Using an IEC 61508-Certified RTOS Kernel for Safety-Critical Systems

Chris Hobbs, Kernel Developer
QNX Software Systems
chobbs@qnx.com

Abstract
This whitepaper presents the characteristics of a safe kernel, and briefly describes the QNX® Neutrino® RTOS Safe Kernel, which has been certified to conform to IEC 61508 at Safety Integrity Level 3.

The paper then describes some important support requirements for developing applications with safe kernels: a comprehensive tool suite, documentation and training.

Introduction
The QNX® Neutrino® RTOS Safe Kernel has been certified to conform to IEC 61508\(^1\) at Safety Integrity Level 3. This safe kernel provides a certified platform on which application developers can implement systems that must also meet stringent requirements of availability and reliability.

This whitepaper describes:
- the basic concepts behind IEC 61508 as they apply to product certification
- the characteristics needed in a safe kernel
- the support that a developer needs when producing an application to run on a safe kernel

The Concepts Behind IEC 61508
During the 1990s, software became an increasingly important component of many systems whose operation was mission- or safety-critical. The first edition of IEC 61508, which appeared in 2000, was the first major attempt to codify practices and standards associated, not only with the production of such code, but with the resulting product. To this end, IEC 61508 provides recommendations or requirements for:

- The processes to be applied during the development lifecycle of software used in critical applications (e.g., the need for a formalized impact analysis process, requirements traceability). Note that IEC 61508 does not impose a particular design methodology (e.g., a waterfall or agile approach).
- the techniques and tools to be applied to the software (e.g., various forms of (semi-)formal methods, fault correction)
- the target failure measures for systems acting in low demand, high demand or continuous mode. A safe kernel at Safety Integrity Level 3 (SIL3) is required to have a probability of dangerous failure below 1 in 10,000,000 per hour of operation. It might be interesting to note that 10,000,000 hours is 1140 years!

During the first decade of the 21\(^{st}\) century the reliance of critical systems on increasingly complex software continued to grow, and it became clear that IEC 61508 needed to be enhanced to incorporate the new techniques that were emerging. Following work from the national bodies represented on the IEC SC65A, a new version of IEC 61508 emerged in 2010. QNX is represented on the Canadian Advisory Committee for IEC SC65A and has contributed to this new edition.

---
\(^1\) Functional Safety of Electrical, Electronic, Programmable Electronic Safety-Related Systems
Characteristics of a safe kernel

IEC 61508 describes the concept of a Functional Safety Requirement. Taken together, the Functional Safety Requirements for a safe kernel provide the minimum level of operation required to avoid hazardous behavior. This section summarizes some of the types of functional safety requirement required for a safe kernel.

Design Safe State

A safe kernel serves as one component of a system. It is essential that it revert to a safe state if it encounters a situation which it cannot handle. This state must be well-defined so that the system designer can accommodate it in the overall system design and modeling.

In its operation, a kernel is fundamentally an event-handler. Once initialization is complete, the kernel handles events arising from program exceptions, external interrupts, and calls made by application code to invoke its services. There are three circumstances under which it needs to move to its design safe state:

- It would otherwise fail to respond to an event in a timely manner. This is its availability.
- It would otherwise respond to an event incorrectly. This is its reliability.
- It would otherwise respond to an event correctly but corrupt its own (or the application’s) internal state in such a way that future events might not be handled correctly. This is its invariance.

While it is essential that a design safe state be defined, it is also essential that its use should be avoided whenever possible: a safe kernel must be designed to degrade gracefully under conditions such as processor and memory overload, in accordance with the techniques highly recommended at SIL3 in IEC 61508.

A True Microkernel

The QNX Neutrino RTOS Safe Kernel is a true microkernel, implementing as user space applications many of the features normally as an integrated part of the kernel.

The communications stacks, file systems and persistent publish/subscribe infrastructure, for example, are implemented as user-space processes. Therefore, failures in those components do not affect the stability of the kernel itself.

Isolation

Isolation covers areas such as the:

- Isolation provided by the safe kernel between application processes: the behavior of one application process not being able to affect other processes. This includes the provision of mechanisms for interprocess communication and interprocess locking that do not allow one process to affect another inadvertently or deliberately except where both processes have agreed to mutual co-operation.

Figure 1: Functional safety management is the “language” that connects safety-related applications.
This requirement needs hardware support to isolate memory address spaces used by different processes and to co-ordinate activity across multiple CPUs in a multi-processor (SMP) system. In particular, it is particularly important that, when applications interact by means of messages, a message is never simultaneously writable by both the sender and receiver.

- Isolation provided by the safe kernel between application processes and the safe kernel itself: no fault condition in an application is able to affect the stability of the safe kernel.

This isolation requires extensive checking on the part of the kernel of all incoming requests by the kernel. It also requires that some resources (e.g., memory) are assigned for the exclusive use of the safe kernel so that it can continue operation in the event of resource exhaustion in application space.

- Prevention of “leakage” of internal kernel information to applications, which includes, for example, the use of opaque handles rather than genuine memory addresses for responding to system calls.

Note that isolation can be provided much more completely between the elements of a kernel itself if it is implemented as a microkernel with as many as possible of the functions normally associated with an operating system running in application space.

Isolation requirements are particularly important when a system mixes applications having different integrity levels: perhaps uncertified legacy code that has to be run on the same processor as high integrity code.

In this case, it is essential, not only that the applications be isolated from each other as described above, but also that it be possible to reserve resources (processing time, memory, etc.) for the higher integrity code.

### Scheduling predictability

It is essential that a developer writing applications to run on a safe kernel be able to predict the behavior of its schedulers. In particular, it is important that schedulers be available to permit the analysis of the scheduling operations by means of techniques such as Deadline and Rate Monotonic Scheduling.

In a system running code at different integrity levels, the scheduler must also be able to provide guaranteed processor resources to the higher integrity code.

Further, to support the scheduling, developers must be able to associate priorities to threads, and the kernel must support the mechanisms necessary to avoid the so-called priority inversion problem: when a lower-priority task is able to block a higher-priority task.

Another form of unpredictability that can be dangerous in a critical application is when the kernel silently allocates resources in a “lazy” mode: the resources are not really allocated when requested but only when used. In most systems, lazy allocation provides a significant increase in...
Adaptive Partitioning Scheduler

The Adaptive Partitioning Scheduler (APS) that forms part of the QNX Neutrino RTOS Safe Kernel is an example of a scheduler able to guarantee minimum CPU percentages to defined groups of threads. By its adaptive nature, APS eliminates the over-engineering required by fixed-partitioning designs, which waste unused cycles and force designers to use more-expensive CPUs. It thus both improves product time to market and does away with the complex task-starvation problems that typically arise during a project’s integration phase. The QNX Neutrino RTOS Safe Kernel also allows lazy allocations to be disabled, preventing resource exhaustion at a point where it cannot be safely handled.

Protection

Circumstances that were not considered by an application designer may occur in practice. To minimize the danger associated with these situations, a safe kernel needs to provide protection to the applications. Preventing one application from writing into the address space of another (see above) is one example of this protection, but other protection can — and should — be provided. In particular, allocating stacks at the thread rather than the process level, and providing stack guard pages to detect and trap application stack overflow are essential.

RTOS architectures

No discussion of reliable systems can be complete without at least a review of RTOS architectures and their implications for a system’s reliability and ability to recover from faults. This section briefly reviews the three most common RTOS architectures: realtime executive, monolithic, and microkernel.

Realtime executive architecture

The realtime executive model is now 50 years old, yet still forms the basis of many RTOSs. In this model, all software components — kernel, networking stacks, file systems, drivers, applications — run together in a single memory address space.

Figure 3: In a realtime executive, any software module can cause system-wide failure.

While efficient, this architecture has two immediate drawbacks. First, a single pointer error in any module, no matter how trivial, can corrupt memory used by the kernel or any other module, leading to unpredictable behavior or system-wide failure. Second, the system can crash without leaving diagnostic information that could help pinpoint the location of the bug.

Monolithic architecture

Some RTOSs attempt to address the problem of a memory error provoking a system-wide corruption by using a monolithic architecture in which user applications run as memory-protected processes.

This architecture, shown in Figure 4, does protect the kernel from errant user code. However, kernel components still share the same address space as file systems, protocol stacks, and drivers. Consequently, a single programming error in any of those services can cause the entire system to crash.
Figure 4: In a monolithic OS, the kernel is protected from errant user code, but can still be corrupted by faults in any driver, file system, or networking stack.

**Microkernel architecture**

In a microkernel RTOS, applications, device drivers, file systems, and networking stacks all reside outside the kernel in separate address spaces, and are thus isolated from both the kernel and each other. This approach offers superior fault containment: a fault in one component will not bring down the entire system.

Figure 5: In a microkernel OS, memory faults in drivers, protocol stacks, and other services cannot corrupt other processes or the kernel. Moreover, the OS can automatically restart any failed component, without need for a system reboot.

Compared to other kernels, the microkernel provides a dramatically faster Mean Time to Repair (MTTR). Consider what happens if a device driver faults: the OS can terminate the driver, reclaim the resources the driver was using, and then restart the driver, often within a few milliseconds. With

**Development Support**

No kernel — not even a microkernel — is an island unto itself. This truth is especially relevant to a safe kernel, which demands that the applications in which it is used conform exactly to strict rules of design, development and implementation.

In practice, development teams working to bring safe kernel applications to market not only face the standard, competing pressures of time, completeness and quality, but they must also be sure that nothing in their product invalidates the safe kernel certification.

The QNX Momentics Tool Suite

The QNX® Momentics® Tool Suite is a comprehensive, Eclipse-based integrated development environment (IDE) with innovative profiling tools that offer maximum insight into system behavior. It provides productivity and quality analysis tools that accelerate all phases of product delivery.

Figure 6: A screenshot from the QNX Momentics Tool Suite’s Mudflap module.
Documentation

Without complete, accurate and up-to-date documentation, developers can not only waste precious time struggling to understand concepts, but also make fundamental design errors that may put in jeopardy the success of a project.

Training

Architects, designers and developers creating a system that incorporates a safe kernel need to be competent both in the design of safety-related systems and in the manner in which an IEC 61508-certified kernel must be handled to avoid invalidating its certification.

A kernel certified to IEC 61508 should be accompanied by a safety manual that describes the constraints under which the safe kernel must be deployed. An organization looking to have its system certified should have at least its primary developers trained in the interpretation of the safety manual.

The more general training on the development of safety-related applications would normally include many of the architectural, design and implementation techniques specifically recommended in IEC61508. This training would include a general introduction to the quantitative study of software failure, and to the tools and techniques associated with the phases of software development:

- Preventing the introduction of faults into code: the choice of programming language, test-driven development, the role of assertions, etc.
- Preventing faults from producing errors: shallow and deep static code analysis, social interaction graphing, fault injection, program testing, etc.
- Preventing errors from causing failures: coherent exception handling, programming by contract, rejuvenation as a design technique, optimistic and pessimistic replication, recovery blocks, co-ordinated atomic actions, etc.

Conclusion

The software on which mission- and safety-critical systems increasingly rely is inexorably becoming more complex. The techniques that were adequate for developing a few hundred lines of code running as a single-threaded application on a realtime executive on a single CPU are inadequate for modern, multi-threaded applications running in an SMP environment. Architects, designers and implementers need the procedural discipline, the components and the tools to produce these new applications cost-effectively. One essential component is a certified kernel on which to develop the application.

IEC 61508 addresses the software’s development lifecycle and the programming and design techniques used in the software. It also provides numerical requirements on the availability of the resulting software. Certification of a product to SIL3 means demonstrating compliance with all these requirements.

IEC 61508 certification, particularly at SIL3, places significant demands on any software. The demands on software as complex as a realtime kernel are extreme, and can only be met by a kernel with an underlying compliant architecture: compliance is not something that can be retrospectively implemented.

The underlying microkernel architecture of the QNX Neutrino RTOS Safe Kernel has permitted it to receive IEC 61508 certification at SIL3. Its accompanying tool suites make it the correct component for any mission- or safety-critical application.
References


