Using Linux for industrial projects – A return of experience

Christian Charreyre

Using Linux for industrial projects – A return of experience

Christian Charreyre

CIO Informatique Industrielle, BP 710 - 1 Rue de la Presse – 42950 St Etienne Cedex 9 - France

Abstract: This article describes industrial projects involving embedded and/or real time Linux. After an introduction dedicated to place Linux regarding the embedded software market, it explains the adoption process of Linux in our company, that was working will legacy RTOS since years. Then it emphasizes 3 aspects of using Linux in industrial projects : using Linux in an embedded device, using Linux in a real time context and porting an application from legacy RTOS to Linux. The article ends up with the problem of management in an industrial context of a software base that moves very quickly, and explains what philosophy to adopt regarding evolutions control.

Keywords: Linux, embedded, real time, return of experience, porting from RTOS to Linux

1. Introduction

Regarding Operating Systems, the embedded software market has been for years divided into a lot of proprietary solutions offered by many specialized companies. The economic model of such offers is based on royalties paid by customers for each device including the O.S., plus development tools (integrated development environment, performance analysis tool etc….) that generally represent important costs, mainly for little companies.

Since a few years, Linux is a new and fast growing actor on this market. In March 2007, the French magazine 01 Informatique[1] stated in his article about RTS Embedded Systems 07 that Linux was now about 50% of the Embedded OS market.

The annual survey of the Web site linuxdevices.com[2] presents the same growing tendencies, and it appears that the couple of ARM processor with Linux OS is a great solution for low price mass market devices.

Our company is specialized since 1990 in software development for embedded and real time applications, and we have used the various legacy real time OS during years. At the beginning of 2000, Linux began to appear in the professional press, not as an embedded solution, but as a great OS for servers.

As the management of the company has worked with Unix solutions on workstations in aeronautics, we decided to invest in Linux technologies applied to embedded and real time projects.

This choice was based on the idea that a Unix like solution useable on low prices devices such as PC based hardware, and with no royalty cost, would certainly be very interesting.

The adoption of this OS as the base of the great majority of our projects has been done into successive steps :

- Use of a Linux distribution on standard PC, then on more industrial form factor based on the PC architecture (PC104 stack, single board computers, Compact PCI racks).

- Then replacement of the distribution by home made reduced embedded Linux.

- As long-term real time OS users, Linux was not able to offer us the traditional performances of legacy solutions in term of hard real time, so use of real time extension coupled to Linux.

- Then finally use of Linux on other architectures that are used in industrial projects, such as PowerPC (VME boards and racks) and Arm (mobile or low cost devices).

With such a step-by-step approach, we have succeeded in the adoption of Linux[1] for the whole spectrum of requirements and hardware architectures of the projects we develop. This adoption has been done progressively during 4 years, and today Linux represents merely 70% of our software development business.

In the next paragraphs, we will emphasize 3 aspects of Linux used for industrial projects :

- Using Linux for an embedded device
- Using Linux in real time context
- Porting an application from legacy Real Time OS towards Linux.

2. Using Linux for an embedded device

2.1 Why does Linux fit to main design issues of an embedded device ?

When you start developing an embedded device, there are some basic characteristics that your software must comply with.

---

[1] Alone or coupled with a real time extension

The first one is that your device will not necessarily be built with an Intel, AMD or whatever x86 derived processor. For many reasons (consumption, long term availability, cost etc…), many embedded devices are built with Arm, PowerPC processors, or even micro controllers.

So the software environment you use must be available on the processor you select, and if possible be usable on the maximum of processor architectures, so that the coupling between your application and your hardware is as limited as possible. If you think on long term, using a software environment that allows you to swap from one processor to another without changing the basic software is a good idea, because it allows you to easily follow the innovations on hardware without great impact on your software.

Linux is particularly adapted to this criterion, as it is available for a very important set of processor architectures, including those that are generally used in embedded devices.

The non exhaustive list of processors and micro controllers used in embedded devices, for which Linux ports exist in kernel 2.6, is :

- X86 architecture (Intel, AMD, Via etc…)
- Freescale 68K and PowerPC
- ARM (many sources)
- H8/300, SuperH
- AVR32
- v850
- Xtensa
- Blackfin

The 2.6.23 kernel offers about 24 distinct architectures[2], and this number is increasing from version to version.

The philosophy of the sources organization suits well with a broad support of processors. The sources content all the currently supported architectures in parallel, and it is at configuration and compilation time that the developer selects which processor he wants.

With such a strategy, all drivers and features not related to a precise architecture of processor are shared among all, so the kernel improvements are available whatever processor you use. The adoption of a new architecture is limited to the files directly depending on the architecture. Even if these files are strategic ones, they represent a small percentage of the total kernel sources.

Another important point when you select an OS for an embedded device is that the hardware requirements of this OS regarding resources (memory and mass storage) must remain reasonable, much more tiny than those of a PC that currently offers hundreds of gigabytes of disk, and at least hundreds of megabytes of memory.

On the embedded market, the size of memory and storage are generally in the range of few megabytes for both. And as embedded devices are sometimes built in very big volumes, each megabyte saved can save big amount of money, due to the number.

Everyone who as already installed a Linux distribution can be sceptic, as Linux distributions generally install gigabytes of software components that require well-dimensioned memories.

In fact, when you work with Linux for embedded device, you must not think to reuse a distribution (who is in any case available only for x86). The good method is to rebuild your Linux file system from scratch, just putting in your device what you really need for its dedicated functions.

With such a strategy, you limit the number of processes that will run on the device, and in consequence the amount of memory that is necessary. You also limit the number of files in the Linux file system[3], and so the size of the mass storage.

A good idea is also to tailor the kernel itself to the precise characteristics of the hardware platform, so that you optimise the kernel requirements in term of memory, and the impact of the kernel and its components on the mass storage.

By following these basic rules, it is possible to target sizes like typically 4 MB of mass storages, and 8 MB of memory, which is acceptable for many devices.

Regarding mass storage, embedded devices generally use Flash instead of rotating hard disks. There are two kinds of Flash disk :

- some that offer an IDE interface (CompactFlash, DiskOnModule), and so emulate an IDE hard disk. The normal IDE drivers of the kernel support them.
- the others of type NAND or NOR flash They are generally managed by the MTD framework of the kernel. A great number of chips are managed by native sources of the kernel. If a chipset is not supported directly in the kernel, the supplier generally offers a driver that he developed himself[4].

---

[2] Some with little variants

[3] With Linux, there is not only the kernel image, but also a root file system that is filled with a Unix like directories structure
[4] This point must nevertheless be checked before the flash choice.

Finally, embedded devices generally need to boot very quickly, and users generally stop them without a clean shutdown, simply by switching them off.

So the software environment must deal with these constraints.

Boot time is the sum of 2 factors, the time before Linux starts (bootloader, and BIOS if X86 architecture), then the time due to Linux start, until the application is launched.

Concerning the first one, the developer must tune the bootloader and eventually the BIOS to reduce their duration.

Concerning Linux itself, with a dedicated embedded distribution, start up duration targets from seconds to few tens of seconds are quite accessible (depending on processor speed, and quantity of basic services started).

In order to deal with switch off without proper shutdown, the Linux kernel offers file systems with journaling capabilities to reduce the risk of file system corruption. This solution is totally usable on a PC as it dramatically reduces the risk of problems, but it is not sufficient on an embedded device that must start in any case, without human intervention. The solution is to work with a file system in RAM disk, so that even if the file system is corrupted during the switch off, the next boot regenerates it : the RAM disk is copied from mass storage to RAM at each start, and it is only the copy that can be altered at switch off.

In conclusion, Linux is able to answer to the main issues we need to deal with in embedded context, either basically or by an adequate strategy of use.

2.2 Ethernet redundancy solution based on Linux

We have used Linux to build the embedded software of a device that was designed to offer redundancy on Ethernet networks.
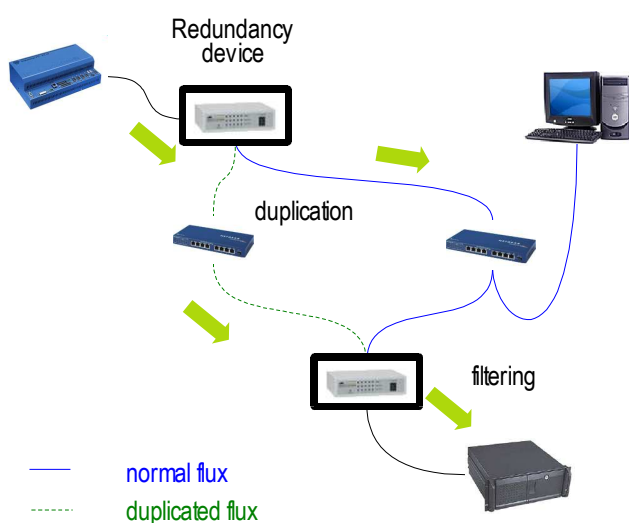

Figure 1: The function of the device

The purpose of the device is to transparently duplicate Ethernet flux between distinct computers, without having to rewrite the existing software applications. So the devices are connected between the computers and the switches, and they duplicate Ethernet frames on the two redundant networks, and then filter the frames received twice.

They act at the Ethernet level of the TCP/IP stack, and they are completely ignored by the computers on the network, and by their software applications.

The customer requirements about the solutions were:

- Complete access to the source code of all the software components

- Possibility to use distinct hardware architectures and form factors for the device

- Use a software with performant and reliable TCP/IP stacks

- Quick availability after power on

We have proposed Linux because all the software components are delivered under the GPL license, so it guarantees the accessibility to the OS sources. As the application was also developed as Open Source, the first requirement was met.

The device possible first configurations used a X86 CPU in Compact PCI rack, and a PowerPC CPU in VME rack. Ethernet ports (3 ports needed per device) were located on the CPU itself, plus an additional multiport board with the same format than the rack. As Linux is available for both architectures and has a huge set of Ethernet drivers, it was also a great candidate for this requirement.

It is well known that Linux TCP/IP stacks are very good, as Linux is the base of the majority of the servers of the Web.

The quick availability after power on was reached thanks to the development of a very tiny embedded distribution, with just the loading of the Ethernet chips drivers, and the start of the Ethernet frames treatment application.

So Linux succeed to build such a dedicated embedded application that acts like a firmware in the device, and allows very versatile configuration without application rewriting, the only change is to select the good Ethernet driver when we go from a configuration towards the over (and of course recompile the sources with the corresponding tool chain if the architecture changes).

3. Using Linux in real time context

3.1 What kind of real time for your project ?

For a company like us that has been for years working with many Real Time Operating Systems

(RTOS), the use of Linux based solutions could not lead us to loose the level of determinism we need in our projects.

When we look back at projects built around legacy RTOS, some are just embedded development without real time characteristics, but some really need to rely on a hard real time environment.

When working with RTOS, as they are both designed for embedded and real time purposes, there was no question regarding the level of determinism that the project needed, in any case the RTOS could sustain any requirement.

With Linux, the developer must answer to the following question during the design phase : "Is the process simply an embedded device, or does it need soft real time performance, or hard real time performance ?".

Depending on the answer to such a question, the strategy to use Linux will be different.

For just embedded needs, Linux will be useable with the strategy explained in § 2.

If the requirements are soft real time, Linux can be used, but the developer will have to adopt an adapted strategy to get soft real time performances.

And if finally the process implies hard real time characteristics, Linux won't be the solution, but the couple Linux + real time extension will be.

The following paragraph will detail the positioning of Linux in term of real time capabilities.

3.2 A tour of real time solutions with Linux

Linux, like Windows, is a general purpose Operating System.

Its main mission is to manage a lot of concurrent processes on a machine (desktop or server), with a global time-sharing among all.

The scheduler's basic strategy is not to advantage a precise process, but make all processes run on a sufficient large time scale, so that all are going head on a fair basis.

It is nevertheless possible to place selected processes in another class, called "real time" scheduling class. When attributed to this type of scheduling, the processes are placed above all standard processes in the scheduler's strategy to allocate the processor(s).

All processes with that class are ordered through increasing priorities. The process with the top priority always preempts processes with lowest priority. If many processes compete at the same level of priority, processes are managed according two policies :

- SCHED_FIFO : no reallocation until explicit release by the running process
- SCHED_RR (Round Robbin) : time sharing among processes in the same priority level.

This kind of scheduling is similar to the scheduling principles of RTOS, so Linux can reach in term of scheduling the same functionality than RTOS, if the developer uses the right scheduling class.

It does not mean at all that Linux is a RTOS, because the important factor is the time to take into account an interrupt, and to react to it at applicative level, in a well-known and limited duration.

Due to the internal nature of the kernel, it is not possible to guarantee in any case this duration, unless setting this limit to very high values, of course.

Until the arrival of kernel 2.6, the kernel code was not preemptible, so that means that when kernel code is currently executing, if an interrupt occurs, the interested process can not react to it until the end of the kernel code. As kernel code execution (drivers, system call etc…) before releasing the processor is not bounded throughout the kernel, there can be no guarantee in term of reaction latency.

In version 2.6 of the kernel, there is a possibility to make the kernel preemptible, through its configuration before compilation.

When this configuration option is selected, the kernel code can be pre-empted. This leads to best reaction latencies than with a not preemptible kernel.

Nevertheless, large portion of the kernel code are executed with interrupts masked, and there are not preemptible sections due to re-entrance problems, so even with preemptible kernel, one cannot guarantee bounded reaction latency.

That's the reason why a preemptible kernel coupled with SCHED_FIFO or SCHED_RR scheduling class can answer to soft real time needs only. In such a configuration, the average latency can be good, but potentially not the worst case. So the true hard real time performance remains impossible to Linux.

To deal with hard real time, the only solution is to work with a real time extension that is coupled to Linux.

The main real time extensions for Linux are :

- RTLinux was the first one, designed by the University of New Mexico. After a first phase where the product was GPL, it divided in 2 products driven by a company, FSMLabs. The basic version of RTLinux was available under a free license (RTLinux/Free), and the performant one became a commercial product

(RTLinux/Pro). WindRiver has recently bought RTLinux to introduce it in is offer as RTCore, beside Linux and VxWorks offer.

- RTAI was derived from the former RTLinux. The Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano, develops it. It is and remains a GPL solution.

- Xenomai has started in the RTAI environment has a new branch, and is now completely autonomous. It is the newest of the 3 extensions, and seems promising. It offers a neutral nucleus with skins API for many RTOS (VRTX, VxWorks, µITRON, RTAI), and a neutral one. It also offers a smooth migration path from Linux to real time and vice versa.

All these extensions use the main rough principles. They are implemented as a set of kernel modules, and so rely on Linux for their startup.

Once the extension is loaded, it takes precedence on Linux by distinct technical means (virtualization of interruptions in RTLinux and initial RTAI, ordered pipeline of interruptions with the new RTAI and Xenomai).

The extension implements a fully deterministic RTOS. When there is no activity in the RTOS, the extension gives the processor to Linux world (Linux is the idle task of the RTOS).

The extension offers an API to make IPC among tasks in the real time part, but also IPC between tasks in the RTOS and processes in Linux.

When using such solutions, the developer has to split its application in 2 parts :

- The part that must be real time, which is implemented in the RTOS

- The part that has no real time constraints, that is implement has legacy Linux process.

For instance, in a data acquisition chain, the reception of the data and their time stamping would be done in the RTOS, with real time performances, and a graphical representation or a send to a storage server over a network would be done in Linux. The exchanges between the two parts would be done through exchange FIFOS.

With such a design approach, the developer "put the right thing in the right place", but this leads to a more complicated design. It is necessary to assure that Linux will have the processor enough to avoid saturating the intermediate buffers through which the real time part sends data to Linux. So there is a trade off between the load of the global system, and the buffers size.

Linux drivers are not available in the extension, so Inputs/Outputs done in the real time part necessitate a Real Time driver.

There are greatly fewer drivers directly available for the extensions than for Linux, but there is a framework to develop Real Time drivers (RTDM). Drivers are generally not built from scratch, but adapted to the extension from their Linux version.

The real time part of the application can be built through 2 types :

- As kernel modules directly running in the RTOS, for the maximum of performance, but less comfort (no memory protection, debug less easy)

- Or packaged as special Linux processes, that are not scheduled by Linux but by the RTOS. This allows protection by the MMU, and facilitates the debug process.

3.3 An example : numerical data recorder for automotive

One of our customers had to develop a new generation of data recorders to store the flux of information exchanged on numerical buses on modern vehicles.

The hardware platform was a PC104 stack with an Intel CPU, a CAN board and another board with DSP, who acts as a specialized co processor for the main CPU.

The requirements for the software were :

- Ability to precisely time stamp the data received on the CAN buses.

- High bandwidth towards a hard disk used for storage.

- Full determinism for the exchanges between the main CPU and the DSP coprocessor.

To fulfil these requirements, it was decided to use a Linux based solution, coupled with RTAI to ensure complete determinism of the software.

We have selected a 2.4 kernel. The project was made after arrival of 2.6 kernel, but for industrial applications, we generally adopt a very conservative approach. We generally start using a major kernel version only a quite long time after its release, so that a maximum of potential problems have been already solved.

Another reason to choose kernel 2.4 is that Linux drivers for CAN and DSP boards were available for 2.4, and not for 2.6.

Our job was to set up the basic software environment (Linux + RTAI base), and to deliver a skeleton of the final application.

Due to the fact that CAN and DSP were managed by the real time part of the application, it was necessary to port the drivers. As there was a single task that would drive a board, we have not used RTDM model, but the existing Linux driver code was adapted and used as a hardware management

library linked with the task. As the task was a kernel module, there was no trouble to access the hardware.

The main job for this port was to remove things related to device nodes, and replace Linux kernel functions by their equivalent for RTAI.

After installing and validating the Linux kernel, the RTAI extension and the ported drivers, we have built the skeleton of the application. The idea was to let the customer deal with the details of the application and to concentrate ourselves on what was related to RTAI :

- Global design between Linux processes and RTAI tasks
- Implementation of IPC and synchronizations between these processes and tasks
- Access to the CAN board and exchanges with the DSP

This job was made thanks to the global requirements written by the customer.

We finally delivered the platform with Linux and RTAI installed, and the skeleton developed. The customer finalized the development by himself : he added the final code round the templates of functions of the skeleton.

Such a development approach allowed using Linux plus real time extension in the new device, with a good share were each company acts where she has the maximum added value.

4. Porting an application from legacy RTOS to Linux

In the industrial world, there are a lot of architectures built around 68K boards with VME interface towards Inputs/Outputs, and legacy RTOS.

Due to the increase in computing needs, it is sometime necessary to replace this generation of boards by new ones, much more powerful.

But in the history of RTOS, there have been great changes on the market. Some have disappeared, some still exist but their relative part has greatly decreased, so the drivers or BSP for these OS are not available for new hardware.

When a complete software application exists and has been validated, and it is necessary to change the hardware, this can trigger the need to port this application towards the new Operating System.

The idea to minimize the impact, and in consequence the costs, of such evolution is to try to preserve as much as possible the sources of the existing application, and to manage the port at the system level.

By remapping system calls of the old OS to the new one, and developing an emulation library when strict remapping is not possible, the application sources are very few impacted. In consequence, the validation phase is much more light than if the application was rewritten for the new environment, and the regression risk is minimized.

This approach is possible only if the developer has good knowledge of the two OS : the old and the new one.

We have managed this kind of project for a port of a SNMP stack from OS9 towards Linux.

The initial stack was running on a 68K CPU, and the new hardware was a PowerQuick (PowerPC family).

OS9 was not available for this new CPU, so it was quickly decided to use Linux.

The SNMP stack was just a part of the application, but the port was a mock up to evaluate both the feasibility and cost of the same port for the entire application.

As mentioned earlier in this article, the initial version used OS9 not for real time purposes, but just as an embedded OS on 68K boards, so there were no time constraints that could justify the need of a real time extension.

The initial sources of the application were written in Ansi C, a few years ago.

So the first problem we had to face was not directly due to Linux, but to the difference between the OS9 tool chain and the current Gnu tool chain : as modern compilers make much more controls than old ones, some syntax errors were detected when compiling for Linux, when there was no trouble when compiling for OS9.

These difficulties must be taken into account when planning a port, even if they are not directly related to Linux, because they can necessitate hours of sources cleanup, if the old compiler was too tolerant.

With the Gnu tool chain, you can decrease the level of controls the compiler makes to avoid correcting the sources, but is it really a good solution ? The best thing is to rely on the compiler's check to get the cleanest sources, in order to avoid future problems. That is what we have done regarding this point, even if this generated additional work.

We faced to another difficulty also related to compilers. As some data structures were not correctly aligned in the initial sources, the old and new compilers did not manage these structures the same way. So there were some troubles when dealing with offsets of members of these structures, or computing their length.

Like sources syntax, this point is not directly due to Linux, but it was important because such problems involved bad functionalities of the ported software, and were difficult to identify.

An important difference between Linux and some RTOS is that RTOS may have a flat memory model,

without any boundaries between global variables spared into different executables.

With Linux, every executable has its own memory space, under the control of the MMU, and global variables of one executable are not seen by the others, except if the developer used dedicated shared memory API.

Under OS9, it is possible to use flat memory model, or a segmented one with MMU. This is the developer's choice.

Unfortunately for the port we faced to, the initial developers used the flat memory model, with shared variables between the distinct executables of the SNMP stack.

An additional difficulty was that the shared global structures had members that were pointers on members on other data, so these pointers were shared between distinct executables.

With a flat memory model, there was no trouble, but with Linux, the pointer that had sense in the executable where the variable pointed to was located, had no sense for other executables.

To solve this, we had 3 solutions :

- Replacing the OS9 architecture with many executables, by a single multi threaded Linux process (convert OS9 executables to Linux threads). In this case, as all threads shared the same addressing space, the shared pointers would be correct in each thread.

- Use the shared memory API to share all needed structures, and replace all pointers by offsets regarding the base of the related shared data, to avoid pointers that have no sense.

- Use the shared memory API to share all needed structures, and try to map all data to the same address for each process, because the API allows to make a hint for the mapped address.

The best possible solutions would have been to use a single multithreaded process, but we did not identify the use of pointers in shared structures early enough.

So we started the port with a multi process architecture, and use of shared memory API.

When during the test phase the pointers problem appeared, we preferred to map all the shared data to the same address for each process, to avoid to swap from multi processes to multi threads.

Converting pointers to offset would have generated too much source rewriting, and after analysing the memory mapping of all processes, there were identical addresses ranges free for all. A hint on these addresses for all shared data were successful for each process, so with this technique, the original data shares were OK even on the Linux port.

The only penalty of this technique was that in case of software evolution, it would be necessary to check that the used address zone still remained available for each process.

The last OS9 features that were not so easy to port were signals and alarms.

In a multi process OS9 environment, developers often use a lot of signals to synchronize processes or trigger functionalities. OS9 signals are very reach, with a lot of values available for users.

Linux signals are poorer, and they are not really intended to be the base of IPC between processes. To minimize the changes on the sources, we have developed some emulation of the OS9 signal API, based on the Linux SIGUSR1 signal, with associated data containing the value of the OS9 signal (OS9 users signal multiplexed to a single underlying Linux signal).

As the original OS9 sources used a huge quantity of alarms running in parallel, we also needed to emulate the alarm API through the launch of alarm threads, one thread per alarm duration.

Concerning other system aspects, it was quite easier to port them, mainly by simple remapping through an include file that was included by all sources. This file was redefining the OS9 system call in the equivalent one for Linux.

For a few system calls, it was necessary to treat them in a simple compatibility library, mainly due to differences in the parameters between the two OS, that prevent simple remapping through the compatibility include file.

In conclusion, for the majority of the system calls encountered, it was not so difficult to port them at system level, thanks to a good knowledge of both OS9 and Linux.

Signals and alarms were a little bit challenging, due to the fact that they are very common tools in OS9, and very often used in the original sources, but less central in Linux, and so less easy to use in the Linux version.

Memory model and compilers differences were the most difficult aspects to face to.

Starting with a flat memory model is a point to check when porting to Linux, because of the consequences on data sharing.

The difference between compilers, mainly if the original one is quite old, can lead to a lot of unforeseen job, because it can impact all the sources set, and potentially generate hours and hours of corrections.

After treating these difficulties, the port was finally achieved and successful.

The test campaigns demonstrated that the ported software features and performances were identical to the initial one.

So this mock up reached its main goals : demonstrate the feasibility of a port of the whole application, help estimate the global cost, and point the main difficulties of such a job.

4. Evolutions management

In the industrial world, when an application has been successfully validated, there is generally no reason to change anything, unless bugs appear.

In the opposite, Linux and open source software are constantly evolving, because project teams deliver new releases at high rates when projects are active.

For instance, minor versions of Linux 2.6 kernel are released about every 2 or 3 months.

That is clear that a company cannot afford to maintain the OS base of its product in phase with such a rhythm of new deliveries.

So is the evolving world of open source software a bad thing regarding the adoption of Linux in industrial projects ?

In fact not really, because you perfectly can "freeze" the situation to that particular version used to develop and validate the application or product. In our company, we do consider that both Linux kernel sources, basic GPL software used and the application itself we have developed are the distinct components of the project, and we manage them collectively as a whole, so that we are able to always restart with the same basic environment.

And if we discover some problems in the OS or in basic GPL software used, we have more liberty than in RTOS world.

With legacy RTOS, there are indeed less new releases. But when you face a bug or a problem with the current release of the RTOS, there is generally no solution but upgrading it to the new one that will correct it, as editors don't want to correct old releases.

With open source software, you have much more solutions to choose from :

- As with RTOS, the first one is to adopt the latest release, if it corrects the problem you face to.

- The second one is to locate in changelogs, or by comparing sources, the precise corrections corresponding to your problem, and to selectively backport them into the version you used when developing. Even if it is not always easy, it can be a way to avoid revalidation process due to OS release change.

- And finally, if the problem is still not corrected in the latest version, you can try to correct it by yourself, and then recontribute it to the project team.

5. Conclusions

In this article, we have browsed distinct aspects of using Linux in the kind of projects that our company is developing, with embedded, real time characteristics.

Today, we can say that according to our point of view Linux is perfectly usable as a base for these kind of projects.

Linux comes with a lot of high quality software development tools, like versioning tools (CVS, SVN …), compilers (GNU chains), documentation tools (doxygen) etc….

Using open source software is interesting due to full access to the sources, absence of royalties for deploying products, or free availability of a lot of developing tools.

Nevertheless, new adopters must have in mind that if buying costs are quite absent, it is necessary to invest in developers knowledge because of the adoption of a new technology, but also because using free software changes the way of working : you need to interact with a community of developers, instead of a supplier's maintenance team, for support, bugs correction etc…

The interesting point is that you convert money that you used to buy software to reinvest it in your team's human skills, and we do think that it is a quite good investment !

6. References

[1] 01 Informatique, n°1897, March 23 rd 07.

[2] www. linuxdevices.com/articles/ AT7065740528.html

7. Glossary

API: Application Programming Interface

BSP: Board Support Package

GPL: GNU Public License

MMU: Memory Management Unit

RTOS : Real Time Operating System