

Case Studies in Linux Ports to Embedded Platforms

Claudio Matsuoka, Gustavo Boiko, Thiago Galesi

Mandriva Embedded Systems Lab
Curitiba, Brazil

***Abstract.** This paper details the technical issues found in late 2005 when porting Linux to two different embedded platforms, namely an XScale-based rugged handheld device for industrial and military applications (previously running Windows CE) and a PowerPC-based board used in a private branch exchange (PBX) system. Both ports are based on kernel 2.6, and the latter makes use of real-time capabilities provided by RTAI.*

1. Linux on embedded systems

Electronic devices, gadgets, handhelds, media systems, appliances, industrial controllers, communication systems, mobile phones and the like are all targets in one of the fastest growing areas for Linux usage expansion. Powering computers from large NUMA-based systems to small PDAs, sometimes hidden in everyday use equipment such as set-top boxes or digital camcorders, the flexibility, features and mindshare offered by Linux is not going unnoticed by industrial and consumer electronics manufacturers.

From the runtime environment standpoint, Linux on embedded systems is quite different from workstation or server-oriented distributions everybody is used to see. Scarce CPU and storage resources make embedded Linux systems, more often than not, just Linux and not GNU/Linux. Typical systems boot off a few megabytes of flash memory and run on small, cheap and power-efficient SoCs designed around an ARM, MIPS or PowerPC core.

Unlike mainstream IA32 or x86_64, one will find that support for many architectures and sub-architectures typically used in embedded systems are currently under heavy development. Also unlike general-purpose PC-class machines, most embedded devices are unique in their design, making port to each device a non-trivial task. Each ARM-based PDA, for instance, has different peculiarities and that reflects in different drivers, addresses and strategies adopted in the process of making Linux run in these devices.

2. The XScale port

2.1. Solution description

In late 2004 the authors were approached by an European company specialized in industrial-class solutions to have Linux and a user interface running on a rugged handheld device for industrial and military applications. The device is IP64 and MIL-STD-810F compliant and built around an ARM-based XScale PXA263 SoC. Subsystems of interest include a cardbus controller, 802.11b/g networking using a CompactFlash card, MMC/SD support, touch-sensitive display and Bluetooth. The ARM architecture [Seal 2000] is widely used in this class of device due to low power consumption, high code density and large availability of system-on-chip designs [Furber 2000].

Linux was chosen as an alternative to Windows CE to allow greater flexibility and customization to specific needs. Current shipments of this product use plain WinCE 4.20.

2.2. Reverse engineering

The port project was commissioned by a systems integrator — not the device manufacturer. Documentation on parts and components are reasonably complete and were supplied as published by their respective manufacturers. Documentation on the device architecture and hardware details, however, was not available, resulting on a non-neglectable amount of time spent on guessing and reverse engineering. The latter was accomplished using tools such as HaRET (described in Section 2.3) and examining portions of device drivers and other source code supplied by the integrator.

Without a clear description of the hardware details, we decided to port and stabilize the Linux kernel before installing the final bootloader, learning as much as possible about the device in the process. Keeping WinCE on the device during the development also allowed us to compare support to different subsystems as development takes place. Again, HaRET was used to boot Linux kernels from SD or CF cards, allowing us to develop Linux while keeping WinCE installed on the same device.

2.3. HaRET

HaRET (a short for *Handheld Reverse Engineering Tool*) is an Open Source utility written by Andrew Zabolotny. HaRET is targeted at ARM-based systems running Windows CE, and, among its many capabilities, can:

- detect and provide general hardware information
- read and write data to/from virtual and physical addresses
- access general-purpose I/O registers (GPIOs)
- execute user-defined scripts
- load and boot the Linux kernel

The ability to load, prepare and execute Linux from within Windows CE makes HaRET an invaluable tool to test experimental kernels. This is especially important when the bootloader already installed on the device (usually Eboot) is not able to boot from the network or from a serial port, not enough information about the system architecture is available to port a bootloader such as U-Boot, or when the system doesn't have any serial or ethernet ports.

2.4. The kernel bundle

The process of booting a Linux kernel from within an operating system with virtual memory and memory protection involves a series of operations that must be well-understood in order to trace the first stages of Linux execution. The concept, although simple, has a non-trivial implementation that depends heavily on the system architecture and the host OS. To see how this works, let's start with a description of how the kernel is actually executed.

Linux boots by execution of its startup entry point at `<arch>/kernel/head.S` — assuming that the kernel is uncompressed and properly located in memory. The entire process, however, starts earlier when the bootloader prepares the execution environment, loads the compressed kernel and initial RAM disk images into RAM and transfer execution to the kernel decompressor¹. ARM

¹Execution-in-place (XIP) systems act differently and will not be detailed in this paper.

Linux bootloaders makes use of a binary object called the “kernel bundle”, comprised of a series of parameter tags, the compressed kernel image and the compressed initial RAM disk, aligned in 4KB boundaries.

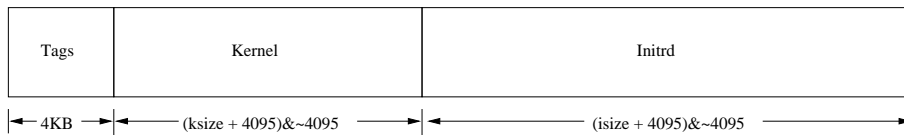


Figure 1. The kernel bundle

Tags are a mechanism used to pass parameters to the kernel, in a sequence of structures starting with 32-bit fields containing the size and type of each tag. Parameters that can be passed through tags include the address of the initial RAM disk, the kernel command line and the board serial number. The tag list must start with ATAG_CORE and end with ATAG_NONE. Table 1 shows the tag list built by HaRET when building its kernel bundle.

Table 1. Example of tags in a kernel bundle

size	00000005
tag	ATAG_CORE
flags	00000000
pagesize	00001000
rootdev	00000000
size	xxxxxxxx
tag	ATAG_CMDLINE
cmdline	root=/dev/ram rw ...
size	00000004
tag	ATAG_MEM
start	a0000000
size	04000000
size	00000000
tag	ATAG_NONE

2.5. Memory management issues

With the kernel bundle loaded in RAM, we cannot simply execute it from within Windows CE. First, WinCE has other tasks being executed, and then it is improperly located in the system memory. The bundle cannot be transferred by HaRET to its final destination, (address 0xa0008000, start of physical memory)² without overwriting WinCE’s MMU tables. A different strategy must be adopted.

To overcome the memory management problems, HaRET allocates a block of physically contiguous memory in high RAM. This memory is used to hold code — the preloader — that relocates the kernel bundle according to a private “page map” table, and the table itself. WinCE switches to single-task mode and control is transferred to the preloader, which takes care of moving the kernel and initrd to their proper locations at the start of memory. This layout is shown in Figure 2.

²In the XScale case, other ARM-based systems may use different addresses.

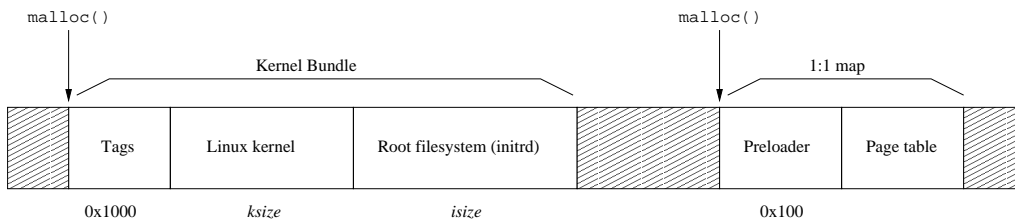


Figure 2. Memory map before running the preloader

An additional memory management trick must be performed in order to disable the memory management unit *while the code is still running*. To perform this critical step, HaRET instructs the MMU to create a 1:1 mapping for the preloader area in the first-level descriptor table, thus making virtual and physical addressing equal and allowing code to continue execution when the MMU is switched off. The preloader then copies the kernel bundle to the start of physical memory, overwriting WinCE.

2.6. Kernel decompression

After executing the preloader, the kernel bundle is in place according to Figure 3 and execution is passed to the kernel decompressor at `arch/arm/boot/compressed/head.S`. A magic number at the start of the code can be checked before jumping to verify if the preloader succeeded in placing the compressed kernel in the correct location.

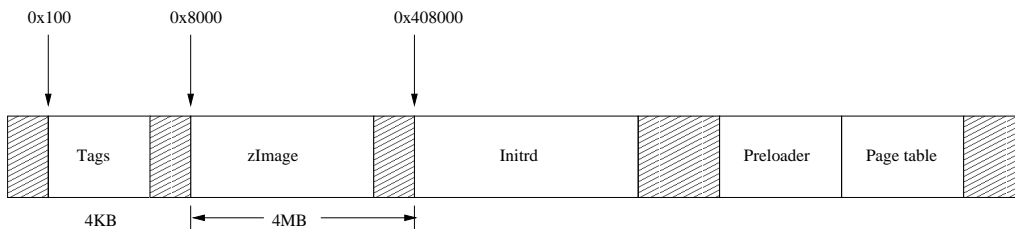


Figure 3. Memory map immediately after running the preloader

After the initial setup done in assembly, actual kernel decompression is handled by `decompress_kernel()` in `arch/arm/boot/compressed/misc.c`, which — if your system has an appropriate console device — prints the well-known “Uncompressing Linux...” message. Only after decompression control is finally transferred to the Linux kernel.

2.7. Boot process debugging

It is worth of note that extensive debugging was necessary in the boot process *before* the initialization of the framebuffer device, and there was no serial port available for a serial console. All tracing was made through a status LED and raw access to the display controller, which was converted into a simple scrollable terminal. Special care was necessary in using the appropriate address translations accessing the display memory before and after the MMU was re-enabled by Linux.

A tracing mechanism is clearly needed — when a boot attempt fails one cannot tell if the kernel is not being loaded, relocated, decompressed, locked up in a driver initialization or if it is only the framebuffer device that is not working correctly. This port had

problems at least in loading, relocation, UART driver initialization causing a system lock, and framebuffer initialization.

2.8. Device drivers

Availability of device drivers of interest prompted the use of the latest 2.6 series kernel at the time of the port. While the port effort started with Linux 2.6.12, it was soon updated to 2.6.14.2 to take advantage of new code introduced in the later kernel.

The initial port effort included the development of the following device drivers:

- Board support
- Improvements in the Epson S1D13806 framebuffer
- Wolfson WC97 touch panel support (through AC97)
- Keypad support via AT keyboard emulation
- PCMCIA controller and CF slot support

The framebuffer driver was later improved to implement rotation to portrait mode and 2D acceleration.

Controllers for many important subsystems, such as UARTs, Bluetooth and I2C are integrated in the SoC itself, and are well-supported by drivers already present in the Linux kernel. Drivers for other devices are still not implemented, such as the Philips ISP1161 USB HCD (support exists in Linux 2.6.14 but it is not functional), AC97-based PCM audio and different sensors.

2.9. Runtime environment and applications

The user-level runtime environment for Linux on the device was built around uClibc and Busybox, with a minimal Bourne shell and configuration tools for Bluetooth, Wi-Fi and PCMCIA sockets. The framebuffer console is used mostly for debugging purposes, and there are no plans to include it in the final product. Remote access during development is provided by telnet over the wireless link, as Bluetooth bandwidth is too limited even for character-based interactive sessions.

The root filesystem is read-only and resides in an Ext2 image loaded as an initial RAM disk. Due to HaRET constraints, the RAM disk size is limited to around 5MB compressed, or around 10MB uncompressed depending on the filesystem contents. This same strategy can be used for the final system when booting from flash, with the rest of the NAND flash memory accessible as a memory technology device (MTD) using JFFS2.

During development, large applications reside on either SD or CF cards mounted at boot time. Test applications included QT/Embedded based GUIs (QTopia and Opie), SDL games and stress test and diagnostic programs. The most notable absence is a modern, freely available web browser for QTopia or Opie. At the time the port was made, only a very old version of Konqueror embedded was available. Commercial alternatives such as Opera are considered for production.

2.10. Sending patches upstream

Patches written for this project will be made available to upstream maintainers using a Git repository after minimum stability criteria are met in tests conducted by the integrator and third-party developers. Further development is scheduled to happen in the second phase of the project in early 2006.

3. The PowerQUICC-based system

3.1. System overview

The implementation of Linux for a PowerQUICC based PBX board and the port of a time-critical application previously running on “bare metal” (i.e. without an underlying OS) represents a complete departure from the scenario built in the XScale device port. In this case, Linux enters as a provider of networking capabilities with ethernet drivers and a robust TCP/IP stack.

The PowerQUICC communications processors are a series of system-on-chip designs built by Freescale Semiconductor around a PowerPC core targeted at networking and communications applications, containing, among other subsystems, baud rate generators, a communications processor module (CPM), PCMCIA controller, as well as support for several other common interfaces. The system also includes components relating to telco infrastructure interfacing and a dial-up serial modem for remote management.

Architectural details of the hardware cannot be disclosed at this time, per request of the device manufacturer. Further information will be made available when the equipment reaches production status.

3.2. Licensing issues

Due to the nature of the application, portions of the code responsible for direct hardware access were moved to kernel drivers in the port process. Drivers covered by a proprietary license were built as modules and not statically linked to the kernel to prevent licensing problems.

3.3. Existing application and board support

The original application did not run under a OS. Several facilities such as real-time management, interrupt handling and task scheduling were done by the application. No memory protection scheme was implemented; the system used 1:1 memory mapping. As such, hardware access were done directly by the application.

Linux support for the platform already existed at the time the port was initiated. However, it was very out-of-date and mostly nonfunctional. With this port, maintenance was resumed, allowing the developers to work closely to the new maintainer and streamline patches directly.

3.4. Real-time systems

Due to the time sensitive nature of the protocols concerning this board the system needs to have time warranties for certain operations. Some common situations require the application to act in timeframes of tens of milliseconds. In addition, the application’s pre-existing real-time handling structures required itself to be called periodically every 10ms.

In this case, the RTAI/Fusion system was used to provide real-time capabilities to the system. RTAI/Fusion is composed by a series of patches to the Linux kernel, as well as kernel modules and userspace libraries. It adds hard real-time scheduling to userspace applications using a simple and flexible API, including API “skins” to ease the transition from other RTOS to Linux.

One of its most important features allows for real-time code to be executed within user-level context. Because of licensing reasons and the application's complexity, it would not be possible to put all time sensitive code in kernel context — this feature alone was decisive for the project viability.

RTAI has proven itself during the project. No considerable RTAI overhead has been detected during development (considering the 10ms periodic task calling). In one of the drivers needed for this project, (one of the trickiest project requirements) ,using RTAI interrupt handling system has allowed interrupts to be handled in hundreds of microseconds (as opposed to tens of milliseconds using conventional Linux interrupt handling systems).

3.5. Device drivers

The use of RTAI in this project has posed an important problem concerning device drivers: it is not possible to make system calls during execution of time critical code in userspace. Any code that makes a syscall is subject to rescheduling (hence, taking RTAI's control from the thread and removing any real time warranties). RTAI handles this by either blocking (that is, signaling the application) whenever a syscall is made, or by switching to a soft real-time mode. None of those options were acceptable for this project. Therefore, communication between the driver and the application was done via shared memory regions, organized as lockless message rings. Messages from the application to the driver are written to the ring, which is checked periodically by a periodic timer in the driver. Messages to the application are also checked by the application's 10ms timer.

3.6. Memory technology devices

The system uses flash memory for system/application, bootloader and configuration. The original system did not use any file system; "files" were assigned to different flash blocks, and data was read directly from flash, since it is directly mapped into the address space. MTD partitions are used to create separate areas for the bootloader (U-Boot), system (which includes the kernel and root filesystem) and application data.

3.7. Runtime environment

The same approach used in the ARM PDA was used in the PowerPC board: the root filesystem resides on a compressed, read-only Ext2 system loaded as a initial RAM disk. Configuration data is stored in a special MTD partition. During the initial development, configuration data was being accessed by means of mapping configuration files in memory (since for development purposes the root filesystem for this board was being mounted via NFS). However, this approach does not work for flash filesystems such as JFFS2. The initial solution adopted was to use a Minix partition mounted over a MTD block. Several problems were encountered with this method, mainly that the system required a complete shutdown sequence to properly store data without corruption.

Later, the usage of memory mapped access to files was replaced by conventional accesses (read and write) so the configuration filesystem could be switched to JFFS2. This approach has payed off: JFFS2 has demonstrated in tests its extreme resilience against data corruption (even on extreme cases) as well as its compression capabilities. The only drawback noted is that JFFS2 (as a fully flash aware file system) reserves some flash blocks for garbage collecting purposes. This poses a problem when only a few blocks are available (even though compression will most likely make up for the lost space).

4. Conclusions

The two different ports presented in this work illustrate the uniqueness of Linux implementations in embedded platforms. Unless the targets are very similar in architecture, physical interface and application, each port will require different patches, subsystems and solutions custom-tailored for that specific device. Unlike desktop PCs, embedded platforms cannot have an “one size fits all” binary distribution for general usage.

The current status of ARM support in the Linux kernel make it relatively easy to port the system to a new ARM-based machine, especially PDAs. Previous work on a system based on the Samsung S3C2440 SoC has shown that basic board support is simple to implement, and it is possible to have the new system booting from HaRET in a few days of work. XScale PXA255 and PXA263 devices are usually similar to the Lubbock evaluation system from Intel, making the existing code a good place to start.

Experience has shown us that a combination of a revision control system such as Subversion [Collins-Sussman et al. 2004] and a patch management system such as Quilt [Grünbacher 2005] or Git could be the best approach to allow agile and organized development and easy submission of patches to upstream maintainers. Git is especially well-suited for parallel development and kernel patch management: it is adopted by upstream maintainers, deals with multiple branches and allow easy patchset sharing among developers.

The approach used by HaRET to boot Linux from WindowsCE on ARM systems makes it an essential tool when porting the kernel to an ARM-based systems without serial or ethernet ports and already run WinCE. This could be extended to other architectures supported by WinCE such as SH3/SH4, i386 and MIPS. It also becomes important when a small amount of reverse engineering is required, which is almost always the case when you're not in direct contact with the engineers who designed the unit you're working on.

Test strategies and existing drivers against the real hardware when planning the port is essential to prevent frustrations caused by existing but nonfunctional code, limited support to features required by the project or last-minute infrastructure changes. Additionally, real-time applications ported from bare metal to an RTOS are generally very hard to trace and debug, especially under high CPU usage.

Applications targeted at embedded systems with raw flash storage, it is important to keep in mind that MTD and JFFS2, due to peculiarities in the way flash memory is accessed, don't support writable mmap access. Applications should use normal read and write access instead in order to be able to use JFFS2. Regular file systems created on writable mtdblock devices result in lack of reliability and may lead to early wearing of certain regions of memory. MTD targeted filesystems should be used whenever possible.

References

- Collins-Sussman, B., Fitzpatrick, B. W., and Pilato, C. M. (2004). *Version Control with Subversion*. O'Reilly, 1st edition.
- Furber, S. (2000). *ARM System-on-Chip Architecture*. Addison-Wesley, 2nd edition.
- Grünbacher, A. (2005). *How to Survive with Many Patches*. SuSE Labs.
- Seal, D. (2000). *ARM Architecture Reference Manual*. Addison-Wesley, 2nd edition.