Porting Embedded Systems to uClinux

António José da Silva
Instituto Superior Técnico
Av. Rovisco Pais
1049-001 Lisboa, Portugal
antonio.jose.silva@ist.utl.pt

ABSTRACT
The emergence of embedded computing in our daily lives has made the design and development of embedded applications into one of the crucial factors for embedded systems. Given the diversity of currently available applications, not only for embedded, but also for general purpose systems, it will be important to easily reuse part, if not all, of these applications in future and current products. The widespread interest and enthusiasm generated by Linux’s successful use in a number of embedded systems has made it into a strong candidate for defining a common development basis for embedded applications. In this paper, a detailed porting guide to uClinux using the XTran-3[20] board, an embedded system designed by Tecmic, is presented. The ported system performance was evaluated by several benchmark tools. The preliminary results looked promising, but there is still room for improvement. Hence, some ideas on how to port the entire system are suggested.

General Terms
Porting, uClinux, Embedded

Keywords
Linux, Operating System, Kernel, XTran-3, Hardware, Driver, Real Time Clock, Flash, JFFS 2, I2C

1. INTRODUCTION
An embedded system is a computer system that is designed to do a well defined set of tasks. Since the first embedded system, built in the early 1960s for the Apollo guidance system[29], other embedded systems were developed targeting different areas. Driven not only by hardware and software advances, but also by a deeper understanding of human needs and marketing possibilities, general purpose and embedded systems became more complex, implementing a growing set of functionalities. Today, there is a multitude of embedded systems available, from portable audio or video players to mobile phones with full capabilities, powerful gaming consoles, home network routers and the list continues, but still, there are some factors that consistently distinguish embedded from general purpose systems:

- Well defined response times (the system must answer to specific tasks without missing deadlines)
- Low hardware requirements (little RAM/ROM, task specific processor)
- Specific hardware dedicated to a well defined set of tasks
- Power Management constraints
- Low cost

Concerning response times, computer systems can be divided in soft and hard real time[26]. In soft real time systems, missing a deadline only degrades performance, unlike in hard real time systems. In hard real time systems, missing a time constraint before giving an answer may be worse than having no answer at all. An example of a soft real time system is a common DVD player. While good performance is desirable, missing time constraints in this type of system can lead to catastrophic scenarios! Like the first general purpose systems, the first embedded systems had no operating system. Since the first operating systems were implemented, their use in general purpose systems became mandatory, given their general purpose nature. On the other hand, because of the reduced feature set available in embedded systems, software that directly controlled the hardware available with very minimal or no multitasking capabilities continued to be used and developed in-house by embedded system vendors. Today, with a vast number of technologies available, many complex functionalities are expected even from the most basic embedded system. A good example are home network routers which began as relatively simple ethernet packet multiplexers and are now expected to have wifi and printer connectivity, packet routing, switching and forwarding capabilities and even firewall rules that can operate in all the OSI network layers. Several of these functionalities require similar software infrastructures that were first available only to general purpose systems.

The cost of implementing advanced functionalities in embedded systems using only proprietary in-house developed software is unfeasible for many companies, which tend to migrate to embedded operating systems, commercial or free. A real example of one of these companies is Tecmic, a Portuguese embedded solution provider. One of Tecmic’s products is the XTran-3 board, an embedded system designed by Tecmic, which is the hardware used in this study. The XTran-3 board is a vital part of Tecmic’s larger “XTran” system, conceived to remote fleet monitoring. In the XTran system, each board (from here on called XTran-3[20]) is installed and connected to each vehicle. Its function is to collect data about a vehicle’s mechanical status, global position and driver details and send it to a central computer probably located in the fleet’s headquarters. Currently, the XTran-3 board does all its tasks using a custom proprietary real time operating system developed by Tecmic.
In this paper, several commercial and free open source solutions are reviewed and compared with each other in terms of their porting to the XTran-3 board. Also, a detailed porting guide to uClinux using the XTran-3 board is presented, including all the relevant steps, from the development environment set up, to the kernel configuration, device driver implementation and application writing. Several benchmarks were also performed to compare the performance of the ported system with the Tecmic firmware and with other popular embedded solutions.

2. RELATED WORK

Although uClinux was already chosen as the operating system to port to the XTran-3 board, other candidate operating systems were also reviewed and analyzed. Besides in-house proprietary embedded operating systems, several commercial solutions began to appear around the mid-eighties, providing several different approaches to an extensible, multi-task and real time operating system for a wider non-specific range of embedded systems. Two notable examples of this type of solutions are QNX and VxWorks, which are still available today. Microsoft, one of the leaders in the operating system market, also implemented its own embedded operating system solution, Windows CE, aiming to provide a high degree of compatibility with its popular general purpose operating system APIs and applications[17]. With the advent of mobile phones in today’s society, several mobile phone providers, namely Nokia, Ericsson, Motorola and Psion, joined efforts to implement a special purpose operating system for those devices: the Symbian Operating System[16], which will also be described and compared. Besides Linux, which made its first appearance in an embedded system in 1998[19], other open source operating systems were ported to run on some embedded architectures, namely, Inferno Os, a descendent from “Plan 9 from Bell Labs” (which is itself a re-engineering of the Unix architecture) and Contiki, a very minimal operating system written initially for the Commodore 64 home console. eCos is an open source embedded operating system commonly used to bootstrap into other operating systems, including Linux. Its mature state and modularized architecture made it into a full featured Operating System and a promising candidate for running on the XTran-3 board.

2.1 Qualitative Approach to the Technical Evaluation

In order to qualitatively evaluate each candidate operating system to run on the target board, the following parameter list was considered sufficient for a basic overview of each projects’ positive and negative aspects[26]:

Install and Configuration Although each evaluated solution has its own configuration and installation process, there is a common denominator specific to the nature of embedded systems. This common denominator consists of the configuration, creation and deployment process of the system image file(s) from the host to the target system.

Task Management This parameter refers to the availability of scheduling policies and methodologies implemented in the evaluated operating systems. The existence of real time scheduling, processes and/or threading resources, preemption and granularity of I/O control mechanisms was also described and compared.

Memory Management The existence of MMU and paged memory support is compared, including page swapping methods and/or memory pool management. Also, the API richness for inter-process communication primitives also contributed as an important evaluation factor.

Interrupt Handling This parameter refers to the existence of different priority levels for interrupts, interrupt nesting and amount of resources available when an interrupt occurs.

Network Support Describes the amount of supported network protocols and device drivers. This parameter was introduced due to the growing importance of network support for embedded systems[29] and also for the XTran-3 board.

Development Tools Due to the different nature of each operating system, several manufacturers provided their own development and deployment tools. Others rely on a specific amount of open source tools. The variety and richness of the development resources is compared among each solution.

Applications The amount of existing applications for each operating system, open or closed source, is evaluated.

![Figure 1: Solution Comparison Scores](image-url)

Each solution was configured and installed in a test environment consisting of a typical PC (Pentium IV 3GHz with 4GB of ram) running Debian Linux and VMWare software for virtual machine emulation. Also, several operating system documentation considered relevant for this analysis was consulted[12][31][5][17][14][16][1][28][10][27][25][13]. Figure 1 shows the qualitative analysis results for each chosen operating system. The given score, ranging from 1 to 5,
is described as follows:

1. Immature project with reported unfixed critical errors
2. Immature project, used mainly for proof of concepts and constrained test environments
3. Mature project with known limitations and reduced functionalities. Small number of use cases in real life production environments
4. Mature project with known limitations and a large number of functionalities. Large number of use cases exist in real life production environments
5. No known critical errors, tested in real production environments. Very large number of functionalities

2.2 QNX

QNX is an Unix like real-time operating system. The QNX Neutrino kernel has a client-server architecture consisting of a microkernel[25] and optional cooperating processes. The microkernel implements only the core services like threads, processes, signals, message passing, synchronization, scheduling and timer services. The microkernel itself is never scheduled. Its code is executed only as the result of a kernel call, the occurrence of a hardware interrupt or a processor exception. Additional functionality is implemented in cooperative processes, which act as server processes and respond to the request of client processes (e.g. an application process). Examples of such server processes are the file system manager, server manager, device manager, network manager, etc. The architecture of QNX allows every application, driver, file system and protocol stack to run outside the kernel, in the safety of memory-protected user space. The interrupt redirector in the microkernel handles interrupts in their initial stage. It also supports interrupt sharing and nested interrupts. The QNX kernel provides all protection mechanisms as foreseen by the different Posix standards (mutexes, semaphores, conditional variables, read/write locks etc). In the test machine, all critical hardware was detected by the QNX installation process. Installation flavors range from a minimal floppy distribution or a full suite of applications bundled in a bootable cd. All tasks and processes are managed using FIFO or sporadic scheduling and have a priority number assigned. Memory management can be with or without MMU without loosing real time behaviour. QNX also supports IPv4 and IPv6 and has a built in web browser which can be included in the deployed image file.

To build custom QNX images, the System Builder application was used. This application is a graphical environment that runs under QNX windowing system and calculates all package dependencies upon the building process. It also has a dietitian feature that can shrink all libraries to contain only the functions used by the current set of applications. Kernel and application development can be done using the Momentics[24] suite which is an IDE based in Eclipse. Although there is a reduced number of applications available for QNX, there is a Java virtual machine implementation, full Posix compliance and several libraries available for development in different programming languages.

2.3 VxWorks

VxWorks 5 is a proprietary, real-time operating system developed by Wind River. VxWorks has been ported to a number of platforms and now runs on practically any modern CPU that is used in the embedded market. This includes the x86 family, MIPS, PowerPC, Freescale ColdFire, SH-4 and the closely related family of ARM, StrongARM and xScale CPUs. The VxWorks kernel features “protection domains” that provide containers for logical resources which define the execution environment, namely, the address space and accessed libraries. By defining containers for each application, the kernel can then intercept illegal accesses from the program being run, emulating by software the use of an MMU, thus making possible the use of memory protection even in mmu-less architectures. Like QNX, VxWorks is multi-threaded and has 256 priority levels for scheduling. It uses preemption and also a prioritized round robin scheduling[27]. Interrupts are handled in a special context outside any tasks context and have some restrictions for the kernel API primitives that can be used inside of it. VxWorks has extensive protocol stack support[27] and network support, including drivers and basic applications used in a networked environment (ssh clients and servers, http servers, etc)[27].

VxWorks comes with the Tornado IDE which comprises an extensive suite of tools and utilities that can be used during the development and debugging phase. However, these tools are integrated mainly using TCL scripts, which need to be parsed whenever an action is taken, making the process of interacting with the gui a tedious task, even in the test machine. Unlike QNX, VxWorks uses a host different from target approach. The host and target are two different machines linked together by serial, LAN or other bus, for communication. A special software called “debug agents” can run in the target system and provide the host with debugging information. VxWorks has a Java virtual machine implementation, is Posix compliant, and has a small number of applications available.

2.4 WinCE

Windows CE is Microsoft’s operating system for embedded systems and service oriented computers. It is supported on MIPS, Hitachi SuperH, ARM and Intel x86 architectures. Windows CE is built from a set of discrete modules, each divided into components that provide specific functionalities. The prime modules are the kernel, the object store, the graphics subsystem and communications components. In addition to these primary modules, other modules are available and provide support for multimedia, COM (Component Object Model), Windows CE shell and device manager. Like QNX and VxWorks, Windows CE uses processes and threads and has 256 levels of scheduling priority. The scheduler only uses round-robin with adjustable time-slices. The kernel has MMU support, but when this feature is enabled, real time performance is disabled[17]. For interrupt support, all service routines run in a special context that uses its own virtual addresses statically mapped by the OEM. API accessible from within the interrupt service routines is very limited, much more than in VxWorks, and an option is given to the OEMs to create a shared memory region by statically mapping a memory region into the interrupt service routine address space. Windows CE also has extensive network support and provides a HTTP server with ASP support, a minimalistic version of the Internet Explorer browser and multiple networking protocol support[17].

Windows CE development and deployment tools are for Windows operating system only, but their level of integration is excellent. The main development tool to use is Microsoft Visual Studio, for which many vendors provide plugins and extensions for their embedded systems. The number and variety of applications available for windows CE is very high and includes many applications written for the
Windows operating system that run without modification in Windows CE. Application developers are not only third party companies but also enthusiasts that need specific functions for their embedded system. Also, a minimal version of the .Net framework, the .Net compact framework, enables the use of the .Net programming language and tools in Windows CE.

2.5 Symbian

Symbian OS is a closed source proprietary operating system built exclusively for mobile phones. It is a preemption enabled multitasking operating system that uses processes, threads and hardware memory protection. As the result of a joint venture between several mobile companies, several innovative and unusual concepts were implemented in Symbian, whose code base was retrieved from Psion’s “Epoc” operating system[16], namely, resource descriptors, a cleanup stack operation and Active Objects. These Active Objects allow linking applications to events and can also be used as a tool to achieve cooperative scheduling[6]. Their use is destined to applications that spend most of its execution time in idle state, waking up upon certain events which may trigger, for example, the activation of several hardware resources. This approach allows the kernel to know that those are special types of system resources thus needing special scheduling policies. The kernel then chooses to divert processor cycles to some other more resource hungry tasks that may be concurrently executing, or even turn off the processor temporarily if no other tasks are executing. Controlling the scheduler indirectly using special types of kernel resources is a unique characteristic unseen in all other solutions reviewed in this paper. Symbian OS is entirely written in C++ and the development kit (which is available to Symbian partners only) has support for Python, Visual Basic and Java ME. The development kit also has a full featured windowing system, with its own font server, that allows all graphical applications to be integrated using a common framework. Symbian OS is also a direct competitor against Windows CE in the PDA and smartphone market and, like its competitor, it was designed having in mind a limited number of architectures, namely ARM and x86.

Symbian is very criticized because of its code base unmeasured growth since its appearance, with all the different vendors implementing their custom extensions instead of contributing to a more centralized architecture. This resulted in an astonishing amount of system and user classes, libraries and functions available, all following their different approaches to solve common problems[7]. Poor documentation is also a weak point of the Symbian operating system, a fact which, added to the huge size of the system API, makes many developers think of the development for the Symbian OS as a complex and frustrating task.

2.6 Linux

Linux is a free and open source operating system kernel. There are many distributions based on the Linux kernel. For the installation on the test machine, Debian Linux was chosen. Despite not being a hard real time kernel, Linux’s preemptive scheduler has O(1) complexity which makes it suitable for some real time applications[5]. Linux’s scheduler supports task queues, preemption under user and kernel mode and four scheduling classes: Normal (dynamic priorities), FIFO, round-robin and Batch. The Linux kernel also supports symmetric multi processor architectures without loosing any of the capabilities and system calls it has on single processor machines[9]. The Linux scheduler has various implementations, each one having unique properties and advantages under certain situations. This kind of flexibility, customization and resource richness available to the developers made the Linux kernel a popular choice adopted in embedded, desktop and server systems. Since version 2.6, the current major version, a considerable amount of new features and development methodologies made the development process much more agile and reactive to changes.

Linux supports up to 64GB of RAM in 32 bit architectures (although the hardware limitation of these type of architectures is still present: each process can only “see” 2GB) and more than 1024 GB in 64bit architectures. The page size is configurable in compilation time. For interrupts, Linux supports SoftIRQs, tasklets, bottom halves, task queues and timers[28]. One of the reasons for Linux’s growing popularity is its wide use in networking applications. Linux has support for a superset of all supported network protocols and drivers seen in the other operating systems that were analyzed in this paper. Linux implementation of network protocols and drivers are considered secure by many developers and administrators[8], who point at faulty user space daemons and applications (not at the kernel) when a Linux system gets compromised[2]. Linux is regarded as a reference in software development, mostly due to the impressive amount of tools available: all the GNU tools, cross platform emulators, compilers and even proprietary tools for development for proprietary closed platforms[31]. There is also an already big (and continuously growing) number of open source applications available, dispelling the myth that one cannot find an open source solution for a specific proprietary application[19]. This growth is also due to several investments in open source technologies by some major IT companies and hardware manufacturers that keep pushing Linux into the mainstream desktop, server and embedded market.

2.7 RTLinux

Linux has its roots in the desktop systems, but evolved as a general purpose operating system. The need for real time behaviour was felt after the first successful attempts for deployment in embedded systems occurred, in 1999[31]. From all the extensions that were implemented to add hard real time support to the Linux kernel, the most popular is the dual kernel approach. In this approach, the “normal” non real time kernel is treated as the lowest priority task of a real time micro kernel, which handles all real time tasks. The new real time kernel, RTLinux, can be compiled as a module and inserted or removed in execution time, adding or taking away real time capabilities from the running kernel. Although the dual kernel approach is invisible to applications, the tasks that run in the real time kernel always run in supervisor mode and do not take advantage of the tested and mature memory protection offered by an unmodified Linux kernel. Also, each process that runs in the new real time kernel is no longer a Linux process and must use the non-standard, Posix based, RTLinux API[11]. The new real time processes are never swapped out and run in the kernel address space, eliminating TLB misses (for those processes). Interrupts in the new kernel have an added flag that indicates the need for real time scheduling. Other interrupts are redirected for the non real time Linux task. As for drivers and supported protocols, the Linux implementation continues to be non-real time. Only the drivers and protocols developed for RTLinux have real time behaviour, hence, the
support is much poorer than the one found in the Linux
vanilla kernel.

Installing and configuring RTLinux is not a straight forward
process because it involves downloading, manually configura-
tion and compilation of the RTLinux module for later inser-
tion. This is comparable to the configuration of the Linux
vanilla kernel. Development tools and applications are the
same as the ones available for Linux.

2.8 uClinux

uClinux was initially a port from Linux 2.0.33 to MMU-
less architectures. Linux kernel was designed having Intel’s
80386 virtual paged memory in mind, which made its use
under processors without an MMU impossible[18]. When
no MMU is present, the operating system sees the entire
memory as a continuous block of readable/writable address
space. uClinux implements a memory management sys-
tem with several policies for incremental memory alloca-
tion, avoiding potential fragmentation problems that can
arise. Part of the new memory management system includes
a “page” implementation in order to maximize compatibil-
ity with the existing page oriented vanilla kernel and driver
code. Linux applications must be ported for running un-
der uClinux[19]. For C applications, this can be done us-
ing the uClibc and its libraries, which play a vital role
in the uClinux distribution. Upon the Linux kernel tran-
sition to the 2.6 branch, much of the code under uClinux
was merged in the vanilla tree, however, both are separate
projects and it is the uClinux team who needs to synchro-
nize the uClinux patches for new Linux releases. Currently,
a uClinux distribution contains a 2.2, 2.4 and 2.6 uC-kernel
branches, pre-configurations for many hardware vendors and
boards, uClibc and a large number of popular applications
which are already ported to uClinux. Like RTLinux and the
Linux vanilla, configuring a uClinux bootable image is a dif-
ficult process. The uClinux configuration system is buggy
and gives strange incompatibility and dependency results for
certain option combinations. The kernel capabilities remain
basically the same for interrupt and task management, only
with some minor details inherent to the target processor
used. The major difference resides in the memory manage-
ment policy which is implemented using a pool with different
block allocation policies.

Developing for uClinux can be done using the GNU tools
and other toolchains provided by hardware manufactur-
ers. Despite the large number of already ported applications (http,
ftp and ssh servers and clients, music players, text editors,
shell and shell utilities, among others) the basis for port-
ing existing applications from Linux to uClinux can be ex-
plained having only two replacements in mind: malloc must
be replaced by one of the substitutes present in the uClibc
(malloc2 or malloc simple) and fork must be replaced by a
vfork + exec call. For complex applications, these rules are
sufficient to render a port impossible to implement without
a major application redesign. For example, replacing mal-
loc with one of the two options mentioned can cause a lot
of fragmentation, depending of the memory consumption
pattern displayed by the application. As for the fork and
vfork+exec approach, a special care should be taken with
some shared and initialized variables whose value will not
be propagated to the child process. This is due to the lack
of implementation for the “copy-on-write” process, present
in the vanilla distribution, which makes marking a memory
region for later copy impossible, forcing the copy operation
to take place immediately[15].

2.9 Inferno

Inferno operating system is a direct descendent from “Plan 9
from Bell Labs”, which aimed to be “what Unix should have
been” since its birth in 1969[13]. Both operating systems are
based in common Unix premises which were implemented in
a more coherent way and can now be applied to all use cases,
with no exceptions. Some of these premises are:

- All resources are files and share a global, unique and
  unambiguous namespace
- All local and remote resources are represented in the
  filesystem using an unique and unambiguous names-
  space
- All local and remote resources communicate using the
  same protocol (called Styx on Inferno OS)

InfernoOS can run on its own, as a native operating sys-
tem but also as an application on other platforms, namely;
Linux, Windows, FreeBSD, Mac OS X and others. Appli-
cations are written in a type safe language, called Limbo,
which uses a byte code intermediate representation. All
these applications can then be executed using just in time
compilation in a virtual machine. Despite being a hybrid
kernel, the Inferno kernel always runs in supervisor mode
providing memory protection only to the virtual machine
level, isolating each separate instance from each other.

For this operating system and development kits, resources
are very low, having only 1MB of ram as its lower limit. The
development kit consists of the Acme IDE which provides
several compilers (including one for the C language), unix
commands, a graphical debugger and all Limbo libraries for
network access, GUI usage and security protocols (encoding
and encryption). Being an open source operating system,
all application and kernel source code is provided.

Inferno OS is not very popular in the embedded market, but
it is tested and runs in various Intel StrongARM develop-
ment boards. The author considers the virtual machine ap-
proach and the uniform architecture mentioned above (sin-
gle namespace, all resources as files, single message format)
to be a good candidate to future embedded applications.
Applications could be compiled once and run on any em-
bedded (or non embedded) operating system similar to the
Java ME concept, but with tight integration with the oper-
ating system.

2.10 eCos

eCos stands for embedded, configurable operating system. It
is a real time open source operating system designed for em-
bedded systems or systems that only need one process with
multiple threads to fulfill all their tasks. eCos is targetted for
systems with very limited resources (less than 1MB of RAM)
that are simply too small to run Linux or uClinux. eCos is
also used as an advanced full featured boot loader, capable of
retrieving other Operating Systems’ images using ethernet,
usb or serial connections[10]. Although it is considered a
minimal operating system, eCos is extensively configurable.
Its kernel has real time behaviour, support for multi-level
and nested interrupts, several scheduling policies, processes,
threads, interprocess communication primitives and offers
Posix compatibility[14]. eCos hardware support list is very
extensive and includes several ARM, Hitachi, H8, Motorola
68000, MIPS and PowerPC processors, along with USB and
TCP/IP stack implementations. Development is done us-
ing common open source GNU tools, being very similar to developing on a Linux platform.

### 2.11 Contiki
Contiki is an open source operating system designed for very small embedded devices, with little memory resources and processing capabilities. Examples of these memory-constrained systems are some 8 bit computers from 1980 (namely the the Commodore 64) and some microcontrollers. It has a multitask monolithic kernel, a simple GUI library sufficient for developing graphic windowed applications and an implementation for the TCP/IP protocol. To run Contiki with all these features only 30KB of ram are needed. Contiki’s kernel is event driven such as the task model cite{Dunkels-GronvallVoight}. Preemption support is configured and enabled individually in each process that needs this type of scheduling policy. Its graphical environment can be made available using a VNC virtual display client.

When running, Contiki is divided in two parts: a core and eventually some loaded programs. The core part consists of the kernel, a set of base services and shared libraries. Shared libraries and other common functionalities are implemented as services, which can be updated or replaced individually, resulting in a very flexible structure. One notable characteristic of this operating system is its memory footprint which can be less then 100 bytes.

### 2.12 Overall Comparison of Selected Solutions

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contiki</td>
<td>3</td>
</tr>
<tr>
<td>eCos</td>
<td>3</td>
</tr>
<tr>
<td>Inferno</td>
<td>3</td>
</tr>
<tr>
<td>uClinux</td>
<td>4</td>
</tr>
<tr>
<td>RTLinux</td>
<td>3</td>
</tr>
<tr>
<td>Linux</td>
<td>3</td>
</tr>
<tr>
<td>Symbian</td>
<td>3</td>
</tr>
<tr>
<td>WinCE</td>
<td>3</td>
</tr>
<tr>
<td>VxWorks</td>
<td>3</td>
</tr>
<tr>
<td>QNX</td>
<td>4</td>
</tr>
</tbody>
</table>

![Figure 2: Overall Solution Scores](image)

The choice of the operating system to use in an embedded system has a vital impact in the overall porting effort. One conclusion can be drawn from the analysis done in the previous sections: there is a real need for an open and generic software architecture that can be extended to most, if not all, embedded systems. Besides WinCE and Symbian Os, which were designed having specific hardware architectures in mind (x86 and ARM), all other reviewed solutions share a common goal: to support as many architectures as possible. Regarding this objective, Inferno operating system represents a major step forward, despite its lack of popularity, because it has a truly universal namespace for system resources and all its applications are deployed using an intermediate, machine independent bytecode format. Inferno OS also has a unique and a global format for kernel messages, implemented with the goal of simplifying kernel development and internal implementation[13]. Of all commercial operating systems analyzed, the one that showed better overall score was VxWorks (see figure 2). For the free operating systems analysed, Linux is clearly the best possible solution. However, when porting the application for the XTran-3 board, the choices are reduced for uClinux and eCos for hardware support reasons. In fact, uClinux and eCos are the only operating systems that provide out of the box support for the M5282 processor present in the XTran-3 board. For the open source solutions, a port would be necessary. Since uClinux scored much better than eCos in the evaluation done in this paper, and also due to the instability of the current M5282 code of the eCos port[21], uClinux is, indeed, the better solution for the XTran-3 board. One alternative solution would be to ask for support to one of the providers for the commercial solutions analyzed. However, the use of a commercial solution is discouraged not only for cost reasons, but also because the benefits of choosing a Linux based alternative are not negligible in the context of Tecnic’s embedded system: vendor independence, less time to market, large hardware support and the fact that it is an extensively used open source product with many open source applications available and a large developer and user base.

### 3. PORTING UCLINUX TO THE XTRAN-3 BOARD
The first step in porting uClinux to any embedded system is to enumerate all the components of the hardware to port. The most important components are the ones considered to be vital for the system to boot, even if it will not be fully functional at first (not all drivers loaded or implemented) or even functional at all (no console shell, for example). These vital components are the processor, the RAM memory and the serial port, or other means necessary for console output. The XTran-3[20] board, for example, is based in a Motorola Coldfire processor (5282), four flash memory modules holding 2 MB each, two ram memory modules of 4MB each, one real time clock chip and an I2C EEPROM that controls two I2C chips. Figure 3 shows how this hardware is connected in the XTran-3 board.

![Figure 3: XTran-3 hardware involved in this port](image)

Besides knowing the hardware to port, one must also have all the necessary software and hardware tools in order to successfully configure, deploy and boot an uClinux image file (containing the kernel and programs that will run) in the target system. This will be explained in the following sections.

### 3.1 uClinux configuration and build system
All the XTran-3 hardware information must be entered into the uClinux configuration system. After configured, the
project can be built, resulting in an image file containing the kernel and user programs (see figure 4). The uClinux configuration system is divided in three steps: vendor selection, kernel configuration and user program configuration. The developer starts the configuration process issuing the command “make config” for the text version or “make xconfig” for the tcl/tk version under a X Windows environment. The vendor “Tecmic” was created, along with the “XTran-3” product in order to accomodate all the settings in one place. This creation process is manual and includes editing a couple of makefiles, configuration files and create a directory structure. Upon the start of the configuration process, existing vendor files and folders are processed and appear as options in the product/vendor selection screens.

The 2.6 kernel version was chosen (2.6.19), along with the designated processor, ram size, interrupt vector location and size, supported filesystems, console driver and the kernel initialization parameters. The image file has a structure similar to the one in figure 4 and must be copied to the target board ram. The copy operation was done in different ways during development. Due to the absence of an ethernet chip in the XTran-3 board (although the slot was there, the prototype used in this work did not have it mounted), the serial port was used initially to transfer the image file. A modification to the Tecmic firmware was implemented in order to transfer the image files using the serial port and then jump to the first address of kernel executable code. This proved to be a very slow process, not only because of the record format used (Motorola S3 records) which increased the total amount of bytes to transfer, but also apparently to a hardware limitation in the XTran-3 board, which resulted in data corruption for transfer speeds greater than 19200 bps under hardware and software flow control. After the initial proof of concept builds were working, Tecmic made a BDM port available to this project, which was immediately put to good use, resulting in transfer speed gains from 45 to 3 minutes.

Figure 4: XTran-3 Image memory layout in XTran-3 ram

3.2 Bootloading to uClinux

Although very complex and self contained, the kernel code is unable to boot by itself, requiring several resources to be already initialized by a special component that runs before the kernel: the boot loader. For the XTran-3 board, the Tecmic test application was used as the boot loader. Upon transfer of the image to the board RAM memory, the boot loader was initialized by manually pressing the board reset button. Booting to uClinux is a new option implemented during development of the porting operation that simply changes the processor’s program counter register to the initial address of the kernel executable code. In the XTran-3 architecture, this is enough for the Linux initialization output to start appearing in the output console. Besides proper console initialization in the uClinux kernel which, when enabled by appending the correct boot parameters, allowed to see what was happening during the boot phase, one of the initial problems faced and solved was a compulsive hardware reset triggered by the board’s watchdog. This happened because one of the kernel’s first task is to rewrite every interrupt vector addresses to be able to handle all system interrupts and this included the watchdog kicking routine. In the absence of watchdog kicks, a software reset was immediately triggered by the watchdog and caused the processor to return to the boot loader execution. To solve this problem, the watchdog initialization code was changed in Tecmic’s boot loader in order to disable it, causing the Linux kernel to boot, say its motd and present the long awaited console to the user. Getting the kernel to successfully boot in the Xtran-3 board was a very difficult task and took half of the porting effort total time. The difficulty resides in the absence of console output (until proper initialization of the console) and the configuration of kernel options (scheduler, addresses for code location and ram disk size) in order to get a minimal, but functional system.

3.3 Real Time Clock driver

The first device ported to uClinux was the real time clock chip present in the XTran-3 board. This device is a Dallas ds2404 which is currently not supported by the Linux kernel. The closest match to what is already implemented for that vendor in the Linux kernel, and probably would actually work, is the Dallas 1 wire bus driver. However, since the current Tecmic design is connected to the processor using the 3-wire pins, the current implementation does not fit the porting purposes. Since this device serves a specific purpose which is very frequent in embedded and non-embedded systems, there is a specific class for real time clocks in the Linux kernel. For proper development of this type of devices, one must implement a specific API for accessing the device and the kernel does the rest from there on.

The DS2404 device exposes its functionality using several memory pages. In these pages, among other information, the current Unix time is stored. For reading and writing a determined amount of bytes from one of the available pages, one must use the scratch pad, which acts as intermediate buffer for each byte to read or write (total capacity of the scratch buffer is 256 bytes). There are 16 pages of 256 non-volatile bits each. The current alarm and time values can be found in the address 0x0200h[4]. The device only supports three different operations which, when called using different arguments, provide full control of the device. These operations are:

- Copy the contents of an address range from a page to the scratch buffer
- Copy the contents of the scratch buffer to a page
- Read the scratch buffer

Reading the current time, for example, involves initializing the device with the address being read, then ask for a copy operation from the contents of the previously given address to the scratch buffer and then read the scratch buffer[4]. Data is sent bit by bit using the GPIO interface of the Motorola Codfire 5282 processor.

Thanks to the RTC class, it was possible to initialize this device as the default device for time retrieval, which means the kernel, upon initialization, will initialize its internal clock (that does not need any device and uses the processor’s prescaler for counting time) with the current time read from the ds2404 device. Also, using the proc API, a character node was created in the /proc filesystem to show the device status:
Upon loading, the new ds2404 module generates a character device node called /dev/rtc0 which can be accessed by any application that was designed to use the Linux kernel rtc API. Examples of these programs are hwclock and date, used to test this module’s functionalities.

3.4 Flash Mapping driver

The XTran-3 board has four NOR flash modules of 2 MB each, directly available through its addressing space. Support for flash memory devices has a special implementation in the Linux kernel[30], as demonstrated in figure 5.

Figure 5: XTran-3 Memory Technology Devices implementation in Linux

Starting from the lowest layer, closer to the hardware, there are two command sets implemented in the Linux kernel for communicating with the flash memory hardware: CFI and Jedec. CFI is desirable because it is supposed to be an evolution of the older Jedec and defines a common method for reading each chip’s topology and other manufacturing data. All four flash chips are siblings from the same model: M29W160DB from ST. After several attempts at using CFI to correctly auto detect each chip, the only way to make the kernel recognize this device was to patch the broken jedec_probe.c file in the kernel code. The final working configuration of their internal FTL algorithms.

However, there are other better options for implementing this layer. One of those options is the use of a filesystem that was designed to work with flash devices[30]. For this work, JFFS2 was chosen. JFFS2 stands for Journalling Flash File System and is a log-structured file system for use with NOR or NAND flash memory devices. Besides the wear leveling capabilities, two of the most notable characteristics of the JFFS2 filesystem are the use of compression (using zlib, bzip2 or rtime algorithms) and a garbage collection algorithm to reduce overall fragmentation[30]. The JFFS2 filesystem is available as an option in the Linux kernel, and was successfully used in the XTran-3 board, providing seamless file access to the four flash chips and enabling the use of directories, files, soft and hard links and access to the device as if it was a normal block device. This is a great improvement over the previous Tecnic software implementation, which used the chip blocks directly to read and write chunks of

With the mapping modules loaded, each device is accessible for reading and writing using two filesystem nodes, a character device, for passing specific flash commands, and a block device for transferring binary data. However, this kind of access alone is not desirable because it is not possible to create files using the filesystem utilities. Currently, there are two different layers to link the mapping layer to the normal filesystem layer. The first one is with the FTL module (flash translation layer). This module creates an additional block node for each pair of device nodes created by the mapping layer. With the new block node, one can use the normal filesystem utilities (mkdosfs, mkext2fs, touch, cp, mkdir, etc) to create and work with the newly created filesystem.

The FTL implementation in Linux is very simple, but does what it is supposed to: it maps and handles flash blocks as if they were normal byte blocks. However, this approach is a naive one because it does not optimize the use and weariness of each block. Flash memory has a limited number of writes before each block gets worn out, so a crucial factor for extending the lifetime of this type of memory is to distribute the write operations across as many different blocks as possible, without loosing filesystem integrity. It is possible, using FTL, to mount a FAT filesystem in a flash chip, but the blocks will wear out quickly because FAT, by design, uses a specific region of the physical device to store the file allocation table. In the uClinux version used in this work, the FTL module compiled, but does a kernel panic upon loading. All known good configurations for this module use the previous major kernel version (2.4) and even with some debugging and questions posed to the uClinux mailing list, no fix for this problem was found.

This type of mapping is what the typical USB pen drives do by hardware. The quality and durability of each device, among other factors, is dependent of the hardware implementation of their internal FTL algorithms.
3.5 LED driver

The XTran-3 board has a set of 13 general purpose LEDs. One of them is directly connected to the processor, the other 12 are divided into two groups, one with 8 and the other with 4 LEDs. Each group is connected to an I2C controller, model PCF8574SO16, but these controllers are not manipulated directly by the processor. Instead, Tecmic simplified the task of having to implement a driver in its custom software and added a master I2C controller EEPROM. The master controller is directly connected to the processor and the other two controllers are connected to the master (see figure 3). To control each LED, one must write an address/value pair in the EEPROM address and, upon entering the execution command, the data is sent across the I2C bus, causing the LED to update its state. The code to write to the EEPROM was heavily inspired in Tecmic’s implementation, and it was integrated in the Linux kernel using a standard character device[12]. The data is entered in the form LED_number=state_flag. For instance, 3=1 means: “lit LED 3”. Changing several states at once is possible using the : character to separate several commands: “3=1:4=0:5=1” means turn on LED 3, off LED 4, lit up LED 5. The device driver copies the data from userspace and sends commands to the EEPROM. Like in the ds2404 driver, several applications can read or write to the file at once because all the critical regions are protected by a spinlock. An example usage of the implemented driver can be done using only the following shell command:

```
# echo "3=1:4=1:5=1:10=0" > /dev/xtranleds
```

The critical region, protected by the spinlock, consists in changing the state of a single LED because once the write protocol begins, it cannot be interrupted, otherwise, the device would eventually enter an incoherent state and become unusable.

3.6 ds2404help application

Although date and hwclock are sufficient to control the rtc clock, there is an additional operation that was not supported by any of the two programs. This operation sets the current alarm date and time in the RTC clock. In the typical PC, it is only used by the motherboard chipset driver for triggering wake on alarm events. For setting the clock and alarm time and date, an application was written: the ds2404help. Despite its name, it works with any rtc driver because it uses the ioctl interface, whose entry points are implemented by the rtc driver, to ask for specific rtc data. The implemented syntax allows one to set the alarm and/or the time with one simple command:

```
# ds2404help -a 02/07/1981:21:48:00 -d 02/07/1981:21:48:00
```

3.7 ledshow application

The XTran LED driver implemented in this port allows direct manipulation of each LED state, but changing several LEDs at the same time can be bothersome and requires the programmer to know which LED number is assigned to which position in the XTran-3 board panel. To simplify the process of bulk state writes to all LEDs, the ledshow application was written. It receives a filename passed as argument and processes it, changing the LED state according to the information in the file. The file is a simple text file and contains a sequence of numbers and LED states whose spatial disposition in the text file is the same as in the LED panel. An example file would be:

```
10
000000
000000
40
111111
111111
20
```

The first number is the number of times the sequence will repeat. The next two rows provide a spatial representation of all the 12 LED states, 1 means on and 0 means off. The number next to the two rows indicates the previous state duration in milliseconds. In this case, the LEDs will remain completely off for 40 milliseconds. Next in the file one can have as many states and durations as desired. In the given example, all LEDs are turned off and then lit for 40 and 10 milliseconds respectively. This sequence is repeated 10 times as specified by the first number in the beginning of the file.

3.8 Updating the configuration scripts

Besides all the C code files containing the implementation for each driver, several updates to existing configuration files (Kconfig files) had to be done so that the new drivers could appear in the configuration screens and be selectable when the M5282 architecture was chosen. Also, several Makefiles had to be changed to enable inclusion of the new kernel object files into the image file.

Userspace programs have a similar method for inclusion in the configuration screens. There is one Makefile for each program and, by defining actions for well known targets, the compilation script can invoke those pre defined targets and generate the executables for inclusion in the romfs image.

4. EVALUATION

In order to accurately access the system’s performance, a rigorous program-based evaluation had to be carried out. Such procedure helped perceiving how the ported system compares to other embedded systems based in uClinux (with similar hardware) and to the proprietary in-house solution from Tecnic.

4.1 Experimental Setup

After all implementations described in the previous section, the ported system was put to test. The test environment consisted in an uClinux image containing all the programs used during development, plus the test tools. All the tools used or implemented have the capability of showing the test results in the output console immediately before the end of its execution, allowing the data to be extracted for further analysis. All the kernel modules developed were loaded in the kernel, but no processes were running in the system besides each test executable file.

4.2 lmbench Tool

The lmbench tool is a widely used test tool for Linux systems[23], providing an extensive amount of programs to test certain aspects of the operating system. By using the lmbench tool, one has a common base for comparison with
results obtained in other architectures and execution environments. The Lmbench suite provides many benchmark results on popular desktop and server environments.

Lmbench version 3 was used in the tests, but since the vanilla version of this tool only compiles under libc, a ported version for uClibc was checked out from the Blackfin Linux project repository and incorporated as an additional program for the customized uClinux distribution produced in this work. Three benchmarks were chosen to run for their significance: lat_proc, lat_systcall and lat_ctx, however, lat_ctx is the most important test because it is widely used for measuring the uClinux performance on several embedded systems.

lat_systcall Measures the amount of time necessary to write a byte in the special node /dev/null. Although it is a very simple test, the result value presents the lower limit of the amount of time necessary to interact with the kernel from user space. The results can be useful for comparing the ported system with other versions of uClinux.

lat_proc Measures the amount of time necessary to create and start a new process. This test was executed in two different modes: the vfork + exit and vfork + execve. Vfork + exit measures the amount of time necessary to create a child process similar to the parent process. Vfork + execve measures the amount of time necessary to create a child process similar to the parent process and start the execution of the child process.

lat_ctx Measures the amount of time necessary for the kernel to make a context switch between a variable number of processes. During the execution of these tests, several processes are created with two pipes, each connecting the previous one to the next, making a ring. Each process reads data from the entry pipe, performs a variable amount of calculations (sum the values in the array) and sends the result to the next pipe. These tests are compared with the result obtained by Tecmic using its own boards, the S3C24A0. This Samsung board contains an ARM processor, model 926ej running at 200MHz. The kernel version used by Samsung is older than the one used in this work, hence, due to improvements constantly being implemented in the Linux and uClinux kernels, some variations are to be expected.

4.3 Custom Console Test Tool

Lmbench runs in every Linux environment, but does not run in Tecmic's proprietary software solution deployed in the XTran-3. To compare the old system and the ported system, a simple test program was written. It measures the time necessary to transfer a specified amount of characters from the console to the kernel and back to the console. Tecmic's firmware was once again changed to perform this task and present the obtained results using the console.

As for the ported system, an userspace program was implemented, using the uClibc API, using simple printf and scanf functions[3]. The pseudocode for this test is the same for both implementations:

```c
begin
  waitForFirstCharacter();
  counterStart = getTimeOfDay();
  do{
    char = receiveChar();
    print(char);
  } while (char != CHAR_END);
  counterEnd = getTimeOfDay();
  print("Time spent: %d", counterEnd - counterStart);
end
```

Note that it is expected that the ported system will have far worse results than Tecmic's firmware. This is because Tecmic's firmware is a real time system and this code will run in exclusive mode (there is no task scheduling and/or context switching). In Linux, the program will be run from user space and use the uClibc functions to access the console, which will in turn do a predefined system call that will trigger the console driver to place the results in a buffer.

4.4 Evaluation Results

Figure 6 shows that the uClinux 2.6.19 scheduler does not get affected by a large number of processes in the system. The context switching time remains steady, which can be explained by the absence of the MMU and the improvements implemented in the kernel scheduler since version 2.6.19. Note that for each buffer size, which varies from 0, 1 or 16 KB, there is an observed time interval, corresponding to the amount of time necessary to calculate the array values' sum. This factor is the “ovr” value showed by the lat_ctx output.

```
<table>
<thead>
<tr>
<th>Number of Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 KB, 209.96 us</td>
</tr>
<tr>
<td>1 KB, 258.70 us</td>
</tr>
<tr>
<td>16 KB, 910.17 us</td>
</tr>
</tbody>
</table>
```

Figure 6: Lmbench lat_ctx results in the XTran-3 board (kernel version 2.6.19-uc1)

In figure 7, the comparison results for a specific lat_ctx test are shown. The test was performed setting the array size to 16KB and making variations to the number of processes. The ported system is 2.3 times faster at context switching, then the 2.6.11 version of uClinux running in a faster processor. This is also explained by the differences in the scheduler and the architecture used. Note the context switching time in uClinux is always faster than Linux even under similar hardware, however, there is a large “ovr” factor difference that, despite not being shown in the results, is directly observable in the system, making the test actually run slower in the ported system versus the Samsung board. This is due to the fact that the ARM processor has a higher clock frequency (200 MHz vs 66.6 MHz) and is generally much faster doing ALU calculations[22] than the Motorola ColdFire 5282.

Figure 8 shows the console test results. Surprisingly, the ported system is faster than Tecnic implementation, at least for serial port transfers. Since the code being tested here consists of the serial driver implementations of Tecnic and uClinux, the test shows that the Linux kernel implementation of the MCF serial driver has better performance than Tecnic's code.

5. CONCLUSIONS AND FUTURE WORK

In this paper, a successful port of uClinux to a proprietary embedded system is presented. Also, a qualitative
analysis to several free and commercial embedded operating systems is done. From all solutions reviewed, uClinux proved to be the best choice for the XTran-3 hardware, a custom embedded system designed by Tecmic and used in this work. Choosing the right Linux solution and kernel depends mainly on system requirements and hardware limitations: real time solutions require RTLinux, no-MMU architectures require uClinux and Linux can be used on all others. One can conclude that in order to gain a realistic perception of each solution’s strong and weak points, an experimental approach is necessary. This practical approach can go from the simple installation of each solution on a development machine, to real deployment attempts on the target embedded system.

Developing for uClinux requires the right tools for building each architecture’s specific code and the environment setup is not tightly integrated or even user friendly, contrary to what happens with all the commercial solutions analysed. uClinux build system is clearly one of the distribution’s weak points since several fixes to the configuration scripts and kernel files had to be implemented in order for the system image to be built correctly. Deploying and running the uClinux image required additional effort because porting uClinux to the XTran-3 board also implied a port of the XTran-3 firmware to run uClinux. This apparently “backwards” task had to be carried out because the uClinux kernel is not capable of booting by itself and needs a bootloader to initialize part of the hardware. In this work, Tecmic’s firmware was modified to serve uClinux’s boot loader needs. After proper setup and deployment of the uClinux distribution, the device drivers and application implementations took place. After the development was concluded, several tests were also performed, including benchmarks with a common Linux benchmarking tool: lmbench3. By planning and defining these porting tasks for each project, a “porting road map to Linux” can be obtained and its accomplishment is considered vital to achieve a functional and mature Linux port for an embedded system.

The proposed solution implemented the drivers for the RTC clock, the mapping driver for the four flash chips and the LED driver. Two applications were also developed to experiment with the new drivers, but both can work with other hardware because the generic kernel API was used. With the uClinux port, the system now provides all Linux benefits: user space programming using the libc, task scheduler, processes, synchronization functionalities, proper filesystem support and a multitude of advancements which were impossible in the original system, or required a substantial amount of investment to be implemented. Despite the successful port in functional terms, several measurements of the system performance were taken using a widely used tool for benchmarking: lmbench3. The preliminary results were promising. The new system performs better at context switching than an ARM based solution using an older 2.6 kernel, even with the differences in clock speed which clearly gave the ARM an advantage. Other reason for this performance improvement is the advancements that get implemented in the Linux kernel in each release. As for the performance comparison from the old versus the ported system, the ported system showed gains when performing a data transfer using the serial port. Although this benchmark was developed using a similar algorithm for both systems, since the uClinux implementation run in user space and used libc, a stronger negative performance impact was expected. This did not happen in the benchmarks run, probably due to the more elaborate implementation of the Coldfire serial port driver implementation in uClinux, compared to Tecnic’s custom implementation. In conclusion, uClinux was ported successfully to run in the XTran-3 board without affecting the system performance, in fact, it actually improved it under certain situations (serial port data transfer).

As to future work, the author concludes that porting the remaining board hardware and applications would give Tecnic a colossal advancement over the previous system. Thanks to the use of libc, future application compatibility is assured. There is now a multitude of applications already included in uClinux, ready to provide advanced network and even multimedia capabilities. Also, much more benefits could be obtained from a Linux port if the target processor had an MMU. The existing ported applications would work out of the box, and, depending on the processor choice, minor or no modifications to the new drivers would be necessary. The author suggests the migration to a processor with a MMU from Motorola. A good example is the Motorola 5745 because it also operates in no-MMU mode, maximizing compatibility with the current uClinux port. The new port would also require a heavy amount of business logic regression tests in order to assure full compatibility with the entire XTran architecture.

6. REFERENCES