ingenieurwissenschaften

Studiengang Systemtechnik

Vertiefungsrichtung Infotronik

Diplom 2006

Gruber Alexander

Evaluation of Solaris and QNX for aeronautic real time simulation software

Professor	Christophe Bianchi François Corthay
Expert	Jean-Marie Calluaud
Begin	04.09.2006
Hand-in date	16.02.2007
Presentation	21.02.2007

Introduction

My name is Alexander Gruber and I'm currently 23 years old. This report is the result of my final diploma work, which I was able to absolve with Airbus in Toulouse as a part of my education as electrical engineer. I visit the University of Applied Sciences HEVs[1] in Sion, Switzerland and follow the course Infotronics.

Airbus is nowadays world leader in producing airplanes and has outsold in 2005 for the first time in history the biggest concurrent Boeing. With a continuous development of know-how and technique, Airbus is producing over 14 different airplane types. But not only the production of these airplanes is important. A long time before such a machine takes off for the first time, simulations are calculating the behaviour of the airplane. But the simulation is not only about the airplane itself. Environment, aerodynamics and flight behaviour are being simulated as well. Therefore, Airbus is using very powerful computers and real-time operating systems.

But this architecture has one big disadvantage, the price. To be able to operate with very specific software, very specific hardware is needed as well and that is everything else than cheap. When the operating systems could be exchanged with other hardware-independent systems, the costs could be decreased tremendously. Sun Microsystems and QNX Software systems have both published now a new version of their latest operating system, running on x86 architectures. This is actually the first Solaris ever, being compatible to x86 instead of SPARC architecture.

During this diploma work, the two operating systems have to be downloaded, installed and studied. To test out the performances, Airbus has already developed multiple test programs called LibTIM and BGenerique. With these tests, the timer performances of a system can be evaluated and judged. Because portability is very important for Airbus, complex real-time simulation application, called DSS (Distributed Simulation Software), has to be ported, compiled and executed.

So far, all the programs and source codes exist already and just need to be changed and modified. But timers are not the only important component for a successful real-time operating system. Therefore, a message queue and semaphore test program has to be implemented and executed. Once all tests are realized, this report should allow the reader to get an idea of these two operating systems and their performances.

[1] Find more information on: www.hevs.ch

Index

Company presentation 5
A little bit of history 5
Airbus 5
The company Airbus France SAS 6
Production site Toulouse 7
Working environment 7
Software department EYYW 8
Solaris 10 10
A little bit of history 10
About Solaris 10 10
Installation 15
Handling 17
Support 18
QNX Neutrino 6.3 19
A little bit of history 19
About QNX 6.3 19
Installation 24
Handling 24
Support 26
QNX Momentics 26
Tick size 32
Test programs 33
LibTIM 33
Generic created Test (BGenerique) 37
Timer Latency (only QNX) 38
Shared memory 38

Fork system call 38
Message queues 40
Semaphores 42
Sorting algorithm 44
Portability 47
Conclusion 48
Oneshot, absolute 48
Oneshot, relative 48
Periodic, absolute 49
Periodic, relative 49
Semaphore 49
Message Queue 49
Illustration Index 51
Links 53
Solaris 10 53
QNX 6.3 53
General information & links 53
Books 53
Appendix 54
Information about the thesis 54
Bench Generique (BGenerique) results 55
Semaphore test results 61
Message queue test results 68
Installing network card under Solaris 10 77
Content of the supported CD-ROM 80

Company presentation

A little bit of history

In October 1920, Emile Dewoitine, one of the early flight pioneers, founded his own proper company: the Dewoitine airplanes construction enterprise. He installed his factory in Toulouse for producing his first transportation airplane, the Dewoitine 338. From 1921 until 1931, this factory produced fighter aircrafts and sailplanes before being nationalized and renamed as Société Nationale de Construction Aéronautique du Midi (SNCAM). This group, which changed it's name into Sud Aviation after a couple of years, had it's first huge technical and commercial success through the realization of 270 airplanes of type "Caravelles" between 1958 and 1973. Further the story continuous with a rather legendary airplane: Concorde, which had his first flight on the 2nd march 1969. Realized in a partnership with British Aircraft Corporation, Concorde is still one of only two ever realized supersonic airplanes used for civil aviation. Even after it's grounding in year 2000, caused by a deadly accident in Paris, it never lost its very special charm and attraction.

January first 1970, the enterprises Sud Aviation, Nord Aviation and SEREB amalgamated for founding the Société Nationale Industrielle Aérospatiale (renamed Aérospatiale in 1984). It took part at founding the joint venture group Airbus Industries, which main interest should be the production of civil aviation aircrafts.

Later on in June 1999, Aérospatiale amalgamated with Matra hautes Technologies (a Lagardère group) and founded the Aérospatiale Matra with the participation of Dassault Aviation, becoming the 2nd largest company in Europe and the 5th largest worldwide in aeronautical defence sector. Next step in logic of fusions Europe wide, EADS (European Aeronautic Defence and Space Company) was founded in July 2000. EADS is a result of fusion between Aerospatiale Matra with its European partners DASA and CASA. This newly founded group places himself as number one in Europe in civil aviation and employs approximately 103'000 workers placed in 90 production sites spread all over the world.

Airbus

Toward the end of year 2000, the European commission gave its agreement to transform the consortium GIE Airbus as an integrated single company. EADS and BAE SYSTEMS created in June 2001 AIRBUS SAS, which is as well divided into four sections: AIRBUS France, AIRBUS Deutschland, AIRBUS España and AIRBUS UK. Nowadays, Airbus belongs to 20% BAE Systems and to 80% EADS.

Airbus, with its 55000 employees spread over 15 manufacturing places in all Europe, has put himself as number one worldwide before its biggest opponent Boeing. With a turnover of 28 billion euros in year 2005, Airbus is in a very comfortable position comparing to Boeing. And an end is not in sight; there are still approximately 2000 aircrafts to be delivered. Airbus is situated in a market for aircrafts with more than 100 places. Nowadays, the fleet consists of single corridor aircrafts (A318-A319-A320-A321) as well as of jumbo jets (A330-A340-A380). The production of the very successful type A300-A310, which is used for civil cargo transport, will come to an end with the delivery of number 878 at the end of year 2007. The latest newly born, A380 (which had its first flight on April 27th 2005) has nearly passed all the certification flights. The first delivery is planned for the end 2006.

At the moment, there is another project being developed: the A400-M, which is a military cargo aircraft. First flight is planned for the end of 2007.

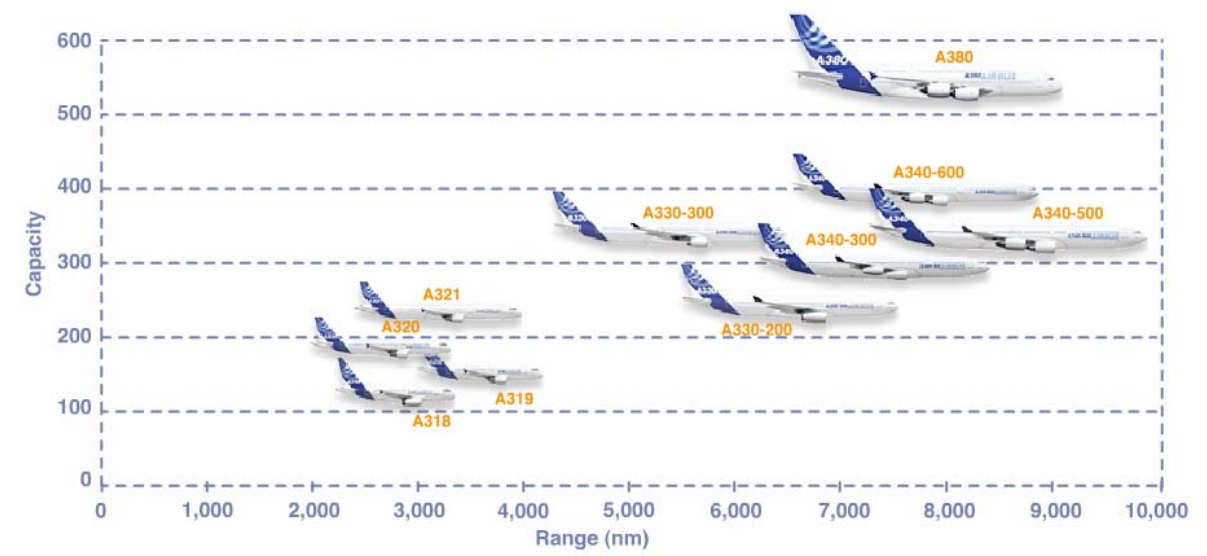

Illustration 1: Range/Capacity Diagram for the different Airbus airplane types[2]

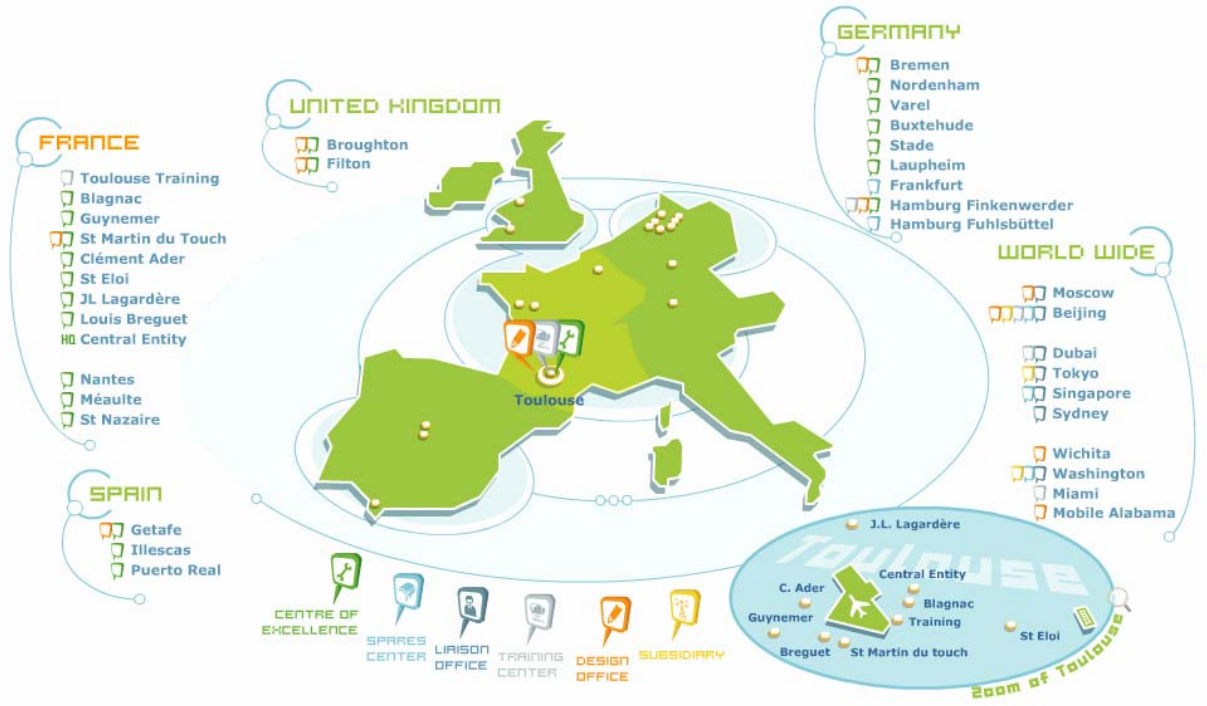

Illustration 2: Airbus Europe with zoom on Toulouse

The company Airbus France SAS

Airbus France SAS is divided into four different production places: Toulouse, Méaulte, Nantes and Saint Nazaire.

[2] 1nm is equal to 1852m

Production site Toulouse

With its 15000 employees, Toulouse is the biggest and most important production site of Airbus. In Toulouse, the underneath listed activities can be found:

- Research office
- Fabrication of iron and titan parts
- Electrical fabrications
- Assembly of the engine pylons
- Production of avionics equipment
- Final assembly line (A380, A340…)
- Commercial furnishings
- In-flight and ground testing
- Administration office

Those 15000 employees are working on sites north west of Toulouse, using a surface of 440 hectares. They can be split up into historical production sites:

- Blagnac
- Saint Martin
- Saint Eloi
- Breguet

And newly built production sites :

- Guynemer
- Clement Ader (A330/A340)
- Lagadère for final assembly line of the A380

Working environment

The Department for Avionics and Simulation Products is a part of the competence centre for systems and test of integration (EY). Founded in 1964 for the Concorde program, it is responsible for delivering electronically products and software systems for the avionics market, which gave him the status of being a company within the company.

It's main activity can be split up into two categories:

- Avionics: realization of electronically embedded equipment for Airbus and ATR (flight commands, alarms, communication, braking…)
- Simulation:
  - Study: evaluation of research simulators of four different types:
    - Prospective (EPOPEE): Prospective study of organization for ergonomically equipment for future cockpits.
    - Conception: Validation of the specifications delivered from the research office.
    - Development, airplane –1: Precise the rules for flight commands, for autopilot and for the interface man/machine inside the cockpit. Allows verifying the aspect "quality of flight".
    - Integrations, airplane 0: Global integration of real equipment, first flight preparations, realization of a maximum of certification tests.
  - Training: Simulators for the equipment education as well as for maintenance teams

Software department EYYW

Within the Department Avionics and Simulation Products, the Software Department is the pole of skill and expertise in field of the software. The main missions of the department are:

- Supply of skills, methods and tools at the production departments for their realization of operational objectives.
- Prospective studies on new technologies, architectures, methods and software workshops.
- Studies aiming at the increase of performance of the systems.
- Implementation, on demand of the Centre of Skill of the company, missions of expertise in its field of skills

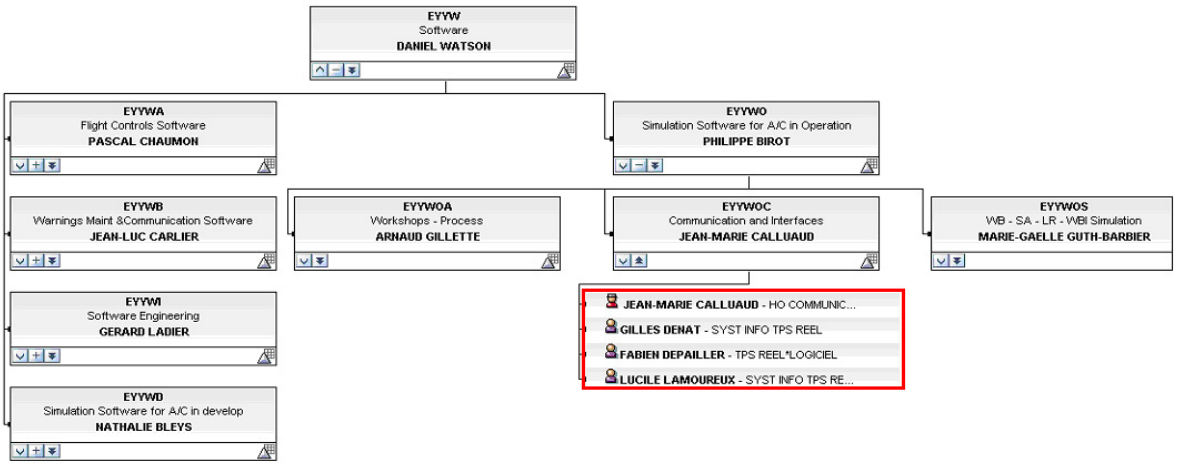

Illustration 3: Organization of EYYW department

The software department is responsible for research, development and maintenance of the software programs. It's defining new architectures for the simulation software programs for the A380 and A400M programs for example. It is as well responsible for the functionality of the already existing software applications.

The group Communications and Interfaces (EYYWOC) has the task to research, propose and conceive data-processing architectures for the simulators as well as for developing and maintaining the software interfacing the different architectural elements. It is within this service that I worked out my diploma work.

So remains the question: why Airbus has interest in my diploma work? A simulation system consists of 4 different layers: application, scheduler, OS and hardware.

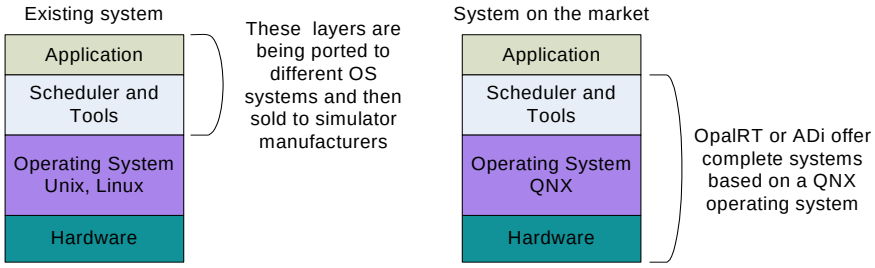

Illustration 4: Why interest in QNX and Solaris?

Nowadays, Airbus is using architecture with different operating systems. While the application and scheduler layer always remains the same, the operating system can change between the simulators. Unix and Linux are the most common so far but maybe in future there might be as well Solaris or QNX. Airbus is selling these two layers as well to simulator manufacturers so that the realism of the simulator is as high as possible. When now a manufacturer should be interested in having these layers compliant with another operating system, Airbus has to be flexible and port the applications to the new OS. To simplify this possible process, the systems are being tested out for their real time performances already in advance.

To reduce costs and development time, OpalRT and ADi are now offering complete systems with the three last layers, means: scheduler and tools, operating system and hardware. Under many different systems, a QNX system is available as well. Therefore, it is interesting to know, if this new operating system is as real-time friendly as the existing systems today.

This diploma work will take place over 5 months and will be divided into several different topics. The first and mandatory task was to gather information about these two operating systems, to download the source and to install them on their target systems. Solaris is not as unknown in Airbus as QNX, because some of the architectures are already Solaris SPARC machines. QNX is completely new for everybody and we were very happy to have the possibility of a presentation, held by Franck Vancoellié and Sean Meroth. Thank you again guys for your time and knowledge!

As mentioned before, I had the chance to accomplish my diploma work in the Software Department. To them, the most important point in a system is the real time performance. To test these systems under real conditions, a complex real time simulation program was ported. This software is responsible to communicate between multiple host machines, all running a simulation software and is called DSS (Distributed Simulation Software). Through the portage of this application, I was able to test out the development tools on each system as well as the POSIX compatibility.

To communicate between different target systems, a precious synchronization is mandatory and therefore, timers cover a very important role in real time programming. By implementing Airbus software called LibTIM and BGenerique, the POSIX compatibility, thread performance and real time priorities could be tested out. Even though the current QNX version was not SMP (Symmetric Multi Processing) compliant, Solaris could be tested out as well under multi-processor environment. QNX Software System was offering a multi processor host machine, capable to run QNX but we decided to do without because of the short amount of time we had left.

All the tasks so far have been more research than active development. To get a better overview on the real time performances concerning time consumption, several test programs were written. Especially semaphores and message queues are an often used tool for process synchronization. The fact, that the programs are not very complicated and on every system exactly identical, makes it very interesting for a final judgement concerning the over all performance of these systems.

Solaris 10

A little bit of history

Solaris is developed and distributed by the Sun Microsystems company with its headquarter in Santa Clara, California, USA. Solaris is a derived version of the BSD (Berkeley Software Distribution) introduced in the early 1990s under the name SunOS 5.0 or Solaris 2. In Solaris is always incorporated the SunOS and is considered as an operating system plus a graphical Environment. In its version name is the version of SunOS included, although the major version has been dropped. For example the new Solaris 10 operating system is based on SunOS 5.10. The picture below shows the historical tree of Solaris 10:

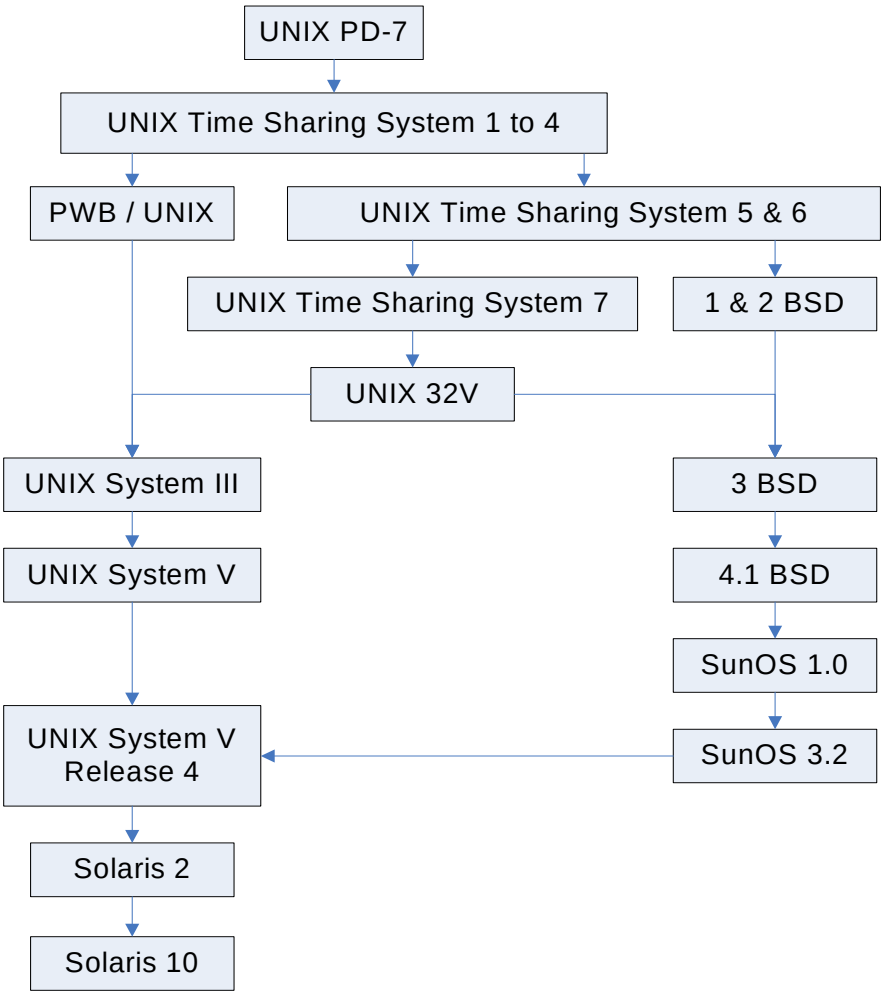

Illustration 5: Historical tree of Solaris 10[3]

About Solaris 10

Solaris 10 is currently freeware, including a 30-day trial licence for its IDE (Integrated Development Environment). But even after the expiration of this trial licence, the operating system itself remains completely functional. As the former operating system versions have only been compatible with SPARC processors, Solaris 10 can now as well be installed on x86

[3] Source: http://upload.wikimedia.org/wikipedia/commons/5/50/Unix_history-simple.png

processor architectures. This makes it very interesting, because the x86 systems are much cheaper as the specific SPARC systems.

Solaris 10 comes along with some features no other operating system can claim so far. This is how Sun describes the new features:

System Analysis Tools

- Powerful thread analysis and monitoring tools, including lockstat, truss, and pstack
- Memory management and debugging tools, including libumem, a high-performance multithreaded memory allocation library with built-in monitoring functions
- Support for Intelligent Platform Monitoring Interface (IPMI), an industry standard for "lights out" management of x64/x86-based servers
- Modular Debugger (mdb) and Kernel Modular Debugger (kmdb), powerful and extensible tools for monitoring and analyzing applications and kernel routines
- System and application core administration and debugging tools
- Sun Validation Test Suite for hardware testing and analysis

Process Accounting and Statistics

The Solaris 10 project and task facilities allow to label and separate workloads, as well as monitor resource consumption by each workload. The extended accounting subsystem captures a detailed set of resource consumption statistics on both processes and tasks. In conjunction with the Internet Protocol Quality of Service (IPQoS) flow accounting module, this subsystem can also capture network flow information on a system.

Enhanced Patch Management

Proper system analysis can be critical to system availability and performance. To this end, the Solaris 10 Operating System includes tools to manually or automatically perform patch management, including analyzing the system to determine which patches are appropriate for your configuration.

Of course there are a lot of other new features as well, but these are the most important features concerning the observation of the operating system. For more information about Solaris 10 and its features, please visit the Solaris home page.[4]

System architecture

The file systems architecture is similar to other UNIX systems. It is hierarchical, which begins with a root directory, and from which the branches of all other directories and file systems are mounted.

As in every UNIX system, the kernel is the core of the system. It is responsible for managing the hardware resources and provides an execution environment for user programs. The Solaris kernel supports an environment in which multiple programs can execute simultaneously. Its primary functions can be divided into two major categories: managing the

[4] Address: http://www.sun.com/software/solaris/index.jsp

hardware by allocating its resources among the programs running on it, and supplying a set of system services for those programs to use.

The basic unit that provides a program's environment is known as a process; it contains a virtual memory environment that is insulated from other processes on the system. Each Solaris process can have one or more threads of execution that share the virtual memory environment of the process, and each thread in effect executes concurrently within the process's environment. The Solaris kernel scheduler manages the execution of these threads by transparently time-slicing them onto one or more processors. Each time a thread is moved off a processor, its complete execution environment is saved, so when it is later rescheduled onto a processor, its environment can be restored and execution can resume.

The kernel itself is grouped into several key components and is implemented in a modular fashion. The key components of the Solaris kernel are described and illustrated below:

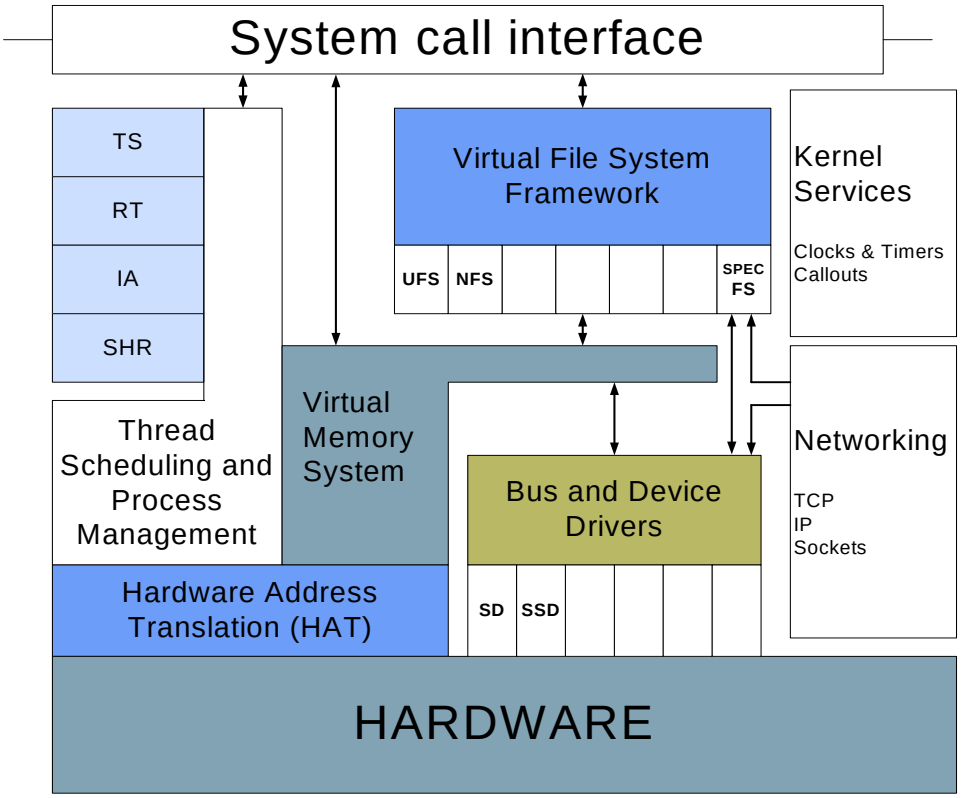

Illustration 6: Key elements of the Solaris 10 kernel

System Call Interface – The system call interface allows user processes to access kernel facilities. The system call layer consists of a common system call handler, which vectors system calls into the appropriate kernel modules.

Process Execution and Scheduling – Process management provides facilities for process creation, execution, management and termination. The scheduler implements the functions that divide the machine's processor resources among threads on the system. The scheduler allows different scheduling classes to be loaded for different behaviour and scheduling requirements.

Memory Management – The virtual memory system manages mapping of physical memory to user processes and the kernel. The Solaris memory management layer is divided

into two layers: the common memory management functions and the hardware-specific components. The hardware-specific components are located in the hardware address translation (HAT) layer.

File Systems – Solaris implements a virtual file system framework, by which multiple types of file system can be configured into the Solaris kernel at the same time. Regular disk-based file systems, network file systems, and pseudo file systems are implemented in the file system layer.

I/O Bus and Device Management – The Solaris I/O framework implements bus nexus node drivers (bus-specific architectural dependencies, e.g., a PCI bus) and device drivers (a specific device on a bus, e.g., an Ethernet card) as a hierarchy of modules, reflecting the physical layout of the bus/device interconnect.

Kernel Facilities (Clocks, timers, etc.) – Central kernel facilities, including regular clock interrupts, system timers, synchronization primitives, and loadable module support.

Networking – TCP/IP protocol support and related facilities. The Solaris networking subsystem is implemented as streams-based device drivers and modules.

Processes, Threads and Scheduling

The Solaris kernel is multithreaded; that means, it is implemented with multiple threads of execution to allow concurrency across multiple processors. This architecture is a major departure from the traditional UNIX scheduling model. In Solaris, threads in the kernel are the fundamental unit that is scheduled and dispatched onto processors. Threads allow multiple streams of execution within a single virtual memory environment; consequently, switching execution between threads is inexpensive because no virtual memory context switch is required.

Threads are used for kernel-related tasks, for process execution, and for interrupt handling. Within the kernel, multiple threads of execution share the kernel's environment. Processes also contain one or more threads, which share the virtual memory environment of the process.

A process is an abstraction that contains the environment for a user program. It consists of a virtual memory environment, resources for the program such as an open file list, and at least one thread of execution. The threads within each process share the virtual memory environment, open file list and other components of the process environment.

Within each process is a lightweight process, a virtual execution environment for each kernel thread within a process. The lightweight process allows each kernel thread within a process to make systems calls independently of other kernel threads within the same process. Without a lightweight process, only one system call could be made at a time. Each time a system call is made by a thread, its registers are places on a stack within the lightweight process. Upon return from a system call, the system call return codes are placed in the lightweight process.

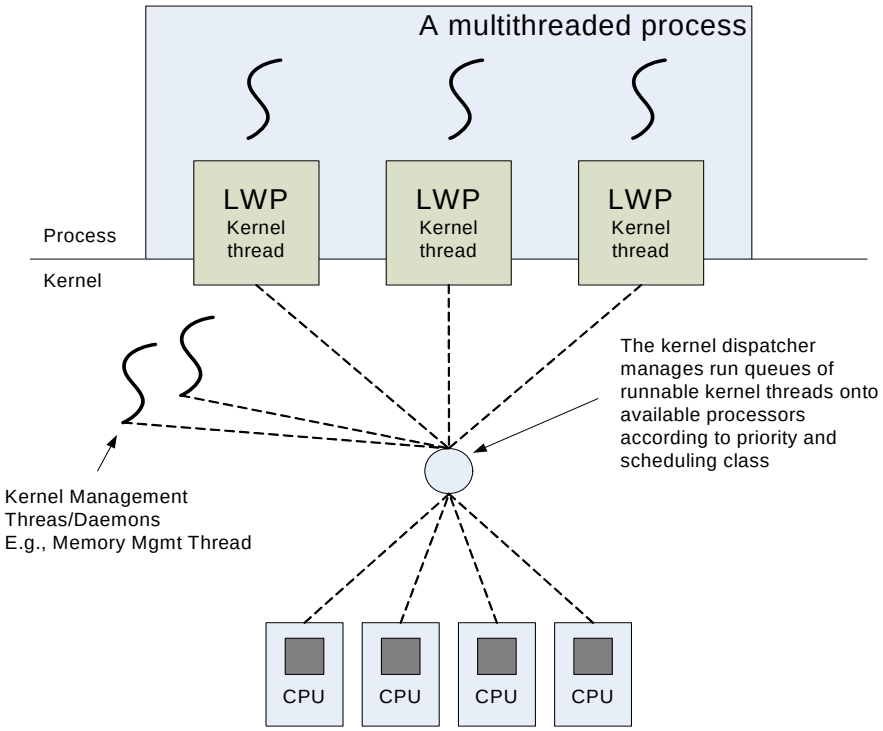

Illustration 7: Kernel threads, processes and lightweight processes

Global Process Priorities and Scheduling

The Solaris kernel implements a global thread priority model for kernel threads. The kernel scheduler, or dispatcher, uses the model to select which kernel thread of potentially many runnable kernel threads executes next. The kernel supports the notion of pre-emption, allowing a better-priority thread to cause the pre-emption of a running thread, such that the better- (higher) priority thread can execute. The kernel itself is preemptable, an innovation providing for time-critical scheduling of high-priority threads. There are 170 global priorities; numerically larger priority values correspond to better thread priorities. The priority name space is partitioned by different scheduling classes, as illustrated in the picture below:

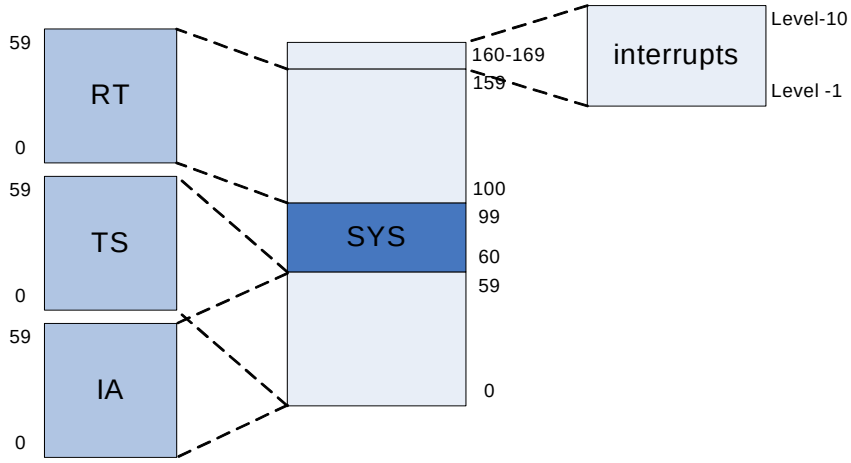

Illustration 8: Scheduling classes of Solaris 10

The Solaris dispatcher implements multiple scheduling classes, which allow different scheduling policies to be applied to threads. The three primary scheduling classes – TS (IA is an enhanced TS), SYS and RT:

TS – The timeshare scheduling class is the default class for processes and all the kernel threads within the process. It changes process priorities dynamically according to recent processor usage in an attempt to evenly allocate processor resources among the kernel threads in the system. Process priorities and time quantums are calculated according to a timeshare-scheduling table at each clock tick, or during wakeup after sleeping for an I/O. The TS class uses priority ranges 0 to 59.

IA – The interactive class is an enhanced TS class used by the desktop windowing system to boost priority of threads within the window under focus. IA shares the priority numeric range with thee TS class.

SYS – The system class is used by the kernel for kernel threads. Threads in the system class are bound threads; that is, there is no time quantum – they run until they block. The system class uses priorities 60 to 99.

RT – The real-time class implements fixed priority, fixed time quantum scheduling. The real-time class use priorities 100 to 159. Note that threads in the RT class have a higher priority over kernel threads in the SYS class.

The interrupt priority levels shown in the picture above are not available for use by anything other than interrupt threads. The intent of their positioning in the priority scheme is to guarantee that interrupt threads have priority over all other threads in the system.

Installation

Because Solaris used to be compatible with SPARC systems only, the development for drivers is still on progress for x86 systems. Therefore, only a few devices are supported. Before the installation can begin, the hardware components of the systems have to be checked in a hardware compatibility list.[5]

Even though not all components of my system were listed up, I was lucky and the system was working fine. Sun is currently working together with a community called Open Solaris. This project allows everybody to be a part of the Solaris development by sending in modified source code. This code is then being observed from some engineers and published on the net for everybody to use. Sun Microsystems has released the complete source code from Solaris in hope to receive some creative suggestions to advance the system and reduce the amount of bugs. Successful modifications will then be bound into the next published official Solaris version from Sun Microsystems. More information about this community project can be found under www.opensolaris.com. The latest version of the Solaris operating system can be found under: www.sun.com.

Once the preparations are finished, the installation of Solaris can begin. This is indeed a very easy task. After booting from your CD-ROM, the installation screen appears. Choose Solaris Interactive Installation and follow the instructions. Solaris can be installed in multiple

[5] Address: http://www.sun.com/bigadmin/hcl/data/sol/

different modes. The picture below shows, how Solaris is build up. For a fully supported system, choose Entire Plus OEM:

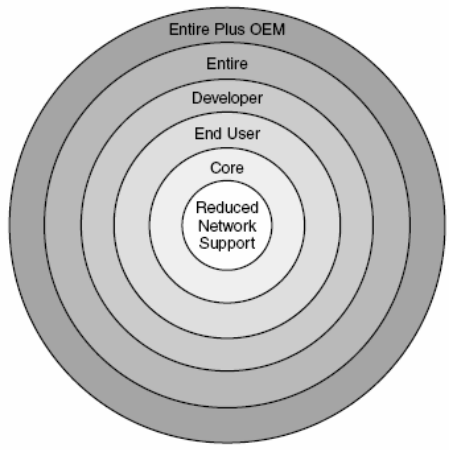

Illustration 9: Solaris installation levels

From now on, the installation should not cause any problems. Make sure, you allocate enough disk space to directory /opt. Because all supplement software packages will be installed per default in this directory.

To develop new software, the Sun Studio 11 is available for download on the Sun Microsystems homepage. In this software included is a GCC compiler. To install the software package, extract the downloaded files into a new folder, enter this folder and execute the installer by typing:

```
# ./installer
```

Default folder for installation is /opt. Once the installation has been completed, the path variable needs to be changed. This can either be done temporary by typing:

```
# PATH=$PATH:/opt/SUNWspro/bin; export PATH
```

or continuously by adding:

```
PATH=$PATH:/opt/SUNWspro/bin
export PATH
```

to the file /etc/profile. If the installation of Sun Studio should cause any other problems, please refer to the Sun Studio Installation guide on the Sun homepage. To get access to the GNU C Compiler (GCC), the file /etc/profile has to be changed once more by adding:

```
PATH=$PATH:/usr/sfw/lib/bin:/usr/ccs/bin/make:/usr/ccs/bin/ar:/usr/ccs/bin/ld
export PATH
```

This allows the shell program to find an executable. When a command is executed on the shell, the operating systems checks out all the folders included in the PATH variable to find the corresponding executable.

With this last step, the basic installation of Solaris 10 is finished. To note that Solaris would usually run with 64bit, but unfortunately for our network card was no 64bit driver available, so the system is now always running with 32bit.

Handling

The start up time is not really extraordinary and takes approximately 1min and 15 seconds. Quicker is the shut down, which takes only about 7 seconds.

As soon as Solaris is started up, the login screen appears and we can choose between two different desktop environments. The first is the Common Desktop Environment and second one is called Java Desktop System, Release 3 and is based on Java. To choose between these desktop environments, click on Options => Session and then choose the preferred environment. The functionality of these environments is exactly the same as well as all the available applications. The difference is just a question of design.

Common Desktop Environment

The navigation is very easy through the principal menu list in the middle bottom of the screen. All applications and configuration tools can be accessed through this menu. Another possibility to show the menu content is the right-click on the desktop.

I find it personally annoying, that a minimized window is not being minimized on the task list as used from Windows or other operating systems, but a shortcut is being created on the desktop. As well the navigation between the different windows is not as easy as normal. By typing alt+tab we can flip between the programs but always in a round-robin function. A specific application cannot be targeted directly and therefore the navigation with alt+tab is not useful.

Similar to windows, the mounting of devices is done automatically. This is working very fine under the Common Desktop Environment but not at all under the Java Desktop System. The shortcuts to open a floppy or CD-ROM drive can be found under both solutions, but only the ones under the Common Desktop Environment are working properly. To open a floppy drive, go through Applications => System_Admin => Open Floppy. Now a new window is opened and the content of the floppy drive is being represented. But when the floppy is now being removed, its content modified and inserted again, then the system is not actualizing the screen as long as we don't open the floppy drive again. This is not very clever and makes working complicated and time consuming.

Because the design of the Java Desktop System was more comfortable to work with, my decision was made pretty quickly.

Java Desktop System

In comparison with the Common Desktop Environment, this environment is similar to Windows. On the bottom of the screen is the task list with the main menu on the left bottom. To open the menu, click on Launch and choose the preferred application. Flipping between two applications is very easy and can be effectuated with alt+tab.

The Java Desktop System comes along with a lot of free software tools like the complete Star Office collection! That makes it very easy and comfortable, because almost everybody is able to work with office products and can start directly after the installation of the operating system, without installing additional software.

Finding a file under Solaris is very easy: click on Launch and choose Finding Files... A new window is opening and the search parameters can be defined. With the option show more

options, we can as well search for a word or a part of a word inside the files. That is a very nice and useful tool, for example to search the source file of a method.

Unfortunately, it is not possible to define a general text size for all the menus and task lists. The result is an almost unreadable representation on the screen. The characters are so small, that it is almost not possible to read them. Even though I found a general setting of the operating system, I was still not able to define it for all the applications. Especially programs to edit source code are really tiring after a short amount of time.

Support

Solaris is not as popular as Linux, but far more popular as QNX on the other hand. While QNX is a special operating system for real time specific applications, Solaris is more general and not absolutely specific for real time. It is not very difficult to find corresponding information on the Internet about this operating system. Of course the support is not as big as with Linux, but not so bad.

The most important Internet community is called Open Solaris as mentioned before under point installation. The idea behind this community is the development of new and the improvement of existing features. Besides of Open Solaris, other communities and interest groups can be found on the Internet. Please check out the link list in the appendix for further addresses of Solaris communities.

Unfortunately, Solaris does not provide the costumers with an included help library. All information concerning the operating system itself or system methods need to be looked up on the Internet. But because Solaris is quite famous, it should not be too hard to find some serious information.

Of course Sun Microsystems is offering for enterprises different support levels:

- Sun Developer Service Plans – A packaged offering for the enterprises that combines Developer Expert Assistance, Sun Software Service Plans and Training credits, priced according to an organization's needs. Sun Developer Service Plans are for development and testing only; plans are product – and version- specific.
- Java Multiplatform Support: Mission/Critical Java Product Escalation.
  Mission-critical deployment telephone support for enterprise customers running applications that use Sun the Java runtime environments (JREs) in environments using Windows, Linux or Solaris operating systems.
- Sun Solution Support Engineering Services
  Direct access to senior support engineers, who will provide proactive and reactive relationship-based services throughout the lifecycle of your solution, including the development phase.

A wide variety of third party software can be found on the Internet. Just note that there exist two versions of Solaris, one for x86 architecture processors and another for SPARC architecture.

QNX Neutrino 6.3

A little bit of history

While the two students Gordon Bell and Dan Dodge were writing a little real-time kernel during their studies in 1980, they were convinced that there was a commercial use for such a system. So they finished their studies and moved to Kanata, Ontario, Canada where they founded Quantum Software Systems. In 1982 they published the first QNX version for Intel 8088 CPU's. While towards the end 80's POSIX became popular, they decided to rewrite the kernel to be accessible on a much lower level. The result was QNX 4. This made porting Unix or BSD packages to QNX much easier.

Another 10 years later, they started to build up a new operation system, fully SMP (Symmetric multiprocessing) capable and POSIX API's compatible, QNX 6. QNX Software Systems is part of the Harman International Company. The picture below shows the historical tree of QNX 6.3:

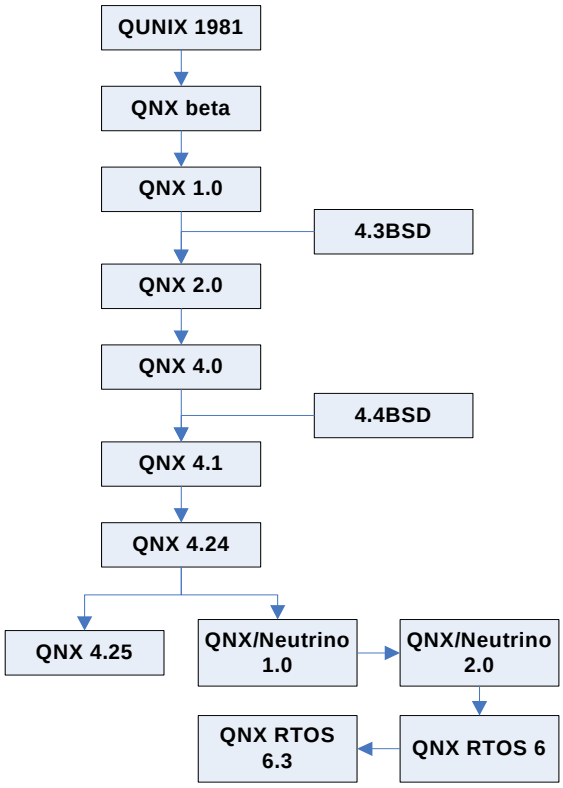

Illustration 10: Historical tree of QNX 6.3[6]

About QNX 6.3

The big difference between QNX and Solaris is the origin. Solaris has been developed out of the SunOS and this itself is derived from UNIX PD-7. QNX is different. From the beginning on, the system was specially built up for one and only purpose: mission-critical applications. QNX is not a derived version of another operating system but a true microkernel operating system. This makes it able to be used in a several of different time critical

[6] Source: http://www.levenez.com/unix/history.html#08

situations. Medical instruments, air traffic control systems or even urgency call centres all trust on the long and very deep experience of QNX to solve such complex and difficult situations reliable. But what makes the microkernel so special comparing to other operating systems? It's the protection of the kernel against any other running process. When a new process is started, it's placed in memory-protected user space. This allows the kernel to restart a crashed process at any time, without affecting either the kernel itself or another running process. Some of the new QNX features are listed below:

Power Management Framework

Using this comprehensive framework, allows to exercise fine-grained control over the power states of every peripheral and to create a customized, application-specific power policy for each system. The framework includes libraries to build power-managed drivers, power-sensitive applications, and a centralized power manager. This is very useful to realize portable products where the power consumption is very important (MP3 Player, Navigation System, etc.).

Instrumented Microkernel

QNX can be started in a normal and an instrumented kernel. The instrumented kernel is very useful to quickly pinpoint timing conflicts, deadlocks, logic flaws, software faults, and a variety of other hotspots, in both uniprocessor and multiprocessor systems.

When the instrumented kernel is started, all the events are being filtered to either static defined or user defined filters. The events are being saved in an event buffer to be read out later on by the system profiler, which is a program delivered together with the Technology Development Kit (TDK). All the events can then be visualized on the screen to observe the exact manner the operating system is working. These activities are being represented in the following picture:

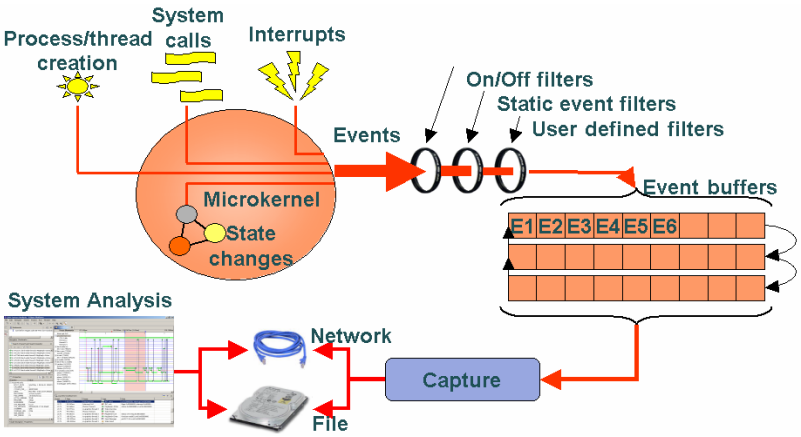

Illustration 11: Instrumented kernel of QNX 6.3[7]

Processor Support

The QNX Neutrino RTOS offers advanced support for the ARM, MIPS, PowerPC, SH-4, StrongARM, Intel® XScale™ Microarchitecture, and x86 processor families. It also supports functions and macros to write processor-independent drivers and applications. This makes it

[7] Source: QNX presentation of Franck Vancoeillé

very interesting and time saving to debug and test the applications before the target processor is chosen or target different processors.

Of course the QNX system can offer many other features to optimize the system. More information about these features can be found on the official QNX homepage.[8]

System architecture

As QNX is a UNIX like system, the file system architecture is similar to Linux. Like Solaris, the root folder is the origin from where all the different folder branches are mounted.

But now comes the big difference, as mentioned further on, the kernel consists only of the most fundamental services like signals, timers and scheduling. Everything else is running in memory-protected user space like the picture below shows:

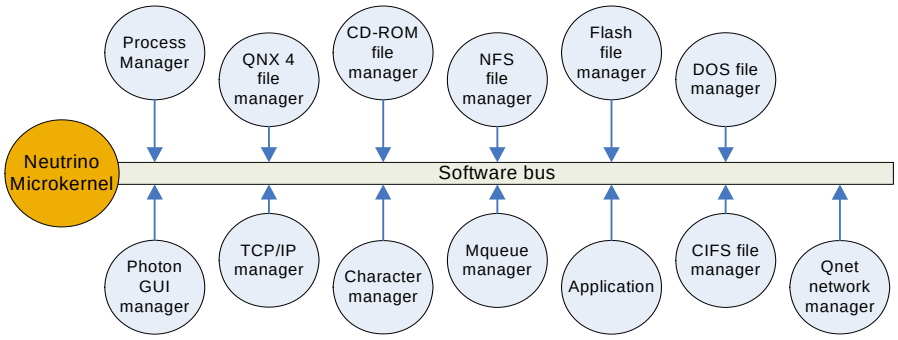

Illustration 12: Organization of the QNX kernel

The communication between the different services is realized with a message passing system. This allows connecting or disconnecting any service at any time without causing problems to the other still running services. Blue screens or total kernel crashes are not possible with such a system. Further is it possible to connect two hosts running QNX over Ethernet and then access to each other's resources over the message-passing bus.

This new and exclusive memory-protection strategy can only be found in QNX and makes it therefore a lot more stable and secure as the following pictures show in comparison with other operating systems:

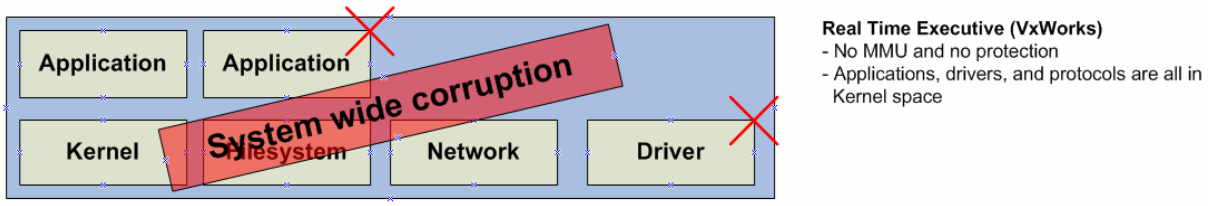

Illustration 13: System architecture of VxWorks

[8] Source: http://www.qnx.com/products/rtos/glance.html

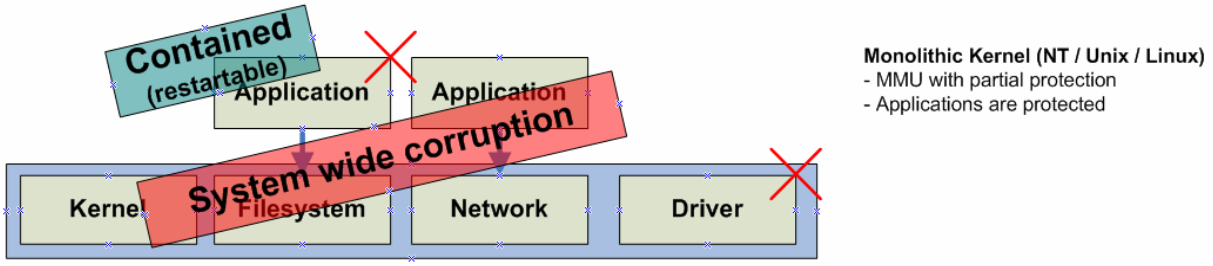

Illustration 14: System architecture of NT, Unix or Linux

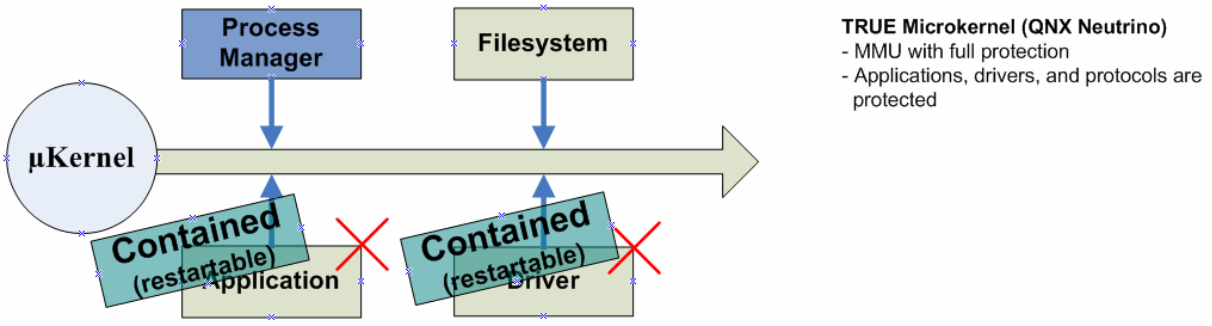

Illustration 15: System architecture of QNX Neutrino[9]

Threads and processes

In some applications, it is required to execute two or more commands concurrently. Therefore, inside the process can be built up several threads. Every thread itself is only activated for a very short time slice (4 times the clock period), before it gives the processor to the next thread. Not every process has to have multiple threads, but at least one. The threads are being executed in the memory space of the parent process and therefore concurrent underneath each other inside this memory space. A process can be compared with a container of threads.

On the following picture, the prioritized threads are shown. The higher the priority level, the sooner a process is being executed in comparison to the other threads with lower priority levels. In this example, thread C will be executed first, because it's priority level is the highest present. The system idle process owns the priority level 0. Non-root users are only allowed to create processes with priorities between 1 and 63, where root users can use the whole spectrum of priorities (1 to 255).

[9] Source of Illustration 13, Illustration 14 and Illustration 15: QNX presentation of Franck Vancoeillé

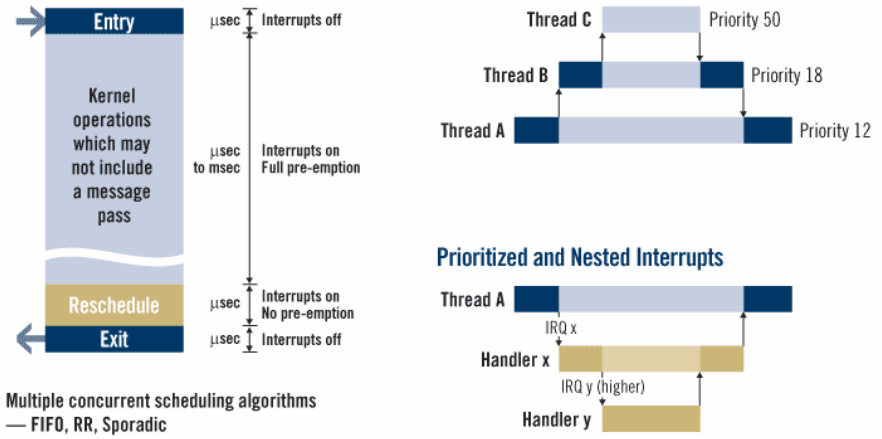

Illustration 16: Thread priority and interrupts[10]

Scheduling algorithms

Should a process consist of more than one thread, it could happen, that both threads are ready to be executed at the same time. Now the kernel tests the different priority levels and executes the thread with the highest level. That works fine as long as every thread has a different priority. When two processes with the same priority are ready at the same time, scheduling algorithms need to be defined to treat the threads scheduling. QNX has therefore defined 3 different modes:

1. FIFO: First in first out, the processes are being executed in the same sequence as they became ready
2. Round-robin: all the processes are being executed one after the other, without any special selection process
3. Sporadic scheduling: Under sporadic scheduling, a thread's priority can oscillate dynamically between a foreground or normal priority and a background or low priority. Using the following parameters, you can control the conditions of this sporadic shift:
   - ⇨ Initial budget (C): The amount of time a thread is allowed to execute at its normal priority (N) before being dropped to its low priority (L).
   - ⇨ Low priority (L): The priority level to which the thread will drop. The thread executes at this lower priority (L) while in the background, and runs at normal priority (N) while in the foreground.
   - ⇨ Replenishment period (T): The period of time during which a thread is allowed to consume its execution budget. To schedule replenishment operations, the POSIX implementation also uses this value as the offset from the time the thread becomes ready.
   - ⇨ Max number of pending replenishments: This value limits the number of replenishment operations that can take place, thereby bounding the amount of system overhead consumed by the sporadic scheduling policy.

The next picture shows, how the sporadic scheduling algorithm works:

[10] Source: www.qnx.com

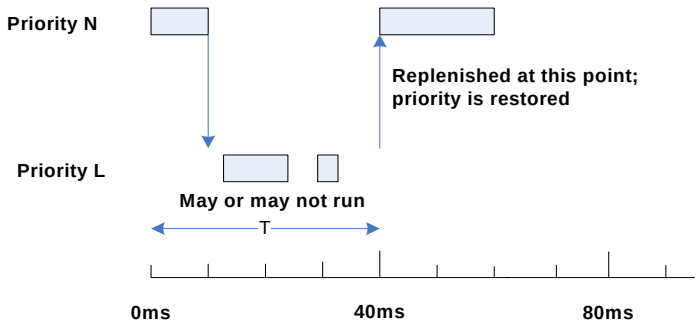

Illustration 17: Priority change of thread while sporadic scheduling

Installation

To test out different operating systems and to obtain at the same time comparable results, the tests should be executed on the same host machine. But unfortunately, QNX is not available for multiprocessor systems at the moment. Of course in future, a multiprocessor version will be distributed. That's why we had to organize another different host machine for QNX with a mono processor system.

QNX is as Solaris not a very common operating system for private use. This makes it not very popular and therefore, the hardware needs to be tested on its compatibility. This can be done on the QNX[11] home page. Of course the compatibility will be increased, the longer QNX is on the market.

The current version of QNX is 6.3. This version is open source and can be downloaded for free under www.qnx.com. The installation-package consists of the operating system QNX Neutrino and a development tool called Momentics TDK. While the development environment is only freeware for 30 days, the QNX operating system itself remains open source. Once the installation package is downloaded, it can be burned on a bootable compact disc and then be installed on the computer. Therefore, just insert the CD-ROM and boot. The installation program will guide through the installation process. When the installer is now being started, the license key needs to be entered. A valid license key can be obtained by registering under qnx.com. Should the given license key be valid, the installation should not pose any problems and continuous.

Handling

One of the first positive things to notice, is the extreme short amount of time that QNX needs to start-up and even less to shut-down. A normal start-up routine takes approximately 30s while it takes only 5 seconds to shut down the system. That is extremely quick and maybe the fastest I have ever seen on an operating system.

To login, the username and password need to be specified. For the first time start up, the username is root without any password. When the system is fully started up, we can see on the left hand-side a configurable menu/program list. Here the most important tools like File Manager, Editor, Terminal and multiple system configurations can be started. Another possibility to access to all the configurations and programs is to click on the left bottom corner on Launch. This opens a new menu and the destination application can be chosen.

[11] Address: http://www.qnx.com/developers/hardware_support/index.html

To organize files and documents, the Photon File Manager is a very easy to use program. Of course an editor is installed as well and it can be opened with the command ped. To open a file for editing, right-click on the file and then choose open with, now enter the short name of the target application (ped for editor).

The terminal can either be started over normal menu navigation, or by a right-click on the desktop with choosing Terminal. For people who prefer a bash shell instead of the normal standard shell of QNX, can download various different shell programs.

Finding a file is not very clever solved. The name of the file or a fraction of the name is sufficient to search a file on the system, but then only the directories and names of the files are listed up. The found files cannot be copied directly to another folder; neither can they be directly opened to edit. It is not possible to search for a term inside a file. Only the filename itself is part of the search criteria.

Even though my test system is running under a Pentium IV processor, sometimes the graphical interface is slowing down a lot! For example, when scrolling down a file in an editor or when scrolling down the shell screen. This is a typical hardware problem. Somehow, the system does not support 100% the graphic card. It is working, but not without any problems. On the official QNX site, there is a list available of supported hardware. Unfortunately, the graphical controller of my test system was not listed up…

Like in almost every UNIX-based system, all the devices have to be mounted to the system. Not like under Windows, where the operating system detects automatically all devices. To use the floppy disk, we need to mount it by typing the following commands:

```
bash-3.1# mount –t dos /dev/fd0 /mnt/disk
```

The first parameter /dev/fd0 is the source and the second /mnt/disk is the target directory to use the floppy drive. Once the mounting is done, we can check the configuration by typing:

```
bash-3.1# mount
/dev/hd0t79 on / type qnx4
/dev/fd0 on /mnt/disk type dos (fat12)
```

As it is shown above, the floppy drive is well connected to /mnt/disk. Of course almost all the floppy drives are supported under QNX. A little bit different it is for USB data storage devices. For a test of your hardware, check the compatibility list. To install for example a Kingston memory stick, the following commands need to be typed:

```
bash-3.1# io-usb –d uhci –v
bash-3.1# devb-umass cam pnp verbose &
[1] 675876
bash-3.1#
```

The output above appears, when no USB memory stick is actually attached to the USB port. But when it is connected, the output looks different:

```
bash-3.1# io-usb –d uhci –v
bash-3.1# devb-umass cam pnp verbose &
[1] 544802
bash-3.1# Path=0 – QNX USB Storage
 target=0 lun=0   Direct-Access(0) – Kingston DataTraveler 2.0 Rev: 1.00

[1]+   Done              devb-umass cam pnp verbose
```

When this appears, hit the enter key and the USB memory stick is successfully mounted on the system. A similar manner is needed to use a CD-ROM. Therefore, type the following commands:

```
bash-3.1# mount /dev/cd0 /mnt/cd
```

Should there be connected more than one CD-ROM. The second drive can be found under /dev/cd1, the third under /dev/cd2…

Support

Because QNX might not be as popular as Linux or other open source operating systems, the support on the general Internet is very poor. It is very hard to find specific and detailed information about QNX. For engineers, it takes sometimes to much time to search and find reasonable information. Therefore QNX provides their costumers with two different support levels:

- Standard support
- Priority support

The standard support is free and consists of the following possible solutions:

- Global Help Centre (Email, phone or fax support)
- Automatic Product Updates (Major releases delivered on CD-ROM)
- Private Standard Support Newsgroup (Talk to support engineers 24/24h
- QNX Developers Network (QDN, active community of developers)
- Searchable Knowledgebase (for solutions of known problems)

The Priority support is **not** for free. Three different levels of priority support can be chosen:

- Bronze: Escalation of critical problems
- Silver: Dedicated support engineer, support for custom hardware, private newsgroup
- Gold: Support for multiple projects, free training and design review, top priority for resolving issues

The needed help or support level is always depending from the user skills. Another support solution is a help library, installed automatically with the operating system. This library can be opened over the Launch => Help navigation. On the left hand-side, there is a gap named Find, enter here a command or function name to receive more information about it. This help file is very useful and exactly equal to the online help files on qnx.com.

At the beginning of my project, two engineers from QNX visited us and gave us a quick introduction into QNX and its possibilities. Sean Meroth and Franck Vancoellié were supporting me personal as well. Of course I would like to thank them here quickly for their patience and knowledge!

QNX Momentics

The QNX Technology Development Kit (TDK) is using Eclipse as base application. Eclipse itself has been developed by IBM and is now running under a not-for-profit consortium of software industry vendors as QNX. I use the TDK to analyse the timer tests to find out why the timer takes from time to time 6 instead of 5ms. Before we can start the IDE, we need to create a different boot image.

4. First of all, the turning kernel has to be instrumented to get the information into Eclipse. Therefore, the file qnxbase.build under the folder /x86/boot/build has to be edited. The expression procnto need to be changed into procnto-instr.
5. The next step is the construction of the file system IFS (Image File System). Open a shell and execute these commands:

```
bash-3.1# cd /x86/boot/build
bash-3.1# mkifs qnxbase.build qnxbase.ifs
```

6. Then the constructed file system has to be copied into the start up folder. The start up folder is doubled, once for the safe and stable system (/.boot) and once for evaluation configurations (/.altboot). This has the advantage of always having a working kernel in the backhand. To copy this file and restart the computer, execute:

```
bash-3.1# cp qnxbase.ifs /.altboot
bash-3.1# shutdown
```

7. When the computer is starting up, the screen says: Hit Esc for .altboot. This can be executed by hitting the Esc key
8. To start the QNX Development Environment, type:

```
bash-3.1# qconn
bash-3.1# qde &
```

9. The software starts up and we can choose Window -> Open Perspective -> Other -> QNX System Information

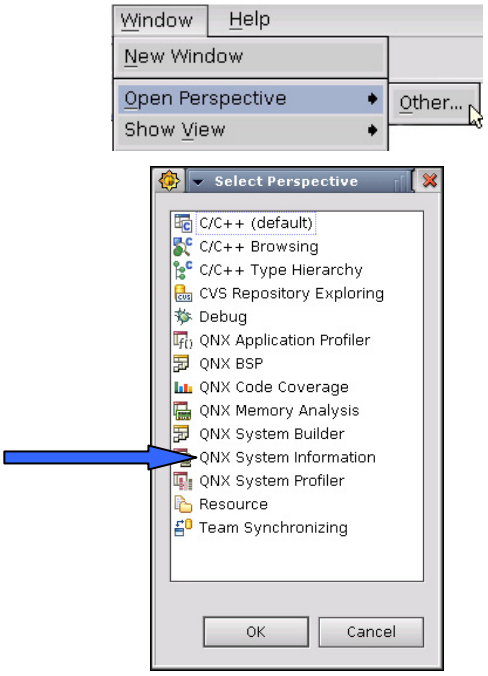

Illustration 18: Open QNX System Information

10. Then the target system has to be connected to the IDE: File -> New -> QNX Target System Project

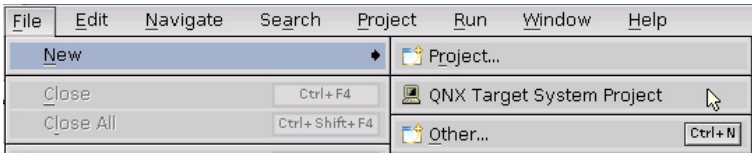

Illustration 19: Create new project

11. Define a name for the target system and activate: Use local QNX connector

12. The next step is the opening of the QNX System Profiler: Window -> Open Perspective -> Other… -> QNX System Profiler

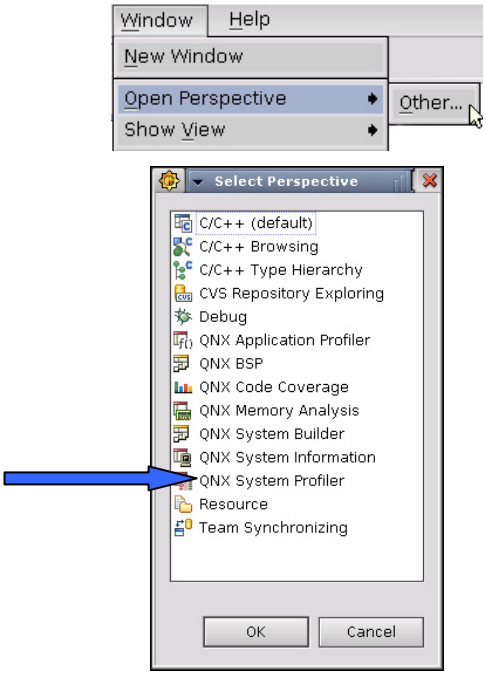

Illustration 20: Open the QNX System Profiler

13. Last and final step is the set up for a kernel tracing. Right-click inside the Target Navigator on the corresponding name and choose Kernel Events Tracing:

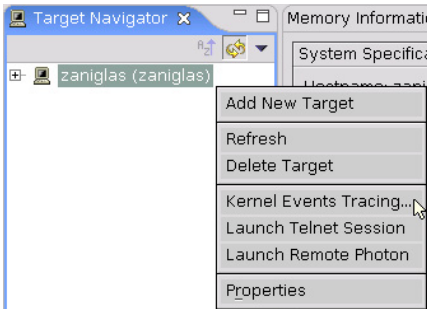

Illustration 21: Open kernel event tracing mode

Choose now for how long the kernel tracing should take place:

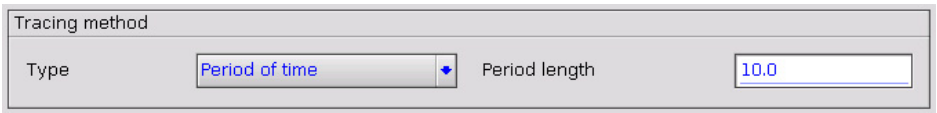

Illustration 22: Define how long the system profiler should be running

Before the test is now being started, make sure a shell is opened to start the target program as well. When everything is set up, the Finish-button of the System Profiler Configuration can be hit and directly afterwards the target program needs to be started. As soon as the configuration of the system profiler is finished, the program is tracing the kernel and saving these results in a file. Double clicking on the log file in your project navigator can now open this file. The result file's extension is .kev. Once the file is opened, we can either take a look at what happened with the diagram above or the Trace Event Log underneath; all the currently

running processes are listed up and we can check out for every process itself when it has the processor and when it is blocked. In this diagram can be found the name latency. This is in fact the output of the test program timer_latency and we can see that the process is being active every 5ms, because timer_latency is testing the timer on 5ms slices.

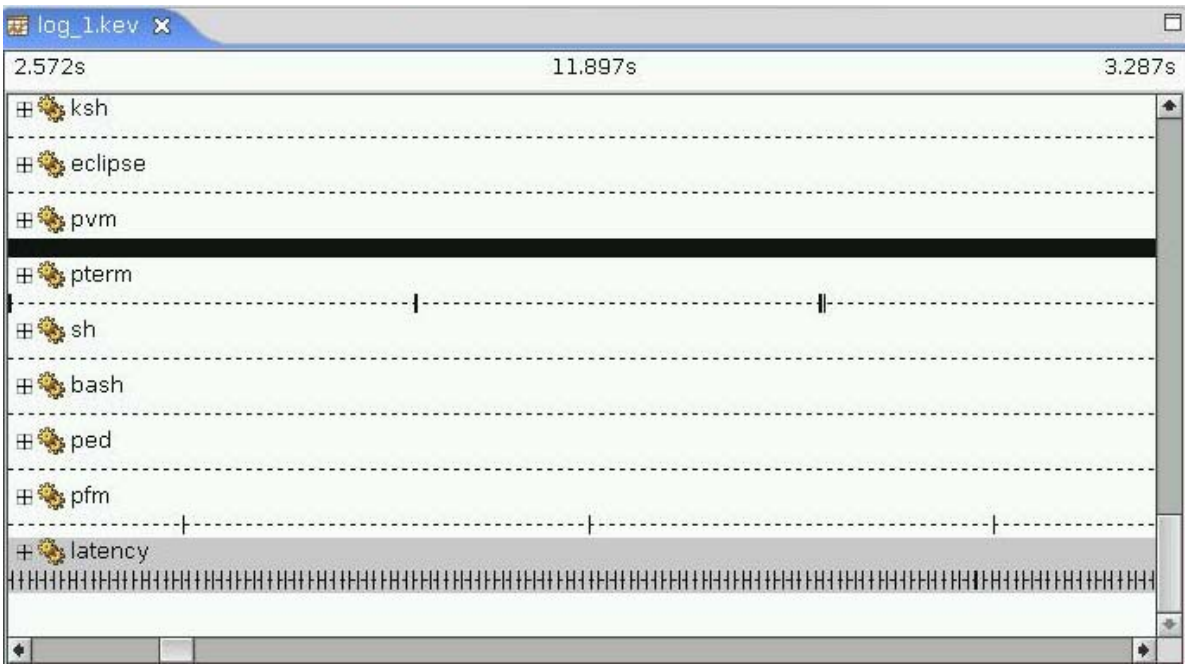

Illustration 23: System Profiler output

All the running processes are listed in this diagram. The name of the process is written on the left hand-side and the timeline is defined on top of it. In this example, the focus is set between 2.572s and 3.287s. The time in the middle is the actual position of the cursor on the timeline. The cursor can as well be used to measure the time between two processes activities. Each process receives the CPU only for a short amount of time. This is shown with the black vertical lines on the horizontal timeline. With a left-click and hold, the focus can be pulled between the time slices. The measured time is written now in the top-middle of the diagram. In the following example it was between 1.516s and 1.521s and equals a time slice of 4.997ms:

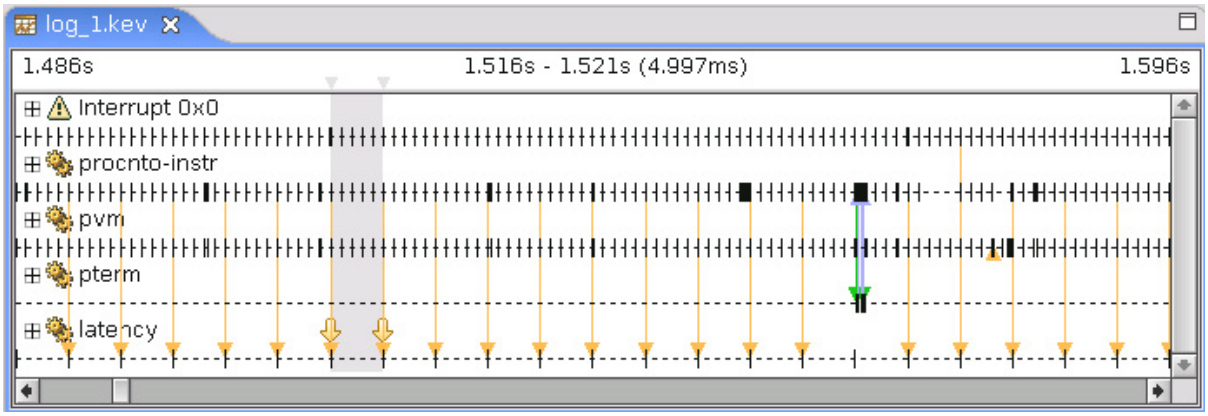

Illustration 24: Time measurement with cursor

The diagram can now be modified to increase the visibility of the picture. The following commands can be used to:

Zoom in and zoom out, it is as well possible to select a part of the diagram and then to zoom this particular zone in

The log file is being represented once with the diagram and a second time with a table of the trace log events. Inside this table, or inside the diagram, these arrows can be used to go to the next or the previous event

With this command, it is possible to show the inter process communication between the different running processes (IPC)

The diagram type can be changed as well between: CPU Usage, CPU Activity, Element activity and Timeline, the mode shown above

Two possible other modes can be seen on the pictures underneath:

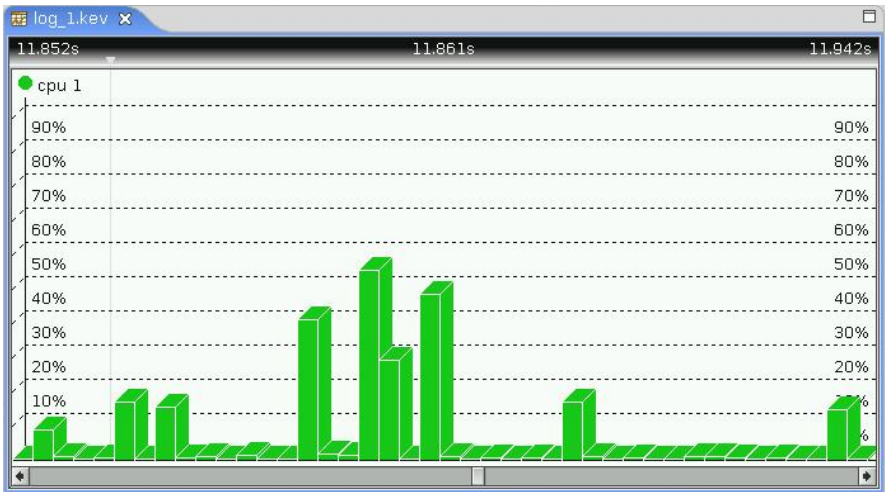

Illustration 25: Diagram of CPU activity

Illustration 26: Diagram of CPU usage

Of course the list of processes consists of all the running processes. But in most cases, only one or some few processes are under observation, therefore, the unwanted processes can be removed by a right-click:

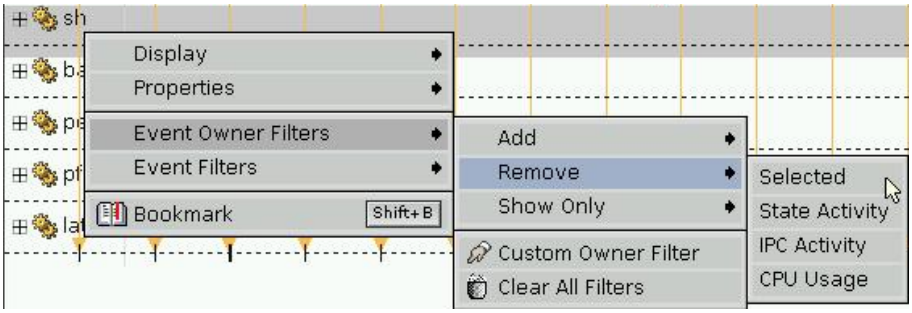

Illustration 27: Define an Event Owner Filter to remove a process

The general handling of the System Profiler is very easy. Unfortunately, the use of System Profiler to analyze process behaviour needs a lot of resources. A 5 seconds log file is using incredible 17MB of space. That is a lot and the work with large log files is getting more and more difficult, the larger the files get. When observing a process problem or behaviour, the focused time slice is mostly not bigger than some milliseconds. To find a couple of milliseconds within a 5 seconds record takes a long time. Once such a behaviour problem is found, it can be bookmarked. So next time the file is being opened, the critical part is quickly found again. To bookmark an interesting point on the timeline, click right and choose Bookmark:

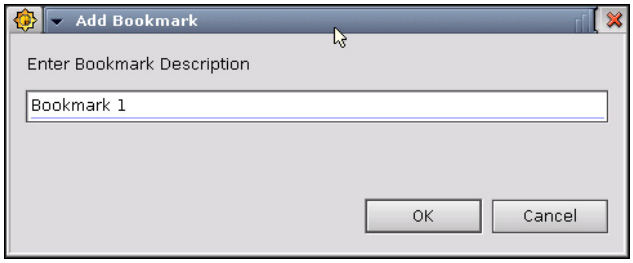

Illustration 28: Add a new bookmark to a log file

When the bookmark is added, it can be read out again from the window underneath the diagram under the name Bookmarks:

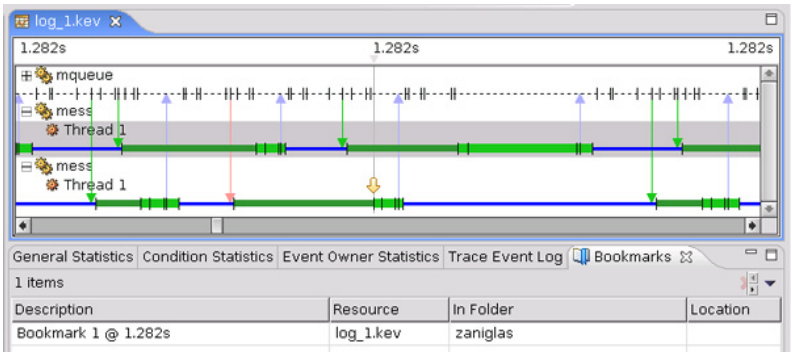

Illustration 29: Choose out of the saved bookmarks

Tick size

QNX Neutrino is polling its processes after a defined time slice. This time slice is normally 1ms but can be changed if needed. To change the tick size, we need to execute a couple of commands as the following peace of source code shows:

```
return_value = ClockPeriod(id,              //clock ID
                           _clockperiod,    //contains the period to set the clock to
                           _clockperiod,    //to store the current period
                           reserved);       //not used => set to 0
```

When the return_value is bitter than zero, then the tick size has been changed successfully. Otherwise, the return_value is equal to –1. The full source code to change the tick size can be found in the appendix.

Test programs

This evaluation will be held on two different systems, because it was not possible to install QNX on a multi-processor machine, a mono-processor machine had to be used as well. The information about the used machines is listed below:

Solaris & Linux host machine		**QNX host machine**	
processor	multi	processor	mono
vendor_id	GenuineIntel	vendor_id	GenuineIntel
model name	Intel(R) Xeon(TM) CPU 3.20GHz	model name	Intel ® Pentium(R) IV 1.70GHz
cpu MHz	3202	cpu MHz	1698
RAM	2048MB	RAM	384MB

Each operating system has its advantages and its weaknesses. To get a better overview on different systems, it is comfortable to implement various test programs to test out the performances and to finally compare them underneath each other. In this chapter, three different test routines are being used. Two of them are already implemented for other systems and have not to be ported. The third routine will be composed of two test programs. One of them will test out the semaphores and the other one the message queue performances. The tests are being observed and time measured to get a final comparison between the systems.

In fact, LibTIM is a tool to execute multiple test routines at once. It has been created for DEC OSF1, IRIX, Linux and other systems. It now has to be ported to QNX and Solaris but first a little introduction:

LibTIM

Because the simulation software programs need precise timers, this library is used to test the performance of different timers available on the preferred operating system. Timers can either be implemented by software (SOFT) or by hardware (HARD). When implemented in software, the system native timers are being used to perform the test. While implemented by hardware is using special OS external (PCI) timer cards. On multi processor systems, the effect of using different CPU's can be tested as well by binding the application to a CPU. But the access to such timers is different on each operating system, which makes the portability more complicated.

This library is now able to communicate with different timers (SOFT and HARD) on different operating systems. Its basic requirements are listed below:

- Access to all different timer types of the platform (POSIX) and hardware timers on the PCI bus (RTOM, MPIO, RCIM, ...)
- Simultaneous access on multiple types of timers
- Dynamic detection of all available timers
- Masking principles for the implementation of the different timers

Once installed, the library is able to perform the following tasks:

- to use a timer (periodic or oneshot, including Microsleep)
- to use FreeCounters (access to internal timer)

To evaluate the timer performances of an operating system, two different programs are used:

- Timer_stability: calculate relative and absolute precision of the timer
- Timer_glich: calculate the relative timer precision over a longer time

Both programs use the LibTIM library:

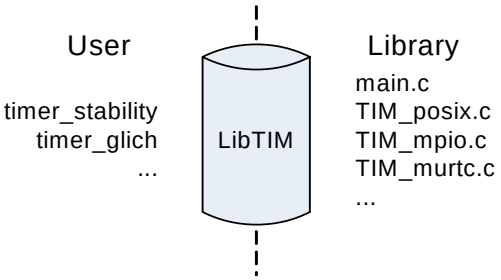

Illustration 30: Use of the library LibTIM

How the program timer_stability works, is described in the following diagram:

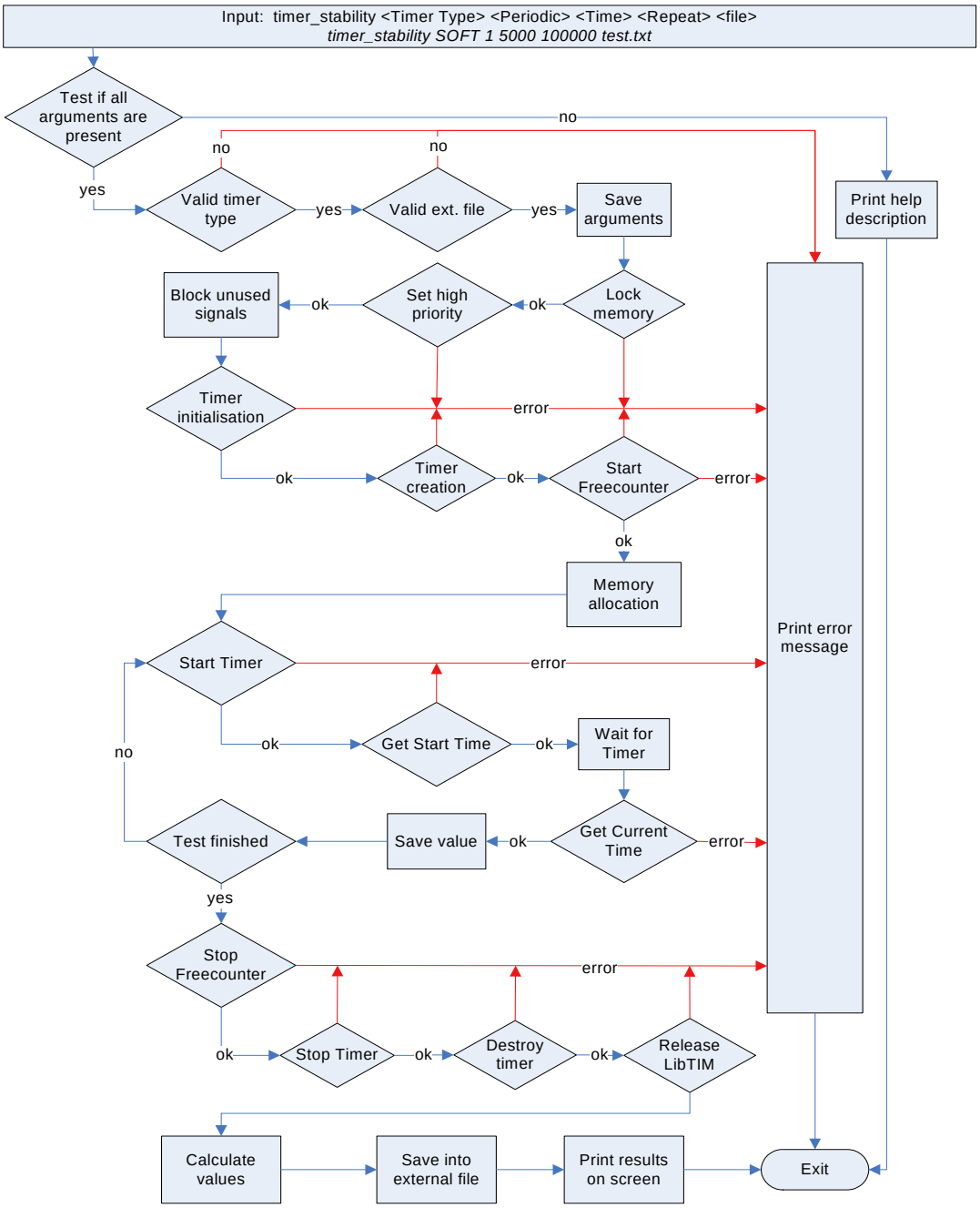

Illustration 31: timer_stability functionality overview

- Valid timer type: check if the specified timer is either HARD or SOFT
- Valid ext. file: check if the file to save the results is a valid file
- Set priority: set process priority to Real Time
- Block unused signals: only allow timer termination signal to be received from process
- Memory allocation: allocate memory depending on how many times the test needs to be repeated
- Wait for timer: wait until the timer sends the termination signal to the process

Between timer_stability and timer_glich is only one principal difference: the treatment of the result data. The picture below describes how the time is being measured:

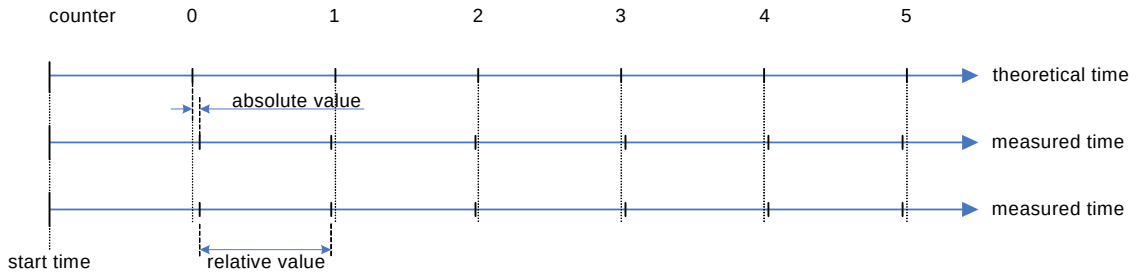

Illustration 32: Diagram of time measurement

To calculate the absolute value, the difference between the theoretical time and the measured time is taken. This allows taking a look on the long time behaviour of the timer. If the diagram shows a continuously increasing line, it means that the delay of the timer in comparison with the theoretical time is getting bigger and bigger. Is the result a flat line instead, means the timer is precise and has no delay. Is the result a continuously decreasing line, it means that the timer is finishing to early (running ahead).

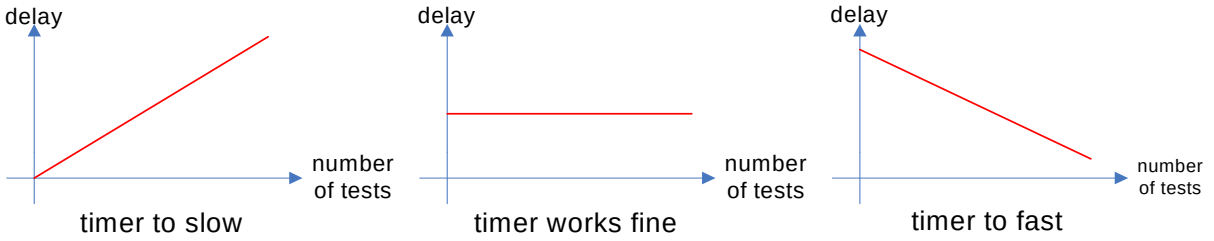

Illustration 33: Absolute value diagrams

The measured time is calculated with this formula:

$$\text{start time} + \{timeslice \cdot (counter + 1)\}$$

The relative value (calculated in timer_glich) only considers the time between two timer signals. This allows visualizing the precision of the timer. On these diagrams can often occur a kind of a mirror behaviour. The reason for such phenomena is the operating system itself. When a timer is not precise, lets say time n is 200ns to short, so the operating system will try to equal this mistake with making the next time 200ns longer. So on the diagram can always be recognized a similarity between too short and too long times.

Another version of the same test is actually working exactly the same. The only difference is the treatment of the result data and the amount of executions. A normal test takes approximately 2min or 24'000 measurements but the long time test takes 6h and executes in this time 4'320'000 measurements. Of course not all the time information can be saved and therefore, the results are being classified:

- Timer precision underneath 100 μs of precision
- Timer precision between 100 μs and 200 μs of precision
- Timer precision between 200 μs and 300 μs of precision
- Timer precision between 300 μs and 400 μs of precision
- Timer precision between 100 μs and 500 μs of precision
- Timer precision between 500 μs and 1100 μs of precision
- Timer precision over 1100 μs of precision

The actual time is not being saved, only the class to get an overview on long-term tests. To execute these tests on Solaris and QNX, the source code needs to be changed, because these systems do not support all the commands used on Linux or other operating systems. All the modified source files can be found on the supported DVD-ROM.

Generic created Test (BGenerique)

To test the performance of an operating system, it is better to perform some long-term tests than just short ones. Therefore Airbus has designed a test program called BGenerique. It is a script in fact and executes timer_stability and timer_glich multiple times in a row. So the system can dynamically be tested over a long time without any human support.

It can be used to perform a simple timer or a more complex test to simulate the simulation software. But in fact, it is the same as the simple test (timer_stability), just with more than one process and different timer properties. The applications listed below are being simulated with its specific properties (Processor bounding when executed on multiprocessor machine):

Application name	Periodic / Oneshot	Time slice	Bound on CPU	Repetitions
ASPIC	Periodic	5 ms	1	240’000
AFDX	Oneshot	16 ms	0	75’000
IONET	Oneshot	10 ms	0	120’000
VISUAL	Oneshot	40 ms	0	30’000
MOTION	Oneshot	20 ms	0	60’000

The standard test takes 20 min. So for every application, it’s necessary to calculate the number of repetitions with this formula:

$$\text{Number of repetitions} = \frac{20 \cdot 60 \cdot 1000}{Timeslice}\left[\frac{\min \cdot \sec \cdot mili\sec}{ms}\right]$$

So instead of only one timer_stability application, now 5 of them are running at the same time. The diagram below shows the different parameters and its files to configure the BGenerique test:

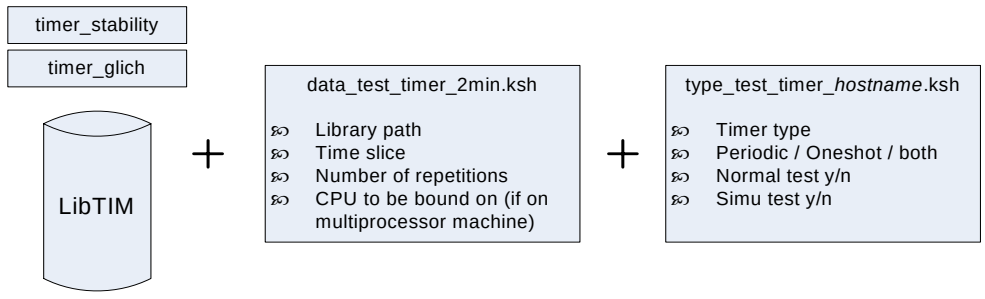

Illustration 34: Parameters and files to configure BGenerique

```
Library path: path of the LibTIM library
Time slice: value of the timer, mostly 5ms
Number of repetitions: how many times should the timer be executed
Timer type: software bound or hardware bound timer (SOFT or HARD)
Periodic/Oneshot: the timer can either be restarted directly without stopping him
(Periodic) or stop the timer and then restart him from new (Oneshot)
Normal test: is a normal timer test with only one timer running at once
Simu test: more than one timer are running concurrently at once
```

Once the test has been executed, we obtain an html-file with all the important data and the diagrams. This makes it very comfortable to test multiple systems on its performance. To

perform this tests on Solaris and QNX, the code needed to be changed and can be found on the supported DVD-ROM. The result diagrams can be found in the appendix.

Timer Latency (only QNX)

The results from the timer_glich and timer_stability tests under QNX are not really satisfying. That was reason enough to implement a new test program but this time, QNX timers are used instead of POSIX compliant timers. In QNX as well as in Solaris, the timer granularity can be adjusted: the tick size for polling the processes can be defined.

Besides of timers, we need as well a lot of other real time specific performances as message queues and semaphores. So the second method is dealing with these services, implemented specifically for each system. But before we can start, we need to take a closer look to some other methods, for example to allocate shared memory.

Shared memory

When we want to use semaphores with more than one process (as it usually happens) then we have to put the semaphore into shared memory, so that it can be accessed by both of the processes. The implementation of shared memory seems to be a little complicated, but isn't really:

```
fd = shm_open("/bolts", O_RDWR|O_CREAT, 0777)

fd:       -1: an error occurred; nonzero otherwise
"/bolts": name of the shared memory
O_CREAT:  create a new message queue, if doens't exist on this name
O_RDWR:   send and receive over the same message queue
0777:     permission bits for the memory


result = ftruncate(fd, sizeof(sem_t))

result:   0 for success, -1 if an error occurs
fd:       file descriptor
sizeof(sem_t):     the length that you want the file to be, in bytes


addr = mmap(0, sizeof(*addr), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0)

addr:
0:        start address of the shared memory
sizeof(*addr):     size of the shared memory
PROT_READ:memory may be read
PROT_WRITE:        memory may be written
MAP_SHARED:        the calling process shares the mapping
fd:       file descriptor
0:        offset
```

Fork system call

The fork system call allows a user to duplicate a running process, and to let run afterwards both processes together. The new created process is called "child" and the original process "parent". When the fork system call is executed, it returns two different values: 0 to the child process and the child's process ID to the parent.

```
pid = fork();
pid:      0 for the child, the child's process ID for the parent
```

So now all the needed functions are explained and timer_latency can be discussed. This program should change the timer granularity and then measure the amount of ticks to determine afterwards the precision and behaviour of the timer. With the following command line, the CPU frequency can be read out:

```
cpu_freq = SYSPAGE_ENTRY(qtime)->cycles_per_sec;
```

Because the process should not be disturbed from anything else than interrupts, the highest possible priority level needs to be found out and then defined in the system configuration:

```
param.sched_priority = sched_get_priority_max( SCHED__RR);
ret = sched_setscheduler (0, SCHED__RR, &param);
```

If the priority change was not successful, ret is equal to –1, otherwise different. The next step is the configuration of the timer parameters:

```
clkper.nsec = 1’000’000 x clock_period;
ClockPeriod(CLOCK_REALTIME, &clkper, NULL, 0);
```

The nano seconds of the timer obtain the value 1 million. This defines a timer granularity of 1ms (clock_period is equal 1). Then the parameters are being saved in the system configuration with the function ClockPeriod. Before we can use the timer, we have to create it:

```
timer_create(CLOCK_REALTIME, &event, &timer_id);
```

Should the return value be equal to –1, the timer cannot be created. Otherwise, it is different from –1. The next step will be the definition of the timer value. In this example, we would like the timer to take 5ms, therefore, timer_length is equal to 5:

```
timer.it_value.tv_sec        = 0;
timer.it_value.tv_nsec       = 1000000 * timer_length;         //oneshot
timer.it_interval.tv_sec     = 0;
timer.it_interval.tv_nsec    = 1000000 * timer_length;         //periodic
```

When finally all parameters are set, the timer needs to be started and returns –1 when the timer could not be started successfully:

```
timer_settime(timer_id, 0, &timer, NULL);
```

Now, the timer is set up and we can wait for the pulse indicating that the timer run out:

```
pid = MsgReceivePulse ( chid, &pulse, sizeof( pulse ), NULL);
```

When the timer sends a pulse, the clock cycles can be read out and therefore it is possible to calculate the passed time. Unfortunately, the result was the same as with timer_glich or timer_stability. Once in a while, the timer is not executing it’s 5ms but 6ms. After discussing this topic with Sean Meroth from QNX, we found out, that this has something to do with the POSIX implementation of QNX in QNX 6.3. Therefore, it does not matter which test I use, the result is always the same:

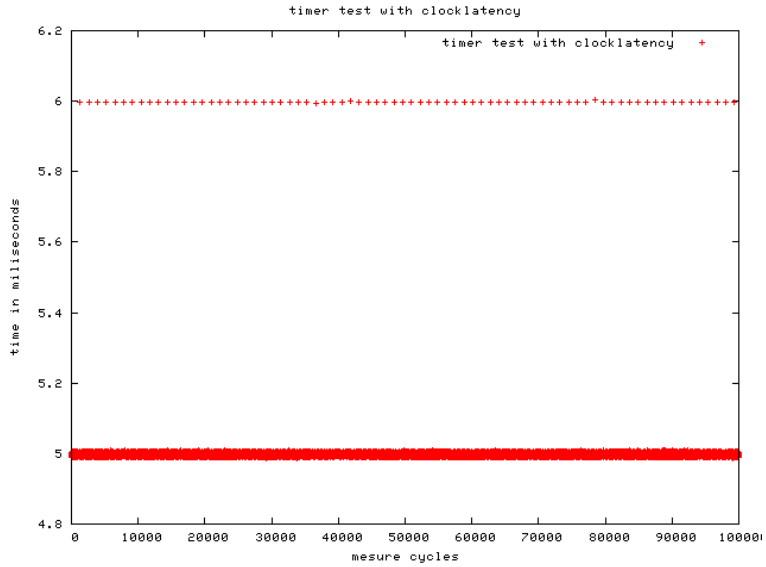

Illustration 35: Result of timer_latency (QNX)

If the timer granularity should be decreased on 0.5ms, the result is similar, but this time, we have regularly results of 5.5ms instead of 5ms. The timer comes to late for one tick slice:

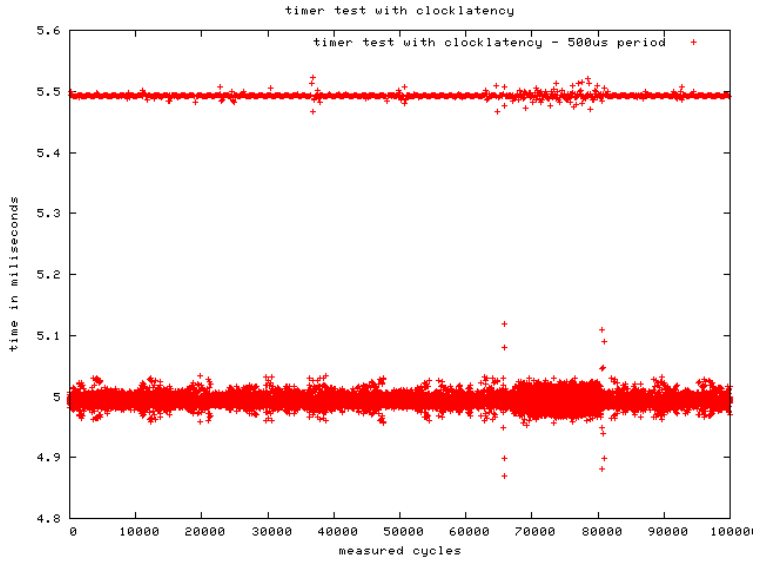

Illustration 36: Result of timer_latency (QNX) with timer granularity of 0.5ms

Message queues

Message queues are being used to communicate between at least two processes. They can be seen as a direct connection between the processes to exchange data. In this example, data is being send from the parent to the child process and back again. As soon as the parent is receiving the message from the child process, the time is measured and the used time to exchange two messages is being calculated:

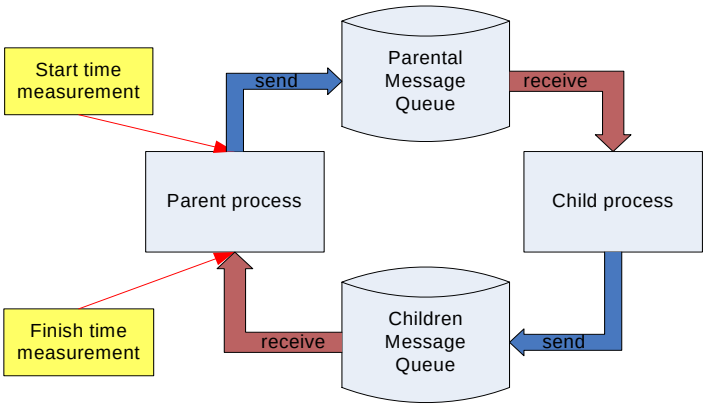

Illustration 37: Time measurement for message queue test program

This turn is now being executed multiple times to receive an average value for the time needed to exchange a message between two processes over message queues. To use a message queue, we need to open and configure it before:

```
mq_child = mq_open("/queue_child", O_CREAT|O_RDWR, S_IRWXU, NULL);

mq_child: message queue descriptor
"/queue_child":    name of the message queue
O_CREAT:  create a new message queue, if doens't exist on this name
O_RDWR:   send and receive over the same message queue
S_IRWXU:  read, write, execute/search by owner
NULL:     no message queue attributes to use
```

Once the message queue is opened, we can send or receive messages:

```
result = mq_send(mq_child, msg_ptr, size, 0);

result:   -1: message send failed; 0: otherwise
mq_child: message queue descriptor
msg_ptr:  pointer on the outgoing message
size:     size of the message (characters)
0:        priority of the message

rec_size = mq_receive(mq_father, msg_ptr_receive, msg_len, NULL);

rec_size: -1: message receive failed; 0: otherwise
mq_father:message queue descriptor
msg_ptr_receive:   pointer on the incoming message
msg_len:  message size of the given queue
```

When a message queue is built up, the system is responsible for the service to work. Should the program be finished and we do not close and unlink the message queue, it remains in the system until the next restart. Therefore we need to execute the following two methods:

```
result = mq_close(mq_child);

result:   -1: close failed; 0: otherwise
mq_child: message queue descriptor

result = mq_unlink(mq_child);

result:   -1: unlink failed; 0: otherwise
mq_child: message queue descriptor
```

Once the test program has finished measuring the time, we can now show the result in a graphic. The diagrams of these tests can be found in the appendix.

The only difference between the source code for Solaris and for QNX is the time measurement. The rest of the code is identical and can be found on the DVD-ROM. QNX is using an own function to find out how many CPU ticks elapsed since the system was started

up. By knowing the CPU frequency, the elapsed time can be calculated out of two tick amounts. This is implemented as it follows:

```
stamp[0] = ClockCycles()
…
stamp[1] = ClockCycles()
addToFile(stamp[0], stamp[1], i, pFile)
…
void addToFile(uint64_t val1, uint64_t val2, int turns, FILE * pFile)
{
          //variable declaration
          float res;
          uint64_t diff, clocks;

          //calcualte difference between time stamps
          diff = val2 - val1;
          //get clock granularity from system
          clocks = SYSPAGE_ENTRY(qtime)->cycles_per_sec;
          res=(float)diff/clocks;                    //corresponds to one full cycle
          fprintf(pFile,"%d  %f\n", turns+1, res);    //write data into file
}
```

Under Solaris, it is a little bit different. Here we use the POSIX function gettimeofday to find out how many seconds elapsed since the start-up of the system. This allows us easy to calculate the time used to transmit data over message queues on Solaris:

```
gettimeofday(&old,0);       //get the actual time
…
gettimeofday(&new,0);       //get the actual time
…

void addToFile(struct timeval old, struct timeval new, int turns, FILE * pFile)
{
          //variable declaration
          float result;
          //calculate how many useconds one cycle took:
          result = ((int)new.tv_usec - (int)old.tv_usec) *  0.000001;
          fprintf(pFile,"%d  %f\n", turns+1, result); //save value into file
}
```

Semaphores

Another very common tool in real time programming are semaphores. But semaphores are not used to exchange data between two or more processes, but to control access of multiple processes on one resource or memory space. By defining the start value of the semaphore, we can decide how many processes have permission to use the resource. For example, when we have the typical producer-consumer situation, where the producer is writing data into memory space and the consumer is reading the data out to continue its treatment. But as soon as they both try to read out or respectively write data in the memory, we do have an access violation problem and the data can be lost. Therefore, we can limit the amount of processes and guarantee like this, that only one consumer or producer is using the memory at once. In this test program, we have 2 processes and the start value of the semaphore is set to 1 (single access). Because we want to use the semaphore with two processes, we need to create a shared memory space and place the semaphore into this memory space. So both processes have the permission to access the semaphore. Under QNX, there exist two possibilities of using semaphores: named and unnamed semaphores. Named semaphores are much easier to initialize and use afterwards but unnamed semaphores are being treated much faster. That’s why I decided to use the unnamed semaphores. The test program can be described with the graphic underneath:

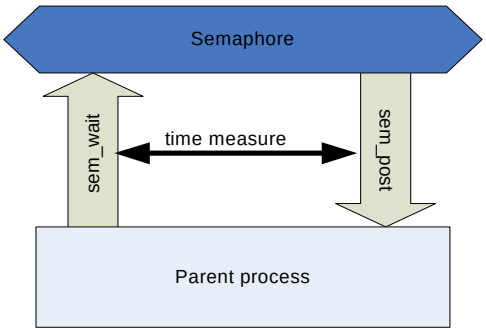

Illustration 38: Time measurement for the semaphore test program

Because it takes very less time to get or post a semaphore, the time measured corresponds to a full access (get and post).

```
sem = (sem_t *) addr;

sem:      semaphore descriptor
(sem_t*): typecast on semaphore type
addr:     name of the shared memory
```

Once the semaphore is placed in the shared memory, we can initialize it:

```
result = sem_init(sem, 1, 1)

result:   0: semaphore initialized successful, nonzero otherwise
sem:      semaphore descriptor
1:        use a shared semaphore
1:        start value of the semaphore
```

If the value of the semaphore is bigger than zero, the process will receive the permission to access and decrement the value. But when the value is already smaller than zero, the demanding process will be blocked as long as another process is releasing (putting) the semaphore. In this case, the value is being incremented.

```
result = sem_wait(sem)

result:   -1: an error occurred; 0: semaphore decremented
sem:      semaphore descriptor

result = sem_post(sem)

result:   -1: an error occurred; 0: semaphore incremented
sem:      semaphore descriptor
```

Before we can leave the program, we need to destroy the semaphore and to unlink the shared memory. When QNX is started up, prcnto is being executed. This is actually the kernel of QNX and is therefore responsible to manage the semaphores. Unfortunately, once the semaphore is set up, it is absolutely mandatory to unlink the semaphore. When this is not done, we will not be able to initialize another semaphore afterwards by using the same semaphore descriptor. So if the program should crash during its execution, the computer has to be started up from new to unleash the semaphore. The proper release of the semaphore and the shared memory can be done like this:

```
sem_destroy(sem)
sem:      semaphore descriptor

shm_unlink(„/bolts“)
“/bolts”: name of the shared memory
```

Sorting algorithm

Once the time measurement tests have been executed successfully, the result files content all the times measured before. To process this information, these result files need to be reorganized. Therefore, all the time measures are read in and then being sorted in increasing manner. At the same time, it can be found out, how many times the same value was measured. Here shown on a simple example:

Program input:

```
1  0.000048
2  0.000003
3  0.000002
4  0.000002
5  0.000005
6  0.000002
7  0.000005
8  0.000002
9  0.000002
10  0.000002
```

Program output:

```
0.000002  6
0.000003  1
0.000005  2
0.000048  1
```

The sorting algorithm itself is very simple. It has to be repeated for each value in the result file and can be described as the picture below shows:

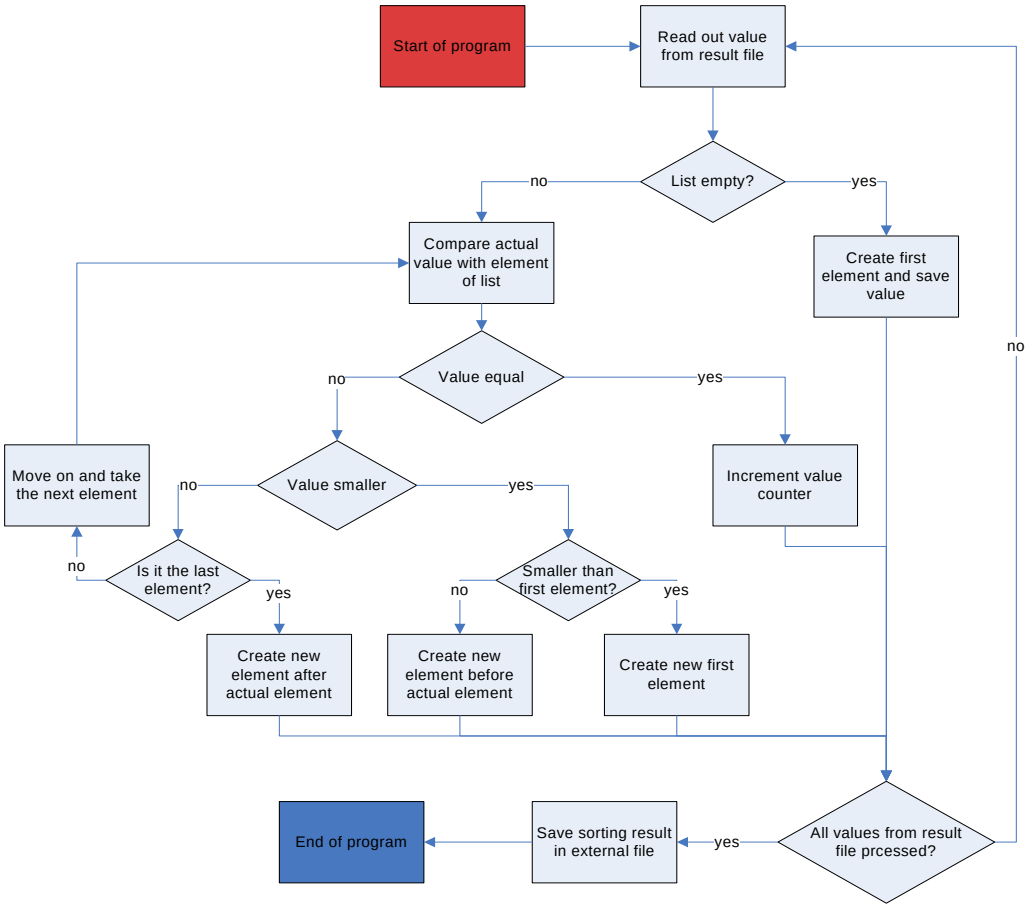

Illustration 39: Sorting algorithm to sort the linked list

The detailed program code can be found in the appendix. The linked list is composed of elements. These elements are defined as structures and consist of the following variables:

```
struct list_item {
         struct list_item * previous, * next;        //save navigation
         float value;              //measured time value
```

```
        int amount;                 //count amount of equal values
        };
```

Should occur a value that cannot be found in the list, a new element has to be added. This happens by calling the subroutine with the according information about the time measure result:

```
//prototype
item * add(float result);

//subroutine
item * add(float result)                        //subroutine to add a new element
{                                               //to the linked list
        item * temp;                                    //temporary item
        temp = (item *) malloc(sizeof(item));           //allocate memory space for item
        if(temp == NULL)                                //test if memory allocation
                                                        //failed
        {
                printf("\nmalloc problem!");
        }
        else
        {       //memory allocation successfull!
                if(print) printf("...created!\n");
                temp->value = result;           //write result value in structure
                temp->amount = 1;               //save amount of equal values
                temp->next = NULL;              //ground the navigation pointer
        }
        return temp;                            //return the new created item
}       //subroutine call

l_item = add(result);                           //add new element
```

Once the element has been created, it has to be place at the right place in the linked list. Therefore, the result saved inside the element is proved and judged according to five different rules:

- The actual value is equal to the value of the list element
- The actual value is smaller than the first elements value => is the smallest value
- The actual value is smaller then the value of the list element
- The actual value is bigger than the value of the list element
- The actual value is bigger than the last elements value => is the biggest value

The new element pointers properties have to be set according to the above-defined rules. The different situations can be seen on the images below:

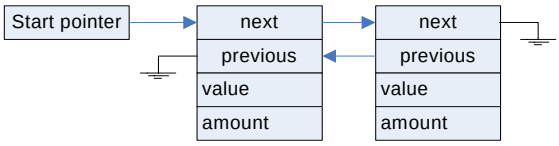

Illustration 40: Existing linked list consisting of two elements

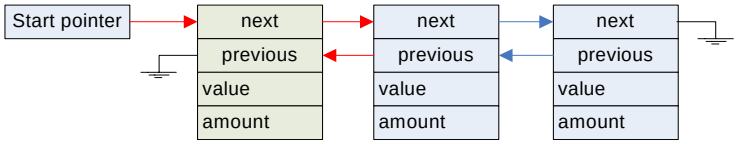

Illustration 41: Insert a new element in front of the list

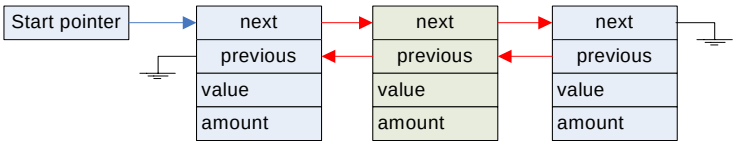

Illustration 42: Insert new element in the middle of the list

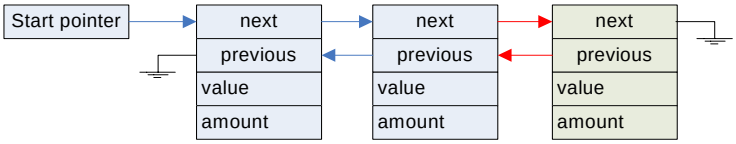

Illustration 43: Insert new element at the end of the list

As you can see in the code earlier on, every element is placed in its own memory space, allocated by malloc(). Once the file is processed, the memory has to be released. The algorithm to delete a linked list like this can be seen in the diagram below:

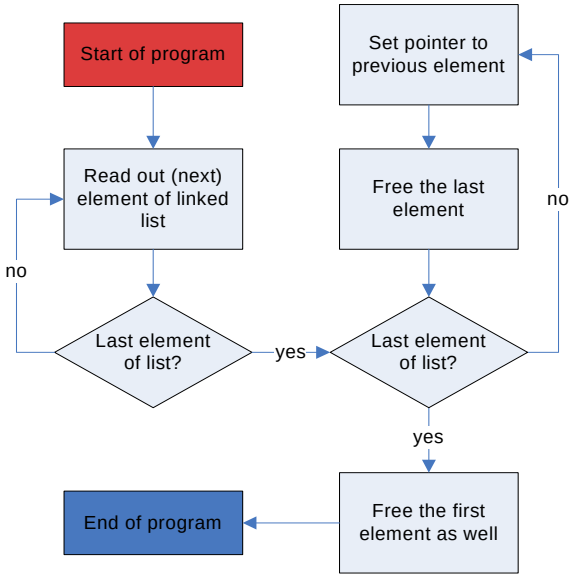

Illustration 44: Algorithm to delete a linked list

Portability

What does portability mean? Portability is the ability of recompiling source code from another operating system without having to make to many efforts. The higher the portability is, the easier it is to compile a program from another operating system. This ability is very important for the simulation software department, because they are not focused on only one operating system.

The portability will be tested in three different stages: LibTIM, BGenerique and finally DSS. Inside the LibTIM are hidden two different programs: timer_stability and timer_glich. The exact manner of functionality of these two programs can be read under the point Test programs. Of course it was not easy in the beginning to understand how these programs work, because they are completely designed to be portable. In reality this means a lot of make files and scripting. But after a couple of days, porting timer_stability and timer_glich was possible without a really big effort.

The second stage is the BGenerique, which is in fact nothing else than a specific use of the LibTIM library. The only difference is it, that the programs are being executed multiple times to receive different results. Are the results similar and stable, the system is working fine as it should.

That was the simple part of the portability, but it's getting more complex when recompiling a complex application. The DSS (Distributed Simulation Software) is used to realize a distribution of the needed CPU power over the Ethernet. Once the system is set up, data are being exchanged between the machines. This happens with ACE & TAO, a version of real time CORBA. Unfortunately, it was not possible to find an already compiled binary source file for this application and the compilation of the source code failed. I was working close together with the responsible engineer at Artal, the subcontractor, which realized the software. After we discussed this topic together with my tutor, we decided to leave the DSS away. The main point of my final diploma work is not to compile a program, but to evaluate the operating system.

Even though QNX and Solaris should be 100% POSIX compliant, a lot of problems occurred during the compilation of DSS. When the applications are simple and easy to understand, it should be no problem to port them in a short amount of time, but it gets extremely complicated and time consuming with complex, big programs. With a lot of help from Franck Vancoellieé and Sean Meroth, two engineers from QNX, I arrived to port ACE & TAO, but to run DSS, we need as well a third tool called DiSCo. This is an application developed by Artal and the compilation unfortunately failed. Within my diploma work, it is not possible to port such a large application to a new operating system. After discussing this topic with Jean-Marie Calluaud and the responsible engineer of the subcontractor, we decided to skip the DSS. Note that this decision wasn't made because it isn't possible to port the application, but too time consuming!

Conclusion

Before a new operating system can be installed, the hardware has to be checked on compatibility. Because most of the computer systems are currently running on Windows or Linux, other systems like QNX or Solaris are not as famous and therefore not as popular on the market as their concurrence. A logical consequence is the low availability of drivers and support applications for such systems. It is therefore recommended to buy the hardware to fit the software and not as mostly usual the other way around. Once the system is checked, the installation should not represent any problems.

The following table represents an overview on the performance of the observed and evaluated systems:

		Solaris 10	QNX 6.3	Linux 2.6.9
OS	Information density on Internet	good	bad	excellent
	Technical support	good	excellent	good
	Available ported applications	bad	good	excellent
	Driver support	bad	bad	good
Simu. Software	POSIX	good	good	good
	Threads	good	good	good
	Timer	good	excellent	good
	Message Queues	good	excellent	excellent
	Semaphores	excellent	good	good

The first executed test was the BGenerique. This application is used to test the system native timer performances. An existing Linux system is used as a reference between the result diagrams. To make a final comparison easier, each test is being judged with points between 1 (worst) and 3 (best). So let's compare the different diagrams:

Oneshot, absolute

This test is executed in one-shot mode, means the timer is started, executed, stopped and then started again. Linux and QNX show both a similar behaviour of a timer leaking behind the theoretical value (Illustration 45,Illustration 49 and Illustration 53). While the delay on Linux is always between 0 and 1ms, QNX is between 1 and 2ms. This behaviour is not so important, because the relative value counts more. Solaris is different, as you can see on Illustration 49, the delay remains stable around 1.39ms. This value is composed of the time the operating systems needs to wake up the timer and to initialize it. Linux 3pt, Solaris 2pt, QNX 1pt.

Oneshot, relative

The relative value is the actual measured duration of the timer to test out its precision. In these diagrams, often a kind of a mirror-effect can be observed. Should timer n be 2ms to short, then the operating systems is setting the timer n+1 2ms longer than the usual value. In this test, Linux is terminating on the last place, because its timer is often not precise. The result spectrum goes from 2ms to early until 1ms to late (Illustration 46)!! Better is Solaris, where the precision is better and the maximal delay moves between 1ms to late and 1ms to early (Illustration 50). But the best result achieved without doubt QNX (Illustration 54). The timer is incredibly precise on 5ms but has from time to time a malfunction of 1ms in its results. This comes from the POSIX implementation of QNX. Even though I tried to find a reason for this effect with the System Profiler, I could not find any difference between a

perfect timer call and the wrong ones. But it has to be said, that this irregularity occurred only about 20 times on 24'000 executions. QNX 3pt, Solaris 2pt, Linux 1pt.

Periodic, absolute

Test number three was the periodic restarting of the timer. This means, that the timer is not stopped after it's termination but restarted directly. The initialization of the timer is not necessary any more, because it is not stopped since the last run. On every executed timing cycle, Linux is increasing its delay with approximately one microsecond. On 24'000 executes, the timer is leaping 24ms behind (Illustration 47). The Solaris timer is still behind the optimal theoretical time (Illustration 51). The time amount in one-shot mode was almost 1.4ms and now in periodic mode 700 μs. The timer needs those 700 μs to wake up and start then the execution. On the bottom of the diagram are some compensation results in sight. These are modified timers to reach a constant level of delay. QNX shows a very similar result to the one-shot absolute value (Illustration 55). The timer is again always behind the theoretical value, between 1 and 2ms. Contrarily to the Linux result, the maximal value is always smaller than 2ms. Solaris 3pt, QNX 2pt, Linux 1pt.

Periodic, relative

And the last and final test of the BGenerique is measuring the precision of each timer in periodic mode. These values represent the delay of each timing cycle. Somehow the Linux results of this test are not satisfying at all (Illustration 48)! The majority of results have a delay of 1ms and the rest of them are trying to compensate the frequently made mistakes. The values are located between a delay of 200 μs and 1.8ms. Much better is the Solaris result (Illustration 52). Almost all results are precise around the theoretical time and only a few results occur outside this range. The maximal delay is around 1ms, while the minimum is –1.5ms. But clearly the best result shows again QNX (Illustration 56). Only a couple of values are on a delay of 1ms. Again are occurring from time to time some 1ms delays. The reason of this delay is to be searched in the implementation of POSIX. QNX 3pt, Solaris 2pt, Linux 1pt.

The following test should show the performance of QNX and Solaris on real-time components like message queues and semaphores. To get a better overview on the performances, the tests are being executed multiple times in a row. These test have only been executed on QNX and on Solaris, so the maximum points to get are two.

Semaphore

This program is using a shared semaphore between two processes. The time measured is the time that a process needs to get and to put a semaphore. Let's start with Solaris and take a look at the Illustration 61 to Illustration 64. The dominant value is 2 μs but as well a lot of 5 μs values. The highest occurring value is around 8ms!! QNX on the other hand is very stable and absolutely fast (Illustration 65 to Illustration 68)! The dominant value is as well 2 μs but the result field is much less spread out. The maximal value reaches 234 μs out of 10'000 cycles! Linux is very close to the other two test results as well (Illustration 57 to Illustration 60) and is only a little bit slower than the concurrence. The highest consumed time reaches 60ms, which is a very high value! QNX 3pt, Solaris 2pt, Linux 1pt.

Message Queue

The final and last program is a program to test out, how long the operating system needs to send and receive a message through a message queue. To start with Solaris again, one

complete cycle takes in most of the cases a value around 33ms. The biggest result is 2.944ms and the fastest message exchange has taken 24 μs (Illustration 73 to Illustration 76). The majority of the values on QNX are 13ms, while none of them is smaller and the biggest has an amount of 17ms (Illustration 77 to Illustration 80). QNX is almost twice as quick as Solaris! Better is only Linux and that for such a short amount of time. Most of the values are between 2 and 3ms! Just a glimpse faster than QNX. Linux 3pt, QNX 2pt, Solaris 1pt.

All the obtained points have to be counted together now:

```
Linux:    3+1+1+1+1+3 = 10 points in total
Solaris:  2+2+3+2+2+1 = 12 points in total
QNX:      1+3+2+3+3+2 = 14 points in total
```

The over all result speaks for QNX in this technical evaluation. The real-time components are working much faster and even more importantly more stable and precise. Solaris was only able to exceed QNX on the BGenerique tests, where the absolute value is just not precise enough. Otherwise, QNX is clearly ahead and offers a reliable and precise real-time operating system. Besides technical innovations, both operating systems come along with a gorgeous design and a clever surface. Even though the systems are based on Unix, it is very easy for a beginner to find what he is looking for. Both designs are well structured and allow a fast and effective working environment.

I'm very grateful to have the possibility to absolve my final diploma work abroad. It was always a great dream for me, to work for Airbus and to see how these wonderful aircrafts learn to fly. Through my diploma work, I could get a glimpse into Airbus and its daily efforts to develop modern and secure aircrafts. I would specially thank to Jean-Marie Calluaud, my tutor, for his support and confidence he gave me through this time. A great thank you as well to the other members of EYYWOC, Gilles Denat, Fabien Depailler and François Trebosc. During the compilation of ACE & TAO, I was working as well together with Francis Versavel from Artal Technologies as well as with Sean Meroth and Franck Vancoellié from QNX Software Systems.

Toulouse, 15.02.2007

Alexander Gruber
Trainee

Illustration Index

Illustration 1: Range/Capacity Diagram for the different Airbus airplane types 6
Illustration 2: Airbus Europe with zoom on Toulouse 6
Illustration 3: Organization of EYYW department 8
Illustration 4: Why interest in QNX and Solaris? 8
Illustration 5: Historical tree of Solaris 10 10
Illustration 6: Key elements of the Solaris 10 kernel 12
Illustration 7: Kernel threads, processes and lightweight processes 14
Illustration 8: Scheduling classes of Solaris 10 14
Illustration 9: Solaris installation levels 16
Illustration 10: Historical tree of QNX 6.3 19
Illustration 11: Instrumented kernel of QNX 6.3 20
Illustration 12: Organization of the QNX kernel 21
Illustration 13: System architecture of VxWorks 21
Illustration 14: System architecture of NT, Unix or Linux 22
Illustration 15: System architecture of QNX Neutrino 22
Illustration 16: Thread priority and interrupts 23
Illustration 17: Priority change of thread while sporadic scheduling 24
Illustration 18: Open QNX System Information 27
Illustration 19: Create new project 27
Illustration 20: Open the QNX System Profiler 28
Illustration 21: Open kernel event tracing mode 28
Illustration 22: Define how long the system profiler should be running 28
Illustration 23: System Profiler output 29
Illustration 24: Time measurement with cursor 29
Illustration 25: Diagram of CPU activity 30
Illustration 26: Diagram of CPU usage 30
Illustration 27: Define an Event Owner Filter to remove a process 31
Illustration 28: Add a new bookmark to a log file 31
Illustration 29: Choose out of the saved bookmarks 31
Illustration 30: Use of the library LibTIM 34
Illustration 31: timer_stability functionality overview 35
Illustration 32: Diagram of time measurement 36
Illustration 33: Absolute value diagrams 36
Illustration 34: Parameters and files to configure BGenerique 37
Illustration 35: Result of timer_latency (QNX) 40
Illustration 36: Result of timer_latency (QNX) with timer granularity of 0.5ms 40
Illustration 37: Time measurement for message queue test program 41
Illustration 38: Time measurement for the semaphore test program 43
Illustration 39: Sorting algorithm to sort the linked list 44
Illustration 40: Existing linked list consisting of two elements 45
Illustration 41: Insert a new element in front of the list 45
Illustration 42: Insert new element in the middle of the list 46
Illustration 43: Insert new element at the end of the list 46
Illustration 44: Algorithm to delete a linked list 46
Illustration 45: BGenerique result Linux, oneshot, absolute result 55
Illustration 46: BGenerique result Linux, oneshot, relative result 56
Illustration 47: BGenerique result Linux, periodic, absolute result 56
Illustration 48: BGenerique result Linux, periodic, relative result 57

Illustration 49: BGenerique result Solaris, oneshot, absolute result 57
Illustration 50: BGenerique result Solaris, oneshot, relative result..... 58
Illustration 51: BGenerique result Solaris, periodic, absolute result..... 58
Illustration 52: BGenerique result Solaris, periodic, relative result 59
Illustration 53: BGenerique result QNX, oneshot, absolute result..... 59
Illustration 54: BGenerique result QNX, oneshot, relative result 60
Illustration 55: BGenerique result QNX, periodic, absolute result 60
Illustration 56: BGenerique result QNX, periodic, relative result 61
Illustration 57: Linux – Semaphore – 100 cycles..... 61
Illustration 58: Linux - Semaphore – 1’000 cycles 62
Illustration 59: Linux - Semaphore - 10'000 cycles 63
Illustration 60: Linux - Semaphore - 100'000 cycles 63
Illustration 61: Solaris - Semaphore – 100 cycles..... 64
Illustration 62: Solaris - Semaphore – 1’000 cycles 64
Illustration 63: Solaris - Semaphore - 10'000 cycles..... 65
Illustration 64: Solaris - Semaphore - 100'000 cycles..... 66
Illustration 65: QNX - Semaphore - 100 cycles..... 66
Illustration 66: QNX - Semaphore - 1'000 cycles 67
Illustration 67: QNX - Semaphore - 10'000 cycles 67
Illustration 68: QNX - Semaphore - 100'000 cycles 68
Illustration 69: Linux - Message Queue - 100 cycles..... 69
Illustration 70: Linux - Message Queue - 1'000 cycles..... 69
Illustration 71: Linux - Message Queue - 10'000 cycles..... 70
Illustration 72: Linux - Message Queue - 100'000 cycles..... 71
Illustration 73: Solaris - Message Queue - 100 cycles..... 72
Illustration 74: Solaris - Message Queue - 1'000 cycles 72
Illustration 75: Solaris - Message Queue - 10'000 cycles 73
Illustration 76: Solaris - Message Queue - 100'000 cycles 74
Illustration 77: QNX - Message Queue - 100 cycles 74
Illustration 78: QNX - Message Queue – 1’000 cycles 75
Illustration 79: QNX - Message Queue - 10'000 cycles..... 77
Illustration 80: QNX - Message Queue - 100'000 cycles..... 77

Links

Solaris 10

Everything about Solaris:	http://everythingsolaris.org/
Solaris applications:	http://fr.solaris-x86.org/applications.html
Solaris documentation:	http://solaris-x86.org/documents/guides/
Solaris compatibility list:	http://www.sun.com/bigadmin/hcl/data/sol/
Download for sun packages:	http://www.sunfreeware.com/indexintel10.html
Tips and tricks about Solaris:	http://www.captain.at/programming/solaris-tricks/
LAN card driver:	http://www.skd.de/e_en/support/driver.html?navid=14
Compiler documentation:	http://developers.sun.com/prodtech/cc/compilers_index.html

QNX 6.3

QNX France:	http://www.qnx.com/popups/index.html?topic=665
All about Eclipse:	http://en.wikipedia.org/wiki/Eclipse_%28software%29

QNX documentation:
http://www.openqnx.com
http://www.levenez.com/unix/history.html#08
http://www.cmdl.noaa.gov/hats/insitu/cats/stations/qnxman/
http://www.operating-system.org/betriebssystem/_english/bs-qnx.htm

Official QNX Homepage:
http://www.qnx.com/products/eval/index.html

What is QNX?
http://www.osnews.com/printer.php?news_id=15272
http://www.osnews.com/story.php/15272/What-Is-QNX/
http://www.operating-system.org/betriebssystem/_english/bs-qnx.htm
http://homes.dsi.unimi.it/~pedersin/AD/qnx.pdf#search=%22qnx%20what%20is%22

General information & links

http://www.wikipedia.org

TAO & ACE:
http://www.cs.wustl.edu/~schmidt/ACE.html
http://www.cs.wustl.edu/%7Eschmidt/TAO.html

C++ Forums & Programming:
http://www.cplusplus.com/
http://cslibrary.stanford.edu
http://cboard.cprogramming.com/index.php
http://publib.boulder.ibm.com/iseries/v5r1/ic2924/index.htm
http://mia.ece.uic.edu/~papers/WWW/books/posix4/TOC.HTM

GNU C Compiler & Debugger
http://gcc.gnu.org/
http://sources.redhat.com/gdb/download/onlinedocs/

Books

Solaris 9: the complete reference, Watters Paul, ISBN: 0072223057
Solaris internals: Core kernel components, Richard McDougall, James Mauro, ISBN: 0130224960
Solaris system administrator's guide, Janice Winsor, ISBN: 3893628177
QNX Neutrino RTOS v6.3, System Architecture, QNX Software Systems International Corporation

Appendix

Information about the thesis

w.gFilière / *Studiengang* : **Systemtechnik**

Etudiant / *Student* Alexander Gruber 0041 79 642 68 21	Année scolaire / *Schuljahr* 2005/06	No TD / *Nr. DA* SI/2006/6
Proposé par / *vorgeschlagen von* Jean-Marie Calluaud jean-marie.calluaud@airbus.com +33 561930737		Lieu d'exécution / *Ausführungsort* Toulouse
		Expert / *Experte* Jean-Marie Calluaud

Titre / *Titel*:

Stage Evaluation of Solaris and QNX for aeronautic real time simulation software

Description / Beschreibung:

At the moment, most of the aeronautic real time simulation software is running on Linux/Unix systems. Sun Microsystems has now published a free Solaris version for x86 computer systems, which makes it interesting to evaluate the performance of this system. Although QNX is not available as freeware, it is as well interesting to compare this operating system with those already used.
Therefore, these operating systems have to be studied and installed. Once the systems are working well, C/C++ code has to be compiled and debugged on those systems. When finally all this is established, original Airbus test benches have to be implemented and adapted for those systems.

Objectifs / *Ziele:*

- Installation and study of Solaris and QNX with its development tools for C/C++
- Study of timer performances on those systems
- Study of SMP technology
- Implementation of original Airbus Flight Simulation Real Time Software

Bench Generique (BGenerique) results

BGenerique is a program to test out an operation system with the two timer test programs timer_stability and timer_glich. More information about the content of these programs can be found earlier on in this report under the point Test programs. On the following pages are shown the results of these tests.

BGenerique on Linux

Host name:	Felagund
Date:	18/10/2006
Operating system:	Linux 2.6.9-22.EL
Architecture:	i386

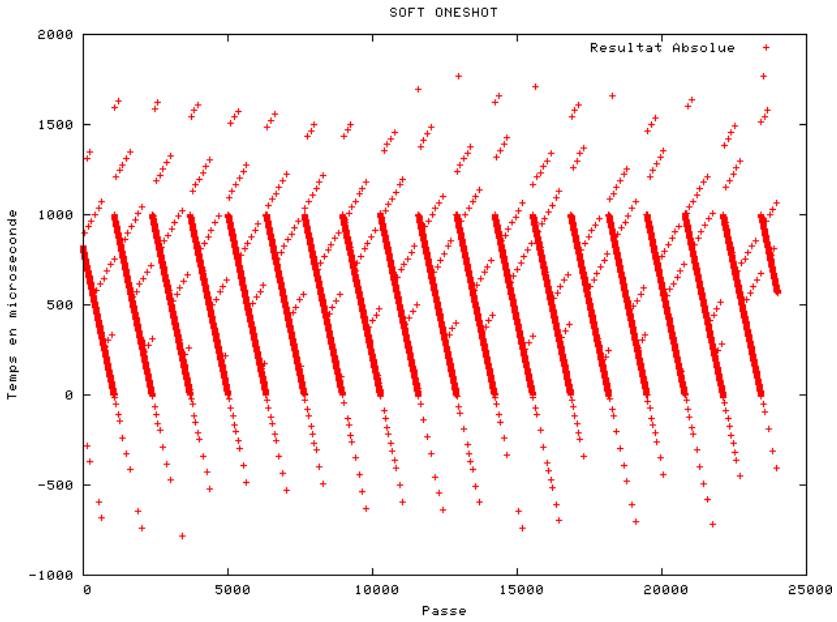

Illustration 45: BGenerique result Linux, oneshot, absolute result

The timer is leaking behind but tries then to minimize the delay. In most of the cases, the delay is smaller than 1ms.

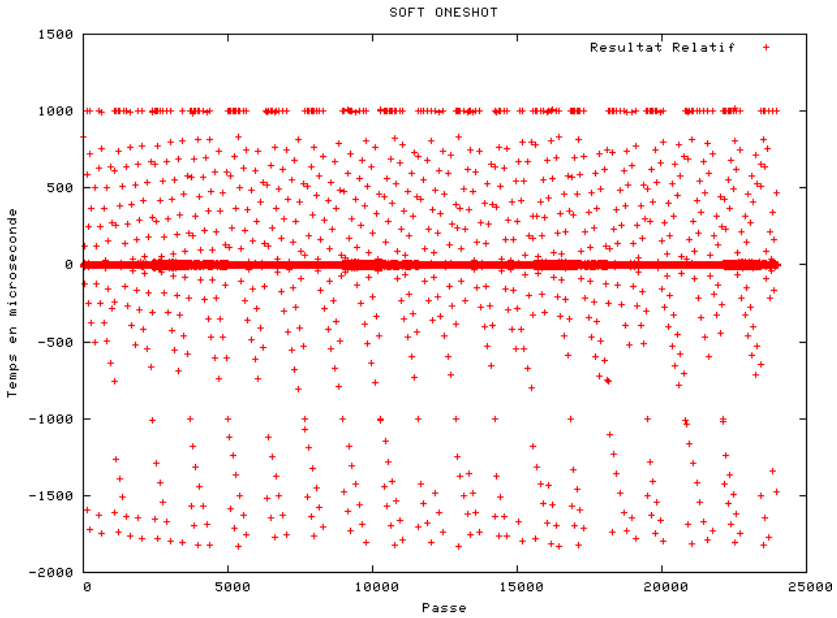

Illustration 46: BGenerique result Linux, oneshot, relative result

The timer is relatively stable and precious. The timer takes for 5ms values between 3.5ms and 6ms.

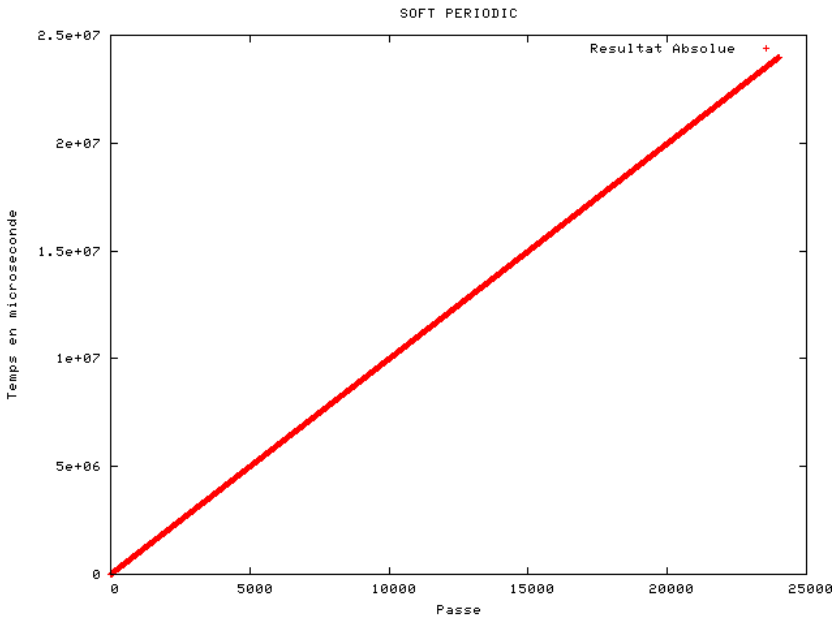

Illustration 47: BGenerique result Linux, periodic, absolute result

This timer has under periodic execution always a little delay. That's the reason why the line is increasing.

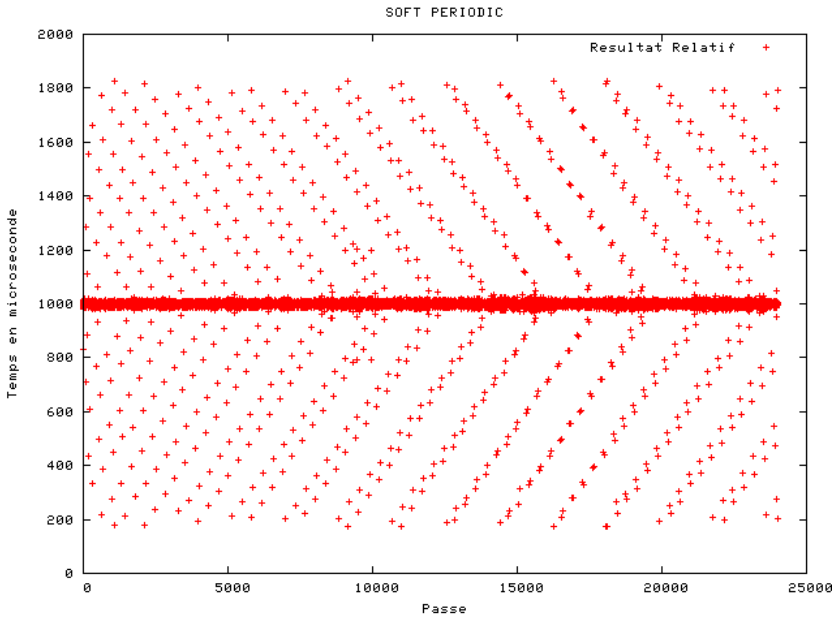

Illustration 48: BGenerique result Linux, periodic, relative result

The look on the relative diagram shows typical corrected timer behaviour. Not only that the timer is very often 1ms to late, but unstable at the same moment.

BGenerique on Solaris

Host name:	Felagund
Date:	03/10/2006
Operating system:	SunOS 5.10
Architecture:	i386

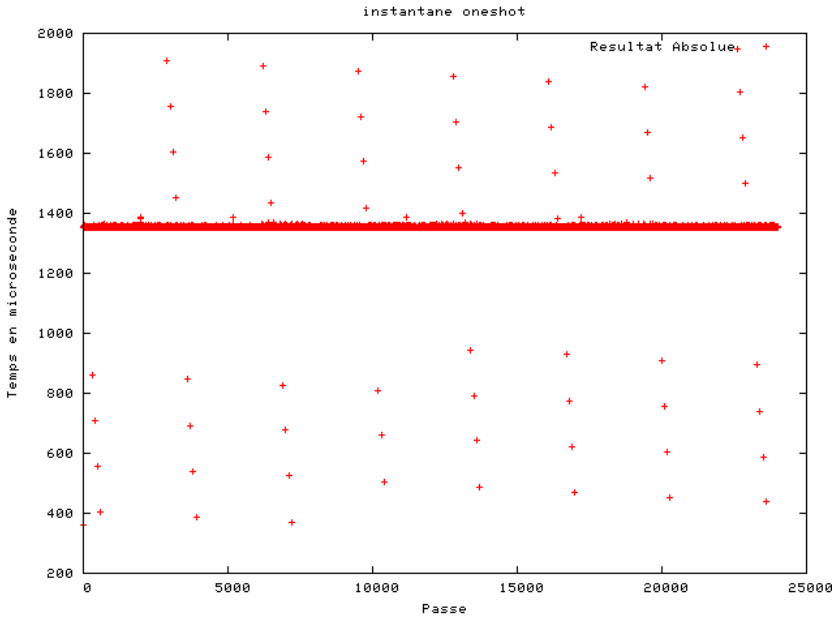

Illustration 49: BGenerique result Solaris, oneshot, absolute result

In the diagram above, the timer has a stable delay of approximately 1.38ms. The timer has to be created and initialized and that takes a certain amount of time.

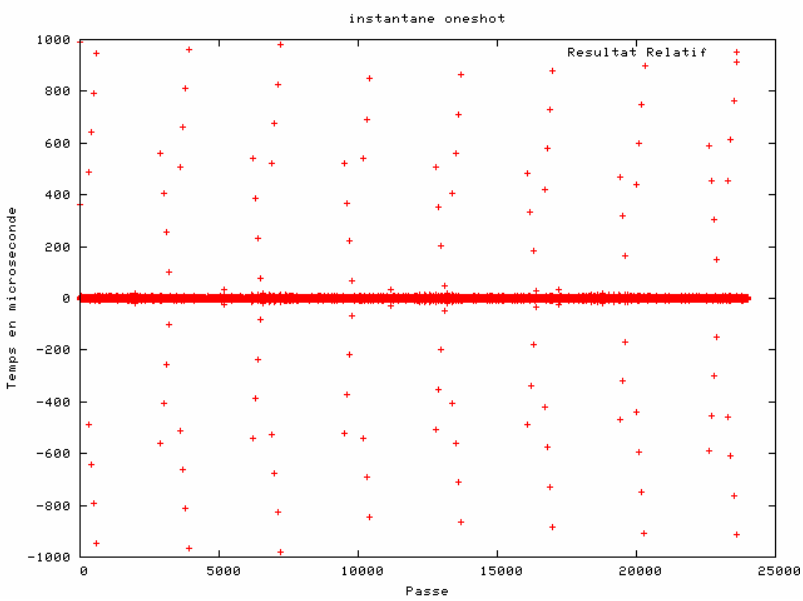

Illustration 50: BGenerique result Solaris, oneshot, relative result

The general timing is not to bad. Most of the values are around 0ms. Typically, the corrections of the operating system can be seen again.

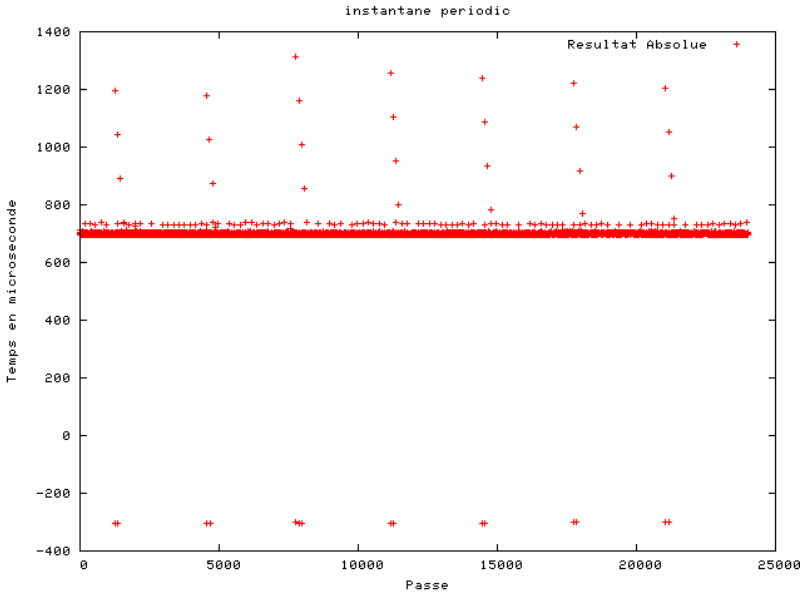

Illustration 51: BGenerique result Solaris, periodic, absolute result

We can see, that between oneshot and periodic, there is a difference of almost 600ms. The difference comes from the initialization. To call and start the timer, the system is using around 700ms.

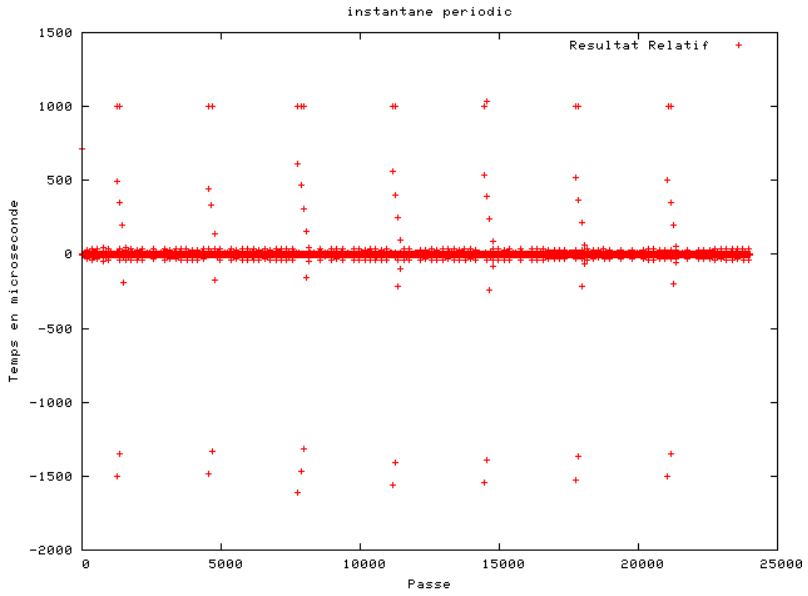

Illustration 52: BGenerique result Solaris, periodic, relative result

The time measurements itself are again pretty stable. On 25'000 measurements are only a few results above or underneath the perfect result.

BGenerique on QNX

Host name:	Zaniglas
Date:	24/10/2006
Operating system:	QNX 6.3.2
Architecture:	i386

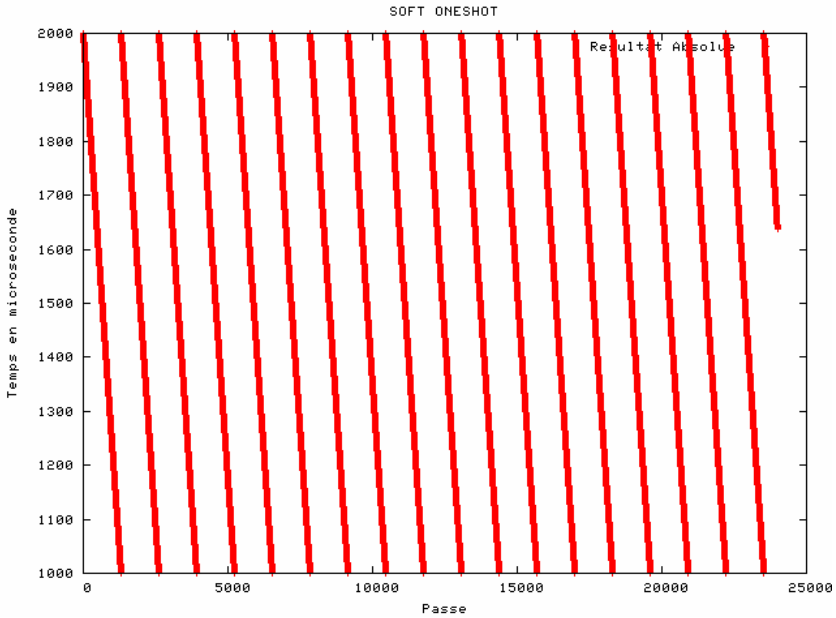

Illustration 53: BGenerique result QNX, oneshot, absolute result

This timer is constantly starting to late and therefore, he is ending to late as well. The delay is always 2ms and then decreasing constantly to 1ms before it flips back to 2ms.

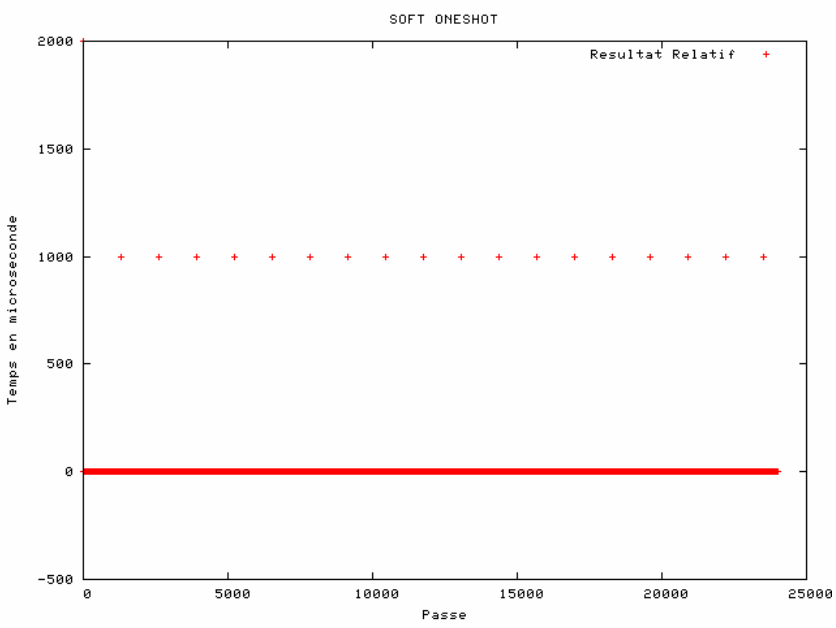

Illustration 54: BGenerique result QNX, oneshot, relative result

Even though the system itself is not stable, the time measured is very stable. Only once in a couple of thousand times, the measured time is not 5ms, but 6ms. This comes from the implementation of the POSIX rules on QNX.

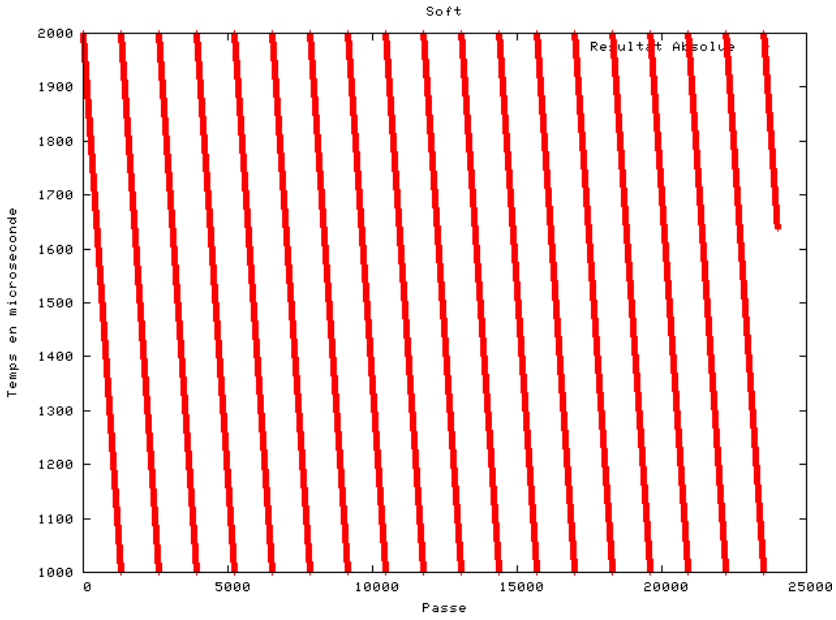

Illustration 55: BGenerique result QNX, periodic, absolute result

It doesn't really matter if the test is being executed in an oneshot or periodic manner. There has to be a reason in the implementation of the operating system as well. A possible solution for such a problem would be the use of interrupts. With an interrupt, we can arrange an electrical interrupt to the core and the kernel is stopped immediately. But with a software timer, there is always the problem of implementation. Even when the timer calls the kernel, the kernel has to finish first his running tasks.

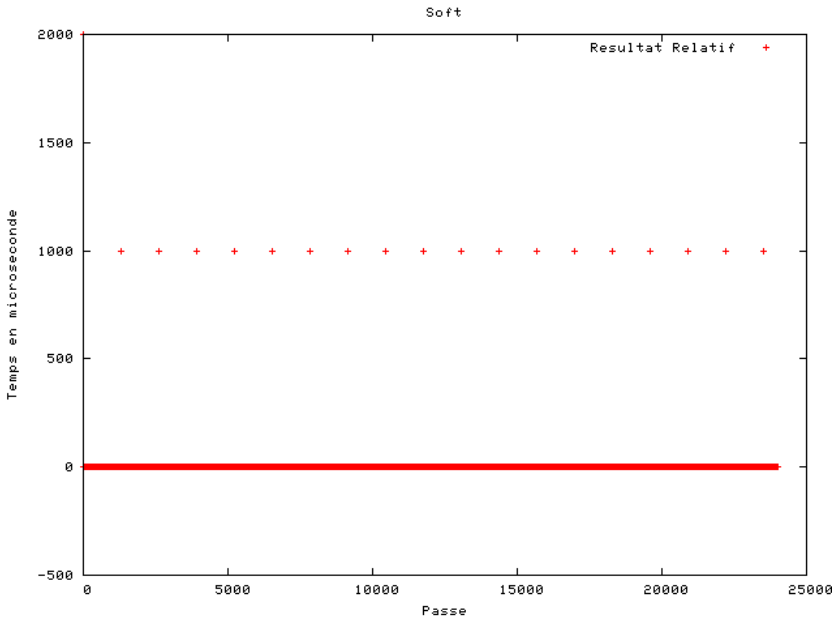

Illustration 56: BGenerique result QNX, periodic, relative result

The relative result diagram is identical to the one-shot test. The measured time is always more or less precise. Why the operating system has a problem from time to time remains a secret.

Semaphore test results

Semaphore test on Linux

Linux - Semaphore - 100		Linux - Semaphore - 1'000			
0.000003	91	0.000003	873	0.000007	1
0.000004	7	0.000004	110	0.000023	1
0.000005	2	0.000005	11	0.000025	1
		0.000006	3		

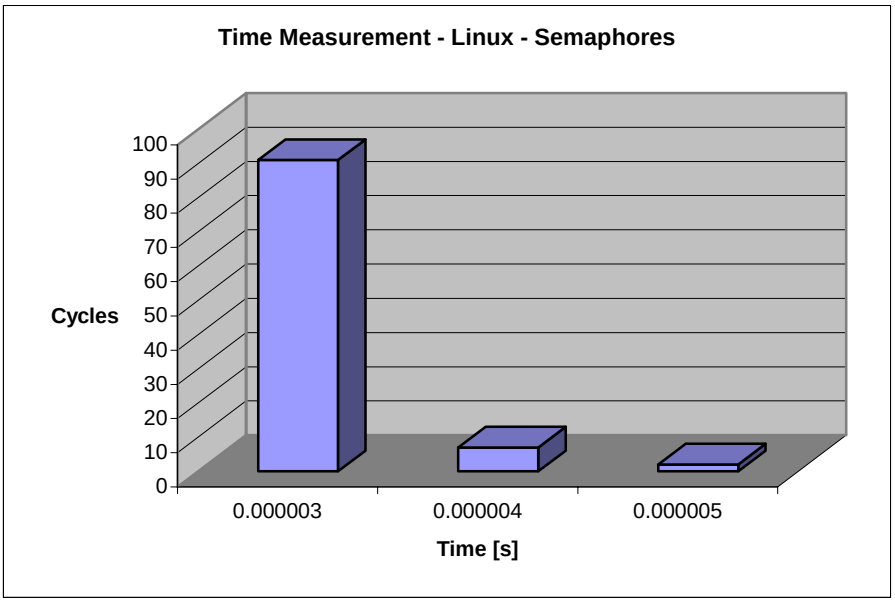

Illustration 57: Linux – Semaphore – 100 cycles

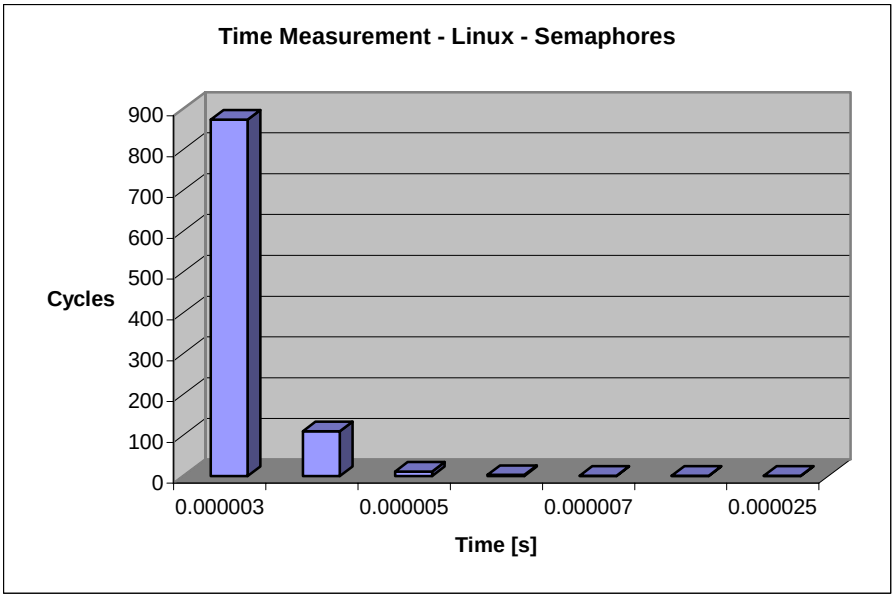

Illustration 58: Linux - Semaphore – 1'000 cycles

Linux - Semaphore - 10'000

0.000002	31	0.000007	1	0.000024	6	0.000030	1
0.000003	8923	0.000009	1	0.000025	8	0.000031	1
0.000004	936	0.000016	1	0.000026	3		
0.000005	60	0.000022	1	0.000027	3		
0.000006	16	0.000023	6	0.000028	2		

Linux - Semaphore - 100'000

0.000002	598	0.000019	1	0.000030	1	0.000080	2
0.000003	90215	0.000021	4	0.000031	1	0.000166	1
0.000004	8158	0.000022	28	0.000032	1	0.000244	1
0.000005	535	0.000023	56	0.000039	1	0.000720	1
0.000006	172	0.000024	28	0.000044	1	0.032040	1
0.000007	11	0.000025	83	0.000046	1	0.040024	1
0.000008	1	0.000026	51	0.000066	1	0.060023	1
0.000009	1	0.000027	20	0.000068	1		
0.000013	1	0.000028	11	0.000070	1		
0.000014	1	0.000029	5	0.000072	2		

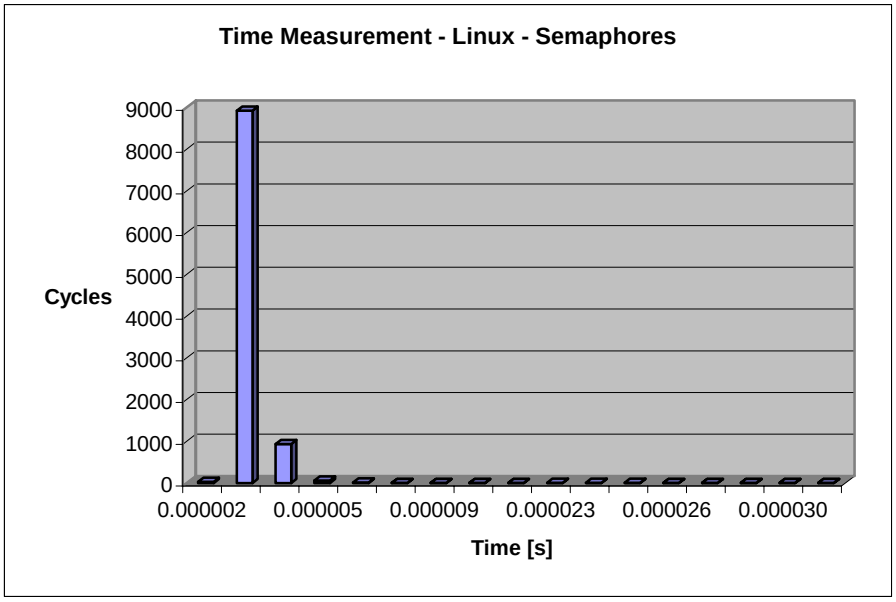

Illustration 59: Linux - Semaphore - 10'000 cycles

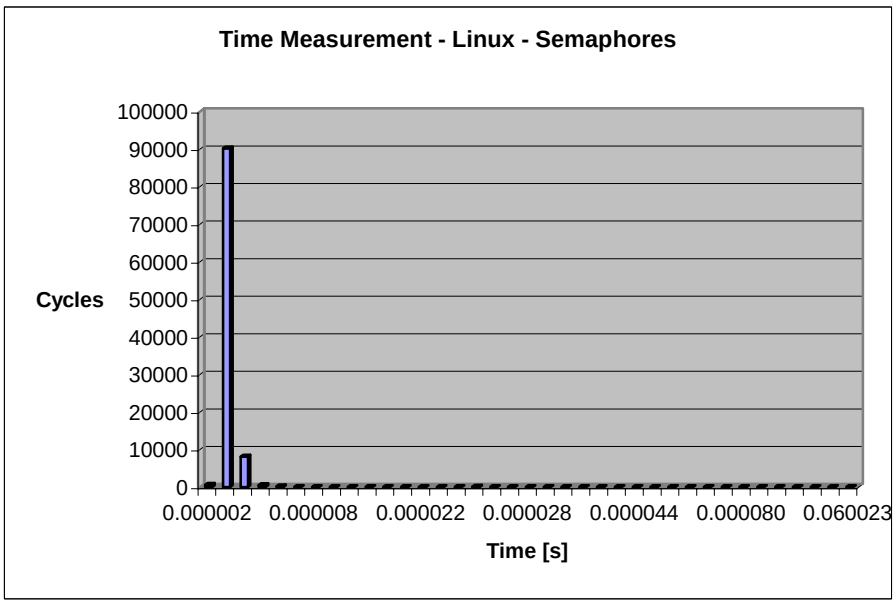

Illustration 60: Linux - Semaphore - 100'000 cycles

Semaphore test on Solaris

Solaris - Semaphore - 100

0.000001	13	0.000005	1	0.000015	1	0.000046	1
0.000002	73	0.000006	3	0.000016	1	0.000109	1
0.000003	4	0.000013	1	0.000024	1	0.000109	1

Solaris - Semaphore - 1'000

0.000001	94	0.000007	1	0.000024	6	0.000047	1
0.000002	636	0.000010	1	0.000025	6	0.000050	1
0.000003	13	0.000014	2	0.000030	1	0.000051	1
0.000004	2	0.000015	33	0.000032	2	0.000079	1
0.000005	104	0.000016	32	0.000033	4	0.000084	1
0.000006	55	0.000017	1	0.000040	1	0.000111	1

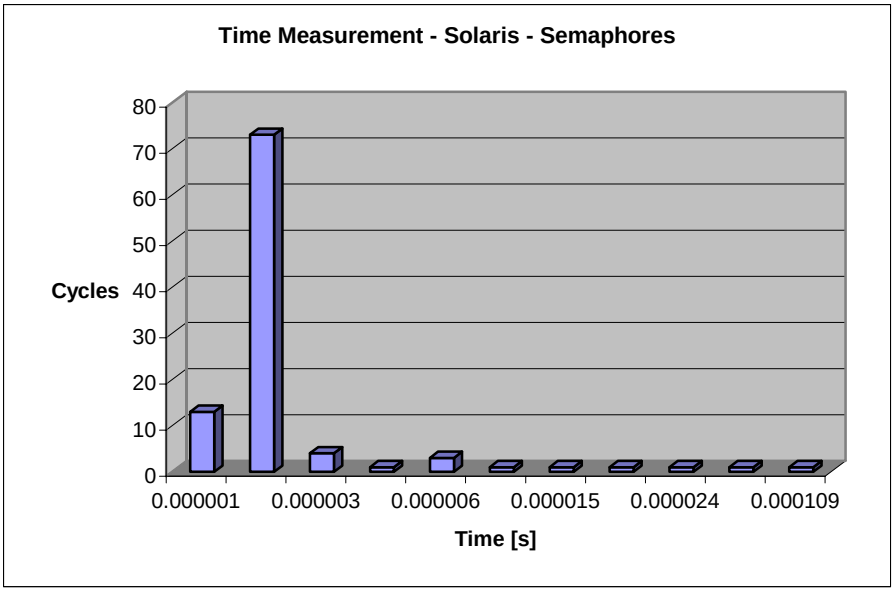

Illustration 61: Solaris - Semaphore – 100 cycles

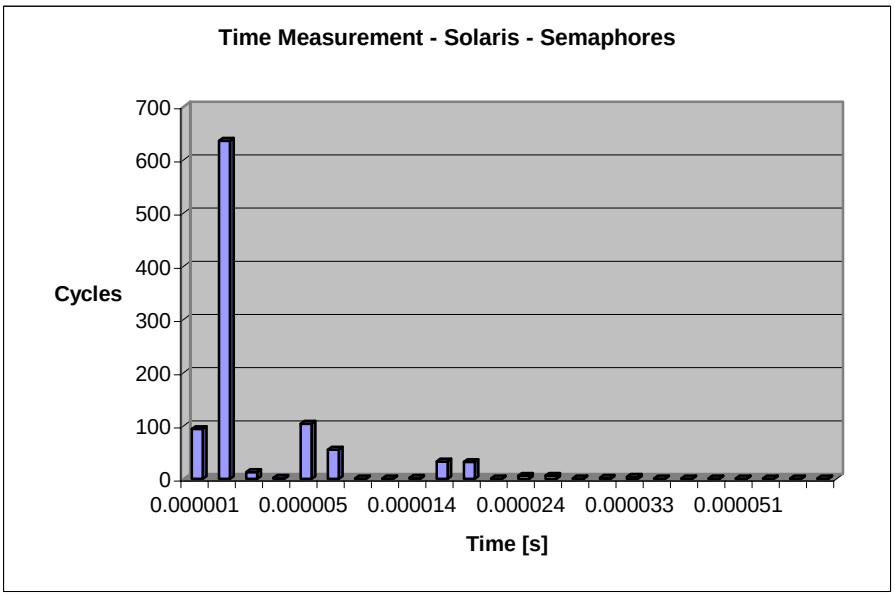

Illustration 62: Solaris - Semaphore – 1'000 cycles

Solaris - Semaphore - 10'000

0.000001	1510	0.000020	1	0.000036	8	0.000076	2
0.000002	7629	0.000021	4	0.000037	6	0.000089	1
0.000003	137	0.000022	15	0.000038	2	0.000090	1
0.000004	11	0.000023	33	0.000040	2	0.000100	1
0.000005	262	0.000024	34	0.000042	1	0.000118	1
0.000006	147	0.000025	5	0.000043	1	0.000123	1
0.000007	20	0.000029	1	0.000049	4	0.000130	1
0.000008	5	0.000030	1	0.000050	3	0.000143	1
0.000010	1	0.000031	1	0.000051	1	0.000155	1
0.000014	20	0.000032	5	0.000059	1	0.000168	1
0.000015	86	0.000033	1	0.000063	3	0.000182	1
0.000016	17	0.000034	3	0.000070	1	0.000386	1
0.000017	1	0.000035	2	0.000075	1	0.007985	1

Solaris - Semaphore - 100'000

0.000001	8220	0.000034	65	0.000070	4	0.000123	1
0.000002	65015	0.000035	20	0.000071	2	0.000124	2
0.000003	2121	0.000036	11	0.000072	2	0.000125	2
0.000004	781	0.000037	2	0.000073	1	0.000128	2
0.000005	11251	0.000038	7	0.000075	12	0.000129	1
0.000006	5038	0.000039	2	0.000076	10	0.000133	2
0.000007	124	0.000040	16	0.000077	3	0.000135	3
0.000008	36	0.000041	75	0.000078	2	0.000136	1
0.000009	34	0.000042	68	0.000079	1	0.000142	1
0.000010	11	0.000043	28	0.000081	1	0.000144	1
0.000011	7	0.000044	12	0.000082	1	0.000146	1
0.000012	17	0.000045	1	0.000083	3	0.000151	1
0.000013	18	0.000046	1	0.000084	5	0.000159	1
0.000014	126	0.000047	4	0.000085	3	0.000161	2
0.000015	2183	0.000048	1	0.000086	5	0.000166	1
0.000016	1892	0.000049	17	0.000087	1	0.000167	1
0.000017	679	0.000050	40	0.000088	1	0.000176	1
0.000018	65	0.000051	28	0.000091	2	0.000181	1
0.000019	12	0.000052	16	0.000092	4	0.000182	1
0.000020	6	0.000053	4	0.000093	4	0.000184	1
0.000021	5	0.000054	1	0.000094	1	0.000192	1
0.000022	10	0.000055	4	0.000097	2	0.000201	1
0.000023	53	0.000056	3	0.000099	1	0.000214	1
0.000024	684	0.000057	4	0.000100	3	0.000243	2
0.000025	493	0.000058	24	0.000101	6	0.000289	1
0.000026	246	0.000059	17	0.000103	2	0.000296	1
0.000027	23	0.000060	12	0.000109	1	0.000301	1
0.000028	7	0.000061	6	0.000110	5	0.000315	1
0.000029	1	0.000064	1	0.000111	1	0.000318	1
0.000030	3	0.000066	2	0.000114	2	0.000339	1
0.000031	5	0.000067	11	0.000117	1	0.000577	1
0.000032	55	0.000068	12	0.000118	3	0.000615	1
0.000033	125	0.000069	3	0.000122	1		

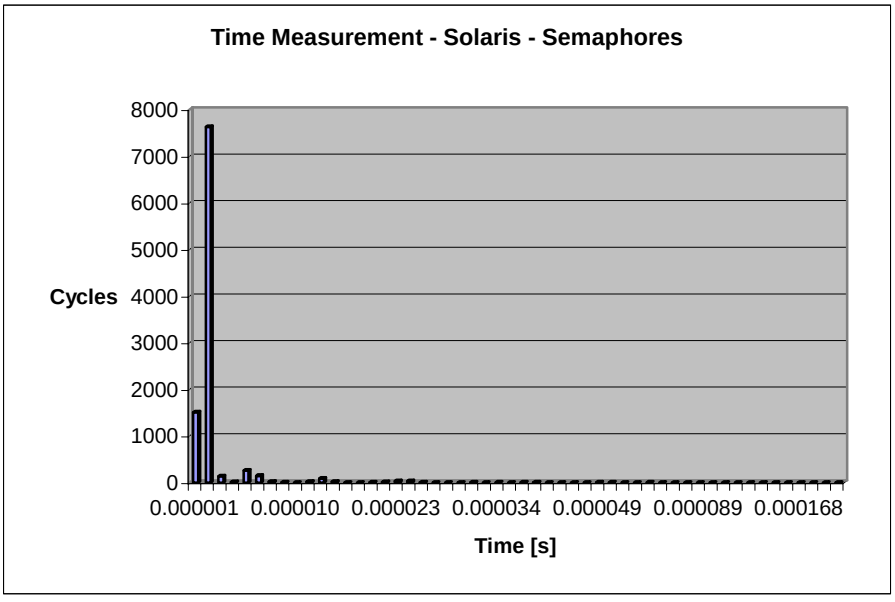

Illustration 63: Solaris - Semaphore - 10'000 cycles

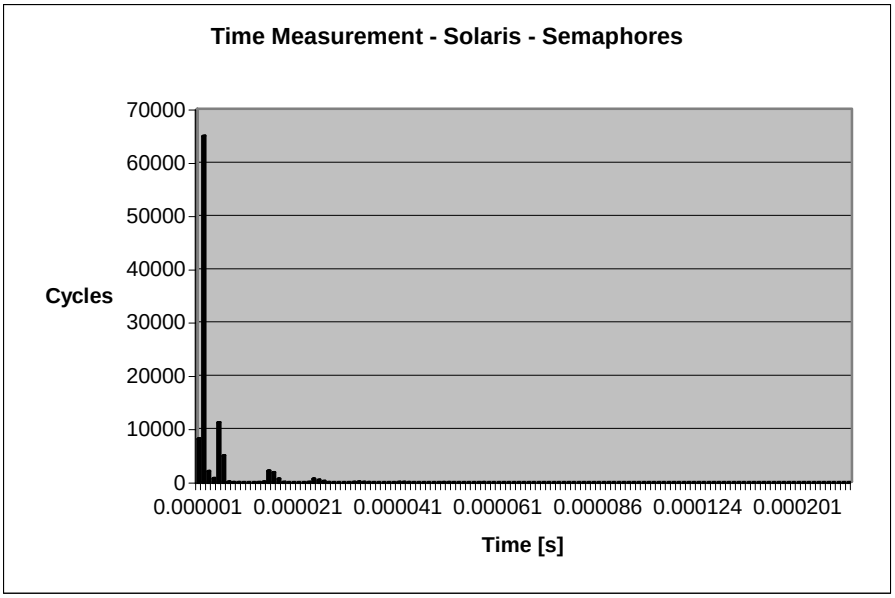

Illustration 64: Solaris - Semaphore - 100'000 cycles

Semaphore test on QNX

QNX - Semaphore - 100		QNX - Semaphore - 1'000	
0.000002	97	0.000002	989
0.000004	1	0.000003	3
0.000005	1	0.000004	2
0.000006	1	0.000005	2
		0.000006	3
		0.000007	1

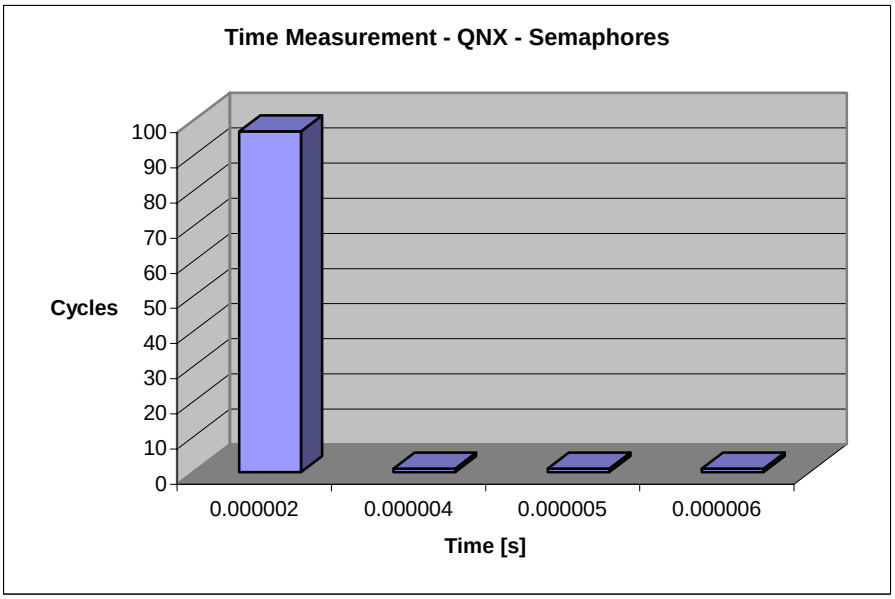

Illustration 65: QNX - Semaphore - 100 cycles

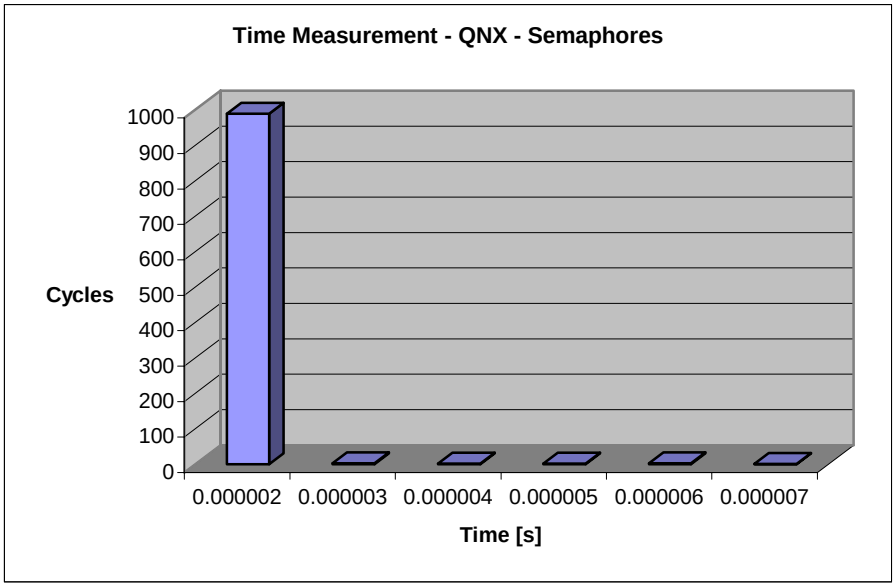

Illustration 66: QNX - Semaphore - 1'000 cycles

QNX - Semaphore - 10'000

0.000002	9928	0.000006	9
0.000003	51	0.000007	1
0.000004	4	0.000008	1
0.000005	5	0.000031	1

QNX - Semaphore - 100'000

0.000002	99053	0.000007	41	0.000018	1	0.000049	1
0.000003	583	0.000008	10	0.000020	1	0.000174	1
0.000004	95	0.000009	1	0.000024	1	0.000234	1
0.000005	72	0.000013	1	0.000025	1		
0.000006	134	0.000016	2	0.000027	1		

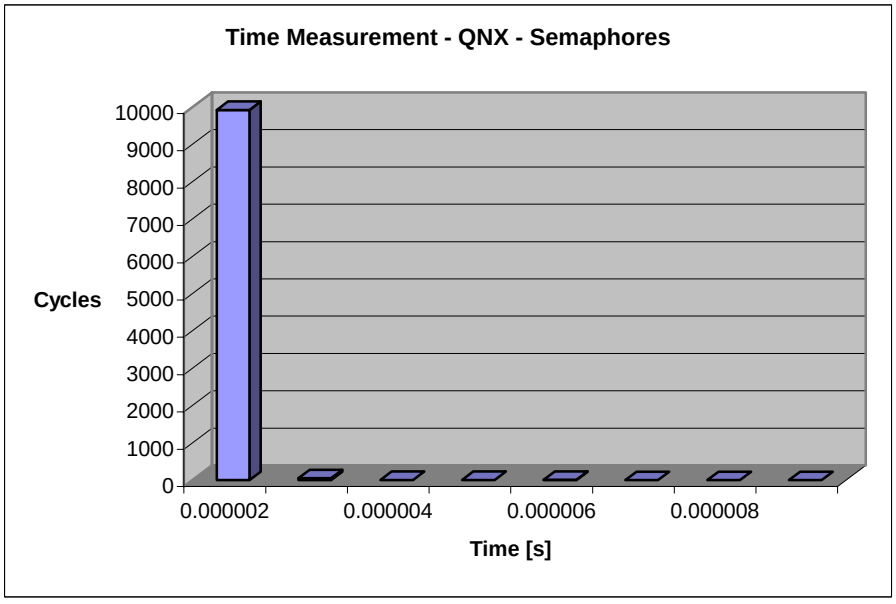

Illustration 67: QNX - Semaphore - 10'000 cycles

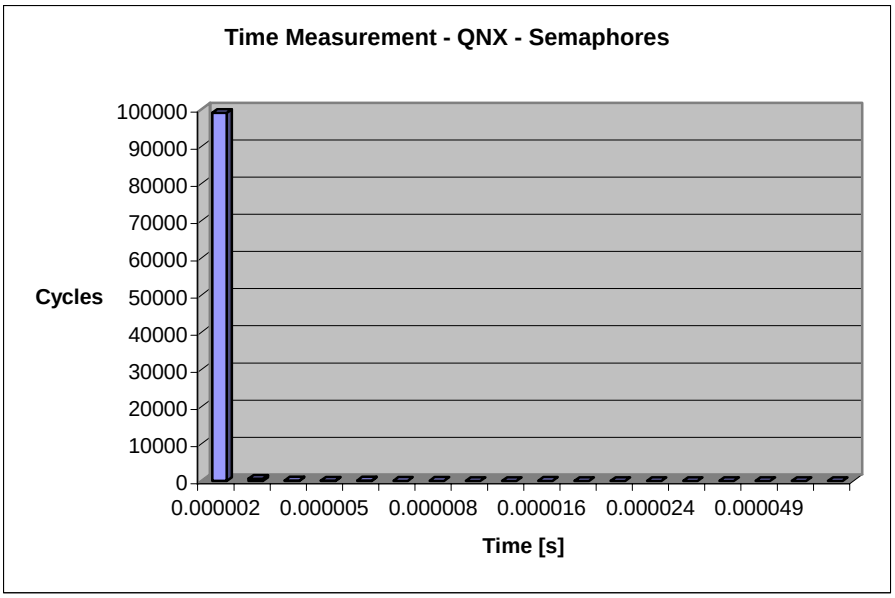

Illustration 68: QNX - Semaphore - 100'000 cycles

Message queue test results

Message queue test on Linux

Linux - Message Queue - 100

0.000012	4	0.000018	1	0.000036	2
0.000013	77	0.000019	1	0.000066	1
0.000014	9	0.000027	1		
0.000015	3	0.000029	1		

Linux - Message Queue - 1'000

0.000012	448	0.000016	4	0.000033	1	0.000083	1
0.000013	413	0.000019	1	0.000034	3	0.000085	1
0.000014	114	0.000025	2	0.000035	3	0.000097	1
0.000015	5	0.000032	2	0.000036	1		

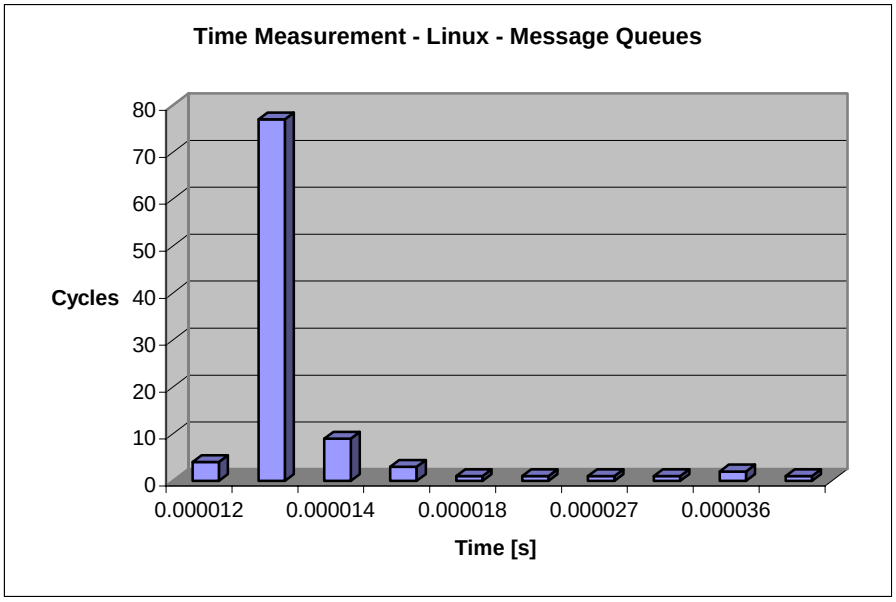

Illustration 69: Linux - Message Queue - 100 cycles

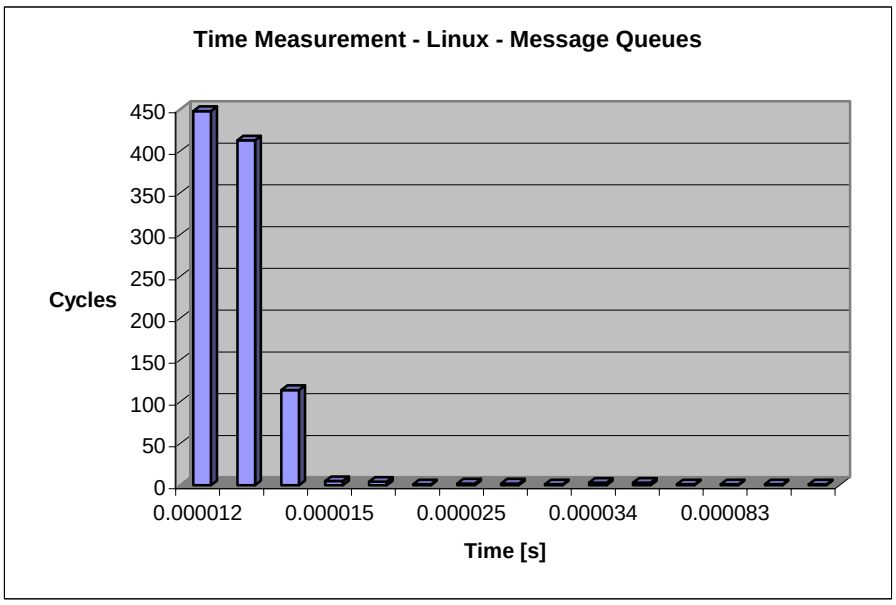

Illustration 70: Linux - Message Queue - 1'000 cycles

Linux - Message Queue - 10'000

0.000012	4746	0.000025	2	0.000038	4	0.000087	3
0.000013	4014	0.000027	1	0.000070	1	0.000088	1
0.000014	996	0.000031	4	0.000077	1	0.000089	1
0.000015	58	0.000032	28	0.000078	2	0.000090	4
0.000016	20	0.000033	16	0.000080	1	0.000091	4
0.000017	1	0.000034	21	0.000083	1	0.000093	1
0.000018	1	0.000035	32	0.000084	1	0.000096	1
0.000019	1	0.000036	15	0.000085	1	0.002897	1
0.000024	1	0.000037	13	0.000086	2		

Linux - Message Queue - 100'000

0.000012	43776	0.000040	7	0.000083	17	0.000114	1
0.000013	43357	0.000041	7	0.000084	7	0.000115	1
0.000014	10323	0.000042	3	0.000085	15	0.000116	1
0.000015	619	0.000043	1	0.000086	10	0.000130	1
0.000016	203	0.000044	1	0.000087	10	0.000136	1
0.000017	16	0.000046	1	0.000088	12	0.000140	1
0.000018	1	0.000047	2	0.000089	10	0.000172	1
0.000019	1	0.000051	1	0.000090	14	0.000179	1
0.000024	2	0.000054	1	0.000091	26	0.000182	1
0.000025	2	0.000055	1	0.000092	22	0.000184	1
0.000026	2	0.000056	1	0.000093	19	0.000189	1
0.000027	1	0.000057	1	0.000094	15	0.000210	1
0.000028	1	0.000058	2	0.000095	8	0.000280	1
0.000029	1	0.000059	2	0.000096	8	0.000452	1
0.000030	1	0.000061	1	0.000097	6	0.000597	1
0.000031	37	0.000066	1	0.000098	3	0.000700	1
0.000032	239	0.000071	1	0.000099	3	0.005012	1
0.000033	228	0.000073	2	0.000100	2	0.005022	1
0.000034	235	0.000076	1	0.000102	2	0.011046	1
0.000035	300	0.000077	4	0.000105	1	0.017468	1
0.000036	192	0.000079	5	0.000106	1	0.018196	1
0.000037	103	0.000080	4	0.000107	1	0.018368	1
0.000038	37	0.000081	6	0.000110	1	0.019156	1
0.000039	14	0.000082	15	0.000113	1		

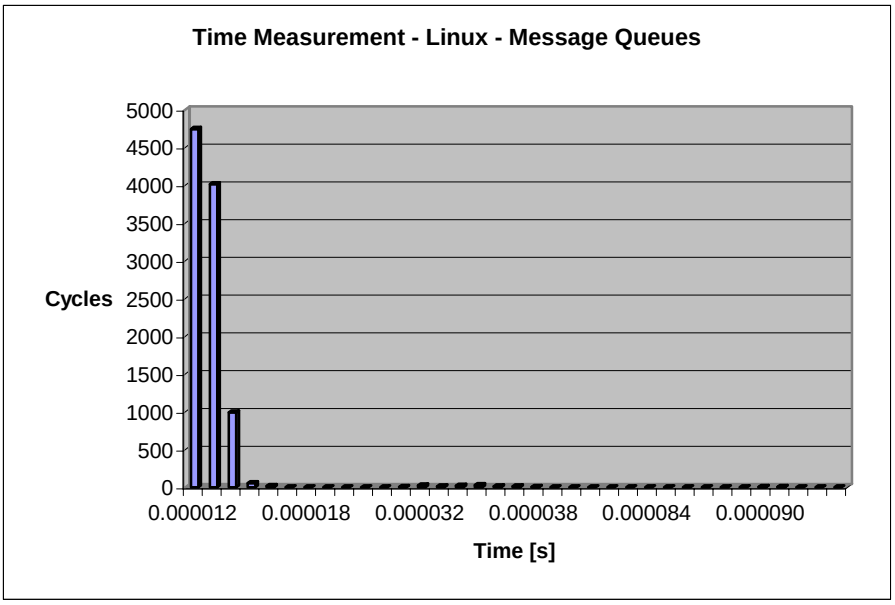

Illustration 71: Linux - Message Queue - 10'000 cycles

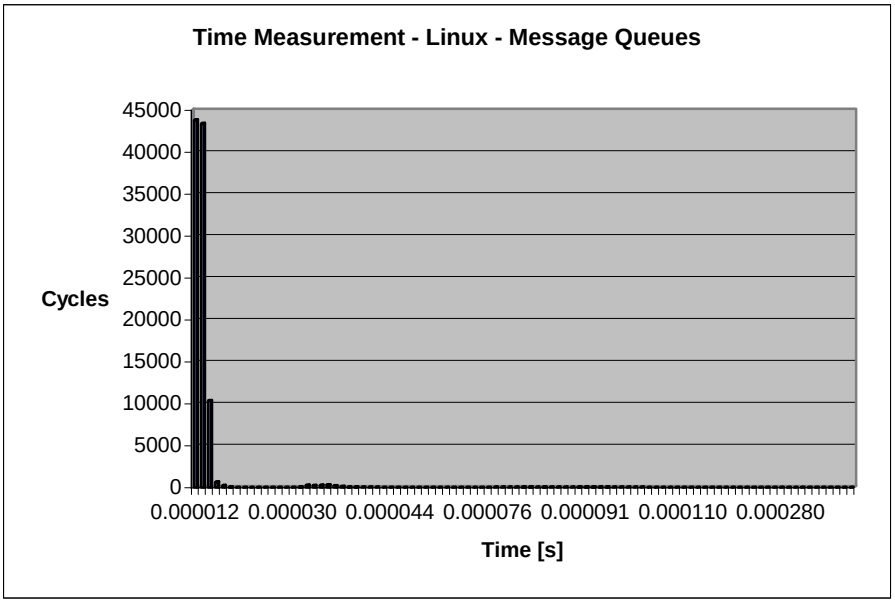

Illustration 72: Linux - Message Queue - 100'000 cycles

Message queue test on Solaris

Solaris - Message Queue - 100

0.000032	4	0.000035	1	0.000040	1	0.000054	1
0.000033	57	0.000036	1	0.000042	1	0.000133	1
0.000034	31	0.000037	1	0.000044	1		

Solaris - Message Queue - 1'000

0.000032	20	0.000047	20	0.000059	1	0.000074	1
0.000033	157	0.000048	1	0.000060	1	0.000076	1
0.000034	58	0.000049	4	0.000061	1	0.000086	1
0.000035	1	0.000051	1	0.000062	2	0.000101	1
0.000036	3	0.000053	3	0.000063	2	0.000129	1
0.000040	4	0.000054	13	0.000064	1	0.000160	1
0.000041	1	0.000055	5	0.000065	1	0.002944	1
0.000042	1	0.000056	5	0.000070	1		
0.000045	41	0.000057	2	0.000071	1		
0.000046	639	0.000058	2	0.000072	1		

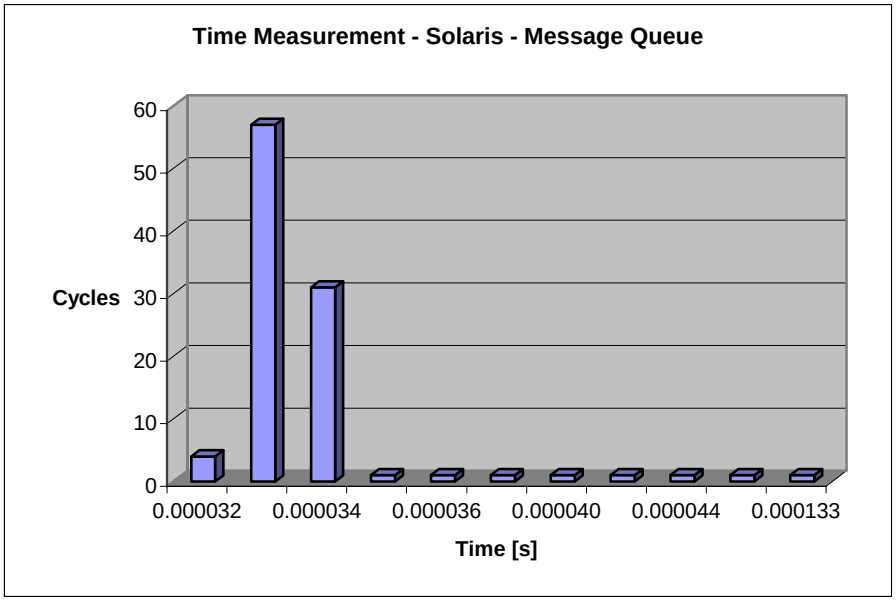

Illustration 73: Solaris - Message Queue - 100 cycles

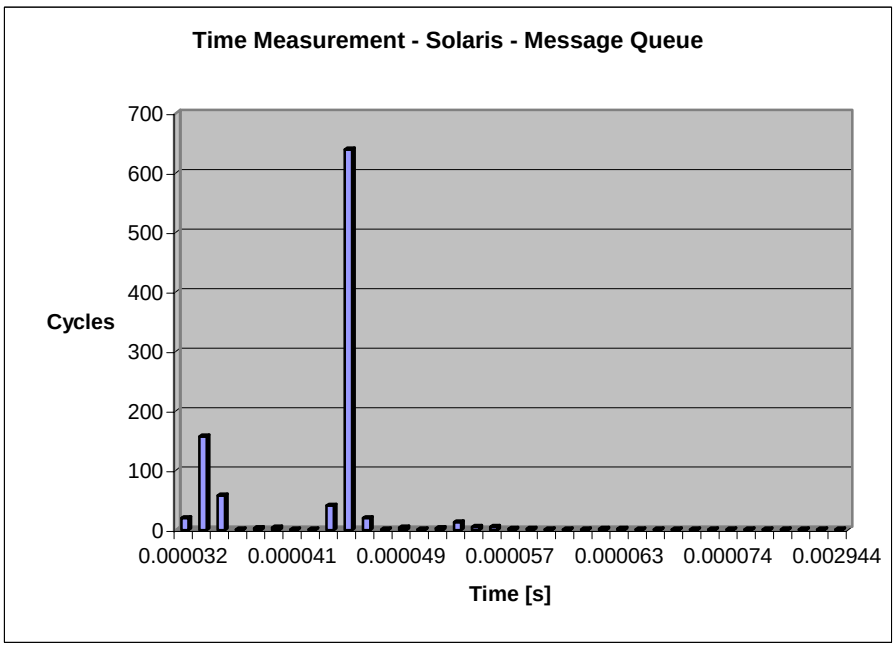

Illustration 74: Solaris - Message Queue - 1'000 cycles

Solaris - Message Queue - 10'000

0.000025	1	0.000042	49	0.000058	4	0.000076	2
0.000026	1	0.000043	25	0.000059	2	0.000077	1
0.000027	1	0.000044	17	0.000060	2	0.000078	2
0.000028	1	0.000045	53	0.000061	2	0.000080	1
0.000029	2	0.000046	1061	0.000062	2	0.000093	1
0.000031	1	0.000047	110	0.000063	1	0.000100	2
0.000032	99	0.000048	31	0.000064	3	0.000101	1
0.000033	3379	0.000049	18	0.000065	7	0.000102	1
0.000034	4385	0.000050	8	0.000066	2	0.000105	1
0.000035	502	0.000051	8	0.000067	1	0.000128	1
0.000036	36	0.000052	8	0.000069	1	0.000164	1
0.000037	29	0.000053	5	0.000070	1	0.000165	1
0.000038	19	0.000054	7	0.000072	1	0.000166	1
0.000039	11	0.000055	3	0.000073	1	0.000624	1
0.000040	19	0.000056	4	0.000074	1	0.002821	1
0.000041	52	0.000057	5	0.000075	1		

Solaris - Message Queue - 100'000

0.000024	2	0.000041	578	0.000057	9	0.000075	1
0.000025	3	0.000042	262	0.000058	13	0.000076	1
0.000026	7	0.000043	109	0.000059	10	0.000077	1
0.000027	8	0.000044	86	0.000060	3	0.000079	1
0.000028	2	0.000045	86	0.000061	10	0.000080	1
0.000029	3	0.000046	82	0.000062	3	0.000081	1
0.000031	6	0.000047	75	0.000063	8	0.000083	2
0.000032	2769	0.000048	29	0.000064	4	0.000086	2
0.000033	49619	0.000049	34	0.000065	4	0.000103	1
0.000034	41987	0.000050	35	0.000067	3	0.000148	1
0.000035	2634	0.000051	25	0.000068	1	0.000605	1
0.000036	320	0.000052	22	0.000069	2	0.000616	1
0.000037	270	0.000053	17	0.000070	1	0.000622	1
0.000038	135	0.000054	12	0.000071	2	0.000624	1
0.000039	192	0.000055	17	0.000073	2	0.000625	2
0.000040	463	0.000056	15	0.000074	2	0.000629	1

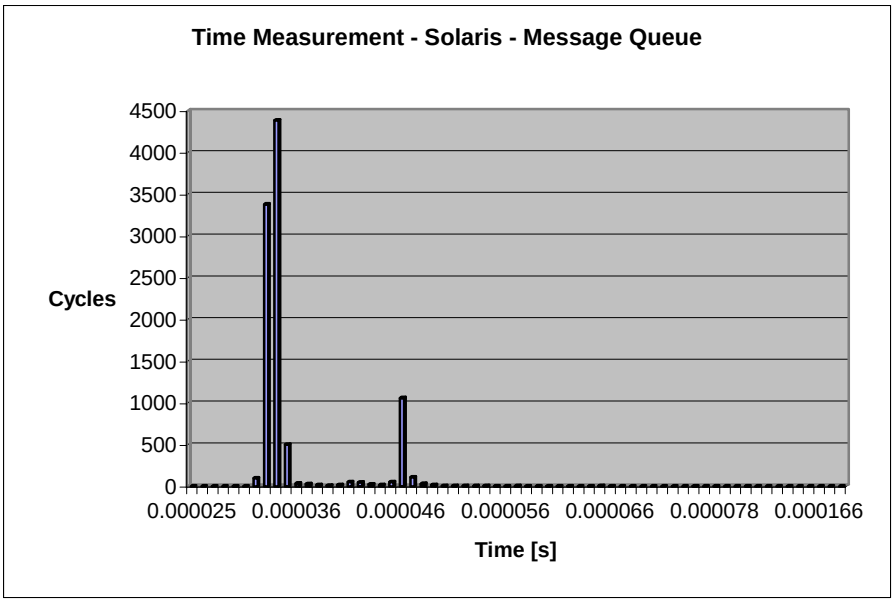

Illustration 75: Solaris - Message Queue - 10'000 cycles

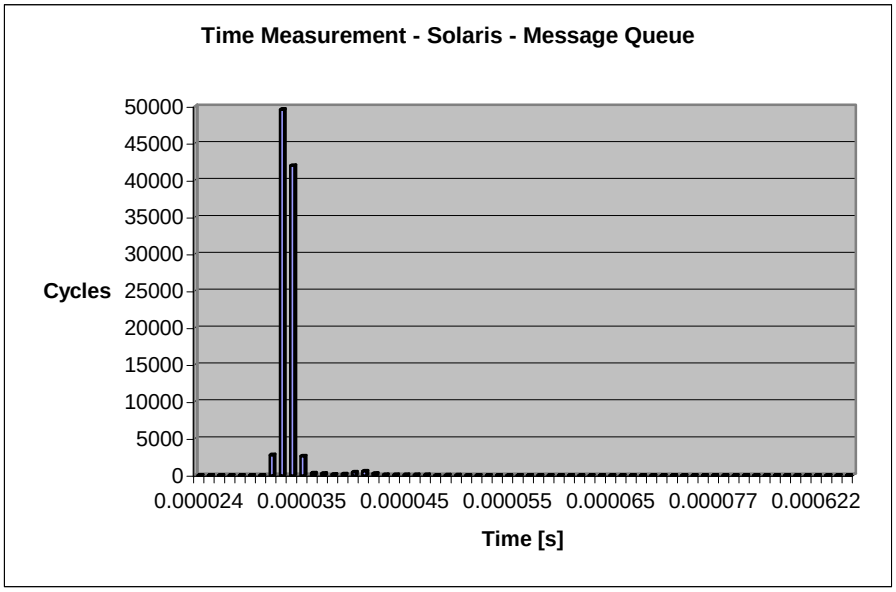

Illustration 76: Solaris - Message Queue - 100'000 cycles

Message queue test on QNX

QNX - Message Queue - 100

0.000013	92	0.000039	1				
0.000014	1	0.000043	1				
0.000015	2	0.000134	1				
0.000018	1	0.072905	1				

QNX - Message Queue - 1'000

0.000013	942	0.000027	1	0.000051	1	0.000243	1
0.000014	14	0.000028	1	0.000052	1	0.000296	1
0.000015	18	0.000029	2	0.000072	1	0.000988	1
0.000016	7	0.000035	1	0.000129	1	0.001773	1
0.000017	1	0.000037	1	0.000154	1		
0.000020	1	0.000044	1	0.000157	1		

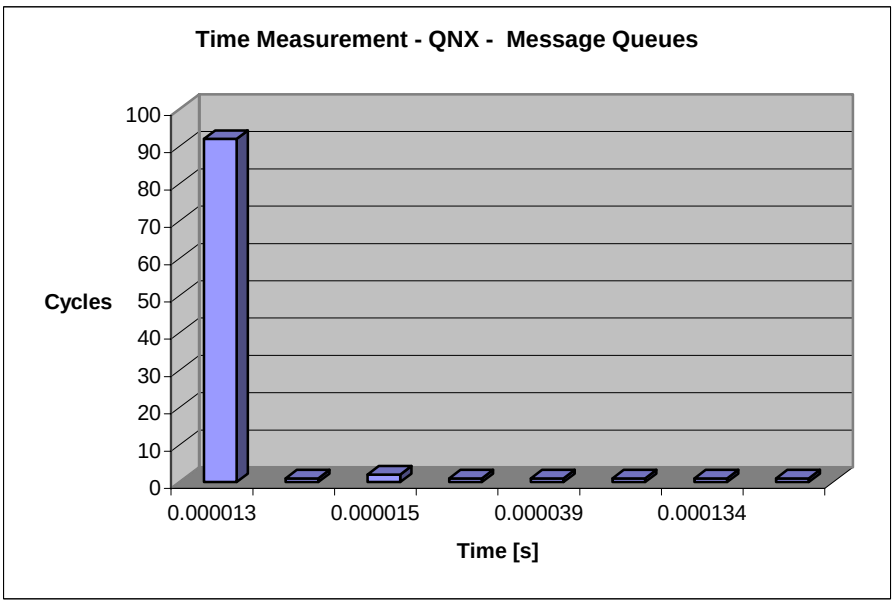

Illustration 77: QNX - Message Queue - 100 cycles

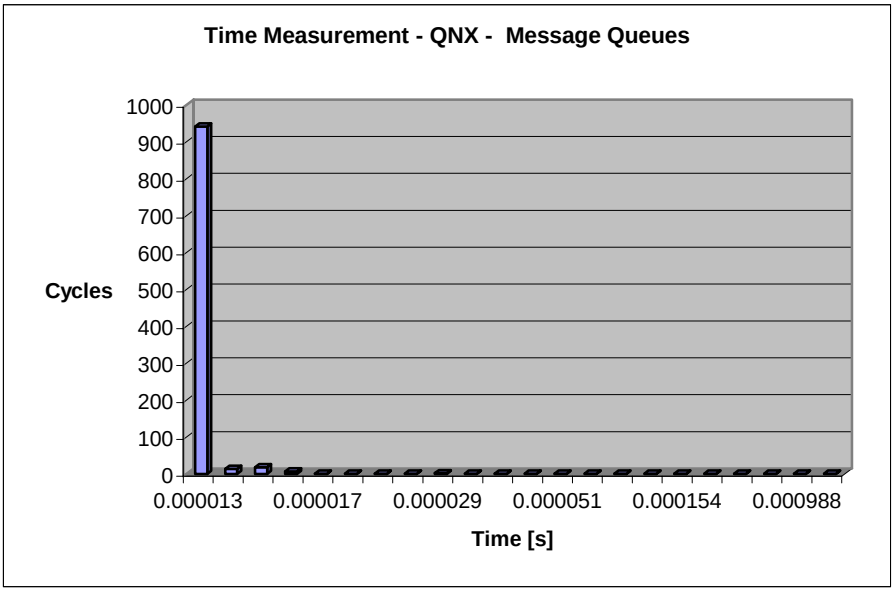

Illustration 78: QNX - Message Queue – 1'000 cycles

QNX - Message Queue - 10'000

0.000013	9519	0.000025	2	0.000041	1	0.000083	1
0.000014	155	0.000026	1	0.000043	1	0.000092	1
0.000015	220	0.000027	1	0.000044	2	0.000123	1
0.000016	45	0.000028	3	0.000047	1	0.000134	1
0.000017	11	0.000029	3	0.000049	1	0.000156	1
0.000018	2	0.000032	1	0.000050	1	0.000227	1
0.000019	1	0.000033	1	0.000052	1	0.000262	1
0.000020	1	0.000034	1	0.000054	1	0.000278	1
0.000021	1	0.000035	3	0.000073	1	0.000413	1
0.000023	4	0.000037	1	0.000076	1	0.001090	1
0.000024	1	0.000040	1	0.000081	1		

QNX - Message Queue - 100'000

0.000013	86462	0.000062	2	0.000120	1	0.000280	1
0.000014	9958	0.000063	3	0.000121	2	0.000281	2
0.000015	2007	0.000064	7	0.000122	2	0.000284	1
0.000016	824	0.000065	4	0.000123	2	0.000294	1
0.000017	90	0.000066	4	0.000124	1	0.000303	1
0.000018	56	0.000067	2	0.000127	2	0.000304	1
0.000019	11	0.000068	3	0.000129	1	0.000312	1
0.000020	7	0.000069	2	0.000131	2	0.000334	1
0.000021	6	0.000070	3	0.000138	1	0.000348	1
0.000022	13	0.000072	8	0.000139	1	0.000356	1
0.000023	21	0.000073	5	0.000141	1	0.000359	1
0.000024	22	0.000074	4	0.000142	1	0.000363	1
0.000025	26	0.000075	3	0.000145	1	0.000368	1
0.000026	16	0.000076	2	0.000147	1	0.000370	1
0.000027	24	0.000077	2	0.000148	1	0.000382	1
0.000028	24	0.000078	1	0.000150	1	0.000385	1
0.000029	21	0.000079	1	0.000152	1	0.000389	1
0.000030	5	0.000080	2	0.000153	1	0.000394	1
0.000031	7	0.000081	2	0.000155	1	0.000406	1
0.000032	8	0.000082	1	0.000156	1	0.000412	1
0.000033	3	0.000084	4	0.000157	1	0.000428	1
0.000034	9	0.000085	1	0.000158	1	0.000490	1
0.000035	13	0.000086	3	0.000159	1	0.000495	1
0.000036	4	0.000088	2	0.000162	1	0.000496	1
0.000037	7	0.000089	1	0.000167	1	0.000573	1
0.000038	12	0.000090	3	0.000169	1	0.000602	1
0.000039	9	0.000091	3	0.000174	1	0.000770	1
0.000040	4	0.000092	4	0.000177	1	0.000993	1
0.000041	5	0.000093	1	0.000179	1	0.001015	1
0.000042	10	0.000094	1	0.000183	1	0.001121	1
0.000043	7	0.000095	1	0.000184	1	0.001132	1
0.000044	14	0.000096	2	0.000186	1	0.001198	1
0.000045	14	0.000097	2	0.000189	1	0.001366	1
0.000046	8	0.000098	1	0.000190	1	0.001370	1
0.000047	11	0.000100	1	0.000191	1	0.001465	1
0.000048	5	0.000101	1	0.000192	1	0.001561	1
0.000049	11	0.000104	2	0.000208	1	0.001873	1
0.000050	1	0.000106	2	0.000211	1	0.001876	1
0.000051	7	0.000107	1	0.000212	1	0.003338	1
0.000052	7	0.000108	1	0.000220	1	0.006891	1
0.000053	4	0.000109	3	0.000221	1	0.006917	1
0.000054	3	0.000110	1	0.000222	1	0.006996	1
0.000055	2	0.000111	1	0.000224	1	0.007075	1
0.000056	1	0.000112	1	0.000233	1	0.007093	1
0.000057	2	0.000113	2	0.000236	1	0.007256	1
0.000058	1	0.000114	1	0.000237	1	0.011107	1
0.000059	6	0.000115	1	0.000241	1	0.017617	1
0.000060	2	0.000116	1	0.000256	1		

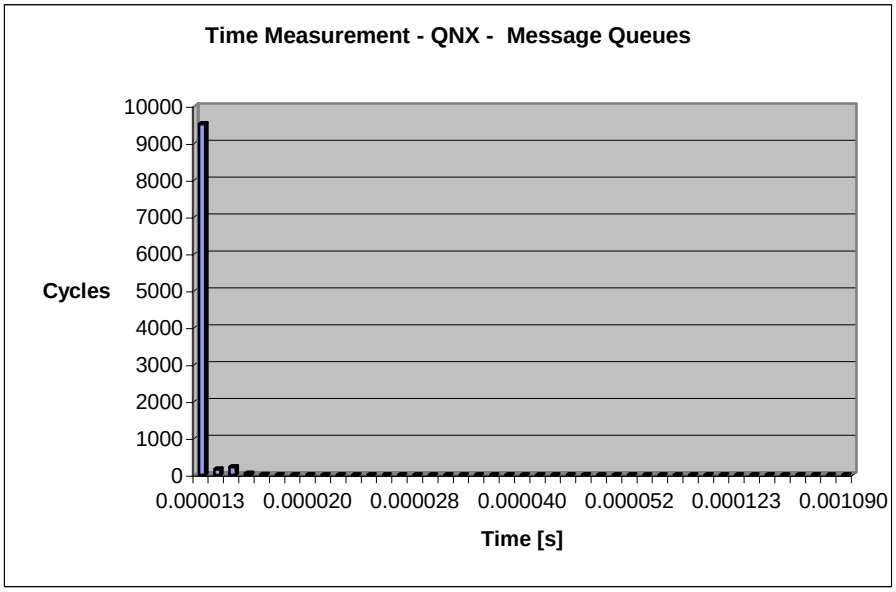

Illustration 79: QNX - Message Queue - 10'000 cycles

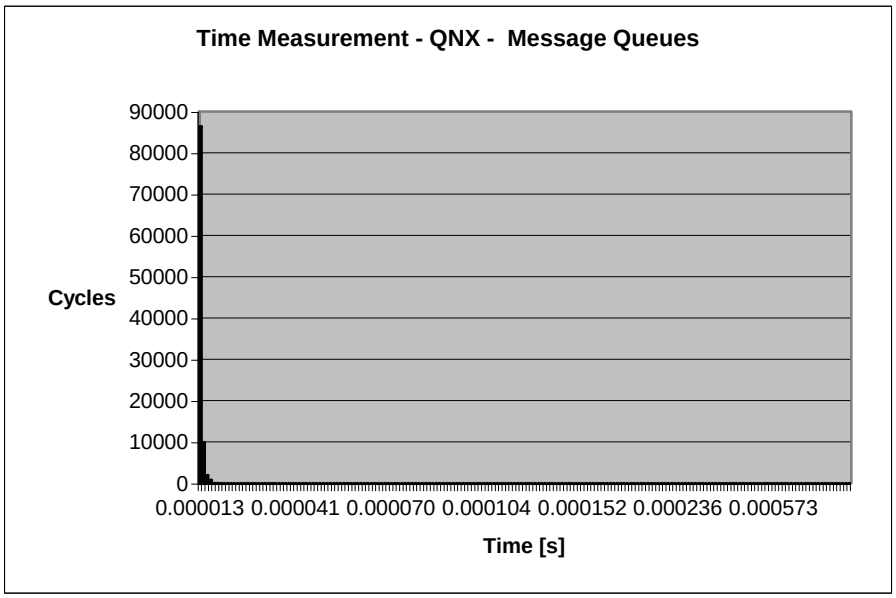

Illustration 80: QNX - Message Queue - 100'000 cycles

Installing network card under Solaris 10

Under Solaris, the command **ifconfig -a** can be used to show the interface configuration of the system. Before the network card has been installed, this command returns us the following configuration:

```
bash-3.00# ifconfig -a
lo0: flags=2001000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4,VIRTUAL> mtu 8232
index 1
        inet 127.0.0.1 netmask ff000000
```

The configuration shows, that there is no network card installed. The entries shown above represent the local loop back.

To install the network card, a driver package needs to be installed. Therefore, the files of the driver package need to be extracted into a new folder, enter this folder and finally install the package. But before we can install this driver package, the Solaris system has to be configured for running in 32-bit mode. This can be achieved by typing the following command lines into the shell:

```
# /usr/sbin/eeprom boot-file = “kernel/unix”
```

After rebooting the system, the running mode can be checked by using typing:

```
# /usr/bin/isainfo –kv
```

If the system running mode has been changed successfully, this command displays the following:

```
32-bit i386 kernel modules
```

Now the driver package can be installed:

```
bash-3.00# pkgadd -d /source folder/sk9521
The following packages are available:
  1  SKGEsol     SysKonnect Gigabit Ethernet Adapter families 32 bit driver
              (i386) 8.12.1.3

Select package(s) you wish to process (or 'all' to process
all packages). (default: all) [?,??,q]: all

Processing package instance <SKGEsol> from </source folder/sk9521>

SysKonnect Gigabit Ethernet Adapter families 32 bit driver(i386) 8.12.1.3
SysKonnect GmbH

----------------------
   IP configuration
----------------------

Do you want to configure the IP interfaces now (y/n)? n

Do you have more SysKonnect Gigabit Ethernet adapters installed (y/n)? n
Using </> as the package base directory.
## Processing package information.
## Processing system information.
   9 package pathnames are already properly installed.
## Verifying disk space requirements.
## Checking for conflicts with packages already installed.
## Checking for setuid/setgid programs.

This package contains scripts, which will be executed with super-user
permission during the process of
installing this package.

Do you want to continue with the installation of <SKGEsol> [y,n,?] y

Installing SysKonnect Gigabit Ethernet Adapter families 32 bit driver
as <SKGEsol>

## Executing preinstall script.
## Installing part 1 of 1.
/etc/rcS.d/S50skge
/kernel/drv/skge
/kernel/drv/skge.conf
/usr/sbin/skge_vlan_config
/usr/share/man/man7d/skge.7d
```

```
  [ verifying class <none> ]
  [ verifying class <master> ]
  ## Executing postinstall script.
  add_drv skge
  starting network interfaces ...
  ifconfig: setifflags: SIOCSLIFFLAGS: skge0: Cannot assign requested address
  skge0 not started
  .

  Installation of <SKGEsol> was successful.

  *** IMPORTANT NOTICE ***
          This machine must now be rebooted in order to ensure
          sane operation.  Execute
                 shutdown -y -i6 -g0
          and wait for the "Console Login:" prompt.
```

After the installation of the network card was successful, the interface configuration has changed now:

```
  bash-3.00# ifconfig -a
  lo0:  flags=2001000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4,VIRTUAL>  mtu  8232
index 1
          inet 127.0.0.1 netmask ff000000
  skge0:  flags=1000862<BROADCAST,NOTRAILERS,RUNNING,MULTICAST,IPv4>  mtu  1500
index 3
          inet 0.0.0.0 netmask ff000000 broadcast 127.255.255.255
          ether 0:0:5a:9e:64:e9
```

The newly installed network card is now shown in the list. But the IP-address and netmask are not set yet. To configure the network card, a couple of files have to be changed, which are all placed under /etc/. Add the name of the network card to the file hostname.skge0 (in this case skge0), add the entry

44.63.25.10 felagund

to the file hosts to configure your preferred IP-address and hostname (felagund) and finally add your preferred netmask to the file netmasks:

44.63.25.10 255.255.0.0

After rebooting your system, the command ifconfig –a should display the right information:

```
  # ifconfig -a
  lo0: flags=2001000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4,VIRTUAL> mtu 8232
          index 1
          inet 127.0.0.1 netmask ff000000
  skge1: flags=1000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4> mtu 1500 index 2
          inet 44.63.25.10 netmask ffff0000 broadcast 44.255.255.255
          ether 0:0:5a:9e:64:e9
```

The network card can as well be configured manually, but in this case, the information will be lost as soon as your system is being rebooted:

```
  # ifconfig skge0 44.63.25.10 netmask 255.255.0.0 up
```

To display the actual routing table, type:

```
  # netstat –nr
```

To reach computers situated outside of the local subnet, a new entry has to be added to the routing table. Therefore, create a new file named defaultrouter add your default router address and save it under /etc/. After successfully adding this entry, netstat –nr will display:

```
# netstat -nr
Routing Table: IPv4
  Destination           Gateway           Flags  Ref   Use   Interface
-------------------- -------------------- ----- ----- ------ ---------
44.0.0.0             44.63.25.10          U         1      0  skge1
default              44.63.25.1           UG        1      0
127.0.0.1            127.0.0.1            UH       23   6998  lo0
```

Content of the supported CD-ROM

On the supported CD-ROM can be found the following files:

- Documentation about Solaris and QNX
- Drivers
- 3rd party software and development tools
- Source Code in Word-format
- Source Code (message queue, semaphore,…)
- Latest report version
- Final Diploma Work presentation