

Getting Linux into a train HVAC (Heating, Ventilating and Air Conditioning)

Markus Kreidl

Liebherr Transportation Systems
Liebherrstrasse 1, A-2100 Korneuburg
markus.kreidl@gmail.com

Prof. Nihcolas Mc Guire

OpenTech EDV Research GmbH
Lichtenstein Str 31, A-2130 Mistelbach
der.herr@hofr.at

Abstract

Open source and free software is great constantly growing technology pool consisting of millions of tools, rapidly developing technologies and a solution for nearly every technical problem. Some parts of the open source world are often determined by asynchronous development and some parts are determined by a kind of strict, slim and goal oriented work cycles. Both parts can be a prolific environment for new technologies and innovations. Industry - more specific railway industry - is one the most conservative markets with respect to technologies, standards and the corresponding products. Those products are developed over years adhering to a strict life cycle and must be maintained over decades. Typically railway solutions (products) are embedded solutions with the focus on low resource and safety constraints. The open source community offers many solutions, which fit into those industrial applications but some of them cannot be used because of the gap between the "Fast and Cool" open source world and the long-term life cycle focused industry. This paper is motivated by the possibility to minimize the gap between the open source and a standards dominated worlds, to show ways how to integrate open source into industrial DLCs, to give facts between proprietary and Linux total development costs and to give a feeling how many systems in different vertical markets are already in use and finally to decrease prejudices (on both sides).

1 Introduction

FLOSS (free libre open source software) over the past years has developed methods and tools to create diverse packages with a high code quality. More and more companies are realizing that those packages, driven by a strong community, can replace commercial products. During the last decades industry developed various processes and standards to assure code and product quality. Most of those process have been followed by companies for a very long time. So industry has no mechanisms at hand to integrated FLOSS in their daily work and simply don't know how to deal with FLOSS.

2 Definition FLOSS

In general FLOSS (Free/libre open source software) is the higher abstraction for open source and free software.

2.1 Free software definition[1]

"Free software" is a matter of liberty, not price. To understand the concept, you should think of "free" as in "free speech," not as in "free beer".

Free software is a matter of the users' freedom to run, copy, distribute, study, change and improve

the software. More precisely, it refers to four kinds of freedom, for the users of the software:

- The freedom to run the program, for any purpose (freedom 0).
- The freedom to study how the program works, and change it to make it do what you wish (freedom 1). Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help your neighbor (freedom 2).
- The freedom to improve the program, and release your improvements (and modified versions in general) to the public, so that the whole community benefits (freedom 3). Again, access to the source code is a precondition for this.

A program is free software if users have all of these freedoms. Thus, you should be free to redistribute copies, either with or without modifications, either gratis or charging a fee for distribution or the services related to implementing changes, to anyone anywhere. Being free to do these things means (among other things) that you do not have to ask or pay for permission.

2.2 Open source definition[2]

Open source doesn't just mean access to the source code. The distribution terms of open-source software must comply with the following criteria:

1. Free Redistribution: The license shall not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license shall not require a royalty or other fee for such sale.
2. Source Code: The program must include source code, and must allow distribution in source code as well as compiled form. Where some form of a product is not distributed with source code, there must be a well-publicized means of obtaining the source code for no more than a reasonable reproduction cost preferably, downloading via the Internet without charge. The source code must be the preferred form in which a programmer would modify the program. Deliberately obfuscated source code is not allowed. Intermediate forms such as the output of a preprocessor or translator are not allowed.

3. Derived Works: The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software.
4. Integrity of The Author's Source Code: The license may restrict source-code from being distributed in modified form only if the license allows the distribution of "patch files" with the source code for the purpose of modifying the program at build time. The license must explicitly permit distribution of software built from modified source code. The license may require derived works to carry a different name or version number from the original software.
5. No Discrimination Against Persons or Groups: The license must not discriminate against any person or group of persons.
6. No Discrimination Against Fields of Endeavor: The license must not restrict anyone from making use of the program in a specific field of endeavor. For example, it may not restrict the program from being used in a business, or from being used for genetic research.
7. Distribution of License: The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties.
8. License Must Not Be Specific to a Product: The rights attached to the program must not depend on the program's being part of a particular software distribution. If the program is extracted from that distribution and used or distributed within the terms of the program's license, all parties to whom the program is redistributed should have the same rights as those that are granted in conjunction with the original software distribution.
9. License Must Not Restrict Other Software: The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be open-source software.
10. License Must Be Technology-Neutral: No provision of the license may be predicated on any individual technology or style of interface.

3 What can be interesting for industry?

Many diverse software packages and complete distributions for embedded systems are part of FLOSS and can be of interest for industrial embedded designs. Selecting the right FLOSS components can fit into almost any design. A basic embedded system consists of a suitable kernel for an architecture, a library which defines system calls and a large amount of generic user land applications. FLOSS components also contains a wide range of tools for building, debugging, tracing and maintaining software. The Linux kernel covers a wide range of 14 different architectures. The Linux kernel is a conservative implementation of a well-specified Unix OS. Despite a rapid community development, portability and documentations make the integration easy. Any Unix-like operating system needs a C library: the library that contains kernel space user space boundary (system calls) and standard I/O and generic functionalities. A wide range of different diverse user land applications is available to provide a full-featured embedded system, which can fulfill challenging industrial requirements. Compilers, build systems, debug and code editing tools are available. Architecture depended tools like compilers and debuggers are covering all Linux kernel target architectures. FLOSS also contains diverse components for bug tracking, source control management and test benches.

3.1 Industrial software development life cycle

What does development cycles do for the industry? A DLC defines required tasks for specifying, developing and maintaining software. A DLC process gives all participants during a development of a product the possibility to interact in a well-defined way. This should give companies the possibility to make repeatable, and predictable developments and to improve of productivity and assure quality, but this is ideal. A typical simplified software DLC would be:

- Definition of software requirements
- Design
- Implementation
- System test
- Maintenance

4 Open Source DLCs

Open source development is not dominated by DLC like industrial product development. Standardization, a strict organization and dedicated responsibilities are instruments to increase the probability of a high code quality. Open source development life cycles attributes are:

- Async package development: FLOSS packages are developed in extremely modularized manner by different cultural parts all over the world.
- Peer review: Agreements are achieved by open public review process.
- Infrastructure: Bug-fixing databases and source control management are maintained in a global way.
- Well defined policies: Well maintained FLOSS packages provide transparent and traceable policies for the package life cycle.
- Integration aspects: FLOSS packages are integrated in a wide range of applications, distributions and architectures.
- Feedback mechanism: One of the most distinctive features of the FLOSS DLC is its feedback mechanism a side of high the high quantity feedback due to open infrastructure (e.g. mailing lists) the feedback quality is also high informed users produce informed feedback.

DLC processes are applied in several big distributions. Debian for example exercises a strict development life cycles with dedicated responsibilities down to the package level. The infrastructure for administration, versioning, test benches, source control management is highly consistent developed to handle a global distributed development at a high level. The reliability of this process is well reflected in the industrial usage.

top500 super computer server statistics [3]

Operating system Family	Market share	
	Count	[%]
Linux	443	88.6
Windows	5	1.0
UNIX	22	4.4
BSD Based	1	0.2
Mixed	29	5.8

Mobile communication sector (Thousands of Units)[4]

Company	Market Shares 2008	
	Sales	[%]
Symbian	72,933.5	52.4
Research In Motion	23,149.0	16.6
MS Windows Mobile	16,498.1	11.8
Mac OS	11,417.5	8.2
Linux	11,262.9	8.1
Palm OS	2,507.2	1.8
Others	1,519.7	1.1

A marked research from the group Venture Development Corp. (VDC) [5] researched willingness to migrate to Linux from a commercial OS. The result of the survey shows that a embedded system manufacturers are increasingly favouring Linux as possible choice.

Future projects	%
Opt free in-house Linux	71
Plan to use a commercial Linux	16
Plan to use a commercial OS	12
Stick at a in-house OS	1
Usage in current projects	%
Commercial OS	39
Linux free in-house	29
Linux commercial	12
No usage of a formal OS	7

Referring to VDC Linux is a an attractive OS choice for embedded development teams. The main reasons in their opinion are:

- Royalty free run-time costs
- Advanced networking capabilities
- Technical features and the large
- Base of engineers familiar with the OS.

A statistic from Market Share [6] showed that Linux based Desktops in July 2009 reached a market share of 1%. In general the workstation market is a slow market which is hard to enter.

5 Kernel DLC

5.1 Developer Hierarchy

To understand the Linux kernel development it is important to know something about the hierarchical organization. Structuring in well defined subsystems is a base for the four levels - ordinary developer, maintainer, trusted lieutenants, and the benevolent dictator - developer hierarchy. An ordinary developer can write patches or proposals and submits his work to a maintainer of a particular component of the kernel. The driver maintainer sends the tested patch to the subsystem maintainers, who are in direct contact with benevolent dictator, which is Linus Torvalds at the moment. He writes code and issues the general direction and plays an integrative role. Andrew

Morton also takes a special place, he is the maintainer of the Linux 2.6 kernel series.

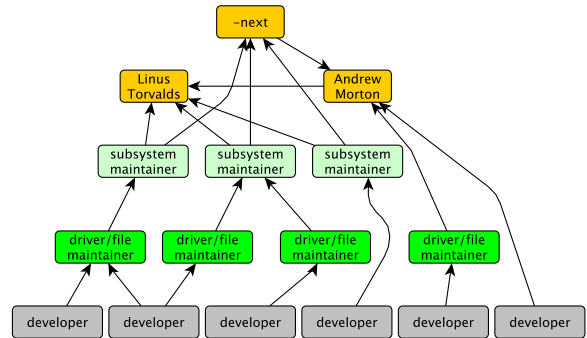


FIGURE 1: Developer hierarchy

For the kernel release 2.6.27 a total set of 10625 patches were submitted from 1015 persons. Averagely, every person submitted 10 patch sets[7].

5.2 The staged process

The kernel development DLC is a well-defined strict process. The process was adapted and optimized over the years according the needs of a global active community.

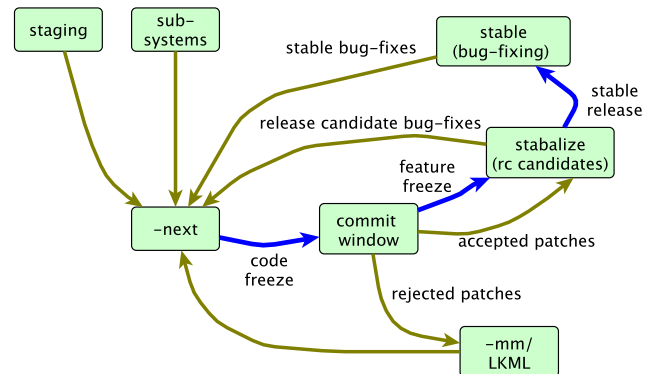


FIGURE 2: Linux DLC

Linux-staging tree The linux-staging tree was created to hold drivers and file systems and other semi-major additions to the Linux kernel that are not ready to be merged at this point in time. It is here for companies and authors to get a wider range of testing, and to allow for other members of the community to help with the development of these features for the eventual inclusion into the main kernel tree Greg Kroah-Hartman (employed by Novell working in the Suse Labs division)

Linux-next tree The linux-next tree contains patches that have the possibility to go into mainline kernel after the current release gets stable. This should help Linus Torvalds to integrate between the subsystems

balanced patches and Andrew Morton can work on future kernel versions without drifting away from the current release. The linux-next tree setup consists of an automated build system, with different build configurations where developer is notified by mail to correct their faults.

The rules for linux-staging:

The linux-next integration testing is not a judgment of your code. The purpose of linux-next is for integration testing and to lower the impact of conflicts between subsystems in the next merge window. You need to ensure that the patches/commits in your tree/series have been:

- submitted under GPL v2 (or later) and include the Contributor's Signed-off-by,
- posted to the relevant mailing list,
- reviewed by you (or another maintainer of your subsystem tree),
- successfully unit tested, and
- destined for the current or next Linux merge window.

LKML (Linux kernel mailing list) For the Linux kernel development is the Linux kernel mailing list the main mailing list. The majority of the announcements, discussions, debates and coordinates are done. The LKML is the the top level mailing list of many different subsystems mailing lists. The LKML is a very high level mailing list wich usally receives 200 to 300 mails each day. The mailing list can be seen as the principal communication channel.

The current kernel release is the base for the development. After a 2 weeks the merge window is closed, and a 7 to 10 weeks stabilizing phase start. This time is determined for small adaptation, testing and integration. Various automated test benches should assure the quality of the system.

6 FLOSS usage in industrial environments

To choose single packages like a xml parser for application development is a common way. For embedded systems it is not common to utilize single packages and to build your own system out of these. Choosing a whole distribution moves a lot of work into the community, so the company has reduced work with package compatibilities and the danger of using orphaned packages. Different scenarios for integrating FLOSS in an DLC industrial environment are:

- Grab and run

The Grabn run method seems to be the easiest way for companies to start development. A software snapshot is taken and used decoupled from

the community. A reason for companies doing it this way is, that they have everything they want for the moment and do not need more. In fact they need a certain amount of time to integrate the software but they have no philosophy how to benefit from the community in the future. There commonly is not even a strategy how to deal with bug fixes, new releases and maintaining....

- Integrate the community

This method has the advantage of consuming the advantages of a big community without paying something back. In this model the company is an active consumer that means there have been methods integrated to dealing with FLOSS during a product life cycle. This way of dealing with FLOSS has no plans to move IP independent developments into the community. The drawback is that the company has no say in what direction development will go in the future. Further more the willingness of the community to accept needs from a particular company will depend on this companies acceptance in the community - pay in cash or pay in code...

- Integrate into the community

This model is similar to the integrate the community. The company gives improvements or drivers back to the community. These modifications feed back in to the community get tested by a bigger spectrum and the overall quality increase. Another positive effect is more extensive bug tracking and peer review. When a company decides to work in a stronger relationship with FLOSS it is important, to be sure about licenses, to be aware the community redundancy, to assign time for communication and to not violate/modify interfaces.

7 Total Cost of Development

The total cost development (TCD) research [8] from the company Embedded Market Forecasters (EMF) is used to reflect how different companies deal with Linux or proprietary operating systems. EMF has analyzed a survey of 606 developers and has reflected the design undertakings and design outcome in relation of embedded Linux versus commercial RTOS.

The following parameters have be analyzed:

- Time from design start to product shipment (design time in month)
- Total lines of new and modified source lines – excluding reuse, commercial off-the-shelf and FLOSS
- Total number of source lines of code written – including reuse, commercial off-the-shelf and FLOSS.
- Percentage of designs that are behind schedule.

- Once the integration cycle starts, developers are asked to grade the accuracy of final design hardware and software versus pre-design expectations. Expectations in performance, systems functionality, and features and schedule are examined. EMF records the percentage of responses for which the final design achieved within 20% of pre-design expectations and within 30% of pre-design expectations.

Vendors Compared: Linux and commercial OSes

Linux	Proprietary OSes
Wind River Linux	Green Hills Integrity
LynxWorks BlueCat Linux	LynxWorks LynxOS
CodeWarrior for Linux	Mentor Graphics Nucleus
MontaVista Linux	Microsoft Windows XPE
RedHat Linux (Ent.)	Microsoft Windows CE
Timesys Linux	Wind River VxWorks

7.1 Design comparison

The following table presents a comparative design undertakings between RTOSes and Linux. Total lines of code written and lines of code indicates the complexity of the undertaking. Costs of design are reflected by

- The average number of software developers per project
- Designs behind schedule
- Number of months behind schedule

	RTOSes	Linux	Ind. Avg.
Avg # SW developers/project	40.5	47.5	30.9
Avg lines of code written x1000	198.5	252.6	186.3
Avg lines of code x1000	704.4	1000	749.3
Total design time (months)	15.5	14.4	15.5
% behind schedule	43.0%	35.8%	41.4%
Avg months off-target for behind schedule projects	3.8	3.9	4

The survey says that taking a superior time-to-market can save a few ten thousands of dollars difference in up-front costs. As a result of the compared data from 2006 and 2007 showed that 7 more developers per project where need in Linux projects than commercial RTOSes while the average design time for Linux project were 1.1 months less. In fact this study gives a reason for convincing to evaluate non commercial OSes.

7.2 Project expectations

EMF examined for 2 years data from more than 1300 respondents. Broadly data was gathered across 10 different vertical markets from a number of different OSes. The following table represents the collected data:

expectations	RTOSes	Linux	Ind. Avg.
below 20% of Performance	66.8	70.0	64.0
Sys functionality	68.4	71.3	65.5
Features & schedule	55.6	59.5	54.3
below 30% of Performance	74.6	75.6	71.8
Sys functionality	75.4	76.2	72.1
Features & schedule	68.1	67.9	64.7

The table presents the results of the data where the results were below a 20% and a 30% pre-design expectations. A significant difference exist between Linux and commercial RTOS design outcomes.

7.3 Linux vendor comparison

EMF took a look at the comparative data between established commercial Linux vendors and the in-house (roll your own Linux). The analysis showed that in-house developers wrote fewer lines of code and fared poorly when it came to final design outcomes compared to their pre design expectations. A plus of the in-house Linux development is the high level of expertise and the lowered level of complexity. The value for insufficient resources stays nearly at the same level for proprietary RTOSes, Linux and in-house Linux. Proprietary RTOSes have a little advance of 2%. A big plus has commercial Linux against RTOSes and in-house Linux regarding the time to develop their concepts and bring them to the market in-time. An in-house Linux is a question of the in-house available capabilities and the willingness of the management to invest in there engineers.

7.4 TCD Analysis

EMF has compared the results of the survey in the year 2003 and 2007 and asked the question why the results for embedded Linux-based designs have improved over this 4 years. The answer is tools. In 2003 tools were very limited and 40% of Linux designs involved in-house sets or limited tools that were available for free download. Different commercial Linux companies developed a comprehensive set of design tools or aquired the assets of other companies like Wind River from FSM Labs. In fact nowadays Linux has the same design outcomes compared with traditional RTOSes.

7.5 TCD Conclusions

The development of embedded Linux design changed a lot in the last years. In general designing with a Linux OS is no longer an expensive and risky undertaking. The use of a commercial Linux OS is more cost effective to an in-house Linux development undertaking. The spectrum of Linux for embedded designs is no longer restricted to a few applications and the designs outcomes data demonstrates the developers can consider using a Linux OS without concern that their product will inferior to end products for a proprietary RTOS.

It should be noted though that some of these numbers may well change in the near future as some of them may be in part due to Linux having only a short history in many companies and thus exhibiting "startup"-costs.

8 Do's and don'ts

8.1 Risk assessment

Before integrating a single piece of software or a whole distributions it is important to make a risk assessment to check out what pain may occur in the future. A fundamental risk assessment needs some time at the beginning, but can save a lot of time and money. A risk assessment that answers the following question can easily be integrated to an industrial DLC during the design phase.

- How is the project/distribution history? Are there many forks? Is the project/distribution developed continuously?
- How big is the community behind the project or distribution?
- How fast is the responsiveness?
- The quality of documentation and bug tracking?
- Can the company deal with different licenses?
- Bugs/patches total received
- Project/mailstone planing

8.2 Do not stick at a version

Sticking at a version is a often used strategy because for the moment everything is satisfied and the effort is straight forward but in fact a decoupling between in-house and the community happens unnoticed. Open source automation development labs (OSADL) has realized this problem, so OSADL offers a stable linux kernel version for over one and a half year that contains bug fixes of versions ahead the current stable version. A continuous upgrade strategy avoids a lot of problem that are occurring when backports are done. In general it is better when defining requirements not to refer to packages + versions number or to refer to a special cpu like powerpc 405gr, but ensure that requirements are generic allowing to satisfy them with current FLOSS offerings.

9 Conclusion

9.1 DLCs related

In general when companies want to use FLOSS in their products they have to adapt their software development process in some parts. That means a longer evaluations phase with a risk assessment during the desing phase is a must. The FLOSS community efforts have resulted in diverse packages for different problems, but the needs for the corresponding projects must be strictly defined to select the best-fit variant. Further more the introduced risk assessment limits risks for the future respectively to get awareness to what can happen in future.

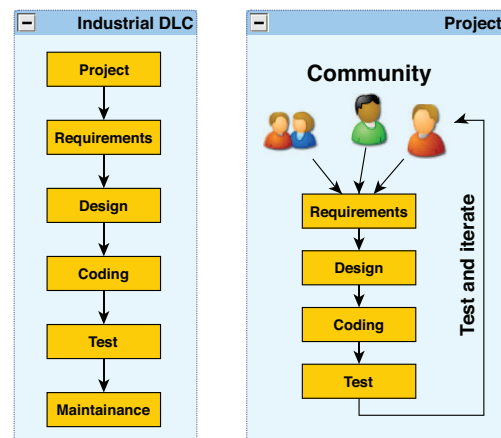


FIGURE 3: *Linux DLC*

The left diagram shows an industrial DLC which passes straight through its phases. Requirements are made based on the project requirements. The coding phase is done based on the architectural design. Each module or function is programmed once. A module and a integration test is done in the test phase. The main-tainance phase can contains tasks to keep the product up and running.

A lot of FLOSS components are not unique like the Linux kernel is. Different needs of different communities are resulting in diverse FLOSS packages from competing project groups. In general FLOSS software has a continuous development cycle, software will be designed, tested and new iteration of design will be initiated.

This behaviour can be a problem for an industrial DLC, but with a risk assessment during the design phase and a FLOSS supervision task during maintainance this can make an integration possible. This supervision task should give engineers the possiblity to monitor (evaluate change logs when a new version is released and monitor the correspondng bug system) used FLOSS packages and also participate in the design phase where needed.

9.2 Impacts

Going to FLOSS can provide an extremely wide scope of perspectives to companies in the railway market. These perspectives can deliver a great resource for companies to persist in a globalised world and changing market demands. Some companies provided conservative solutions for many years, but embedded technologies have progressed and so modern solutions can not be provided without transition to modern platforms. Surely for modernisation selecting a modern proprietary OS is a possibility, but using FLOSS has a lot more advantages. FLOSS can bring a wide range of technologies into a company. It is possible to increase know-how in the company unit, give engineers the possibility to understand systems in a deeper way. Know-how and innovation is nowadays one of the only ways to ensure jobs unless you are a pure production company. Costs, project expectations, complexity and time to market are not an argument against FLOSS, which was shown by the TCD analysis. To minimize risks a Linux vendor can be chosen or a service contract with one of the several embedded Linux support companies can be done. This helps to be in time and speed up know-how transfer to the company. For application developers daily work keeps the same. A wide range of reliable source editing tools, debuggers and compilers is available. For embedded platform engineers changes

can be substantial, they have the possibility to go deep into the system, no stone should be left unturned and no dark corner should be left unilluminated. Engineers have to keep up to date what is changing and try to get a feeling what is going on in the community and where the decisions are moving project to. Core developers should stay at the front lines where technology is evolving to bring it into the company - The Linux kernel and FLOSS is a great place to get into innovations.

References

- [1] <http://www.gnu.org/philosophy/free-sw.html>
- [2] <http://www.opensource.org/docs/definition.php>
- [3] <http://www.top500.org/stats/list/33/osfam>
- [4] <http://www.gartner.com/it/page.jsp?id=910112>
- [5] <http://www.eetindia.co.in/SEARCH/ART/embedded+Linux.HTM>, 2007-10-08 Linux a favourable choice for embedded market
- [6] <http://marketshare.hitslink.com/operating-system-market-share.aspx?qprid=8>
- [7] http://remword.com/kps_result/2_6_27_lt.html
- [8] http://embeddedforecast.com/images/Embedded_Linux_TCD_Analyze