, who have read and corrected this report.

October 24, 2005

Håkan Kvist
Maria Larsson

Contents

List of Figures

List of Tables

Part I

Background

Chapter 1

Introduction

> *I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since April, and is starting to get ready.*
>
> - Linus Torvalds, 26 Aug 1991

1.1 Background

Linus Torvalds did most likely not realize the impact that Linux would have on the market, when he began writing on his own hobby operating system, as a student at Helsinki University. Linux has got the interest of companies like IBM, Sun and HP, which are now shipping Linux as an alternative operating system to Microsoft Windows on smaller servers and their own UNIX implementations on mainframes.

However, Linux is not only being improved for use in large scale 64 bits mainframes. It is also evolving for smaller applications where memory and resources are limited.

Hardware is getting faster, smaller and cheaper. The days when embedded applications were written largely in assembly language for systems with very small memory footprints are definitely over. Currently most embedded applications are using real-time embedded operating systems with small overhead, but as performance and resources improve in the embedded market, general purpose operating systems like Linux look more and more attractive for actors in the embedded market.

1.2 Purpose

The goal of the thesis is to port an embedded application developed at TAC from an existing real-time operating system to an embedded version of Linux. During this work, the following questions are to be answered.

- What are the benefits and drawbacks of Linux compared to a traditional RTOS?
- How can an appropriate development environment be set up?
- What problems are to be expected when porting to Linux?
- Which important design choices must be made during API implementation?
- Does Linux provide sufficient real-time properties?

Though the thesis deals with a specific porting process, it can be expected that similar difficulties and dilemmas will be encountered in other ports. Hopefully, this report can be helpful to anyone interested in porting an embedded application from an RTOS to Linux.

1.3 Methodology

The migration to Linux is performed by first porting the software to a desktop computer running standard Linux. After that an embedded Linux distribution is ported to the target platform, in order to move the ported software to the embedded system. The methodology for the thesis was outlined as follows:

- Create a development environment suitable for embedded Linux development.
- Port parts of the software to Linux running on a PC.
- Submit general improvements back to the original software.
- Port uClinux, an embedded Linux distribution, to the embedded system.
- Verify that the ported software also runs on the original hardware, using uClinux.
- Evaluate the performance of the embedded Linux system compared to the original system.

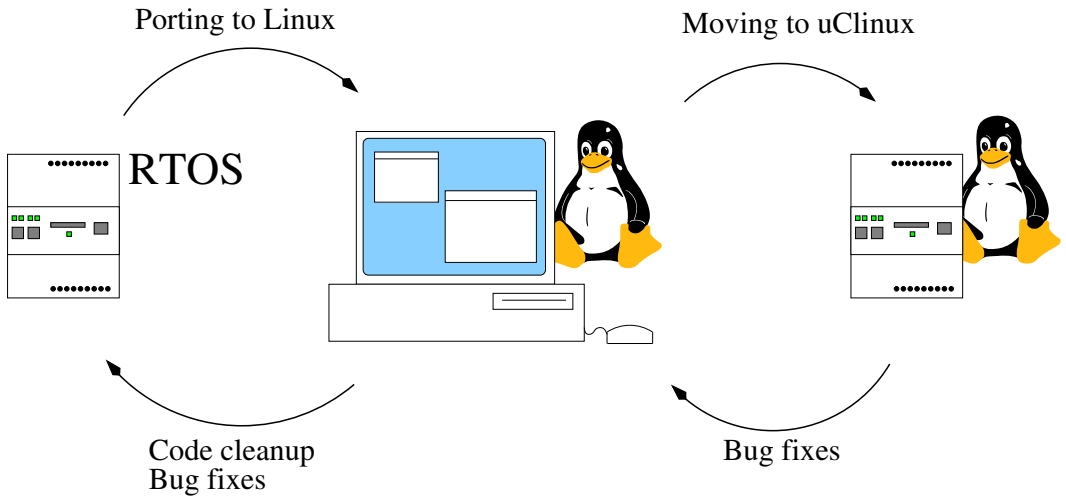

Figure 1.1: The work flow towards an embedded Linux solution.

The reason for this procedure is that it is much easier to port the software to a PC, where you have access to a large set of good debugging tools. Once the system runs on the PC, the process of moving the software to another Linux environment should not be very difficult. By delaying the move to the specific target system, this method is also more hardware independent and thus simplifies the process of moving the system to another hardware platform. This argument is specially important in this project, because the current hardware of the embedded system is likely to be replaced in the near future.

1.4 Report Outline

The report consists of twelve chapters, divided into three distinct parts: *Background*, *Migration* and *Evaluation*. The first part lays the foundation by presenting the thesis and describing the GNU/Linux operating system as well as the system to be ported. The motivation behind the porting is also presented. The second part describes the different steps of the migration and explains the problems that occurred and how they were solved. The third part of the report consists of an evaluation of the migration work and of the ported system.

Part I: **Background**

- Chapter 1: **Introduction**
  A presentation of the master thesis.
- Chapter 2: **The GNU/Linux Operating System**
  A description of the GNU/Linux operating system.
- Chapter 3: **Embedded Linux**
  A survey of the use of Linux in embedded systems.
- Chapter 4: **The Domino System**
  A description of the system that is to be ported to Linux.
- Chapter 5: **RTOS to Linux Porting Considerations**
  A comparison between Linux and a traditional RTOS, pointing out issues that need to be considered before migrating to Linux.

Part II: **Migration**

- Chapter 6: **Development Environment**
  A presentation of the different tools that were used to create the development environment and the motivation behind these choices.
- Chapter 7: **API Implementation**
  A description of the OS interface and our implementation of it.
- Chapter 8: **Porting to Linux running on a PC**
  A description of the process of porting a portion of the software to Linux running on a standard PC.
- Chapter 9: **Moving to uClinux running on the Target Hardware**
  A description of the process of moving the product to the embedded platform running uClinux.

Part III: **Evaluation**

- Chapter 10: **Real-Time Performance Measurements**
  A comparison of the real-time properties of the system running on Linux and the original system.
- Chapter 11: **Discussion**
  A discussion of the difficulties of porting to Linux and the advantages and disadvantages of the proposed solution.
- Chapter 12: **Conclusions and Further Development**
  A final presentation of our conclusions and proposals for further developments.

Chapter 2

The GNU/Linux Operating System

2.1 History

2.1.1 Linux

The first version of the Linux kernel was created in 1991, as the hobby project of a young Finnish student named Linus Torvalds. The project started as a terminal emulator, which got more and more functionality until it evolved into an operating system. Linus's work soon got attention from other developers around the world, of which some began to contribute improvements to the Linux software, in about the same way as the users of UNIX in the beginning distributed patches between each other[1].

The amount of contributors to the Linux kernel still increases, but Linus remains as the project coordinator. This means he has the final word when it comes to deciding what contributions should become part of the kernel.

2.1.2 GNU

Richard M. Stallman decided to quit his job at MIT Artificial Intelligence Lab in 1984, in order to implement a free operating system with the name GNU (GNU's Not UNIX). Free in this context means that you were allowed to modify the source for your own needs, but your changes had to be made public.

The first project was GNU Emacs, a free version of the popular Emacs editor. It was distributed on the Internet through FTP servers, and sold on tape for those without Internet access. In 1985 Stallman founded the Free Software Foundation (FSF) in order to promote free software. FSF employees have written a lot of software packages, of which two of the most well known are the GNU Compiler Collection (a set of compilers targeting multiple operating systems), and GNU Coreutils, which features replacement versions for all the usual UNIX tools (e.g. cp, ls and []). While the FSF has successfully created a lot of popular tools, their own UNIX replacement kernel, GNU HURD, is not widely used.

In 1989, the FSF created the GNU General Public License (GPL) for its own software and for others who want to use it. The GPL is a license which grants the end user the following rights[2].

[1] In the beginning of UNIX, AT&T offered UNIX for free for universities but without support or bug fixes. This immediately forced users to exchange information, ideas and fixes with one another [Sal94].

[2] The list is taken from the Wikipedia page about the GPL, `http://en.wikipedia.org/wiki/GPL`, August 2005.

- The freedom to run the program, for any purpose.
- The freedom to study how the program works, and modify it. (Access to the source code is a precondition for this.)
- The freedom to redistribute copies.
- The freedom to improve the program, and release the improvements to the public. (Access to the source code is a precondition for this.)

The GPL seeks to ensure that the above freedoms are preserved in copies and in derivative works. This means that if you link GNU-licensed software into your own software, the whole project will automatically get licensed under the GPL. This quite controversial side of the GPL has led to statements like "Linux is cancer" from closed source company leaders. Despite this, the GNU GPL is probably the most used license for free and open source software today. A less "aggressive" license is GNU LGPL, *Lesser Gnu Public License*, which allows linking of LGPL software to software with any license. The C library shipped with GCC is licensed with LGPL in order to allow GCC to be used for non-open software.

2.1.3 GNU/Linux

Torvalds created the Linux kernel, which alone was unusable, but together with tools from the GNU project you have a working operating system. Since version 0.12 the Linux kernel is licensed under the GPL, and therefore the combination of GNU software and the Linux kernel makes up the base for a free GNU-licensed operating system. For this reason it is argued that you should refer to this combination as the GNU/Linux operating system.

Today GNU/Linux is widely used all over the world, by big companies and individuals. It is being actively developed with contributions from both individuals and large corporations, for use in many different systems, among others: large mainframes, desktops and embedded systems.

2.2 POSIX

POSIX stands for "Portable Operating System Interface for UNIX" and is a collection of standards defining the API of a UNIX operating system [Wal04]. The standard was developed by IEEE and is formally named IEEE 1003.1. UNIX is a registered trademark of The Open Group, and to put the UNIX name on a system, it must be certified by the Open Group. Since this is a costly process, the GNU/Linux operating system is formally not a UNIX system, although it conforms to the POSIX standard.

2.2.1 The Shell

A shell is an interactive text-based program, from within all system management can be done through a set of commands. Although Linux nowadays offers a well-developed graphical user interface, the shell continues to be an important part of any POSIX system.

2.2.2 User Management

POSIX systems are generally multiuser systems, and hence they provide means to distinguish between different users in the system. A user mainly consists of a username and a password associated with a user ID. A user has a set of privileges or rights, which determines his or her

permissions to access system resources and files. To be able to share privileges, users can be members of one or several groups.

2.2.3 The File System

An important design property of POSIX systems is the fact that everything is represented by files. This includes for example regular files, directories, sockets and physical devices. Each file has a set of attributes specifying which users and groups can read, write or execute the file.

The file system on a POSIX system is a single hierarchal directory structure. Everything originates from the root directory, represented by '/', and then expands into sub-directories. All partitions are logically connected to the structure under the root directory by "mounting" them under specific directories.

2.2.4 The Process Model

A process is basically a program in execution. The identity of a process consists mainly of a process ID and an associated user ID, which determines permissions of the process. The environment of a process is composed of the command line arguments (the argument vector), and a set of name value pairs (the environment vector).

The POSIX systems are multitasking and the scheduling is preemptive. POSIX supports three different scheduling policies: `SCHED_FIFO`, `SCHED_RR` and `SCHED_OTHER`. `SCHED_FIFO` represents rate-monotonic scheduling, which is a strict priority-based scheme. The term FIFO refers to the fact that processes with the same priority are served in a first-in first-out fashion. Using the round robin policy (`SCHED_RR`), the processes are time-sliced within a priority level [Wol01]. Processes that do not require real-time scheduling should use the `SCHED_OTHER` policy, which uses the standard Linux time-sharing scheduler.

2.3 Processes and Threads

Being a POSIX system, the GNU/Linux operating system conforms to the POSIX process model. Like most modern operating systems, Linux supports both processes and threads. An attempt to define these two concepts would be to declare that a process is an execution of a program, while a thread is separate, concurrent execution context within a process [SGG00]. The most important difference between processes and threads is that processes execute in separate address spaces, while threads belonging to the same process share address space with each other.

However, the Linux kernel handles threads similarly to how it handles processes. Processes and threads are for example scheduled in the same way. Threads can be created by the `clone` system call, which creates a new process and allows for total control of properties which are to be shared between the parent and the child. The system call for creating new processes, `fork`, is actually just a special case of `clone`. Although `clone` can be used directly in application programs, it is much more preferable to use the POSIX thread (pthread) library, which provides mechanisms for creating, terminating and synchronizing threads.

2.4 Memory Management

Linux supports virtual memory, which is a technique to allow a large logical address space to be mapped to a smaller physical address space [SGG00]. To satisfy this equation, a portion

of the memory must reside in secondary storage and be brought into memory on demand.

Virtual memory makes it possible for Linux processes to execute in their own address spaces, and prevents them from overwriting each other's memory as well as kernel code and data. Processes are also prevented from overwriting their own code.

When running uClinux (Linux without MMU support), the convenience of memory protection is lost. It is no longer possible to use virtual memory, since the translation from logical to physical addresses is done by the MMU.

2.5 Interprocess Communication

Linux provides an environment with powerful mechanisms for allowing processes and threads to communicate with each other. Interprocess communication (IPC) is any kind of data exchange between processes, and may involve anything from extensive data transfer to merely letting another process know that some event has occurred. The IPC facilities provide two major services: synchronization and data communication.

2.5.1 Synchronization

Synchronization is a very important aspect of concurrent programming and can be defined as coordination with respect to time. The need for synchronization exists for example in a producer-consumer relationship or when simultaneous access to resources must be prevented.

Semaphores

The most frequently used synchronization mechanism is the semaphore. Linux provides both binary and counting semaphores through a UNIX System V conforming API. The semaphores are uniquely identified by integer keys, and can be accessed from any process. The semaphores are arranged in arrays of up to 64 semaphores [Jon05] and operations can be performed on individual semaphores or on the whole array.

An alternative is the POSIX semaphores, which are much lighter weight than the System V semaphores. A POSIX semaphore structure defines a single semaphore and not a whole array. Currently the Linux implementation of POSIX threads does not support process-shared semaphores, and hence they can only be used between threads.

Mutexes and Conditions

The POSIX thread library offers several additional facilities for synchronization between threads. Mutual exclusion is commonly accomplished by a mutex, which is similar to a binary semaphore. There are three types of mutexes, which differ in the way they handle the situation where a thread tries to lock a mutex it already owns. A "fast" mutex causes a deadlock when simply suspending the calling thread, while an "error checking" mutex returns with a deadlock error. A "recursive" mutex records the number of times the thread has locked the mutex, and doesn't return to the unlocked state until it has been unlocked the same number of times.

A condition variable is a synchronization device that allows threads to suspend their execution until some predicate on shared data is satisfied. A condition variable must always be associated with a mutex, to avoid a race condition when a thread prepares to wait for a condition variable and another thread signals the condition just before the first thread actually waits for it.

Signals

Signals are primarily used to inform processes of the occurrence of asynchronous events. A process can register to handle a particular type of signal and provide a signal handler to be executed when the signal is received. A process with appropriate permissions can send a signal to another process or to a whole process group.

2.5.2 Data communication

Pipes

A pipe is a one-way communication facility, often used to connect the output of one process to the input of another process. The feature is widely used on the command line, but it can also be utilized within programs. The common practice is to let the parent process create a pipe and then pass it to two child processes so that they can communicate. Technically a pipe is a pair of file descriptors, of which one is open for writing and one is open for reading.

There are two types of pipes: anonymous and named pipes. The anonymous type is the most common and it is usually the one intended when simply referring to a pipe. A named pipe, which also can be called a FIFO, works like a regular pipe except that it exists in the file system so that any process can find it.

Sockets

The BSD conforming socket API allows for communication over a network, but sockets can just as well be used between separate processes on the same machine. Unlike pipes, sockets provide full-duplex communication. The communication is accomplished with the TCP or the UDP protocol, with IP addresses and port numbers identifying the communicating parties.

Message Queues

Linux message queues, which conform to the System V UNIX model, provide means for sending messages between processes. A message queue is identified with an integer value and can be created and destroyed by a process. The message structure can be arbitrarily defined.

Shared Memory

The fastest type of interprocess communication is based on shared memory. Since Linux processes exist in different address spaces, special facilities are needed to accomplish the shared memory.

An efficient way to share memory between two processes is to use the `mmap` system call, which maps the contents of a file into user program space. Another way to let multiple processes attach a segment of physical memory to their virtual address spaces is to use the System V shared memory API.

2.6 I/O System

As previously mentioned, physical devices appear as normal files and hence they are protected by file permissions. Devices are either block devices, character devices or network devices. Block devices include all devices that allow random access, such as hard disks and

CD-ROMs, and can be accessed directly by applications. Most other devices, apart from network devices which handle the network interface, are character devices. Character devices, such as loudspeakers or mouses, do not need to support all the functionalities of regular files.

Chapter 3

Embedded Linux

3.1 History

The interest in getting Linux to run on embedded devices is not a phenomenon that has popped up recently. An old project is ELKS (Embedded Linux Kernel Subset), started in 1995, which aims at getting a subset of the Linux kernel to run on 8086 processors.

Some embedded applications must operate in time critical real-time environments. In order to get real-time properties with Linux, the RTLinux project was started in 1997. Here the Linux kernel runs as a process under a real-time kernel which takes care of the real-time threads together with scheduling of the applications and Linux itself.

In 1998 the uClinux project got on the Internet with a port of the Linux kernel for MMU-less 68000 based systems with more architectures to follow, thereby opening a new world of smaller[1], simpler and thus cheaper processors for Linux.

Beginning in 1998-1999 companies like Lineo and Montavista have pushed for Linux in the embedded market. In recent times embedded systems are used almost everywhere and Linux is gaining popularity rapidly. Figure 3.1 shows the market growth of embedded Linux in the past few years and a prediction of the near future[2]. The diagram shows the world wide investments in embedded Linux, bundled products and related services.

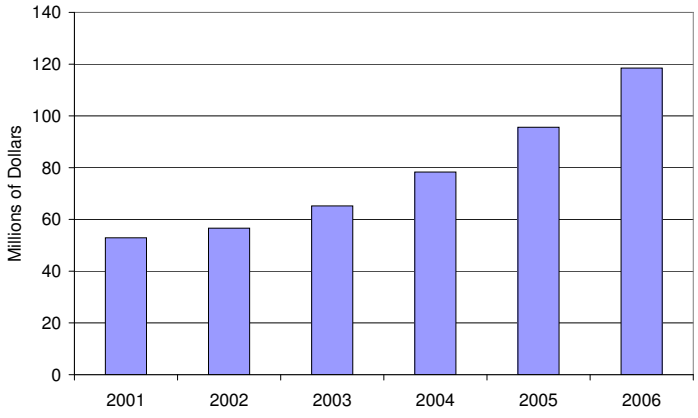

Figure 3.1: Embedded Linux market growth

[1]The MMU usually occupies a large amount of the silicon on a CPU.

[2]The data has been obtained from the Venture Development Corporation (VDC) by Montavista.

3.2 Embedded Linux Distributions

With suitable hardware, it is possible to use a general purpose Linux distribution, such as SUSE or Red Hat, in a non real-time embedded system. However, it exists a variety of Linux distributions customized for use in embedded systems.

When obtaining a distribution, there are basically three options. The easiest solution is to use a commercial distribution, which gives you a thoroughly tested kernel and user applications with continuous support and maintenance. A commercial distribution typically comes with kernel, tools and utilities pre-configured for a specific host and target. A popular choice today is Montavista Linux, which includes a complete embedded Linux operating system and cross development environment.

In addition to the commercial distributions, there are a number of open source alternatives freely available for download. For instance, the Embedded Debian Project provides a Debian distribution customized for embedded systems. There also exist hard real-time alternatives, such as RTLinux and RTAI (Real-Time Application Interface). The hard real-time properties are accomplished by running Linux as the lowest priority task within a minimalistic real-time kernel.

The project behind the MMU-less Linux kernel also provides an open source distribution under the same name as the kernel, uClinux. Although many open source distributions are well maintained and supported, there are no guarantees. Also, it may be necessary to make adjustments to fit the specific configuration. However, there are commercial companies offering help with most open distributions.

The third option is to build your own distribution with your own selection of kernel version, applications and libraries. This is of course the most difficult solution, but it can result in an operating system completely customized to fit your needs. However, you have to maintain the system yourself and you can not count on any support.

3.3 Real-Time Properties

Standard Linux does not guarantee any maximum response times, and hence it is not a hard real-time system. Linux was originally designed to maximize throughput and not to minimize response times. However, the real-time properties of Linux have improved significantly during the past years. The 2.6 kernel contains a number of additions that increases the support for real-time applications, among which kernel preemption points are one of the most important [Gup03]. In former kernel versions, system calls could not be interrupted and hence other processes could be blocked for a long time. With kernel preemption points, a system call can be interrupted at predefined places in the code.

Another important improvement is a more efficient scheduling algorithm. It now runs in constant time, which means that the time it takes to schedule a task does not depend on the number of tasks in the system [Raj04]. Further, the timer granularity has been increased and the synchronization mechanisms have been improved. The older LinuxThreads implementation of the POSIX thread library has been replaced by the improved Native POSIX Thread Library (NPTL).

Through a more modularized architecture, the 2.6 kernel also has better support for custom design. The most important changes made by the uClinux project has also been adopted into the standard kernel, which means that standard Linux kernel can be built with or without MMU support.

These improvements imply that Linux is rapidly approaching the performance of a proper real-time operating system. It is therefore likely that even mainstream Linux already or in the

near future will provide sufficient real-time support for many embedded applications.

3.4 uClinux

The embedded Linux distribution used in this project is uClinux, intended for small MMU-less processors and developed in the Embedded Linux/Microcontroller project[3]. The first uClinux version was released in 1998 and has since then grown to be one of the most widely used embedded Linux distributions. While originally developed for the Motorola 68000 chip, the number of available ports is increasing. Among the supported architectures are Motorola ColdFire, ADI Blackfin, ETRAX, ARM7, ARM9 and Intel i960. Today the uClinux distribution incorporates the 2.0, 2.4 and 2.6 kernel and includes a collection of libraries, utility programs and tool chains.

3.4.1 Programming for uClinux

The most evident difference between uClinux and a standard Linux distribution is the absence of a virtual memory system. This means that all processes execute in a single physical address space and there are no memory protection mechanisms. Any process is able to access any memory location, without causing segmentation faults. This fact makes it more difficult to develop applications for the uClinux platform.

With a few exceptions, uClinux offers a full Linux API. An important deviation is the lack of the `fork` system call, which makes an exact copy of the original process and executes it simultaneously. Virtual memory is used to map the memory from the parent process to the child and only the memory modified by the child is actually copied. With no virtual memory, uClinux can therefore not provide the `fork` system call. Instead applications need to use the `vfork` system call, in which the parent is halted until the child exits or loads a new program by a call to `execve`.

With a virtual memory system, each application seems to have a large memory all by itself. If the physical memory becomes full, data can just be swapped in and out from disk. This is not possible on uClinux, and hence developers must be prepared for the event that the memory is full. Also, the problem of memory fragmentation becomes more important on an MMU-less platform. When trying to allocate a piece of memory, it could easily arise a situation where the total amount of free memory is sufficient, but the allocation fails because memory must be allocated on contiguous addresses.

3.4.2 uClinux/ARM

Unlike many of the other uClinux architectures, the uClinux ARM code was not incorporated in the 2.6 kernel. The uClinux/ARM 2.6 project is a uClinux based project carried out by Hyok S. Choi at Samsung, which aims to port the 2.6 kernel to MMU-less ARM architectures[4]. Using an ARM7 based processor without MMU, this is the Linux port employed in this project.

[3] http://www.uclinux.org
[4] http://opensrc.sec.samsung.com

Chapter 4

The Domino System

4.1 Introduction

TAC is a company focused on developing, manufacturing and marketing building control solutions. One of the key products is the TAC Xenta series, which is a family of LonWorks based controllers. This thesis focuses on the TAC Xenta 511, which is a controller with a web-based presentation system. Using a standard web browser, the operator can view and control the devices in the LonWorks network via the Internet or a local intranet. The embedded software platform used in this device is called Domino.

The two most important components of the Domino system are the operating system and and the application, which is implemented as a number of software modules. Between these components is a generic OS interface, introduced to facilitate porting to another operating system, a limited implementation of the C language library and a socket interface. The application code is not supposed to use any operating system specific system calls, but only use functions from these three interfaces.

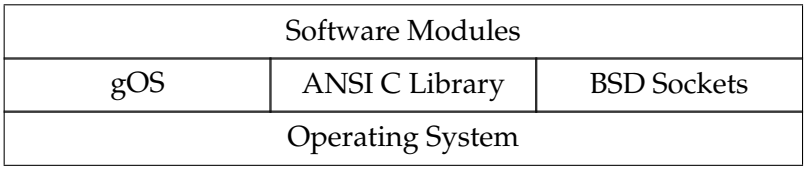

Figure 4.1: The Domino platform architecture

4.2 Hardware

The hardware used during this thesis is based on an ARM processor. ARM uses a RISC architecture with low power consumption and is thus popular on the embedded market. The specific chip used is a SoC from Samsung, based on the ARM7TDMI core, which lacks MMU and FPU. In our configuration the processor runs at 50 MHz. The platform is equipped with 16 MiB of RAM and 16 MiB of flash for storing the Domino application and associated data. The communication ports include two RS 232 ports and one 10 Mbps Ethernet interface. In addition to these common standard ports, there are ports for LonWorks network and RS 485. If extra storage is required, a MMC[1] flash memory can be installed.

[1]Multi Media Card, a flash memory card standard.

Figure 4.2: TAC Xenta 511

4.3 The Etnoteam Operating System

The operating system that is part of the Domino platform is called the Etnoteam Operating System (EOS). It is a scalable, multi-tasking RTOS designed for real-time embedded applications [eos99]. The architecture is modular, which makes it possible to simply remove features that are not required and hence reducing the size and complexity of the system. The system was developed with special focus on portability, and it supports a number of hardware architectures.

EOS was the product of the Italian company Etnoteam, which no longer maintains the system. Any maintenance must therefore be made by TAC, which is one of the reasons why it is highly desirable to move to another operating system.

4.3.1 Task Management

In EOS, as in a typical RTOS, the units of work are referred to as tasks. A task is an execution unit with its own stack and register content, identified with a globally unique task ID. All system resources and global variables are shared by all tasks. At any time, a task is either executing, ready for execution, suspended by another thread, waiting for a resource to become available or asleep. Any running task can change its own state and suspend any other task.

Each task is assigned a priority, which is used by the CPU scheduler to decide which task is allowed to execute. There are two ranges of priorities that divide the tasks into high priority tasks and normal priority tasks. If there are high priority tasks ready to execute, the scheduler gives the control of the CPU to the task with the highest priority. If all ready tasks have normal priority, the scheduling is based on the round robin policy and the tasks are assigned time slices based on their priorities.

4.3.2 Memory Management

Traditional RTOSs seldom make use of virtual memory, since page faults (occurring when a desired memory item is not currently resident in main memory) can introduce unexpected delays. Accordingly, the EOS operating system has no support for virtual memory, and hence

all tasks execute in a single physical address space. With the current hardware this is actually a necessity, since the lack of a MMU makes it impossible to utilize virtual memory. EOS does not provide any mechanism to protect the tasks from overwriting each others memory areas. It is also possible for tasks to overwrite their own code as well as kernel memory.

The memory management in EOS is fairly straightforward. There is a global memory pool, which is used to allocate stacks and system resources. Tasks are able to allocate memory blocks from the global memory, which they are responsible to return when no longer needed. EOS also provides a mean to handle local storage for each task.

4.3.3 Interprocess Communication

Synchronization

EOS offers three types of synchronization mechanisms: semaphores, conditions and events. The mechanisms can for example be used to implement mutual exclusion. EOS makes no attempt to prevent deadlocks.

A semaphore consists of an integer value that can be seen as the number of free units of a resource. A task grabs a resource by decrementing the semaphore and releases the resource by incrementing the semaphore. If there are no free resources when a task tries to decrement the semaphore, it must wait until a resource is released.

A condition is a simple boolean value managed by the operating system, that can be used instead of semaphores to implement simple synchronization. A task can wait for a condition or a combination of conditions.

Events are simple messages that can be sent between tasks. As with conditions, task can wait for a specific combination of events.

Data Communication

Since all tasks can access each other's memory, the fastest and easiest communication between tasks is done through shared memory. However, in accordance with good programming practices, EOS provides mechanisms to make the communication more explicit and abstract the synchronization work.

In addition to events, which are used mainly for synchronization, EOS provides message passing through ports. The ports are global mailboxes and the messages simply consist of memory addresses. To transfer data between a producer and a consumer task, EOS offers buffer pools. A pool is a set of equally sized buffers, which can be allocated by the tasks. Asynchronous communication can be achieved by the signal mechanism, which provides a way to make a task execute a particular action upon receipt of a particular signal.

4.3.4 I/O System

File I/O is handled by the EOS Portable File System, which provides tasks with a file system API independent of actual physical media. The media dependent part of the file system is implemented by installable modules in a lower level.

Management of a particular type of peripheral is handled by a driver, which consists of a set of routines with a device independent interface that can be dynamically activated and deactivated. The drivers define interrupt handlers that are executed when an interrupt is received.

Network communication is made available to application programs by means of the EOS socket API, which apart from a few deviations is compatible with the BSD socket API.

4.4 The Generic OS Interface

The Domino Generic OS (gOS) interface was developed by TAC to minimize the operating system dependencies, in order to facilitate a future porting to another operating system [gos00]. The interface functions can be divided into eight categories: task management, semaphore handling, message queue handling, time management, memory management, file system handling, system tracing and power failure handling. The complete API can be found in appendix C.

The interface functions were chosen to represent the smallest subset of functions that the Domino software demands from an operating system. A portion of the EOS system calls were considered unnecessary and therefore have no equivalent in the generic interface. Accordingly, semaphores are the only type of synchronization mechanism and message queues is, apart from shared memory, the only type of interprocess communication.

However, gOS does not only contain renamed versions of a subset of EOS system calls. A number of additional features have been added to the interface, such as support for system tracing and power failure handling. Most notable are the additions to the task concept, which has developed towards a UNIX style process. Each task has its own set of open files and environment variables and its privileges are determined by a user id. The arguments passed to a newly created task have the same form as for a C main function: an integer value representing the number of arguments and an argument vector. Some of these additions impact the behaviour of functions in the C library, which have been implemented accordingly.

4.5 The C Library

Some functions of the ANSI C library are heavily OS dependent, and these functions must be adapted in order to work with the underlying kernel. These include file handling functions such as `fopen` and `printf`, the memory management functions `malloc` and `free` and the time functions `time` and `clock`. Some of the functions in the ANSI C library are heavily interconnected to the gOS functions, especially with respect to user authentication and file permissions.

4.6 The Application Software

The main purpose of the application software is to present data through a web-based interface, which is also used to configure the device.

4.6.1 Architecture

The software has a modularized architecture, which allows the same source code to be used in the generation of a number of different products. Through a simple configuration step, system features can be selected and deselected. The modules described here are considered the most important, and they are also included in the portion of the software chosen for porting.

4.6.2 Variable Server

The most important module is probably the variable server, which is the central component around which the other modules are situated. The variable server basically provides a unified way of accessing different variables in the system, while hiding the actual access operations.

The variables can be read or modified independently or in groups. It is also possible to be notified whenever a particular variable changes and to define the time interval between notifications.

The variable server acts as the spider in the web, with a number of modules registered to maintain certain variables and a number of modules registered to access certain variables. The variables are declared and sent through the network according to the Abstract Syntax Notation 1 (ASN.1) standard. The variables can be accessed directly through the variable server interface or through an IP interface provided by a daemon process. Apart from the variables handled by the different modules, there is a set of general system variables like date and time.

4.6.3 Web Server

The web server used in Domino was originally part of the EOS operating system, but it has been extracted and turned into an application module. It makes use of the SSL module, which is another extracted part of EOS. The web interface first asks the user to log in to the system, and then presents the available choices depending on the configuration of the device.

Incoming HTTP requests are handled by installable web modules, which are activated depending on the URL requested. Each request can be handled by one or several modules. For instance, a simple file request first goes through the authentication module and if the access was granted the file module returns the contents of the file.

4.6.4 Trend Logging

One of the most important features of the system is the possibility to log variables, and present them in a web page either as data or in a graph. There is a special web module responsible for displaying trend log data on the web. Towards the variable server, the logging module acts both as a variable user and a variable provider, since it both fetches variable data and provides the logging data.

4.6.5 Alarm Handling

The alarm module supervises a set of variables and issues an alarm if a variable has reached a value satisfying a specified condition. As any other signal in the system, an alarm can be logged and viewed as a graph on the web site, where the user also has the possibility to acknowledge or block alarms.

4.6.6 Command Line Interface

The Domino system provides a command line interface, called the Domino Shell (DSH), which can be accessed through the web site or through the serial interface. The DSH interface makes it possible to monitor system data by listing information from the different modules. It also includes commands to affect the behaviour of the program, e.g. by adding a web user or even restarting the whole system. Through UNIX inspired commands like `cd`, `rm` and `cat`, users can browse and manipulate the file system.

Chapter 5

RTOS to Linux Porting Considerations

5.1 Architecture

The architectural differences between Linux and a traditional RTOS are substantial. This becomes most evident in a comparison between the task model of an RTOS and the process model of a Linux system. The different execution models are illustrated in figure 5.1.

5.1.1 The RTOS Model

RTOS-based applications usually consist of a set of tasks executing in a single shared physical address space. The RTOS model is highly exposed to corruption, since every task can overwrite the memory of any other task, including the operating system. Not only can errors corrupt the whole system, they are also typically difficult to trace. The result of an error might not show up until a later time, and then it can appear anywhere in the system.

5.1.2 The Linux Model

In the Linux programming model, applications execute in their own protected address spaces. By the use of virtual memory implemented by an MMU, they are prevented from overwriting code or data of other processes as well as their own code. Attempts to access prohibited memory results in a segmentation violation signal, which causes the process to terminate. A failure is therefore discovered immediately, and it is easy to see when and where it occurred.

In addition to processes, Linux provides simultaneous execution also by using threads. Threads, which are the Linux equivalence to RTOS tasks, execute in a single process and share memory with each other. While threads belonging to the same process are free to access each other's memory, they still receive a segmentation violation signal if they try to access the memory of other processes or to overwrite their own code.

5.1.3 Processes and Threads

Since RTOS tasks have a closer resemblance to threads than to processes, the natural choice when porting an application from an RTOS to a Linux platform is to map each RTOS task to a Linux thread. Although this is surely the easiest way, it is not necessarily the best. The choice is highly dependent on the properties and demands of the application to port. Of course, a solution could include a combination of processes and threads. To build a basis for this decision, the properties of processes and threads will be further investigated.

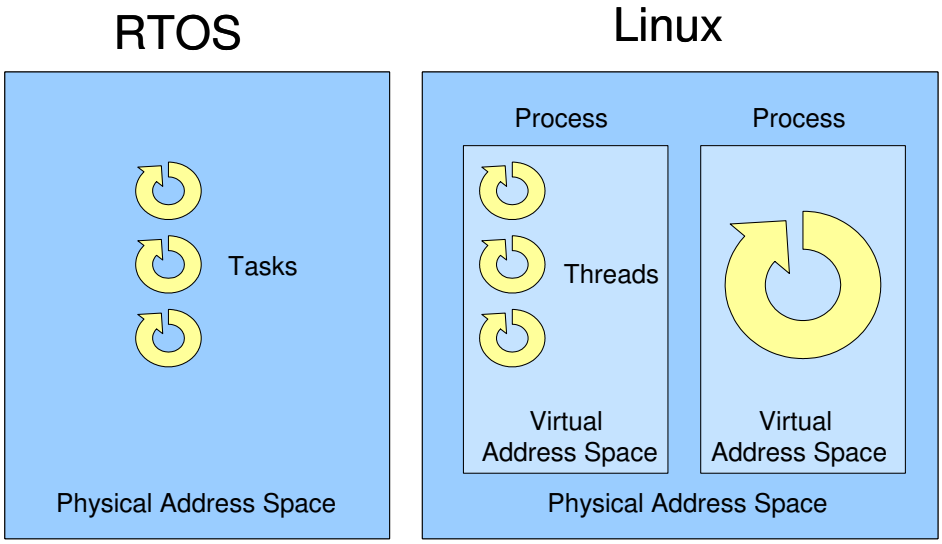

Figure 5.1: Concurrent executions models

From a performance point of view, threads are better than processes. Unlike processes, threads share code and data space and hence threads are faster to create and the context switch overhead is smaller.

The second advantage with threads is the ease of communication. While threads can communicate with each other through shared memory, processes need to make use of some kind of IPC facility. Apart from the need to rewrite the application to use IPC instead of shared memory, it also decreases performance.

In addition, threads are usually created in the same way as RTOS tasks. One simply gives a pointer to the function defining the thread. In order to create a new process, a program file must be loaded into memory. This means that one needs to create an executable for every task in the system.

Threads also introduce some problems. The synchronization between threads is more complicated than the synchronization between processes. Since the data space is shared among threads, file descriptors for open files and sockets are shared. Therefore, a multi-threaded application must protect resources against simultaneous access. Communication between processes on the other hand must be explicitly defined, therefore it is easier to identify where synchronization is needed than when working with threads. The implicit communication through shared memory makes thread synchronization a bit trickier. However, synchronization should not be a problem during a port, since the tasks are hopefully already properly synchronized.

A more alarming disadvantage of threads is the need to use thread-safe or reentrant functions. In most C libraries many functions are not reentrant, since they make use of static variables[1]. This is not a problem for processes, since they execute in separate address spaces and cannot share variables.

Processes can also benefit from memory protection, which makes it easy to discover errors. Threads are allowed to write in each other's memory areas, and there is no way to distinguish between correct and erroneous writes. Further, a thread that fails will terminate all other threads belonging to the same process. When a process fails, it simply exits and does not affect the execution of other processes.

[1]The only validated C99 library, Dinkum C library (http://www.dinkumware.com), can be configured as thread-safe.

5.1.4 Application Properties

Eventually, the choice between threads and processes depends on the properties of the application. For applications with hard real-time requirements, processes are likely to introduce too much overhead and therefore threads are preferable. The choice also leans towards threads if the application is imbued with use of shared memory. In this case the application needs to be heavily rewritten to be able to use processes.

On the other hand, if the real-time requirements allow for the performance penalties introduced by processes, there are a lot of benefits to gain in converting the tasks into processes. This requires that the tasks are fairly independent and isolated with little or well-defined communication.

It may often be possible to divide tasks into groups, sufficiently isolated from each other to form processes. The tasks belonging to a group are more dependent on each other and therefore suited to run as threads inside the process. A recommended porting method is to first map tasks to threads. When everything is working as expected, tasks or groups of tasks can be extracted to form isolated processes.

5.2 Application Programming Interfaces (APIs)

The interface between the applications and the operating system is made up by a set of functions or system calls. The system calls can be accessed directly or through library functions. All functions available for applications are defined by a set of Application Programming Interfaces (APIs). The available APIs are specific to each operating system, and API accommodation is therefore a most vital task when porting applications from one operating system to another.

5.2.1 Accommodation Strategies

There are three fundamentally different strategies when accommodating the API of one operating system to the API of another:

- Equivalence
- Emulation
- Recoding

Equivalence

When facing a function unsupported by the new operating system, the most natural solution is to find an equivalent function. Linux offers a rich set of APIs, and the possibility of finding a suitable function is fairly high. In this case, the application may be slightly modified to use the Linux supported function and it may also be necessary to include additional header files. To avoid changing the application, simple wrapper functions can be implemented.

Emulation

When no equivalent function exists, the unsupported function may be implemented in an emulation layer between the application and the operating system. For common RTOSs like VxWorks and pSOS, it exist commercially available emulation kits [Wei01]. These kits emulate more or less the whole RTOS API and hence simplify the porting process significantly.

Recoding

The third strategy is to rewrite the application to avoid the need for the unsupported function. In cases where such a rewrite is impossible or very difficult, it may be necessary to consider loosing some application functionality.

5.2.2 Linux APIs

In order to increase code portability, Linux and other open systems rely on the use of standard, common APIs. The most important standard is POSIX, which is a family of standards defining the functionality of a UNIX system. The most interesting parts of the POSIX standard include POSIX.1, POSIX.1b and POSIX.1c. POSIX.1 defines the core of a POSIX system, including key features like process management and the file system. POSIX.1b focuses on IPC and synchronization, while POSIX.1c defines POSIX threads. The POSIX standard is widely spread and it is not uncommon that an RTOS implements parts of the standard. In that case, typically subsets of the IPC and thread API are implemented.

Linux also implements large subsets of the SVR4 (System V, Revision 4) UNIX standard and versions of the BSD (Berkeley Software Distribution) UNIX interfaces. The System V API for interprocess communication and the BSD socket API are de facto standards in Linux systems.

Many of the API functions used in embedded applications are simply standard C language routines. The C language libraries provided by Linux support the ANSI C standard from 1989 as well as a large fraction of the most recent C standard, ISO C99 from 1999. Since the RTOS implementations often are subsets of the complete C library, the C functions used by an RTOS application will surely be supported on the Linux platform.

5.2.3 Interprocess Communication (IPC) and Synchronization

The selection of IPC and synchronization facilities may constitute a major difference between two operating systems. Luckily, the Linux operating system supplies the users with a wide range of mechanisms, and often it exists an equivalence for the RTOS construct used by the application. However, even if a certain type of mechanism exists, it is not certain that both operating systems support exactly the same operations. The accommodation of APIs can be performed using one or a combination of the previously suggested strategies.

RTOS IPC	**Linux IPC**
Semaphores	System V or POSIX semaphores
Mutexes	POSIX mutexes or condition variables
Message queues and mailboxes	Pipes/FIFOs or System V message queues
Shared memory	System V shared memory or `mmap`
Events and RTOS signals	Signals
Timers and task delays	POSIX timers (on some architectures), System V timers or `sleep` / `nanosleep`
Watchdogs, task registers, partitions/buffers	Emulation or recoding

Table 5.1: RTOS and Linux IPC

Table 5.1 shows how different IPC and synchronization utilities of an RTOS can be mapped to the Linux IPC API. Common mechanisms like semaphores and message queues can be moved into the Linux world relatively easy, while RTOS specific facilities like watchdogs and task registers introduce more difficulties. Since Linux does not offer any similar mechanisms, these APIs must be emulated. Alternatively, the application may be rewritten.

5.3 The Benefits of Porting to Linux

Before migrating to another operating system, it is important to investigate the advantages and disadvantages of such an action. Since Linux is a general purpose operating system, it has a lot more features than an RTOS. Hence, in many aspects Linux is superior to any RTOS. A conceivable disadvantage of Linux is a decrease in performance. Since an RTOS has been designed to meet hard real-time requirements, it can be suspected that the performance under Linux will be poorer. However, compared to other general purpose operating system, Linux is considered to be a high performance OS. In the rest of this section some of the advantages of moving from a traditional RTOS to embedded Linux are pointed out.

5.3.1 An Open Source System

Linux is a royalty-free, open source operating system. It is based on robust solutions which anyone can view, criticize or improve on. This makes it possible to develop your own extensions to suit special needs, which may often be the case in embedded development.

Because of the high number of developers, the GNU/Linux operating system goes through a constant improvement. New functionality is added and bugs are fixed very quickly. In addition, the Linux community is very active and offers support through forums and mailing lists.

The fact that Linux is free does not mean that it comes without a license. Linux is licensed under GPL, which contains some obligations regarding source distribution[2]. When migrating to Linux, there is a good chance of finding other interesting open source software. Importing code from an open source project into your own project must however be done carefully, since there are a variety of licenses in the open source world. Some licenses might be incompatible with each other and some might change the license of your own software. An overview of the most common licenses is presented in table 5.2[3].

License	Free Linking	Free Derivation	Unrestricted Re-licensing	Copyright Privileges
GPL	No	No	No	No
LGPL	Yes	No	No	No
BSD	Yes	Yes	No	No
NPL	Yes	Yes	No	Yes
MPL	Yes	Yes	No	No
Public Domain	Yes	Yes	Yes	No

Table 5.2: An open source license overview

[2]See Chapter 2 for more information on the GPL license.

[3]This compilation of open source licenses was presented by Claes Lundholm at the Nohau seminar about Montavista Linux in Gothenburg, April 2005.

5.3.2 Important Features

The GNU/Linux operating systems is a full-fledged OS with support for a variety of network and file handling protocols, which are very important features for an embedded system. The operating system is modular, which makes it easy to slim down. This property makes the OS very suitable for systems with limited resources.

Together with the operating system comes an abundance of utility programs. With Linux running in an embedded system, you can easily perform a remote login using SSH. This will give you access to a Linux shell and all the utility programs installed.

Linux also supports a variety of hardware platforms, which decreases the hardware dependencies. With the Linux operating system, the task to move the system to another hardware platform is relatively simple.

5.3.3 Development Advantages

Contrary to a traditional RTOS, Linux provides memory protection, an advantageous feature primarily during development. Erroneous memory accesses can often be discovered immediately, and do not corrupt the whole system. For processors without MMU, there is a Linux version called uClinux. Because of the hardware limitations, this version does not support memory protection. However, since an application can easily be moved between uClinux and standard Linux, you have the possibility to test and debug the application on a standard PC with memory protection.

Using Linux as the host operating system, you can benefit from the advantage of developing on the same platform as the application should run on. For time independent testing, there is no need for cross-compilers and downloading the software to the target hardware.

Linux also provides a very good environment for software development, including a rich set of development and debugging tools[4].

[4] A comprehensive list of Linux development tools can be found at http://www.hotfeet.ch/~gemi/LDT.

Part II

Migration

Chapter 6

Development Environment

6.1 Background

The selection of development tools is crucial, since it can have a huge impact on the efficiency and success of software projects. Currently, the development work for the Domino system is carried out using any personally chosen IDE under the Windows XP operating system. The IDEs most frequently used are Borland CodeWright and Microsoft Visual Studio. Compilation of the software is done using Borland Make with the Norcroft ARM C compiler and configuration management is handled by Microsoft Visual SourceSafe. The software can be debugged through a JTAG interface using the Lauterbach Trace32 debugger, which allows for both high-level and assembler debugging.

The goal is to create a development environment for Linux which integrates all the different aspects of development, such as code editing, configuration management, compiling and debugging. To further integrate the different tasks of the developers, it is also desirable that the chosen IDE supports a number of programming languages and not just the C language.

6.2 Operating System

The choice of host operating system naturally fell on Linux, which allows for development and testing on the same platform. When selecting a Linux distribution it is mostly a matter of personal taste, as all important tools are available on all the common distributions. In this project SUSE Linux 9.2 has been used.

As access to software only available under Windows XP was a necessity, it was decided to continue using the Windows XP operating system, and run Linux on top of it. This was accomplished by the use of VMware Workstation, a software product for creating virtual machines in which other operating systems can run. This solution provides the possibility to run Linux, while still being able to use important Windows applications in a convenient way.

6.3 Integrated Development Environment

There are a variety of different Integrated Development Environments (IDEs) available for the Linux operating system. As the chosen IDE eventually is to be adopted by developers used to Windows, a self-explanatory graphical user interface was one of the main requirements. For this reason, text-based IDEs like Emacs was taken out of consideration.

Currently, the two most popular Linux IDEs for C/C++ development are KDevelop and

Eclipse[1]. KDevelop is a C/C++ focused IDE made primarily for the K Desktop Environment (KDE), while Eclipse, written in Java, is more of a platform for building your own IDE using plug-ins. Because of its portability and extensibility, Eclipse was determined to be the best choice.

6.3.1 Eclipse

Eclipse is an open source, platform-independent framework, primarily used for building IDEs. The functionality of Eclipse is controlled by the plug-ins that are added to the system. Eclipse was originally an IBM project, but in 2001 the development was taken over by the Eclipse Foundation, a non-profit software vendor consortium.

The plug-ins for Java development are the most widely used, but plug-ins for a large number of other languages are available. The CDT (C/C++ Development Tools) project provides the Eclipse platform with a fully-functional C and C++ IDE. Since the project is relatively young, the C/C++ support is not as developed as the Java support, but the maturity level is currently quite sufficient and it is quickly improving.

6.4 Configuration Management

Configuration management (CM), including source code management and version control, is the key to managing and controlling a large software project. Among companies working in a Windows environment, one the most frequently used configuration management tools is probably Visual SourceSafe from Microsoft. This product is notorious for its unpredictable behaviour [Goo04]. In fact, Microsoft doesn't even use it for its own internal development.

In the open source world, the Concurrent Versions System (CVS) has been the de facto standard for a long time. It is a well-reputed and robust system, but over the years a number of problems have become noticeable. Designed explicitly to replace CVS, Subversion is beginning to take over as the leading open source CM tool. Atomic commits and support for moving and renaming files are a few of the features of Subversion, missing in both SourceSafe and CVS [CSFP05]. The following table points out the most important differences between these three systems[2].

	SourceSafe	**CVS**	**Subversion**
Atomic commits	No	No	Yes
File/directory renaming	No	No	Yes
File/directory copying	Yes	No	Yes
Line-wise file history	No	Yes	Yes
Documentation	Medium	Excellent	Very good
Ease of deployment	Very good	Good	Medium
Networking support	Limited	Good	Very good
Portability	Limited	Good	Excellent
Web interface	No	Yes	Yes
License	Proprietary	Open source	Open source

Table 6.1: A version control system comparison

[1] According to the "IDE of the year" poll, which was part of the "2004 LinuxQuestions.org Members Choice Award".

[2] The comparison was performed by the "Better SCM" project, `http://better-scm.berlios.de`.

Being the superior open source version control system, Subversion was chosen to be a part of the development environment. The existence of a Subversion plug-in for the Eclipse platform further motivated this decision. For the Windows platform there is a highly commended Subversion client called TortoiseSVN, which is implemented as a Windows shell (Explorer) extension.

6.5 Compilation

Working in a Linux environment, the GNU toolchain is the obvious choice when looking for compiler tools. The key component is the GNU Compiler Collection (GCC), a set of compilers for languages like C, C++ and Fortran. Written by Richard Stallman in 1987 as an open source compiler for the GNU project, it was originally pronounced GNU C Compiler as it only included support for the C language. GCC has now become the compiler supporting the largest number of processors and operating systems, with developers all around the world. Accompanying GCC in the GNU toolchain, GNU Binutils offer a collection of programming tools such as a linker and an assembler.

The appurtenant tool for automating the compiling process is the GNU Make utility. It basically keeps track of which files need recompiling after a source code modification, avoiding recompilation of the whole project. Although not used in this project, GNU Make is often used together with the GNU Autotools, which provide a convenient way to build software for different platforms.

6.6 Debugging

It exists a large number of debugging tools for the Linux environment. Most famous is the GNU Debugger (GDB), which is the last link of the GNU toolchain. GDB itself defaults to a command line interface, but it exists a number of graphical GDB front-ends, such as the Data Display Debugger (DDD) and the GNU Visual Debugger (GVD). In this project GDB was used together with the front-end provided by the Eclipse C/C++ environment. Other debugging and profiling tools found useful during the project were strace, Valgrind and Ethereal.

The strace[3] debugging tool prints out all the system calls made by a program. In this project it has been particularly helpful for tracking file and socket operations.

Valgrind[4] is an excellent tool for finding memory related errors. It works by emulating the CPU and checking all memory references, in order to discover for example memory leaks, use of uninitialized memory and inappropriate memory accesses. Valgrind can also perform detailed profiling, aiming to speed up and reduce memory use.

When debugging the HTTP communication modules we had great help of Ethereal[5], a network protocol analyser. With help from Ethereal we could easily see what kind of data had been sent, and use this information to draw conclusions about errors that occurred.

6.7 Cross Compilation

Problems were encountered when trying to use the toolchains provided by the uClinux/ARM 2.6 project, since those only include compilers with support for little endian processors. Us-

[3] http://www.liacs.nl/~wichert/strace
[4] http://www.valgrind.org
[5] http://www.ethereal.com

ing a target platform with big endian byte order, the solution was to build our own toolchain. This was performed using Crosstool[6], which is a set of scripts written by Dan Kegel to simplify the building of cross-compiler suites.

In order to successfully build against the target, one must make sure that the compiler uses the correct include files and not the build system's include files or libraries. However, this is handled by the uClinux build system and did not cause any worries.

6.8 Remote Debugging

In order to debug on the target hardware, a hardware debugger from Lauterbach (the Trace32 in-circuit debugger) was used. This debugger supports a lot of different operating systems, among them Linux 2.4, Linux 2.6 and uClinux 2.4.

The Lauterbach debugger enables debugging from the first instruction of the boot sequence, which was most helpful when debugging the boot sequence of the Linux kernel.

On a running system, applications can be debugged remotely with GDB. In that case a "GDB server" runs the application on the target, while being controlled by a "GDB client" running on a workstation. The server/client GDB debugging has not been used during this project.

[6] http://kegel.com/crosstool

Chapter 7

API Implementation

7.1 Domino Generic OS Interface

Instead of using the raw system calls of the underlying operating system, the Domino software calls functions specified in the generic OS (gOS) interface. A large part of the interface was developed with the functions of EOS in mind, which resulted in a close resemblance between the EOS system calls and the generic interface functions. Hence, implementing the functions for another operating system is a more complex task.

During API implementation a combination of the three accommodation strategies presented in section 5.2.1 was used. Many of the gOS functions had equivalents in the Linux APIs, while others had to be implemented from scratch.

Although there were situations when the easiest solution would have been to rewrite the application code, this was not often done. An important requirement, which is not present in all porting situations, was that it should be possible to generate executables for both EOS and Linux from the same source code. Satisfying this requirement while avoiding to sully the code with preprocessor directives, the source code could not easily be changed in order to fit the Linux operating system.

During the implementation of the API, a few interface functions were found to be used very infrequently. Since these functions were not easily implemented in Linux, it was decided that the best solution was to remove them from the API. When removing the functions, parts of the application code had to be rewritten.

7.2 Tasks

The Domino generic OS supports a number of task management functions, e.g. for creating, deleting, suspending and resuming tasks. A task can also suspend its own execution by sleeping for a specified number of milliseconds or by waiting on a child task to exit. The API functions actually refer to the tasks as processes, but in order to distinguish them from Linux processes, it was considered preferable to refer to them as tasks in this context. A task is identified by a task ID, a name, a priority and a user ID, properties which can all be retrieved using the API. Apart from the task ID, these properties can also be changed dynamically.

7.2.1 Processes or Threads

When implementing the task API, the most important design choice was whether to implement the tasks using Linux processes or threads. An investigation of the application structure

and a survey of the communication between the different tasks concluded that a process solution would require an enormous rewriting of the application. To a large extent, the tasks rely on the fact that they are executing in the same address space, and almost all communication depends on shared memory. The whole structure of the variable server, a most significant part of the system, is based on the ability to send function pointers between tasks. The variable forwarding is also performance critical, which further implies the need of shared memory. With this taken into account, converting the tasks into processes would probably not be the best solution, even if the conversion could be easily performed.

The chosen solution was consequently to map each task into a thread. However, the door was left open for the possibility to extract isolated tasks and turn them into processes at a later stage. The most obvious candidate for such a transformation was the web server. A more independent web server would also introduce the possibility to exchange the current web server in favour of a third party application.

7.2.2 A POSIX Thread Based Implementation

Having decided on mapping the tasks into threads, the API was naturally chosen to be implemented using the POSIX thread (pthread) library, which is available on all modern Linux systems.

The POSIX thread library contains the standard thread creation, termination and synchronization functions. Unfortunately, there is no support for thread suspension and hence an implementation of the suspend and resume functions of the gOS API would be very awkward. As it turns out, these functions are practically never used and they could therefore be removed from the API.

A task, as specified by the gOS API, has its own ID, name, priority, stack size, environment variables and set of open files. The task ID was conveniently implemented using the thread ID of the POSIX threads. However, some application code depended on the ID to be in a certain interval and hence needed to be rewritten. The pthread library also supports assignment of priority and stack size during thread creation. The chosen scheduling policy was `SCHED_RR`, which schedules the threads strictly based on priority and uses round robin scheduling among threads with the same priority. This policy is better than the `SCHED_FIFO` policy, because it prevents a thread from being indefinitely blocked by a thread with the same priority. Other thread-specific data, such as name and user ID, was implemented using the Thread Local Storage (TLS) facility. A thread-specific data item can be accessed from all functions and is indexed by a key, which is global to all threads.

Unfortunately, the Linux gOS implementation does not support thread-specific environment variables and open files. The support for environment variables could be implemented rather easily by utilizing thread local storage, but this has not been done since the feature actually isn't implemented in the current EOS version. Having a separate set of open files for each thread is a more complicated problem, as it is strongly related to the implementation of the C library.

To be able to redirect the standard streams for each thread independently, the input and output functions are redefined to use thread-specific streams implemented with TLS.

7.3 Semaphores

The API provides counting semaphores, which are identified by a name and an ID. The semaphores can be created and deleted by any task. The available operations on a semaphore are set and get, that increases and decreases the semaphore counter respectively, and a special reset operation, that releases all waiting tasks and reinitializes the counter.

The semaphore handling was implemented using POSIX semaphores, which is the recommended alternative for a thread based solution. System V semaphores, which are more flexible and can be used between different processes, are more heavy weight and hence slower than POSIX semaphores.

The gOS get and set semaphore functions mapped naturally to the POSIX wait and post functions, while the reset function needed a bit more work. Also, the identification of the semaphores was not straightforward. A POSIX semaphore is identified by a pointer to the semaphore struct, while a gOS semaphore has a name and a number.

7.4 Message Queues

The message queue API defined in the generic interface supports task initiated creation and deletion of queues. When sending or receiving messages, the queue is identified by a system unique number. A message can be marked as urgent, which means that the message should be put first in the queue.

In Linux, an implementation of message queues would normally either make use of pipes or the System V message queue API. However, these facilities only support a strict FIFO message handling, and hence it would be impossible to place a message first in queue. As it turns out, the old message queue implementation did not make use of any EOS specific mechanisms, and it could therefore be incorporated in the Linux API implementation without major modifications.

7.5 Time Management

The API functions available for time management provide means for getting and setting the system time, as well as creating timers. The timers are used for setting a semaphore either periodically or once after a specified time interval. A timer is just a new task, with the sole purpose to sleep a time interval and then set a semaphore.

The System V timers could not be used, since they work by raising a signal when the time interval has passed. When running several timers simultaneously, there is no way to determine which timer has expired. POSIX timers do have this capability, and the work to bring them to Linux is ongoing by the "high resolution timers project"[1]. The time management functions were implemented using the `gettimeofday` and `settimeofday` system calls, which gets and sets the time since the Epoch (January 1, 1970). The timers were implemented on top of the gOS API.

7.6 Memory Management

The memory management API consists of two functions, which are the equivalents of the C library functions `malloc` and `free`. In EOS, these functions are only used by the locally implemented `malloc` and `free`. Since these are available on the Linux platform, there is actually no need to implement a Linux version of the memory allocation API. To be compliant with the EOS behaviour, where applications in theory have the choice between the C memory functions and the gOS memory functions, simple wrapper functions have been provided.

[1] http://high-res-timers.sourceforge.net

7.7 File System

The file system functions of gOS are very similar to those of EOS. The API contains functions for searching directories, creating and deleting files and directories, renaming files and receiving file status. It also contains some features not available in EOS, such as access control. Every access to the file system is preceded by a check to control that the user ID of the task has proper access rights. In addition, every task has its own current directory, and may specify files relative to this directory.

The implementation of the file API was rather straightforward, as the Linux API contains all the needed features. The implementation of a current directory for every task would be very easy if the tasks were implemented as processes, as Linux already implements this feature for processes. However, the current directory is shared among all threads belonging to the same process. A thread-specific current directory would be simple to implement using thread local storage, but C functions like `fopen` would still use the current directory of the process. Unfortunately, this problem remains unsolved in the Linux gOS implementation. In practice, this doesn't introduce any problems since all file paths specified in the application are absolute.

7.8 System Tracing

The system trace functions are called by EOS, whenever a process is started or terminated and at every context switch. This provides a way to collect profiling information.

Since EOS and the Domino software are built together into one big executable this is the preferred way to perform profiling. However under Linux the kernel is separated from the applications and it is not desirable to make changes within the kernel. It is not possible to achieve the same behaviour under Linux without modifying the source code of the Linux kernel. However, with professional profiling tools like GNU gprof[2], there is really no need for such a feature.

7.9 Power Failure Handling

The power failure handling has been introduced because certain operation can not be interrupted without loosing important data. These operations must be able to finish their execution, before the power goes down. For this reason, the hardware generates an interrupt 10 ms before the power is lost. During this time, the critical operations should be able to finish.

The gOS API provides a way for tasks to inform the system when they are performing such critical operations. When a power failure interrupt is received, these tasks are assigned a higher priority than other tasks.

So far, the Linux implementation of gOS does not support power failure handling. This has not been prioritized in the porting process, since it is not a necessary feature to have until the other bits and pieces of the system are working.

[2] A profiler based on gprof, a call graph execution profiler [GKM82].

Chapter 8

Porting to Linux running on a PC

8.1 The Planning Phase

Before starting the porting work, it is important to know what parts of the software should be ported and in what order. In a time limited project like this, it is of course difficult to estimate an appropriate amount of code to be ported. In such a situation, it is good to identify the parts that are necessary to port, the parts that would be nice to include and the parts that are less important.

In this project, the variable server and the web server were considered to be the most important modules to port, while the trend logging and alarm handling modules were chosen to be ported in a later stage. The rest of the modules were to be ported as time admitted. The project was also restricted to deal only with functionality available inside the embedded system, and not features depending on other devices. In addition, only functionality available for the TAC Xenta 511 should be ported.

The web server was chosen to be the first module to port, partly because it is an important module and partly because it was considered relatively easy.

8.2 The Porting Process

The first step was to import the application code into the new development environment, and create a makefile. The work was then performed iteratively, adding one source file at a time. The following list describes the basic steps of the process.

1. Add a carefully selected source file to the makefile.
2. Compile the software.
3. Resolve symbolic issues for implemented APIs.
    - Write stub header files and substitute the original RTOS header files.
    - Change the makefile to include application header files in the compilation.
4. Address unimplemented APIs and data structures.
    - Rewrite the software to use functions from the gOS API or the C library instead of the EOS system calls.
    - Add referenced source files to the build script.
    - Provide stub functions.

5. Repeat step 2-4 until the software compiles without errors.
6. Debug the software.
7. If the porting is not finished, start from the beginning again.

To incorporate the latest changes in the Domino software, the Linux project was regularly synchronized with newly checked out code from the source management system. All code changes were therefore carefully documented. In order to avoid an increasing amount of changes at each synchronization, many of the code changes in the Linux project could be incorporated directly into the original project. Thus improving portability and platform independency of the original source code.

8.2.1 Header Files

Since it should still be possible to compile the code to run under EOS, just exchanging EOS header files with Linux header files was not an option. Instead, the EOS header files were replaced by new files, including the corresponding Linux header files and possibly defining needed EOS constructs.

The EOS header file `socket.h` provides an example of this procedure. This file is replaced by another file with the same name, which includes the needed Linux header files `sys/socket.h` and `sys/types.h`. Since EOS does not treat files and sockets in exactly the same way, there is a need for a special close function for sockets (`close_socket`). Under Linux, both files and sockets are closed using the `close` function, and hence the replacement header file redefines the symbol. In addition, the application uses the EOS socket errors and the EOS socket ID type, which therefore needs to be defined in the header file.

```
#include <sys/types.h>
#include <sys/socket.h>

#define SOCKET_INVALID (-1)
#define SOCKET_ERROR   (-1)

#define close_socket close

typedef int socket_id_type;

extern int get_socket_last_error();
```

Figure 8.1: Replacement header file for socket.h

8.2.2 Application Rewriting

Ideally, the software modules should only use functions from the generic OS interface, but in practice the EOS system calls are sometimes used. For natural reasons, this is especially true in the modules that originated from the EOS operating system. Since the operating system code is part of the Domino project, there is actually a blurred line between the operating system and the application.

The goal is to be able to use the same application code for both operating systems. Therefore there must not be any operating system dependencies left or introduced in the code. All

uses of OS dependent system calls must be removed and replaced by generic functions or C library functions.

8.2.3 Stub Functions

A stub function replaces an original function, in that it has the same interface, but contains as little code as possible if non at all. Stub functions have been used thoroughly during the project. First of all, stub functions have been used to act as an interface to unported modules. In many cases, these stubs have been replaced by the real implementations at a later time. Secondly, the interface to the hardware needed to be stubbed.

8.2.4 Debugging

The behaviour of the code may be different when run under Linux compared to when run under EOS. Therefore the porting process is not finished when the software compiles, it also needs to be debugged. In most cases, this is because the software contains bugs that for various reason don't appear in the EOS environment.

The Linux memory protection makes sure that many memory bugs are caught immediately, by raising either a segmentation fault signal or a bus error when an illegal memory location is accessed. The target hardware has no such protection and hence allows for all memory accesses. Most found bugs involve inappropriate reads, since they do not corrupt the system and hence can be performed without notice in EOS.

It is important to realize that not all memory related errors can be caught by the memory protection mechanisms. Although an access outside the range of an array causes undefined behaviour [HS02], it is perfectly legal from the OS perspective as long as the access is within the process' address space. Memory debugging was therefore performed using Valgrind, which is able to detect such errors and also helped to find many cases of uninitialized memory.

Some coding errors were also discovered because of timing differences. Running Linux in a virtual machine on top of Windows resulted in a timing problem, where the Linux clock was too slow and sometimes needed to synchronize itself with the Windows host. For this reason, the clock made a jump in time which made otherwise unnoticed mistakes visible.

Further complications were introduced due to the employment of the newly implemented gOS interface. Although a test suite has been implemented to ensure the correctness of the gOS implementation, part of the misbehaviour was found to be the result of gOS bugs.

8.3 Module Porting

This section describes the process of porting the different modules of the Domino platform. In total over twenty modules were ported, entirely or partially, but only the most important and difficult modules are discussed here.

8.3.1 Makefile

In the beginning of the project, a simple makefile was used. As more and more source files and modules were incorporated, a more sophisticated build system was created. The traditional method of building a large Linux project is to use recursive make, which means that the top-level makefile recursively invokes the make program on the sub-directory makefiles. However, the recursiveness might introduces a number of problems, both regarding correctness and building time [Mil98]. To produce correct results, one must make sure that the

makefiles are invoked in the correct order with regard to their internal dependencies. Often it is necessary to do more than one pass over the sub-directories to build the whole system, which leads to extended build times.

Although the size of the project does not imply that recursive make would lead to much damage, it was decided to use a single top-level makefile. To keep this makefile at a maintainable size, one makefile and one dependency file is included from each module. The module makefile contains all the building information relevant for the specific module, while the dependency file is automatically generated and contains all dependencies of the module.

This building system is superior to the building system used for EOS in several ways. First, the EOS makefile does not contain any dependency information, which makes it necessary to rebuild the entire project in order to be sure that the executable is up to date. Further, the EOS build system uses a single makefile, which is huge and difficult to maintain.

8.3.2 Web Server

Being the first part of the system to be moved to Linux, the porting of the web server was by far the most time consuming. The connections between the different modules turned out to be fairly strong, which resulted in an seemingly endless addition of source files before anything could be compiled. Not being very familiar with the software, it was difficult to identify the code pieces that were actually needed for the basic web server function. It was for example a bit problematic to remove the SSL support, since the web server is intended to always use SSL. As the code became more familiar, suitable dividing lines could be recognized and stub functions created. At first, only the file web module was ported. The other web modules were incorporated as the modules adherent to them were ported.

The time it took to port the web server was also prolonged by the fact that it was first ported without consideration of compatibility for the EOS platform. This requirement was later taken into account and the previous work needed to be redone.

When the web server code eventually compiled, the debugging phase followed. The problems were mainly memory related, but there were also a few bugs found in the gOS implementation. To test the web server, the file system of the embedded system had to be used. Apart from the actual web pages, the application needed a set of configuration files. During debugging, it was discovered that a large percentage of the HTML pages contained special tags unrecognized by the browser. These tags were supposed to be replaced by values fetched from the variable server. As this module was not yet ported, the real web site had to be replaced by a few simple pages.

8.3.3 Variable Server

Having ported the first module, the following modules were much easier. One problem during the port of the variable server module, was the handling of ASN.1 variables. To be able to quickly allocate and deallocate such variables, the variable server implements its own memory management. When a task needs to use the variable server, a larger memory area called nibble memory is allocated and used to store the variables. This way it is not necessary to allocate memory for each individual variable. The task-specific memory pointers are stored in an array, which is indexed by the task ID. This worked fine in EOS, were the task identifiers are small numbers. The Linux thread identifiers can assume any value, which makes it impossible to use them as indexes. The memory pointers were instead saved in the thread local storage provided by the pthread library.

8.3.4 Trend Logging and Alarm Handling

When moving the trend logging and alarm handling modules, timing became an issue. Because of a faulty implementation of the C library function `clock`, these two modules behaved in a very strange way. Once the problem was identified, it could be solved rather quickly. The timing problems caused by VMware also makes the trend logs look a bit peculiar, but this is only a problem when running in an emulated machine.

8.3.5 The Domino Shell

The Domino shell (DSH) constitutes the command line interface of the Domino platform. The Linux counterpart is of course a Linux shell, but to interact with the Domino application it must be possible to issue DSH commands from the Linux command line. One solution is to start the Domino shell as an application, but then the possibility to alternately issue DSH commands and Linux commands would be lost. It would also be impossible to utilize pipes and file redirections. For these reasons, it was decided that Linux required a partially separate implementation of DSH, where the commands were issued independently.

In the original implementation the user communicates with the Domino shell through the serial port. The implementation for Linux uses sockets instead, which makes it possible to handle an arbitrary number of DSH users. The DSH task, running as a thread in the Domino process, acts as a server and waits for clients to connect. To be able to handle several simultaneous DSH users, a new thread is started for each new client.

A Linux user who wishes to issue a DSH command starts the DSH client program, which connects to the server and sends the command. In order to know what privileges the DSH command should run with, the server first requires the client to authenticate itself with a username and a password. This of course refers to the Domino user information and not the Linux username and password. Before the command is started, the user ID is changed and the standard streams are redirected to the socket.

To make the issuing of DSH commands as easy as possible, one symbolic link is created for each DSH command. All links point to the DSH client program (`dsh`), which looks at the way it was invoked and sends the corresponding command to the server. This way the DSH commands can be started just as easily as the usual Linux commands. There are however DSH commands, such as `date`, which conflict with Linux commands. This problem is solved by starting the client program directly, providing the command name as an argument (e.g. `dsh date`).

It is quite tedious to provide the username and password for every DSH command and to avoid this, the login information is saved to a file the first time a DSH command is issued. It is still possible to log in as a different Domino user at a later time. By starting the `dsh` program without any arguments, the user can update the stored username and password. To avoid the security risk of sending plain text passwords over the network, the DSH commands can not be invoked from another computer. Instead, one must log in to the system using SSH and issue the commands from there.

In addition to the migration of the DSH platform to Linux, the actual DSH commands needed to be ported. Apart from a set of system commands, every module contributes a number of commands applicable to that module. Certain commands were not ported, since it exist equivalent commands in the Linux environment.

8.4 Porting Issues

8.4.1 CPU Scheduling

For a real-time application, the way the CPU scheduling works is very important. In EOS, there are two different scheduling policies. High priority tasks are scheduled strictly based on priority, while low priority tasks are scheduled according to a round robin policy. In Linux, the scheduling policy can be specified for each individual task. However, both real-time scheduling policies strictly assign CPU time to the task with highest priority. The policies only differ in the way they schedule tasks with the same priority. Since the Linux scheduling method is actually a bit stricter than that of EOS, this difference does not constitute a problem.

Also, the priority range differs between the two operating systems. While Linux priorities range from 1 to 99, EOS only manages priorities between 1 and 15. Because all EOS priorities are inside the Linux priority range, this difference causes no problems.

8.4.2 Root Privileges

As real-time scheduled processes are able to preempt any other process in the Linux system, only root processes may activate the real-time scheduling policies. In order to schedule the tasks based on priorities, the Domino process therefore must run with root privileges. This is also necessary due to the web server functionality, which makes use of port 80. Only root processes are able to open ports below 1024.

8.4.3 C Library Functions

Part of the EOS C library functions were developed locally, for use only by the Domino software. Some of these functions behave differently than their Linux counterparts, which causes problems when the application depends on the deviating behaviour.

Some of the locally implemented C functions make use of a task-specific data structure which is part of the generic interface. This structure holds information regarding the open files and the environment variables of each task. It also contains function specific data, to turn functions that normally use static variables into thread-safe functions. With the use of the GNU C library, these accommodations are lost. In Linux, all threads within a process share open files and environment variables. Since the tasks are implemented as threads and not processes, it is not possible to achieve the behaviour of the original system without rewriting the library functions. This has not been done, since it was decided not to be worth loosing the comfort and safety of the functions in the GNU C library.

One of the library functions that have been implemented as a reentrant function for the Domino platform is `strtok`. In the GNU C library, the `strtok` function is not thread-safe, but there is a reentrant version called `strtok_r`, which takes an extra parameter. A possible solution to the problem would be to rewrite the application code to use the reentrant version. Because the function was used very frequently, the problem was instead solved by overloading the C library `strtok` function with a reentrant version using TLS.

Another diverging function is `clock`, which returns the value of the real-time clock (the number of ticks since system start-up). The correct behaviour is to return an approximation of the CPU time used so far by the current process [KR88]. As the application expects the faulty behaviour, and this behaviour happens to be the same as the behaviour of the gOS function `gOSGetTime`, all uses of `clock` were replaced by uses of `gOSGetTime`.

8.4.4 The Socket API

With a few deviations, EOS sockets are compatible with the BSD socket API. When porting applications from EOS to Linux, which has a fully BSD conforming socket API, these deviations become important.

Often the difference between EOS and Linux is that EOS lacks some feature that Linux provides, which constitutes no problems during the port. For instance, EOS does not provide full integration of sockets and files and hence file I/O functions can not be used to operate on sockets. For this reason, EOS provides two special socket functions, `ioctl_socket` and `close_socket`, to replace `ioctl` and `close` when used on sockets. These functions need to be redefined under Linux.

The `select` function, which waits for a number of file descriptors to change status, has a parameter that specifies the highest-numbered descriptor to be watched. This parameter is ignored in EOS, which always considers all socket descriptors. Hence, calls to this function often have this parameter set to zero, which means that the function has no effect in Linux. The easiest solution is to change the function calls to pass the maximum integer value instead of zero.

While Linux makes use of the global variable `errno` to communicate error codes to the calling process, EOS provides a function named `get_socket_last_error`. Since the error codes are the same, the function could be easily implemented.

Finally, the EOS socket identifiers are of the type `socket_id_type` while Linux uses the `int` type. The `socket_id_type` is actually defined as an `int` in EOS, and this could simply be done in Linux as well.

8.4.5 Case Sensitivity

Unlike Windows file names, Linux file names are case sensitive. The development of the Domino project has been carried out in a Windows environment, and hence the developers have not needed to worry about case when specifying file names. One effect of this is that the include statements do not match the actual header file names with regard to case, and therefore cause compilation errors under Linux.

A solution to this problem is of course to change all include commands into specifying the file names correctly. To avoid this tedious work, instead a script was created purposed to change all filenames to lower-case letters and to modify the include statements accordingly. This was the easiest way to solve the problem, and it also helps to prevent future mistakes. From now on, file paths are written in all lower-case letters.

A related problem is that Windows uses backslash (\) to separate directories in file paths, while Linux uses slash (/). The current development environment allows for both symbols in the include statements, and hence backslashes can sometimes be found in file paths. Changing the backslashes to slashes was also performed by a script.

8.4.6 File System Placement

In the EOS file system, the Domino files are stored directly in the root directory. This is very impractical in Linux, since the files become mixed up with the Linux files. Also, the Domino file system contains directories, such as the `sys` directory, that conflict with Linux directories. Consequently, the Domino file system needed to be stored in a sub-directory.

Changing the file system path to all the Domino files introduced a new problem, since all file paths in the application were absolute. To simplify the porting process, TAC changed their implementation and specified all file paths in a single file. By defining a macro, the Domino root directory could be specified. Unfortunately, this root directory was also added

to partial paths and the URIs used for the comparisons that decide which web modules are to handle an incoming request. Although such mistakes could be easily corrected once discovered, this was a continuous problem throughout the project.

The placement of the Domino root directory further down in the file system hierarchy, led to a lengthening of the file paths. This resulted in unpredictable behaviour caused by buffer overflows. When overflowing a stack allocated buffer, there is a risk that the return address is overwritten. The problems were naturally solved by increasing the buffer sizes.

8.4.7 Assembly Routines

To achieve better performance, a few extensively used functions have been written in assembly language. Since the ARM assembly code obviously can not be used on the Intel processor, the code was rewritten in C. The performance loss is not critical, since the Intel processor is much faster than the ARM processor. When moving the software to the target hardware, the assembly routines can be utilized again.

8.4.8 Hardware Dependencies

The PC uses little endian byte ordering while the ARM processor uses big endian byte ordering. The byte ordering is mostly important when doing external communication. This is done through the BSD socket interface, which provides functionality for making sure that addresses and ports are in the correct byte order. Sine EOS was written with both little and big endian compatibility in mind, the porting to Linux on x86 caused almost no problem with respect to endian issues (only one function had to be modified).

Because of the memory protection offered by the MMU, memory cannot be directly accessed on the PC. In the original software, the non volatile memory of the embedded system is directly accessed for storing important data. In the PC version this memory had to be emulated by first allocating the same amount of dynamic memory, and then providing a pointer to the allocated memory. When the application exits, the volatile memory is stored on disk in order to load it in to memory again at startup.

The embedded device uses flash memory for storing the file system whereas the PC uses a hard drive. A flash memory has a limited number of erase-write cycles, which means that one should not write to flash if it is not necessary. Therefore parts of the Domino file system which contains temporary data are kept in RAM. Since the hard drive does not suffer from this limitation, all parts of the Domino file system was put on hard drive when developing under Linux.

Chapter 9

Moving to uClinux running on the Target Hardware

9.1 C Language Libraries for Embedded Linux

On a standard GNU/Linux installation the most common C library is the GNU C library (glibc). This library has support for multiple operating systems and also contains some exotic features that are seldom used.

In an embedded system where every single byte of memory is valuable, glibc is not the optimum choice in terms of memory consumption. There are several other C libraries that can be used instead, and the most popular for embedded GNU/Linux today is uClibc. The goal of uClibc is to minimize the size of the library, which is achieved by only targeting GNU/Linux systems and eliminating redundant code.

In order to work as a drop-in replacement for glibc, uClibc uses the glibc header files. Since uClibc only contains a subset of the functions included in glibc, the header files might declare functions that are not actually implemented.

The library can be used on platforms with and without MMU. Since the target platform of this project lacks MMU, uClibc has been compiled without MMU support. This means that some functions are not available, of which the most important is the `fork` system call.

9.2 Porting uClinux

The uClinux/ARM 2.6 project has provided support for the SoC that the target hardware is built around. This means that support for the serial ports and the Ethernet interface is provided. The SoC itself can handle access to the external interfaces in both big and little endian mode, but the hardware setup only allows for big endian mode. Unfortunately, only little endian mode is supported by the uClinux/ARM 2.6 project and hence minor changes had to be made in the serial and Ethernet drivers. The rest of this section describes a few other target specific issues that were found during the work of porting uClinux to the embedded system.

9.2.1 The Serial Port

It was discovered that the serial console has a peculiar behaviour. A debug console is opened at `ttyS1`[1] in the initial boot sequence. This is forced to 19200 bps and apparently no attempt

[1] The second serial port.

is made to see if this device is already in use. The console on our setup happens to run on the very same interface, which forced usage of 19200 bps for the console as well and not 9600 bps as for EOS. The behaviour was not changed in the Linux kernel since we wanted to keep changes against the standard kernel to a minimum.

9.2.2 Memory Alignment

On the ARM platform memory accesses must be 32 bit aligned. A non-aligned memory access results in a trap, which can be caught by a routine in the kernel. This routine is called an *alignment trap handler* and is optional to include in the kernel, since it may have impact on performance. If an alignment trap occurs and the system doesn't have the handler installed, the kernel will print out a warning message with debug information and the application or the whole system will most probably malfunction. In our specific case the only alternative was to not include the alignment trap handler, since it contains inline assembly utilizing the MMU. As earlier pointed out, our system does not have an MMU.

However running without the trap handler should generally not cause any trouble, since the compiler should make sure that memory accesses are aligned.

9.2.3 Flash Memory

The target hardware includes 16 MiB of NAND flash memory which is used by EOS to host the file system with the Domino application and associated data files. NAND flash memory differs from NOR flash memory in that it can't be directly mapped to memory addresses. It has to be accessed sequentially through a driver, about the same way as a hard drive.

Fortunately Linux includes support for many different NAND flash chips through the Linux Memory Technology Devices project[2]. In order to compile the flash support under the ARM architecture, the alignment trap handler must also be included in the kernel. So unfortunately, the flash memory does not run with our SoC under uClinux without reimplementing the alignment trap handler.

9.2.4 NFS

Since the flash memory was not available, the root file system of the embedded device had to be stored elsewhere. The amount of dynamic memory in the system was not enough to host both the file system and to provide space for the executing applications, so another solution had to be found. An NFS (Network File System) server was set up on a standard Linux desktop PC and the target device was configured as an ordinary NFS client.

9.3 Running Domino on uClinux

Once a fully functional uClinux system was up and running, the Domino application could be ported to the target hardware. From an application's point of view, the differences between a standard Linux system and a uClinux system are very small. Since it was known in advance that the application should eventually run on uClinux, the porting to standard Linux was done without using functions unavailable on uClinux. Due to this fact, the application could run on uClinux without modifications.

When compiling the Domino application for uClinux, different optimization levels were tested. It was observed that there was no significant change in size of the executable file when

[2] http://www.linux-mtd.infradead.org

using the second level of optimization compared to when compiling with the "optimize for size" flag. For this reason it was decided that the application should be compiled with the second level of optimization in order to make the execution as efficient as possible.

An overview of the memory usage in EOS and in uClinux is presented in figure 9.1. Since EOS and Domino are compiled together, it is not possible to see the amount of memory used by each component.

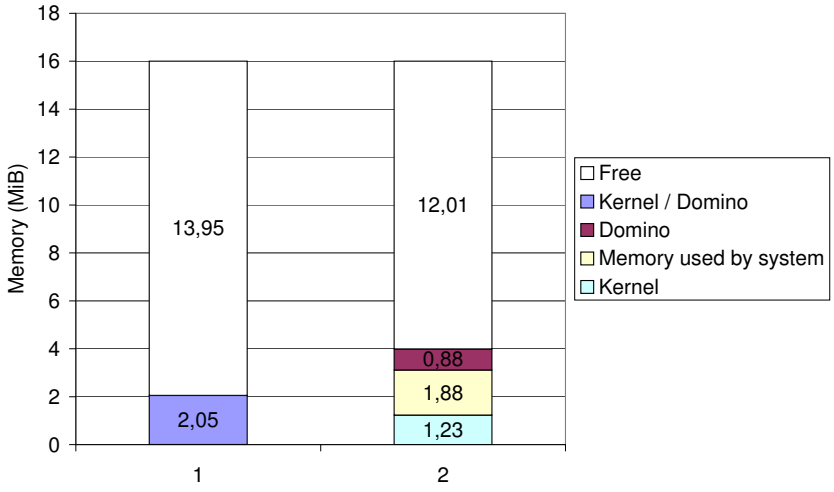

Figure 9.1: Memory usage

It can be observed that the uClinux system uses almost twice as much memory as the EOS system. The largest part of the allocated memory is used by system processes, including system buffers and caches as well as the shell and other utilities. The difference between uClinux and EOS is not large enough to cause any problems when running the Domino application. The remaining 12 MiB of free memory is definitely enough to handle the dynamic memory consumption of the application running under normal circumstances. The configuration used during this project utilizes on average 2 MiB of dynamic memory.

It should also be considered that the Domino application of the uClinux system does not include as much functionality as the EOS version, since not all software modules have been ported. The source files used for building the Domino binary for uClinux contains about 130 000 lines of code, while the source code compiled for EOS contains 350 000 lines (not counting the operating system code). This implies that the entire Domino executable should consume over 2 MiB of memory, when all modules have been ported. When reaching such sizes, it might be better to use an optimization level focused on reducing the code size.

Part III

Evaluation

Chapter 10

Real-Time Performance Measurements

10.1 Benchmarks

To be able to evaluate the performance of the uClinux system, a benchmark suite containing a number of small test cases has been created. The same benchmarks are run on both uClinux and EOS to collect measurements that can be used to compare the two systems.

The benchmarks can be divided into two categories, which focus on performance at different levels. The system performance tests evaluate system functionality such as the TCP/IP stack and the gOS benchmarks test the implementation of the gOS API.

When comparing the performance of two systems, it is important that the testing conditions are as equal as possible. In the comparison between EOS and uClinux there are a few differences that may affect the fairness of the comparison. In EOS the file system resides on a flash memory, but in uClinux it resides on a network file system. The amount of work needed for accessing the file systems can not be considered as equal.

Timer interrupts in the uClinux/ARM 2.6 kernel are by default generated at 100 Hz, but EOS generates timer interrupts at 1000 Hz. In order to make a fair comparison between EOS and uClinux, the timer interrupts were raised to 1000 Hz in uClinux. Although the timer generates interrupts at 1000 Hz in EOS, the internal clock was only updated at 100 Hz intervals. This rate was also adjusted to 1000 Hz, in order to get the same time granularity in both systems.

In order to measure the performance the two systems will achieve when they are in use, the benchmarks were compiled with the same optimization levels that are used when compiling the Domino application. In the case of uClinux, the O2 optimization flag was passed to the GCC compiler. The Norcroft ARM C compiler was run at the default level, without additional optimizations.

10.1.1 System Performance

The system performance benchmark consists of three tests, measuring context switch overhead, memory allocation performance and network communication speed. To be able to run the tests on both EOS and Linux, task creation and IPC are done using the gOS interface.

The context switch benchmark measures the time it takes to switch between a number of tasks. This is done by letting each task wait on a separate semaphore, which is initially set to zero. The chain is started by increasing the value of the first semaphore. When a task is released, it sets the semaphore of the next task in line. The communication overhead is calculated and subtracted from the measured time.

The memory allocation test iteratively allocates and releases memory blocks of various sizes, using `malloc` and `free`. The test is written to produce memory fragmentation.

Network communication speed is measured by sending data blocks of increasing size through a socket set up between two tasks. By using the loopback interface, the TCP/IP stack can be tested without interference from the external network. The same test is also performed using an external client program, which receives data from the benchmark program. When sending data over the loopback interface several different send and receiver buffer sizes were used.

10.1.2 gOS Performance

The performance of the gOS functions is crucial for the performance of the application, because all operating system specific mechanisms must be accessed through this interface. Fast underlying system calls are not useful if they cannot be effectively utilized through the gOS API. The gOS benchmark measures the following operations.

1. **Tasks**
   (a) A new empty task is started and terminated.
   (b) The ID of the current task is obtained.
   (c) The name of the current task is obtained.
   (d) The name of the current task is changed.
   (e) The priority of the current task is obtained.
   (f) The priority of the current task is changed.
   (g) The user ID of the current task is obtained.
   (h) The user ID of the current task is changed.
2. **Semaphores**
   (a) A semaphore is created and immediately deleted.
   (b) A set and a get operation is performed on a semaphore.
3. **Timing**
   (a) The current time is obtained.
   (b) A task is started with the purpose to iteratively sleep for a predefined amount of time. Every time it wakes up it measures the delay between the correct and the actual awakening.
   (c) The above test is carried out while the Domino application is running and heavily used. The timer task is assigned the highest priority level.

Message queues are not tested since they have the same implementation in both operating systems. Instead of using operating system dependent IPC facilities, the implementation utilizes gOS semaphores. Any message queue measurements will therefore only reflect the performance of the semaphore API implementation.

File system functions are not tested either, because the unfairness in the comparison was considered to be too high.

10.2 Results

10.2.1 System Performance

The results of the system performance tests were quite different for the two operating systems. The most alarming result was the big difference between the calculated context switch times. Depending on the number of tasks, EOS was 14 to 30 times faster than uClinux. The only positive observation was that the percentage difference between the two systems seemed to decrease with the number of tasks.

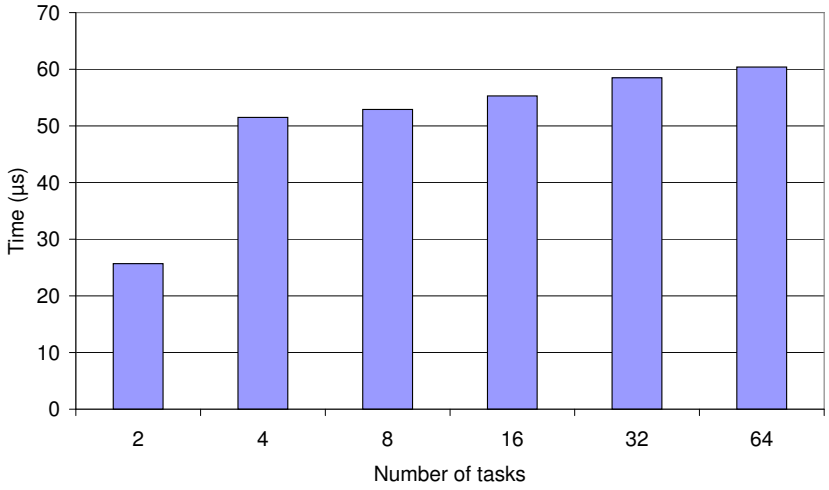

Figure 10.1: EOS context switch overhead

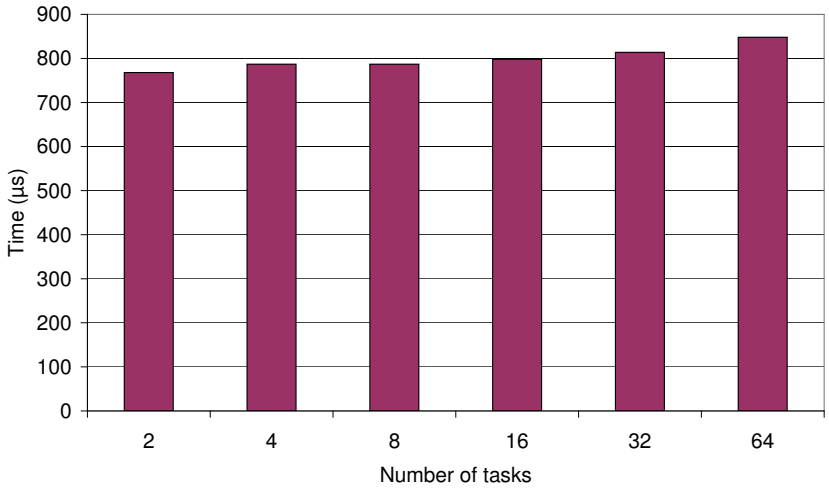

Figure 10.2: uClinux context switch overhead

Memory allocation was also done faster in EOS, although the difference was not quite as large. The test was completed about 25% faster when run under EOS compared to when run under uClinux.

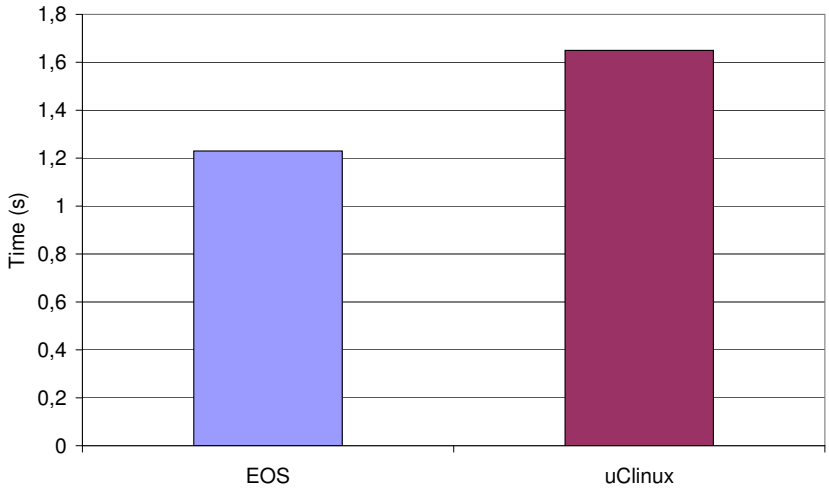

Figure 10.3: Memory allocation

The network communication test on the other hand turned out better for uClinux. Only the best result for each platform is presented, as the difference between the platforms was of far larger magnitude than the differences within each architecture. Using the internal loopback interface, uClinux reached a speed of 2.85 MiB per second. In the external network test, the speed was limited by the hardware. The result of 1.18 MiB per second is near the maximum communication speed of 1.19 MiB per second, which can be achieved on a 10 Mbps network interface.

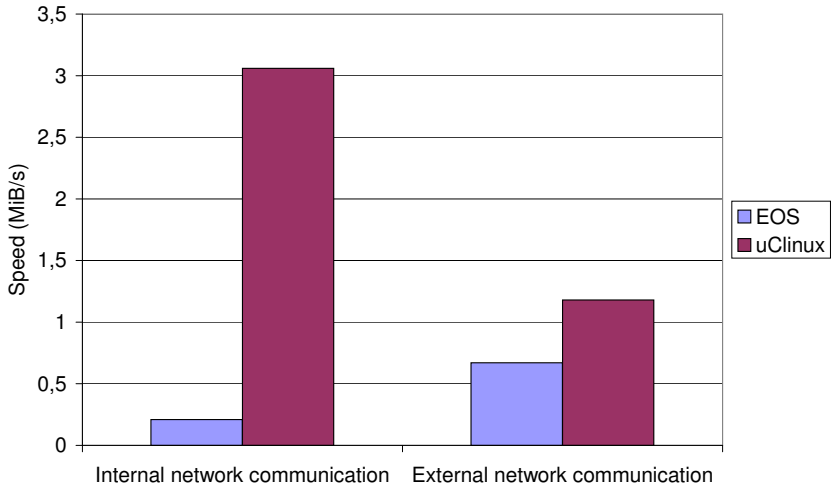

Figure 10.4: Network communication speed

The network capabilities of EOS were not very impressing, showing an external communication speed just over half of the maximum speed. A surprising result was that the internal communication speed was actually lower than the external.

10.2.2 gOS Performance

Most of the gOS benchmarks did not reveal any significant differences between the two systems. One of the results worth commenting, was the time it takes to start a task and await its termination. This was more than four times faster in EOS.

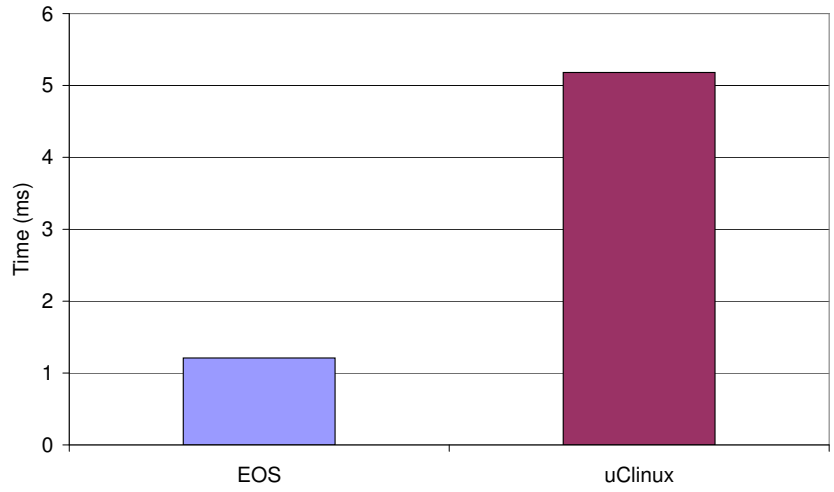

Figure 10.5: Task creation and termination

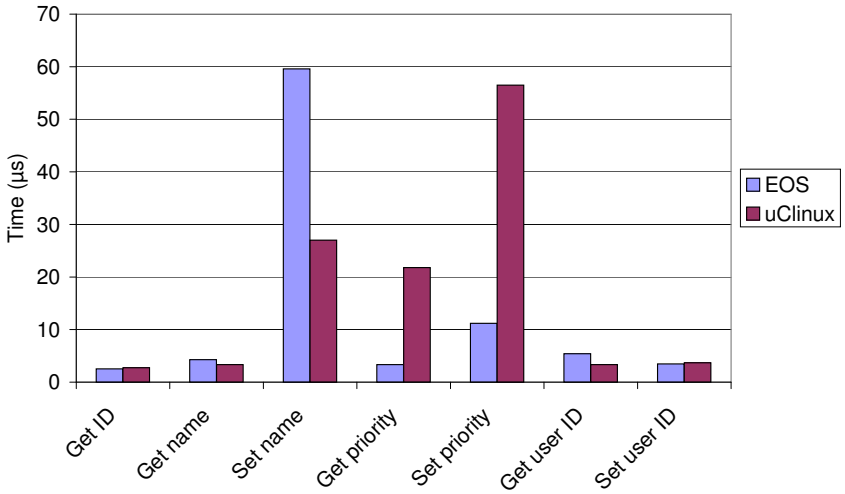

Figure 10.6: Task operations

Most other task operations were performed in about the same amount of time in EOS and uClinux. The priority operations were considerably slower in uClinux though, but this is of small importance for the Domino application. Firstly, the uClinux times are not that bad and secondly the operations are seldom used. Another significant result was the time it takes to obtain the name of the current process, an operation that uClinux performed much faster. Since this operation is also rather uncommon, the result is not very important.

The semaphore operations were performed faster by the uClinux system. The set and get operations were only slightly faster, but semaphore creation and deletion were over six times faster.

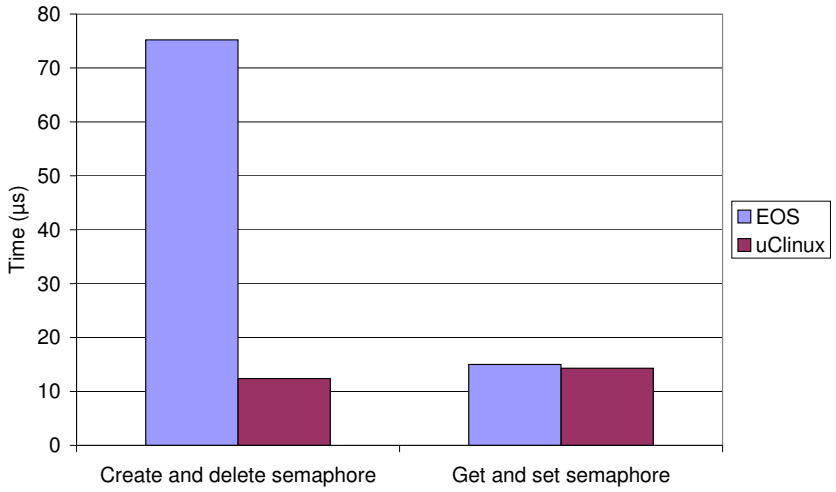

Figure 10.7: Semaphore operations

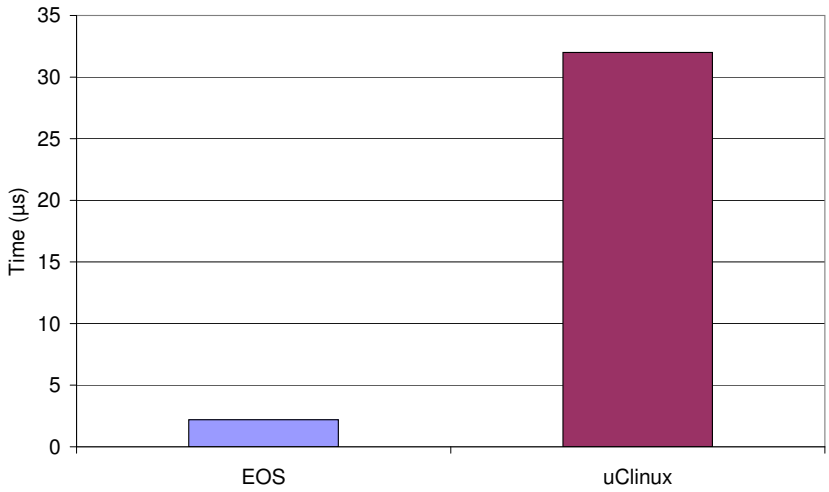

Figure 10.8: Obtaining the current time

Being a real-time operating system, EOS was superior to uClinux in the timing benchmarks. The current time was obtained much faster in EOS, and the timer tests performed perfectly. The sleeping time was set to 500 ms, and the EOS timer was always right on time. After increasing the frequency of the timer interrupts in the Linux kernel, uClinux also managed to provide acceptable results. However, although the uClinux timer in most cases showed small deviations, the delays sometimes reached alarming levels. This was particularly evident when running the timer task together with the Domino application, and thereby increasing the system load. The diagrams show the delay between the correct and actual awakening. It can be observed that uClinux is always at least 1 ms late.

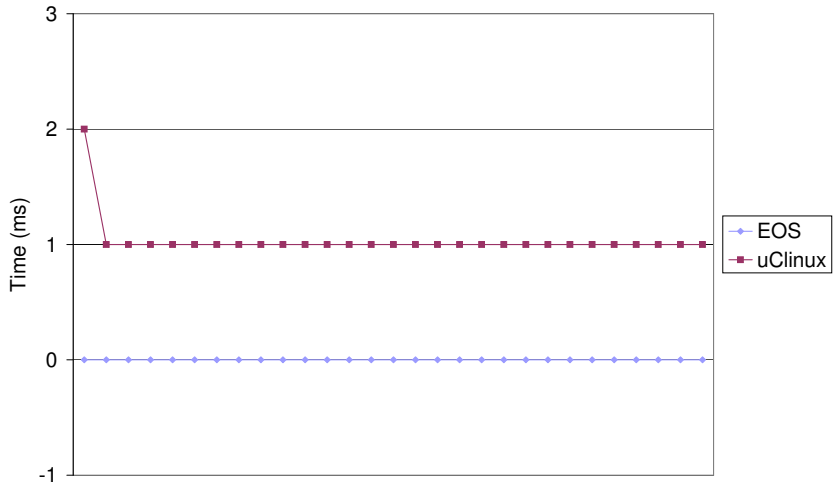

Figure 10.9: Timer delay with no load

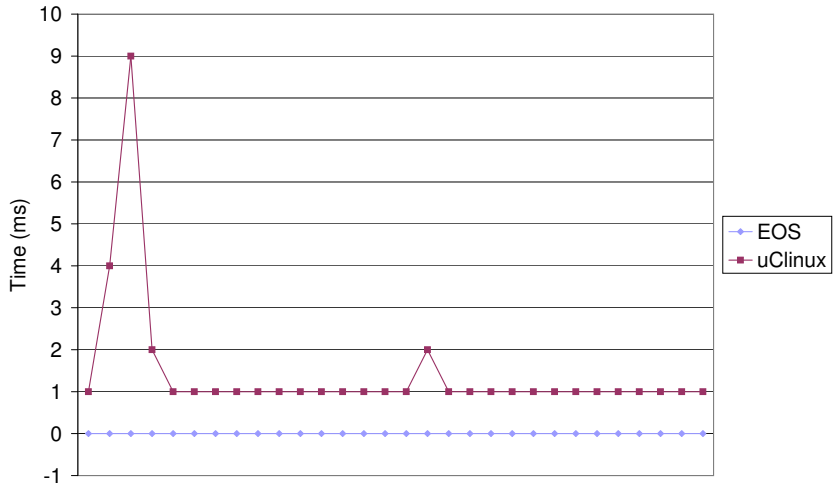

Figure 10.10: Timer delay while Domino is running

Chapter 11

Discussion

11.1 Development Environment

The choice to run Linux as the host operating system in a VMware virtual machine initially sounded like a really good solution. It provided the valuable possibility to work with multiple operating systems in a convenient way. Unfortunately, the performance of the virtual machine proved to be rather poor, even when run on a fast computer. It was particularly slow when using Eclipse, which runs inside a Java virtual machine (JVM).

Perhaps a better solution would be to install Linux as the default operating system and run Windows in a virtual machine if needed. This way computation intensive work, such as compiling, would be performed on a real machine. Another option is to install both operating systems on the same machine, and decide at boot time which OS to use. However, with this alternative the easy switch between the two operating systems is lost. Finally, one may of course install Linux and Windows on two separate computers.

Apart from the slowness, the work with the Eclipse platform has been an overall good experience. The framework as such was found to be very feature rich and easily customizable. Not all of the plug-ins have reached a satisfactory maturity level though. Instability problems have been observed in both the C/C++ plug-in and the Subversion plug-in. As the Eclipse development projects are very active, the future still looks very promising. During the work with this project, new releases have become available for both the C/C++ development plug-in and the Eclipse platform.

Even though the Eclipse Subversion plug-in showed a few problems, the Subversion revision control system lived up to the high expectations. One of the most important improvement compared to the old SourceSafe repository was the possibility to access the Subversion server from any operating system and even through a web-based interface. TortoiseSVN, the Windows Subversion client, was found to be a very good and easy to use tool.

The GNU tools for compiling and building software, i.e. GCC and GNU Make, were used with total satisfaction. However, troubles arose when debugging the threaded application using the GNU Debugger. There were situations where the program behaved totally different when run in the debugger compared to when run outside the debugger. Although the problems could be circumvented, they could never be explained. Also, the use of the `vfork` system call seemed to mess up the debugging.

Debugging on the embedded device was performed with the Lauterbach Trace32 in-circuit debugger. When debugging uClinux, Trace32 was hosted under SUSE Linux and when debugging EOS, Trace32 was hosted under Windows XP. The Trace32 system is very helpful when debugging both uClinux and EOS. However it seems like the Windows version is a little bit more mature than the Linux version. The only real problem arose when debug-

ging a specific thread, a task switch occurred and another thread misbehaved resulting in a crash. This caused some headaches, since the Trace32 tool would still present the part of the code that was debugged before the task switch and one could not be sure about which thread that was causing the crash.

11.2 API Implementation

The most important design question when implementing the operating system API was to decide whether to map the RTOS tasks to processes or threads. Given the application properties and the amount of time available, it was concluded that threads were the only realistic choice. Considering thread disadvantages like concurrency complexity and the lack of memory protection, it can be discussed if processes would be a better option given enough time. However, with the extensive use of shared memory this would result in a total reimplementation of large parts of the Domino software. Since the migration to processes also comes with drawbacks, such as lower communication speed between tasks, this alternative is strongly discouraged. It is important to remember that this conclusion is based on the properties of the specific application, and that another conclusion may be drawn in a different porting project.

The current API implementation does contain a number of unsolved problems. The most difficult problem is probably the thread-specific set of open files, which enforced a rather awkward solution for redirection of standard streams.

Another unsolved problem is that there are no access permission controls when trying to open files through the C library. The lack of a thread-specific current directory is less important, since only absolute paths are used. These two problems could be solved by introducing wrapper functions for the library functions used to access files. This could be done in one of two ways. Either the functions are overloaded or a number of new gOS functions are added. By overloading the functions, the application code does not need to be changed. However, by using gOS functions it is clearer to the programmer that the library functions are not called directly.

11.3 Porting to Linux running on a PC

A variety of difficulties were discovered during the work of porting the Domino software to a Linux PC. Some of the problems are specific to this project, while others are likely to be encountered during most ports. Though satisfactory solutions often could be found, some problems could gain from a discussion on alternative solutions.

When porting a portion of the software, it is difficult to decide where to place the delimiters. A relatively large amount of the time may be used to separate the selected portion from the rest of the software. To avoid this work, one could try to port the entire application all at once. Because of the time constraints this was not a possible alternative in this project, and it would probably not be a good idea anyway. By starting with a small selection, the porting process can be completed faster and possible problems are encountered at an earlier stage.

The decision to place the Domino file system in another directory than the root directory introduced countless problems. The complications are mostly caused by the fact that the URLs and the file paths no longer are identical. The URLs should not be affected by the placement of the file system. To solve the problem, the path to the root directory was added to the paths. Since no natural delimiter could be found where this addition could take place, this is done at numerous places in the code. A better solution to this problem is therefore needed. The easiest way is of course to actually place the file system in the root directory, but this

alternative is not recommended. It is important to make a clear distinction between the operating system and the application and not mix the files with each other. A second approach is to introduce wrapper functions for the file access functions as previously mentioned and to place the addition of the root directory there.

11.4 Moving to uClinux running on the Target Hardware

The porting of the Domino application from Linux to uClinux was not difficult, since the APIs are nearly the same. The difficulty was instead to get uClinux running on the target hardware. The lesson that can be learned is that if you are looking for a solution that is just to click-download-install-run, you should run little endian. Most precompiled cross-compiler suites and most of the default configurations in the uClinux distribution assume that the target runs little endian. It takes a little more time to get everything to run big endian, but it is by no means impossible.

The main drawback with the uClinux system is that the flash memory could not be utilized, which forced the use of NFS for hosting the file system. This is a convenient solution during development, since the files don't have to be downloaded to the target hardware every time a change has been made to the source or the configuration. However, it does not work as a final solution for the embedded system, which needs to store the files locally. To get the flash memory to work, an alignment trap handler that don't use the MMU needs to be implemented. This is a rather time-consuming task and could therefore not be done during this project.

Another issue that can be discussed is the quality of the uClinux kernel code for our specific target. One critical bug was found in the timer code, which made the system clock run erroneously. A fix for this bug was created and sent to the maintainers. Due to the presence of this bug, it can be assumed that the code has not been extensively tested.

11.5 Real-Time Performance Measurements

The results of the performance tests clearly showed the strengths and weaknesses of the two operating systems. While EOS was superior when it came to timing accuracy and task management, Linux proved its excellent networking capabilities. In an overall judgement, it needs to be said that EOS showed better performance than uClinux. However, this is only to expect when comparing a real-time operating system with a general purpose operating system. A few attempts will be made to try to explain the results.

A possible reason for the bad results achieved in the task creation and context switch tests may be a poor thread library implementation. uClinux uses the old LinuxThreads library instead of the new NPTL implementation, which has been proved to be up to three times more efficient [tim04]. By running additional tests on the uClinux platform, it was actually discovered that context switches between processes were almost three times faster than context switches between threads. However, this can only partly explain the great difference between EOS and uClinux. The results are also a bit unexpected as a study made at Samsung [CY05] showed that uClinux has very good context switching performance on the ARM9 processor. Even if the measurements were done on different hardware, the results differ surprisingly much. The exceptionally good results of EOS in these two tests are understandable, since EOS is a real-time operating system designed to perform such operations very fast.

The timer tests also clearly show the difference between Linux and an RTOS. While the EOS timer is correct every time, delays of varying sizes are seen in uClinux. The most disturbing observation is not the average delay on about 1 ms, but the great variation. Delays

as large as 9 ms were observed.

Operating system functions are generally slower on Linux than on EOS, because Linux makes use of system calls. A system call is a routine that is executed by the kernel to accomplish something on behalf of the calling process [MOS01]. The procedure of transferring control to the kernel and than back to the process introduces substantial overhead. In EOS, operating system functions are no different than ordinary function calls and hence there are no real system calls. This performance difference helps to explain why some functions, such as the function for obtaining the current time, are slower on Linux.

The higher network performance in uClinux on the external interface is to large extent explained by the lack of memory copies in the driver for the specific NIC. The NIC automatically fetches data from a ring buffer in RAM. Instead of copying data into this buffer, Linux changes the addresses to the different segments in this buffer, so that it points to the data received from the upper layer. This means that the device driver is also responsible for freeing the allocated memory in the buffers when transmission has completed.

In EOS the send buffer is fixed, and data that is to be sent is copied into the send ring buffer within the device driver. This means that EOS stresses the memory bus much more than uClinux for the same amount of data transmitted out on the network.

Chapter 12

Conclusions and Further Development

12.1 Conclusions

The work performed during this thesis has showed that a migration of the entire Domino software to Linux is absolutely not an impossible task. During this project the core components of the Domino software were ported and thereby the main transition problems were discovered and solved. During the porting of the remaining parts, the new problems found will most likely not affect the whole software as the earlier problems did.

When juxtaposing EOS and uClinux, it can be concluded that the migration to Linux undoubtedly has an impact on performance. However, the decrease in performance is not severe and the real-time properties will most likely be good enough for the Domino application. Nevertheless, an application instance that runs at the limit of what the EOS platform can handle will probably not be able to work without glitches on uClinux.

12.1.1 Recommendations for Future Linux Ports

The aim of this thesis was to provide a basis for anyone attempting to migrate from an RTOS to embedded Linux, and not just to find the difficulties in this specific porting process. Therefore a few recommendations for future Linux ports will be presented.

First of all a thorough investigation must be performed, where the nature and requirements of the application are carefully evaluated. Then it is time to consider how much the system and the development process can benefit from moving to Linux and which possible drawbacks exist. Will the drawbacks be of minor or major importance and can there be something done to reduce their influence?

When initiating the porting, it is strongly recommended that the target is a standard desktop computer running Linux. Running on a desktop computer will ease debugging significantly, due to the various tools available for desktop computers. Since there undoubtedly will be platform specific issues to take care of with respect to memory handling etc, the use of good debugging tools will greatly impact the progress rate of the porting process.

It is important that people with knowledge of the whole system participate in the porting process, at least for providing guidance when outlining the porting strategy. Changing the behaviour slightly in one software module may break functionality in another, and this may not be noticed immediately. It is therefore important to discuss changes and their impact before going through with them.

To simplify porting process all tasks should in the initial stage be handled as threads. When the entire application is ported and runs stable, an overview of the tasks can be done, where isolated tasks are identified and turned into processes.

During the API implementation work, find out which RTOS features the application actually utilizes and to what extent. Only implement the features of the RTOS API that is used by the application. Seldomly used functions should be investigated and their use should be considered. Perhaps the code calling the function can be rewritten in another way or maybe it already exists a more common function with the desirable behaviour? The goal should be to remove seldomly used exotic functions with more common ones in order to keep the API small and consistent.

12.2 Further Development

The work started in this thesis will continue at TAC, and hopefully eventually result in a commercially available Linux based TAC Xenta 511 product. The obvious first task during this further development is to port the remaining modules of the Domino software to Linux. During this process it is desirable to first have a look at the whole system and identify parts of the software that should be redesigned in order to provide an OS independent solution. Decreasing the OS dependency will make it easier to target Linux and EOS from the same source tree, without introducing a lot of "hacks".

When the entire system is ported, the different tasks should be evaluated in order to find tasks that might be converted into processes. One reason for converting some of the tasks into processes is that it will provide a stricter line between the different modules in the software, which will make it easier to replace a module with third party software. Memory corruption will also be easier to track down when running the software in separate processes, since processes can not access each others' memory areas.

There is also a desire to create a unified build system for EOS, Linux and uClinux. Maintaining several build systems makes it more difficult to verify that the different build systems are coherent with each other and that they produce the same result. Moreover, the current EOS build system does not maintain dependency information, which is a most wanted feature.

Migrating to Linux opens up a whole new world of available software, often with open source licenses. This can be utilized in the Xenta 511 product, for instance by exchanging the EOS FTP server with a properly maintained third party FTP server. Another module that may be desirable to replace is the EOS SSL module, for which OpenSSL[1] is a good alternative. The reason for this is that since EOS is no longer maintained, there will be no further upgrades on the EOS SSL software and thus no security fixes when attacks against the ciphers used in older SSL implementations are found. Using a well maintained open source library, the application will be able to handle the new ciphers needed for providing a strong secure channel in the future.

[1] A free well known open source SSL library, http://www.openssl.org.

Appendices

Appendix A

Abbreviations

ANSI	American National Standards Institute
ARM	Advanced RISC Machine
API	Application Programming Interface
ASN.1	Abstract Syntax Notation 1
BSD	Berkeley Software Distribution
CDT	C/C++ Development Tools
CM	Configuration Management
CPU	Central Processing Unit
CVS	Concurrent Versions System
EOS	Etnoteam Operating System
FIFO	First-In First-Out
FPU	Floating Point Unit
FSF	Free Software Foundation
FTP	File Transfer Protocol
GCC	GNU Compiler Collection
GDB	GNU Debugger
GNU	GNU's Not UNIX
gOS	Generic Operating System
GPL	General Public License
HIRD	HURD of Interfaces Representing Depth
HTML	Hypertext Markup Language
HTTP	Hypertext Transport Protocol
HURD	HIRD of UNIX-Replacing Daemons
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
I/O	Input/Output
IP	Internet Protocol
IPC	Inter-Process Communication
JTAG	Joint Test Action Group
JVM	Java Virtual Machine
KDE	K Desktop Environment
MMC	Multi Media Card
MMU	Memory Management Unit
NFS	Network File System
NIC	Network Interface Card
NPTL	Native POSIX Thread Library

OS	Operating System
PC	Personal Computer
POSIX	Portable Operating System Interface for UNIX
PPP	Point-to-Point Protocol
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RR	Round Robin
RTOS	Real-Time Operating System
SCM	Source Configuration Management
SoC	System-on-Chip
SSH	Secure Shell
SSL	Secure Socket Layer
SVR4	System V Release 4
TCP	Transmission Control Protocol
TLS	Thread Local Storage
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator

Appendix B

Nomenclature

Address Space - A range of logical or physical addresses available for a processor or a process.

ANSI C - A definition of the C language and its standard library, developed by the American National Standards Institute. ANSI C is sometimes referred to as C89 or simply Standard C.

Application Programming Interface (API) - An abstract definition of the interface to a piece of software.

Benchmark - A test suite that serves as a basis for comparison or evaluation.

Big Endian - A data representation of a multibyte value which has the most significant byte stored at the lowest memory address.

Binary Semaphore - A semaphore that can only undertake two values. It is often used to achieve mutual exclusion.

BSD UNIX - A forked version of the original AT&T UNIX that was developed at the university of Berkeley, California, USA. Later versions had all AT&T derivative code removed and were released freely to the public.

Configuration Management - The process of controlling and documenting all changes made during the development of a software project.

Context Switch - The process of switching from one task to another, which involves storing the context of the current task and retrieving the context of the new task. The context of a task may include processor state, virtual address space, scheduling information and file handles.

Counting Semaphore - A semaphore with more than two states, often used to protect multiple resources of the same type.

CPU Scheduling - The operating system mechanism that assigns CPU time to the different tasks in the system.

Cross-Compiler - A compiler that produces object code for another platform than the one it runs on.

Deadlock - A situation in which two or more tasks are unable to proceed, because each is waiting for an event that can only be caused by another of the waiting tasks.

Embedded System - A computerized device designed to perform a dedicated function.

Endianness - An architectural attribute specifying the representation of multibyte values. The two possibilities are called big endian and little endian.

Free Software - Software which the user is free to run, copy, distribute, study, change and improve.

Hard Real-Time System - A system that can guarantee a maximum response time.

HVAC - An acronym for "heating, ventilation and air-conditioning", also known as climate control.

Integrated Development Environment (IDE) - A set of software development tools accessible from a single user interface. The environment includes tools such as editors, compilers and debuggers.

Interprocess Communication (IPC) - Communication between different processes, which is accomplished through a set of operating system facilities.

Little Endian - A data representation of a multibyte value which has the most significant byte stored at the highest memory address.

LonWorks - A networking platform addressing the special needs of control applications, popular for the various functions within buildings such as lightning and HVAC.

Memory Management Unit (MMU) - A hardware component handling the memory accesses requested by the CPU. It is responsible for the translation between logical and physical addresses.

Memory Protection - The capability of preventing a process from accessing the memory of other processes. Memory protection is usually implemented in hardware, with the use of an MMU.

Mutex - A MUTual EXclusion facility. Before entering the critical section, the executing task locks the mutex. No other task is able to lock the mutex until the thread that owns it has unlocked it.

Mutual Exclusion - The process of ensuring that only one task at a time has access to a shared resource.

Open Source - A software distribution philosophy that allows anyone to read and modify the source code free of charge.

POSIX - A set of standards for operating system interfaces based on UNIX.

Preemption - The act of interrupting a currently running task and replacing it with another.

Process - A running program, possibly containing multiple threads of execution.

Real-Time Operating System (RTOS) - An operating system designed for real-time applications, usually employed in embedded systems.

Reentrant - Reentrant code allows multiple threads to share a single copy of the instructions and still be able to interleave execution. The key is to keep task-specific data in separate memory areas, that are distinct for each task.

Response Time - The amount of time passed between the occurrence of an event and the response of the system.

Scheduling Policy - An algorithm that decides which of the processes ready for execution to allocate to the CPU.

Semaphore - A synchronization mechanism that consist of an integer value that can be increased or decreased through two operations. Trying to decrease a non-positive semaphore value causes the calling task to block. There are binary semaphores and counting semaphores.

Soft Real-Time System - A system that attempts to give an answer within a maximum response time, although no guarantees are given.

Synchronization - A type of communication between processes or threads that aims to constrain the relative order of execution. Semaphores and mutexes are synchronization mechanisms.

System V - AT&T System V was the last UNIX version released from AT&T. Other UNIX distributions based on this release are Sun Solaris and SCO UnixWare.

Task - An abstract unit of computation.

Thread - An executing unit with its own stack and register content. Memory is shared between all threads belonging to the same process.

Thread Local Storage (TLS) - A feature providing means to store thread-specific data.

Thread-Safety - A code property that makes it possible for multiple threads to execute the code simultaneously. A piece of code is thread-safe if it is reentrant or if it is protected by a mutual exclusion mechanism.

Virtual Memory - A technique for mapping a large logical address space to a smaller physical address space.

Appendix C

The Generic OS API

C.1 Process Management

```
eError gOSStartProc(tMain Main, int Priority, int StackSize,
                    int NofFiles, int argc, char *argv[],
                    char *envp[], tProcID *pPID)
eError gOSStartUserProc(tMain Main, int Priority, int StackSize,
                        int NofFiles, int argc, char *argv[],
                        char *envp[], int UserID, tProcID *pPID)
eError gOSSuspendProc(tProcID PID)
eError gOSResumeProc(tProcID PID)
eError gOSDeleteProc(tProcID PID, int ReturnValue)
eError gOSGetProcID(tProcID *pPID)
eError gOSSleep(int Nofms)
eError gOSSetProcName(char *pName)
eError gOSGetProcName(char **ppName)
eError gOSSetProcPrio(tProcID PID, int Priority)
eError gOSGetProcPrio(tProcID PID, int *pPriority)
eError gOSInitChildDeathSemaphore(int Count)
eError gOSWait(tProcID *pPID, int *pStatus)
eError gOSNoWait(tProcID *pPID, int *pStatus)
eError gOSGetUID(int *pUserID)
eError gOSSetUID(int UserID)
eError gOSHome()
```

C.2 Semaphores

```
eError gOSCreateSemaphore(char *pName, int Count, tSemID *pSemID)
eError gOSDeleteSemaphore(tSemID SemID)
eError gOSSetSemaphore(tSemID SemID)
eError gOSGetSemaphore(tSemID SemID, int Timeout)
eError gOSResetSemaphore(tSemID SemID, int Count)
eError gOSGetSemaphoreID(tSemID *pSemID, char *pName)
```

C.3 Message Queues

```
eError gOSCreateQueue(char *pName, int NofEntries, tQueID *pQueID)
eError gOSGetQueue(char *pName, tQueID *pQueID)
eError gOSDeleteQueue(tQueID QueID, void (*Cleanup)(void *pMsg))
eError gOSReceiveMsg(tQueID QueID, void **ppMsg, int Timeout)
eError gOSSendMsg(tQueID QueID, void *pMessage, char UrgentFlag)
```

C.4 Memory Management

```
void *gOSAllocateMemory(int Size)
eError *gOSFreeMemory(void *pObj)
```

C.5 Time Management

```
eError gOSGetTime(unsigned int *pTime)
eError gOSSetTime(unsigned int Time)
eError gOSSetSemDelayed(tSemID SemID, int Delay, tTimerID *pTimerID)
eError gOSSetSemPeriodic(tSemID SemID, int Period, tTimerID *pTimerID)
eError gOSDeleteSemPeriodic(tSemID SemID, tTimerID Timerid)
```

C.6 File System

```
eError gOSCurrentDir(char *pDir, int BufSize)
eError gOSOpenDir(char *pPath, tDir *pDir)
eError gOSCloseDir(tDir Dir)
eError gOSNextFile(tDir Dir, char *pFileName, int BufSize,
                   tFileAttributes *pFileAttributes)
eError gOSDiskStat(tDir Dir, tDiskStat *pDiskStat)
eError gOSGetDirSize(char *pDir, unsigned long *pSize)
eError gOSChangeDir(char *pDir)
eError gOSCreateDir(char *pDir)
eError gOSDeleteDir(char *pDir, char DeleteRecursive)
eError gOSRenameFile(char *pFromFile, char *pToFile);
eError gOSDeleteFile(char *pFile);
eError gOSFullPath(char *pFileName, char **ppFullPath);
eError gOSGetFileTime(char *pFile, unsigned long *pSeconds,
                      unsigned long *pUseconds);
eError gOSGetFileSize(char *pFile, unsigned long *pSize)
eError gOSGetFileType(char *pFile, eFileType *pType)
int gOSExec(char *pFile, char *argv[])
```

C.7 System Tracing

```
void gTaskCreate(int TaskID)
void gTaskKill(int TaskID)
void gTaskSwitch(int OldID, int NewID)
```

C.8 Power Failure Handling

```
eError gOSEnterPFSafe()
eError gOSExitPFSafe()
eError gOSRunPFSafe()
int PowerFailTask(int argc, char *argv[])
```

Bibliography

[CSFP05] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. O'Reilly Media, 2005.

[CY05] Hyok-Sung Choi and Hee-Chul Yun. *Context Switching and IPC Performance Comparison between uClinux and Linux on the ARM9 based Processor*. Software Platform Lab, Digital Media R&D, Samsung Electronics, 2005.

[eos99] *EOS User Documentation*. Etnoteam, 1999.

[GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. *gprof: a Call Graph Execution Profiler*. Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, Pages 120-126, 1982.

[Goo04] Jonathan Goodyear. *Die VVS Die!* CoDe Magazine, September/October 2004.

[gos00] *Domino Generic OS Software Design Description*. TAC AB, 2000.

[Gup03] Ravi Gupta. *Linux 2.6 for Embedded Systems - Closing in on Real Time*. LynuxWorks, 2003.

[HS02] Samuel P. Harbison and Guy L. Steele. *C - A Reference Manual*. Prentice Hall, fifth edition, 2002.

[Jon05] M. Tim Jones. *GNU/Linux Application Programming*. Charles River Media, 2005.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[Mil98] Peter A. Miller. *Recursive Make Considered Harmful*. AUUGN Journal of AUUG Inc., 1998.

[MOS01] Mark Mitchell, Jeffrey Oldham, and Alex Samuel. *Advanced Linux programming*. New Riders Publishing, 2001.

[Raj04] Sp. Raja. *Linux for Real Time Requirements*. Integrated SoftTech Solutions P Ltd, 2004.

[Sal94] Peter H. Salus. *A Quarter Century of UNIX*. Addison-Wesley, 1994.

[SGG00] Abraham Silberschatz, Peter Galvin, and Greg Gagne. *Applied Operating System Concepts*. John Wiley & Sons, first edition, 2000.

[tim04] *Migrating to the 2.6 Linux Kernel*. TimeSys Corporation, 2004.

[Wal04] Linus Walleij. *Att använda GNU/Linux*. Studentlitteratur, 2004.

[Wei01] Bill Weinberg. *Moving from a Proprietary RTOS To Embedded Linux*. Montavista Software, 2001.

[Wol01] Wayne Wolf. *Computers as Components - Principles of Embedded Computing System Design*. Academic Press / Morgan Kaufmann Publishers, 2001.