Faculdade de Ciências da Universidade de Lisboa

Instituto Superior Técnico

DARIO: Distributed Agency for Reliable Input/Output

Project FCT POSC/EIA/56041/2004

Bullet Linux: a Real-Time Platform for Industrial and Aerospace Embedded Applications

DARIO Technical Report RT-08-02

J. Craveiro

June 2008

Faculdade de Ciências da Universidade de Lisboa

Instituto Superior Técnico

Bullet Linux: a Real-Time Platform for Industrial and Aerospace Embedded Applications

To be submitted for publication: please do not distribute

Technical Report: DARIO RT-08-02
Authors: J. Craveiro
Date: June 2008

This work was partially supported by the FCT through Projects POSC/EIA/56041/2004 (DARIO) and the Large-Scale Informatic Systems Laboratory (LASIGE).

LIMITED DISTRIBUTION NOTICE
This report may have been submitted for publication. In view of copyright protection in case it is accepted for publication, its distribution is limited to peer communications and specific requests.
©2008, Project DARIO - Distributed Agency for Reliable Input/Output.

Bullet Linux: a Real-Time Platform for Industrial and Aerospace Embedded Applications

João Craveiro
LaSIGE / FCUL
jcraveiro@lasige.di.fc.ul.pt

Abstract— **In the field of embedded systems, one has to cope with: the scarcity of resources typical in such devices; and the diversity of existent hardware (processors, network interfaces). A balance between how much featured an operating system solution should be to address both issues must be reached. In that sense, we are evaluating GNU/Linux as a solution for embedded systems that addresses this balance. For such, we show the genesis of such a solution, the Bullet Linux, in three main vectors: kernel, system library, and system tools. We also extend the suitability of Bullet Linux to: embedded systems with real-time requirements, through the integration of RTAI; and to architectures presenting spatial and temporal partitioning, like ARINC 653, a standard for aerospace applications. Adding GNU/Linux to such an architecture aims to allow running existent applications or interpreted scripts without the need for a port of the application or interpreter to an RTOS.**

I. INTRODUCTION

Embedded systems, and particularly industrial embedded systems, are often physically small in order to fit in small places, to be easily carried or to be hidden. One of the problems the world of embedded systems faces is the lack of computing resources relatively to larger computers. These small systems can also be composed of a wide variety of hardware devices, like different CPU architectures, different network interface controllers, different types of mass storage and a set of other application-specific devices. A wise choice of operating system must be made in order to balance functionality and available resources.

In this work, a case is made for a GNU/Linux system (the Bullet Linux) to serve this purpose, and the foundations are laid to extend the suitability of this system to more demanding environments. The paper is organised as follows:

The paper is organised as follows. Section II describes of the state of the art. Section III describes of the build tools and process, and the obtained results (namely, size). Section IV presents the analysis of results against a typical GNU/Linux distribution, in terms of overall size and functionality. Section V studies the suitability of Bullet Linux to satisfy real-time requirements, including the incorporation in an architecture compliant with the ARINC 653 specification. Section VI concludes the paper.

II. STATE OF THE ART

Linux is an open source operating system kernel available free of charge and maintained by developers from all the world. The source code is accessible for everyone and people are encouraged to contribute with their own code. For this reason, the Linux kernel is extremely portable between computer architectures and supports a massive variety of hardware devices. And there's always space for more.

The **Linux kernel 2.6** brought fundamental improvements over 2.4 for embedded system and towards real-time systems. A new preemption system allows the preemption of kernel tasks, where user applications are no longer locked until the end of all pending system calls before they can continue executing. This significantly reduces the latency of user applications and increases the overall system responsiveness. Along with a new $\mathcal{O}(1)$ scheduler [1], the preemption of kernel tasks resulted in a performance improvement and better user interactivity.

This was further improved with a newer scheduler, the **Completely Fair Scheduler (CFS)**, which wields $\mathcal{O}(1)$ complexity for choosing a task and $\mathcal{O}(\log n)$ for reinserting a task after it has executed. This is accomplished by the substitution of runqueues for a red–black tree (a self-balanced binary search tree): the elements are sorted so as to reflect the timeline of future execution, thus choosing a task is a matter of picking the root of the tree (which is independent from the numbers of tasks therein). The CFS also uses nanosecond granularity accounting, abandoning the notion of timeslices and the need for specific process interactivity heuristics; it offers, though, the possibility to tune the scheduler to accomplish workloads suitable for desktop (low latency — default) or server (good batching) environments.

Another novelty introduced in the 2.6 line of the Linux kernel was the **Completely Fair Queueing (CFQ) I/O scheduler**, which applies concepts from network scheduling to disk scheduling: it maintains I/O queues per process, and attempts to distribute the available I/O bandwidth equally among all I/O requests. CFQ presents similar bandwidth results than the previously used Anticipatory Scheduler (AS), but lower latency, and has replaced AS as the default in the Linux kernel from version 2.6.18 onwards (although Red Hat, two years earlier, had included in RHEL 4 a 2.6.9 kernel configured with CFQ as default).

While this is suitable for suitable for soft responsiveness requirements, real-time systems may further benefit from another I/O scheduler available in the Linux kernel: the **Deadline I/O scheduler**, which aims to guarantee that an I/O request begins to be served at a specified time. The Deadline I/O scheduler performs inferiorly than CFQ in balancing transactions across multiple drives or file systems, which is, by principle, a less

common to encounter in embedded systems [2].

Versions 2.6.16 to 2.6.24 also saw components of the PREEMPT_RT patchset being merged into the kernel mainline. This includes userspace priority inheritance, high-resolution timers, threaded interrupt handlers, sleeping spinlocks, and full kernel preemptibility.

An increased modularity allows one to easily select the smallest set of features required for each system, avoiding unnecessary code. For systems with limited resources this is very important. Additionally, the added support for a wider variety of hardware devices, computer architectures and the improved build tools help enhance the pace of development of the kernel itself. This flexibility makes Linux, and specially Linux 2.6, a good choice for real-time embedded systems.

III. Bullet Linux

There are some GNU/Linux distributions available for embedded systems but none of those are exactly what we need. Some are commercial or targeted at a specific type of device. Others simply have too much unnecessary features or already have their own kernel modifications. Balancing that and the ease of setting up a new Linux operating system, we analysed how to build a specific (and further ahead, real-time ready) version of a Linux operating system targeted for embedded systems and applications. The main vectors for achieving an effective balance between functionality and available resources are: configuration of the Linux kernel, use of a smaller system library, and provision of the standard Unix utilities and tools in a more resource-efficient way.

A. Linux kernel 2.6

An example of the configuration process for the Linux kernel is depicted in Figure 1 [3]. Exploiting the configurability of the Linux kernel, it became possible to produce a small kernel image for Bullet Linux, as shown in Figure 2.

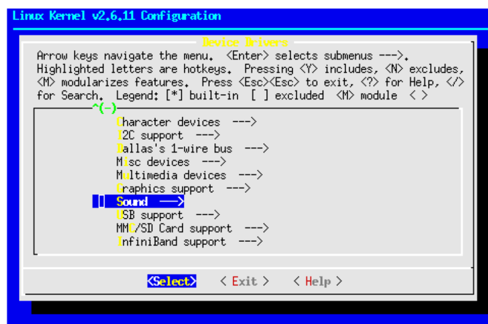
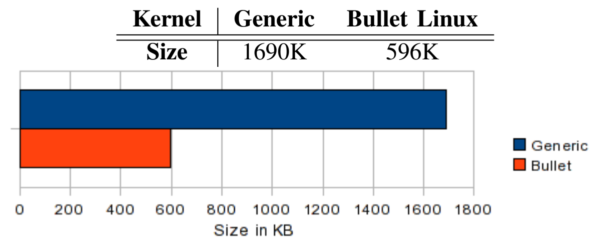


Fig. 1. Kernel configuration screen

The kernel for Bullet Linux is built from a standard, unpatched source tree, exactly as distributed by the developers. The absence of customised patches ensures easier upgradeability and less compatibility issues between different versions.

Independently to building a working minimal Linux distribution for embedded systems, work was made to account how much size could be spared or required depending on how the



Kernel	Generic	Bullet Linux
Size	1690K	596K

Fig. 2. Size comparison between a generic Linux Kernel and Bullet Linux Kernel

TABLE I

Kernel (vmlinuz) size comparison

Configuration	Size
Bare	596K
Bare + Network	872K
Bare + Network + 802.11	911K
Bare + Network + Bluetooth	939K
Bare + Network + USB	974K
Bare + Network + IRDA	1010K

kernel is configured. Thus, a handful of kernel configuration profiles were surveyed. The base, what we'll call here a "bare kernel", is a kernel with the bare minimum to function — basic device support, no network or any kind of peripheral connection, no peripherals or multimedia device drivers, and support for no more than the basic file system types.

The additional features that were tested for kernel size footprint were:

- network support (with basic/generic network interface drivers);
- 802.11 stack support (no WPA though, only WEP);
- Bluetooth support;
- USB (Universal Serial Bus) support;
- IrDA support (including some infrared dongle drivers).

The results obtained are that on Table I. For comparison sake, they refer only to vmlinuz — the statically linked executable file that contains the Linux kernel image.

B. A small system library

One of the most important components of a UNIX like system is the system library. The most used system library is the C library GNU libc. However, there are alternative design options to this library that are more appropriate for embedded systems. **uClibc** is a C library specially developed for embedded systems, which we found adequate among the available options. It features almost all GNU libc functionality but it's small size makes it appropriate for system with low resources. This was accomplished by rewriting the code with size optimisations in mind and by modularising some functionalities which allows the configuration of the uClibc library adapting it to the requirements of the target system.

There are alternative small footprint C libraries available, such as **newlib** (created by Cygnus, of eCos fame, and now maintained by Red Hat developers) and **diet libc** (a more

recent initiative that only implements a subset with the most important and commonly used features). uClibc was chosen for its maturity and for how well other tools used integrate with it (BusyBox, Buildroot).

The use of uClibc allowed Bullet Linux to keep a small size, when comparing against the use of a standard GNU libc. Figure 3 shows the immediate advantages brought by uClibc. The compilation of GNU/Linux utilities and tools against uClibc brings, then, added benefits.

Library	GNU libc 2.4	uClibc 0.9.29
Size	1828K	448K

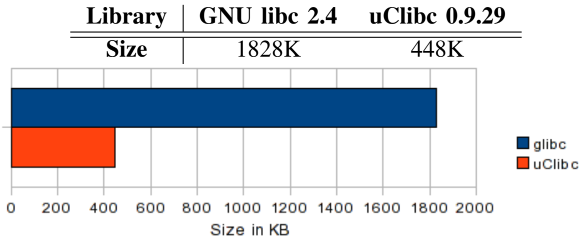


Fig. 3. Size comparison between a standard GNU libc and uClibc

C. GNU/Linux utilities and tools

To complete a Linux based operating system, we need some well known tools on Linux and UNIX systems that everyone expects to find. For this purpose we selected **BusyBox** as a set of those tools bundled together in a single file. These tools were rewritten to be smaller than their original counterparts. This is accomplished by code optimisations and by the absence of some of the features offered although maintaining the most important functions.

BusyBox is also highly modular and configurable, as shown in Figure 4: it provides a wide array of tools that can be included at this stage, some if which can be fine tuned as to only included a part of the available options (e.g., tar with no support for the -j option to handle .tar.bz2 files).

The tools made available by BusyBox include, but are not limited to:

Archival utilities b(un)zip2, g(un)zip, tar
Core utilities cat, chgrp, chmod, chown, chroot, cmp, cp, dd, df, diff, du, echo, head, install, ls, mkdir, mknod, nice, pwd, rm, sha1sum, seq, sleep, sort, tail, touch, uname, wc
Console utilities clear, reset
Editors awk, ed, patch, sed, vi
Search utilities find, grep, xargs
Init utilities init, poweroff, halt, reboot, mesg
Login/password management addgroup, delgroup, adduser, deluser, getty, login, passwd, su, sulogin
Linux Ext2 programs e2fsck, mke2fs
Linux module utilities insmod, lsmod, rmmod, modprobe
Linux system utilites dmesg, fdisk, more, (u)mount, support for loopback mounts
Miscelaneous crond, crontab, eject, less, time
Networking utilities hostname, httpd, ifconfig, ping, route, telnet, tftp, traceroute, wget
Process utilities free, kill(all), ps, top, uptime
Shells ash (highly configurable), hush, lash, msh.
System logging syslogd, logger

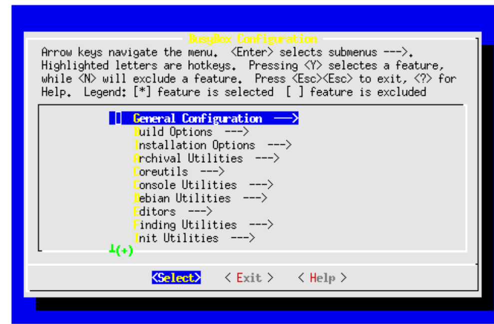


Fig. 4. BusyBox configuration screen

Even using shared libraries, standard GNU tools can use a lot of space, which is a real problem when dealing with resources shortage. BusyBox by itself allows a notable size gain and, when compiled against uClibc extends that margin even further. Figure 5 shows that size gain obtained from the usage of BusyBox.

Toolset	Standard GNU tools	BusyBox 1.5.0-svn
Size	8297K	264K

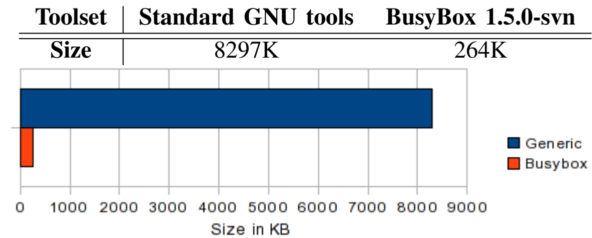


Fig. 5. Size comparison between a set of standard GNU Linux tools and busybox

D. Building process

By putting together a specially configured Linux kernel 2.6 for our prototype systems, a smaller system library and a restricted set of system tools, it became possible to build an entire GNU/Linux operating system that can fit on an 1,44MB floppy disk.

The kernel is compiled from unpatched sources with a specific configuration for the existing devices and interfaces of the prototype systems (i386 based, ethernet network, no hard disk drive). **Buildroot** is used to facilitate the configuration and build process of the uClibc system library and the BusyBox toolset; it configures builds and prepares the cross compiler environment for the later build of the system library and toolset (an example of the configurability of Buildroot is shown in the Figure 6). This cross compiling environment is necessary because the target architecture for Bullet Linux may be different from the architecture of the build system. The system library and toolset are also tailored to be built as small as possible, while maintaining all the important functionalities.

Bullet Linux is extremely customisable, due to its main components' flexibility and modularity. The kernel is configurable and very modular, buildroot permits the customisation

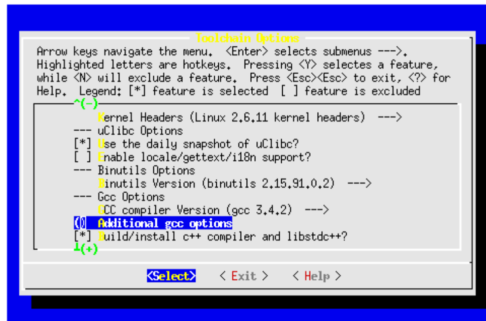


Fig. 6. Buildroot configuration screen

of the build environment and partial configuration of the system library and toolset. Further refining of the system library and toolset is also possible by doing so directly from their respective source tree.

E. Remote booting Bullet Linux

Remote booting is the process of booting an operating system that resides on a different system. For instance, a central server maintains one or more operating system image files and sends them, by request, to clients. This is very useful in situations where:

1) the embedded system does not have a permanent storage device like a hard disk, or;
2) it is important to have a fresh unaltered operating system each time the computer boots.

The embedded system can request the operating system and load it from the network directly to RAM. This can be done, for example, by using the PXE protocol (Pre-boot eXecution Environment), an industry standard protocol that enables PXE ready clients to use a network interface for remote boot. The protocol uses DHCP (or BOOTP) for client network configuration and to distribute the operating system file name and file server address. The clients will then download the operating system using TFTP from the file server.

Project Etherboot is an effort to maintain a free and open source software package that implements a network boot process which is compatible with the PXE protocol and can be used to load and boot Bullet Linux remotely. A Bullet Linux image file must first be prepared and available for download in a TFTP server. The image file is in the ELF (Executable and Linking Format) and contains a kernel and, optionally, a file system. The DHCP server must also be configured to be used in a remote boot environment. This configurations consists on adding some DHCP options with the address of the TFTP server and the name of the Bullet Linux file.

Etherboot can be either compiled from source or used in one of the pre-compiled versions available on-line. The binary can then be stored on a bootable device like a flopy disk or an EPROM that can be found in many interface adaptors. By burning an Etherboot binary into an EPROM of an embedded system, the booting process will then follow the aforementioned steps:

1) The embedded system loads Etherboot software from the EPROM
2) The Etherboot software implements a small DHCP client with which it broadcasts a discovery message
3) A DHCP server responds to this message offering:
   - network configuration parameters;
   - a file server address, and;
   - the file name of the Bullet Linux image.
4) The client can then acknowledge the offer and download the file using TFTP
5) The file is then loaded and used to start Bullet Linux

After loading Bullet Linux, the embedded system can continue its normal function.

IV. Analysis

A. Overall size analysis

As referred before, one of the main purposes of Bullet Linux is to be a small operating system. A targeted and well defined functionality with little or no overhead or extras. Bullet Linux was built with Kernel version 2.6.19.2, uClibc 0.9.29 and BusyBox 1.5.0-svn (2007-01-24) compiled against uClibc 0.9.29.

All the main components of Bullet Linux exist on a desktop distribution, but a desktop distribution is far from being comparable to Bullet Linux. The size gain of Bullet Linux can be analysed by comparing each of its components individually with the equivalent in a desktop distribution. Typically, a desktop distribution is built with standard or lightly patched Kernel compiled with a modular approach; a modular Linux Kernel is composed of an image plus a set of files that correspond to different modules. Modules are loaded into memory only if considered necessary by the system or the user. A typical modular kernel has an image size slightly above 1.5M and a set of modules with about 15M. The system library is a fully featured GNU libc 2.X (libc 6) along with many other smaller less generic libraries. The set of tools in a desktop distribution are compiled against it's glibc and occupies the biggest slice of storage space: it can reach several gigabytes.

Globally, Figure 7 summarises the analysis of size in two distinct situations: a generic Linux distribution and Bullet Linux.

B. Functionality

Like it has previsouly been said, the overall size benefit does not come without possible decreases in available functionality. Besides the lower portability that derives from excluding some device drivers and support for some file system types, this decrease can be specially be observed at the utilities level, in two main fronts: shell, and core utilities. There is also some extra functionality than can be brought into Bullet Linux: support for interpreted/scripting languages. This is particularly interesting to take advantage of the benefits of integrating GNU/Linux in a ARINC 653 based architecture, as will be detailed in Section V-B.

Linux Kernel	Generic Bullet	1690K 596K
System Library	glibc 2.4 uClibc 0.9.29	1828K 448K
System Tools	Generic BusyBox 1.5.0-svn	8297K 264K

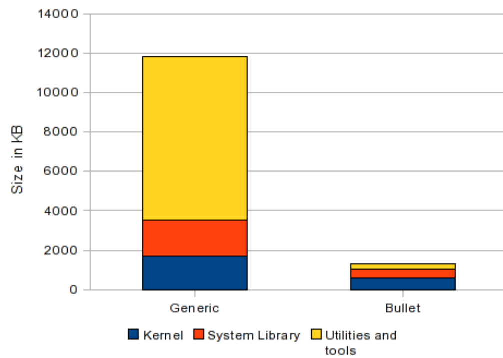


Fig. 7. Overall size comparison

1) Shell: BusyBox provides a few shell options, the most sophisticate of which is the Almquist Shell (ash). Although compatible with the Bourne shell and suitable for low memory systems, ash lacks some extras provided by other shells, like the ubiquitous Bourne Again Shell (bash). By equipping a system with only the Almquist Shell, one has to be conscious that most of the scripts one might use in a standard GNU/Linux system use bash-specific syntax constructs, and will need to be adapted in such a case. One solution, with the downside of demanding a slightly more resourceful embedded system, is to compile the Bourne Again Shell against the same uClibc toolchain, and included it the image; this can be automated by enabling the inclusion of the 'bash' package when configuring Buildroot (Section III-D).

2) Core utilities: Another issue also affecting compatibility with existent scripts is the choice of core utilities to include when configuring BusyBox (Section III-C). Several scripts may rely (even if just in one single line) on the assumption that a certain utility is present — which may be the same utility one deemed irrelevant when configuring BusyBox. This has happened during the experiments described in this paper. The inclusion of the seq core utility (which prints a sequence of numbers) was overlooked when first building Bullet Linux; this would reveal a limitation, when auxiliary scripts related to RTAI and to the performed latency test required such an utility (namely, for instance, to create device nodes /dev/rtf0 to /dev/rtf9).

3) Interpreted/scripting languages: The size reduction driven approach to Bullet Linux (i.e., including the most of the required tools in the BusyBox executable, and the least possible as fully-fledged independent tools) leaves out the support for interpreted/scripting languages. BusyBox does not support any of these interpreters (save for the aforementioned shells), so support must be added at an increased sacrifice of disk space. Buildroot provides the possibility to add support for some of them at configuration time; the available packages are:

- microperl (Perl without OS-specific functions like crypt and readdir, aimed at being built on as many machines as possible);
- python;
- ruby, and
- tcl.

V. Guaranteeing real-time operation

A. Bullet Linux with RTAI

RTAI (Real-Time Application Interface) is an open source project for providing a Linux-based operating system with real-time capabilities. RTAI employs a modified version of ADEOS, which proposes a nanokernel hardware abstraction layer (HAL) approach. In this approach, the HAL provides the mechanism to pass interrupts to a real-time operating system (not implementing real-time capabilities on its own) [4]. The global RTAI architecture and integration with the Linux kernel is portraied in Figure 8.

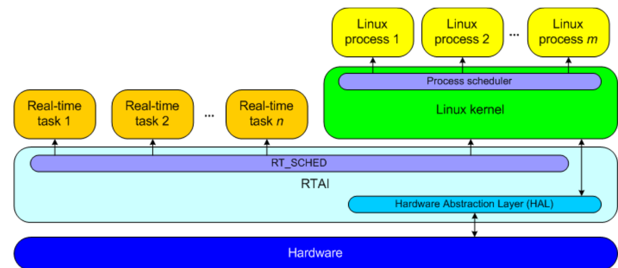


Fig. 8. RTAI architecture

1) Modules: RTAI presents a module-oriented structure, with its funcionality spread among a few kernel modules. Currently, there are five core modules, each with clear capabilities as follows:

RTAI Core module that offers the basic real-time services of interrupt and timer handling — replacing Linux in doing so.

RTAI_SCHED Real-time task scheduling and management (creation/destruction), plus synchronization and communication between real-time tasks. RTAI_SCHED offers support for both periodic (predefined time slice) and one-shot (time slice for the following run is defined at the end of the current one) scheduling modes.

RTAI_FIFOS The basis of communication between real-time tasks and Linux processes, through FIFO ("First In, First Out") channels.

RTAI_SHM Shared memory between real-time tasks and Linux processes.

LXRT (LinuX Real-Time) Allows all services provided by RTAI, and its schedulers, to be used in user space (thus providing a way to create user mode real-time tasks)

2) Kernel vs. user mode: When creating a real-time task, the choice between a kernel and a user mode one is somewhat personal, albeit driven by technical considerations. On the one hand, kernel mode tasks allow for minimum latency and can be run at a higher frequency. On the other hand, user mode tasks integrate better with the surrounding GNU/Linux world, for they can access more resources. Hardware access is granted to both approaches, but if one needs to handle hardware interrupts, it is preferable to use a kernel mode task (despite the ISR support in user space given by RTAI) [5].

3) Experimental build: To start the construction of a real-time Linux for embedded systems, Bullet Linux was built with a slightly different set of tools (namely, their versions) and requirements: Linux kernel 2.6.19.2 (submitted to the HAL patch distributed with RTAI), uClibc 0.9.28, and Busybox 1.2.2.1 (compiled against uClibc 0.9.28, and with a few additional required built-in applets).

The version of RTAI used was 3.5, and was compiled against the same uClibc 0.9.28 toolchain as Busybox. The combined solution was verified as working with a specific test included in the RTAI workbench. This test verifies task switching latency by means of measuring the difference between the expected and actual switch times of a task as performed by the RTAI scheduler.

B. GNU/Linux in a partitioned architecture

The AIR architecture, based on the ARINC 653 standard [6] (defined for aeronautical applications which is being adapted to aerospace applications as well), specifies an architecture presenting spatial and temporal segregation, and allows for the use of different operating systems in different partitions [7]. A broad overview of the AIR architecture is represented in Figure 9.

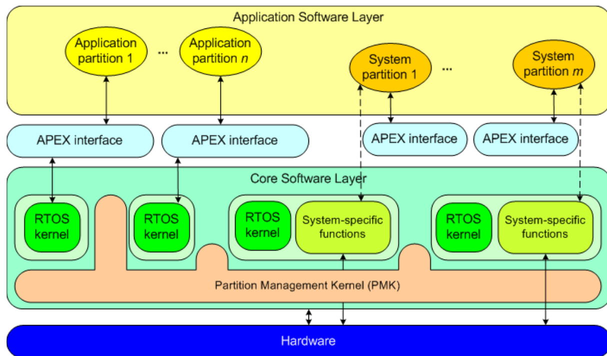


Fig. 9. Overview of the AIR architecture

1) Spatial segregation: The forementioned ARINC 653 partition concept ensures, by design, that an application does not corrupt memory or data of an application running in a different partition. Nevertheless, the specification only mandates and has for granted that this spatial segretation is observed. The operating system is required to ensure this, in a way that is not provided by ARINC 653 [6], [8].

Securing this spatial segregation requires specific mechanisms that may be implemented in a hardware memory management unit (MMU). An important issue is that SPARC LEON processors, out of the two common ways to achieve memory protection (segmentation and paging), feature only paging.

2) Temporal segregation: The notion of having one partition's activies not interfere with the timing of the others' activies is conceptually ensured, in ARINC 653, by a fixed, cycle-based partition scheduling. Each partition is assigned one or more time windows inside a Major Time Frame (MTF) with a fixed duration; the partition agenda for that MTF is defined off-line, and periodically repeated [6].

In the AIR architecture, temporal segregation is ensured by the AIR partition scheduler [7]. Work is being done towards the possibility of defining (off-line) several static agendas, and allowing the agenda to be dynamically (on-line) chosen, to be applied a the beginning of the next MTF.

3) Why GNU/Linux: Porting applications from Linux to one of the RTOS one might be using (RTEMS, eCos, VxWorks, etc.) can be a complicated task, and definitely not an error-free one [9]. To address this portability issue, we are evaluating the approach of having one partition of such a system run GNU/Linux. That partition interfaces with the AIR PMK (Partition Management Kernel) through a subset of the AIR APEX (Application Executive) Interface. This integration is shown in Figure 10.

In this scenario, existent applications for GNU/Linux can be used or created without the further effort of having to port them to a particular RTOS. Furthermore, the benefits of scripting languages widely used in the GNU/Linux world, like Perl, Python, or Tcl, can be brought into scene, something which would otherwise depend on a port of the intepreter to a particular RTOS.

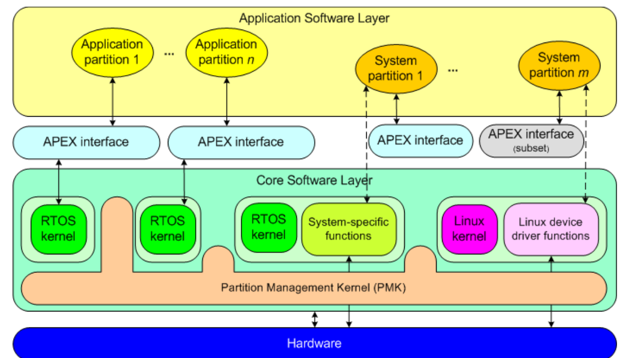


Fig. 10. Overview of the intended incorporation in the AIR system architecture

4) Current direction and future work: The issues regarding the assurance of spatial segregation with paging only are currently being addressed. The solution being envisioned consists of using assembler directives to align data sections (code, data and stack) to 4K page boundaries.

In the scope of the integration of Bullet Linux, evaluation is currently being done on whether the eventual employment

of a non-real-time GNU/Linux solution will not contaminate the real-time properties of the whole system. One important issue in this regards is process scheduling (in this case, inside a given partition); for this matter, ARINC 653 only mandates the following:

1) One of the main activities of the operating system is the arbitration of several processes inside the partition wanting to acquire the processor.
2) Each process has a priority.
3) The scheduling algorithm is priority preemptive. When rescheduling, the "ready" process with the highest priority receives the processor (among several processes with the highest priority, the oldest is selected).
4) Periodic and aperiodic scheduling of processes is supported (although telling both apart from each other is not required).
5) All the processes share the resources allocated to the partition.

Bullet Linux, even without the addition of a real-time executive like RTAI, seems up to the par for this requirements (for instance, a priority-based preemptive scheduler can be chosen at kernel configuration time). An even more concerning issue to be dealt with is time management.

C. Control of event handling

In most embedded systems, like Bullet Linux, external events trigger an interruption so that the application can recognize that something has changed in the real world. Unlike tasks, scheduled by a priority-based algorithm, interrupts are immediately processed, postponing any other processor activity (except perhaps, other interrupts). This can cause severe temporal interference to the running tasks because there is no standard mechanism capable of accounting and controlling interrupt processing.

Uncontrolled interrupt processing is especially dangerous in applications with stringent time and fault constraints. In real-time systems, the standard approach is to accommodate the event generation worst case scenario into the scheduling conditions. Because of the simplifications made, this is in general too pessimistic and the system ends up with an over-dimensioned system [10]. Moreover, a simple integration of the worst case scenario is not desirable because, in some systems it may not be even possible to determine and even if it is, in presence of faults a huge amount of interruptions can overload the processor, causing missed deadlines in some, or even all, tasks. As an example, the lunar landing system that almost failed due to a disconnected device that flooded the system with interrupts [11].

Following a previous work [12], Bullet Linux should integrated a specialised component to supervise and limit the rate of interrupt generation.

VI. Conclusions

In this work, we reach to an understanding that Linux kernel can be used as the basis for a fully functional operating system for embedded systems with scarce storage resources, with the (also assessed) possibility of adding real-time capabilities through existent real-time executives (RTAI, Xenomai).

This solution is also being envisioned to be integrated in a segregated (in time and space) system, so as to be able to run non-critical applications existent for GNU/Linux, or previously developed scripts in interpreted languages (Python, Perl, . . .), without having to port the application or the interpreters to the RTOS used in the other partitions. Further work is being initiaded, and includes:

1) consolidating the architecture;
2) building a prototype, and;
3) testing/evaluating the prototype.

Acknowledgment

This work was partially supported by EU and FCT through Project POSC/EIA/56041/2004 (DARIO) and through the FCT Multiannual Funding Program.
Project DARIO Web site — http://dario.di.fc.ul.pt/.

References

[1] J. Aas, "Understanding the Linux 2.6.8.1 CPU scheduler," Feb. 2005.
[2] D. J. Shakshober, "Choosing an I/O scheduler for Red Hat Enterprise Linux 4 and the 2.6 kernel," *Red Hat Magazine*, no. 8, June 2005. [Online]. Available: http://www.redhat.com/magazine/008jun05/features/schedulers/
[3] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 2nd ed. O'Reilly, December 2002.
[4] A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, and C. Taliercio, "Performance comparison of VxWorks, Linux, RTAI, and Xenomai in a hard real-time application," *Nuclear Science, IEEE Transactions on*, vol. 55, no. 1, pp. 435–439, Feb. 2008.
[5] G. Racciu and P. Mantegazza, *RTAI 3.4 User Manual, rev. 0.3*, Oct. 2006.
[6] Airlines Electronic Enginerring Committee (AEEC), *Avionics Application Software Standard Interface (ARINC Specification 653 - Part 1 (Supplement 2), required services)*. ARINC, Inc., 2006.
[7] J. Rufino, S. Filipe, M. Coutinho, S. Santos, and J. Windsor, "ARINC 653 interface in RTEMS," in *Data Systems in Aerospace (DASIA 2007)*, May 2007.
[8] J. Rufino and N. Diniz, "ARINC 653 in space," in *Data Systems in Aerospace (DASIA 2005)*, June 2005.
[9] L. M. Kinnan, "Application migration from Linux prototype to deployable IMA platform using ARINC 653 and Open GL," *Digital Avionics Systems Conference, 2007. DASC '07. IEEE/AIAA 26th*, Oct. 2007.
[10] K. Sandstrom, C. Eriksson, and G. Fohler, "Handling interrupts with static scheduling in an automotive vehiclecontrol system," in *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications*. Hiroshima, Japan: IEEE, Oct. 1998, pp. 158–165.
[11] J. Regehr and U. Duongsaa, "Preventing interrupt overload," in *Proc. of the ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2005)*, Chicago, IL, June 2005.
[12] M. Coutinho, J. Rufino, and C. Almeida, "Control of event handling timeliness in RTEMS," in *Proceedings of the 17th IASTED International Conference on Paralel and Distributed Computing Systems - PDCS 2005*. Phoenix, Arizona, USA: IASTED, November 2005.