Postgraduate thesis
Information and Communication Technology

Simulated testing in Mobile Datacom Systems (GPRS / UMTS)

Aida Arslanagic & Elisabeth S Siqveland

Grimstad, Norway
Spring 2000

Abstract

The assignment described in this thesis is given by Ericsson AS and gives an evaluation of simulation and testing tools used in their development projects.

Since Ericsson wants to change their testing process the task was to evaluate and suggest a solution to how this could be done. In order to make testing available for all developers it was required to run tests independent of hardware. Thus they wanted to simulate the hardware resources.

Originally, the intention was to use the test environment used otherwise in Ericsson. This proved to cause many problems because the project became dependent of other developers work, and the instability of the test environment caused delays. In order to avoid these problems the assignment had to be redefined. This was done in collaboration with the supervisor; and it was decided to develop a new testing environment.

Several tools had to be used in order to develop an appropriate testing environment. This meant acquiring knowledge about the tools used otherwise for software development at Ericsson. This included among others, protocol development tool, STREAMS, client – server programming using C-language, and programming real-time applications in the Tornado environment (VxWorks and VxSim).

The purpose for the assignment was to compare a real-time operating system, VxWorks, and a software package, VxSim, which simulates VxWorks on a UNIX Workstation.

The test results show that VxSim is the fastest alternative of the two. The tests also reveal that VxSim is unstable and unreliable. On the other hand VxWorks is very reliable and stable. The test results vary depending on the processor type used for testing.

VxSim is well suited for testing in the early stages of the project, thus it does not support all the features available in VxWorks.

VxWorks and VxSim may very well be used jointly in the development and testing processes.

Preface

This master engineering thesis is part of the "IKT Sivilingeniør" education at Agder University College in Grimstad. This thesis deals with problems regarding simulated testing in mobile data communication systems.

The assignment is given by Ericsson AS and gives an evaluation of the aspects of their testing tools. It also provides an evaluation of how a simulation tool could function in their environment.

This researched work is based on co-operation with Karsten Vileid who provided us with information and knowledge about the theoretical part of the assignment. We would like to take the opportunity to thank Karsten for his time and interchanging ideas and solutions.

We would also like to thank Bjørn Corneliussen and Roar Walderhaug for their time and help during the project.

And finally we would like to thank our supervisors at HiA, Øyvind Hanssen and Jan P Nytun for helping us with the administrative part of the thesis.

Grimstad
Spring 2000

Aida Arslanagic   Elisabeth S Siqveland

Table of Contents

List of figures, tables, and graphs

Acronyms

AP	Application Processor
API	Application Programming Interface
ATM	Asynchronous Transfer Mode
BSS	Basestation SubSystem
CPU	Central processing Unit
DP	Device Processor
E1	PowerPC 860
FIFO	First – In – First – Out
GGSN	Gateway GSN
GPRS	General Packet Radio Service
GSN	GPRS Support Node
HAM	PowerPC 603
IEEE	The Institute of Electrical and Electronics Engineers
I/O	Input / Output
IP	Internet Protocol
k	1024 byte
LCP	Link Control Protocol
MMU	Memory Management Unit
MS	Mobile Station
NCP	Network Control Protocol
OS	Operating System
POSIX	Portable Operating System Interface
PPC	PowerPC 604
PPP	Point – to – Point Protocol
RISC	Reduced Instruction Set Computer
RPC	Remote Procedure Call
RTOS	Real – Time Operating System
SGSN	Serving GSN
SLIP	Serial Line Interface Protocol
SMS	Short Message Service
TCB	Task Control Block
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
ULIP	User – Level Internet Protocol

Chapter 1 Introduction

This thesis deals with problems regarding simulated testing in mobile data communication systems.

The assignment is given by Ericsson AS and gives an evaluation of the aspects of their testing tools. It also provides an evaluation of how a simulation tool could function in their environment.

Background

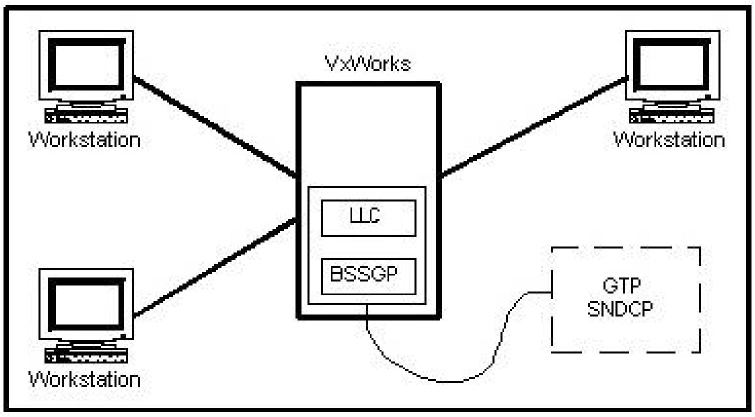


Figure 1 Ericsson's testing environment.
The Workstations are UNIX SPARCs running Solaris operating system. The test nrhric.sim executes on the Workstations while the simulated signaling takes place on the VxWorks target. The target is called DP, which also handles the payload traffic, i.e. signaling between GSNs.

The signalling in Ericsson's Serving GPRS Support Nodes (SGSN) and Gateway GPRS Support Nodes (GGSN) is handled by an Application ProCessor (AP), which runs on a Sun Workstation, using SunOS. Testing is complex and difficult in this environment. Thus Ericsson developed a simulated AP, called nrhric.sim, to make testing of payload less complex.

Ericsson has also developed a Device Processor (DP), which handles payload traffic. Their DPs run on a PowerPC CPU's, using VxWorks real-time operating system.

When testing payload they use nrhric.sim on one Workstation to handle signalling. Afterwards the Workstation is connected to the appropriate DPs.

It is not possible for many people to run the test at the same time because testing takes place on PowerPC CPUs, which is a limited resource. The problem can be avoided if UNIX Workstations can be used to simulate the DPs. This means that the testing of payload will be independent of PowerPC CPUs and thus can be run on all UNIX Workstations.

VxSim is a software packet, which can be used to simulate VxWorks on a UNIX Workstation.

Assignment

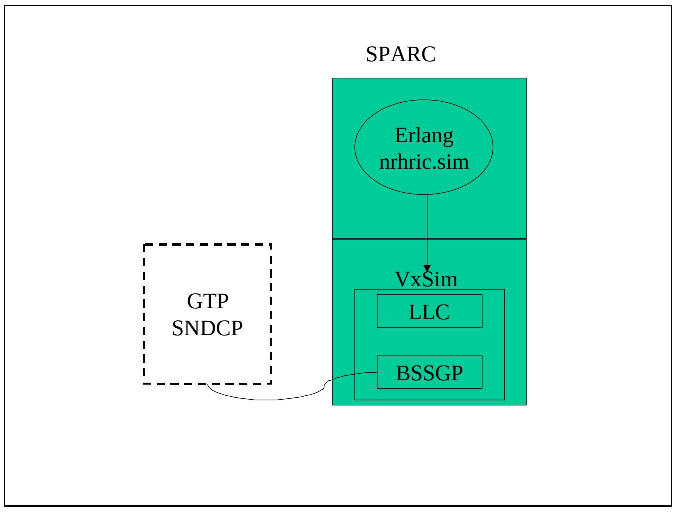


Figure 2 Simulated VxWorks environment, using VxSim.
Both nrhric.sim and the signaling take place on the UNIX SPARC Workstation, i.e. there is no need for the DP.

Ericsson wants to change the existing testing process to make it possible to test from all UNIX Workstations. Our task is to evaluate and suggest a solution to how this can be handled in their environment.

Redefinition of Assignment

Originally our task was to simulate the existing environment using nrhric.sim. Along the line this proved to cause many problems. We became dependent of all the people working on this project, both in Grimstad and Gøteborg. Their environment also proved to be unstable for us to use.

To avoid these problems our supervisor and we decided to redefine the testingpart of the assignment. We decided to develop our own testing environment, which builds on the principles of the existing environment.

Structure

This thesis consists of seven chapters, each discussing an aspect of the assignment.

Chapter 1, “Introduction”, presents an overview of the assignment.

Chapter 2, “Research and Study”, describes the line of action and the tools used to solve the given assignment in detail.

Chapter 3, “Problem solving”, describes the existing testing environment. It also introduces our solution to the testing environment problem. In addition it provides a description of how to compile the existing code for VxSim.

Chapter 4, “Implementation and Results”, describes the implementation of the new testing environment and the assumptions based on theory. It also gives a review of the test criteria. And finally it presents the results measured during testing, for both VxSim and VxWorks.

Chapter 5, “Colloquy of Researched Work”, gives an overview of the benefits and drawbacks with both VxSim and VxWorks. It offers a discussion of the achieved test results. And introduces data transmission protocols available for VxSim.

Chapter 6, “Discussion of VxWorks and VxSim”, compares the features in VxWorks and VxSim.

Chapter 7, “Further Study”, gives an overview of aspects regarding further development of this project.

Chapter 8, “Conclusion”, gives a conclusion based on the results discussed in the earlier chapters.

Chapter 9, “References”

This thesis also includes two appendixes.

Appendix A, Code, for both STREAMS and UDP environment.

Appendix B, Testing Results

Chapter 2 Research and Study

In order to complete the assignment we had to learn UNIX, STREAMS, C, VxSim and VxWorks. We also had to learn about GPRS and the new nodes in the network, in order to test software developed for SGSN and GGSN.

Thus the first step was to improve our UNIX skills. Therefore we attended a UNIX user course, where we learnt the basic commands and functions supported by UNIX.

Afterwards we started to study other subjects of the assignment in order to obtain necessary competence. We used several methods to accomplish this. For VxWorks and VxSim study we used course material used otherwise for VxWorks training. In addition we used the Internet for information search about subjects of learning.

STREAMS and C were learnt during development of the testing environment.

We were also invited to attend a seminar regarding Ericsson's GPRS network, including SGSN and GGSN.

Development Tools

Real time operating system, VxWorks

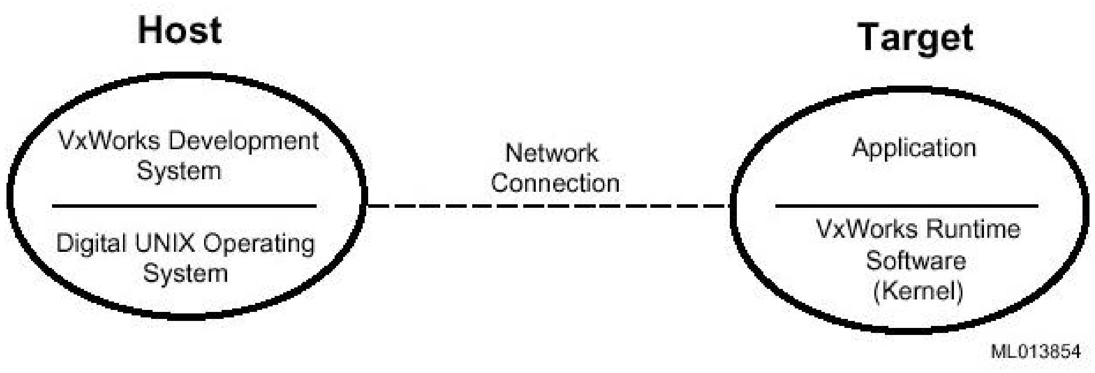

Host
Target
VxWorks Development System
Digital UNIX Operating System
Network Connection
Application
VxWorks Runtime Software (Kernel)
ML013854

Figure 3 VxWorks Host Target Configuration

VxWorks is a commercial operating system for embedded real-time applications. It requires a host workstation, running UNIX or Windows, for program development. All program development is done on a host machine where the target software is cross-compiled to load and run on various target CPU architectures. The VxWorks systems can also be used as real-time servers in a networked environment

VxWorks is flexible, scalable, reliable and available on all popular CPU platforms. The VxWorks RTOS comprises the core capabilities of the micro kernel along with advanced networking support, powerful file system and I/O management, and C++ and other standard run-time support. These core capabilities can be combined with add-on components.

VxWorks is not part of the bundled operating system and it will not be seen unless it is being used. It supports concurrent tasking, is multi-threaded and supplies all intertask messaging functions[1]. VxWorks is "UNIX-friendly", meaning that it interoperates nicely with the UNIX environment and provides many of the UNIX C-library routines[2]. It also supports disk- and network-based file systems together with remote debugging from workstations.

The development software in VxWorks consists of software that runs on host computer, and run-time software, that runs on target computer, as shown in Figure 3. The VxWorks shell, which is a part of the run-time software, accepts C language expressions. It runs on the target computer. The flexible VxWorks network software permits multiple target systems to be connected to a single backplane or connected to an Ethernet link. UNIX systems and VxWorks systems can be used together in a hybrid application. VxWorks´ open design is highly portable and complete across all supported processors, allowing application migration between architectures with minimal effort.

The VxWorks programming interface is tightly integrated with the VxWorks kernel. The VxWorks **kernel** is that portion of the system software that schedules task execution and coordinates the use of system resources. The routines in this interface are arranged in libraries, according to the kind of work they perform.

A network that includes VxWorks nodes can be connected with other networks in a complex configuration. VxWorks was the first RTOS to integrate industry-standard TCP/IP networking facilities optimised for real-time applications.

VxWorks applications typically require fast, predictable responses to external events. To meet this need, VxWorks provides interrupt-driven, priority-based scheduling.

The basic of the VxWorks run-time system is the highly efficient wind microkernel. The microkernel design minimizes system overhead and enables fast, deterministic response to external events.

The run-time environment also provides efficient intertask communication mechanisms, permitting independent tasks to coordinate their actions within a real-time system.

VxWorks is designed for scalability, enabling developers to allocate scarce memory resources to their application, rather than to the operating system. Furthermore, these individual subsystems are themselves scalable, allowing the developer to optimally configure VxWorks run-time software for the widest range of applications. Also, TCP, UDP, sockets, and standard Berkeley network services can all be scaled in or out of the networking stack as necessary.

[1] This includes semaphores, pipes, sockets, and TCP/IP interprocessor communications.
[2] File I/O, sockets, RPC, etc

Applications can be made portable by using the VxWorks POSIX API. In fact, POSIX offers the only practical route for writing applications that are portable across a wide range of computer systems. POSIX is a set of standards and draft standards developed by the IEEE Computer Society to promote portability of application programs across UNIX system environments. VxWorks POSIX API can be used to develop new applications for VxWorks targets and to port existing applications, or application components, to VxWorks systems from other development and target platforms.

Real-Time Computing Using VxWorks

In a real-time computer application, user-written code is combined with an operating system and hardware to form a highly responsive, coherent system. Real-time systems must be able to perform operations such as context switching, intertask communication, and synchronization efficiently.

A real-time application responds predictably to external events. In contrast to other operating systems, a real-time system is normally dedicated to one application. All of the hardware and software resources that comprise the system are focused on that application.

In a real-time system, the execution time for kernel functions must be minimal. The efficiency and predictability of the interrupt processing, context switching, intertask communication, I/O are especially important.

Applications that satisfy real-time requirements can be build With the VxWorks tools.

Host and Target Systems

UNIX host systems and VxWorks target systems are complementary. A UNIX host system can be fully loaded, with a large main memory, large disks, backup media, printers, terminals, and other I/O devices. A completed VxWorks target system, on the other hand, usually has only the resources required by the real-time application. The production version of the target system can be configured to eliminate the software tools that have been used for testing and debugging. A minimal system may comprise a processor, some serial I/O channels for process monitoring and control, and an Ethernet connection.

The VxWorks tools are compatible with UNIX operating system at many levels, especially in the extensive networking facilities. Distributed applications can incorporate VxWorks and UNIX systems; or a group of VxWorks systems can comprise an entire distributed application.

When a target system is part of a network, the other systems on the network can be any computer systems that use TCP/IP networking facilities. The flexible VxWorks network software permits multiple target systems to be connected to a single backplane or connected to an Ethernet link.

VxWorks Application Programming Interface

The VxWorks programming interface is tightly integrated with the VxWorks kernel.

VxWorks applications typically require fast, predictable responses to external events. To meet this need, VxWorks provides interrupt-driven, priority-based scheduling.

Task Control Routines

An application can spawn any subroutine as a separate task, with its own context and stack. Using task control routines, an application can suspend, resume, and delete tasks. An application can also change the priorities of its tasks. Each task in a VxWorks application consists of executable code, a local stack, and a TCB. The local stack is available to the application developer for task-specific data. The task control block is a data structure maintained by the kernel to store other task-specific information.

Application Design and Task Management

VxWorks provides for extensions to tasks, called hooks, to be executed when a task is created, switched, or deleted. When a task is deleted, the kernel does not automatically deallocate the memory that the task has used, because that would use time that might be critical.

Task Scheduling

The VxWorks kernel schedules tasks according to priorities you can assign. The kernel preempts a low priority task to execute a higher priority task whenever the higher priority task becomes ready to run. Therefore, the kernel always gives the processor to the highest priority task available. VxWorks provides 256 priority levels.

Interrupt Handling

Interrupts are critical in real-time systems because they provide links between applications and the hardware being controlled. When an interrupt occurs, it changes the state of the processor, allowing a user-defined routine to execute in response. VxWorks uses the fastest possible means to deal with interrupt by running interrupt service routines in a special context, outside of the task context.

Intertask Communication Management

In multitasking applications, the individual tasks must communicate in order to cooperate. VxWorks offers a variety of intertask communication mechanisms that meet a wide range of requirements (message queues, pipes, semaphores, signals).

Input and Output Handling

The I/O interface for VxWorks applications is uniform, device independent, and compatible with the UNIX I/O system.

Network Hardware Options

Network connections can be made by using Ethernet, SLIP, Processor backplanes or Custom interfaces.

VxSim

VxSim, the VxWorks Simulator, can be used to simulate a VxWorks application in a prototyping and test-bed environment on the UNIX host computer. In most respects, its capabilities are identical to a true VxWorks system running on target hardware. Users link in applications and rebuild the VxWorks image exactly as they do in any VxWorks cross-development environment. The VxWorks system image is an object module on the host.

The difference between VxSim and the VxWorks target environment is that in VxSim, the image is executed on the UNIX machine itself as a UNIX process. There is no emulation of instructions, because the code is for the host's own architecture. A ULIP driver is provided to allow VxSim to obtain an internet IP address and communicate with the host (or other nodes on the network) using the VxWorks networking tools.

VxSim is a complete prototyping and simulation tool for VxWorks applications. By providing full VxWorks simulation on the host workstation, VxSim eliminates the need to purchase additional or evaluation target hardware. It enables application development to begin before hardware becomes available, allowing a large portion of software testing to occur early in the development cycle.

Embedded software developers in many industries often need to begin application development, as the hardware is being designed and debugged. VxSim gives a stable environment to prototype the software to the VxWorks API.

VxSim is a comprehensive prototyping and simulation tool for VxWorks applications. It is intended to assist developers of high-performance embedded systems using custom hardware.

Because target hardware interaction is not possible, device driver development may not be suitable for simulation. However, the VxWorks scheduler is implemented in the UNIX process, maintaining true tasking interaction with respect to priorities and preemption. This means that any application that is written in a portable style and with minimal hardware interaction should be portable between VxSim and VxWorks.

Development organizations that stand to benefit most from VxSim include those waiting for target hardware, those with limited hardware budgets, and those creating application modules that do not require direct hardware access.

Precise Simulation for Efficient Prototyping

VxSim is a port of the VxWorks operating system to the host environment. VxSim accurately implements many of the sophisticated features of VxWorks, including the DOS file system, full UNIX-style networking (TCP/IP, rlogin, etc.) and support for up to 16 targets sharing a common backplane – plus simulation of the VxWorks Backplane Driver for interprocessor communication.

The VxWorks scheduler is fully implemented in the simulator, maintaining true multitasking activity with respect to priorities and preemption.

Complete Tornado Development Environments

In addition to a comprehensive VxWorks simulation, VxSim includes the complete suite of Tornado development tools and optional Wind River tools and accessories. When the hardware becomes available, developers can switch seamlessly to the Tornado development environment for a specific architecture to complete development.

VxSim Data Transmission Protocols

Point - to - Point Protocol (PPP)

PPP provides a standard method for transmitting of multi-protocol datagrams over point-to-point links. The object with PPP is to provide a common solution for easy connecting between different hosts, bridges, and routers. Thus PPP is designed to allow different network protocols to communicate. The protocol is also extensible to meet the requirements of tomorrow.

PPP provides encapsulation for multi-protocol datagrams, a LCP to establish, configure, and test the data-link connection. It also provides NCP for establishing and configuring different network-layer protocols.

User - Level Internet Protocol (ULIP)

ULIP is default network interface for VxSim. Only one process at a time can open the same ULIP device. ULIP has a non-portable nature, which makes it difficult to use in interaction with other protocol stacks.

When debugging applications in system mode, a separate ULIP channel must be enabled for the target. This enables communication with Tornado tools independent of VxWorks.

PPP vs. ULIP

VxSim uses both PPP and ULIP interface to simulate the network IP connectivity.

PPP is available for Solaris2 hosts only, while ULIP can be used for both Solaris2 and SunOS4 hosts.

If one interface is installed on a Solaris host it must be removed before installing the other interface.

PPP is currently the best choice, because of its multi-protocol nature and scalability.

STREAMS

STREAMS is a general, flexible facility and a set of tools that supports implementation of services for high-performance networks. It provides full-duplex processing and data transfer path between device driver and application, supports multiplexing and also allows re-use of standard modules. It consists of a set of system calls, kernel resources, and kernel routines.

STREAMS is a tool well suited for development and implementation of protocol stacks. STREAMS' building blocks represent a protocol layer in the protocol stack. Its flexibility allows easy modification of the stack, like adding and deleting layers. C-language is used for developing STREAMS protocol stacks.

STREAMS is very effective development tool since it reduces duplicated code and buffer space. The standard interface and mechanism enable modular, portable development and easy integration of high performance network services and their components.

It is important to mention that STREAMS does not impose any specific network architecture, but has a user interface that is upwardly compatible with the character I/O user level functions.

There are many advantages and benefits provided by STREAMS development tool.

The most important features of STREAMS are flexibility, portability, and reusability in development of UNIX system communication services. It can also be used to easy create modules that offer standard data communications services and the ability to manipulate these. STREAMS simplifies the creation of modules with its a service interface. The STREAMS service interface is a specified set of messages and the rules that allow passage of these messages across the boundary.

Within STREAMS it is possible to interchange alternate protocol modules and device drivers if they are implemented to an equivalent service interface. The modules developed with STREAMS can be used with different drivers on different machines as long as the compatible service interfaces are implemented.

From the user level point of view, it is important to remember that kernel programming, assembly, and link editing are not required to create the interconnection between modules. STREAMS also provides the capabilities to manipulate modules from user level.

STREAMS Structure

On the top of the STREAMS structure is a STREAMS application that runs in user space. The STREAMS messages can be sent and received by applications due to the application interface to the STREAMS framework.

Every Stream consists of at least a Stream head at the top and a Stream end (called STREAMS driver) at the bottom. This part of STREAM runs in the kernel space. The Stream head processes all system calls made by a user level process on a Stream. Additional modules, which consist of linked pairs of queues, can be inserted between the Stream head and Stream end. This can de done if additional processing of the data being passed along the STREAM is required. Data is passed between modules in messages. Data travelling from the Stream head to the Stream end is said to be travelling downstream, or in the write direction. On the other hand, data travelling from the Stream end to the Stream head is said to be travelling upstream, or in the read direction.

Figure 4 illustrates a typical STREAMS structure.

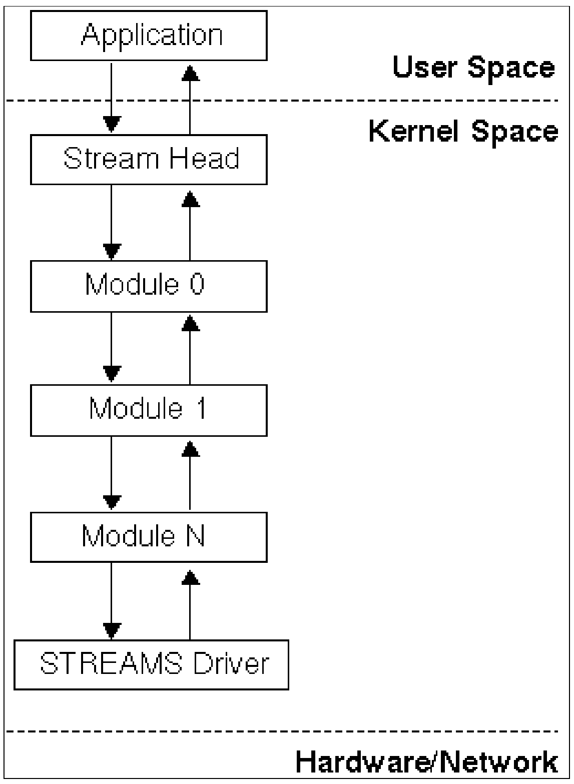


Figure 4 Example of STREAM

Stream Head

The Stream head contains of a set of routines and data structures that provides an interface between user processes and the Streams in the kernel. It is created when the application issues an open system call.

The Stream head performs several major tasks.

It is used as an interpreter of a standard subset of STREAMS system calls, e.g. write and putmsg. These system calls are afterwards translated into a standard range of STREAMS messages, consisting of both data and control information. The next step is to send the messages downstream to the next module or the driver.

The Stream head also receives messages sent upstream from the driver. The STREAMS messages from kernel space are transformed to a format appropriate to the system call (such as getmsg or read) made by the application.

Modules

Module is a defined set of kernel-level routines and data structures used to process data, status, and control information. When developing a protocol stack, the modules also represent protocol layers. Modules process data as it passes from the Stream head to the Stream end and back. If the driver can perform all of the necessary processing on the data, a STREAM has zero modules. Otherwise a STREAM will require additional modules for processing of data.

Each module consists of a pair of queues, which contain data and pointers to other structures that define what each module does. One queue handles data moving downstream and the other handles data moving upstream. Modules are used to manipulate messages as they flow through the Stream.

Depending on their processing requirements, applications request particular modules be pushed onto the STREAM. The Stream head assembles the modules requested by the application and then routes the messages through the pipeline of modules.

Messages are used to pass the information between modules. There are defined several types of messages within the STREAMS environment, but they are usually divided into the two categories, normal and high priority.

Messages with normal priority are processed in the order that they are received using flow control and queuing mechanisms, while priority messages are passed along the stream in an expedited manner.

STREAM End

The Stream end is a special form of STREAMS module. It can be either a hardware[3] or pseudodevice[4] driver. Regardless of the driver type, the Stream end receives messages sent by the module above it, interprets them, and performs the requested operations. It then creates an appropriate message, which contains data and control information to the application. The message is then sent upstream towards the Stream head. The driver also provides and manages a path for the data to and from the hardware device, and services interrupts issued by the device controller.

Although STREAMS drivers and modules are quite alike, there are several differences between them. Device drivers are initialised and deinitialised by the open and close system calls, while modules use the other system calls. Device drivers can have one or more interrupt routines to be accessible from a hardware interrupt as well as from the Stream. They can also be connected to multiple STREAMS, called multiplexor.

STREAMS Messages and Queues

STREAMS Messages

The objects passed between STREAMS modules are pointers to messages. As illustated in Figure 5, STREAMS messages consist of one or more linked message blocks. Each STREAMS Message Block is a linked triplet of the following components, two structures (message block, and data block), and a variable length data buffer. Messages can be assigned different priorities, based on their intended purpose and their queuing priority.

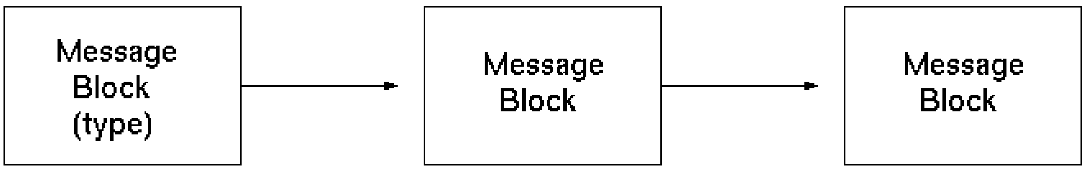


Figure 5 A message structure

Message block contains pointers to the information that the message owner can manipulate. These blocks can be chained for multi-part messages. The structures two fields are the read and write pointers into the data buffer. Messages are sent through a Stream by successive calls to the put procedure of each module or driver in the Stream.

Data block contains information about buffer characteristics, e.g. reference counts and limits and sizes.

The data buffer is a location in memory where the data of a message are stored. It contains the actual data that makes up the message. STREAMS imposes no alignment rules on the format of data in the data buffer.

[3] The Stream end provides communication between the kernel and an external communication device
[4] The Stream end is implemented in software and is not related to an external device

When data is travelling downstream the Stream head will create and fill in the message data structures. On the other hand, when data is travelling upstream the STREAM end will create and fill in the message data structures.

STREAMS modules and drivers must provide open, close, and configuration processing, while the other kinds of processing are optional.

STREAMS Queues

STREAMS uses queue structures to keep information about given instances of a pushed module or opened STREAMS device.

Status information, a pointer to routines processing messages, and pointers for administering the Stream is stored in a data structure called queue. Queues are always allocated in pairs, one queue for the read-side and other for the write-side. Each driver, module and the Stream head have their queue pair, which is allocated whenever the Stream is opened or the module is added onto the Stream.

Message queue is a linked list of messages waiting for processing. The first block of a message on a queue contains links to preceding and succeeding messages on the same queue, as illustrated in Figure 6. It also has a link to the succeeding block of the message. The queue structure contains queue head and tail.

STREAMS utility routines enable developers to manipulate messages and message queues.

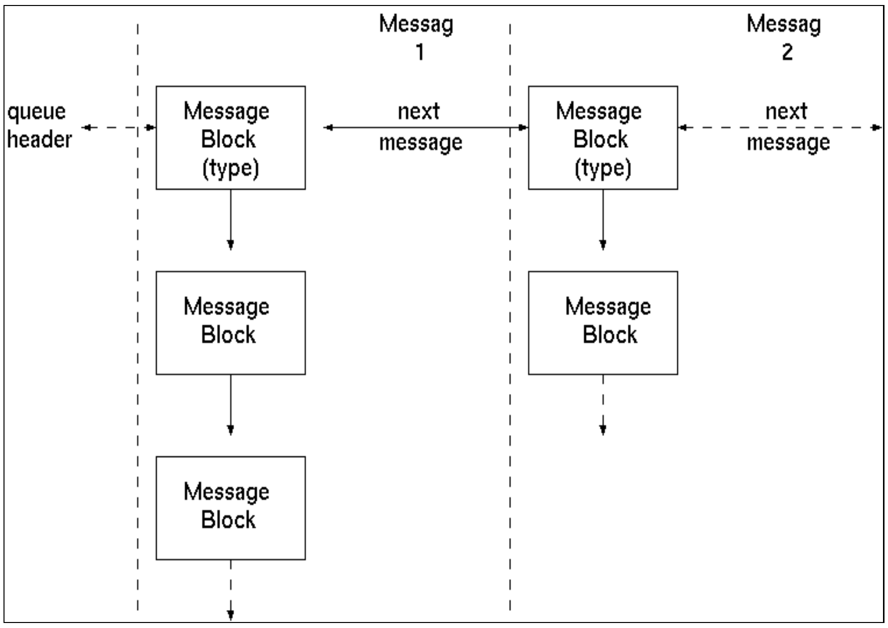


Figure 6 Messages on a Message Queue

STREAMS Queue Structure contains procedures and limits, list of messages to be processed and link to next queue to be serviced.

Message queues grow when the STREAMS scheduler is delayed from calling a service procedure due to system activity, or delay in the procedure execution. The service procedure on the queue processes queued messages in a FIFO manner. When a message is queued, it is placed after the messages of the same priority already on the queue. This affects the flow control parameters associated with the band of the same priority.

Chapter 3 Problem Solving

In the early stages of the project it was decided to divide the assignment into two subtasks.

The first task was to find out if it was possible to use VxSim to simulate the code written for GRPS supporting nodes. A solution for what needed to be done in order to use VxSim should also be suggested.

The second task was to test features of VxSim and compare the results with VxWorks. The original intension was to test in the existing environment used otherwise in the Ericsson. Due to problems that aroused during testing, it proved to be unsuitable for the purposes of the project. At this time, the original test environment was unstable and that led to many problems caused by external factors. There were many people who modified, improved and worked on the same environment. This had a negative influence on the test results. This also led to another problem: the project became too dependent of other people and their work. This meant long delays, something we had neither influence nor control of. The solution to these problems was to develop a new test environment, which would test the features of VxSim.

Porting to VxSim

To make things easier Makefiles are used to compile and link executable files for VxWorks, VxSim, and Solaris. Ericsson AS use Makefiles in the project, developed for the purposes of their GPRS software. These Makefiles define setting flags for the source code and compiling environment (originally VxWorks and Solaris).

Before the source code could be compiled for and ported to VxSim, it was necessary to verify that VxSim was adequate for simulation and testing of Ericsson’s software. This involved verifying of several aspects of Ericsson’s source code for VxSim. These aspects include STREAMS and POSIX features, C/Erlang[5] interpreters and the existing GPRS libraries.

STREAMS is a commonly used tool for software and protocol development in Ericsson. Therefore the first step towards porting of VxWorks source code to VxSim was to find out if VxSim supports STREAMS. Otherwise it would be useless for simulation and testing of Ericsson’s software.

In order to make use of STREAMS features, Vxsim had to be reconfigured. This involved modifying one of two configuration header files, configAll.h or config.h.

configAll.h is the global configuration header file, meaning that its parameters define the default configuration for all targets. There are several macros specified in configAll.h that define defaults for provided facilities in VxSim (e.g. kernel configuration parameters, I/O system parameters, system tasks, MMU/cache modes, etc).

[5] Programming language developed by Ericsson

config.h is a target-specific configuration header file, which contains definitions that apply only to the specific target. Its definitions override and extent the base configurations specified in configAll.h. config.h defines numerous of macros, such as memory addresses and sizes, interrupt vectors, clock rates, etc. It is read whenever a VxWorks or VxSim image is built.

It is strongly advised against modifying configAll.h because that affects all targets in the network. After editing the configuration files, it is necessary to rebuild VxSim and reboot the target for the changes to take effect.

In order to add the STREAMS facility to VxSim the appropriate macro must be placed in the included-facilities section in config.h header file. Still, other problems could arise, and among others, mismatch in versions regarding VxSim and STREAMS. There were some uncertainties regarding the compatibility between STREAMS v2.0 and VxSim v1.0.1, used at Ericsson. This caused no problems, since the versions proved to be compatible.

The other problem that occurred during porting was use of POSIX features. POSIX is another feature commonly used for software development in Ericsson. Since this feature is not included in VxSim by default, the POSIX macro had to be placed in the included-facilities section in config.h header file.

GPRS software in Ericsson is a complex environment consisting of different programming languages, mainly C and Erlang. Specific libraries are developed in order to provide communication between these two languages. These were adequate for VxWorks and had to be recompiled so that they would be executable for VxSim. This also made it possible to use Erlang source code during simulation and testing in VxSim.

Other Ericsson specific library and header files had to be recompiled in order to be executable for VxSim.

Certain modifications of the Makefiles had to be done in order to compile the existing source code for VxSim. This involved several additions to the Makefile.

First of all the Makefile was extended to compile source code for VxSim environment. A special flag is used to indicate VxSim environment and a specific VxSim compiler, ccsimso, was introduced. ccsimso is recommended because it is a great possibility that it would be a compiler for other VxWorks cross development. It is also possible to use other compilers, for example a native compiler. A new type of CPU, SIMSPARCSOLARIS, had to be used for VxSim. SIMSPARCSOLARIS refers to the architecture family of the Solaris operative system used for software development in Ericsson.

It is of crucial importance to avoid compiling using UNIX header files. The user source code must only be linked to VxSim libraries. The use of UNIX libraries might cause overlapping of several functions.

All compiled binaries are dynamically linked when loaded on target. The compiler must therefore create relocatable object modules without linking. This is done when binaries are loaded into VxSim. In order to accomplish this a special flag (-r) must be used during compilation.

Original Testing Environment

GPRS is a new service designed for digital mobile networks and is used for carrying end user's packet data between GPRS terminals. GPRS optimises the use of network and radio resources, allowing the network subsystem to be reused with other radio access technologies.

The radio interface resources can be shared dynamically between speech and data services as a function of service load and operator preference. Various radio channel-coding schemes allow bit rates from 9 kbit/s to more than 150 kbit/s. User Charging will typically be based on the amount of data transferred.

GPRS introduces two new network nodes in the GSM network. The SGSN keeps track of the individual MSs location and performs security functions and access control. It also performs authentication and cipher setting procedures based on the same algorithms, keys, and criteria as in existing GSM. The GGSN provides interworking with external packet-switched networks, and is connected with SGSNs via an IP-based GPRS backbone network

In order to access the GPRS services, a MS shall first make its presence known to the network by performing a GPRS attach. This operation establishes a logical link between the MS and the SGSN, and makes the MS available for SMS over GPRS, paging via SGSN, and notification of incoming GPRS data.

In order to send and receive GPRS data, the MS shall activate the packet data address that it wants to use. This operation makes the MS known in the corresponding GGSN, and interworking with external data networks can commence.

Figure 7 illustrates a suggestion to how the SGSN and GGSN should be implemented. The test environment developed in Ericsson is used to test delay and throughput for the SGSN and GGSN.

The delay specifies maximum delay a packet of specific size should experience through network because it is impossible to guarantee delay for bursty sources. The responsibility of the SGSN is to ensure fair distribution of delay for incoming packets with an appropriate scheduling algorithm.

The throughput defines the maximum rate at which data is to be transferred across the network. SGSN will limit throughput of payload traffic in order not to overload the BSS with traffic. This means that SGSN is responsible of flow control in network.

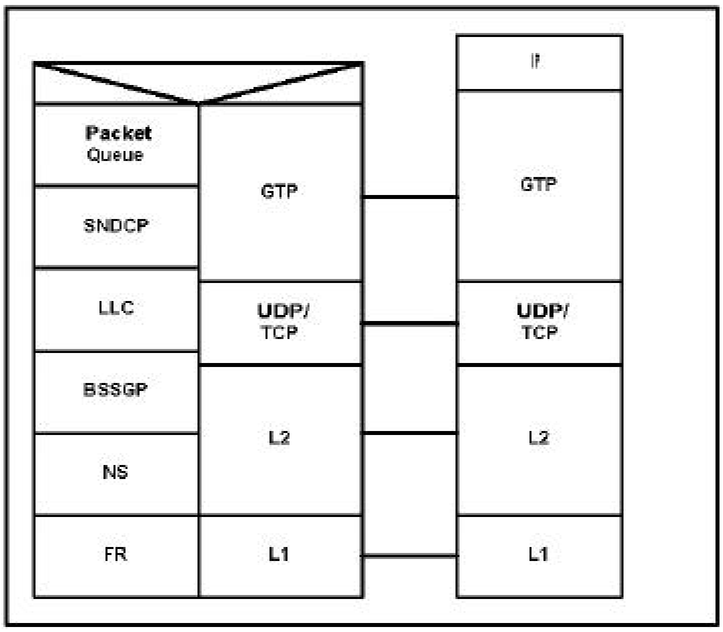


Figure 7 SGSN and GGSN in GPRS network

After a brief introduction in Ericsson's GPRS software and testing routines, a testing process in the existing environment could start. This involved recompiling and simulating the source code. The first step was to compile the software for VxSim and load it on the target. After this was done, the simulating part of the assignment could also begin. This proved to be far more complicated than expected due to several problems.

The original test environment was unstable and many people modified, improved and worked on the same environment. This had a negative influence on the test results and made it almost impossible to test and simulate it. In addition the test process became too dependent of other people and their work, which meant long delays. Therefore, a new testing environment, which would test the basic features of VxSim, had to be developed.

New testing environment

In order to test and compare the features of VxWorks and VxSim properly, a new test environment similar to the original had to be developed. To achieve this, a STREAMS environment, as illustrated in the Figure 8 was developed.

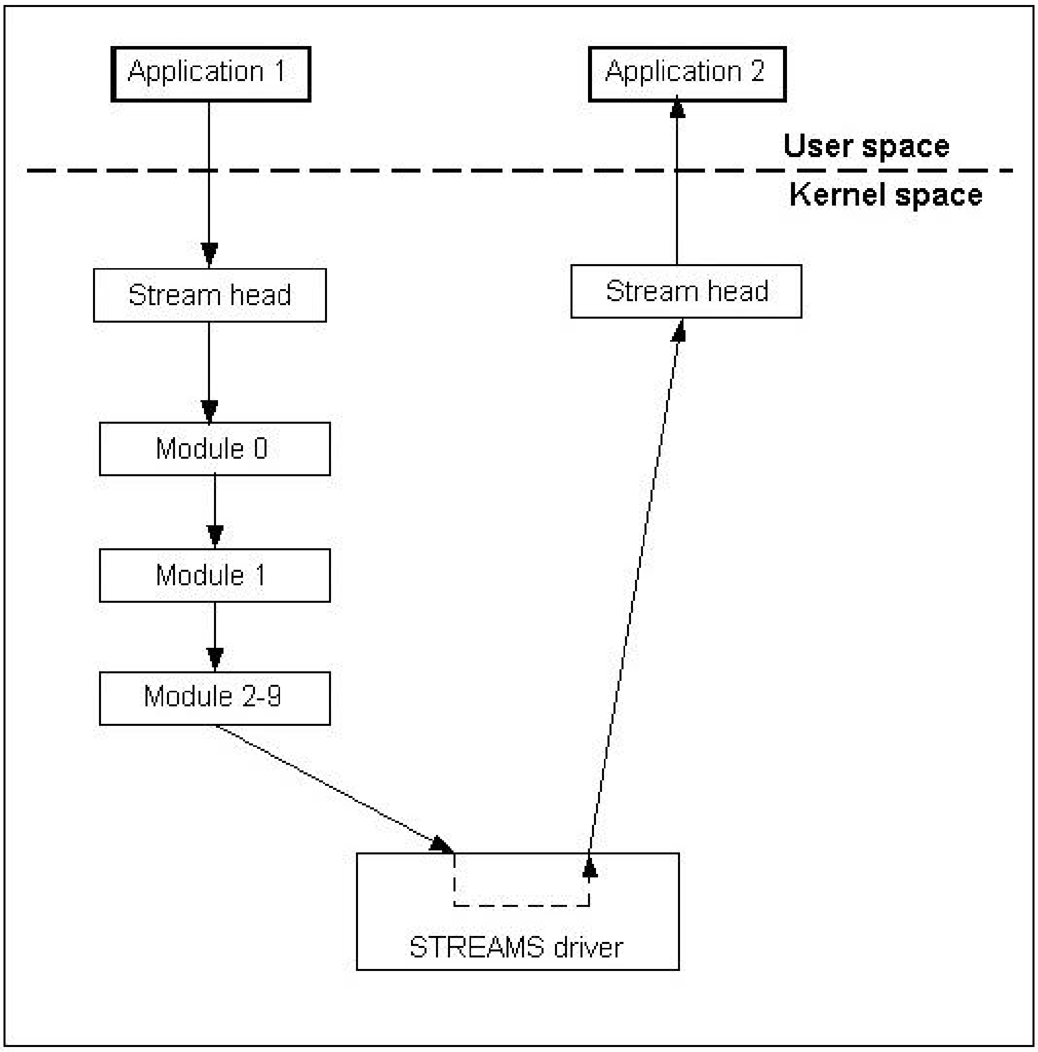


Figure 8 The test STREAMS environment

UDP transmits all the signalling between the Ericsson's GSNs. To simulate the payload transmission, UDP server was programmed on the top of STREAMS. The server takes a file as input, which is sent to the driver through the modules. To simulate the delay in the GSNs, the modules were programmed to perform some processing of the incoming messages. This processing is simple and is added to cause some delay during the simulation.

It was also desirable to test capabilities of modules to perform memory allocation and management. Ericsson GSNs are put together in linked lists. For that purpose each module was extended to create and initialise linked list. Therefore it was desirable to test if this could be done in modules and how it influences the performance.

At the Stream head the file is divided into STREAMS messages, as illustrated in the Figure 9, and sent downstream on the appropriate write queue. The modules receive the messages from the read queue and process them according to their message type.

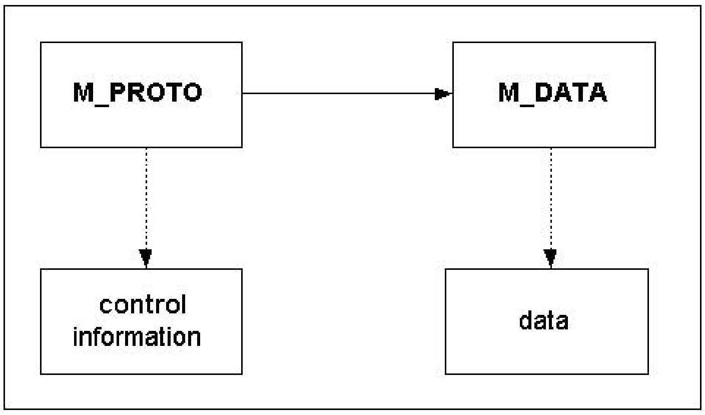


Figure 9 STREAMS message

The modules also add the appropriate header after processing, as illustrated in the Figure 10. Since M_DATA[6] and M_PROTO[7] messages are commonly used in the Ericsson's GSNs, modules were implemented to process only these messages. The other messages would not be recognizable to the modules and would be sent downstream.

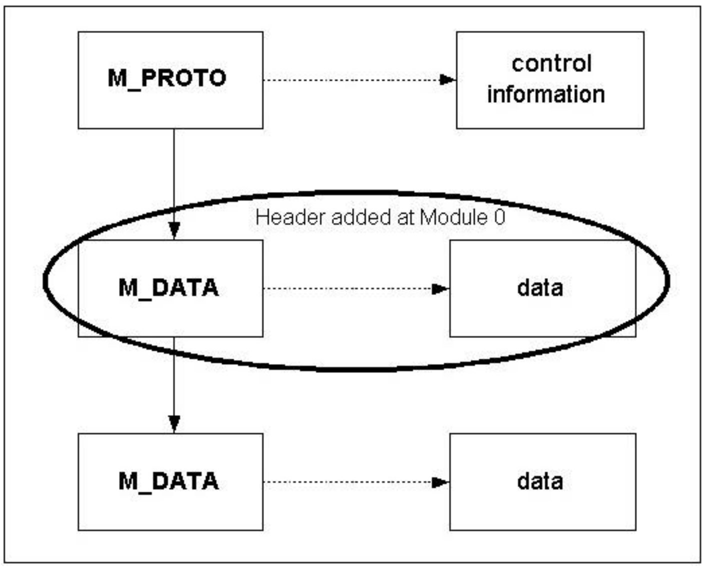


Figure 10 STREAMS message after adding the header

The top-most module, Module 0, counts the appearances of a given character and creates its header. The value is then written to the header. The next module, Module 1, counts all the characters in the message and writes the value to the header created at the module above. The modules also create and initialise a linked list containing 200 nodes. The delay caused during these operations is adequate the processing delay at the actual GSNs.

At the Stream end, the STREAMS Driver, will turn messages around and send it back to the UDP application.

These instructions are adequate to instructions the drivers in Ericsson's GSNs perform, which send messages to the upper protocol layer for further processing. The test driver sends them to an application, where the payload data is then written to the file.

In order to compare VxWorks and VxSim, the time it takes to build the testing STREAMS environment, send and process messages in the modules and receive the messages at the application in user space was measured. Whenever one of these events occurs, a timer is started. The payload size and frequency are random, depending on the size of files sent to the UDP server and the number of the clients requesting the service.

[6] M_DATA: User data message for I/O system calls
[7] M_PROTO: Protocol control information

The timer used during testing is a modified version of the timer used otherwise in Ericsson's GSN software.

The testing was executed on the different processors used by Ericsson in the GSNs. The purpose with testing on different processor architectures was to find out which one is the most suitable for VxWorks and VxSim. It was also desirable to find out if it makes a difference which tool is used on which processor.

There are three different processors used as the basis for test and simulation environment in the Ericsson, PowerPC 603 (HAM), PowerPC 604 (PPC) and PowerPC 860 (E1).

PowerPC 603 (HAM)

The PowerPC microprocessor is a low-power implementation of the PowerPC RISC architecture. It offers high-level performance combined with low-cost design and has the ability to execute multiple instructions in parallel.

The microprocessor's design is super scalar, capable of issuing three instructions per clock cycle. The PowerPC 603 has separate 8k physically addressed instruction and data caches, which are two-way set-associative. It also contains separate MMUs for instructions and data. One of the advantages of PowerPC 603 is its flexible bus interface, which is implemented as a selectable 32- or 64-bit data bus and a 32-bit address bus.

PowerPC 604(PPC)

The PowerPC 604 is a 32-bit implementation of the PowerPC RISC architecture, jointly developed by Apple, IBM, and Motorola. The PowerPC architecture specification has seen a number of implementations from both IBM Microelectronics and Motorola.

The microprocessor's design is super scalar, capable of issuing four instructions per clock cycle. The PowerPC 604 has a separate 16k physically addressed instruction and data caches, which are four-way set-associative and provide parity checking at the byte level. Just as PowerPC 603, PowerPC 604 contains separate MMUs for instructions and data. The PowerPC 604 features a 64-bit external data bus and a separate 32-bit external address bus.

PowerPC 860 (E1)

PowerPC 860 is highly integrated microprocessor optimised for the telecommunications, internetworking and data communications markets.

The PowerPC 860 has a separate 4k -instruction cache and 4k data cache. Similar to PowerPC 603 and 604, PowerPC 860 contains separate MMUs for instructions and data. The PowerPC 860 has up to 32-bit data bus, with dynamic bus sizing feature (8, 16 and 32 bits). It also supports a 32-bit address bus.

Chapter 4 Implementation and Results

Implementation

The test was done on three different DPs used at Ericsson, in addition to VxSim.

Ericsson’s DPs have different clock frequencies; PowerPC 604 is the fastest with 300MHz clock frequency, PowerPC 603 has 200MHz clock frequency and PowerPC 860 is the slowest, with 33MHz clock frequency. The clock frequency is one of the crucial factors for the test results. The other factors are processor’s design and memory management.

VxSim was run on an UltraSPARC 5 (CPU: UltraSparc |||i 360 MHz) and UltraSPARC 1 (CPU 143 MHz). Both workstations run Solaris operative system. UltraSPARC 5’s other features include Memory 256MB (50ns), and SWAP size 699.4MB. Memory 256MB and SWAP size 526.8MB are UltraSPARC 1’s respective features.

The test program was loaded and executed on each target as well as on VxSim. There was no other user processes on target that might interfere with the test program. Since VxSim is executed on the local UNIX host it was impossible to exclude all other user processes. VxSim is a UNIX process and is therefore affected by both system and other user processes. It also means that VxSim is not affected by any delays in the network. Still several other sources might cause errors during testing, for both VxWorks and VxSim.

VxWorks might be affected by delays in the network, which could bring errors when receiving test results. Since ATM network is used at Ericsson with speed of 100Mbps, it is reasonable to assume that the network would not affect the results. Bus speed is usually much lower than the network, and could cause some delays during transmission of data. Another source of error might be the system clock, which might cause timing errors during testing. The timer’s precision might also interfere with the timing and give inaccurate values.

VxWorks is a real time operative system and there are system processes running in background. These have high priority and can suspend other processes executing on target. This can have negative effects on the timing and might cause errors. The time measured would then include the time it takes to execute both the test program and the system process.

Testing is done with different number of STREAMS modules, ranging from two to nine. Each module performs some processing and memory allocation and management. During testing it was desirable to find out if memory management can be done in STREAMS modules. It is a well-known fact that memory management instructions take a lot of processor time. Therefore it was also required to find out how this would affect processor efficiency. Therefore each module is designed to create a linked list, consisting of 200 nodes.

After the STREAMS environment is build, payload is sent through the modules for processing. The test program measures the time it takes for data packages to be processed in the STREAMS environment, plus time for memory management in each module. Payload size is 1k in the first test case, and 3k in the second. Seeing that processing in the real GSNs is not known, the modules are designed to cause delay based on the size of the payload. This induce that the larger the payload size, the larger the delay. This is analogue to the payload processing in the GSNs.

All test results are given in milliseconds

Assumptions

It is assumed that the DPs with higher clock frequency and faster memory management would, according to this, give the best results.

PPC would therefore give the best test results because of its high clock frequency, 16k cache and ability to execute four instructions per clock cycle. This means that the measured values for PPC should be lower than for both HAM and E1.

HAM with its 200MHz clock frequency, 8k cache and three instructions per clock cycle should be second best in test. Although it has lower performance compared with PPC, its performance would still be much better than E1's. Still the difference between PPC and HAM should not be major because of the similarities between the two DPs. On the other hand, the difference in measured values between these two DPs and E1 should be large, mainly because of E1's low clock frequency and small caches. The other factors have little influence on the test results because E1 is build on the same principles as PPC and HAM.

VxSim is a software package meaning that it is not as reliable as hardware when it comes to timing. Because of this it is reasonable to assume that VxSim will be slower than both the PPC and the HAM. It is also likely to assume that it will be as fast as E1, but more unstable. Still it is hard to predict the behaviour of VxSim because of several other processes that influence it.

The purpose with the test was also to find out what influence payload size has on time it takes to process data in modules.

It is assumed that the more modules being built the longer it takes, regardless of the tools being used.

It is also assumed that the larger payload, the more time it would take the modules to process the data regardless of the tool. Therefore the test cases were done using 1k payload and 3k payload. The purpose was to determine how modules act when different payload sizes are processed and what the consequence is for the time being measured.

Since VxSim is run on the local machine, it was assumed that both the CPU and the Memory had some influence on the results. It is assumed that the UltraSPARC 5 is faster than the UltraSPARC 1.

Test Criteria

In order to test VxSim features some test criteria had to be set. These criteria included timing of payload processing and building of the STREAMS environment, in addition to memory usage in VxSim. The main focus laid on testing of payload processing and building of the STREAMS environment. It also involved comparison of these results with the VxWorks' results.

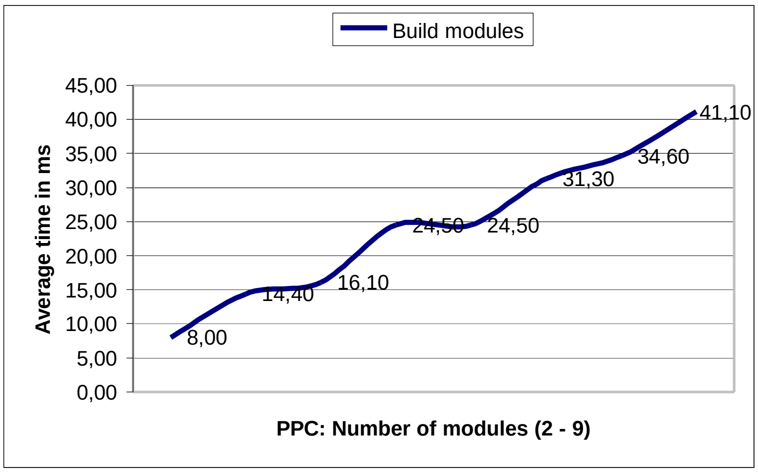
The tests were run twenty times for VxSim, and ten times for VxWorks. This is done to deal with the fact that VxSim is a simulation program, and thus not as stable or accurate as VxWorks. Twenty times is still not enough to give an absolute or accurate result, but it is enough to give an indication of how VxSim will behave given the test criteria.

Due to the fact that not all machines are UltraSPARC 5s , the tests were also run on an UltraSPARC 1. These tests were not as thorough as the one made for comparison between VxSim and VxWorks. Still they give an indication of the CPU usage for both Workstations.

PPC Test Results

PPC is in general fast when building STREAMS environment. The measured values increase with number of modules in STREAMS. Still they do not increase linearly.

Graph 1 illustrates graphical presentation of data for building STREAMS environment with two through nine modules.



Graph 1 Build modules, PPC

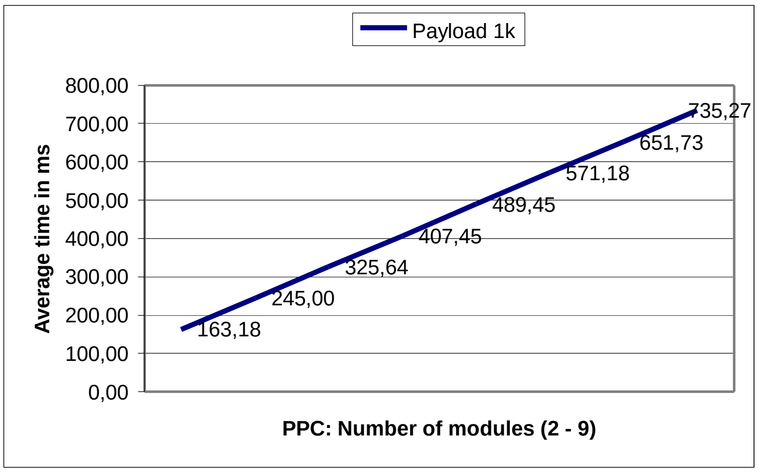
The average values increase, but not by the same or constant value, as illustrated in Graph 1. The increase values vary from 0 (difference when five and six modules are built) to 8.4 (difference when fire and five modules are built).

Table 1 Standard deviation, building modules

# modules	2	3	4	5	6	7	8	9
Stdev	8,43	5,06	0,32	8,96	8,96	5,38	5,06	8,33

Table 1 shows standard deviation when STREAMS environment is built. Standard deviation is lowest for test results when STREAMS environment with fire modules is build (0.32) and highest for STREAMS environment with five and six modules (8.96). This means that test results when there are four modules are most reliable and accurate. Test results for five and six modules contain most uncertainty and inaccuracy in results.

The measured average values, when payload size is 1k, increase linearly with the number of modules in STREAMS environment by approximately 82.15, as illustrated in Graph 2



Graph 2 Payload 1k, PPC

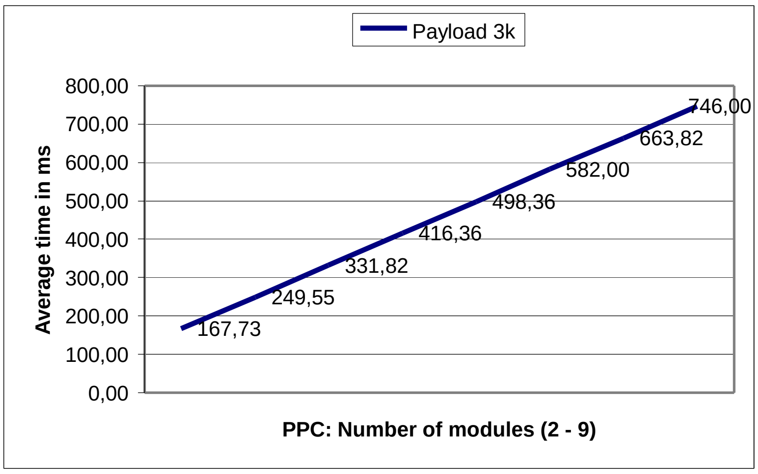
Average values increase form 163.18 ms when payload is sent through two modules to 735.27 ms when there are nine modules.

Table 2 Standard deviation, 1k payload

# modules	2	3	4	5	6	7	8	9
Stdev	6,54	7,75	8,46	8,61	8,39	7,82	5,16	5,24

Standard deviation is lowest for test results when 1k payload is sent through eight modules (5.16) and highest for STREAMS environment with five modules (8.61), as shown in Table 2. Standard deviation values are not high and show some inaccuracy and uncertainty in results. Test results can thus be considered stabile and reliable since standard deviation values are low.

The measured average values when payload is 3k increase linearly with the number of modules in STREAMS environment by approximately 82.61, as illustrated in Graph 3.



Graph 3 Payload 3k, PPC

Graph 3 illustrates how average values increase form 167.73 ms when payload is sent through two modules to 746 ms when there are nine modules.

Table 3 Standard deviation, 3k payload

# modules	2	3	4	5	6	7	8	9
Stdev	5,08	0,93	5,27	0,50	4,86	5,33	6,35	7,75

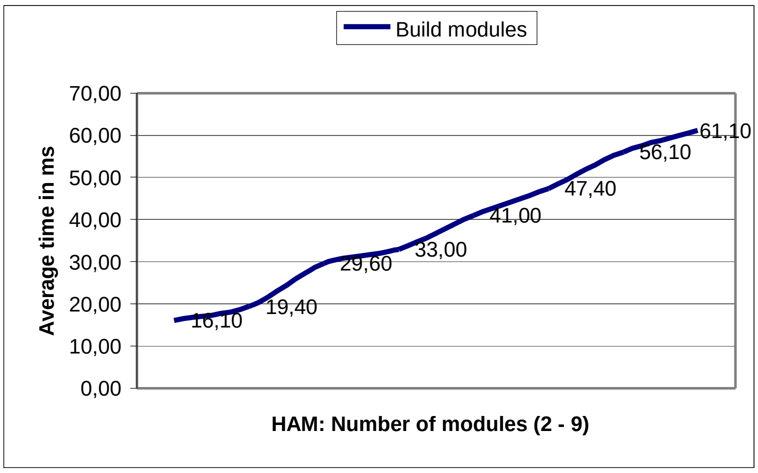
Standard deviation is lowest for test results when 3k payload is sent through five modules (0.5) and highest for STREAMS environment with nine modules (7.75), as shown in Table 3 above. Standard deviation values are quite low and show some inaccuracy and uncertainty in results. Thus test results can be considered stabile and reliable since standard deviation values are neglectable.

As one can see from test results, the payload size has not a major influence on time it takes to process data in modules. It does take longer time process data when payload is greater, but the difference is small. Thus the difference grows bigger when number of modules increases.

HAM Test Results

HAM is, as expected, slower than PPC when building STREAMS environment. HAM can still be considered as quite fast when it comes to building STREAMS environment. The measured values increase with number of modules. Thus there is not linear dependence between the measured values and number of modules.

Graph 4 illustrates graphical presentation of data for building STREAMS environment with two through nine modules.



Graph 4 Build modules, HAM

As the test results show, the difference between the first average values is quite small (3.3 ms), while it increases strongly to 10.2 ms for next cycle. The difference between two next cases sinks to 3.4 ms, while it for the rest of the cases grows more linearly.

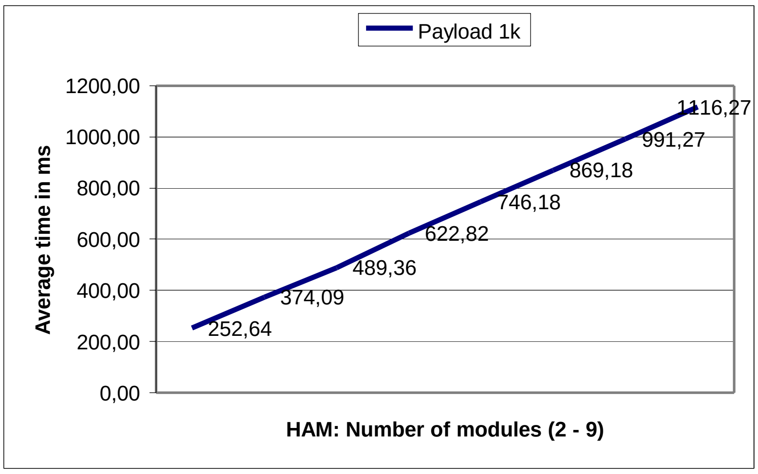
Table 4 Standard deviation, building modules

# modules	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Stdev	0,32	7,17	7,17	0,00	8,43	5,06	8,76	8,36

Table 4 shows standard deviation when STREAMS environment is built. Standard deviation varies form 0 for test results when STREAMS environment with five modules is build to 8.76 for STREAMS environment with eight modules. This means that test results for five modules are most reliable and accurate. Test results for eight modules contain most uncertainty and inaccuracy in the results.

HAM is also slower then PPC when payload is sent through STREAMS environment. This applies for both 1k and 3k payload.

The measured average values, when payload size is 1k, increase linearly with the number of modules in STREAMS environment by approximately 123.37, as illustrated in Graph 5.



Graph 5 Payload 1k, HAM

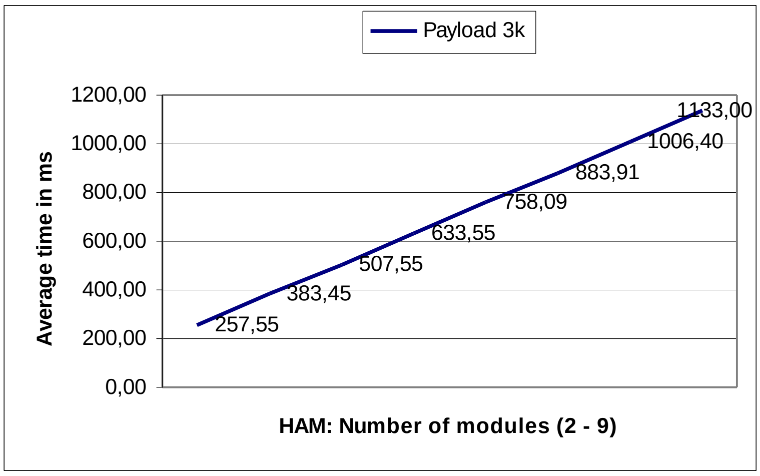
Average measured values increase form 252.64 ms when payload is sent through two modules in STREAMS environment to 1116.27 ms when there are nine modules in STREAMS environment.

Table 5 Standard deviation, 1k payload

# modules	2	3	4	5	6	7	8	9
Stdev	6,67	8,93	29,76	8,28	8,26	5,65	8,36	0,47

Table 5 shows standard deviations when 1k payload is send through modules. Standard deviation is lowest for test results when 1k payload is sent through nine modules (0.47) and highest for STREAMS environment with fire modules (29.76). This means that test results for nine modules are most reliable and accurate. Test results for four modules contain most uncertainty and inaccuracy in the results.

The measured average values, when payload size is 3k, increase linearly with the number of modules by approximately 125.06, as illustrated in Graph 6.



Graph 6 Payload 3k, HAM

As illustrated in Graph 6, average values increase form 257.55 ms when payload is sent through two modules to 1133 ms when there are nine modules.

Table 6 Standard deviation, 3k payload

# modules	2	3	4	5	6	7	8	9
Stdev	8,32	0,52	8,32	0,52	8,15	0,30	8,26	0,00

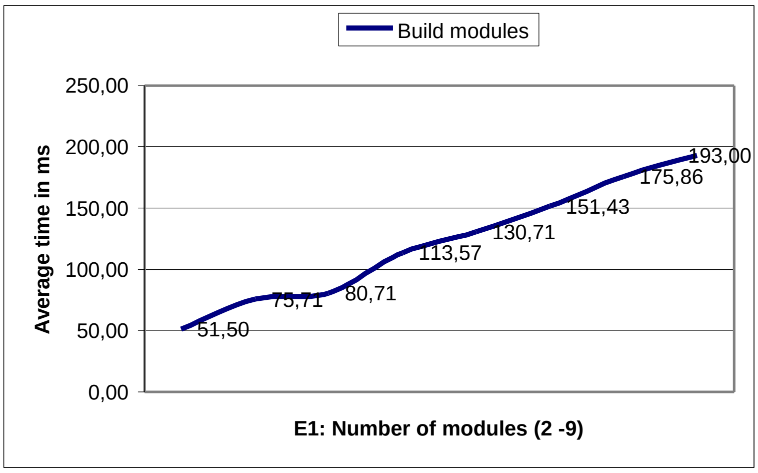
Table 6 shows standard deviations when 3k payload is send through modules. Standard deviation is lowest for test results when 3k payload is sent through nine modules (0) and highest for STREAMS environment with two and four modules (8.32). Standard deviation values are not high and show slice inaccuracy and uncertainty in results. Test results can be considered stabile and reliable since standard deviation values are neglectable. Still the standard deviation is quite high in some test cases, which can imply unreliable results and unstable environment.

Payload size has no major influence on time it takes to process data in modules. The measured time is slice greater for payload size 3k.

E1 Test Results

As expected, E1 is the slowest DP both when building STREAMS environment and when sending payload through modules. The measured values increase with number of modules considerably.

Graph 7 illustrates graphical presentation of data for building STREAMS environment with two through nine modules.



Graph 7 Build modules, E1

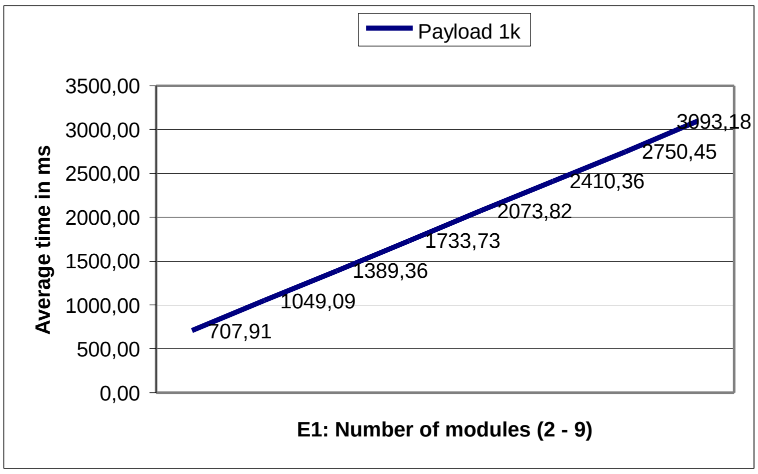
As the test results show the difference between the first two average values is quite large (24.21 ms), while it decreases strongly to 5 ms for next cycle. The difference between next cases shows more linear dependence between average measured time and number of modules.

Table 7 Standard deviation, building modules

# modules	2	3	4	5	6	7	8	9
Stdev	8,26	8,86	9,65	5,38	7,23	7,09	8,97	8,33

Table 7 shows standard deviation when STREAMS environment is built. Standard deviation is lowest for test results when STREAMS environment with two modules is build (6.3) and highest for STREAMS environment with nine modules (9.38). This means that test results for two modules STREAMS environment are most reliable and accurate. Test results for nine modules STREAMS environment contain some uncertainty and inaccuracy in results. Standard deviation values are somewhat higher than for both PPC and HAM. This means that E1’s test results are the most unstable.

The measured average values, when payload size is 1k, increase linearly with the number of modules in STREAMS environment by approximately 340.75, as illustrated in Graph 8.



Graph 8 Payload 1k, E1

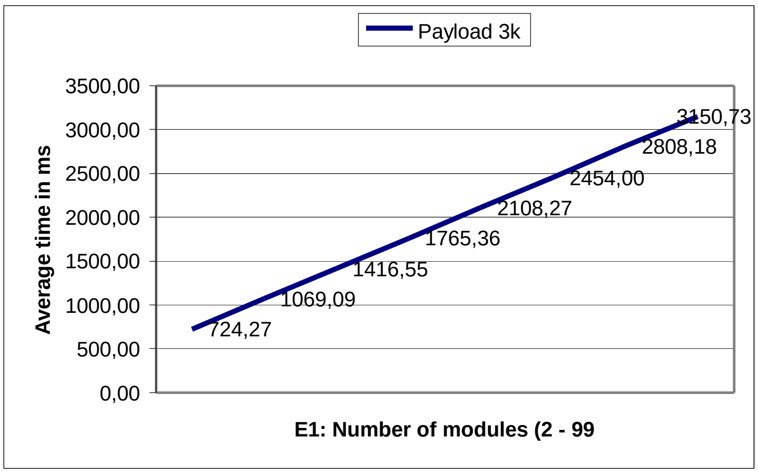
Average measured values increase form 707.91 ms when payload is sent through two modules to 3093.18 ms when there are nine modules.

Table 8 Standard deviation, 1k payload

# modules	2	3	4	5	6	7	8	9
Stdev	8,35	7,38	8,46	0,47	8,99	11,22	0,93	8,07

Table 8 shows standard deviations when 1k payload is send through modules. Standard deviation is lowest for test results when 1k payload is sent through five modules (0.47) and highest for seven modules (11.22). Standard deviation values are generally low and show some inaccuracy and uncertainty in results. Test results can be considered stabile and reliable since standard deviation values are neglectable. Thus standard deviation is quite high in some test cases, which can imply unreliable results and unstable environment.

The measured average values, when payload size is 3k, increase linearly with the number of modules in STREAMS environment by approximately 346.64, as illustrated in Graph 9.



Graph 9 Payload 3k, E1

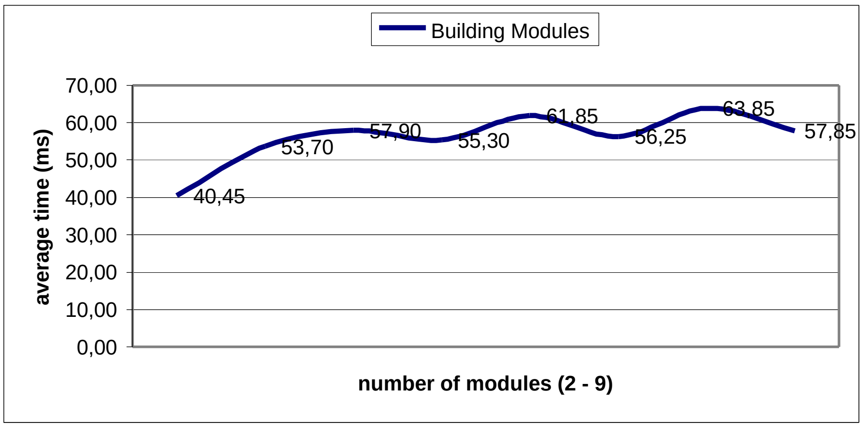
As illustrated in Graph 9, average values increase form 724.27 ms when payload is sent through two modules to 3150.73 ms when there are nine modules.

Table 9 Standard deviation, 3k payload

# modules	2	3	4	5	6	7	8	9
Stdev	9,13	6,88	7,84	11,63	8,88	8,05	8,26	5,10

Table 9 shows standard deviations when 3k payload is send through modules. Standard deviation is lowest for test results when 3k payload is sent through nine modules (5.1) and highest for STREAMS environment with five modules (11.63). Standard deviation values are quite high and show a deal of inaccuracy and uncertainty in results. Thus test results can be considered stabile and reliable since standard deviation values can be considered as tolerable.

VxSim



Graph 10 Build modules

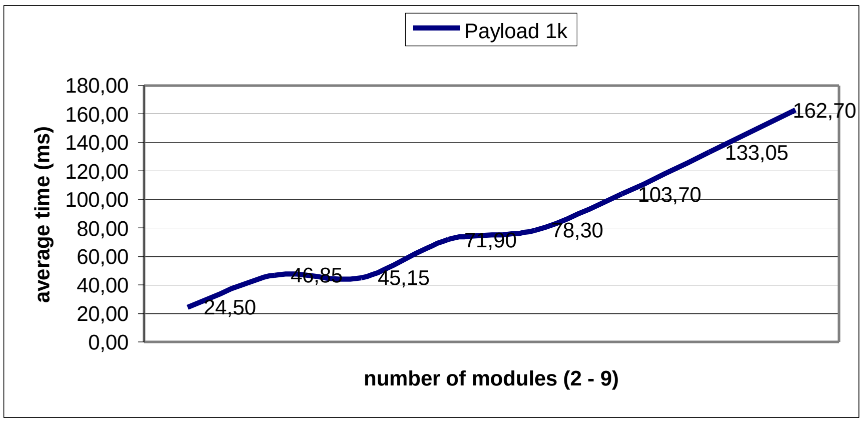
Graph 10 illustrates the average time it takes to build the STREAMS environment, with two to nine modules using VxSim.

As graph 10 illustrates, no certain conclusions as to how long it will take to build modules in VxSim can be given. The graph also indicates that the time it takes to build modules does not depend of the number of modules being build.

Table 10 Standard deviation, building modules

# modules	2	3	4	5	6	7	8	9
Stdev	51,16	52,57	52,47	53,28	56,03	54,57	72,44	43,93

The standard deviation calculated for building modules vary from 43.93 ms (9 modules) to 72.44 ms (8 modules). These values indicate that VxSim is unreliable and unstable when it comes to building the STREAMS environment.



Graph 11 Payload 1k

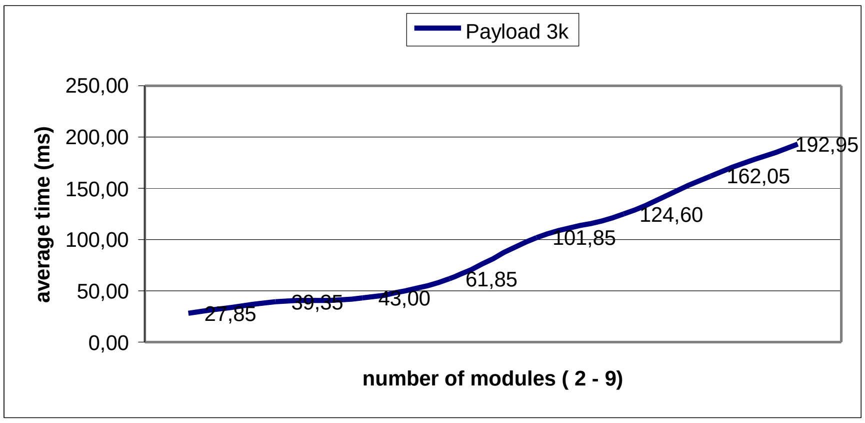
Graph 11 illustrates the average time it takes to pass payload of 1k through two to nine modules using VxSim.

As graph 11 illustrates the measured time depends on the number of modules. This makes sense, as the payload is not just passed through the modules. As discussed earlier the modules also build linked list and process the payload before passing it on.

Table 11 Standard deviation, 1k payload

# modules	2	3	4	5	6	7	8	9
Stdev	10,24	12,33	11,74	37,99	30,44	44,12	56,13	41,04

The standard derivation calculated for passing 1k payload through the modules vary form 10.24 ms (2 modules) to 56.13 ms (8 modules). This indicates that VxSim is unstable and unpredictable when it comes to passing 1k payload through the modules.



Graph 12 Payload 3k

Graph 12 illustrates the average time it takes to send payload, both 1k and 3k, through the modules.

As graph 12 illustrates that the time also depends on the number of modules when the payload is 3k.

Table 12 Standard deviation, 3k payload

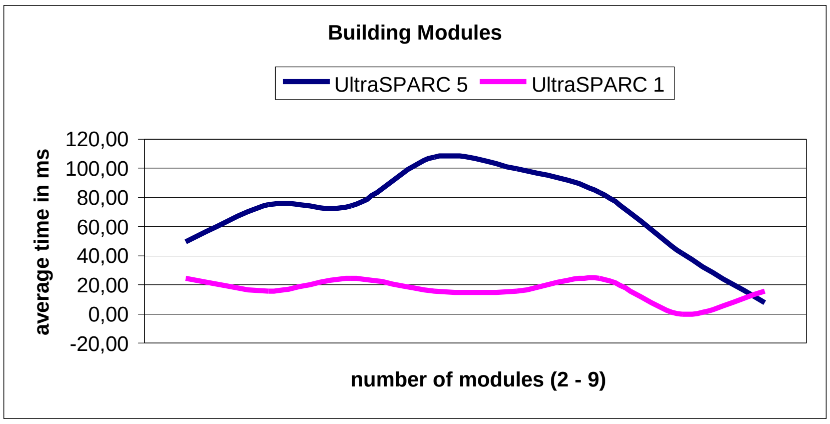
# modules	2	3	4	5	6	7	8	9
Stdev	12,29	9,70	13,80	12,08	41,28	47,09	42,93	42,87

The standard deviation calculated for passing 3k payload through the modules vary form 9.70 ms (3 modules) to 47.09 ms (7 modules). This indicates that VxSim is unpredictable and unstable when it comes to passing 3k payload through the modules.

The standard deviation is calculated from the time measured in the twenty tests run. In these cases the values are relative high, and there is quite a big gap between the values calculated for each module. An interesting observation is that the standard deviation does not seem to depend on the number of modules. This could indicate that the tests run for VxSim are being influenced by other factors like internal UNIX processes and other processes running on the machine.

UltraSPARC 5 and UltraSPARC 1 test results

The test was run only once, when it comes to building modules. This was done because the main focus was on the behaviour experienced when sending payload. This decision was made based on the tests run on VxSim earlier. They indicated that the number of modules being build did not have much influence on the results.



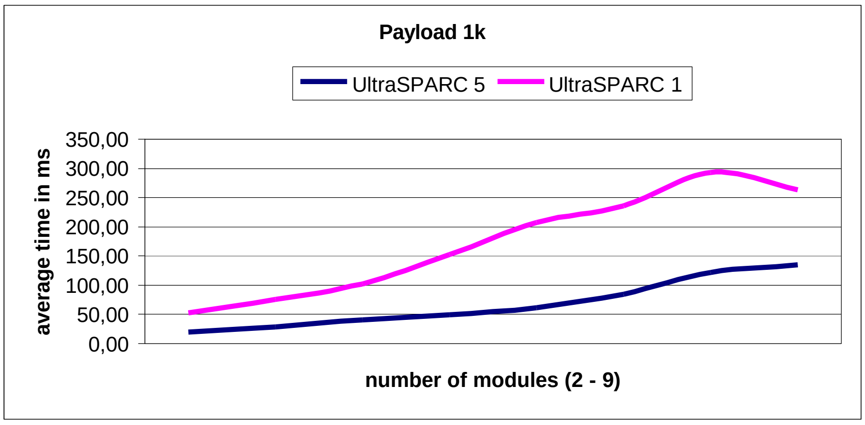
Graph 13 Build modules

Graph 13 illustrates the average time it takes to build from two till nine modules with both the UltraSPARC 5 and the UltraSPARC 1.

As Graph 13 illustrates, when it comes to building modules the UltraSPARC 1 is faster than the UltraSPARC 5. This is surprising, but it also supports earlier observations. The results are not influenced much by the number of modules being build or by the CPU. The time differs much from test to test, as for the previous tests. The time varies in the same range regardless of the number of modules being built.

This is mainly caused by the fact that building modules is a simple task that does not require to many resources.

When testing the payload the tests were run approximately ten times. This was done to accommodate earlier experiences. Those tests indicated that the payload and number of modules were important factors for the results.



Graph 14 Payload 1k

Graph 14 illustrates the average time it takes to pass a 1k payload through two to nine modules using both the UltraSPARC 5 and the UltraSPARC 1.

Table 13 Standard deviation, UltraSPARC 5

# modules	2	3	4	5	6	7	8	9
Stdev	7,87	7,95	12,38	8,82	12,02	25,96	39,76	35,11

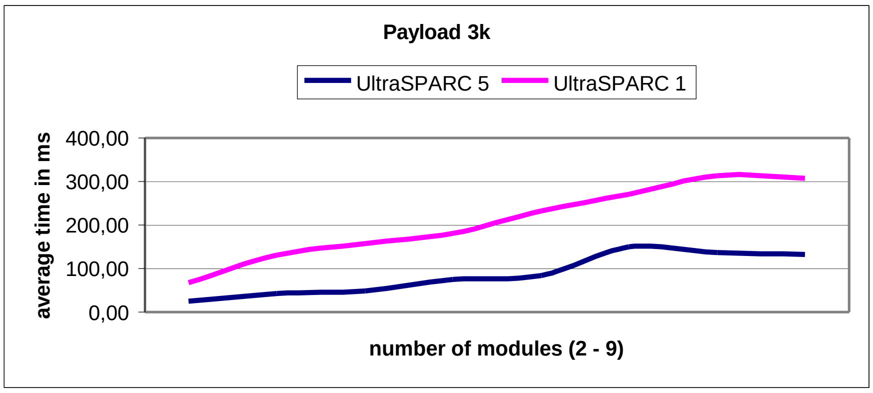
The standard deviation calculated for passing 1k payload through the modules varies form 7.85 ms (2 modules) to 39.76 ms (8 modules) for the UltraSPARC 5 (table 13).

Table 14 Standard deviation, UltraSPARC 1

# modules	2	3	4	5	6	7	8	9
Stdev	14,59	23,05	19,45	36,25	42,90	32,12	52,57	24,55

The standard deviation calculated for 1k payload through the modules vary form 14.59 ms (2 modules) to 52.57 ms (8 modules) for the UltraSPARC 1 (table 14).

The values vary considerably for both the UltraSPARC 5 and the UltraSPARC 1. This suggest that the CPU does not have much influence on the stability and reliability, when processing1k payload using VxSim. What also seems to be the case is that the UltraSPARC 5 is somewhat more stable and reliable than the UltraSPARC 1.



Graph 15 Payload 3k

Graph 15 illustrates the average time it takes to pass a 3k payload through two till nine modules for both the UltraSPARC 5 and the UltraSPARC 1.

Table 15 Standard deviation, UltraSPARC 5

# modules	2	3	4	5	6	7	8	9
Stdev	8,96	14,54	0,67	41,06	35,77	40,83	40,63	31,48

The standard deviation calculated for passing 3k payload through the modules vary form 0.67 ms (4 modules) to 40.83 ms (7 modules) for the UltraSPARC 5 (table 15).

Table 16 Standard deviation, UltraSPARC 1

# modules	2	3	4	5	6	7	8	9
Stdev	31,49	32,66	29,63	32,91	51,87	45,16	39,88	35,20

The standard deviation calculated for passing 3k payload through the modules vary form 31.49 ms (2 modules) to 51.87 ms (6 modules) for the UltraSPARC 5 (table 16).

The values vary considerably for both the UltraSPARC 5 and the UltraSPARC 1. This suggest that the CPU does not have much influence on the stability and reliability, when processing3k payload using VxSim. What also seems to be the case is that the UltraSPARC 5 is somewhat more stable and reliable than the UltraSPARC 1.

Chapter 5 Colloquy of Researched Work

Benefits of Hardware, VxWorks

VxWorks has also many advantages just as VxSim.

VxWorks is flexible, scalable, and available on all popular CPU platforms. It is not part of the bundled operating system and it is only seen when it is being used. It supports concurrent tasking, is multi-threaded and supplies all intertask messaging functions. VxWorks´ open design is highly portable and complete across all supported processors, allowing application migration between architectures with minimal effort. UNIX development environment is commonly used in Ericsson for software development. Therefore it is desirable to have a real-time operating system that interoperates nicely with the UNIX environment. VxWorks fulfils this requirement and also provides many of the UNIX C-library routines. UNIX systems and VxWorks systems can thus be used together in a hybrid application.

Since VxWorks is hardware, it is more accurate and reliable than any simulation tool, including VxSim. It has also available more features than VxSim since it is an operating system and not a simulation tool. This includes direct hardware interaction with device drivers and accessing files in a given file system.

When developing an application with VxWorks it is possible to take advantage of many available additional tools. Advanced debugging tool, CrossWind and WindView a graphical viewing and monitoring of multitasking application are just some of them. These tools are helpful when testing, timing, and debugging applications. Final product is small and stand-alone.

The existing network configuration allows many CPU’s to cooperate. It makes it also possible to connect a network that includes VxWorks nodes with other networks in a complex configuration. VxWorks integrates industry-standard TCP/IP networking facilities, which is a very useful in datacom application development.

Applications can be made portable by using the VxWorks POSIX API that makes the applications portable across a wide range of computer systems. POSIX is commonly used feature for software development in Ericsson.

Even though VxWorks has a lot of advantages, there are some disadvantages too.

VxWorks is expensive to purchase and assumes that the hardware is already available. This means long delays before testing can start.

In order to reboot VxWorks the developers would have to wait for the hardware to restart the VxWorks image. This has a negative effect on the modern in the development face of the project.

It is also a limitation attended with the number of developers who can test simultaneously. This might be a problem when there are many developers and the project is approaching the deadline.

Benefits of Simulation Tool, VxSim

There are both benefits and drawbacks attended with use of a simulation tool.

Any application should be portable between VxSim and VxWorks with minimal hardware interaction due to the VxWorks scheduler. This means that the code written for VxWorks can be reused for VxSim. This is a very important feature for Ericsson since the amount of their source code is considerable.

Since VxSim is a simulation tool it enables early software testing

VxSim is a complete prototyping and simulation tool for VxWorks applications. This enables testing before hardware is available and it allows testing early in the development cycle. Development organizations will also benefit from not having to wait for target hardware to become available.

Since Ericsson is a typical development organization it will profit from use of VxSim.

Organizations have a lot to gain by using VxSim instead or in addition to VxWorks both financially and modern.

VxSim is not as expensive as the hardware, which is important for organizations like Ericsson. This also means lower budgets for software development projects.

Because VxSim is a simulation tool and runs on the local host machine it is fast to reboot and reload the environment. This would save the organization a lot of time, seeing that the developers do not have to wait too long for the target to be accessible again. This is especially a problem when the binaries are large.

The simulation tool makes it possible for everybody to run tests simultaneously regardless.

Another important feature is its user interface. Developers familiar with VxWorks interface will have no problem changing to VxSim, since the user interfaces are very similar.

Even though VxSim has a lot of advantages, there are some disadvantages too.

One of them is the lack of target hardware interaction. Therefore device driver development may not be suitable for simulation.

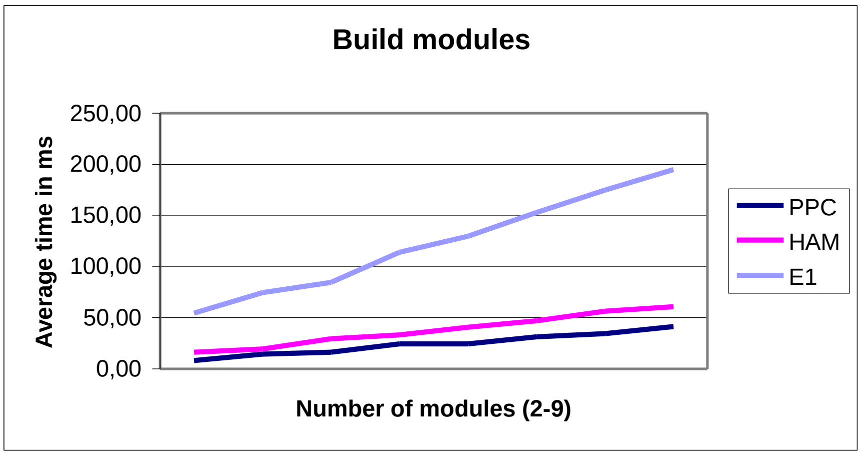
There is no direct network connection for VxSim thus it simulates network traffic by the use of internal loopback IP addresses. There are only 16 reserved IP addresses for VxSim (127.0.1.0 – 15). This means that the software has to be rebooted when the addresses are all been used. In order to restart the software one has to be logged on as root on the local host. This implies several problems for the developers.

VxSim is not suitable for debugging as VxWorks since CrossWind feature is not supported. This also applies for WindView. This might be caused by the fact that only a evaluation version of VxSim was used.

Another problem is the missing synchronization between WindSh and target. This means that the work has to be done in the same shell, the shell used to load the environment. This may cause problems for UNIX developers who are used to use many shells.

Discussion of VxWorks Test Results

As expected, E1 is slowest and PPC is fastest and gives the best results. The difference between measured values for PPC and HAM are small, while corresponding values for E1 differ a lot, as illustrated in Graph 16.

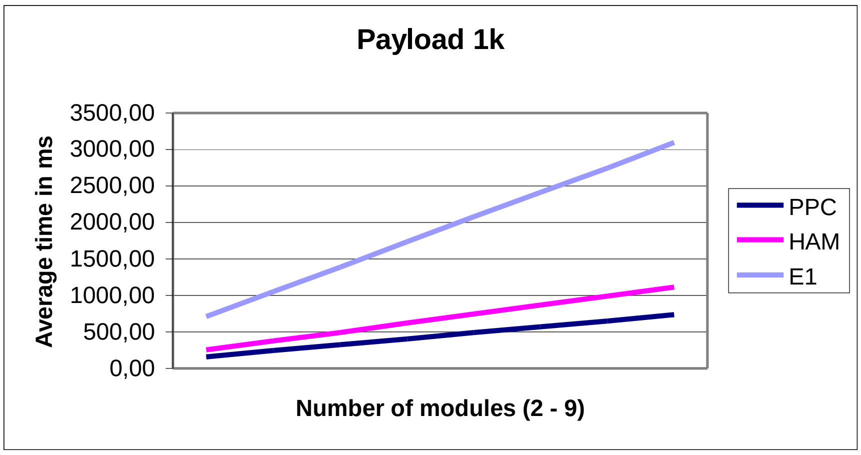


Graph 16 Build modules (PPC, HAM and E1)

The difference it takes to add one more module to STREAMS environment increases strongly for the E1 processor. The same difference for HAM and PPC processors is quite small, e.g. to add a module to STREAMS will not have a major effect on efficiency. Thus there is no general rule for how time values change and increase and they can also be equal in several test cases.

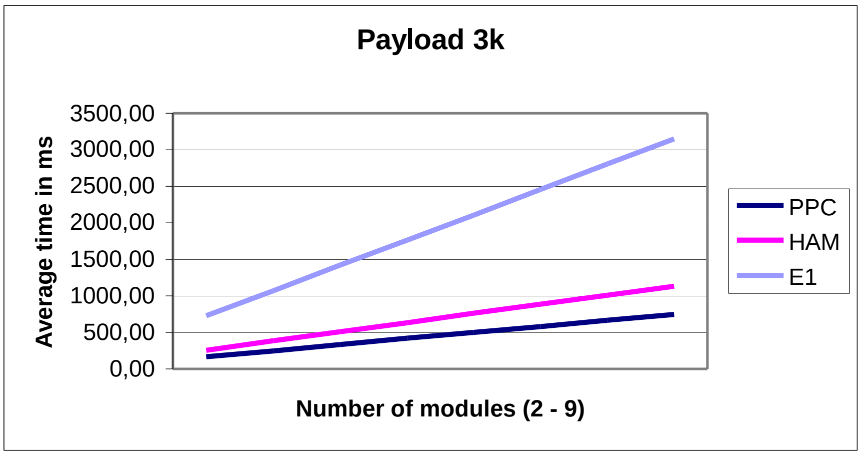
All processors have in common that there is no linear dependence between number of modules and time it takes to create and install them.

As one can see from test results, the payload size has not a major influence on time it takes to process data in modules. The test proved that there is no significant difference in processing 1k and 3k payload, although it takes more time to process 3k payload.



Graph 17 Payload 1k (PPC, HAM and E1)

Graph 17 illustrates measured time values for PPC, HAM and E1 when 1k payload is processed in modules. PPC is clearly the best to process the payload regardless of number of modules in STREAMS. E1 is clearly the slowest processor.



Graph 18 Payload 3k (PPC, HAM and E1)

Graph 18 illustrates measured time values for PPC, HAM and E1 when 3k payload is processed in modules. PPC is still the best to process the payload regardless of its size. The difference in the performance of E1 vs. PPC and HAM increases strongly when payload is larger. E1 is much slower when there are more modules in STREAMS compared to PPC and HAM. Adding an extra module to STREAMS makes a larger difference for E1 than for both PPC and HAM.

The delay is mainly caused by memory allocation when linked list is created. This is a major problem for E1, for which it takes as long as three seconds to process the payload, which is an “eternity” for telecom applications.

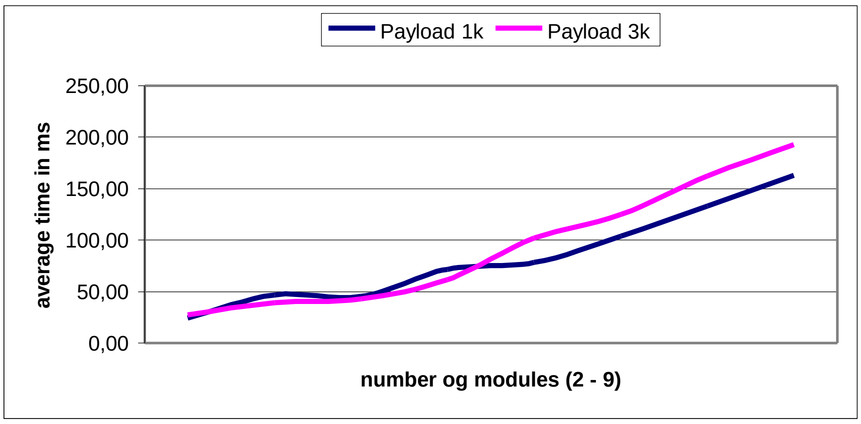
Discussion of VxSim Test Results

As mentioned earlier, the number of modules being build does not seem to have much influence on the results.

It is impossible to give an exact answer to this behaviour. Most likely there are several factors that cause this behaviour.

VxSim is run on the local machine. Thus the time would most likely depend on the following factors:

?? the machines CPU and memory
?? other processes running on the machine
?? the timer used to measure time, and its correctness
?? the tuning of the machine



Graph 19 Payload, both 1k and 3k

Graph 19 illustrates the result from both 1k and 3k payload. As the graph illustrates, it is apparent that the payload does play a role when it comes to the efficiency of VxSim. As expected, the results indicate that the time used increases with the payload and the number of modules. The difference might have been expected to be bigger. There are several reasons why the difference is not larger. Some of them could be explained from the fact that the tests were run just ten times, a powerful CPU, and processing of payload is done very fast, so the time used mostly depends on the making of the linked lists. Others are harder to explain and could come from the fact that there are different processes running on the machine during the testing, although this factor was kept to a minimum.

UltraSPARC 5 vs. UltraSPARC 1

The tests illustrates that VxSim on both the UltraSPARC 5 and the UltraSPARC 1 does not demand much CPU or memory resources.

When run on the UltraSPARC 1 VxSim demands less than 2.0% CPU resources when building modules, less than 6.6% CPU resources when processing 1k payload, and less than 6.5% CPU resources when processing 3k payload.

When run on the UltraSPARC 5 VxSim demands less than 1.5% CPU resources when building modules, less than 2.6% CPU resources when processing 1k payload, and less than 2.0% CPU resources when processing 3k payload.

The memory usage was between 5MB and 6MB for both CPU's.
Other test results, discussed earlier, indicate that VxSim is faster when run on the UltrSPARC 5 than on the UltraSPARC 1, for both 1k and 3k payload. When it comes to building modules it is hard to draw a conclusion based on the test results, due to the lack of more tests.

These results are as expected, seeing that VxSim is run on the local host.

Chapter 6 Discussion of VxWorks and VxSim

All source code is written for VxWorks. Thus it is desirable to use a simulation tool, which is portable with minimal hardware interaction. That is why VxSim is well suited for Ericsson's purposes.

It is very important to start software testing early in the development process. VxSim is a very powerful tool when it comes to testing at these early stages. On more important feature is that testing can start before the needed hardware is available.

VxSim and VxWorks user interfaces are similar, meaning that the developers would not experience major problems converting to VxSim.

Among the disadvantages with VxSim, it is important to mention the lack of target hardware interaction. Thus device driver development may not be supported.

VxSim has no direct network connection, thus it simulates the network traffic. Developer will experience different problems regarding addresses available for simulation.

Perhaps the biggest disadvantage of VxSim is that it runs on Solaris OS, which is not a RTOS. Since Ericsson develops telecom software, the use of VxSim is limited.

VxSim does not support features like CrossWind and WindView. This can cause some problems during development and might have a negative impact on the development process. The missing synchronization between WindSh and target may also cause problems for developers.

On the other hand there is VxWorks. VxWorks is a flexible and scalable real-time operating system. It interoperates with UNIX environment and these operating systems can be used to develop hybrid applications.

VxWorks gives more accurate and reliable results because it supports direct hardware interaction. It is also suitable for device driver development. CrossWind and WindView features are supported with VxWorks, and this makes VxWorks a very powerful development tool.

VxWorks allows many CPU's to cooperate due to its portable network configuration. It also integrates standard TCP/IP networking facilities which is an important feature for Ericsson.

It may seem like VxWorks is the best choice, but still that is not always the case.

VxWorks is more expensive than a simulation tool like VxSim. It is only usable when hardware is available, which might imply some delay in the testing process.

The test results clearly show the behaviour of both VxWorks and VxSim. When comparing these results, they indicate that VxSim is faster, but much more unreliable than VxWorks.

When it comes to building modules VxWorks and the PPC gives the best results, in addition to being the most reliable DP. VxSim run on the UltraSPARC 5 is fast but very unreliable, since the measured time varies considerably. HAM also shows better test results than VxSim, and as PPC it is more reliable and stable.

The results regarding the processing of 1k payload indicate that VxSim is by far the fastest, run on both the UltraSPARC 5 and the UltraSPARC 1. It is not as unreliable when it comes to processing payload as it is when building modules. VxWorks and the PPC is not as fast as VxSim, but it is more stabile and reliable.

The results regarding the processing of 3k payload shows the same tendency as when 1k payload is processed. VxSim is still the fastest and VxWorks is the most reliable and stable one.

The DP’s with VxWorks are a limited resource. Therefore the number of developers who can test simultaneously are very limited. The use of simulation tools allows developers to test whenever it is required regardless of the number of developers already using it. This would save the organization a lot of time, seeing that the developers do not have to wait too long for the target to be accessible again. This is especially a problem when the binaries are large.

Chapter 7 Further Study

The solution suggested by this thesis is not optimal, since the actually GPRS software was not tested. Instead the features of VxWorks and VxSim were tested and compared, which gives an indication of the behaviour of the respective tools.

There is noting to gain by develop the testing environment described in this thesis further, since the goal is to simulate the GPRS software.

Thus the best way to continue this study would be to use Ericsson's testing environment. This would give a better knowledge of the behaviour of both VxWorks and VxSim when developing GPRS software.

In order to continue this study, using Ericsson's testing environment, it is necessary to port the existing source code to VxSim. A solution to how this could be done is suggested in chapter 3, Problem Solving.

The test used in this thesis does not cover all aspects connected to GPRS software development. These aspects include; large number of subscribers, priority and quality of service. These can only be tested thoroughly using Ericsson's testing environment.

Chapter 8 Conclusion

A typical development organization like Ericsson has a lot to gain by using a simulation tool like VxSim both financially and modern.

All in all VxSim is a fast and less expensive simulation tool. Still it may not be regarded the only alternative due to underlying OS and unreliability. On the other hand VxWorks is very reliable and stabile RTOS. Although VxSim is the fastest VxWorks is close behind. This implies for both PPC and HAM.

Another aspect is the cost of hardware vs. the costs related to software. Hardware is very expensive and development organizations like Ericsson have a lot to gain by using a simulation tool in addition to hardware. This could mean lower software development budgets.

It is important to choose a development tool according to the kind of development being done. If direct hardware access is required, as in case of device driver development, VxWorks is the best alternative.

As indicated in this report, VxSim is well suited for testing in the early stages of the project. VxSim could be used during the whole project, although it would not be recommended, seeing that VxSim runs on Solaris, which is not a real-time operating system. This means that VxSim can only be used in addition to VxWorks.

Chapter 9 References

[1] Wind River Systems, Inc (1997). *VxWorks Programmer's Guide, 5.3.1, Edition 1*. Alameda, CA: Wind River Systems, Inc.
[2] Wind River Systems, Inc (1998). *Tornado Training Workshop*. Alameda, CA: Wind River Systems, Inc.
[3] Wind River Systems, Inc (1997). *VxWorks Reference Manual, 5.3.1, Edition 1*. Alameda, CA: Wind River Systems, Inc.
[4] Wind River Systems, Inc (1998). *VxSim User's Guide, 5.3.1, Edition 1*. Alameda, CA: Wind River Systems, Inc.
[5] UNIX System Laboratories (1992). *UNIX system V Release 4: Programmer's Guide: STREAMS*. Englewood Cliffs, New Jersey: Prentice-Hall
[6] Motorola. *PowerPC 604 microprocessor* [Online]. Web-address: http://www.mot.com/SPS/PowerPC/library/fact_sheet/604.html [28. May 2000]
[7] HM Computing Ltd. *PMC Modules* [Online]. Web-address: http://www.hmcomp.demon.co.uk/pmc.htm [28. May 2000]
[8] Motorola. *PowerPC 603 Microprocessor* [Online]. Web-address: http://www.mot.com/SPS/PowerPC/library/fact_sheet/603.html [28. May 2000]
[9] mobileGPRS.com. *All about General Packet Radio Service (GPRS) on mobile networks* [Online]. Web-address: http://www.mobilegprs.com [29 May 2000]
[10] GNU make. *A Program for Directing Recompilation* [Online]. Web-address: http://www.jcu.edu.au/docs/gnu/make_toc.html [29. May 2000]

Appendix A: Code

a. UDP Environment

i. udpClient.c

```
/* UDP client program */

/* Code to be executed by client. Writes request/message into
socket and optionally receives a reply back. */

/* --- UNIX header files --- */
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/time.h>
#include <arpa/inet.h>
#include <strings.h>
#include <fcntl.h>
#include <sys/errno.h>

/* --- Project headers --- */
#include "client.h"

/* Global declarations */
int sock_fd; /* Socket file descriptor */
struct request testReq; /* Request to send */
struct msg_data msg;    /* Message/payload to send */
struct sockaddr_in serverAddr;
int msgLen;             /* Length of message to send */

/* Forward declarations */
void sendRequest (char *fileName);
int awaitReply (int sock, long timeout, char *buf, int size);

int main (int argc, char *argv[]){

  char ch; /* Want reply? 'y'/'n' */
  char replyBuf [REPLY_SIZE_MAX]; /* Space for server */
  char *fileName; /* Name of the file to be sent. */
  unsigned short port = SERVER_PORT_DEFAULT; /* Server's port
number */
  unsigned long serverIP; /*Server's IP address * /

  fileName = (char *)malloc (sizeof(char *));

  /* Get the server ip address, port number and file to send.
     Write a message if all parameters are not written. */

  if((argc < 3) || (argc > 4)){
    printf("Usage: %s IPaddr [port] filename.\n", argv[0]);
    exit(1);
  }
  if((serverIP = inet_addr(argv[1])) == ERROR){
    printf("UDP client: invalid ip address.\n");
    exit(1);
  }

  if(argc == 3)
```

```
    port = (unsigned short) atoi(argv[2]);

  if(argc == 4)
    fileName = argv[3];

    /* create socket */
    if((sock_fd = socket(PF_INET, SOCK_DGRAM, 0)) == ERROR){
      perror("Error creating socket.\n");
      exit(ERROR);
    }
    /* build server socket address */
    bzero(&serverAddr, sizeof(struct sockaddr));
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(port);
    serverAddr.sin_addr.s_addr = serverIP;

    /* file to send */
    printf("File to send:\t%s \n", fileName);

    /* check for reply */
    printf("Would you like a reply (Y/N):  \n");
    scanf("%c", &ch);

    switch(ch){
    case 'Y':
    case 'y':
      testReq.reply = TRUE;
      break;

    default:
      testReq.reply = FALSE;
      break;
    }

    /* send file to server */
    sendRequest(fileName);

    /* if expecting reply, read and print it */
    if(testReq.reply){
      int tries;
      int timeout = TIMEOUT_CONST;
      for(tries=0; tries<MAX_TRIES; tries++){
      if(tries > 0){
        printf("Retransimtting ... \n");
        sendRequest(fileName);
      }
      if(awaitReply(sock_fd, timeout, replyBuf, REPLY_SIZE_MAX)>0)
        break;
      timeout *= 2;
      }
      if(tries < MAX_TRIES)
      printf("Reply from server: \n %s\n", replyBuf);
      else{
      fprintf(stderr, "UDP client: timeout.\n");
      close(sock_fd);
      exit(ERROR);
      }
    }
    close(sock_fd);
}
```

```
/*****************************************************************

Routine: sendRequest

Description: send file to server

*****************************************************************/
void sendRequest (char *fileName){
  int fileFd; /* File descriptor */
  int j;      /* Counter */
  int ret;    /* Error indicator during seding */

#ifdef DEBUG
  printf("Opening file %s...\n", fileName);
#endif

  /* opening file to send */
  if((fileFd = open(fileName, O_RDWR,644)) < 0){
    printf("Open failed %s.\n", fileName);
    return;
  }

#ifdef DEBUG
  printf("File opened.\n");
  printf("Sending file...\n");
#endif;

  memset(&msg.msg,0,DATA_SIZE);

  /* "making" datapacket to send */
  while((msgLen = read(fileFd, msg.msg, DATA_SIZE-1)) > 0){
    printf("MsgLen: %d\n", msgLen);
    msg.msg[msgLen+1] = '\0';
    testReq.size_data = sizeof(struct msg_data);
    /*last packet in sequence */
    if((msgLen+1) < DATA_SIZE){
      testReq.segm = 0;
      for(j=msgLen+2; j<DATA_SIZE; j++) msg.msg[j] = '\0';
    }
    /* more packets in sequence */
    else testReq.segm = 1;

    testReq.message = msg;

#ifdef DEBUG
    printf("Message: %s\n", testReq.message.msg);
    printf("Seg: %d\n", testReq.segm);
    printf("Size: %d\n", testReq.size_data);
#endif

    /* send packet 'testReq', size 'msgLen+12', to server at
serverAddr via
       socket 'sock_fd' */
    ret = sendto(sock_fd,(caddr_t)&testReq,msgLen+12,0,(struct
sockaddr *)&serverAddr,
             sizeof(struct sockaddr));
#ifdef DEBUG
    printf ("sendto: %d\n", ret);
#endif
    if (ret == ERROR){
      perror("Sendto");
      close(sock_fd);
      exit(ERROR);
    }
  }

#ifdef DEBUG
  printf("Closing file ...\n");
  printf("Seg: %d\n", testReq.segm);
#endif

  /* close file after sending last packet */
  if(close(fileFd) == ERROR ){
    printf("Error while closing file: %s.\n", fileName);
    return;
  }

#ifdef DEBUG
  printf("File closed.\n");
#endif
}

/*****************************************************************

Routine: awaitReply

Description: wait for reply with a timeout

*****************************************************************/
int awaitReply (
            int sock,     /* socket file descriptor */
            long timeout, /* in 1/100 s units */
            char *buf,    /* where to put reply */
            int size      /*maximum # bytes to receive */
            )
{
  struct sockaddr_in sAddr;
  int addrSize = sizeof(struct sockaddr);
  struct fd_set readFds;
  struct timeval tval;
  int nBytes;

  FD_ZERO(&readFds);
  FD_SET(sock,&readFds);

  tval.tv_sec = timeout/100;
  tval.tv_usec = (timeout%100)*10000;

  if(select(sock+1,&readFds,NULL,NULL,&tval) != 1)
    return(ERROR);

  nBytes = recvfrom(sock,buf,size,0,(struct sockaddr
*)&sAddr,&addrSize);

  /* make sure the reply came from our server! */
  if ((sAddr.sin_addr.s_addr != serverAddr.sin_addr.s_addr) ||
    (sAddr.sin_port != serverAddr.sin_port))
    return ERROR;

  return nBytes;
}
```

ii. udpServer.c

```
/* Code of UDP server
   Server reads from local socket and displays client's message.
   If requested, server sends a reply back to t he client.
*/

/* --- VxWorks header files --- */
#include "vxWorks.h"
#include "sockLib.h"
#include "netinet/in.h"
#include "inetLib.h"
#include "string.h"
#include "stdio.h"
#include "stdlib.h"
#include "ioLib.h"

/* --- Project headers --- */
#include "udpHeader.h"
#include "strm_decl.h"
#include "timer.c"

/* Global declarations */

int fd;  /* File descriptor */
char writeFile[]  = "/Core/coreUser/etoxaia/serverfile"; /*file to
write to */
int opened = FALSE; /* FALSE=file not opened */

extern int retfd0, retfd1; /*STREAMS file descriptors */


/*****************************************************************

Routine: udpServer

Description: reads from local socket, sends them to STREAMS
             and writes processed data to a file

*****************************************************************/
extern void udpServer (int port){

  struct sockaddr_in testAddr;   /* Server's IP address */
  struct sockaddr_in clientAddr; /* Client's IP address */
  struct request clientRequest;  /* Data packe t to receive */
  int sockFd;                    /* Server's socket */
  u_short clientPort;            /* Client's port (native) */
  char inetAddr[INET_ADDR_LEN];  /* Buffer for client ascii IP
address */
  char reply[32];                /* Size of reply  message */
  int seg = 1;
  char* temp_msg;

  if(port == 0)
    port = SERVER_PORT_DEFAULT;

  /* create socket */
  if((sockFd = socket(PF_INET, SOCK_DGRAM, 0)) == ERROR){
    perror("socket");
    exit(ERROR);
  }

  /* build socket address */
  bzero((char *)&testAddr, sizeof(struct sockaddr_in));
  testAddr.sin_family = AF_INET;
  testAddr.sin_port = htons(port);
  testAddr.sin_addr.s_addr = INADDR_ANY;

  /* bind socket to local address */
  if(bind(sockFd, (struct sockaddr *)&testAdd r,
       sizeof(struct sockaddr)) == ERROR){
    perror("bind");
    close(sockFd);
    exit(ERROR);
  }


  FOREVER{
    int sockAddrSize = sizeof(struct sockaddr);
    int ret;

    printf("UDP server is alive\n");

#ifdef DEBUG
    printf("UDP server is alive\n");
#endif

    do{
      /* initialize memory buffer */
      memset(&clientRequest.message.msg,0,DATA_SIZE);
      /* packet arriving on sockFd */
      ret = recvfrom(sockFd,(char *)&clientRequest, sizeof(struct
msg_data)+12,0,
            (struct sockaddr *)&clientAddr, &sockAddrSize);

#ifdef DEBUG
    printf ("recvfrom: %d\n", ret);
#endif
    if (ret == ERROR){
      perror("recvfrom");
      close(sockFd);
      exit(ERROR);
    }

    seg = clientRequest.segm;

    if(opened == FALSE){

#ifdef DEBUG
      printf("Deleting file ...\n");
#endif

      /* remove file before writing new data to it */
      if(remove(writeFile) != OK){
      printf("Delete failed %s.\n", writeFile);
      perror("remove ");
      }

#ifdef DEBUG
      printf("Opening file ...\n");
#endif
```

```
      /* open file for writing */
      if((fd = open(writeFile, O_CREAT | O_RDWR, 0777)) < 0){
        printf("Open failed %s.\n", writeFile);
        perror("open ");
      }
      opened = TRUE;

#ifdef DEBUG
      printf("File opened.\n");
#endif

    }

#ifdef DEBUG
    printf("Seg =\t\t%d\n", seg);
    printf("Message:%s\n", clientRequest.message.msg);
    printf("Writing to file!\n");
#endif

#ifdef DEBUG
 printf("Starting timer (send message).\n");
#endif

 /* start timer */
  startTimer();

  /* send data to STREAMS for processing */
  if(send_msg(retfd1, (char *)&clientRequest.message) < 0){
    printf("Error while sending message to streams.\n");
    perror("send_msg ");
  }

  /* receive the modified message from STREAMS */
  temp_msg = recv_msg(retfd0, DATA_SIZE+364);

  if(!strcmp(temp_msg, "error")){
    printf("temp_msg: \t\t%s\n", temp_msg);

#ifdef DEBUG
 printf("temp_msg: \t\t%s\n", temp_msg);
#endif
      printf("Error while receiving message from streams.\n");
      perror("recv_msg ");
    }

    clientRequest.size_data = strlen(temp_msg);
    /*write message 'temp_msg' with data size
'clientRequest.size_data' to
      file with file descriptor 'fd' */
    write(fd, temp_msg, clientRequest.size_data);

    /* stop timer and display measured values */
    stopTimer();

#ifdef DEBUG
 printf("Timer stopped (receive message).\n");
#endif

    }while(clientRequest.segm == 1);

    /* if last packet in sequence, close file */
    if(opened == TRUE && clientRequest.segm == 0){

#ifdef DEBUG
    printf("Closing file ...\n");
#endif

      if(close(fd) == ERROR ){
      printf("Error while closing file: %s.\n", writeFile);
      perror("close ");
      }
      else opened = FALSE;

#ifdef DEBUG
    printf("File closed.\n");
#endif

    }

    /* convert IP address to ascii for display */
    inet_ntoa_b(clientAddr.sin_addr, inetAddr);
    clientPort = ntohs(clientAddr.sin_port);

    printf("File received from: Internet address %s,""port
%d\nwrote to:%s\n",
         inetAddr, clientPort, writeFile);

    /* if client request a reply, send one */
    if(clientRequest.reply){
      strcpy(reply, "Server received your message.\n");
      if(sendto(sockFd, reply, strlen(reply)+1,0,(struct sockaddr
*)
            &clientAddr, sockAddrSize) == ERROR){
      perror("sendto");
      close(sockFd);
      exit(ERROR);
      }
      memset(reply, 0, 32);
    }

  }
  close(sockFd); /* never get here */
}
```

iii. udpHeader.h

```
/* Header file for UDP environment (UDP server and UDP client).
   Defines common information for both server and client
*/

#ifndef _UDPHEADER_H
#define _UDPHEADER_H

#define SERVER_PORT_DEFAULT (5001) /* default port for server */
#define MSG_SIZE (1200)   /* packet size */
#define DATA_SIZE (1024)  /* payload size */
/*#define FILE_NAME (256)*/

/* Structure used for message payload */
struct msg_data{
  char msg[DATA_SIZE];
};

/* Structure used for request/message from client */
struct request {
  int reply;     /* Reply to client request */
  int size_data; /* Size of payload */
  int segm;      /* 0=last packet in squence; 1=more packets */
  struct msg_data message; /* Payload data */
};

/* Size of reply from server */
#define REPLY_SIZE_MAX (500)

#define FOREVER for (;;)

/* Function declarations */
extern void udpServer (int port);
extern void udpServer1 (int port);

#endif
```

iv. client.h

```
/* UDP client header file */
/* Client executes on UNIX side. */
/* Initial timeout waiting for reply: 1s. */

#include "udpHeader.h"

#define TIMEOUT_CONST (100)
#define MAX_TRIES (7)

#define ERROR (-1)
#define TRUE  (1)
#define FALSE (0)
```

v. VxServer.c

```
/* Starts vxWorks UDP server.
   Code executes on VxWorks target.
*/

#ifdef VXWORKS

/* --- VxWorks header files --- */
#include "vxWorks.h"
#include "sockLib.h"
#include "taskLib.h"
#include "netinet/in.h"
#include "inetLib.h"
#include "string.h"
#include "stdio.h"
#include "stdlib.h"
#include "ioLib.h"

/* --- Project headers --- */
#include "udpHeader.h"

int tvxServ;

void vxServer (int port_num){
  /* creates and activates task 'vxServer' with priority '59',
stack size
     '10000' and entry point of function to execute */
  tvxServ = taskSpawn("vxServer", 59, 0, 10000, (void *)udpServer,
port_num, 0,
                  0,0,0,0,0,0,0,0);

}

/*void vxServer1 (int port_num){

  taskSpawn("vxServer1", 59, 0, 5000, (void *)udpServer1,
port_num, 0,0,0,0,
          0,0,0,0,0);

}*/
#endif
```

b. STREAMS Environment

i. strm_mod.c

```
/* This file contains the source code for test module */

/* --- System headers --- */
#include <sys/types.h>
#include <sys/param.h> /* contains definitions for NULL */
#include <sys/stream.h>
#include <stdio.h>
#include <fcntl.h>


#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/time.h>
#include <arpa/inet.h>
#include <sys/errno.h>

#ifdef VXWORKS
#include <string.h>
#elif SUNOS5
#include <strings.h>
#endif

/* --- Project headers --- */
#include "strm_decl.h"
#include "udpHeader.h"
#include "list.h"

/* Module declarations */
#define TEST_MOD_STID TEST_MOD_ID1 /* Module id number */
#define TEST_MOD_NAME "testmod" /* Module name */
#define TEST_MOD_HI_WAT 0x600  /* High water mark */
#define TEST_MOD_LO_WAT 0x300  /* Low water mark */

static int testmod_open (queue_t *queue_p, int devp, int oflag,
int sflag);
static int testmod_close (queue_t *queue_p, int flag);

static int testmod_wput (queue_t *queue_p, mblk_t *msg_p);
static int testmod_rput (queue_t *queue_p, mblk_t *msg_p);

static void count_char(queue_t *queue_p, mblk_t *msg_p);
static void list(queue_t *queue_p, mblk_t *msg_p);

static struct module_info testmod_info=
{
  TEST_MOD_STID,  /* Module id number */
  TEST_MOD_NAME,  /* Module name */
  0,              /* min packet size */
  INFPSZ,         /* max packet size */
  TEST_MOD_HI_WAT,/* High water mark */
  TEST_MOD_LO_WAT /* Low water mark */
};
```

```
/* Read queue structure */
static struct qinit TestmodRInit =
{
  &testmod_rput,
  NULL,
  &testmod_open,
  &testmod_close,
  NULL,
  &testmod_info,
  NULL
};

/* Write queue structure */
static struct qinit TestmodWInit =
{
  &testmod_wput,
  NULL,
  NULL,
  NULL,
  NULL,
  &testmod_info,
  NULL
};

/* testdrvinfo structure holds the qinit structures for the driver
*/
struct streamtab testmodinfo =
{
  &TestmodRInit, /* Points at qinit structure for the read queue
*/
  &TestmodWInit, /* Points at qinit structure for the write que ue
*/
  NULL,
  NULL
};

queue_t *writeq_gp;

int module_nr = 1; /* module number identifier*/


/*******************************************************************

Routine: testmod_open

Description: This routine is called to open the TEST_MOD Module
             in a Stream

*******************************************************************/
static int testmod_open (queue_t *queue_p, /* read queue */
                   int devp,  /* device number at the Stream end
*/
                   int oflag, /* no meaning for modules */
                   int sflag  /* indicates kind of open */
                   )
{

#ifdef DEBUG
 printf("Now we're inside testmod_open.\n");
#endif
```

```
 if(sflag != MODOPEN){
   printf("Configuration error when trying to open module.\n");
   return(EINVAL);
 }
 if(queue_p->q_ptr != NULL)
   return (FUNC_SUCCESS); /* already opened before, so just return
OK */

 /* put module number in module's "private" data space */
 queue_p->q_ptr = malloc(sizeof(int));
 *(queue_p->q_ptr) = module_nr;
 /* enable the put and service routines of the module */
 WR(queue_p)->q_ptr = queue_p->q_ptr;
 writeq_gp = WR(queue_p);

 printf("Module nr \t\t%d\n", module_nr);

#ifdef DEBUG
 printf("Module nr \t\t%d\n", module_nr);
#endif

 module_nr++;

 return (FUNC_SUCCESS);
}


/*******************************************  ********************

Routine: testmod_close

Description: This routine is called to close the TEST_MOD Module
             in a Stream

*****************************************************************/
static int testmod_close (queue_t *queue_p, int flag ){
#ifdef DEBUG
 printf("Now we're inside testmod_close.\n");
#endif

 queue_p->q_ptr = NULL;
 WR(queue_p)->q_ptr = NULL;

 return (FUNC_SUCCESS);
}


/*****************************************************************

Routine: testmod_wput

Description: This routine is called to process messages arrived at
             the write queue of the TEST_MOD Module

*****************************************************************/
static int testmod_wput (queue_t *queue_p, /* pointer to write
queue */
                   mblk_t *msg_p  /* pointer to the arrived
message */
                   )
{
```

```
#ifdef DEBUG
 printf("Now we're inside testmod_wput.\n");
#endif
 switch(msg_p->b_datap->db_type){
 case M_PROTO:
   /* initialise linked list in the module */
   list(queue_p, msg_p);
   /* process the message */
   count_char(queue_p, msg_p);
   break;

 case M_IOCTL:
   putnext(queue_p, msg_p);
   break;

 default:
   /* pass on message since STREAMS message type is not recognised
*/
   putnext(queue_p, msg_p);
   break;
 }

 return (FUNC_SUCCESS);
}


/******************************************************************

Routine: testmod_rput

Description: This routine is called to process messages arrived at
             the read queue of the TEST_MOD Module

******************************************************************/
static int testmod_rput (queue_t *queue_p, /* pointer to read
queue */
                   mblk_t *msg_p  /* pointer to the arrived
message */
                   )
{

#ifdef DEBUG
 printf("Now we're inside testmod_rput.\n");
#endif

 switch(msg_p->b_datap->db_type){
 case M_PROTO:
   putnext(queue_p, msg_p);
   break;

 case M_IOCTL:
   putnext(queue_p, msg_p);
   break;

 default:
   /* pass on message since STREAMS message type is not recognised
*/
   putnext(queue_p, msg_p);
   break;
 }
 return (FUNC_SUCCESS);
}


/*******************************************************************

Routine: count_char

Description: This routine is called to process messages arrived at
             the write queue of the TEST_MOD Module

*******************************************************************/
static void count_char(queue_t *queue_p, /* pointer to write queue
*/
                   mblk_t *msg_p  /* pointer to the arrived
message */
                   )
{

  unsigned int nc=0;     /* number of characters in the message */
  unsigned int na=0;     /* number of 'a'/'A' in the message */
  int s_mod;             /* message size */
  mblk_t *header_buf;    /* new header of the message */
  unsigned char *h_rptr; /* read pointer of the message data */
  unsigned char *h_wptr; /* write pointer of the message data */
  char s1[10];           /* header data */
  int tmp, digit, len, t;
  int cont;              /* cont=1->put data in header */

#ifdef DEBUG
 printf("Now we're inside count_char.\n");
 printf("Module number \t\t%d\n", *queue_p->q_ptr);
#endif
 /* which module number */
 switch(*queue_p->q_ptr){

 case 1:
   nc = 0;
   /* count characters in message */
   h_rptr = msg_p->b_cont->b_cont->b_rptr;
   nc = strlen(h_rptr);

   if(nc>9) digit = 1;
   else digit = 0;

    printf("Number of characters:\t\t\t%d\n", nc);
    cont = 1;

#ifdef DEBUG
 printf("Number of characters:\t\t\t%d\n", nc);
 printf("The message:\t\t%s\n", h_rptr);
#endif
 break;

 case 2:
   na = 0;
   /* count character 'a'/'A' in message */
   h_rptr = msg_p->b_cont->b_rptr;
   h_wptr = msg_p->b_cont->b_wptr;
   while(h_rptr != h_wptr){
```

```
#ifdef DEBUG
 printf("Checking the character %c\n", h_rptr[0]);
#endif
 if((h_rptr[0] == 'A') || (h_rptr[0] == 'a')) na++;

 h_rptr = h_rptr + 1;
   }

   if(na>9) digit = 1;
   else digit = 0;

   printf("Number of A/a's:\t\t\t%d\n", na);
   cont = 1;

#ifdef DEBUG
 printf("Number of A/a's:\t\t\t%d\n", na);
#endif
 break;

 default:
   cont = 0;
   break;
 }

 /*s_mod = sizeof(int); */

 switch(*queue_p->q_ptr){
 case 1:
   /* put num of characters in message header */
   sprintf(s1, "%d", nc);
   strcat(s1, "  ");
   s_mod = strlen(s1);

#ifdef DEBUG
 printf("s_mod:\t\t\t%d\n", s_mod);
#endif

   /* update message pointers */
   msg_p->b_cont->b_rptr = msg_p->b_cont->b_rptr - s_mod;
   bcopy(s1, msg_p->b_cont->b_rptr, s_mod);
   break;

 case 2:
   /* create header */
   /* allocate memory for header_buf, use allocb */
   header_buf = allocb (64, BPRI_MED);

   header_buf->b_cont = msg_p->b_cont;
   msg_p->b_cont = header_buf;

   /* initialise message pointers */
   msg_p->b_cont->b_rptr = msg_p->b_cont->b_wptr = msg_p->b_cont-
>b_datap->db_lim;
   sprintf(s1, "%d", na);
   strcat(s1, "  ");
   s_mod = strlen(s1);
#ifdef DEBUG
 printf("s_mod:\t\t\t%d\n", s_mod);
#endif

   /* update message pointers */
   msg_p->b_cont->b_rptr = msg_p->b_cont->b_rptr - s_mod;
   /* put num of A/a characters in message header */
   bcopy(s1, msg_p->b_cont->b_rptr, s_mod);
   break;

 default:
   break;
 }

 /* put data into header */
 if(cont){
   printf("\t\t\tHeader\t\t");
   h_rptr = msg_p->b_cont->b_rptr;
   while (h_rptr != msg_p->b_cont->b_wptr){
     if(digit == 1) len = strlen(s1)-(strlen(s1)-1);
     else len = strlen(s1);
     for(tmp=0; tmp<len; tmp++)
       printf("%c", h_rptr[tmp]);
     h_rptr = h_rptr +1;
   }
     printf("\n");
 }

 for(t=0; t<=(15*DATA_SIZE); t++) ;

#ifdef DEBUG
 printf("Now we're done with count_char.\n");
#endif

 putnext(queue_p, msg_p);

 return;
}


/******************************************************************

Routine: list

Description: This routine is called to initialize a linked list
for
             each module.

******************************************************************/
static void list (queue_t *queue_p, mblk_t *msg_p){

  initList();

  return;
}
```

ii. strm_drv.c

```
/* This file contains the source code for test diver */

/* --- System headers --- */
#include <sys/types.h>
#include <sys/param.h>
#include <sys/stream.h>
#include <stdio.h>
#include <sys/sysmacros.h>

/* --- Project headers --- */
#include "strm_decl.h"

/* Driver declarations */
#define TEST_DRV_STID TEST_DRV_ID1 /* Driver id number */
#define TEST_DRV_NAME "testdrv" /* Driver name */
#define TEST_DRV_HI_WAT 0x600  /* High water mark */
#define TEST_DRV_LO_WAT 0x300  /* Low water mark */

static int testdrv_open (queue_t *queue_p, int devp, int oflag,
int sflag);
static int testdrv_close (queue_t *queue_p, int flag);

static int testdrv_wput (queue_t *queue_p, mblk_t *msg_p);

static struct module_info testdrv_info=
{
  TEST_DRV_STID,  /* Driver id number */
  TEST_DRV_NAME,  /* Driver name */
  0,              /* min packet size */
  INFPSZ,         /* max packet size */
  TEST_DRV_HI_WAT,/* High water mark */
  TEST_DRV_LO_WAT /* Low water mark */
};

/* Read queue structure */
static struct qinit TestdrvRInit =
{
  NULL,
  NULL,
  &testdrv_open,
  &testdrv_close,
  NULL,
  &testdrv_info,
  NULL
};

/* Write queue structure */
static struct qinit TestdrvWInit =
{
  &testdrv_wput,
  NULL,
  NULL,
  NULL,
  NULL,
  &testdrv_info,
  NULL
};
```

```
/* testdrvinfo structure holds the qinit structures for the driver
*/
struct streamtab testdrvinfo =
{
  &TestdrvRInit, /* Points at qinit structure for the read queue
*/
  &TestdrvWInit, /* Points at qinit structure for the write q ueue
*/
  NULL,
  NULL
};

typedef struct {
  int minor_number;
  queue_t *queue_p;
} TP_DRV_T;

TP_DRV_T tp_rqueues_g [2]; /* this holds read queues for the
driver;
                         max 2 upper streams */


/****************************************************** ***********

Routine: testdrv_open

Description: This routine is called to open the TEST_DRV Driver
             in a Stream

******************************************************************/
static int testdrv_open (queue_t *queue_p, /* read queue */
                   int devp,  /* device number at the Stream end
*/
                   int oflag, /* no meaning for modules */
                   int sflag  /* indicates kind of open */
                   )
{
  TP_DRV_T *rqueue_p;
  unsigned long minor_number;

#ifdef DEBUG
  printf("Now we're inside testdrv_open.\n");
#endif

  /* initialise driver and its write and read queues */
  minor_number = minor(devp);
  rqueue_p = &tp_rqueues_g[minor_number];
  rqueue_p->minor_number = minor_number;
  rqueue_p->queue_p = queue_p;
  queue_p->q_ptr = WR(queue_p)->q_ptr = (char *) rqueue_p;

  return (FUNC_SUCCESS);
}
```

```
/*****************************************************************

Routine: testdrv_close

Description: This routine is called to close the TEST_DRV Driver
             in a Stream

*****************************************************************/
static int testdrv_close (queue_t *queue_p, int flag){
#ifdef DEBUG
 printf("Now we're inside testdrv_close.\n");
#endif
 ((TP_DRV_T *)queue_p->q_ptr)->queue_p = NULL;
 queue_p->q_ptr = NULL;
 WR (queue_p)->q_ptr = NULL;

 return (FUNC_SUCCESS);
}


/*****************************************************************

Routine: testdrv_wput

Description: This routine is called to process messages arrived at
             the write queue of the TEST_DRV Driver.
             The driver turns message around towards the Stream
head.

*****************************************************************/
static int testdrv_wput (queue_t *queue_p, /* write queue */
                   mblk_t *msg_p     /* message */
                   )
{
  unsigned long minor_number;

  printf("Turning message around.\n");
  printf("Messagetype: %d\n", msg_p->b_datap->db_type);

#ifdef DEBUG
  printf("Now we're inside testdrv_wput.\n");
  printf("Turning message around.\n");
  printf("Messagetype: %d\n", msg_p->b_datap->db_type);
#endif
  switch(msg_p->b_datap->db_type){

  case M_PROTO:
    minor_number = ((TP_DRV_T *)queue_p->q_ptr)->minor_number;
    if((minor_number%2)==1)
      putnext(tp_rqueues_g[(minor_number-1)].queue_p, msg_p);
    else
      putnext(tp_rqueues_g[(minor_number+1)].queue_p, msg_p);
    break;

  default:
    msg_p->b_datap->db_type = M_DATA;
    minor_number = ((TP_DRV_T *)queue_p->q_ptr)->minor_number;
    if((minor_number%2)==1)
      putnext(tp_rqueues_g[(minor_number-1)].queue_p, msg_p);
    else
      putnext(tp_rqueues_g[(minor_number+1)].queue_p, msg_p);
    break;
  }
  return (FUNC_SUCCESS);
}
```

iii. build_strm.c

```
/* Creates and builds STREAMS environment for VxWorks.
   Manages sending to and receiving messages from STREAMS
   (modules and driver).
*/

/* --- System headers --- */
#ifdef VXWORKS
#include "vxWorks.h"
#include "taskLib.h"
#include "strmLib.h"
#include "ioLib.h"
#endif

#include <sys/types.h>
#include <sys/param.h>
#include <sys/stream.h>
#include <sys/errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <stropts.h>
#include <fcntl.h>
#include <string.h>

/* -- System headers for ioctl --- */
#ifdef SUNOS5
#include <unistd.h>
#include <stropts.h>
#endif

/* --- Project headers --- */
#include "strm_decl.h"
#include "timer.c"

/* Global declarations */
int retfd0, retfd1;  /*STREAMS file descriptors */
int NUM_MODULES;     /* Number of modules pushed on driver */

extern struct streamtab testdrvinfo;  /* Driver definitions */
extern struct streamtab testmodinfo;  /* Modu le definitions */

/* Unitdata request */
struct unitdata_req {
  long PRIM_type;
  long DEST_addr;
};

/* Unitdata indication */
struct unitdata_ind{
  long PRIM_type;
  long SRC_addr;
};
```

```
/*************************************************************** **

Routine: build_strm

Description: creates and installs drivers and modules.
             Measures also time needed to perform these
operations.

****************************************************************/
extern void build_strm (
                  int num_modules /* number of modules to create
*/
                  )
{
  int count, t; /* counters */

/*  char driver_name[20];
  char convert_int[10];
  int max_minor = 1;
*/

#ifdef DEBUG
 printf("Starting timer (build stream env).\n");
#endif
  /* start timer */
  startTimer();

#ifdef DEBUG
  printf("Building streams ...\n");
  printf("Installing drivers.\n");
#endif

  printf("Building streams ...\n");
  printf("Installing drivers.\n");
   /*for (ct = 0; ct < max_minor; ct++){

      strcpy(driver_name, "testdrv" );
      sprintf(convert_int, "%d", ct);
      strcat(driver_name, convert_int);

      if(strmDriverAdd(driver_name, &testdrvinfo, 0, NULL,
               SVR3_STYLE_OPEN, SQLVL_GLOBAL) != OK){

      perror("Error while building streams (driver).\n");
      }
*/
  /* add STREAMS driver #0 to the STREAMS subsystem */
  if(strmDriverAdd("/dev/testdrv0", &testdrvinfo, 0, NULL,
               SVR3_STYLE_OPEN, SQLVL_GLOBAL) != OK){

    perror("Error while building streams (driver).\n");
    /*return;*/
  }

  /* add STREAMS driver #1 to the STREAMS subsystem */
  if(strmDriverAdd("/dev/testdrv1", &testdrvinfo, 0, NULL,
               SVR3_STYLE_OPEN, SQLVL_GLOBAL) != OK){

    perror("Error while building streams (driver).\n");
    /*return;*/
  }
  /* }*/

#ifdef DEBUG
  printf("Installing modules.\n");
#endif

  printf("Installing modules.\n");

  /* add a STREAMS module to the STREAMS subsystem */
  if(strmModuleAdd("testmod", &testmodinfo, NULL, SVR3_STYLE_OPEN,
                   SQLVL_GLOBAL) != OK){
    perror("Error while building streams (module).\n");
  }

#ifdef DEBUG
  printf("Opening drivers.\n");
#endif

  /* open driver #0 */
  if((retfd0 = open("/dev/testdrv0", O_RDWR,644)) < 0){
    perror("Open failed (driver 0).\n");
    return;
  }
  printf("\tfd0 = \t%d\n", retfd0);

  /* open driver #1 */
  if((retfd1 = open("/dev/testdrv1", O_RDWR,644)) < 0){
    perror("Open failed (driver 1).\n");
    return;
  }
  printf("\tfd1 = \t%d\n", retfd1);

#ifdef DEBUG
  printf("Opening modules.\n");
#endif

  /* open modules */
  NUM_MODULES = num_modules;
  for(count=0; count<num_modules; count++){
    /* insert module between the Stream head and driver */
    if(ioctl(retfd1,I_PUSH, (int)"testmod") < 0){
      printf("ioctl I_PUSH failed %d.\n", count);
      return;
    }
    /* busy waiting (e.g. possible initialisation routines) */
    for(t=0; t<=100000; t++) ;
  }

#ifdef DEBUG
  printf("Done.\n");
#endif

  /* stop timer and display values */
  stopTimer();
#ifdef DEBUG
 printf("Timer stopped (build stream env).\n");
#endif

}
```

```
/*****************************************************************

Routine: close_strm

Description: closes drivers and modules

*****************************************************************/
void close_strm (void){
int count;
#ifdef DEBUG
  printf("Now we're inside close_strm.\n");
#endif

  /* close driver #0 */
  if(close(retfd0) != OK){
    printf("Error while closing driver (testdrv0).\n");
    return;
  }

  /* close all modules */
  for(count=0; count<NUM_MODULES; count++){
    if(ioctl(retfd1, I_POP, (int)"testmod") < 0){
      printf("ioctl I_POP failed %d.\n", count);
      return;
    }
  }

  /* close driver #1 */
  if(close(retfd1) != OK){
    printf("Error while closing driver (testdrv1).\n");
    return;
  }
}


/*****************************************************************

Routine: send_msg

Description: sends a message to the STREAMS environment.

*****************************************************************/
extern int send_msg (int fd,    /* STREAMS file descriptor */
                 char *buf  /* message */
                 )
{

  struct strbuf ctlbuf;  /* control part of the message */
  struct strbuf databuf; /* data part of the message */
  struct unitdata_req unitdata_req; /* message */

#ifdef DEBUG
  printf("Now we're inside send_msg.\n");
#endif

  unitdata_req.PRIM_type = UNITDATA_REQ;

  ctlbuf.len = sizeof(struct unitdata_req);   /* length og data */
  ctlbuf.buf = malloc (sizeof(unitdata_req));
  ctlbuf.buf = (char *)&unitdata_req;         /* data buffer */
  databuf.buf = malloc(MODBLKSZ);            /* length og data */
  databuf.buf = buf;
  databuf.len = MODBLKSZ;                    /* data buffer */

  printf("Sending message.\n");
  printf("Size:\t%d\n", databuf.len);

#ifdef DEBUG
  printf("Sending message.\n");
  printf("Message:\t%s\n", databuf.buf);
  printf("Size:\t%d\n", databuf.len);
  printf("Prim type:\t%ld\n",unitdata_req.PRIM_type);
#endif

  /* create and send message downstream */
  if(putmsg(fd, &ctlbuf, &databuf, 0) < 0){
    perror("Error sending message.\n");
    return(-1);
  }

  return (databuf.len);
}


/*******************************************************************

Routine: recv_msg

Description: receives a message from the STREAMS environment.

*******************************************************************/
extern char *recv_msg (int fd,  /* STREAMS file descriptor */
                   int len   /* size of message */
                   )
{

  struct strbuf ctlbuf;   /* control part of the message */
  struct strbuf databuf;  /* data part of the message */
  struct unitdata_ind unitdata_ind; /* message */
  int retval;
  int flagsp;

#ifdef DEBUG
  printf("Now we're inside recv_msg.\n");
#endif


  ctlbuf.maxlen = sizeof(struct unitdata_ind); /* maximum buffer
length */
  ctlbuf.len = 0;                              /* length of data
*/
  ctlbuf.buf = malloc (sizeof(unitdata_ind));
  ctlbuf.buf = (char *)&unitdata_ind;          /* buffer */

  databuf.maxlen = len;        /* maximum buffer length */
  databuf.len = 0;             /*  length of data */
  databuf.buf = malloc(len);

  flagsp = 0;

  printf("Receiving message.\n");
```

```
#ifdef DEBUG
  printf("Receiving message.\n");
#endif

  /* receives a message from the Stream head */
  if((retval = getmsg(fd, &ctlbuf, &databuf, &flagsp)) < 0){
    printf("Error receiving message.\n");
    return ("error");
  }

  printf("Length of received message: \t%d\n\n", databuf.len);

#ifdef DEBUG
  printf("Message received:\t%s\n", databuf.buf);
  printf("Length of received message: \t%d\n\n", databuf.len);
#endif

 unitdata_ind.PRIM_type = UNITDATA_IND;

#ifdef DEBUG
    printf("Prim_ind type:\t%ld\n", unitdata_ind.PRIM_type);
#endif

  if(unitdata_ind.PRIM_type != UNITDATA_IND){
    errno = EPROTO;
#ifdef DEBUG
    printf("PRIM type:\t%ld\n", unitdata_ind.PRIM_type);
#endif
    return ("error");
  }

  if(retval){
#ifdef DEBUG
    printf("retval:\t%d\n", retval);
#endif
    errno = EIO;
    return ("error");
  }

  return (databuf.buf);
}


/********************************************************* ********

Routine: loop_send

Description: sends num_og_msg messages to the STREAMS environment.

*****************************************************************/
extern void loop_send (int fd, int num_of_msg){
  int a;
  char message[100];
  char convert_int[10];

#ifdef DEBUG
  printf("We're now inside loop_send.\n");
  printf("FD:\t%d\n\n", fd);
  printf("Num of msg:\t%d\n\n", num_of_msg);
  printf("Task delay!!!!!!!!!\n");
#endif
```

```
  /* used for debugging */
  taskDelay(600);
#ifdef DEBUG
  printf("Starting timer (send message).\n");
#endif
  /* starts timer */
  startTimer();

  /* generate messages to send downstream */
  for(a=0; a<num_of_msg; a++){
    strcpy(message, "Hello from E & A aaaaaaa#");
    sprintf(convert_int, "%d", a);
    strcat(message, convert_int);

#ifdef DEBUG
  printf("Sending message no. %d.\n", a);
#endif
      /* send messages downstream */
      if(send_msg(fd, message)<0){
        perror("Error while sending message.\n");
        return;
      }
  }

  /* stop timer and display values */
  stopTimer();
#ifdef DEBUG
 printf("Timer stopped (send message).\n");
#endif

}


/*******************************************************************

Routine: loop_recv

Description: receives num_of_msg messages from th e STREAMS
environment.

*******************************************************************/
/*void loop_recv (int fd, int len, int num_of_msg){

  int a;

#ifdef DEBUG
  printf("We're now inside loop_recv.\n");
  printf("FD:\t%d\n\n", fd);
  printf("Num of msg:\t%d\n\n", num_of_msg);
#endif

#ifdef DEBUG
 printf("Starting timer (receive message).\n");
#endif
  start timer
  startTimer();

  for(a=0; a<num_of_msg; a++){

#ifdef DEBUG
```

```
  printf("Receiving message no. %d.\n", a);
#endif
     receive a message from the Stream head
     if(recv_msg(fd, len)<0){
       perror("Error while receiving message.\n");
       return;
     }
  }

  stop timer and display values
  stopTimer();
#ifdef DEBUG
 printf("Timer stopped (receive message).\n");
#endif
}
*/
```

iv. main_build.c

```
/* Builds and closes STREAMS environment in vxWorks.
   Code executes on VxWorks target.
*/

/* --- System headers --- */
#ifdef VXWORKS
#include "vxWorks.h"
#include "taskLib.h"
#include "strmLib.h"
#include "ioLib.h"
#endif

#include <sys/types.h>
#include <sys/param.h>
#include <sys/stream.h>
#include <sys/errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <stropts.h>
#include <fcntl.h>
#include <string.h>

/* -- System headers for ioctl --- */
#ifdef SUNOS5
#include <unistd.h>
#include <stropts.h>
#endif

/* --- Project headers --- */
#include "strm_decl.h"
#include "timer.c"

void main_build (int num_mod){
  int b_strm;
  /* creates and activates task 'strm_build' with priority '59',
stack size
     '2000' and entry point of function to execute */
  b_strm = taskSpawn("strm_build", 59, 0, 2000, (void
*)build_strm, num_mod,0,
                 0,0,0,0,0,0,0,0);

}

void main_close (void){
  int c_strm;
  /* creates and activates task 'strm_close' with pri ority '59',
stack size
     '2000' and entry point of function to execute */
  c_strm = taskSpawn("strm_close", 59, 0, 2000, (void
*)close_strm, 0,0,0,0,0,
                 0,0,0,0,0);

}
```

v. main_test.c

```
/* Tests STREAMS environment in vxWorks.
   Code executes on VxWorks target.
*/

/* --- System headers --- */
#ifdef VXWORKS
#include "vxWorks.h"
#include "taskLib.h"
#include "strmLib.h"
#include "ioLib.h"
#endif

#include <sys/types.h>
#include <sys/param.h>
#include <sys/stream.h>
#include <sys/errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <stropts.h>
#include <fcntl.h>
#include <string.h>

/* -- System headers for ioctl --- */
#ifdef SUNOS5
#include <unistd.h>
#include <stropts.h>
#endif

/* --- Project headers --- */
#include "strm_decl.h"

extern int retfd0, retfd1;  /*STREAMS file descriptors */

void main_test (int num_of_msg){
  /* creates and activates task 'm_send' with priority '59', stack
size
     '5000' and entry point of function to execute */
  taskSpawn("m_send", 59, 0, 5000, (void *)loop_send, retfd1,
num_of_msg,0,0,
          0,0,0,0,0,0);

  /* m_receive has a higher priority and can suspend m_send when a
message
     is received */
  /* creates and activates task 'm_receive' with priority '58',
stack size
     '2000' and entry point of function to execute */
  taskSpawn("m_receive", 58, 0, 5000, (void *)loop_recv, retfd0,
1000,
          num_of_msg,0,0,0,0,0,0,0);

}

/*
void main_test2 (int num_of_msg){
  taskSpawn("m_send", 59, 0, 5000, (void *)loop_send, retfd1,
num_of_msg,0,0,0,
          0,0,0,0,0);

  taskSpawn("m_receive", 59, 0, 5000, (void *)loop_recv, retfd0,
1000,
          num_of_msg,0,0,0,0,0,0,0);

}
*/
```

vi. strm_decl.h

```
/* Header file for STREAMS environment.
   Defines common information for modules, drivere
   and application in user space
 */

#ifndef _STRMDECL_H
#define _STRMDECL_H

/* --- System headers --- */
#include <sys/types.h>
#include <sys/param.h>
#include <sys/stream.h>

#define FUNC_SUCCESS   0

/* --- Identifiers --- */
#define TEST_DRV_ID1 1 /* Driver  identifier */
#define TEST_MOD_ID1 2 /* Module identifier */

/* --- Message declarations --- */
#define NUM_MESSAGES 100 /* Number of messages */
#define MODBLKSZ 1188    /* Size of message blocks */

/* --- Primitives initiated by the service user  --- */
#define UNITDATA_REQ 2

/* --- Primitives initiated by the service provider  --- */
#define UNITDATA_IND 5

/* Function declarations */
extern void build_strm (int num_modules);
extern void close_strm (void);
extern void loop_send (int fd, int num_of_msg) ;
extern void loop_recv (int fd, int len, int num_of_msg);

extern int send_msg (int fd, char *buf);
extern char *recv_msg (int fd, int len);

#endif
```

c. *Linked List*

i. list.c

```
/* --- VxWorks header files --- */
#ifdef VXWORKS
#include "vxWorks.h"
#include "taskLib.h"
#include "timer.c"
#endif

/* --- System headers --- */
#include <stdio.h>
#include "stdlib.h"

/* --- Project headers --- */
#include "list.h"

#ifdef VXWORKS
node *start;
#endif

/* Global declarations */
int NUM_OF_NODES = NUM_NODES;

int COUNT = 0;


/*****************************************************************

Routine: initList

Description: creates and initialises linked list

*****************************************************************/
#ifdef VXWORKS
extern void initList (void){

  COUNT=0;

 start = NULL;
 start = (node *) malloc(sizeof(node));
 create(start);

#ifdef DEBUG
 printf("Done creating list!\n");
 printf("Node's ID: %d\n", start->nodeID);
 printf("Node's name: %s\n", start->name);
#endif

}
#endif

/* Create list of nodes by default NUM_NODES=200 */
extern void create (node *pnode) {

  int t;

#ifdef DEBUG
  printf("You're now inside create\n");
  printf("Number of nodes %d\n", NUM_NODES);
  printf("Making node number %d\n", COUNT+1);
#endif

  pnode->nodeID = COUNT+1;
  pnode->name = malloc (sizeof(string));
  pnode->name = "testnode";

#ifdef DEBUG
  printf("Node's name: %s\n", pnode->name);
#endif

  if (COUNT==NUM_NODES-1){
    pnode->next = NULL;

#ifdef DEBUG
    printf("Last node\n");
#endif
  }
  else {
    pnode->next = (node *) malloc (sizeof(node));
    COUNT++;
    create (pnode->next);
  }
  for(t=0; t<10000;t++) ;
  return;
}


/*****************************************************************

Routine: locateListNode

Description: locates node with a given node id

*****************************************************************/
#ifdef VXWORKS
void locateListNode (int id){

  node *ptnode =  NULL;
  ptnode = (node *) malloc(sizeof(node));
  ptnode->name = (string) malloc(sizeof(string));
  ptnode = start;

#ifdef DEBUG
  printf("You're now inside locateListNode\n");
#endif

  locateNode (ptnode, id);
}
#endif

/*#ifdef SUNOS5*/

/* Find node */
/*node **/extern void locateNode (node *pnode, int id){
#ifdef DEBUG
  printf("You're now inside locateNode\n");
#endif

  if(pnode->nodeID == id){
```

```
#ifdef DEBUG
    printf("(findNode) Node's id : \t%d\n", pnode->nodeID);
#endif
    printf("Node's id : \t%d\nNode's name: \t%s\n", pnode->nodeID,
pnode->name);
  }
  if(pnode->next->nodeID == id){
#ifdef DEBUG
    printf("(findNode) Node's id : \t%d\n", pnode->next->nodeID);
#endif
    pnode = pnode->next;
    printf("Node's id : \t%d\nNode's name: \t%s\n", pnode->nodeID,
pnode->name);
    /*return pnode;*/
  }else{
    if (pnode->next->next == NULL)
      return; /*NULL;*/
    else
      locateNode(pnode->next, id);
  }
}
/*#endif*/


/******************************************************************

Routine: displayList

Description: displays all nodes in linked list

******************************************************************/
#ifdef VXWORKS
void displayList(void){

  node *ptnode =  NULL;
  ptnode = (node *) malloc(sizeof(node));
  ptnode->name = (string) malloc(sizeof(string));
  ptnode = start;

#ifdef DEBUG
  printf("You're now inside displayList\n");
  printf("Start node's ID: %d\n", start->nodeID);
  printf("Start node's name: %s\n\n\n", start->name);
#endif

  display(ptnode);

#ifdef DEBUG
  printf("\nStart node's ID: %d\n", start->nodeID);
  printf("Start node's name: %s\n\n\n", start->name);
#endif

  /*free(ptnode->name);*/
/*  free(ptnode);*/
}
#endif

/* List nodes */
extern void display (node *pnode) {

#ifdef DEBUG
```

```
  printf("You're now inside display\n");
#endif

  /*if(pnode == NULL) return;*/

  if (pnode->next != NULL) {
    printf("Node's ID: %d\n", pnode->nodeID);
    printf("Node's name: %s\n", pnode->name);
    display (pnode->next);

  }else{
    printf("Node's ID: %d\n", pnode->nodeID);
    printf("Node's name: %s\n", pnode->name);
  }
  return;
}


/*****************************************************************

Routine: addListNode

Description: adds node to linked list (at the end of list)

*****************************************************************/
#ifdef VXWORKS
void addListNode (void){

  node *ptnode =  NULL;
  ptnode = (node *) malloc(sizeof(node));
  ptnode->name = (string) malloc(sizeof(string));
  ptnode = start;

#ifdef DEBUG
  printf("You're now inside addListNode\n");
#endif

  ptnode = addNode(ptnode);

#ifdef DEBUG
 printf("Node's ID: %d\n", start->nodeID);
 printf("Node's name: %s\n", start->name);
#endif

 /*free(ptnode);*/
}
#endif

/* Add nodes */
extern node *addNode (node *pnode) {

   node *pn = pnode;

#ifdef DEBUG
  printf("You're now inside addNode\n");
#endif

  if (pnode != NULL) {
    while (pnode->next != NULL)
      pnode = pnode->next;
    pnode->next = (node *) malloc (sizeof(node));
    pnode = pnode->next;
    pnode->next = NULL;
    NUM_OF_NODES++;
    COUNT++;
    pnode->nodeID = NUM_OF_NODES;

    pnode->name = (string) malloc (sizeof(string));
    pnode->name = "testnode";

#ifdef DEBUG
    printf("Number of nodes %d\n", (COUNT+1));
    printf("Node's ID: %d\n", pnode->nodeID);
#endif

    return pn;
  }
  else {
    pnode = (node *) malloc (sizeof(node));
    pnode->next = NULL;
    pnode->nodeID = 0;
    NUM_OF_NODES = 1;
    pnode->name = (string) malloc (sizeof(string));
    pnode->name = "testnode";

    return pnode;
  }
}


/******************************************************************

Routine: removeListNode

Description: removes node from linked list (at the bigining of
list)

******************************************************************/
#ifdef VXWORKS
void removeListNode (void){
  /*node *ptnode =  NULL;
  ptnode = (node *) malloc(sizeof(node));
  ptnode->name = (string) malloc(sizeof(string));
  ptnode = start;
*/
#ifdef DEBUG
  printf("You're now inside removeListNode\n");
#endif

  start = removeNode (start);

#ifdef DEBUG
  printf("Node's ID: %d\n", start->nodeID);
  printf("Node's name: %s\n", start->name);
#endif

  /*free(ptnode);*/
}
#endif
```

```
/* Remove nodes */
extern node *removeNode (node *pnode) {

   node *tempNode;

#ifdef DEBUG
  printf("You're now inside removeNode\n");
  printf("Node removed node id: %d\n", pnode->nodeID);
#endif

  tempNode = pnode->next;
/*  free(pnode->name);*/
  free(pnode);
  COUNT--;

#ifdef DEBUG
  printf("Number of nodes %d\n", (COUNT+1));
#endif

  return tempNode;
}


/*********************************************************** ******

Routine: clearList

Description: removes all nodes from linked list

*****************************************************************/
#ifdef VXWORKS
void clearList (void){

#ifdef DEBUG
  printf("You're now inside clearList\n");
  printf("Node's ID: %d\n", start->nodeID);
  printf("Node's name: %s\n", start->name);
#endif

  clear(start);
  NUM_OF_NODES = NUM_NODES;
  COUNT = 0;
}
#endif

extern void clear(node *pnode){

#ifdef DEBUG
  printf("You're now inside clear (list)\n");
  printf("Node's ID: %d\n", pnode->nodeID);
  printf("Node's name: %s\n", pnode->name);
#endif

  while(pnode != NULL)
    pnode = removeNode(pnode);
}
```

ii. list.h

```
/* Header file for linked list routines.
   Defines common information for linked list processing.
*/

#ifndef _LIST_H
#define _LIST_H

/* --- VxWorks header files --- */
#ifdef VXWORKS
#include "vxWorks.h"
#include "taskLib.h"
#include "time.h"
#endif

/* --- System headers --- */
#include "stdio.h"
#include "stdlib.h"

/*#define NUM_OF_NODES 5*/

#define NULL 0

#define NUM_NODES 200 /* number of nodes in the list */

typedef char * string;

/* Structure used for list nodes */
struct list_element {
  int nodeID;  /* node identifier */
  string name; /* node name */
  struct list_element *next; /* pointer to the next node */
};

typedef struct list_element node;

/* Function declarations */
extern void initList (void);
extern void create (node *pnode);
extern void display (node *pnode);
extern node *addNode (node *pnode);
extern node *removeNode (node *pnode);
extern void locateNode (node *pnode, int id);
extern void clear(node *pnode);

#endif
```

iii. Vxlist.c

```
#ifdef VXWORKS
#include "vxWorks.h"
#include "taskLib.h"
#include "time.h"
#endif

#include "stdio.h"
#include "stdlib.h"

#include "list.h"

void list_test(void){
  taskSpawn("linked_list", 59, 0, 5000, (void *)initList,
0,0,0,0,0,0,0,0,0,0);
}
```

d. Timer

i. timer.c

```
/* --- System header files --- */
#ifdef WXWORKS
#include "vxWorks.h"
#include "stdio.h"
#include "taskLib.h"
#include "tickLib.h"
#endif

#ifdef SUNOS5
#include <stdio.h>
#endif

#include "time.h"
#include "timers.h"
#include "stdlib.h"
#include "sys/time.h"

/* --- Global declarations --- */
struct timespec tp;       /* Stores current time values */
time_t tstart, tstop;     /* Start and stop time in nsec */
long startTime, stopTime; /* Start and stop time in sec */

/* forward declarations */
static void displayValues (void);


/******************************************************************

Routine: startTimer

Description: starts timer

******************************************************************/

static int startTimer (void){
  /* get the current time of the clock */
  if(clock_gettime(CLOCK_REALTIME, &tp) == ERROR){
     printf("Error while starting timing!\n");
     return(ERROR);
    }
#ifdef DEBUG
  printf("Start measuring time!\n");
#endif
  startTime = tp.tv_nsec;
  printf("Start ns time:\t\t%ld\n",tp.tv_nsec);
  tstart = tp.tv_sec;
  printf("Start s time:\t\t%ld\n",tp.tv_sec);
  return(OK);
}
```

```
/*********************************** **************************

Routine: stopTimer

Description: stops timer and displays values

**************************************************************/
static int stopTimer (void){
  /* get the current time of the clock */
  if(clock_gettime(CLOCK_REALTIME, &tp) == ERROR){
     printf("Error while starting timing!\n");
     return(ERROR);
   }
#ifdef DEBUG
  printf("Stop measuring time!\n");
#endif
  stopTime = tp.tv_nsec;
  printf("Stop ns time:\t\t%ld\n",tp.tv_nsec);
  tstop = tp.tv_sec;
  printf("Stop s time:\t\t%ld\n",tp.tv_sec);
  /* display values */
  displayValues();
  return(OK);
}


/**************************************************************

Routine: displayValues

Description: calculates and displays values in msec

**************************************************************/
static void displayValues (void){
  time_t sec;
  long nsec;
  long msec;

  if((tstop < tstart) || ((tstop == tstart) && (stopTime <
startTime)))
    printf("Error\n");

  else{
    sec = tstop - tstart;

    if(stopTime >= startTime)
      nsec = stopTime - startTime;
    else
      nsec = (stopTime - startTime);

    msec = (sec * MILLISEC) + (nsec / MICROSEC);

    printf("Measured time: \t\t%ld ms\n", msec);

  }
}
```

Appendix B: Test Results