LINUX: A MULTI-PURPOSE EXECUTIVE SUPPORT FOR CIVIL AVIONICS APPLICATIONS?

Serge Goiffon and Pierre Gaufillet

AIRBUS France - Avionics and Simulation Products, 316 route de Bayonne, 31060 Toulouse Cedex 03, France

Abstract: Software plays an important role in modern aircraft. Criteria like portability, reusability, cost savings, maintainability, lead suppliers of equipment and functions to adopt standards. Integrated Modular Avionics is the new paradigm for the avionics world. The ARINC 653 standard has been chosen as the application programming interface, despite the predominant position of POSIX standard in other industries. Nevertheless, both standards have different but complementary approaches, and a multi-purpose adaptable open source software like Linux could be a good candidate to host avionics applications.

Key words: Linux, POSIX, ARINC 653, Open Source, avionics application, DO-178B

1. NEW TRENDS IN CIVIL AVIONICS

The amount of embedded software in modern aircraft has continuously increased over the past 30 years. Today, the complexity of avionics functions has reached a point where a classical “make all from scratch” development strategy is superseded. Future aircraft will continue to embed more and more software. Embedded avionics equipment will tend to become more and more a generic platform whose goal will be to provide allocation and control of computing resources needed by avionics functions : CPU, memory and input/outputs. This new trend is known as Integrated Modular Avionics.

In this challenging industrial context, PDS¹, like operating systems, are a key factor to reduce the time to market of avionics applications: introducing an hardware abstraction layer solves the problem of hardware component obsolescence, and adopting industrial standards, like POSIX², or avionics standards, like ARINC 653³, ensures durability and portability of applications.

Nevertheless, this new trend leads to an increase in the certification cost that is not always compatible with the use of a PDS. This is so because the product does not necessarily fulfill the quality development process required by DO-178B⁴, and the equipment supplier may not accept the extra cost to achieve conformance. The quasi monopolistic position of the PDS supplier may further compound the problem.

In this context, the Linux open source model can provide an alternative solution: low cost access to technology, freedom to modify the source code to adapt it to avionics constraints. But it also raises new questions that need to be addressed: ability to comply with avionics hardware components and embedded constraints, and availability of certification material.

2. POSIX VS. ARINC 653

The ARINC 653 standard has been chosen as the avionics application programming interface for future aircraft, despite the predominant position of the widely used POSIX standard in other industries. ARINC 653 first issue was published in 1997. One year before, POSIX standard real-time extensions were approved as an international standard, ISO/IEC 9945-1:1996 (including 1003.1b, 1003.1c) and in 1998 the first real-time profile standard, IEEE Std 1003.13-1998 was published, enabling POSIX to address embedded real-time applications and smaller footprint devices.

POSIX and ARINC 653 not only offer different programming interfaces, but also carry different design concepts listed in Table 1.

¹ Previously Developed Software

² POSIX stands for Portable Operation System Interface

³ ARINC 653 is document describing the avionics application software standard interface

⁴ DO-178B is a RTCA document “Software Considerations in Airborne Systems and Equipment Certification”. It provides guidance on designing, specifying, developing, testing, and deploying software in safety-critical avionics systems.

Table 1. Comparing POSIX and ARINC 653

POSIX	ARINC 653
Event driven execution model	Cyclic based execution model
Multi-processing, multi-threading execution model	Same as POSIX but terminology used is partition for process, and process for threads
Priority preemptive scheduling for processes and threads	Within partition time slice : priority preemptive scheduling of A653 processes with deadline management
No temporal partitioning	Temporal partitioning : fixed allocation of time slices to partitions in a repetitive time frame pattern
I/O interrupt driven	I/O polling and I/O partitioning
Memory segregation	Same as POSIX but terminology is spatial partitioning
Socket	Sampling and queuing ports on I/O
Inter Process Communication	Sampling and queuing ports on RAM
Mutex and condition variables	Buffer, blackboard, semaphore, event
Timers	No timers like POSIX but API service to wait for timeout
Signal management	Health monitoring

In ARINC 653, partitioning is a key concept allowing to host multi-criticality level applications⁵ and ensures that whatever the behavior of a partition, it shall not perturb, either directly nor indirectly, the execution of other partitions. This fault isolation mechanism is the basis to build an incremental certification demonstration.

The choice of ARINC 653 has consequences on the development process of applications. Like all newly adopted standards, it lacks tools and interoperability features with existing development platforms (mostly based on UNIX/POSIX). On the contrary, Linux and its conformance to industrial standards, has allowed to built over years, a collection of tools (the GNU development toolset is a revealing example) to ensure interoperability and allow native development models.

3. EMBEDDING LINUX IN AVIONICS EQUIPMENT

Linux has shown over the past 8 years, the ability to adapt to a wide range of architectures, from multi-processor mainframes to embedded platforms, allowing interoperability of applications with the rest of the world. Linux is supported by a large community of developers, testers, and

⁵ An application is made of one or more partitions, executed on the same or different modules.

its success and durability also relies on the adoption of standards. In addition to supporting many hardware platforms, the new 2.6 kernel release offers soft real-time characteristics with good performances.

Recently, the Avionics and Simulation Department of Airbus, has investigated, for a research project, the possibility of switching from a POSIX real-time COTS operating system embedded on a real equipment (with soft real-time constraints), to a standard Linux kernel. The main objectives to achieve were:

- to acquire a good knowledge of kernel internal mechanisms, so as to be able to port it to specific embedded equipments, and adapt/develop drivers for specific avionics I/O devices,
- to limit the effort of porting existing applications to the new platform,
- to draw up an inventory of available documentation from the point of view of the certification process.

The porting of Linux has followed the six main steps described below:

1. adapt low level hardware services to specific hardware components: boot sequence (no BIOS), interrupt controller, real-time clock management, RS232 serial link (for console debugging).
2. downsize the kernel footprint in order to make it compatible with the limited hardware resources of the avionics equipment.
3. build a kernel image (with ram disk and console output capabilities) including minimal root file system with the busybox⁶ tool. Such a configuration provides a standalone machine with limited functionality.
4. include Ethernet driver and TCP/IP stack so that embedded equipment can be seen as a diskless computer. Such a configuration provides distant file system mounting capability, and allow debugging with gdb, automated test suites launching, etc ...
5. adapt/migrate existing hardware specific drivers from the previous COTS operating system to Linux (avionics buses, wired input/outputs, FLASH, EEPROM). This step is achieved by looking for mechanisms in Linux kernel that are similar to the ones used in the COTS kernel (synchronization of tasks, communication between tasks, timeouts management). During this step, a simple FLASH file system was implemented to suppress cache disk latency and allow direct execution of application code from FLASH.

⁶ BusyBox provides a fairly complete UNIX command environment for any small or embedded system. It was first released by LINEO team, but is today maintained by Erik Andersen (www.busybox.net)

6. migrate existing applications. This simply led us to recompile and build applications with GNU tools in the new Linux environment. The code had to be reworked for parts that were not compliant to POSIX API.

The results of this study showed no technical obstacle, or incompatibility with avionics hardware, which would prevent the embedding of the Linux kernel. However, it has been noticed that major subsystems like Memory Manager, Virtual File System, Network, and Inter-Process Communication, turn out to be unnecessarily complex compared to the needs in critical embedded systems. The main problem we identified remains a significant work of reverse-engineering in order to produce the certification material required by DO-178B (coding standards, design documents, traceability, code coverage, ...).

4. FUTURE WORK

Today, Linux offers much more than what embedded avionics applications would need. The kernel uses complex algorithms and dynamic resource allocation mechanisms to optimize system resource usage, whereas critical embedded systems need simple, deterministic behavior. It may require a significant effort to gain a fine understanding of the kernel design, possibly rework part of it, and produce a dependability case that is accepted by the Certification authority.

Future work could then focus on kernel design, reverse-engineering and validation activities (see [1] for a more complete analysis of the use of open source software in critical embedded systems):

- introduce temporal partitioning mechanisms and implement A653 API services.
- include static pre-allocated system resources in the kernel. Resources are set up once at system initialization time and remain valid until module reset.
- increase system modularity by allowing the use of simpler or dedicated software components (memory management, scheduling, file system).
- keep the kernel as close as possible to the Linux current development tree (no forking): only introduce changes that gain acceptance from the Linux community,
- use/develop tools to reverse-engineer the code and build missing certification material (software design, dynamic behavior of system, analysis of partitioning mechanisms, ...).

- apply validation methods like model checking or formal proof of properties on safety related parts.
- characterize the system robustness or fail-silent behavior with fault injection techniques.

REFERENCES

- [1]B. Bérard, P. Coupoux, Y. Crouzet, P. David, Y. Garnier, S. Goiffon, G. Mariano, V. Nicomette, L. Planche, I. Puaut, J-M. Tanneau, and H. Waeselynck, *Logiciel libre et sûreté defonctionnement – cas des systèmes critiques*, P. David and H. Waeselynck (Eds), Hermes Science, ISBN 2-7462-0727-3, 2003 (in French).