USING REAL-TIME LINUX IN SAFETY-CRITICAL APPLICATIONS

Thomas E. Bihari
AMT Systems Engineering, Inc., 1760 Zollinger Road, Columbus, OH 43221
bihari@amtsys.com

Prabha S. Gopinath
TestQuest, Inc., 7566 Market Place Dr., Eden Prairie, MN 55344
prabha.gopinath@testquest.com

Abstract
Real-time Linux variants are being considered for use in safety-critical applications in medicine, aviation, and other areas. The open-source philosophy has much to offer in these applications, but certification concerns impose additional requirements on the verification processes applied to these applications.

1 Introduction

Linux and its various real-time extensions (collectively "Real-Time Linux") are arguably among the more reliable software systems in use today. The open-source philosophy encourages the rapid detection and correction of software defects, and the rapid dissemination of the corrected software to the user community.

Safety-critical application in medicine, aviation, and other areas are increasingly in need of the capabilities, such as networking and graphical interfaces, provided by a full-featured operating system like Real-Time Linux.

However, adoption of Real-Time Linux for safety-critical applications is hampered by the requirements placed on such software by the agencies that certify these products, for example the Federal Aviation Administration and the Food and Drug Administration in the USA.

2 RTCA/DO-178B guidelines

As an example, consider avionics software. In the USA, the FAA advocates the use of the RTCA/DO-178B software development guidelines by developers of avionics software. These guidelines specify processes and data items (e.g., code, documents) related to:

- Software Project Planning
- Software Development (requirements, design, coding, integration)
- Software Verification (review, analysis, testing)
- Software Configuration Management
- Software Quality Assurance
- Liaison with Certification Authorities

The requirements for these processes depends on the failure condition categorization of the system, and the software's potential contribution to system failures. DO-178B defines Software Levels A through E. Level A is assigned to those systems for which a failure of the system would likely be catastrophic. Level E is assigned to non-safety-related systems. Most avionics systems fall into Levels B through D.

While there is some flexibility in the structuring of the data items, DO-178B generally suggests the following "documents" (among others):

- A document containing the system requirements.
- A document containing the software requirements.

- A document containing the software design.
- The source code modules.
- The executable code module(s).
- A document containing the test cases.

The contents of each of these documents must be traceable to related points in the other documents. For example:

- Each software requirement must trace to one or more system requirements, and vice versa.
- Each software design point must trace to one or more software requirements, and vice versa.
- Each test case must trace to one or more software requirements, and vice versa.

3 The DO-178B verification process, in brief

The FAA certifies entire systems (e.g., a navigation system), not individual software subsystems such as an embedded real-time operating system. Therefore, it is not sufficient to verify a real-time operating system entirely as an isolated subsystem.

DO-178B places a strong emphasis on verification. Each data item must be verified via some combination of reviews, analysis, and testing. Any change to one data item requires re-verification of that data item and all other data items that are traceable to/from it. Data items that are human-readable are typically verified by review. Executable code is typically verified by testing. The higher the software Level (E to A), the more rigor and independence (i.e., the person developing the data item cannot verify it) is required in the verification.

Testing of the executable code has the following objectives:

- The executable code complies with the software requirements (i.e., the executable code satisfies each software requirement).
- The executable code is robust with the software requirements (i.e., the executable code detects and handles erroneous situations).
- The executable code is compatible with the target hardware (e.g., the code runs fast enough, fits in the available memory, etc.).

As a verification of the testing itself, the results of the execution of all tests cases is analyzed to verify that the tests exercise *all* of the software (structural coverage analysis). The definition of "all" depends on the software Level.

- For Level A software, each condition of every decision point must be exercised independently ("modified condition / decision coverage").
- For Level B software, each branch of every decision point must be exercised ("decision coverage").
- For Level C software, every statement must be exercised ("statement coverage").

If some portions of the executable code are not exercised during testing, they are analyzed to determine the cause, and then resolved. The possibilities are:

- The requirements-based test cases were incomplete - some requirements were not adequately tested. In this case, better test cases should be developed.
- The requirements were incomplete. In this case, the requirements should be enhanced and test cases added to cover them.
- The code is "dead code" that has no effect and is not needed. It should be removed.

- The code is "deactivated code" that is not needed during normal operation but is useful in other circumstances (e.g., a hardware jumper may activate it during bench testing). In this case, analysis and testing should be used to show that the code cannot be executed in normal operation.

4 Verifying Real-Time Linux

Verification of Real-Time Linux consistent with DO-178B would seem to require a significant effort, and, in its full generality, it would. However, the problem may not be as intractable as it appears.

4.1 Partitioning

DO-178B makes allowances for verifying individual software subsystems within a single system to different Software Levels, if the system is proven to be partitioned into subsystems that isolate faults. Hardware-based memory protection and other isolation techniques may be used to minimize the likelihood that a failure of one software subsystem can trigger a failure of another subsystem. If this can be accomplished, then the system can be partitioned into a (small, if possible) safety-critical subsystem and a (larger) non-critical subsystem. This architecture may be reasonable, for example, for a medical device that controls a potentially dangerous treatment process, while also printing paper reports.

4.2 Real-Time Linux subsets

Many real-time applications require only a minimal real-time kernel. Real-Time Linux is already available in minimal configurations that, for example, fit on a floppy disk. These minimal configurations could be verified with much less effort than would be required for a full configuration.

4.3 Open-source data items

One of the advantages of the open-source philosophy is the opportunity for many users to invest individually moderate amounts of effort, but collectively large amounts of effort.

The standard methods required to perform DO-178B compliant verification are not particularly complex. Relatively straightforward, web-based forms for requirements, tests, etc., could manage the process.

4.4 Embedded support for automated testing

Testing needs to be done, in most cases, for each entire system or product containing Real-Time Linux, with a corresponding variety of software configurations and hardware platforms.

This situation is ideal for the development and use of embedded support for automated testing tools, for testing both Real-Time Linux and the entire system into which it is embedded. Proper use of automated testing drastically improves the speed and accuracy of the testing process. Test cases can be accumulated and re-run for each system under test.

5 Conclusion

The possibility of verifying Real-Time Linux in this way raises several questions:

- *Would verifying Real-Time Linux be worth the effort?* It has been our experience that when we apply rigorous verification to a system as complex as Real-Time Linux, we *always* discover bugs, some serious. Furthermore, the effort required is not considerably more than is typically expended on ad hoc testing. Even if Real-Time Linux were not to be used on safety-critical systems requiring certification, the increased reliability would probably be valuable for commercial products and other widely used systems. The standardization of embedded support for automated testing itself would have a significant pay off.

- *Who would manage and maintain the effort?* This effort would require more structure than is typical for open-source efforts. Is the open-source philosophy amenable to an effort that must be structured?

- *Who would contribute to the effort?* Would the development community contribute to this effort with the same enthusiasm that they contribute to the development of Real-Time Linux?

References

[1] RTCA, *RTCA/DO-178B, Software Considerations in Airborne Systems and Equipment Certification*, 12/1992.