LINUX 2.6.33.7.2-RT30

Authors: Luc Perneel (1, 2), Hasan Fayyad-Kazan (2) and Martin Timmerman (1, 2, 3)

1: Dedicated Systems Experts, 2: VUB-Brussels, 3: RMA-Brussels

http://www.dedicated-systems.com

E-mail: info@dedicated-systems.com

EVALUATION REPORT LICENSE

This is a legal agreement between you (the downloader of this document) and/or your company and the company DEDICATED SYSTEMS EXPERTS NV, Diepenbeemd 5, B-1650 Beersel, Belgium.
It is not possible to download this document without registering and accepting this agreement on-line.

1. **GRANT**. Subject to the provisions contained herein, Dedicated Systems Experts hereby grants you a non-exclusive license to use its accompanying proprietary evaluation report for projects where you or your company are involved as major contractor or subcontractor. You are not entitled to support or telephone assistance in connection with this license.
2. **PRODUCT**. Dedicated Systems Experts shall furnish the evaluation report to you electronically via Internet. This license does not grant you any right to any enhancement or update to the document.
3. **TITLE**. Title, ownership rights, and intellectual property rights in and to the document shall remain in Dedicated Systems Experts and/or its suppliers or evaluated product manufacturers. The copyright laws of Belgium and all international copyright treaties protect the documents.
4. **CONTENT**. Title, ownership rights, and an intellectual property right in and to the content accessed through the document is the property of the applicable content owner and may be protected by applicable copyright or other law. This License gives you no rights to such content.
5. **YOU CANNOT**:
   - You cannot, make (or allow anyone else make) copies, whether digital, printed, photographic or others, except for backup reasons. The number of copies should be limited to 2. The copies should be exact replicates of the original (in paper or electronic format) with all copyright notices and logos.
   - You cannot, place (or allow anyone else place) the evaluation report on an electronic board or other form of on line service without authorisation.
6. **INDEMNIFICATION**. You agree to indemnify and hold harmless Dedicated Systems Experts against any damages or liability of any kind arising from any use of this product other than the permitted uses specified in this agreement.
7. **DISCLAIMER OF WARRANTY**. All documents published by Dedicated Systems Experts on the World Wide Web Server or by any other means are provided "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. This disclaimer of warranty constitutes an essential part of the agreement.
8. **LIMITATION OF LIABILITY**. Neither Dedicated Systems Experts nor any of its directors, employees, partners or agents shall, under any circumstances, be liable to any person for any special, incidental, indirect or consequential damages, including, without limitation, damages resulting from use of OR RELIANCE ON the INFORMATION presented, loss of profits or revenues or costs of replacement goods, even if informed in advance of the possibility of such damages.
9. **ACCURACY OF INFORMATION**. Every effort has been made to ensure the accuracy of the information presented herein. However Dedicated Systems Experts assumes no responsibility for the accuracy of the information. Product information is subject to change without notice. Changes, if any, will be incorporated in new editions of these publications. Dedicated Systems Experts may make improvements and/or changes in the products and/or the programs described in these publications at any time without notice. Mention of non-Dedicated Systems Experts products or services is for information purposes only and constitutes neither an endorsement nor a recommendation.
10. **JURISDICTION**. In case of any problems, the court of BRUSSELS-BELGIUM will have exclusive jurisdiction.

Agreed by downloading the document via the internet.

DOCUMENT .CHANGE LOG

Issue No.	Revised Issue Date	Para's / Pages Affected	Reason for Change
1.0	25 Mar 2011	All	Initial version
1.01	2 April 2011	some	Typo changes
1.02	14 April 2011	ALL	Rephrasing some statements and commenting
1.03	03 May 2011	ALL	idem
1.04	05 May 2011	ALL	Idem
1.05	13 May 2011	ALL	Almost Final
1.06	16 May 2011	ALL	Almost Final + font changes
1.07	30 May 2011	All	Final

1 Document Intention

1.1 Purpose and scope

This document presents the qualitative evaluation results of the real-time **Linux** operating system (Linux with its real-time patches). The testing results of this operating system employed on an x86 processors can be found on our website. (www.dedicated-systems.com)

The layout and the content of this report follow the one depicted in "The evaluation test report definition" [Doc. 3] and "The OS evaluation template" [Doc. 4]. See section 1.4 of this document for more detailed references. Therefore these documents have to be seen as an integral part of this report!

Due to the tightly coupling between these documents, the framework version of "The evaluation test report definition" has to match the framework version of this evaluation report (which is 2.9). More information about the documents and tests versions together with their corresponding relation can be found in "The evaluation framework" see [Doc. 1] in section 1.4 of this document.

1.2 Document issue: the 2.9 framework

This document shows the results in the scope of the evaluation framework 2.9.

1.3 Conventions

Throughout this document, we use certain typographical conventions to distinguish technical terms. Our used conventions are the following:

Throughout this document, we use certain typographical conventions to distinguish technical terms. Our used conventions are the following:

- ***Bold Italic*** for OS Objects
- **Bold** for Libraries, packets, directories, software, OSs...
- `Courier New` for system calls (APIs...)

1.4 Related documents

Those are the documents that are closely related to this document. They can all be downloaded using following link:
http://www.dedicated-systems.com/encyc/buyersguide/rtos/evaluations

Doc. 1 The evaluation framework
This is the evaluation framework document. It indicates the available documents, their names, numbers, versions and their corresponding relations. It is the base document of the evaluation framework.
EVA-2.9-GEN-01 Issue: 1 Date: April 19, 2004

Doc. 2 What is a good RTOS?
This document presents the criteria that Dedicated Systems Experts use to give an operating system the label “Real-Time”. The evaluation tests are based upon the criteria defined in this document.
EVA-2.9-GEN-02

Doc. 3 The evaluation test report definition
This document presents the different tests issued in this report together with the flowcharts and the generic pseudo code for each test. Test labels are all defined in this document.
EVA-2.9-GEN-03 Issue: 1 April 19, 2004

Doc. 4 The OS evaluation template
This document presents the layout used for all reports in a certain framework.
EVA-2.9-GEN-04 Issue: 1 April 19, 2004

Doc. 5 Linux 2.6.33.7.2-RT30 on an X86 platform.
This document presents the test results of the OS on a specific platform
EVA-2.9-TST-LNX-x86-104 Issue: 1 May 5, 2011

2 Introduction

This chapter talks about: 1) the OS that we are going to test and evaluate, 2) the real time patch integrated in this OS to achieve some real time performance and behaviour tests, 3) the library used for interaction between the testing applications and the kernel, 4) the CPUs that such OS supports.

2.1 Overview

The evaluation project started in 1995 and as such accumulates a long experience with different (RT) OS. Today more and more embedded systems are equipped with **Linux** solutions using more or less real-time patches. Different vendors like MontaVista, Windriver, and Lynuxworks have now **Linux** variants in their product portfolio.

Since the kernel version 2.4, a lot of improvements regarding real-time behaviour found their way into the standard "**Vanilla**" kernel. There is a well maintained real-time patch available (both have their origins from Ingo Molnar) called ***RT_PREEMPT*** patch. Remark that some real-time features (like priority-inheritance ***mutexes***, introduced in version 2.6.18) are already in the **Vanilla** kernel.

We believed that it is the time to test this kernel by our standard real-time behaviour evaluation framework and find out how well it behaves.

For this evaluation, we used the standard **glibc** library as the **µClibc** package does not include yet the Native POSIX Thread Library (**NPTL**). Moreover, **µClibc** also does not use ***futexes*** which means that it also does not have any support for priority inheritance (which must be available when considering real-time behaviour). Further **µClibc** uses internal protection systems (***mutex***, ***semaphores***) and signals for the ***pthread POSIX*** layer, which behaves differently while compared to the usage of direct **NPTL** calls. From a real-time point of view, using **µClibc** in its current form makes the kernel real-time support unavailable in user space. It has to be said that there is an active **NPTL** branch in the **µClibc** code base. Therefore we suspect that it is only a matter of time before it will become available in the official releases.

It is a pity that the **µClibc** wagon is not yet on the **NPTL** rail. Using the **buildroot**, **µClibc**, and **busybox** combo makes it easier to have an embedded **Linux** platform with a small storage footprint. But without the **NPTL** support, real-time applications cannot be used in user space, unless you use direct system calls to the kernel.

Remark that the ***RT_PREEMPT*** patch degrades throughput performance which means that it should be used only when your project has low latency requirements. This is normal and a fundamental rule in real-time software: latency improvements have a negative impact on throughput and vice versa. Some quick measurements using an NFS mount stressing network and disk showed a negative throughput impact between 5 and 10% by enabling the ***RT_PREEMPT*** patch!

2.2 Evaluated (RTOS) product

The operating system OS that will be evaluated is **Vanilla Linux** 2.6.33.7 with real-time patch v30.This RT patch was the latest version officially released by OSADL (the Open Source Automation Development Lab) on December 21, 2010. Being as OSDAL's latest stable release was our main reason for testing this version. The RT patches can be found at http://www.kernel.org/pub/linux/kernel/projects/rt/.

The evaluation of this kernel version (2.6.33.7.2-rt30) was performed using several performance and behaviour tests. The testing results are applicable only to this version as other versions may have other significant performance figures and behaviour.

The library used between the testing applications and the kernel is the **glibc** version 2.11.1 as mentioned before. This interfacing library is important because user applications (when using POSIX calls) can access the real-time features of the kernel only if this library supports them. Otherwise, direct system calls in user space applications are needed.

2.3 Supported CPU

☺ One of the advantages of using **Linux** OS is its support for most available CPUs around, although the stability can be sometimes an issue for less common-used platforms. As **Linux** is used a lot on x86 processors, it is very stable on these PC-like platforms. Also the ARM version is used a lot (portable/low power devices) and thus very stable.

For less-used platforms, this can however be an issue. For instance, we had to solve some problems for a customer running **Linux** on a MIPS based platform due to very sporadic crashes on that platform. We discovered that they were solved in more recent kernels. This also demonstrates that the open source community works. However, in real-life embedded software development, it is not a real option to update each month the kernel to a newer version.

Remark as well that some **Linux** behaviour can change seriously between versions! So being on the bleeding edge can be… indeed bleeding, be warned!

3 Evaluation Results summary

Remember that the tested and evaluated product is **Vanilla Linux** 26.33.7 with ***RT_PREEMPT*** patch v30. If correctly used and configured, the ***RT_PREEMPT*** **Linux** system has the internals to provide some real-time characteristics.

Compared with the traditional RTOS that supports also memory protection between processes, the worst case latencies in **Linux** ***RT_PREEMPT*** are still around 5 to 10 times slower (depending on the RTOS you compare with). Our study and measurements show the latencies are bound and therefore this **Linux** version may be labelled Real-Time.

TAKE CARE: Using a wrong driver or wrong configuration can destroy real-time behaviour. You need to follow the detailed rules described in this document to insure RT-behaviour.

3.1 Positive points

- No license fees
- Source code available
- Extensible

3.2 Negative points

- The real-time characteristics of the OS are present only when everything is configured and built correctly (and not for all drivers)
- GPL license is not completely free…
- Setting up a complete embedded target from scratch is a daunting task.
- **uClibc**, which is used a lot in embedded systems, does not have currently **NPTL** support and as such cannot provide real-time characteristics to the user level. Thus, **glibc** should be used.

3.3 Ratings

For a description of the ratings, see [Doc. 3].

RTOS Architecture	0	6	10
OS Documentation	0	4	10
OS Configuration	0	6	10
Internet Components	0	10	10
Development Tools	0	6	10
Installation and BSP	0	4	10
Test Results	0	4	10
Support	0	N.A.	10

Although [Doc. 3] gives a description of the ratings, comparison with other reports on other OS should help you to understand the scoring.

4 Installation and BSP

Installation and BSP	0	▨	▨	▨	4							10

Today, there are enough ***Linux*** *distributions available which makes it easy to install a host development system.*

However, building a complete environment for a custom board stays a daunting task if you do everything yourself (build cross-toolchains, libraries, kernel and setup root system). Furthermore you will need a boot loader (mostly u-boot is used). Luckily, most board vendors deliver their boards with all tools needed and preconfigured kernel / root file system.

4.1 Installation

Setting up an embedded **Linux** platform requires two nuts to crack:

- Configuring and adapting the kernel to your platform (setting up the Board Support Package or BSP).
- Configuring and creating the root file system.

Our main focus is this report is on the OS real-time behaviour. As an x86 platform is used, installing a standard **Linux** could be enough as a starting point. Also there is no need for the cross compilation tools. All this makes it easy to setup a system. However, creating a root file system for a custom board, then building everything from scratch is a daunting task. Our experience learned us that there are always some unexpected problems which require some considerable time to solve. Once you deviate from the most common used CPU (the X86) to the less used CPU, the more trouble one might expect.

4.1.1 Installation on Host

An **Ubuntu** 10.04 **Linux** workstation is used as a host for our testing, the installation of the OS was easy and quick without encountering any problems as such installation requires only a limited **Linux** Knowledge... Today most **Linux** distributions are easy to install. Only when you go to the “custom build” systems (e.g. **Gentoo**), you will need more time and **Linux** experience to set everything up.

4.1.2 Installation on target

On the target we also installed initially the **Ubuntu 10.04 Linux** distribution, but in this case a bare minimum install (using the alternate installer). This is already a bit more complex task as you need to use terminal text based menus.

Once the system boots and runs, we need only to copy the fresh built kernel from our host to the target boot partition. This home built kernel with the **PREEMPT_RT** patch was built in a way that all the required modules are built in the kernel itself. This makes it possible to boot the target without an ***initrd*** RAM disk.

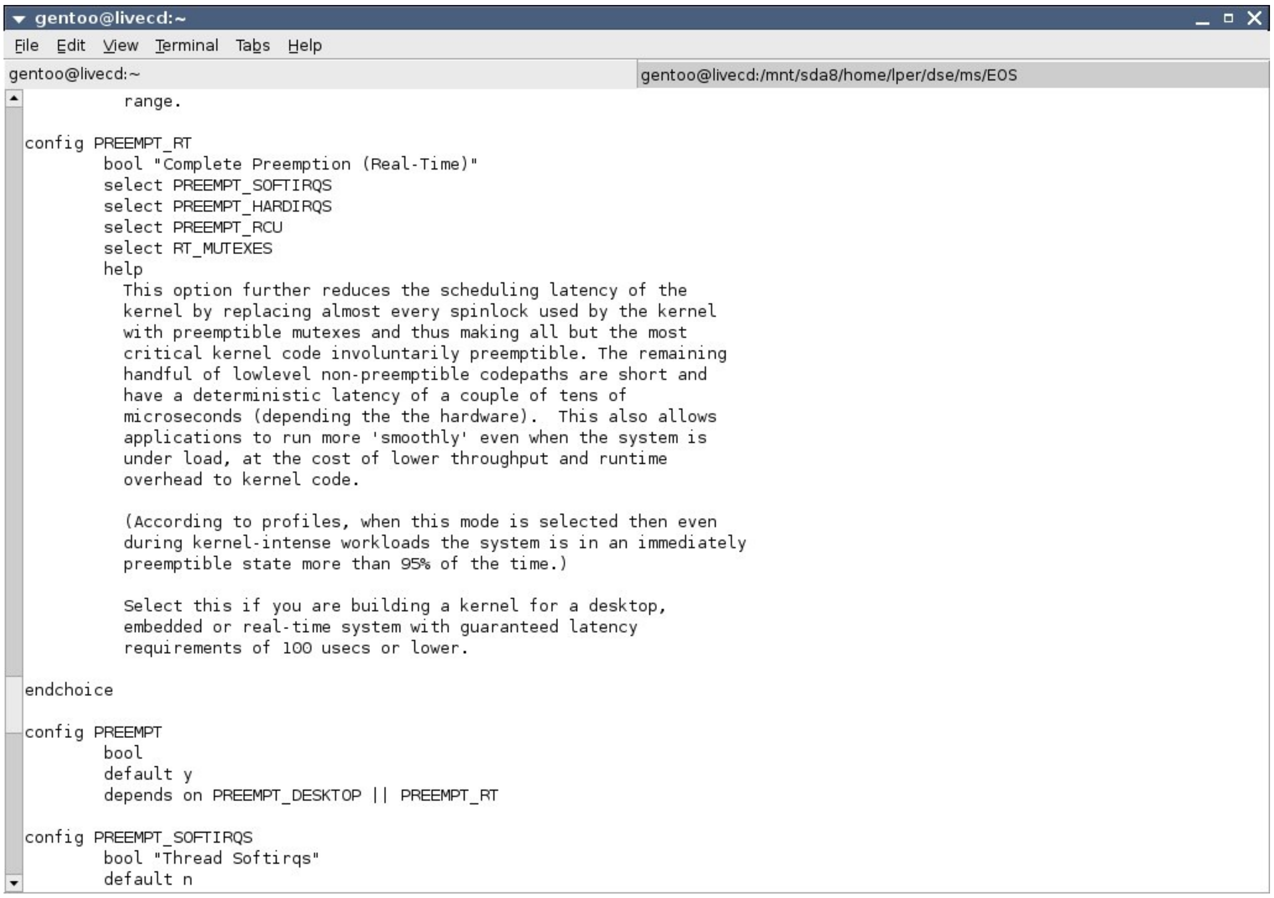
The ***init*** scripts of the target root file system are adapted in order not to load any other modules. This was done to avoid building these modules and storing them in the ***/lib/modules/*** (kernel version)/ directory. This procedure makes it easy to update the kernel.

4.2 Setting up the kernel configuration

Setting up the kernel configuration is needed to define how the kernel will behave. For example, will it be pre-emptable or not? Also, the kernel configuration should define the hardware it needs to support (linked in the kernel itself or by using loadable modules). Performing such things can be done via tools delivered with the **Linux** kernel source tree. Let us discuss these.

4.2.1 Kconfig

These configuration tools use a collection of ***Kconfig*** files (located in the different subdirectories of the source tree) where options, dependencies, default settings and even comments are stored. An example is shown in the figure 1.


gentoo@livecd:~
File Edit View Terminal Tabs Help
gentoo@livecd:~
gentoo@livecd:/mnt/sda8/home/lper/dse/ms/EOS

```
            range.

config PREEMPT_RT
        bool "Complete Preemption (Real-Time)"
        select PREEMPT_SOFTIRQS
        select PREEMPT_HARDIRQS
        select PREEMPT_RCU
        select RT_MUTEXES
        help
          This option further reduces the scheduling latency of the
          kernel by replacing almost every spinlock used by the kernel
          with preemptible mutexes and thus making all but the most
          critical kernel code involuntarily preemptible. The remaining
          handful of lowlevel non-preemptible codepaths are short and
          have a deterministic latency of a couple of tens of
          microseconds (depending the the hardware).  This also allows
          applications to run more 'smoothly' even when the system is
          under load, at the cost of lower throughput and runtime
          overhead to kernel code.

          (According to profiles, when this mode is selected then even
          during kernel-intense workloads the system is in an immediately
          preemptible state more than 95% of the time.)

          Select this if you are building a kernel for a desktop,
          embedded or real-time system with guaranteed latency
          requirements of 100 usecs or lower.

endchoice

config PREEMPT
        bool
        default y
        depends on PREEMPT_DESKTOP || PREEMPT_RT

config PREEMPT_SOFTIRQS
        bool "Thread Softirqs"
        default n
```

Figure 1: Kconfig contents

☺ These ***Kconfig*** files are well organised and use an internal tree structure of configuration options. The configuration tools use these configuration files to provide us with a menu structure or even a graphical interface for setting up the configuration.

When creating your own BSP for porting a **Linux** kernel to your platform, the best practice is to add an extra entry in these configuration files for this custom platform. This limits the dependencies between the port and the kernel (making it easier to merge later newer kernel releases).

4.2.2 "make" config

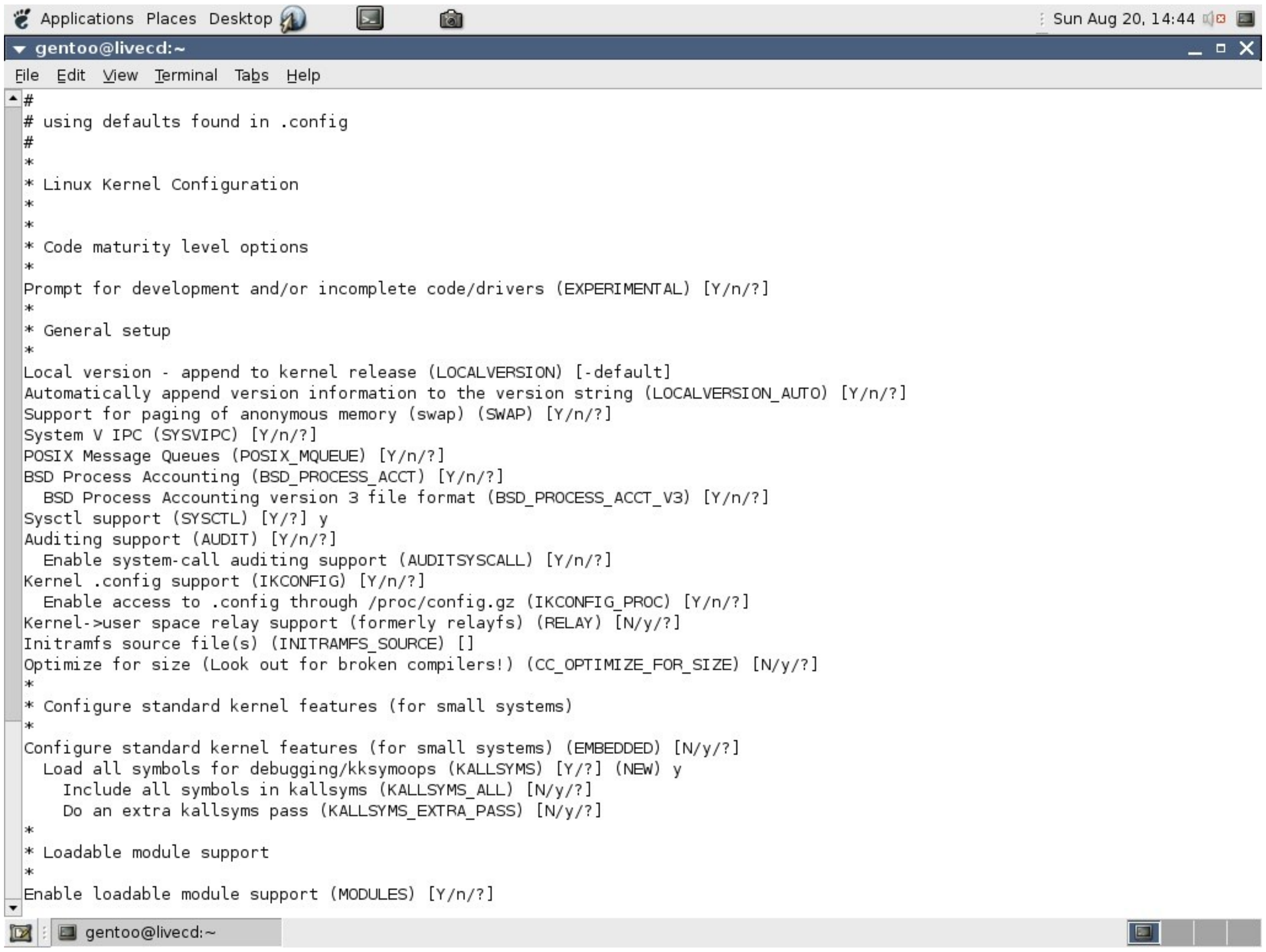
To set-up and/or build a kernel, the good old ***make*** utility is used. For starting a configuration from scratch, we need to select the target architecture (CPU type). If you don't select one, the "x86 architecture" is used by default.

So you could issue following command for setting up a MIPS target:

```
$ make config ARCH=mips
```

When the "***config***" make option is used, the old **Linux** configuration system which uses a text based shell script, will be used to configure the kernel. Today, this is not very useful when someone wants to set-up a system from scratch because there are too many questions need to be answered and also there is not an overview about the different sections. Moreover, it is not possible to change just one option without going through all questions… Luckily the current setting is the default answer.

This configuration option should only be used in exceptional cases. As we don't find examples of these cases, we suggest avoiding the usage of this old fashioned approach. Figure 2 presents a screenshot of the input screen.



```
#
# using defaults found in .config
#
*
* Linux Kernel Configuration
*
*
* Code maturity level options
*
Prompt for development and/or incomplete code/drivers (EXPERIMENTAL) [Y/n/?]
*
* General setup
*
Local version - append to kernel release (LOCALVERSION) [-default]
Automatically append version information to the version string (LOCALVERSION_AUTO) [Y/n/?]
Support for paging of anonymous memory (swap) (SWAP) [Y/n/?]
System V IPC (SYSVIPC) [Y/n/?]
POSIX Message Queues (POSIX_MQUEUE) [Y/n/?]
BSD Process Accounting (BSD_PROCESS_ACCT) [Y/n/?]
  BSD Process Accounting version 3 file format (BSD_PROCESS_ACCT_V3) [Y/n/?]
Sysctl support (SYSCTL) [Y/?] y
Auditing support (AUDIT) [Y/n/?]
  Enable system-call auditing support (AUDITSYSCALL) [Y/n/?]
Kernel .config support (IKCONFIG) [Y/n/?]
  Enable access to .config through /proc/config.gz (IKCONFIG_PROC) [Y/n/?]
Kernel->user space relay support (formerly relayfs) (RELAY) [N/y/?]
Initramfs source file(s) (INITRAMFS_SOURCE) []
Optimize for size (Look out for broken compilers!) (CC_OPTIMIZE_FOR_SIZE) [N/y/?]
*
* Configure standard kernel features (for small systems)
*
Configure standard kernel features (for small systems) (EMBEDDED) [N/y/?]
  Load all symbols for debugging/kksymoops (KALLSYMS) [Y/?] (NEW) y
    Include all symbols in kallsyms (KALLSYMS_ALL) [N/y/?]
    Do an extra kallsyms pass (KALLSYMS_EXTRA_PASS) [N/y/?]
*
* Loadable module support
*
Enable loadable module support (MODULES) [Y/n/?]
```

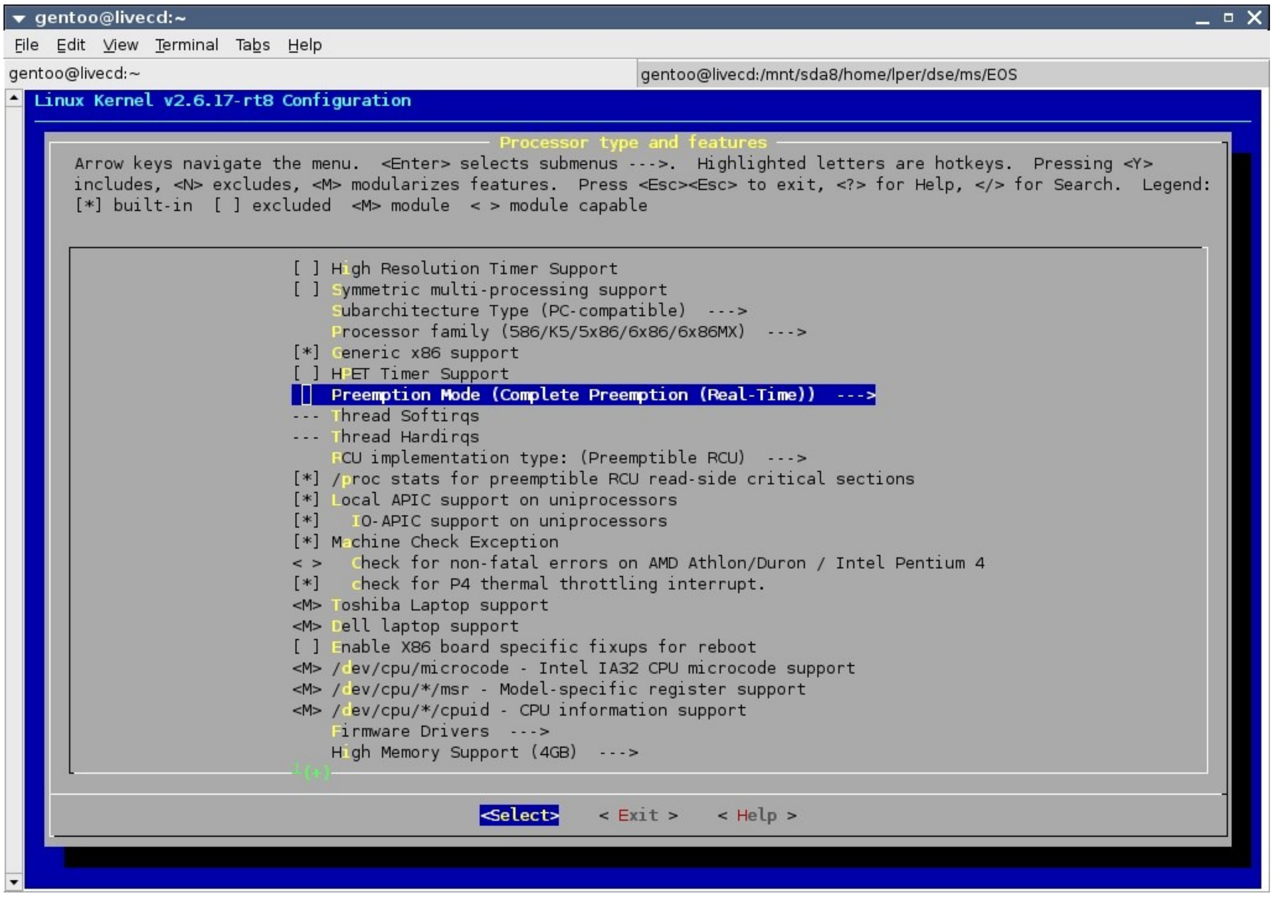
Applications Places Desktop
Sun Aug 20, 14:44
gentoo@livecd:~
File Edit View Terminal Tabs Help
gentoo@livecd:~

Figure 2: "make config" screenshot

A far better alternative is to use the ***menuconfig*** option instead.

In this case, you have a clear view on the different sections by a menu based selection system (figure 3). It is displayed without any special graphics, which supports it to be used on a terminal interface.

Another advantage of this option is the possibility to search for specific ***CONFIG*** tags which, after the search will be displayed as a list within the configuration options in tree format, indicating where to find this specific ***CONFIG*** tag. You cannot jump directly to these configuration options from the search screen).


gentoo@livecd:~
File Edit View Terminal Tabs Help
gentoo@livecd:~
gentoo@livecd:/mnt/sda8/home/lper/dse/ms/EOS

```
Linux Kernel v2.6.17-rt8 Configuration

                    Processor type and features
  Arrow keys navigate the menu.  <Enter> selects submenus --->.  Highlighted letters are hotkeys.  Pressing <Y>
  includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend:
  [*] built-in  [ ] excluded  <M> module  < > module capable

                      [ ] High Resolution Timer Support
                      [ ] Symmetric multi-processing support
                          Subarchitecture Type (PC-compatible)  --->
                          Processor family (586/K5/5x86/6x86/6x86MX)  --->
                      [*] Generic x86 support
                      [ ] HPET Timer Support
                          Preemption Mode (Complete Preemption (Real-Time))  --->
                      --- Thread Softirqs
                      --- Thread Hardirqs
                          RCU implementation type: (Preemptible RCU)  --->
                      [*] /proc stats for preemptible RCU read-side critical sections
                      [*] Local APIC support on uniprocessors
                      [*]   IO-APIC support on uniprocessors
                      [*] Machine Check Exception
                      < >   Check for non-fatal errors on AMD Athlon/Duron / Intel Pentium 4
                      [*]   check for P4 thermal throttling interrupt.
                      <M> Toshiba Laptop support
                      <M> Dell laptop support
                      [ ] Enable X86 board specific fixups for reboot
                      <M> /dev/cpu/microcode - Intel IA32 CPU microcode support
                      <M> /dev/cpu/*/msr - Model-specific register support
                      <M> /dev/cpu/*/cpuid - CPU information support
                          Firmware Drivers  --->
                          High Memory Support (4GB)  --->
                      ↓(+)

                     <Select>    < Exit >    < Help >
```

Figure 3: “make menuconfig" screenshot

If you are running an X-Windows environment with the ***QT toolkit*** installed, then you can use ***xconfig***. This shows you the complete tree structure together with a detailed help for each option. It also facilitates browsing the different options and enables you to have a good understanding of the different kernel features that should be used on your platform.

“make” gconfig is very similar to ***xconfig***, but it uses the ***GTK toolkit*** instead.

In general, you should know the **Linux** kernel internals in order to setup a configuration that is specific for your platform. Surely, when the RAM size is limited, disabling some kernel features and modules can decrease the required memory size for running the kernel. Sometimes you need to take a look in to the code to understand the impact of some options.

Anyhow, setting up a BSP configuration is a difficult task for any operating system and requires some (serious) expertise from the developer.

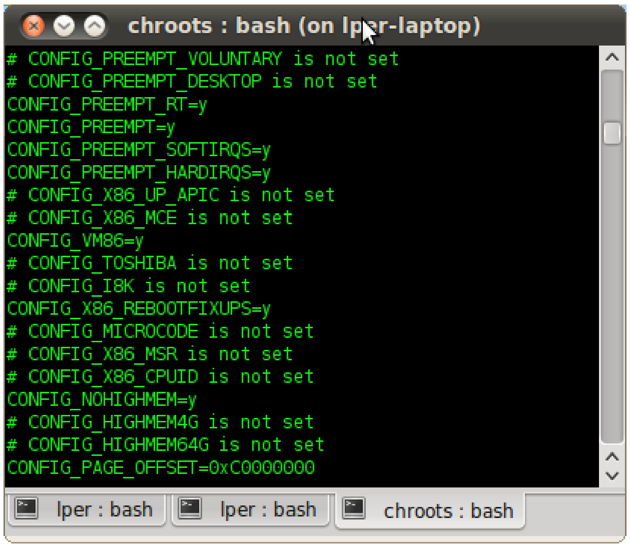
Remark as well that some modules and/or kernel options can affect the real-time performance. So you should be careful in this issue and it is recommended that you start from a specialised vendor release or from an OSADL version.

4.2.3 .config

The selected configuration settings will be finally stored in a ***.config*** file (figure 4). This ***.config*** file will be used by the ***makefile*** for setting the compile options, creating configuration header files and selecting the files that should be compiled.

This file can be easily found in a running **Linux** distribution in the directory ***/proc*** under the name ***config.gz*** (path: ***/proc/config.gz)*** (at least if this option is built in the kernel).

This file can also be manually adapted (if you are experienced!) and can be used as a base for setting up a new configuration (by using ***make oldconfig***).



```
# CONFIG_PREEMPT_VOLUNTARY is not set
# CONFIG_PREEMPT_DESKTOP is not set
CONFIG_PREEMPT_RT=y
CONFIG_PREEMPT=y
CONFIG_PREEMPT_SOFTIRQS=y
CONFIG_PREEMPT_HARDIRQS=y
# CONFIG_X86_UP_APIC is not set
# CONFIG_X86_MCE is not set
CONFIG_VM86=y
# CONFIG_TOSHIBA is not set
# CONFIG_I8K is not set
CONFIG_X86_REBOOTFIXUPS=y
# CONFIG_MICROCODE is not set
# CONFIG_X86_MSR is not set
# CONFIG_X86_CPUID is not set
CONFIG_NOHIGHMEM=y
# CONFIG_HIGHMEM4G is not set
# CONFIG_HIGHMEM64G is not set
CONFIG_PAGE_OFFSET=0xC0000000
```

Figure 4: ".config content"

We started with the ODSAL kernel configuration. We removed only the unnecessary kernel modules in order to have the minimal kernel footprint. (E.g. we did not put USB support in the kernel). You can see our configuration as an optimistic "best possible" configuration.

Further, we changed some kernel configurations because their default settings can seriously jeopardize the real-time behaviour. For more details about this, see our theoretical evaluation report DOC [5].

5 Qualitative evaluation

This report is limited to the qualitative evaluation of the technical aspects of the product under test. The quantitative test results on a specific platform (e.g. X86) can be found in other reports as [Doc 5].

The scores in this document are given according to: 1) our gathered experiences from evaluating previous versions of this product; 2) comparison of this product with other competing tested products. Although all effort has been done to be as rigorous as possible in the scoring, one should accept that they are partially based on a subjective criterion.

This part of the evaluation has the sole intention to help the application designer in his work. Indeed, each (commercial) RTOS is using particular architectural approaches and working models. This has a serious performance and behaviour consequences influencing considerably application designs.

Reading this document helps in understanding how to configure the OS in order to insure RT-behaviour. We expect the reader to have already a very good knowledge about Linux internals.

5.1 OS Architecture

RTOS Architecture	0	▨	▨	▨	▨	▨	▨ 6					10

__Linux__ with the __RT_PREEMPT__ patch has today the necessary features to provide some real-time behaviour and as such it may be used for handling soft real-time requirements, but only if different conditions are met.

The __Linux__ operating system was not initially developed with real-time characteristics in mind and this is still true for the __Vanilla__ kernel (although some real-time improvements were merged in the mainline kernel). You will need the __RT_PREEMPT__ patch together with a correct build and configured kernel to achieve real-time characteristics. Even with the __RT_PREEMPT__ patch, some modules can break real-time behaviour. Remark that not all releases have the same stability required for embedded systems. As such, be sure to use known stable releases (like the OSADL ones or the ones from the different vendors) and be aware that the behaviour can change seriously between releases!

5.1.1 The Linux scheduler dissected

This section will be a detailed description about **Linux** scheduler. We will gradually introduce the different aspects together with explanations of their impact on the scheduling behaviour. We start with the CFS scheduler as introduced in version 2.6.23.

5.1.1.1 Scheduling Classes

Linux supports different scheduling classes where threads can belong to. These classes are classified based upon the working environment.

- In a normal desktop/server (non-real time) environment, two classes are used only:

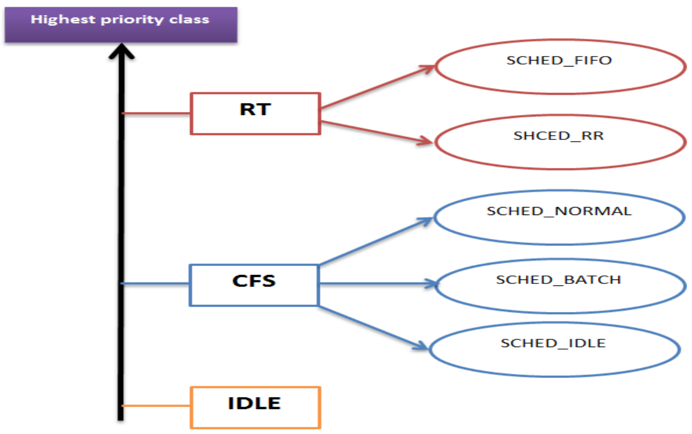
- The "Completely Fair Scheduling (CFS)" Class where the pthread scheduling functions uses one of those scheduling policies: ***SCHED_NORMAL, SCHED_BATCH*** and ***SCHED_IDLE***. This scheduling class has been completely reworked (by Ingo Molnar upon ideas of Con Kolivas and others) and merged into the 2.6.23 release of the **Linux** kernel.
- The "Idle" class which is used only for the idle threads (one for each processor).

➢ In the real-time environments, one class is used only:

- The Real-Time Scheduling Class: it is the POSIX fixed-priority real time scheduling, for ***SCHED_RR/SCHED_FIFO*** policies with 99 priority levels.

Since version 2.6.23, these classes started to have their own abstract interface with the kernel scheduler, which makes it easier to adapt scheduling classes in the future.

All these classes are sorted according to their priorities, starting from the highest priority towards the lowest priority (from RT -> CFS -> IDLE) as shown in the following figure.



In case there are no "ready-to-run" threads in the high priority class, then the lower-priority classes are asked for the next thread to be scheduled and executed. There is always one "idle" thread in the idle class (lowest-priority class at the bottom) ready to run if no threads in higher classes can be activated. Due to this strict sorting, adding more classes will not make much sense. The abstraction of scheduling classes makes things more loosely coupled with regards of the central scheduler, which is always a good thing. Due to the strict prioritization of these classes, adding extra classes is not that useful. However, the loosely coupling makes it easy to try-out new schedulers and tests them without the need to rewrite a lot of parts in the kernel. This loosely coupling is used in the academic world to experiment with different type of schedulers.

The kernel will use this interface to the schedule class to ask for the next thread to run from this class. How the class handles internal prioritizations is hidden from the central scheduler. This is a clean way to split responsibilities.

5.1.1.2 CFS threads (SCHED_OTHER or now called SCHED_NORMAL)

We will start our review with normal user threads (scheduler class ***SCHED_NORMAL***), which has been completely reworked since **Linux** kernel version 2.6.23. For simplicity, we will start with a single processor system (thus no SMP support). In this simplified case, the scheduler has only to decide if a thread should run or not.

CFS wants to be fair in scheduling threads. In an ideal "fair" world, each running thread would receive exactly the same amount of CPU cycles. Ingo calls this a "virtual runtime". For instance, if there are 100 runnable threads in the system, you would receive a virtual part of 1% from the CPU time. Depending on how much you have already used of your virtual part, the threads are ordered in a balanced tree (rbtree), thus causing an O (Log (n)) lookup delay for thread switching.

The ready thread, which has used the smallest portion of the CPU time allocated to it, will have the highest priority in this tree and thus become the next scheduled thread. This pushes I/O bound threads in front (as these are not CPU intensive and thus use only a smaller part of their virtual runtime) and as such improves the I/O bandwidth.

The problem is however what "fair" means. In the kernel version 2.6.23, fairness was done between all threads in the system. In this case a user running 100 threads would receive ten times more CPU power than a user running only 10 threads… So this was not fair at the user level! Therefore, the 2.6.24 kernel version adapted this behaviour to a new "fairness" approach composed of 2 cascading layers: an upper layer between the users themselves (but other grouping systems may be used as well) and the lower layer between the threads of each user. In mean time, the fairness between users is dropped, but the generic grouping of threads can still be done. Remark that such grouping is not something to be setup easily!

But it is still possible to prioritize threads in the CFS scheduling system (make them less fair!). This is not done by using special strict "priority" as the priority is used only to indicate how much virtual CPU time a thread may consume while competed with other threads. For this the "nice" call is used to change the amount of virtual runtime of any thread. The "nice" values range between -20 and 19, where a high "nice" value means less CPU resources usage (thus a smaller part of the CPU: +19 means about 1.5% CPU power). Indeed being "nice" means that you give others something…

Remark as well that there are different configurable scheduling parameters which affect the latency/throughput trade-off. Those parameters can be adapted by using the sysctl interface. But of course, if you really need low latency, it is better to use RT threads. This will be discussed in the next section.

5.1.1.3 RT Threads

As we are looking to the latency figures in our tests, we are not interested in "fair" scheduling. Our aim is to have short latencies when needed. Achieving such aim is done via the usage of ***SCHED_RR*** or ***SCHED_FIFO*** scheduling policies. Both policies activate the RT scheduler, which has a strict higher priority than the fair scheduler as described above.

Moreover, the priorities in real-time threads are handled in a strict way: if there is a high priority thread ready to run, and the CPU is currently running a thread of lower priority than the ready thread, then the running lower priority thread will be pre-empted. Other commercial RTOS have exactly the same behaviour. **Linux** foresees 100 priorities in the Real-Time scheduling class, which is for some extent a low range compared to other RTOS (most use 256 priorities), but it is adequate for most projects (having less than 100 threads running in the real-time class).

The difference between ***SCHED_RR*** and ***SCHED_FIFO*** is the scheduling behaviour for threads having the same priority. In the FIFO case, each thread will run until it completes (e.g. until it blocks), while in the ***Round Robin (RR)*** case, the thread will use the CPU for a specific amount of time and then will be put on the back of the queue after this time slice is passed. In both cases, threads going from the blocked to the ready state are put on the tail of the ready queue for this priority, although there are some small differences in behaviour:

Indeed we noticed during our tests some specific behaviour differences between the scheduling classes ***SCHED_RR*** and ***SCHED_FIFO***:

- Creating a new thread in the ***SCHED_RR*** case puts it in FRONT of the "ready-to-run" queue of its priority.
- Creating a new thread in the ***SCHED_FIFO*** case puts it in the BACK of the "ready-to-run" queue of its priority.

Similar behaviour was noticed when priorities are changed, lowering the priority while it is the only thread in the current priority level:

- In case of ***SCHED_RR***, lowering the priority to the next lower queue with ready-to-run threads will put the thread in front of the FIFO. If it was the only ready thread in the queue it left, it will keep running.
- In case of ***SCHED_FIFO*** it will stop running and be put on the back of the FIFO.

Remark that the ***SCHED_RR*** behaviour is similar to the thread behaviour in **Windows CE**, while the FIFO case matches the behaviour of most other RTOS.

A good article on how the internals are designed, can be found in the "**Linux** Journal, issue august 2009: Real-Time kernel scheduler" written by Ankita Garg.

In this article some "SMP" features are explained. The scheduler will always follow these strict rules, even if therefore a thread has to migrate to another CPU (run queue). The way to reduce the overhead for this is done by using only a selective number of CPUs for scheduling real-time threads (the root domain concept). In that way the number of cross CPU actions/locking can be decreased. A number of CPUs can be put in one "root domain", for which all threads for a process (or a group of processes) will only be scheduled on these CPUs.

As long you divide your CPUs in a way that the root domains do not overlap (e.g. a CPU is not used more than in one root domain) then there is no need to use locks between these domains. This speeds up thread switch latency (even if the switch does not need a thread migration, locking has impact on thread switch latency).

The threads themselves can also be bound to a specific CPU by using an affinity mask, but this avoids thread migration only. The lock between multiple CPUs will still be needed (as there can be threads not setting the affinity mask). By using one CPU only in the root domain, there is no need for inter-CPU locks.

5.1.1.4 RT scheduling in practice…

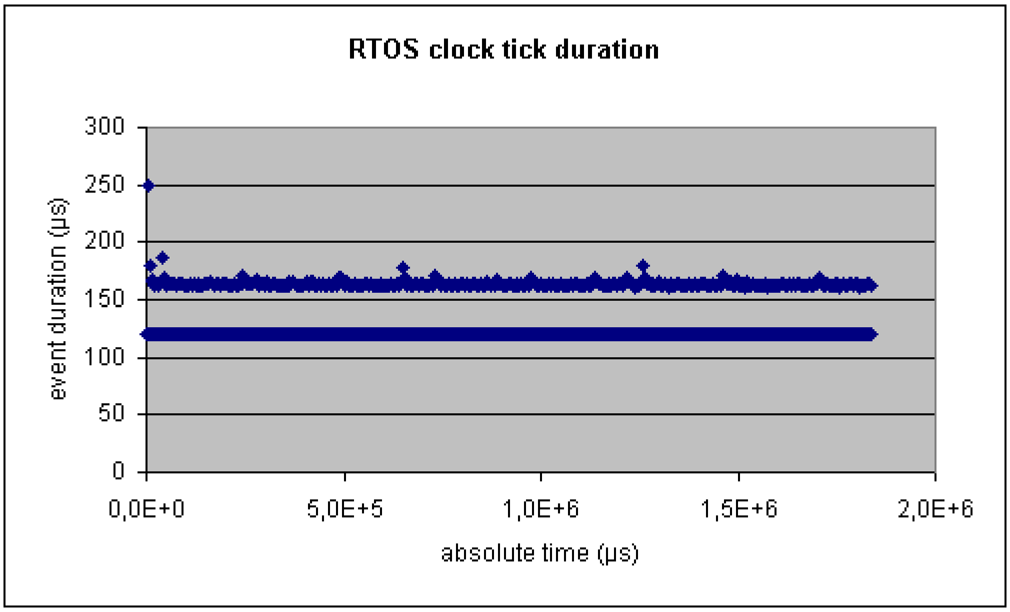
All above is theory… Once we are taking a closer look to the kernel source (and documentation), it seems that some mechanisms are built-in to prevent starvation of normal threads from real-time threads!

☹ This is really bad! If someone writes a real-time thread which has an endless loop, the system should end up in starvation (e.g. any thread of lower priority will NEVER have the CPU, at least on a single CPU system)! This is wanted and REQUIRED behaviour for a system with real-time capabilities! Anything to avoid this is really wrong. It shows that some designers do not understand what real-time means!

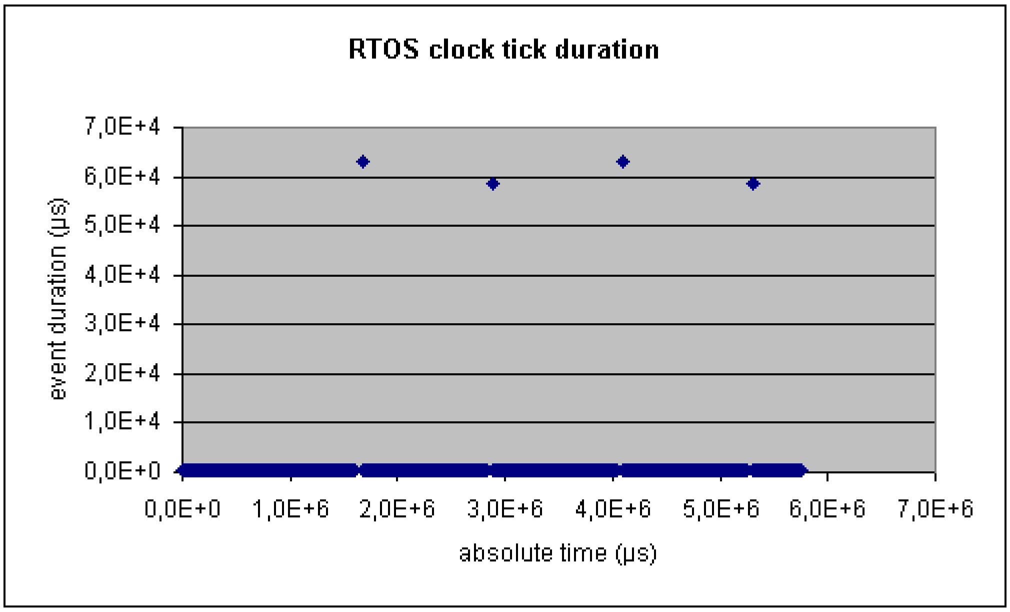
Luckily this "bad feature" can be disabled (take care: it isn't by default), by modifying a parameter in the ***/proc/sys/kernel*** space. You should set the ***sched_rt_runtime_us*** to -1 for disabling this feature. By default, each second, a time slice of 50 ms is reserved for non-real-time threads, which is surely enough to completely screw your real-time behaviour!

This is shown in the diagrams below. In this test, we are going to measure the time required by a real-time priority thread that performs a loop calculation. At some moments, the **Linux** scheduler timer tick will occur and delay the activity. That's why you see the two lines: the bottom line shows the duration of the calculation loop if no scheduler timer interrupt occurred while the upper line shows if a timer interrupt occurred during the test loop. This test is used to measure the timer tick interrupt handling duration. Besides this high priority real-time thread we have one user thread running a dummy loop running at normal priority using the ***SCHED_OTHER*** class.

The diagram below is the result if starvation protection is disabled: clearly this behaves as it should… the user thread running at normal priority never activates (thus never delays the real-time thread).



The diagram below shows the result if the starvation protection feature is enabled (e.g. default behaviour): each second, the normal priority thread activates and just takes 50ms of the real-time thread! You can't even see anymore the two lines as they are just zero compared with the huge 50ms delay! This is really bad RT behaviour.



Recently there has been some developers' activity to have group scheduling for real-time threads. This permits to divide the amount of CPU power between real-time groups. Again this is very dangerous approach: what if a thread of high deadline (priority) is located in a group, but the scheduler decided to schedule the other group threads? Clearly this will produce an unpredictable behaviour that we all aim to avoid at all cost (at least in real-time systems).

The approach used here is to give some pseudo real-time characteristics for different users (or even virtual machines). But then it is difficult to talk about "real-time" performance. It is probably better to talk about threads having "improved latency" compared with fair threads.

In our tests all these features are disabled (either by configuration at runtime, or by kernel configuration at build time), so we measure in the best case scenario (for RT of course).

All the above shows you that the scheduler is constantly evolving and changing. As a consequence, upgrading to a new kernel can be a risky business! Also changing the kernel build configuration can seriously impact the behaviour.

Below is an overview of the **Linux** tasking model:

	OS under evaluation
Supported memory models	Full virtual memory
Thread/process priority levels	100 real-time priority levels
Max. number of processes	Depending on available memory
Max. number of threads	Depending on available memory
Scheduling policies	CFS time slicing, for non-real-time threads (***SCHED_NORMAL***) Prioritized FIFO for real-time threads (***SCHED_FIFO***) Round-robin scheduling for real-time threads (***SCHED_RR***)
Number of documented thread states	4: "running", "running-***mutex***", "sleeping", "disk sleep" There are also different stopped/exit states.

5.1.2 Memory Management

Memory management in **Linux** is similar to most general operating systems. It has a protected kernel space (running in supervisor mode) which can only be accessed by system calls (using traps). Further, the different user processes are protected from each other by using memory protection between them.

Linux achieves such memory protection via the usage of the MMU (Memory Management Unit) concept. In fact, **Linux** cannot run without this feature (there exists uC**linux** for this, but this version deviates quite a lot from standard **Linux**).

In **Linux**, the virtual memory paging system mostly uses page sizes of 4KB, although others are possible as well (depending on CPU architecture). There is a system interface present for using “huge pages”, which are 2MB in size on **Linux** x86, and as such this limits the number of TLB exceptions and page faults to handle. Furthermore these “huge pages” cannot be swapped out, making the behaviour predictable. The downside of this is that these “huge pages” can be allocated only in an early phase of system initialisation (e.g. when the physical memory is not yet fragmented).

Having virtual memory that can be swapped away to some slow storage medium is of course a bad idea for keeping real-time deadlines. We will discuss some memory management specifics and the impact on real-time behaviour further in this document.

	OS under evaluation
MMU support	Yes, obligatory
Physical page size	4KB typically (but can be configured depending of architecture)
Swapping/Demand Paging	Yes, but can be disabled, but not completely for read-only sections.
Virtual memory	Yes
Memory protection models	Kernel / user and inter-process protection

5.1.2.1 Copy-On-Write

Normally, **Linux** never reserves physical memory when virtual memory is requested. Instead, upon access of some virtual page which is not yet present in physical memory, a page-fault will occur. It is at that particular moment the memory manager in the kernel will try to allocate a physical page and map it in the virtual memory page.

For instance, when you load an executable, the initial zeroed data will be located in a virtual memory region, for which all virtual pages are mapped (read-only) to the zero page of the kernel (a physical page containing only zeros). All the reads will return zero and will not cause a page fault. The first write however will cause a page fault (as the zero page is read-only) and at that moment a new physical page is allocated, cleared with zeros and mapped to the virtual memory page causing the fault.

The same is true for allocation routines like `malloc`. The `malloc` routine will be handled in the libraries (in user space) if possible. If extra memory is needed, it will be requested from the kernel using the `mmap` (or brk) system calls. Again the kernel will only return virtual memory, not physical memory! This also means that you can allocate much more virtual memory than there is physical memory present in the system. We gave once a small demo to a customer to prove this point. It was done on some embedded platform (having 64MB ram), where it was possible to allocate 3.5GB (over multiple processes) of virtual memory using `malloc`! Yet the system did not have any swap!

Of course, once you write to virtual memory which is not yet mapped to a writable physical page, an out-of-memory condition could happen (in which case some random process can be killed… or for root processes it can even lead to a system lock on some kernel versions).

There are ways to try to prevent this by setting some over commit settings in the ***/proc/sys/vm/*** directory. However, none of these ways can prevent such out-of-memory condition in all cases, but at least they can lower the occurrence probability of such conditions.

A similar problem occurs for the thread stack. While the stack is growing, page faults can occur which will allocate physical memory. Luckily you can set the maximum available stack upon thread creation.

A typical use case of such sparse allocated memory is in large arrays used in mathematical calculations. Typically in such arrays, only the diagonal contains “non-zero” values. So you can do the calculations while there is no need to allocate all physical memory for the arrays (the zero values are just read-only mappings of the zero page).

It is clear that such way of working, although efficient for desktops and servers, cannot be used for real-time systems. Page faults should not happen during a critical low latency path!

Therefore the `mlockall ()` system call exists. This call obliges the kernel to directly allocate the physical pages for any virtual memory allocation (which is active and/or will become active in the future). Using `mlockall` (what we do in our tests) at the beginning largely solves this problem. As an alternative, allocating and accessing all required memory by writing at least one byte on each page can be used to allocate the physical memory.

In older kernels we know that the stack allocation could still be a problem, but in the tested kernel, the physical memory allocation for the virtual stack was done upon thread creation (if mlockall was set, otherwise the copy-on-write is used). This is very good indeed! However be aware that you should explicitly set the stack size for each thread you create. **Linux** uses by default (depending of architecture) a 2MB stack. In low memory platforms, you will encounter problems if you create a couple of threads. In our test we configured the threads with 4 pages of stack.

5.1.2.2 Swapping pages

The page faults caused by not yet allocated physical memory is not the only case; page faults can also happen by pages which are moved (swapped) into memory from some (slow speed) storage. Retrieving these pages is even slower than allocating physical pages!

Luckily, you can configure the kernel in order not to use this swap support feature, or you can just do not reserve any swap to avoid swapping altogether. Also the `mlockall ()` call will ensure that memory of your process is not swapped away.

☹ However, take care! There is one scenario that cannot be prevented which is swapping away the read-only pages (like executable sections in a program). Indeed, read-only pages were read at some time during start-up from some storage medium. As such they can be re-read from this medium at any time. **Linux** is clever in doing such things, and thus it can drop these physical pages (as they are "backed" on some storage). This is clever; for instance, most ***init*** code will run only once…

However, to assure deadlines, such things should not happen!

There is no way to disable this in the kernel, but there is an easy work around: start your application/libraries from "***tmpfs***" (which is a RAM file system layer). Of course the latter has some impact on memory usage, but it is the only way to avoid dropping read-only pages.

During our tests, the kernel was configured without swap and our process used `mlockall ()` at start-up. Further our application was statically linked with the libraries and started from a RAM file system (thus first copied to ***tmpfs***).

5.1.2.3 Memory pools and heap

In user space, memory allocation is done using the standard C library calls "`malloc`" which are provided for free by the **glibc** library. There is no support for fixed block size memory pools or anything alike (something typically used in real-time systems to provide predictable allocation time). So you will need to add your own layer above the standard library one for adding such support.

As remarked earlier, the virtual memory allocated using "`malloc`" call function is not necessarily to be physically available by default, until your first write to a page of it. Thus, for real-time systems, it is better that you set the `mlockall ()` flag for current and future allocations at application start-up!

There are some tuning parameters in "**glibc**" to adapt the behaviour of the allocation routines. For instance you can configure the quantity of free memory that should be available before returning it to the system. You can set also from which allocation size the directly mapped memory (multiple of page size) should be used instead of the common heap pool.

The advantage of using directly mapped memory is that no fragmentation can occur (the block can be freed at once, releasing all pages to the system). The disadvantage is of course that you will over-allocate (as you need always to allocate a multiple of the page size).

When you plan to write device drivers (modules), then of course the user space allocation routines cannot be used. The kernel uses its own specific memory manager. Two cases are possible:

- The required memory block is large (e.g. “pages”) and there is no requirement that the physical pages are continuous, then a call with similar behaviour as malloc can be used: `vmalloc ()`. This call will allocate the virtual memory you need (the same rules apply as `malloc`: e.g. the physical pages are not necessarily present). It is clear that due to the nature of paged virtual memory, this cannot be used by interrupt handlers.
- The required block size is small, or you need continuous memory that needs to be always available (hardware access, interrupt handlers), then fixed block size memory pools are used. This is called the “***slab***”.

As `vmalloc` behaves similarly as `malloc` (although handled in kernel space), we will only discuss more in detail the *slab* allocator.

As mentioned before, the ***slab*** allocator uses fixed memory block pools. You can create a specific pool yourself, so it may be used for any block size. Further if you use `kmalloc ()` to allocate memory, it will allocate memory from binary sized pools (8, 16, 32 …). You can use the command “***cat /proc/slabinfo***” to see more information on the ***slab*** usage (if the kernel is configured to provide you these statistics).

The ***slab*** uses directly the mapped memory (e.g. how the kernel accesses the physical memory), therefore it is not page-able. Further the allocation routines are fast (if you configured the specific pool yourself and large enough).

As the ***slab*** is not paged, care should be taken with larger memory blocks (e.g. more than a page), as this may quickly become a problem if the memory gets fragmented (causing an out-of-memory condition in the kernel). As always, the best is to allocate enough upfront.

Remark that due to the nature of virtual memory and the file system caching used in **Linux**, it is hard to answer the typical embedded system question “how much memory there is still free?…”. Using ***/proc/meminfo***, you can have a view on it. Cached memory (e.g. file system cache) can largely be freed but never in a complete way. Remark that the read-only execution sections and ***tmpfs*** are cached as well (in fact ***tmpfs*** is just cache without a physical storage below). If the execution sections are removed from the cache, it would have a serious performance impact (e.g. they have to be paged in again when you use them). If you only check the “free” memory part, then you underestimate the free memory as a lot of cached data can be probably thrown away without any impact (files you accessed once some time ago).

As a consequence, there is still no easy answer to the raised question “how much memory there is free?” A useful approach is to have a memory allocation application (just `malloc` and touch pages) and you try it out on your system until you start to see performance degradation or out-of-memory conditions.

5.1.3 Interrupt Handling

In the kernel, interrupts are handled in two layers:

- The direct interrupt handler (e.g. where the processor is in the exception state, which started execution at some interrupt vector). This is also called first level interrupt handler
- The second level interrupt handler (which runs in the kernel space). This is also called the bottom half interrupt in **Linux** terminology (or soft interrupt in more recent kernels).

As in all other operating systems, the first level interrupt handler may never use blocking calls. This of course limits the available calls that are allowed in this part of the interrupt handler. For instance, paging is not possible, nor any blocking lock operation (releasing is possible).

It also means that the first level interrupt handlers have always a higher priority than any thread in the system and as such, badly written first level interrupt handlers (e.g. taking a long processing time) have negative influence on real-time latency (except of course if the real-time requirement is only located in the interrupt handler).

5.1.3.1 Softirq

The second level interrupt is recently called the ***softirq***. It is an important one in order to understand latencies (surely in the non-real-time kernel).

In normal kernels (not using RT preempt patch), just before the first level interrupt handler would return, there is a check if some more processing should be done (e.g. thus if there is a second level interrupt or softirq pending). If this is the case, some stack manipulation is done with a direct jump to the softirq thread context (depending on CPU architecture). Also interrupts are re-enabled. This means that under normal circumstances, the second level interrupts are handled directly after the first level interrupt. As a result, such interrupts have a priority between the first level interrupt (as these are enabled again) and normal threads (as the scheduler is not used for the transition).

Long time ago, only one ***softirq*** level existed and all second level interrupt were queued on one FIFO. Today there are about 10 ***softirq*** levels (queues). Each time the ***softirq*** handler starts, it starts processing the highest priority queue.

As one can imagine, a lot of ***softirq*** processing will cause starvation for user threads. Alike limiting the RT thread CPU usage if starvation is coming close, the same feature is implemented in the ***softirq*** handler (yep this is fun, again we reverse priorities!). If the ***softirq*** handler finishes handling one pending second level interrupt, it checks then if there is yet another one pending (which is possible as the interrupts are enabled). On each loop, a counter is decremented (set at 10 when ***softirq*** handler is entered). Once this counter reaches zero, the ***softirq*** thread is activated (called the ***softirq*** daemon: ***softirqd***) and an exit happens. This means that suddenly the second level interrupt handlers are postponed and run in a thread with the priority just like other user tasks!!!!

Again this has been implemented to avoid “user space” starvation, but this feature has a major impact on latencies. Surely on embedded devices where the available processor performance is limited compared to desktop/server systems, this starvation can surely happen. For instance on a **Linux** based gateway where a lot of network traffic passes (video, data, VOIP, WIFI, …) this can easily lead to a serious jitter during heavy load situations. This jitter then impacts VOIP and video quality.

Remark that on older kernels, the ***softirqd*** had even the lowest priority in the system, making this jitter even worse!

Clearly, to fulfil some low latency requirements, this is a wrong architecture. However, here the ***RT_PREEMPT*** patch comes to help. So let us take a closer look on how this patch addresses these issues!

5.1.3.2 Softirq in RT_PREEMPT

Let's come back to the problems with the normal kernel ***softirq*** handling:

- ***Softirqs*** are handled directly after the first level interrupt and as a consequence, they have impact on latency for all threads in the system.
- Under heavy load, the ***softirqs*** can be dispatched to a thread, making the latency of the ***softirq*** handling huge.

Using the ***CONFIG_PREEMPT_SOFTIRQS*** settings, each ***softirq*** queue has its own thread (using the fixed real-time priority 49).

☺ Now the behaviour is always the same: ***softirqs*** are ALWAYS handled in a thread context and their priority is NEVER lowered, even not during heavy load making the behaviour predictable.

As ***softirqs*** are just threads running at real-time priority, it is possible to have user threads with higher real-time priority than the ***softirq*** threads and as such the latter will never be interrupted by the former, which helps improving latency if needed.

Further it is possible from user level to adapt the priorities of these threads, which gives you some flexibility in priority assignments.

Still the first level interrupts have higher priority and may affect latency. Therefore the ***RT_PREEMPT*** patch goes even further: they introduce the "***hardirq*** threads"!

5.1.3.3 Hardirq threads

Depending on your processor/hardware architecture, the ***RT_PREEMPT*** patch enables the concept of first level interrupts handled in the thread context.

What interrupt handler does in case of ***hardirq*** threads is:

- Disable the received interrupt (so you need to be able to mask selective interrupts on the interrupt controller, otherwise this system cannot be implemented)
- Activate the ***hardirq*** thread linked with this interrupt.
- In the ***hardirq*** thread, call the original first level interrupt handler

By default, the ***hardirq*** thread runs at real-time priority 50. Also this priority can be changed, but of course it should always have a priority which is higher than the second level interrupt handled in the ***softirq*** thread. Otherwise, the programmer paradigm that the interrupt handler cannot be interrupted by threads (implicit locking) would not be valid any more, leading to unexpected behaviour and crashes!

☺ So now user threads can have higher priorities than the interrupts (remark that only privileged users can create threads with real-time priority)!

Of course you have to know what you are doing in this case, but this is what real-time engineers are used to. The liberty of choosing priority levels for ***hardirqs***, ***softirqs*** and real-time user threads will help you a lot to achieve lower latency requirements.

5.1.3.4 Timer tick interrupt

Not all interrupts can be handled using this ***hardirq*** mechanism (e.g. you need the ability to mask an individual interrupt). Furthermore, some interrupts are so closely coupled with kernel behaviour, which means that putting such interrupts on a thread level will be a bad approach. Therefore the operating system timer tick for instance cannot be moved to a ***hardirq*** thread.

☹ Our tests show that the timer tick has a serious overhead compared to other evaluated RTOS which means a serious increase in the latencies. But for real-time behaviour, predictability is of course more important than the absolute timings.

5.1.3.5 Conclusion

	OS under evaluation
Handling	Prioritized and nested
Context	System (kernel)
Stack	Depends on CPU
Interrupt-to-task communication	When using hardirqs, thread to thread. Otherwise non-blocking calls.

5.1.4 Synchronisation mechanisms

The typical inter-thread synchronization mechanisms are ***semaphores*** and ***mutexes***.

One of the features that was removed from the ***RT_PREEMPT*** patch and moved to the **Vanilla** kernel is the priority inheritance mechanism of the ***mutex*** in order to avoid priority inversions. Any system with real-time requirements must have such mechanism (or some other means to avoid priority inversions). The good news is that the priority inheritance mechanism works correctly as detected in our test suite. But you need to set the priority inheritance feature explicitly because by default it is not. Anyhow, it does not make sense to use priority inheritance for non-real-time threads, as these have no real “priority” concept. For real-time threads, all ***mutexes*** should be configured using priority inheritance.

Internally in the kernel ***mutexes*** are called ***futexes***. This is transparent for the programmer if the **glibc** library is used because it abstracts the internal kernel workings with the POSIX ***mutex*** behaviour. Remark that there is no support for this feature when using **uClibc** as C library (which is used a lot on small embedded systems).

While ***mutexes*** are needed for locking purposes (e.g. protecting simultaneous altering of shared data between threads), ***semaphores*** are meant to be used for signalling purposes or counting (like for a message queue). Both concepts are completely different: a ***mutex*** has an owner (only one thread can have the lock and only this thread can release the lock) while there is no owner concept involved with ***semaphores*** (as any thread can signal the ***semaphore***). As ***semaphores*** do not have the owner concept, they cannot be used for priority inheritance (the owner inherits the priority of the thread asking the lock…). Therefore it is a bad idea to use ***semaphores*** for protecting shared data between real-time threads!

We know that in most computer science courses it is stated that a ***mutex*** is a ***semaphore*** with maximum count 1. However this statement is wrong. ***Mutexes*** have a total different behaviour than ***semaphores***; it is the ownership that makes the difference! ***Mutexes*** should be used for locking (***critical sections***) only, not for synchronization which is something for ***semaphores*** only (like waking up a thread).

5.1.5 Other RT_PREEMPT improvements

As you already noticed during the discussion, a large part of the original developments for the ***RT_PREEMPT*** patch have been moved into the **Vanilla** kernel (like the priority inheritance, ***mutex***, and the RT scheduling). However there are still differences: the interrupt handling threads and ***softirq*** handling is one of them, but another important one is the rework of kernel locks.

The original **Linux** kernel was built around one big kernel lock which means that the kernel was not pre-emptable. This is of course bad if it comes to small latencies. This has been changed a lot in recent kernels, where also the kernel is pre-emptable.

However this is going step by step. The first step was to release the big kernel lock during long running loops only. Today the kernel is largely pre-emptible (if configured as such).

The ***RT_PREEMPT*** will adapt further some locking mechanisms in the kernel. For instance, it will transform spinlocks to pre-emptable ***semaphore*** implementations. This means that the busy time of a CPU is lowered, which can be considered as a nice step forward in order to improve latencies.

☺ These internal kernel locks use priority inheritance as well, which is an important feature for real-time behaviour.

All these changes should help you to achieve short latencies.

☹ However, some drivers are not well written and can lock the scheduling for a long time. So always start on the safe side and only use drivers that are known to behave well when real-time behaviour is important.

5.1.6 Other Linux issues

Achieving the correct configuration of a kernel with real-time responses is not an easy task because there are a lot of options to be considered and being careful of enabling some features. For instance some power management technics in software can be evil for real-time behaviour (depending on CPU and software techniques used).

It is recommended that you configure your kernel according to the configurations used with the test machines at http://www.osadl.org (these run continuously and monitor delays).

Of course you can use versions maintained by commercial vendors as well, as they will have probably better knowledge of the pitfalls creating longer latencies. But of course their support is not for free.

Linux has not been made from the ground up to be real-time (like dedicated RTOS). Instead the known real-time issues should be solved one-by-one. But by doing so, it is difficult to prove that the chosen configuration will meet the real-time requirements: e.g. it is difficult to prove that all bad settings as mentioned above are removed. For traditional RTOS, real-time aspects are incorporated from inception! This is the main difference between **Linux** and traditional RTOSes.

5.2 API richness

Most RT POSIX calls are in.

There is some lack of typical APIs as used in real-time systems: no fixed block size partitions, no event flags, no message queues…

Of course, these lacking features can be easily built on top of the available API. Probably you can even find open source projects which created such libraries for you.

IMPORTANT NOTICE: the purpose of the tables below is only to make an inventory of the kernel/library related APIs. They make an inventory of basic real-time objects and features present in the POSIX and OS proprietary interfaces.

While interpreting these results, the reader should keep in mind that these tables cover a strictly defined set of system calls only. As it is very hard to compare the different API's from the different operating systems, no points are given for this section. The reader should use this section to check if the API calls he needs for his application are available or not.

In general, the more system calls available, the easier it is for the programmer to directly use a call he needs in the framework of his application. Of course, one can always write a proprietary library to extend the available system calls or use an open source library doing the same.

Sophisticated calls implemented in the OS will also seriously reduce the application code length and one might expect that the code implemented in the OS is better debugged and tested (by a lot of users) than code implemented in an application (or proprietary library).

5.2.1 Task Management

Thread management	YES
Get stack size	✓
Set stack size	✓
Get stack address	✓
Set stack address	✓
Get thread state	-
Set thread state	-
Get TCB	-
Set TCB	-
Get priority	✓
Set priority	✓
Get thread ID	✓
Thread state change handler	-
Get current stack pointer	-
Set thread CPU usage	-
Set scheduling mechanism	✓
Lock thread in memory	✓[1]
Disable scheduling	-

[1] For all threads in the process (mlockall)

5.2.2 Clock and Timer

Clock	**YES**
Get time of day	✓
Set time of day	✓
Get resolution	✓
Set resolution	-
Adjust time	✓
Read counter register	-
Automatically adjust time	-

Interval timer	**YES**
Timer expires on an absolute date	✓
Timer expires on a relative date	✓
Timer expires cyclical	✓
Get remaining time	✓
Get number of overruns	✓
Connect user routine	✓

5.2.3 Memory Management

Fixed block size partition	**NO**
Set partition size	-
Get partition size	-
Set memory block size	-
Get memory block size	-
Specify partition location	-
Get memory block – blocking	-
Get memory block - non blocking	-
Get memory block - with timeout	-
Release memory block	-

Fixed block size partition	**NO**
Extend partition	-
Get number of free memory blocks	-
Lock/unlock partition in memory	-

Non-fixed block size pool	**NO**
Set pool size	-
Get pool size	-
Make new pool	-
Get memory block size	-
Get memory block – blocking	-
Get memory block - non blocking	-
Get memory block - with timeout	-
Release memory block	-
Extend pool	-
Extend block	-
Get remaining free bytes	-
Lock/unlock pool in memory	-
Lock/unlock block in memory	-

5.2.4 Interrupt Handling (in kernel)

Interrupt handling	**YES**
Attach interrupt handler	✓
Detach interrupt handler	✓
Wait for interrupt – blocking	✓
Wait for interrupt - with timeout	✓
Raise interrupt	✓
Disable/Enable hardware interrupts	-
Mask/Unmask a hardware interrupt	✓
Interrupt sharing	✓

5.2.5 Synchronization and Exclusion Objects

Counting semaphore	**YES**
Get maximum count	-
Set maximum count	-
Set initial value	✓
Share between processes	✓[2]
Wait - blocking	✓
Wait - non blocking	✓
Wait - with timeout	✓
Post	✓
Post – Broadcast	-
Get status (value)	✓

Binary semaphore	**NO**
Set initial value	-
Share between processes	-
Wait - blocking	-
Wait - non blocking	-
Wait - with timeout	-
Post	-
Get status	-

Mutex	**YES**
Set initial value	-
Share between processes	-
Priority inversion avoidance mechanism	✓
Recursive getting	✓[3]
Thread deletion safety	-

[2] Using a named semaphore
[3] Depend on your initialization of the mutex

Mutex	**YES**
Wait – blocking	✓
Wait - non blocking	✓
Wait - with timeout	✓
Release	✓
Get status	-
Get owner's thread ID	-
Get blocked thread ID	-

Conditional variable	**YES**[4]
Pend non-blocking	-
Pend with timeout	✓
Pend in FIFO / priority order	✓
Broadcast	✓
Priority inversion	-

Event flags	**NO**
Set one at a time	-
Set multiple	-
Pend on one	-
Pend on multiple	-
Pend with OR conditions	-
Pend with AND conditions	-
Pend with AND and OR conditions	-
Pend with timeout	-

POSIX signals	**YES**
Install signal handler	✓
Detach signal handler	✓
Mask/unmask signals	✓

[4] Using pthread lib

POSIX signals	YES
Identify sender	✓
Set destination ID	✓
Set signal ID	✓
Get signal ID	✓
Signal thread	✓
Queued signals	✓

5.2.6 Communication and Message Passing Objects

Queue	NO
Set maximum size of message	-
Get maximum size of message	-
Set size of queue	-
Get size of queue	-
Get number of messages in queue	-
Share between processes	-
Receive – blocking	-
Receive – non blocking	-
Receive – with timeout	-
Send - with ACK	-
Send - with priority	-
Send – OOB (out of band)	-
Send - with timeout	-
Send – broadcast	-
Timestamp	-
Notify	-

Mailbox	NO
Set maximum message size	-
Get maximum message size	-
Share between processes	-
Send - with ACK	-
Send - with timeout	-
Send – broadcast	-
Receive – blocking	-
Receive – non blocking	-
Receive – with timeout	-
Get status	-

5.3 Development methodology

Linux development is preferred to be done on **Linux** hosts although it is not required that the host and target have the same CPU type or family. Using cross-compilation tool chains will solve that problem.

A major problem in **Linux** is that it is not a complete system; there is the kernel, the libraries, and the applications (like shells) in which the user should combine them all in a root file system. Compare this with FreeBSD for instance, which is delivered as a complete system!

Therefore creating an embedded target from scratch can be a daunting task! But for an x86 platform, it will not be probably that complicated. For other platforms, creating the cross GNU tool chain will take time. Sure you can use buildroot as a quick way to do this, but then this uses the **uClibc** library which does not have **NPTL** support yet! But even by using buildroot, it can take a lot of work and time to get something up and running.

Once everything is in its place, then there are different debugger tools available for remote debugging (gdbserver/gdb combo).

Concerning kernel debugging, debugging of kernel code with a remote host was possible (using KGDB) only since kernel version 2.6.26. Recently (2.6.36) using KDB is possible from an X environment. KDB is however not a source aware debugger. So things start to improve, but only recently.

5.4 Documentation

OS Documentation	0	▨	▨	▨	▨ 4							10

__Linux__ documentation is typical for open source projects: first of all you have the code as documentation… Luckily there are a lot of other sources of documentation as well. You have the Documentation directory in the kernel source tree. You have the kernel.org website, where also the latest man pages can be found.

So there is a lot of documentation available, but not in one nice package. It is scattered around on the internet. So it will take some time for a newbie to find his way in this. Surely once you need cross toolchains, real-time behaviour and so one, it is even harder to find the correct documentation.

5.5 OS Configuration

OS Configuration 0 ▨▨▨▨▨ 6 ☐☐☐☐ 10

*The basic kernel configuration tool (**make menuconfig**) is, once you know and understand it, easy to use and contains basic documentation on the features.*

The number of configurable items is huge: kernel (build and runtime) libraries, setup of root system and so on. Using wrong combinations can make your system fail to boot, but much worse it can lead to system instabilities.

It will take some time before you have enough experience to setup everything correctly.

Using the standard mechanisms to configure and setup everything will put you against a steep wall if you are a newbie. Once you have experience it becomes more trivial.

The ***Kconfig*** system used to configure the kernel is well structured and contains also help information. Once you know how the "***make menuconfig***" system works and how configurations are stored, it is not so difficult to change the kernel build configuration.

However, the number of optional configurations of your kernel is huge. So setting up a configuration can take a long time. Luckily there are defaults present for most common platforms.

Selecting the different required device support is also daunting because of the overwhelming number of modules available. Almost for any old hardware, you will find a driver. The only problem that could happen is with the newest parts (hardware devices) because it could take some time before the support for such devices is made available in the kernel.

Besides the kernel configuration at build time, there are different settings that may be changed at runtime (using the ***/proc/*** or ***/sys*** file systems). Again you will need to know what you are doing. Surely if you have specific requirements (like for instance real-time behaviour), it is important that these are set correctly. **Remember, the default settings are not usable for real-time systems.**

You will need to have at least an initial root file system to boot, and all the modules that are required to mount your root file system should be built into the kernel.

Finally, you need also to set-up a root file system for the different services and applications that need to be configured, for instance the network settings.

Be sure that building a system from scratch will take a lot of time. However, most board vendors already created ports for their boards, which can fasten the integration work.

5.6 Installation and BSP

Installation and BSP	0	▨▨▨▨ 4 □□□□□□	10

Besides configuring the kernel, a lot more is required to get your platform up and running.

Starting from scratch: you need to build the cross toolchain and setup a workable root file system. You need a lot of in-depth knowledge before this is feasible to do. So a newbie will have to tackle a steep wall before succeeding.

Luckily, most board vendors will deliver their board with the needed environment and tool chains, which makes it easier to start.

5.7 Internet Support

Internet support	0	▨▨▨▨▨▨▨▨▨▨	10

No problems here, whatever you want should be possible...

5.8 Development tools

Development tools	0	6	10

Linux *(the kernel) on its own does not have graphical tools.*

Hackers will typically use text based tools like vim, ctags, gdb, git. For open source IDEs you will need to take a look to Eclipse.

There are a lot of different tools available (for instance Emacs to name one), but these are not all integrated.

As there are so many open source projects out there, you can find for almost anything a solution. For an integrated IDE only Eclipse is a candidate. For most others multiple alternatives are possible.

	Present (Yes/No)	Integrated in IDE (Eclipse)	Standalone	Command based	GUI based
Editor					
Color highlight	Yes	✓	✓	✓	✓
Integrated help	Yes	✓			✓
Automatic code layout	Yes	✓			✓
Compiler					
C	Yes	✓	✓	✓	✓
C++	Yes	✓	✓	✓	✓
Java	Yes	✓	✓	✓	✓
Ada	Yes				
Assembler	Yes	✓	✓	✓	✓
Linker		✓	✓	✓	✓
Incremental	Yes	✓	✓	✓	✓
Symbol table generation	Yes	✓	✓	✓	✓
Development tools					
Profiler	Yes	✓	✓	✓	✓

	Present (Yes/No)	Integrated in IDE (Eclipse)	Standalone	Command based	GUI based
Project management	**No**				
Source code control	**Yes**	✓	✓	✓	✓
Revision control	**Yes**	✓	✓	✓	✓
Debugger					
Symbolic debugger	**Yes**	✓	✓	✓	✓
Thread sensitive debugging	**Yes**	✓	✓	✓	✓
Mixed source and disassembly	**Yes**	✓	✓	✓	✓
Variable inspect	**Yes**	✓	✓	✓	✓
Structure inspect	**Yes**	✓	✓	✓	✓
Memory inspect	**Yes**	✓	✓	✓	✓
Register inspect	**Yes**	✓	✓	✓	✓
Target connection					
JTAG	**No**				
BDM	**No**				
Serial (via ROM-Monitor or app?)	**Yes**	✓	✓	✓	✓
Network (via ROM-Monitor or app?)	**Yes**	✓	✓	✓	✓
System analysis tool					
Tracing	**Yes**	✓	✓	✓	✓
Thread information	**Yes**	✓	✓	✓	✓
Interrupt information	**Yes**	✓	✓	✓	✓
Loader (for instance uboot)					

	Present (Yes/No)	Integrated in IDE (Eclipse)	Standalone	Command based	GUI based
TFTP boot loader	Yes	✓	✓	✓	✓
Serial boot loader	Yes	✓	✓	✓	✓
Load separate modules	No				

Paragraphs may be started with a "smiley" depending on how the writer interprets the paragraph:

☺ This is something I like…

☹ This is some aspect that I don't like at all…

💣 This is clearly a major problem (and which invalidates the "RT-VALIDATED" logo)

☠ This is clearly a major problem (and which causes the "DID NOT QUALIFY" logo)