Lappeenrannan teknillinen korkeakoulu

Tietotekniikan osasto

Real−time Linux in electric drive systems

Diplomityön aihe hyväksytty Tietotekniikan

osaston osastoneuvoksessa 17.5.2000

Diplomityön ohjaaja: Professori Olli Pyrhönen

Diplomityön tarkastaja: Professori Jari Porras

___/___−2000 _________________

Ossi Laakkonen

Lintulahdenaukio 8 A 62

00500 Helsinki

050−3530354

ABSTRACT

Lappeenranta university of technology
Department of information technology
Ossi Laakkonen

Real−time Linux in electric drive systems

Master’s thesis
2000

47 pages, 13 figures, 7 tables and 4 appendices
Supervisor: Professor Jari Porras
Keywords: RTLinux, real−time operating system, electric drive, embedded system, open source, kernel

The use of real−time operating systems in embedded systems is growing all the time. Embedded computers are used more and more in different systems for example controlling electric drives. Nowadays the control of electric drives is handled by fast special processors such as a DSP, which makes programming and updating slow and difficult because it is made with low−level assembler language. The solution is general purpose processors and real−time operating systems. Commercial real−time operating systems are expensive and it may be difficult or even impossible to get the source code. Linux is a non−commercial open source operating system, so you can use and modify it freely. There are many extensions to Linux that make Linux a real−time operating system. You can choose between hard and soft real−time versions. There are several development environments to Linux but they need improvements before they can be used in large scale in industry. Real−time Linux is not capable of fast control loops (<100 μs) due to the speed of the processors. However, once faster processors are developed improvements will follow.

TIIVISTELMÄ

Lappeenrannan teknillinen korkeakoulu
Tietotekniikan osasto
Ossi Laakkonen

Reaaliaika Linux:in käyttö sähkökäyttöjärjestelmissä

Diplomityö
2000

47 sivua, 13 kuvaa, 7 taulukkoa ja 4 liitettä
Tarkastaja: Professori Jari Porras
Hakusanat: RTLinux, reaaliaikakäyttöjärjestelmä, sähkökäyttö, sulautettu järjestelmä, avoin lähdekoodi, kerneli
Keywords: RTLinux, real−time operating system, electric drive, embedded system, open source, kernel

Reaaliaikaisten käyttöjärjestelmien käyttö sulautetuissa järjestelmissä on kasvamassa koko ajan. Sulautettuja tietokoneita käytetään yhä useammassa kohteessa kuten sähkökäyttöjen ohjauksessa. Sähkökäyttöjen ohjaus hoidetaan nykyisin yleensä nopealla digitaalisella signaaliprosessorilla (DSP), jolloin ohjelmointi ja päivittäminen on hidasta ja vaikeaa johtuen käytettävästä matalan tason Assembler−kielestä. Ratkaisuna yleiskäyttöisten prosessorien ja reaaliaikakäyttöjärjestelmien käyttö. Kaupalliset reaaliaikakäyttöjärjestelmät ovat kalliita ja lähdekoodin saaminen omaan käyttöön jopa mahdotonta. Linux on ei−kaupallinen avoimen lähdekoodin käyttöjärjestelmä, joten sen käyttö on ilmaista ja sitä voi muokata vapaasti. Linux:iin on saatavana useita laajennuksia, jotka tekevät siitä reaaliaikaisen käyttöjärjestelmän. Vaihtoehtoina joko kova (hard) tai pehmeä (soft) reaaliaikaisuus. Linux:iin on olemassa valmiita kehitysympäristöjä mutta ne kaipaavat parannusta ennen kuin niitä voidaan käyttää suuressa mittakaavassa teollisuudessa. Reaaliaika Linux ei sovellus nopeisiin ohjauslooppeihin (<100 μs) koska nopeus ei riitä vielä mutta nopeus kasvaa samalla kun prosessorit kehittyvät. Linux soveltuu hyvin rajapinnaksi nopean ohjauksen ja käyttäjän välille ja hitaampaan ohjaukseen.

Table of Contents

ACRONYMS

API	Application Programming Interface
ARP	Address Resolution Protocol
BSD	Berkeley Software Distribution
CCITT	Consultative Committee for International Telegraph and Telephone
CLI	Clear Interrupt
CPU	Central Processing Unit
CSR	Control and Status Register
DDP	Datagram Delivery Protocol
DNS	Domain Name Server
DTE	Data Terminal Equipment
FTP	File Transfer Protocol
GPL	GNU General Public License
GUI	Graphical User Interface
I/O	Input/Output
ICMP	Internet Control Message Protocol
INET	Internet
IP	Internet Protocol
IPC	Inter Process Communication
IPv6	Internet Protocol version 6
IPX/SPX	Internetwork Packet Exchange/Sequenced Packet Exchange
LCD	Liquid Crystal Display
LED	Light Emitting Diode
LILO	Linux Loader
NFS	Network File System
NCP	NetWare Core Protocol
PC	Personal Computer
PCI	Peripheral Connection Interface
PIC	Programmable Interrupt Controller
PnP	Plug and Play

POP3	Post Office Protocol 3
PPC	Power PC
PPP	Point−to−Point Protocol
RCS	Revision Control System
RTC	Real−Time Clock
SCSI	Small Computer Systems Interface
SLIP	Serial Line Internet Protocol
SMB	Server Message Block
SMP	Symmetric Multi Processor
SMTP	Simple Mail Transfer Protocol
STI	Set Interrupt
SYSV	System V
TCP/IP	Transfer Control Protocol/Internet Protocol
TTY	Terminal Type
UDP	User Datagram Packet
WWW	World Wide Web
X25	CCITT recommendation for packet switched network DTE interfacing
x86	Intel 80x86 microprocessor family

1. INTRODUCTION

The aim of this work was to determine if the Linux and especially some version of the real−time Linux was mature and reliable enough to be an operating system in future electric drive systems. Presently special processors like DSPs are used with tailored operating systems that are quite hard and slow to upgrade because almost all programming is made with low−level assembler language, which is not the best programming language to program large complicated operating systems. The upgrading cycle is getting faster and faster and with this old solution it is far too expensive to make new versions in such a short period. A new solution to the problem is to use general purpose processors with an existing operating system like RTLinux to control electric drive systems. With an existing operating system you have the full range of development tools, including high−level programming language compilers, simulators and debuggers and last but not least large scale hardware device drivers. RTLinux is a hard real−time extension to Linux and it is one of the most reliable and mature versions of real−time Linux and that is why it was chosen this project. RTLinux is also a non−commercial version of real−time Linux. This means no support from any vendor. Also there is a very good supporting network working with Linux, and RTLinux. Today’s electric drive systems, for example frequency converters, need very fast control cycles (5−25 μs) and my intention is to find the limits of the RTLinux.

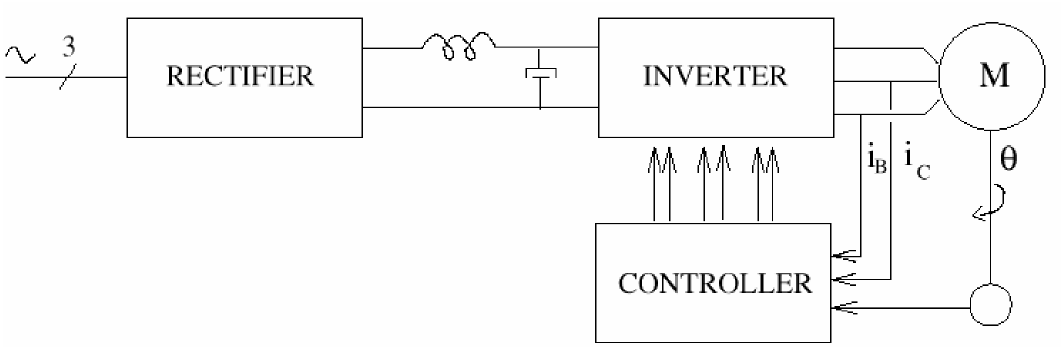


Figure 1.1: Typical AC drive [Cec1999]

Figure 1.1 shows there is a controller in the electric drive system. It will be replaced with RTLinux and a general purpose processor if RTLinux is fast enough to do the controlling algorithms. The best situation would be were there is only RTLinux controlling the inverter. [Figure 1.2].

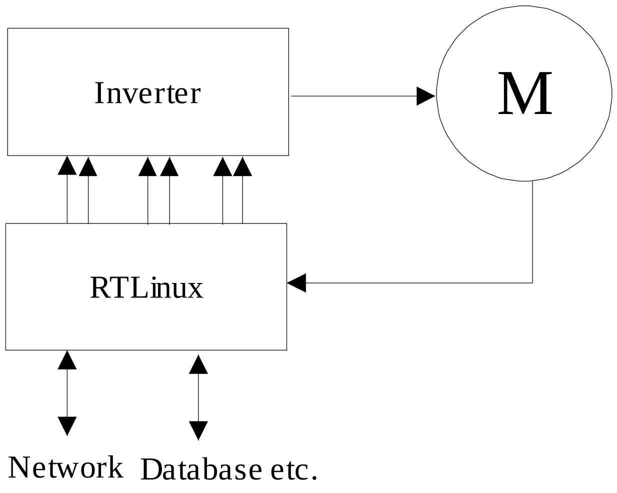


Figure 1.2: Future AC drive

2. EMBEDDED AND REAL−TIME SYSTEMS

2.1 Embedded system

What is an embedded system? That is probably the first question many people might ask. You can see nowadays embedded systems almost everywhere or at least embedded systems exist everywhere. However, usually you don’t even know where they exist. For example when you use an elevator you use an embedded system because there is a small computer controlling the elevator and that computer is an embedded system [Wil1999], [Web2000]. Or if you use an electric shaver you use an embedded system to control the battery charge and so on. Even your car may have its own embedded system, at least if you have a fancy new car. Mobile phones must also be mentioned because I am from Finland ("the home of mobile phones") where at least half of the population use an embedded system every day. As you can see embedded systems are not rare and only used in industry and hi−tech companies [Göt1999].

A few years ago one definition of the embedded system was that it must not have a user interface. Another definition was that it must be tailored to a purpose, not any general purpose system. The first definition is not valid anymore because the user interface is one of the most important part of any system. Elevators have buttons, shavers have some kind of display (LEDs or LCD) and cars have indicators (=user interface). More and more embedded systems are connected to a network and where sophisticated user interfaces exists. The second definition is also kind of obscure because you can build your own embedded system with ordinary components like x86 Intel processor and some peripherals that are used in home PCs.

An embedded system can be a tiny micro controller with few bytes of memory and some I/O or powerful microprocessor with many Mbytes of memory, fast connection to a network and sophisticated operating system or anything between them. You must remember that there is no exact definition to describe an embedded system.

2.2 Real−time system

There are two different types of real−time systems. Soft real−time system and hard real−time system. The difference between those two systems is small but significant. When you need precise timing you must have a hard real−time system but if you can allow some missing time limits you can have a soft real−time system.

2.2.1 Hard real−time system

There is no standardized definition of a hard real−time system but many people have tried to define it. Here is one definition of a hard real−time system which is from Phil Wilshire's presentation from the Real Time Linux Workshop, Vienna 1999.

A real−Time system is:

- a system capable of performing a function within guaranteed time boundary
- were timing is in µs and not in minutes
- were a jitter is about 25 µs (or approx.)
- were the maximum jitter boundary is defined but never exceeded
- were there is a high repetition rate in the range of a few 10 µs to 100 µs
- were there is 100% reliability with no missing beats.

As you can see there are some time limits the above definition but they are not standardized. You can just as well have a real−time system which uses much bigger or even smaller time limits and it is still a real−time system. The last definition is the hardest to implement but with the right hardware and a good operating system you can make a hard real time system. A real−time system must be deterministic before it can be a hard real−time system. Hardware affects the time limits too so a real−time system is as good as its weakest link.

2.2.2 Soft real−time system

A soft real−time system can be defined like a hard real−time system but there is one difference. A soft real−time system can not guarantee 100% reliability so it can some times miss some time limits. A soft real−time system guarantees a good average performance (i.e. usually interrupt service startup time) but with a small probability the maximum interrupt service startup time will be longer.

2.2.3 Different approaches to implement non real−time services to real−time system

There are three different ways to implement a real−time system with non real−time services such as TCP/IP, file access, display etc. You can add non real−time services to a real−time kernel directly, like in VxWorks and QNX. This causes the kernel size to increase much and makes kernel less deterministic. Also it is quite difficult to write a non real−time processes in real−time space. Another way is to modify the standard kernel to be completely pre−emptable like in RT−IX. However, this is very often difficult and is not used often. Lastly, and maybe the best way is to run a non real−time kernel as the lowest priority task on real−time kernel like in RTLinux and Windows NT Embedded (real−time version) [Hon1997]. This gives you the ability to use standard Linux services (tools, development environments etc.) and real−time capabilities simultaneously.

3. LINUX

The development of the Linux started in 1991 when Linus Torvalds (a Finnish University student) wanted to create a new operating system which is better than Minix. The current stable version of Linux is 2.2 and it was released January 25th, 1999 [WWW2].

Linux is a Posix compliant operating system with SYSV and BSD extensions. This means that it looks like a Unix but the source code is different. Linux itself is just a kernel where different distributions with different utilities and applications have been built around it.

Linux was developed under GPL (GNU General Public License) and it is an open source i.e. its source code is available to everyone. Linux is a low−cost alternative to commercial operating systems and therefore it is becoming more popular [Hon2000]. Also its reliability and features are a big advantage.

3.1 Supported processors

Linux supports a lot of different hardware platforms. Almost any processor can be selected and Linux will work with it. The most popular processors are Intel x86 family, AMD x86 compatible processors, PowerPC and Alpha. Linux also supports tens of other processors and micro controllers. Porting Linux to a different board is not that difficult it has been designed to be portable [Che1999].

3.2 Why choose Linux for your operating system?

Usually when you start a new project you have to choose an operating system. With commercial operating systems there are a couple of major problems [WWW3], [WWW1]:

- they are costly
- possibly they are not an open source
- no built−in networking
- poor reliability
- expensive development tools
- low performance
- lack of support.

Linux is an exception because it is not a commercial operating system. It has a lot of advantages:

- low−cost (only ten’s of dollars)
- source code freely available
- GPL
- stable
- excellent network facilities
- multi−user
- reliable
- backwards compatible
- embeddable
- real−time extensions
- comes with a complete development environment
- ideal environment to run servers
- wide variety of commercial software [Tol2000]
- easily upgraded
- supports multiple processors
- true multi tasking
- excellent window system

However, this system does have some disadvantages:

- replication of the packages

- installation manuals and documentation need development
- inadequate PnP support
- most installations presume a network connection
- system administration is difficult

In table 3.1 Linux is compared to Microsoft Windows NT Server 4.0 and you can see the benefits of using Linux.

3.3 Kernel

The kernel is the heart of the operating system. It is responsible for the management of the memory, processes, tasks and peripherals. It provides all the essential services to other parts of the operating system and applications. It is loaded into the memory after booting and therefore you should not build too big kernel to avoid wasting memory especially in embedded systems where there is not much available memory. [War1999].

3.3.1 Installing a new kernel

When you start your Linux machine for the first time after installation there will be a general purpose kernel working that may not have all the drivers that are needed and will also contain a lot of "junk" you do not need so you have to compile a new kernel. Firstly, the source code of the kernel will be requested and then the kernel will need to be configured. This happens with a *make config* command and after typing the command you have alist of selectable kernel parts. You just have to pick all needed parts and in some cases know some hardware settings and after that you can compile new kernel. Compilation of new kernel starts with *make dep* command that insures that all dependencies, like include files, are all in place. Then you should type *make clean* that cleans old object files that old kernel version leaves behind. After that command *make bzImage*

Table 3.1:
Microsoft Windows NT Server 4.0 versus UNIX
By
John Kirch
Networking Consultant and Microsoft Certified Professional (Windows NT)

Component	**Linux**	**Windows NT Server 4.0**
Operating System	Free, or around $49.95 for a CD−ROM distribution	Five−User version $809 10−User version $1129 EE 25−User Version $3,999
Free online technical support	Yes, Linux Online or Redhat	No
Kernel source code	Yes	No
Web Server	Apache Web Server	No
FTP Server	Yes	Yes
Telnet Server	Yes	No
SMTP/POP3 Server	Yes	No
DNS	Yes	Yes, though reports indicate that it is a broken implementation with limited functionality.
Networking	TCP/IP, IPv6, NFS, SMB, IPX/SPX, NCP Server (NetWare Server), AppleTalk, plus many other protocols	TCP/IP, SMB, IPX/SPX, AppleTalk, plus many other protocols
X Window Server (For running remote GUI−based applications)	Yes	No
Remote Management Tools	Yes, all tools	Web Administrator 2.0 (a recent addition) offers a large, but still not complete, set of tools.
News Server	Yes	No
C and C++ compilers	Yes	No
Perl 5.0	Yes	No
Revision Control	Yes, RCS	No
Number of file systems supported	32	3
Disk quotas support	Yes	No
Number of GUIs (window managers) to choose from	4	1

compiles new kernel. If you wish to clean all the mess that compilation of the new kernel made you can use *make mrproper* command that cleans every object file and also your config file. If you selected some kernel parts to be used as a module you have to compile these modules also and that happens with *make modules* and *make modules_install* commands.

Installing a new kernel is relatively easy. Firstly, *lilo* (Linux Loader) is informed of where the new kernel exists and *lilo* command is executed. The *lilo* configuration file is usually located in */etc/* directory and its name is *lilo.conf*.

3.3.2 Monolithic kernel and modularity

Linux is a monolithic kernel i.e. it is a large program which includes all the functions needed to run the computer of which the functions ???? access to all kernel data structures and routines [Rus1999]. Another type of kernel is a micro−kernel where all functionality is divided into separate units that have a communication mechanism between them.

A Linux kernel is modular, smaller, and more flexible than a micro kernel. However there is some extra code in the modules (*init_module* and *cleanup_module* functions) with extra data structures.

The main advantage of the Linux kernel is the ability to install modules into the kernel without recompiling a kernel. That means you can dynamically link your own modules into a kernel while running Linux and unlink them also. This is a great advantage because a basic kernel can be made small and all needed drivers for example can be loaded when needed. Most kernel modules are usually drivers, file systems or pseudo−device drivers. Kernel modules can be loaded and unloaded by the user (*insmod* and *rmmod)* or by the kernel daemon (*kerneld)* which loads modules only when needed and unloads them after use.

Loadable modules bring flexibility but also responsibility. Dynamically loadable modules mean less efficient use of the memory and the resources of the kernel. Also when the module is loaded it has the same capabilities as normal a kernel and if it is not properly coded, it will crash whole the kernel which will require a reboot. The kernel code has access to the whole memory area and serious damage will occur if the wrong bits are written to the wrong places. Usually when a program is written in the user space (not a kernel module) no serious damage occurs because the user space programs do not have access to critical services or devices.

The following explains what happens when the kernel module is loaded to the memory:

1. *Insmod* reads the *module.o* file and copies it to a temporary buffer and fixes all unresolved symbols.
2. System call *load_module* is called and the kernel allocates memory using *vmalloc.*
3. User space image of the module is copied to the in−kernel buffer and the *init_module* is called.

3.3.3 Kernel mechanism

Linux uses some mechanisms and general tasks so that the kernel parts work efficiently. Firstly, one of the most important mechanisms is Bottom Half Handling. This mechanism allows some work to be done later. If for example the device driver sets an interrupt the operating system starts to handle the interrupt immediately. Often there is some work that could wait for a while so the Linux kernel can queue that work to be executed later. There are 3 data structures that are used, *bh_mask* which informs which handlers are in use, *bh_active* tells the user what handler should be called and *bh_base* has pointers to handlers [Figure 3.1]. There are 32 different Bottom Half Handlers and five of them are specific: *Timer*, *Console*, *Tqueue*, *Net* and *Immediate*. Table 3.2 describes these handlers. *Bh_active* is checked at the end of each system call just before returning to the calling process.

Task queues are used to process work later. Timer task queues are processed when the timer task bottom half handler runs. A task queue is a simple data structure which includes a linked list of data structures that contain the address of the routine and the pointer to some data. The routine is called when the data structure element is processed. There are three queues created and managed by the kernel *timer*, *immediate* and *scheduler*, but anything in the kernel (device driver etc.) can create and use task queues.

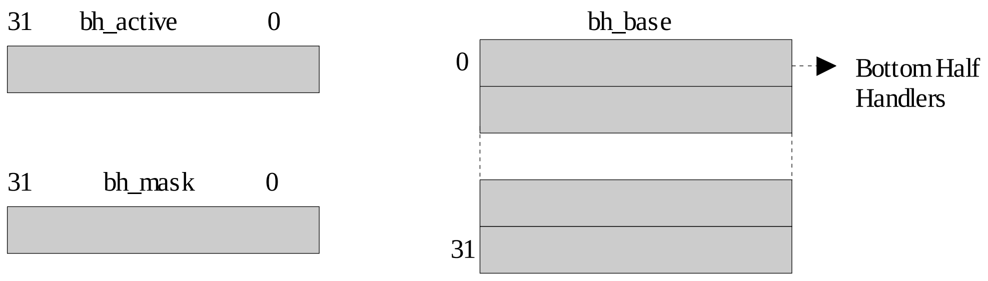


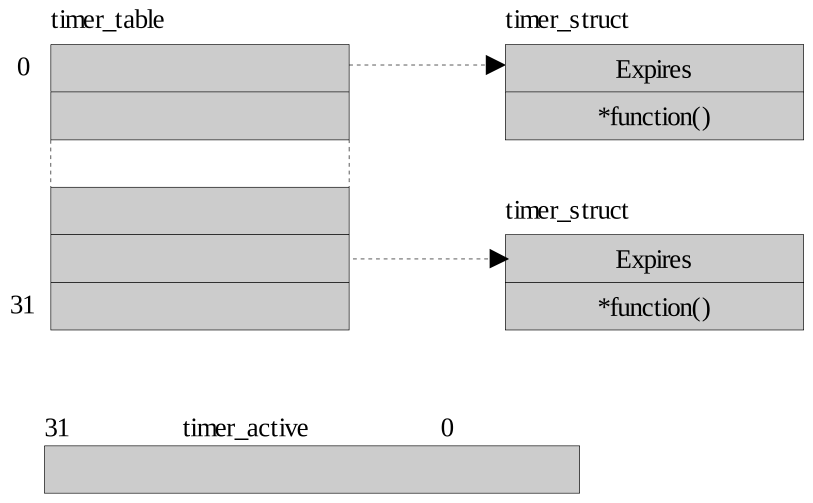
Figure 3.1: Bottom Half Handler data structures

Table 3.2: Bottom Half Handlers	
TIMER	This handler is marked active every time periodic timer interrupts, used to drive kernel's timer queue mechanism
CONSOLE	This handler is used to process console messages
TQUEUE	This handler is used to process *tty* messages
NET	This handler handles general network processing
IMMEDIATE	This handler is used by several device drivers

Every operating system needs a programmable interrupt timer because it needs to schedule an activity in the future. Also a mechanism is needed to schedule activities at a relatively precise time. A periodic interrupt is called a system clock tick and it controls the system's activities.

Linux measures time in clock ticks since the system is booted and all system times are based on this measurement called *jiffies*. There are two types of timer mechanism in Linux. An older is a static array of 32 pointers to timer structure and a mask of active timers [Figure 3.2]. A new recent mechanism is a linked list of data structures that are in ascending expire time order [Figure 3.3]. Every time system clock tick occurs, the timer bottom half handler is marked active so when the scheduler runs, the timer queues will be processed.
Often a process must wait for a system resource and that can take a lot of time.

Also the waiting process may use a lot of CPU time doing nothing. The solution to this is Wait Queues. Where , when the process starts to wait for some resource



Picture 3.2: Old timer mechanism

or event it is put to the end of the wait queue. This process can be interruptible or un−interruptible. An interruptible process may be interrupted while it is waiting in the wait queue. When the waiting process continues to run it first removes itself from the wait queue. Wait queues can be used for synchronize access to system resources.

Data structure or codes can be protected by Buzz Locks (Spin Locks) or semaphores. A buzz lock is a single integer field that is used as a lock, it allows just one process to access data structure or code at the same time. When the process attempts to enter some critical section it changes the lock's value from 0

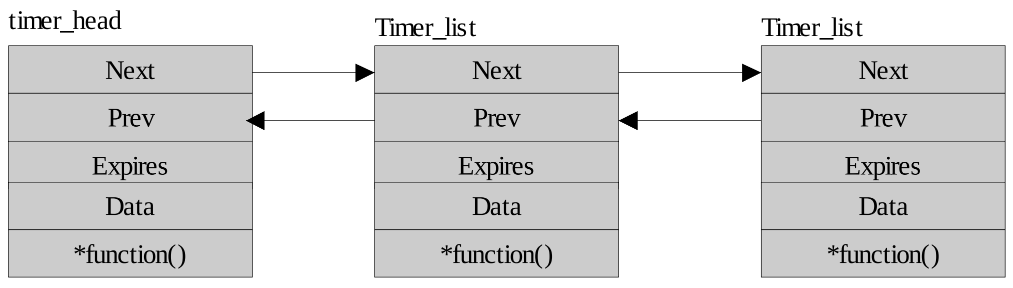


Figure 3.3: New timer mechanism

to 1. If the value is already 1 it tries again and again in a tight loop. When the process leaves a critical section it changes the lock’s value from 1 to 0. The accessing memory location of the lock is atomic so it cannot be interrupted by any other process. When the lock changes from 1 to 0, the first process that reads the 0 can access the critical section. Semaphores are a more sophisticated way to use critical sections. Semaphore data structure contains following fields:

• Count	Keeps the count of the processes wishing to use the resource.
• Waking	Count of the sleeping processes waiting for the resource.
• Wait queue	Processes that are waiting in the queue.
• Lock	A buzz lock used for accessing the waking field.

The count field initial value is 1 and when some process wants to access the resource, it decreases this value by 1. Now the process owns the resource and other processes can not access it. When a process leaves a critical section it increases the count value by 1. If some other process wants to enter the critical section while there is already some other process it decreases the count value by one (from 0 to −1) and returns to the wait queue. When the first process leaves the critical section it checks the count and if it is less than 1 it awakens the next process in the wait queue.

3.4 Memory management

Memory management is one of the main functions of an operating system. The need for memory is almost always bigger than the actual physical memory size and therefore, a virtual memory system has been developed. It allows large address space, much larger than the actual size, and protected virtual address space to processes in way that they cannot affect other. Also a memory management subsystem allows fair share of the physical memory to processes and shared virtual memory which means that there is only one copy of a particular program in the physical memory and all of the processes that use it share it.

When a program is executed the processor first reads an instruction from the memory and then decodes it. During the decoding process the processor may need to fetch or store some data from or to the memory. Next the instruction is executed and the following instruction in the program is read. In a virtual memory system all of the addresses are virtual and they must be converted to physical addresses before use. The operating system contains a set of tables that consists of all the information needed for the conversion.

3.5 Interprocess communication mechanism

Processes use signals and pipes for communication with other processes and the kernel. Also the System V IPC (inter Process Communication) is supported by Linux. Signals are one of the oldest interprocess communication systems in Unix systems. They are used to signal asynchronous events to processes [Bar2000]. They can also be used to give control commands from shell to child process.
A signal can occur when you hit the keyboard or some error happens. There is quite a big set of signals that can be generated by the kernel or other processes if they have the privilege to do that. Table 3.3 show the signals used in test workstation.

Processes can ignore most of the signals if they want but they cannot ignore a *sigstop* or a *sigkill* signal. *Sigstop* causes the process to halt the execution and a *sigkill* causes the process to exit. Signals don’t have priorities so if two signals are generated at the same time they can be handled in any order. Processes may have their own handlers for the signals or they can send signals to thekernel which has default handlers.

Table 3.3: Linux signals

sighup	sigint	sigquit	sigill	sigtrap	sigiot	sigbus	sigfpe
sigkill	sigusr1	sigsegv	sigusr2	sigpipe	sigalrm	sigterm	sigchld
sigcont	sigstop	sigtstp	sigttin	sigttou	sigurg	sigcpu	sigxfsz
sigvtalrm	sigprof	sigwinch	sigio	sigpwr			

Pipes are used in shells to redirect the output of one command to the input of another command. These pipes are unidirectional byte streams when the commands cannot see the pipe so they act like they would normally do.

A simple pipe could be:

```
$ ls −lR | grep .txt > file.txt
```

The above lists all the filenames in the current directory and its sub directories ending with ".txt" to the file named *file.txt*. However, this is a very simple pipe and much more complicated pipe would consist many commands.

3.6 Interrupts and interrupt handling

Nowadays computers use a lot of different hardware to perform different tasks so there should be some way to use these devices together. They could be used synchronously but this would not be efficient. A better solution would be to use interrupts. For example when a device needs a bus time it could just set an interrupt and wait for the processor to serve that interrupt. Devices are connected to the interrupt controller (PIC) and the interrupt controller is connected to processor. That saves the interrupt pins on the CPU. Usually a Real Time Clock (RTC) is hardwired to the CPU so it can not be masked. Interrupt controllers have a mask and status registers and they can mask interrupts by setting bits in the mask register and status register shows currently active interrupts. Figure 3.4 shows a typical construction of the interrupt hardware.

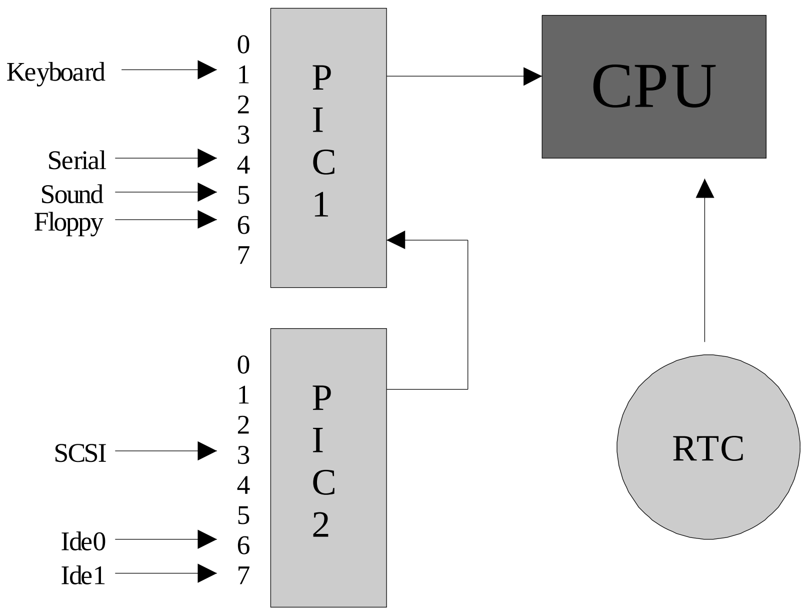


Figure 3.4: Interrupt hardware

Each system has a different interrupt routing mechanism so the operating system must be flexible enough to work with these. For example some of the processor’s interrupt pins could be connected to a PCI slot where different hardware (ethernet card, SCSI controller etc.) could be connected so flexibility is needed.

There are two ways to handle interrupts in a CPU. The first is to use an *interrupt mode* where interrupts are masked out and only one interrupt is served at a time. The other way is to use priorities where higher level interrupts may interrupt lower level ones, when a lower level interrupt is served. The better way is much more complicated because a CPU status must be saved before serving a higher level interrupt. Also the high level interrupt handling code must be carefully written.

3.7 Device drivers

One main purpose of the operating system is to hide the hardware devices from the user. This can be done by using device drivers that simplify access to these devices. The CPU is not the only intelligent part of a computer and therefore a common interface is required in order to use the different hardware controllers and their control and status registers (CSR). It is not possible to write drivers inside every program so Linux kernel contains the drivers. This makes it possible for all hardware devices to look like regular files.

Linux supports three different types of devices: character, block, and network. Character devices are read and write without buffering (serial port */dev/cua0*). Block devices can be written to and read from in multiples of block size and they are accessed via a buffer. Block devices are the only ones that support a mounted filesystem. Network devices use a BSD socket interface [see next chapter].

All device drivers share same common feature:

• Kernel code	Drivers are part of the kernel code.
• Kernel interfaces	Drivers must provide standard interface to kernel.
• Kernel mechanism	Drivers use standard kernel services (memory allocation, and services interrupt delivery, wait queues).
• Loadable	Drivers can be loaded when needed and unloaded when no longer needed.
• Configurable	Configuration of the drivers to be included within the kernel can be changed before compiling the kernel.
• Dynamic	At the system boot each device driver is initialized and it looks for the hardware and if it does not exist it causes no harm except uses little of system memory.

3.8 Networks

Linux has always been an network operating system. It was developed using Internet and WWW to exchange information between developers and users. Linux networking is modeled to 4.3 BSD compatible and it supports BSD sockets with some extensions and full range TCP/IP networking [Figure 3.5]. This interface was selected because it is very popular and it helps to port applications between Linux and Unix platforms.

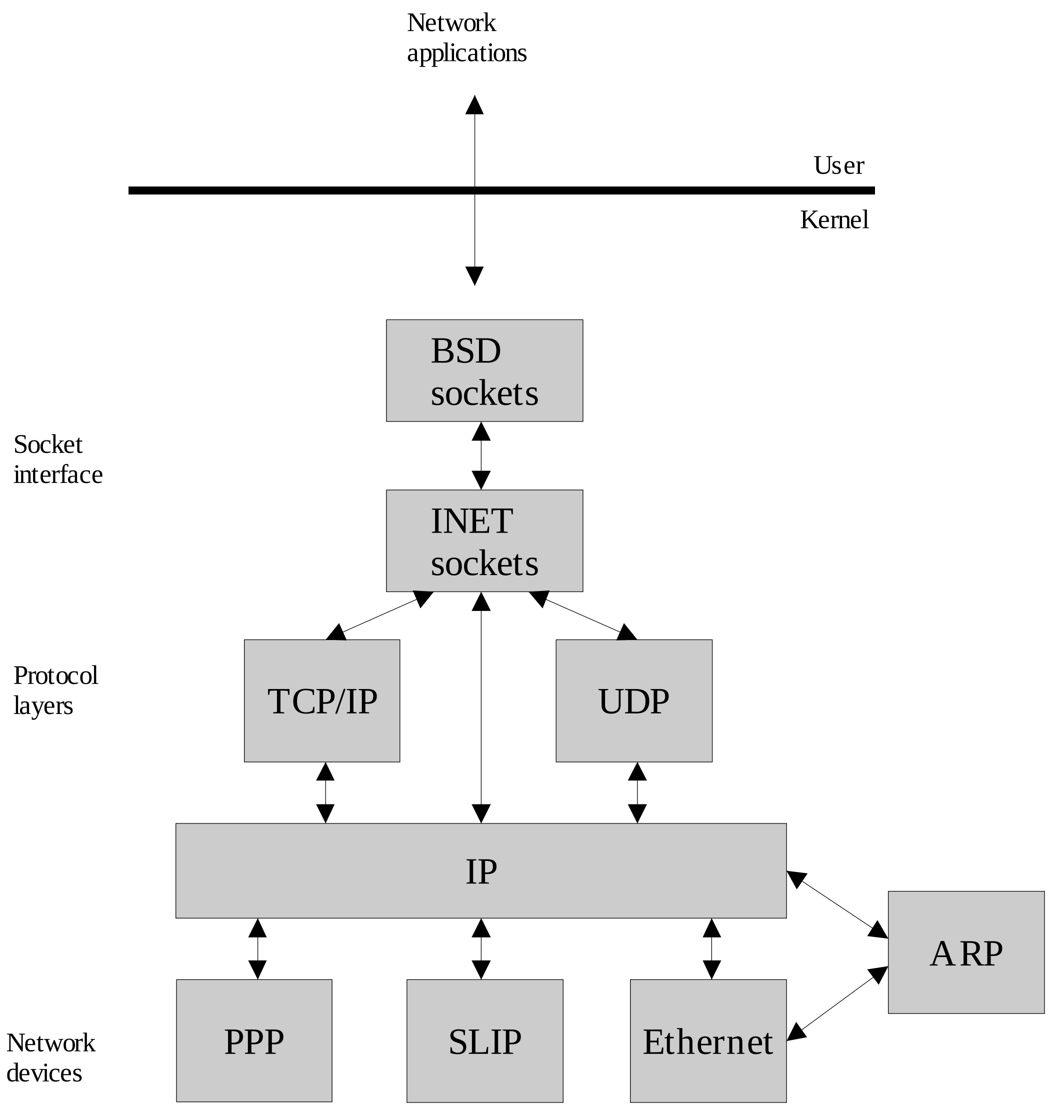


Figure 3.5: Linux network layers

A BSD socket interface is a general interface which supports various networking protocols and interprocess communication mechanisms. Each communication process has its own socket that describes the end of the communication link.

Socket may be said as a special case of pipe but the limit of data that it can contain is unlimited. Different classes of socket are called *address families* and Linux supports the following socket address families:

• INET	The Internet address family (TCP/IP)
• UNIX	Unix domain socket
• AX25	Amateur radio X25
• IPX	Novell IPX
• APPLETALK	APPLETALK DDP
• X25	X25

Linux also supports many socket types. Not all the address families support all types of services. The different socket types are:

• Stream	Reliable two way sequenced data stream, no lost, corrupted or duplicated data, supported by TCP/IP in INET address family.
• Datagram	Two way data transfer, data can be lost, duplicated, corrupted or arrive in the wrong order, supported by UDP in INET address family.
• Raw	Allows direct access to the underlying protocols for example a raw socket to an ethernet device to see raw IP data traffic.
• Reliable Delivered Messages	This is like datagram but the data is guaranteed to arrive.
• Sequenced Packets	This is like stream socket but thedata packet size is fixed.
• Packet	This is not a standard BSD socket type and it allows processes to access packets directly at the device .

3.9 Modules

Kernel modules are usually written in C−language and compiled to object code (for example *module.o*) and not to executable. Modules must have the following components:

• int init_module(void)	This function is invoked when the module is loaded.
• void cleanup_module	This function is invoked when module is removed.
• main_function	This function contains the program code.

A real−time module must also include:

• #include rtl.h	This is a real−time header.
• #include rt_sched.h	This is a real−time scheduler.
• #include rt_fifo.h	This is a real−time fifo between user and kernel tasks.

Appendix 1 contains an example of a simple real−time kernel module.

4. EMBEDDED REAL−TIME LINUX

The main aim of this master’s thesis was to find out which real−time Linux extension should be used. After some research RTLinux was chosen because it prooved to be ready and working well. There are however, other real−time Linux extensions for example RTAI which appear to be as good as RTLinux but the tests were made with RTLinux because it is more stable. Also some commercial products were available but this study concentrated only on a non−commercial operating system.

There is one big difference between RTLinux and RTAI and it is stability. Developers of RTAI implement new features to the system quickly and the stability of the system may become quite obscure. But developers of RTLinux are developing a stable and conservative system so they test all new features carefully before implementing them to RTL.

RTLinux is a small patch file which causes little changes to the original Linux kernel. When no real−time modules are loaded Linux works like before, but when you load a real−time module the real−time kernel pushes the normal kernel up and a new layer comes between the interrupt hardware and the Linux kernel. The *cli()* and *sti()* functions are replaced with *s_cli()* and *s_sti()* functions in Linux code. Figure 4.1 shows the basic structure of the RTLinux [Esp1998].

Difficulties arose when selecting the proper hardware. One basis was to choose cheap hardware. The Intel processor was selected because it was cheap and it works well with RTLinux. Some other processors are also supported (by RTLinux) but the Intel processor fits to this studies’ purposes well. Embedded systems usually run without a hard drive so the hardware needs to be selected so that it supports some other mass storage where startup files can be stored. Of course the hard drive makes it easier to build the system but after the system is working the hard drive is not necessary anymore. AMPRO’s Little Board PC supports a FLASH IDE disk and has a lot of peripherals integrated on the board and last but not least it is quite cheap so it was chosen to be used in this study.

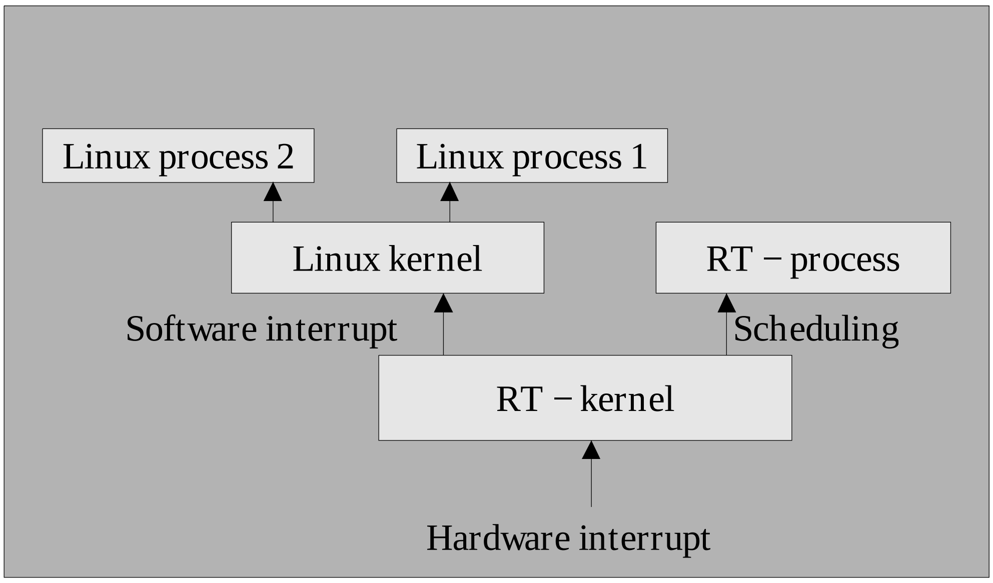


Figure 4.1: Structure of RTLinux

4.1 Hardware

The embedded system that was used was AMPRO's Little Board/P5x.
It includes:

- Pentium 166 'Tillamook' processor
- 64 MB DRAM
- 512k synchronous secondary cache
- embedded−PC BIOS in Flash EPROM
- 4 buffered serial ports (with RS−232, RS−485, TTL options)
- 2 Universal serial bus (USB) ports
- IrDA port
- multi−mode IEEE−1284 parallel port
- floppy controller
- dual PCI−bus EIDE/UltraDMA drive controllers
- flat−panel/CRT display controllers

- Adaptec UltraSCSI controller
- built−in Adaptec SCSI BIOS
- ethernet 100BaseT LAN interface
- Compact Flash solid−state IDE disk 32 MB
- standard PC keyboard and speaker interfaces

Plus:

- Samsung SyncMaster 15GLe monitor
- Unikey keyboard
- Logitech mouse
- Quantum Fireball 8 GB hard drive

4.2 Booting

Figure 4.2 shows what is needed to boot an embedded computer from a Flash disk and then run it from a RAM disk. When building an embedded system it is advised not to use any hard drives because they are very vulnerable to errors (power failure and so on). Also using a RAM disk is much faster than using hard drives.

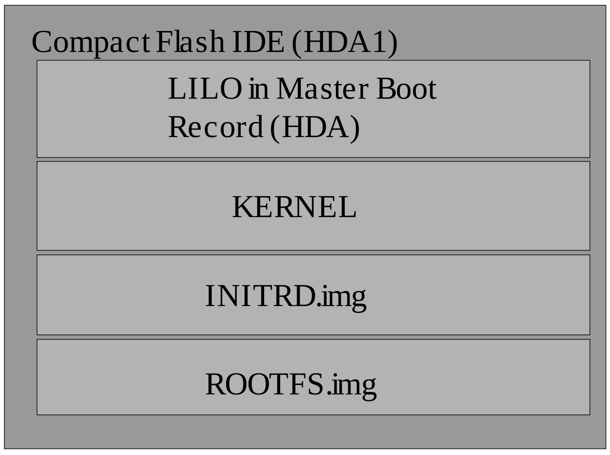


Figure 4.2: Files needed in a FLASH disk

4.2.1 Boot sequence

Lilo is in the first sector of the HDA (FLASH disk) which is called the Master Boot Record (MBR). When the HDA is set to the boot drive *lilo* boots from MBR. Next *lilo* loads a compressed kernel image and uncompresses it into a ram disk and boots it. When the kernel is booted it loads the initial ram disk (INITRD.img) and then mounts it as a root. Linuxrc script is executed and after that initial ram disk is unmounted and ramdisk (*/dev/ram)* is mounted as a root. Now the system boots as a normal system except only ram disk is used.

4.2.2 Initrd.img

Initial ram disk contains following directories and files:

• **bin**	All binaries used in the linuxrc must be here and they must be statically linked.
• mount	
• umount	
• zcat	
• sh	
• **dev**	All devices that are used must be here.
• hda1	
• ram	
• etc...	
• **etc**	
• mtab	Shows currently mounted filesystems (this file is empty and it is maintained by the mount command when it is executed).
• **lib**	**I**f statically linked binaries do not exist then you need to put some libraries here (these libraries are pretty large so prefer statically linked binaries).

•	**mnt**	This directory is the mount point for mounting /dev/hda1 during bootup.
•	linuxrc	This is a script which is executed during bootup. Here is a sample linuxrc:

```
mount  −t ext2 −o ro /dev/hda1 /mnt
zcat /mnt/boot/ram30.img.gz > /dev/ram
umount /dev/hda1
```

this script does the following:

1. Mount flash disk (/dev/hda1) which has a ext2 file system at the mount point (/mnt) to be read−only.
2. Uncompress and copy the root filesystem image (ram30.img.gz) to /dev/ram.
3. Unmount the flash disk (/dev/hda1).

4.2.3 Rootfs.img

The root filesystem must contain all files and directories that are needed to run RTLinux. Because there is 64 MB of RAM available, 30−40 MB can be used for the root file system and the rest will be enough for applications. It is still hard to reduce the root file system’s size to the level in Red Hat 6.0. Possibly some other distribution of Linux would have been better for building an embedded system but Red Hat was familiar so it was used. Also some ready embedded versions exist that are only a few hundred kbytes so they would have been much better versions to use. However, one aim at this study was to know how hard it is to embed Linux.

4.3 Installing real−time extension to Linux

First thing to do is to get a clean kernel source and a real−time patch to the kernel. The kernel source must be patched and configured and then a support for the hard real−time must be selected. When the new kernel is compiled configure *lilo* and rerun it. Then reboot your computer. After reboot RTLinux exists and you can start making your own real−time programs.

Firstly all of the examples programs were checked. Some of them did not work but a few of them did so there was something to study. First example that was examined was named Frank (see code from appendix 2).

On the real−time side (frank_module.c) it creates five real−time fifos (lines 75−80), two real−time thread (lines 82−90) and one handler (line 92) for one of the fifos. Handler (lines 42−57) is a function which will be executed every time there is data in the current fifo (from user space process to real−time task). It reads control messages from that fifo and sends it to the right thread through the other fifo. It also wakes up the thread when it is not yet running. The thread reads the fifo if there are control messages and writes data to the other fifo in the user space process.

The user space process (frank_app.c) starts with sending control messages (lines 43−55) to two different threads through the fifo and handler, and then starts waiting for data from the two fifos (lines 57−77). When data is in the fifo process reads it and prints it out on the screen. When reading loop is executed the process sends STOP messages to the threads and ends. The threads will be in suspend mode until they are again awakened.

4.4 Basic functions in real−time programming

There are not many functions you need to know before you can do something of your own. The most important functions are creating fifo and controlling threads.

The following gives a brief list of these functions with a short explanations.

Real−time functions:

• rtf_create(fifo_number, fifo_depth)	This creates a real−time fifo.
• pthread_attr_init(&attr)	This initializes a thread attribute object *attr* with the default value for all of the individual attributes used by a given implementation.
• sched_param.sched_priority(priority)	This sets a scheduling priority.
• pthread_attr_setschedparam scheduling (&attr, &sched_param)	This sets thread attribute object priority.
• pthread_create (&tasks[0],	This creates a new thread (task[0]) with &attr, thread_code, argument) attributes of *attr* using *thread_code.*
• rtf_create_handler(fifo_number, code)	This creates a handler for real−time fifo and executes *code* every time there is data in the fifo.
• rtf_destroy(fifo_number)	This destroys real−time fifo.
• pthread_delete_np(task[0])	This deletes thread (task[0]).
• rtf_get(fifo_number, &msg, sizeof(msg))	This reads data from fifo to struct msg.
• rtf_put(fifo_number, &msg, sizeof(msg)	This writes *msg* to fifo.
• pthread_wakeup_np(task[0])	This wakes up thread (task[0]).
• pthread_wait_np()	Thread waits for next period.
• rtl_printf()	This is real−time printf (takes about 250 micro seconds to execute!).
• pthread_make_periodic_np (task[0], gethrtime(), time_period)	This marks task[0] ready for execution after a *time_period*, *gethrtime* returns time in nanoseconds since the system bootup.

• pthread_suspend_np(task[0])	This suspends executions of thread until pthread_wakeup_np is called.

Next functions are needed in the user space processes to get something working:

• ctl=open(dev_name, arguments)	This creates file descriptor *ctl* for device (usually /dev/rtfX).
• write(ctl, &msg, sizeof(msg))	This writes data *msg* to device *ctl.*
• FD_ZERO(rfds)	This clears read file descriptor set *rfds.*
• FD_SET(ctl, &rfds)	This sets *rfds* to *ctl.*
• SELECT(FD_SETSIZE, &rfds, NULL, NULL, tv)	This waits for *rfds* to change status for *tv* time.
• FD_ISSET(ctl, &rfds)	This checks if *ctl* is set inside read file descriptor set *rfds.*

It can be seen above that there are really not too many functions to memorize before you can create your own real−time programs. Of course you must know how to write in C−language therefore a knowledge of C programming would be advantageous before starting real−time programming.

4.5 Communication between real−time tasks and non real−time tasks

A Linux kernel can be preempted by a real−time task and therefore real−time tasks cannot call any Linux routines safely. However, there must be some communication method [Bar1997] between these two different tasks because a real−time task cannot use I/O devices such as hard disk to save data. Also real−time task does not have functions for displaying for example graphics or even text. There is one function, *rtl_printf,* which is similar to ordinary printf but it is so slow that it is better not to use it in real−time tasks. It was noticed that *rtl_printf* can easily take 250−300 micro seconds just to print one word and that is too much if you need a task running in 50 µs periods.

There are at least two ways of handling communication between real−time and non real−time tasks.

4.5.1 Real−time fifo

Simple FIFO buffers are used in RTLinux for moving data between the rt−process and the Linux process [Figure 4.3]. These buffers are called real−time fifos and are allocated to the kernel address space. The FIFO must be opened from the real−time side and when it is opened it looks like a file to Linux processes. There are few basic functions that are needed to handle FIFOs.

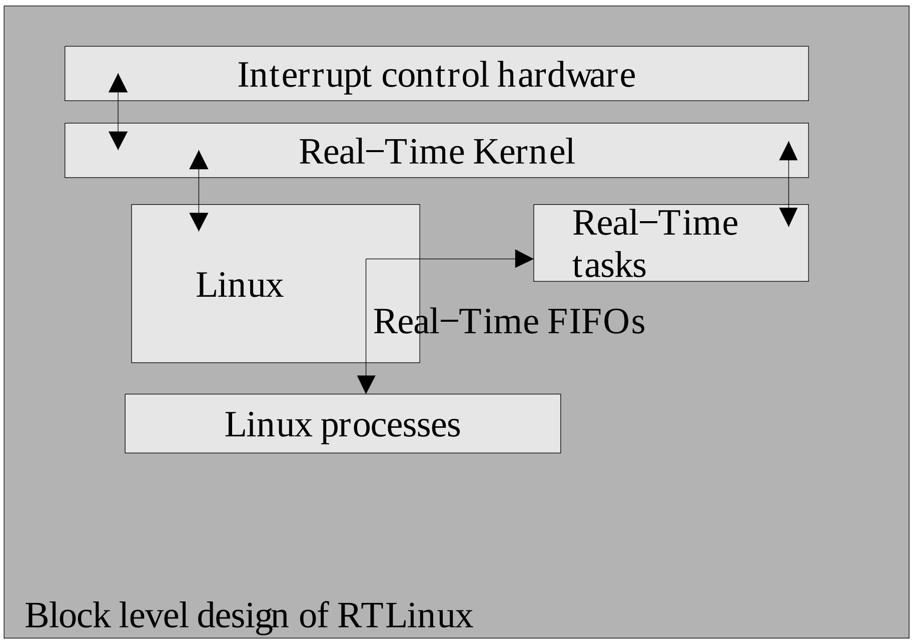


Figure 3.3: Real−time FIFO

Functions used in init_module and cleanup_module:

- rtf_create

- rtf_resize
- rtf_destroy
- rtf_create_handler

Functions used in rt_tasks and handlers:

- rtf_put
- rtf_get
- fifo_setup_sleep
- fifo_wake_sleepers
- fifo_irq_handler

Functions used in user tasks for file operations:

- open
- close
- read
- write

Read and write functions are atomic and non−blocking on the real−time side. This avoids the priority inversion problem [Leh1990]. Linux processes see rt−fifos as a character device. You can create a handler in a rt−process to handle data in fifo so there will be no need tocontinuous loop polling fifo which weakens the performance of the system. In Linux process you can use *select* function to wait for new data. Real−time FIFOs are fast and it takes only 20 µs to send data to a fifo and get it back from the other fifo (round−trip). You can use fifos also between two rt−tasks. The data transfer rate is typically over 100 MB/s in a modern x86.

4.5.2 Shared memory

Shared memory is a block of memory that is not used in user processes. It is defined in boot time and determined in a */etc/lilo.conf* file. Command *append="mem=63m"* sets a shared memory block above 63 Mbytes. If you have 64 Mbytes of memory it will be a block, 1 Mbyte in size. A Pentium class processor can handle 4 MB of shared memory and earlier processors only 1 MB and that depended on how big the memory page size was.

A user space process needs the physical memory to be mapped to its private address space and this can be done using the following commands. Firstly the file descriptor is opened and set to point to the device. This can be done with the *open* command and */dev/mem.*

```
fd = open("/dev/mem", O_RDWR)
```

Fd is an integer and it becomes the file descriptor that points to */dev/mem*. After this shared memory must be mapped to the user space address using the *mmap* command.

```
ptr = (SOME_STRUCT *) mmap(0, sizeof(SOME_STRUCT),
        PROT_READ | PROT_WRITE, MAP_FILE | MAP_SHARED,
        fd, BASE_ADDRESS);
```

Ptr is a type of SOME_STRUCT and can be used to point to shared memory. The Rt−process can access shared memory simply by making the pointer to shared memory.

```
ptr = (MY_STRUCT *) BASE_ADDRESS;
```

There is a big problem using shared memory with older kernels (2.0 and older). When memory is allocated the kernel uses a *kmalloc* function and that allocates only contiguous blocks of memory and the maximum block size is limited to 128 kB and in practice it is hard to allocate even 16 kB of memory. That is fixed in newer kernels and they use a *vmalloc* function and the virtual address space so block size does not matter.

4.5.3 MBUFF

There is also another way to use shared memory [Mot1999]. Tomasz Motylewski has written a driver for shared memory and it is called MBUFF. It uses a *mbuff.o module* and a */dev/mem* device. Shared memory (MBUFF) does not need to be reserved at system start up and it is not limited by memory paging. This shared memory is logically contiguous and it can not be swapped out so it fits perfect for real−time purposes. Half of the main memory can be shared.

4.5.4 Watchdog mechanism

If a non real−time system is used there is no need to be concerned about a process crashing and using all the resources of the processor because the operating system will share the resources and handle the time sharing. Usually we have a system which shares the processor time in time slices and each process has its own slice in the processor. When the time slice is over the next process goes into the processor and the previous begins waiting for the next time slice. This principle is used in non pre−emptive systems.

If we have a system that uses pre−emptive time sharing it is little bit different. In pre−emptive systems processes have different priorities and a higher priority processes get the processor before lower priority processes. If a lower priority process is in the processor when a higher priority process wants the processor the lower priority process is pre−empted and the higher priority process gets the processor.

In RTLinux, we have a real−time kernel that coexists with a basic Linux kernel. All interrupts are handled by the real−time kernel and are passed to the Linux task only when no real−time tasks are running. The basic principle is that the Linux kernel is only an idle task to the real−time kernel. This means that when there are no real−time tasks running the Linux kernel and non real−time processes can run.

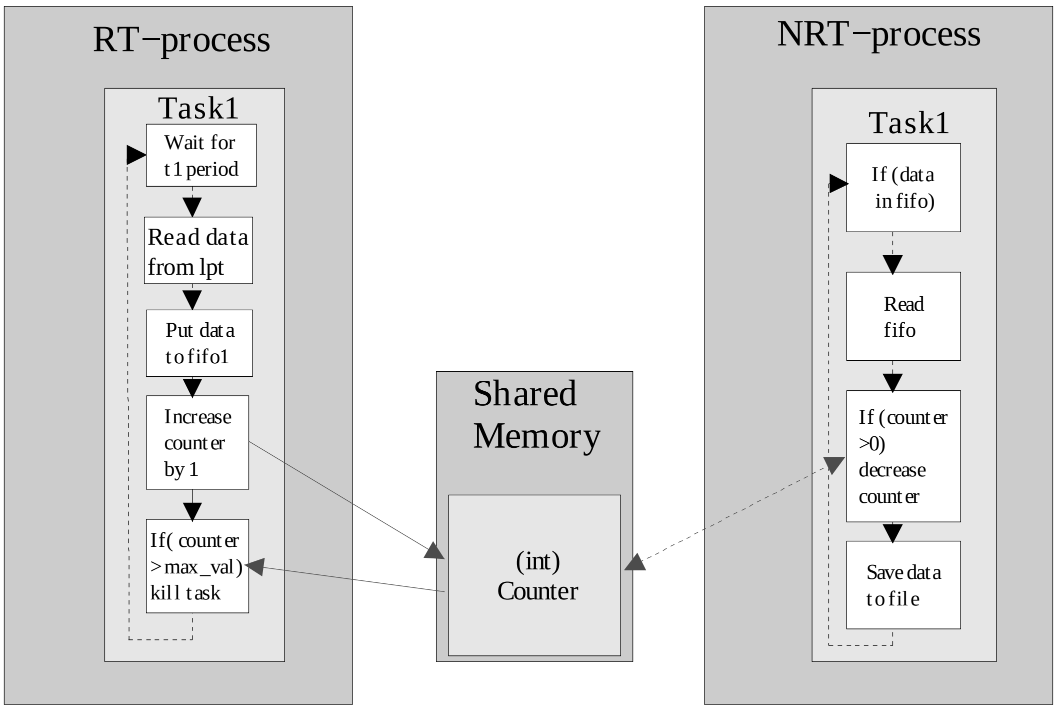


Figure 4.4: Watchdog system

If the real−time task is running it can not be pre−empted before it is over [Yod1997].

When we have a real−time process that is periodic and the period time is close to the hardware limit we have to be concerned what happens if the limit is somehow passed. In RTLinux the process will run in the processor and no other processes can have the processor before it is stopped. This means that the computer is jammed from the user point of view because peripherals (keyboard, mouse etc) will not work. The solution is the watchdog mechanism will stop the real−time task if it uses all the resources.

In this case there is a variable in the shared memory that can be modified by both real−time and non real−time tasks [Figure 4.4]. When a real−time task reads data from a printer port and puts it into a fifo it also increases the counter. And every time a non real−time task reads the value from the fifo it decreases the counter. If the real−time task uses all the processor time and the non real−time process does not run, the counter will increase and when some predefined limit is passed the real−time process will be suspended or kill itself.

Another way to run the watch dog mechanism is using a timer which sends an interrupt to the real−time process if it reaches some predefined value. The interrupt will kill or suspend the real−time process. A non real−time process will reset the timer so if the real−time process does not use all the processor time the timer will not reach the limit and everything will work fine.

4.6 Timer

There are two different kind of timer modes in RTLinux, periodic and oneshot. An example will help to explain the difference between the two modes. If you have one task with a period of 1000 time units then you just have to set the clock to interrupt every 1000 time units. Or if you have task with a period of 300 time units, the clock is set to interrupt every 300 time units. This works well but if you have two tasks with different periods you might have problems. For example if you have tasks with periods of 427 and 978 time units you can set the clock to interrupt every 200 time units so after the second interrupt you have to wait 27 clock ticks and then execute the task. But for the second task you have wait 4 interrupts and then wait 178 clock ticks to execute the task. This results in a lot of wasted time and therefore RTLinux uses a different solution to solve the problem, oneshot mode. In oneshot mode the timer is dynamically reset after the interrupt by a scheduler. So if the timer is set to interrupt after 427 time units it can be reset after that to interrupt after 551 time units to get an interrupt after 978 time units. This allows for both tasks to get their own interrupt in time.

Timing accuracy depends on which mode is used. In periodic mode is the measured in periods but in oneshot mode timing accuracy is about 2.5 ns.

Table 4.1: Timer modes

Mode	***Advantage***	***Disadvantage***
Periodic	More efficient because there is less timer re−programming	The tasks must be multiple of a base period
Oneshot	Task periods can be unrelated	Need more timer re−programming so the maximum task frequency is lower (10 μs re−programming delay)

You can use four different timer sources:

• Timer0	This is 100 Hz clock interrupt in Linux, oneshot or periodic in RTLinux.
• Timer1	This is used to refresh ram but not used for that purpose anymore.
• Timer2	This is connected to a speaker and it is responsible for the frequency of the peep
• TSC	This is Time Stamp Clock, Pentium internal counter, increments at the core frequency, can be accessed faster than 8253 timer chip and is more accurate and has less overflows

The above timers use memory area 0x44−0x5F and are also aliased to the memory area 0x40−0x43.

5. TESTING RTLINUX

The aim of this thesis was to find out if the RTLinux was capable of independent motor controlling functions i.e. is it fast enough to do that job or does it still need some special purpose processor (for example DSP) to cooperate with. Tests were made in a stand alone machine so there never was any motor attached to it but that was not the main idea. The processor’s own clock was used to measure the jitter and the maximum latency of the scheduled task and also a digital oscilloscope was used to collect same information.

5.1 Jitter measurement (with RTLinux example)

The jitter is the difference between the maximum and minimum time delay when starting a scheduled process or starting an interrupt service routine when the interrupt comes. There should not be much jitter in a real−time system because it causes non deterministic behavior in the system and that is what is to be avoided.

There is one testing program in the /*examples* directory which can be used for testing the jitter with a scheduled task. It measures time between scheduled time and actual execution time of a real−time task. This will give a pretty good picture of the jitter.

The measurements were made in two different modes, periodic and oneshot that is supported by RTLinux. Periodic mode did not give good results with this testing program because the calibration of timer chip 8253 against the CPU time stamp clock (TSC) was not precise and gave inaccurate results. So the tests were made in oneshot mode for this testing program. Figure 5.1 shows how big the jitter is when there is no other load than the measurement program. The real−time task does nothing else but measures the jitter so there is not much load from the real−time side either.

The first measurement was made with a task period of 50 µs. Other measurements were also made with other periods but there was no differences in the results. A faster processor was also tested (Celeron 466) and all the jitter tests showed that the jitter did not decrease even if a faster processor was used. It was concluded that depends on what hardware platform was used.

Figure 5.1: Jitter with no load

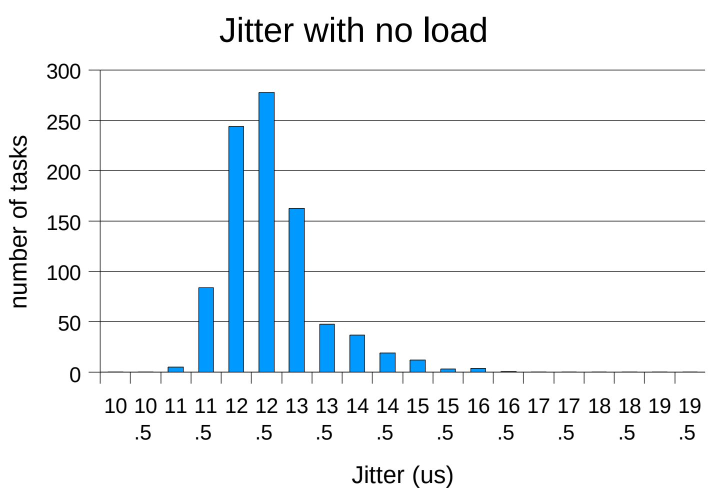
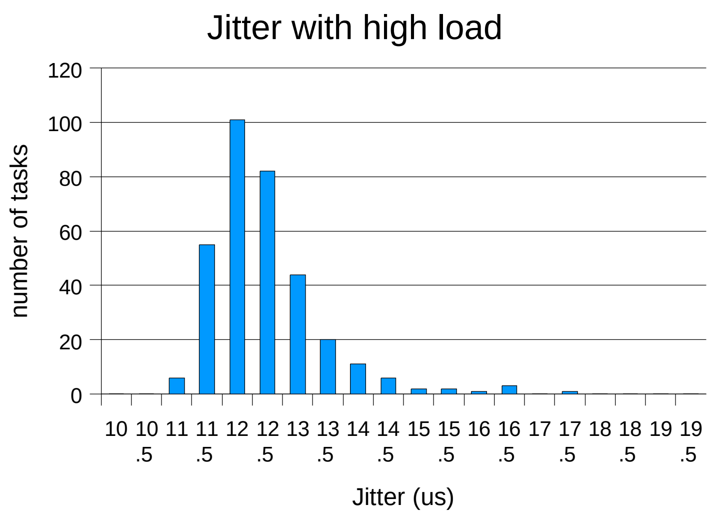


Figure 5.2 shows what happens when there is a high load on the Linux side. In this case a *grep* command was used and the CPU load was between 98−100 per cent. It can be seen that a high load on the Linux side has almost no affect on the jitter.

Figure 5.2: Jitter with high load applied



5.2 Jitter measurement (with own test program)

More information about how the RTLinux behaves with different loads on the real−time side was needed so a small test program was made. It was used to test RTLinux and the information was collected with a digital oscilloscope. The test program (Appendix 3) was a simple real−time module where three different tasks (lines 24−155) were scheduled to different periods and the jitter was then measured. Firstly the module was tested to determine if there was any affect when the task priorities were changed (table 5.1 rows 1 and 2). Also different modes and period times were tested and it was noticed that there was quite a big difference in the results. You can see there is a great results with the oneshot timer when the period times were not too high. The tasks were only simple one bit switching (lines 59−62, 105−108, 149−152) on a parallel port so there was no hard calculation. All measurements in table 5.1 were made with a Pentium 166.

Table 5.1: Jitter with simple tasks (Pentium 166)

Number of rt−tasks	Periods of task(s) (us)	Priorities of tasks	Jitter (us)	Mode	Timer period
3	25 50 100	1 2 3	18 57 277	Oneshot	
3	25 50 100	3 2 1	25 10 18	Oneshot	
3	25 50 100	3 2 1	27 29 20	Periodic	25 us
3	25 50 100	3 2 1	31 69 90	Periodic	10 us
3	50 100 150	3 2 1	45 94 145	Periodic	10 us
3	50 100 150	3 2 1	5 0 4	Oneshot	

Table 5.2 shows what happens when you add some calculation (lines 56−57, 102−103, 146−147) to real−time tasks. There were simple loops inside each task that did a simple addition operation. The first and fastest task did this addition a 100 times, the second 200 times and slowest task 300 times. The jitter become many

times bigger but if the period times of the were increased the jitter returned to normal.

Table 5.2: Jitter with 3 tasks (Pentium 166)

Number of rt−tasks	Periods of task(s) (us)	Priorities of tasks	Jitter (us)	Mode	Timer perioc
3	50 100 150	3 2 1	44 22 21	Oneshot	
3	50 100 150	3 2 1	47 99 65	Periodic	10 us
3	50 100 150	3 2 1	50 50 57	Periodic	50 us

Table 5.3 shows some measurements with a Celeron 466. It can be seen in the first row there was no calculations in the tasks but the rest of the cases were made with tasks that did some calculation and a bit switching.

Table 5.3: Jitter with simple tasks (Celeron 466)

Number of rt−tasks	Periods of task(s) (us)	Priorities of tasks	Jitter (us)	Mode	Timer period
3	25 50 100	3 2 1	2 4 33	Oneshot	
3	25 50 100	3 2 1	25 13 35	Oneshot	
3	50 100 150	3 2 1	4 43 30	oneshot	
3	50 100 150	3 2 1	50 50 57	Periodic	50 us
3	100 200 300	3 2 1	3 45 52	Periodic	50 us
3	100 200 300	3 2 1	5 5 41	oneshot	

The execution times for the these tasks used in these measurements were between 1.8−2.9 micro seconds. The fastest task lasted a maximum of 2.2 micro seconds, the second fastest, 2.4 micro seconds and the slowest 2.9 micro seconds. This did not affect the to jitter much because they stayed at the same level.

5.3 Interrupt service latency

When the interrupt service latency was tested a program that generates interrupt and serves it was used (Appendix 4). It was easy to measure latency with an oscilloscope because the parallel port was used to generate the interrupt. The system was simple because all that was needed was to connect one data pin (pin 9) to the interrupt pin (pin 10) and set the data pin (sets also interrupt) (line 35) with the scheduled task and reset the data pin with the interrupt service routine (lines 24−29). Next the time that the data pin was set was measured and the result was excellent because the interrupt latency was never bigger than 13 micro seconds. In fact latency was always between 9−13 micro seconds which is quite usable for different applications. When other tests were made with a Celeron 466 processor the interrupt service latency was only 6−9 micro seconds so it depends on how fast the processor is. However, a 6−9 micro second level is good enough for many electric drive systems so RTLinux could be used in some products with Intel hardware.

The time it takes to return from the interrupt service routine was also measured. With the Pentium 166 it was about 2 micro seconds and it did not change under any circumstances. There was no difference with a faster processor.

6. APPLICATIONS AND DEVELOPMENT ENVIRONMENTS

If RTLinux is ever going to be used in an electric drive system it must have good development tools and environments. It is not enough that you just have a good operating system but you must be able to do something with it. RTLinux has one advantage because you can use the same development tools that are used in Linux. This means that there is a lot of graphical tools that can be used so it is not necessary to use only basic text tools. There is also some tools developed to in RTLinux.

6.1 RTNET

RTNET is a implementation of the networking protocols used with real−time Linux extensions. It provides direct access to IP−based networking from a real−time code. It supports 3c59x and Tulip ethernet cards. Real−time drivers can be used as normal Linux networking drivers with minor loss of performance. Real−time drivers cannot exist simultaneously with normal drivers. Cards can be configured using *rtifconfig* which is similar to *ifconfig*. RTNET supports IP, ICMP and UDP protocols. The programming interface is identical to the standard socket interface and since RTNET is based on Linux networking it should be able to communicate with existing network stacks. RTNET is in beta stage but it is quite stable and could be used in applications [ftp://ftp.lineoisg.com/pub/rtnet/].

6.2 DSL

DSL is a C++ library for real−time dynamic system simulation under RTLinux. You can easily implement a block diagram of the system and generate code to simulate it in real−time. DSL provides basic classes to easily implement new components. Different parts of the system may have different scheduling, variable values may be logged and changed during simulation. DSL is in the beta stage.

6.3 RTLT

A Real−Time Linux Target is a Simulink based Real−time Graphical Control Environment. With RTLT you can implement real−time code from a Simulink block diagram to Intel based hardware. You don't need to do any C−programming because you can build your system with a Simulink front end. This is a commercial product [http://qrts.com/products/rtlinuxtarget/].

6.4 RTiC−LAB

Real Time Controls Laboratory is a Hard Real Time Controller Implementation and Simulation Environment for Linux and real−time Linux. It supports a few Computer Boards A/D and D/A cards. With RTiC−LAB you have real−time access to plant states, plant I/O, controller states, controller parameters (scalar or matrix) and hard real time environment for plant modeling and run time data can be saved to *stdio* or data file [http://128.143.47.231/~efh4v/rtic−lab.html].

6.5 COMEDI

A control and measurement device interface is a project to develop open−source drivers, tools, and libraries for data acquisition. It provides real−time support for a lot of hardware, high level library and application level device independence [http://stm.lbl.gov/comedi/].

7. CONCLUSIONS

When building a new embedded Linux operating system start with some existing embedded package, not with some full version of normal Linux. This study was started with a Red Hat and that was a big mistake. It took far too much time to get it working from a small flash disk because even the minimum installation was too big and contained a lot of non−usable files. Maybe some other version would have been more user friendly like Slackware but a ready embedded Linux such as MiniRTL is the best choice.

RTLinux version 2.0 was used which is not the newest version anymore but a same version was used for the whole study. Version 2.2 is the newest stable version and some newer unstable versions are also available. Version 2.2 supports SMP and offers POSIX style API. The best version to be used now is 2.2 because some bugs have been fixed and improvements have been made.

The performance of RTLinux was the main aim of this project and some measurements were carried out to determine the limits of the RTLinux. These measurements showed that RTLinux was not capable of controlling electric drives by itself and it still needs other processors to cooperate with it to handle the fastest control loops. If the application can handle interrupt service latencies of 10−20 micro seconds then there will be no problems with RTLinux. Interrupt service latencies were measured in different situations. Latencies between 9−13 micro seconds with a Pentium 166 and 6−9 micro seconds with Celeron 466 were measured. With some other hardware (PPC, Alpha) latencies should be smaller with RTLinux.

The next interesting subject was the jitter. If you need a deterministic operating system you don’t want to have big jitters. The size of the jitters were measured for redefined execution times to tasks (scheduled periodic tasks) with different time levels. It was obvious that a jitter gets bigger when the scheduling time approache the hardware limit. A jitter was just few micro seconds with a oneshot timer for long time periods. These tasks did not have much functionality. That means you are not moving near the hardware limit. Then when you get closer to the hardware limit the jitter gets bigger and the system does not work anymore within the time limits. RTLinux was a good real−time operating system if you used period times bigger than 100 micro seconds. But with other hardware, performance could be better than this.

RTLinux is still a young system that there are not enough different development environments and other tools available but every week someone releases something new and the situation is getting better. Linux is ported to dozens of different platforms and porting RTLinux to these platforms should not be too big challenge. Also there is quite many device drivers available to RTLinux and much more to normal Linux so that should not be a reason to reject RTLinux.

REFERENCES

[Bar1997] Barabanov, Michael. M.S. Thesis, A Linux based Real−time operating system. June 1997 (URL: http://www.rtlinux.org/rtlinux.new/documents.html)

[Bar2000] Bar, Moshe. The Linux Signals Handling Modes. Linux Journal, May 2000, p.48−52.

[Che1999] Chen, Zhu. Porting Linux to a PowerPC Board. Linux Journal, October 1999, p.58−62.

[Cec1999] Cecati, Carlo. Microprocessor s for Power Electronic and Electrical Drives Application. Industrial Electronics, vol. 46, no. 3, September 1999. http://sant.bradley.edu/~ienews/99_3/drCECATI/paper.htm

[Esp1998] Espinosa, Garc, Terrasa. Extending RT−Linux to Support Flexible Hard Real−Time Systems with Optional Components. Electrical Engineering, Volume 1474, Issue , p. 41−, October 1998

[Göt1999] Götz, Mangiagalli, Mäkijärvi, Perez. Embedding Linux to Control Accelerators and Experiments. Linux Journal, October 1999, p.64−71.

[Hon2000] Honkanen,Tuomo. Linux tulee sulautettuihin. Prosessori 2/2000 p. 48

[Hon1997] Honkanen, Tuomo. Windows NT reaaliaikaan. Prosessori 9/1997 p. 45

[Mot1999] Motylewski, Tomasz. MBUFF Manual (URL: ftp://ftp.rtlinux.com/rtlinux/v2/rtlinux2.0/rtl/drivers/mbuff/)

[Rus1999] Rusling, David A. The Linux Kernel. (URL: http://www.linuxdoc.org/LDP/tlk/tlk.html)

[Lec1999] Lehoczky, Rajkumar, Sha. Priority inheritance protocols: An Approach to real−time synchronization. IEEE Transactions on Computers, September 1990, p.1175−1185,

[Tol2000] Tolonen, Pekka. Netti pursuaa Linux−sovelluksia. MikroPC, 8/2000, p.44−49.

[War1999] Ward, Brian. The Linux Kernel HOWTO. June 1999 (URL: http://www.linuxdoc.org/HOWTO/Kernel−HOWTO.html#toc2)

[Web2000] Webb, Warren. Real−Time Software Reigns in Post−PC Products. EDN, February 2000, p. 95−102.

[Wil1999] Williams, Joel R. Embedding Linux in a Commercial Product. Linux Journal, October 1999, p. 50−57.

[Yod1997] Yodaiken, Victor. The RT−Linux approach to hard real−time. 1997 (URL://rtlinux.org/rtlinux.new/documents/papers/whitepaper.html)

[WWW1] http://www.linuxlinks.com/local/why.shtml

[WWW2] http://www.linux.org/info/index.html

[WWW3] http://www.seul.org/docs/whylinux.html