DOT/FAA/AR-02/118

Office of Aviation Research
Washington, D.C. 20591

Study of Commercial Off-The-Shelf (COTS) Real-Time Operating Systems (RTOS) in Aviation Applications

December 2002

Final Report

U.S. Department of Transportation
Federal Aviation Administration

Technical Report Documentation Page

1. Report No. DOT/FAA/AR-02/118	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle STUDY OF COMMERCIAL OFF-THE-SHELF (COTS) REAL-TIME OPERATING SYSTEMS (RTOS) IN AVIATION APPLICATIONS		5. Report Date December 2002
		6. Performing Organization Code
7. Author(s) Vivek Halwan and Jim Krodel		8. Performing Organization Report No.
9. Performing Organization Name and Address United Technologies Research Center 411 Silver Lane East Hartford, CT 06108		10. Work Unit No. (TRAIS)
		11. Contract or Grant No. DTFA03-01-P-10129
12. Sponsoring Agency Name and Address U.S. Department of Transportation Federal Aviation Administration Office of Aviation Research Washington, DC 20591		13. Type of Report and Period Covered May 2001 – May 2002
		14. Sponsoring Agency Code AIR-100

15. Supplementary Notes

The FAA William J. Hughes Technical Center COTR was Charles Kilgore.

16. Abstract

Commercially available real-time operating systems (RTOS) are seen as candidates for use in airborne-embedded software systems by airframe and engine manufacturers, because of the perceived cost and time savings associated with using commercial off-the-shelf (COTS) components.

Software professionals have pursued the reuse model established in the hardware arena for using COTS hardware components when building a system. Hardware designs can be fabricated from subassemblies and other components. Software designers have not been as effective in establishing their own reuse for COTS software components. Nevertheless, software component reuse is still sought as a means for increasing software development productivity and reducing development costs and schedule times.

This report takes a detailed look into the safety and certification issues of using a COTS RTOS in aviation applications. RTOS attributes are detailed and their safety-related properties are discussed along with considerations to address when integrating a COTS RTOS with an application in an aviation system.

17. Key Words COTS, Software, Commercial Off-The-Shelf, DO-178B, RTOS		18. Distribution Statement This document is available to the public through the National Technical Information Service (NTIS) Springfield, Virginia 22161.	
19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages 42	22. Price

Form DOT F 1700.7 (8-72) Reproduction of completed page authorized

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ACRONYMS

API	Application Programming Interface
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
DMS	Deadline Monotonic Scheduling
FAA	Federal Aviation Administration
FIFO	First-In First-Out
I/O	Input/Output
IMA	Integrated Modular Avionics
ISR	Interrupt Service Routine
MAFALDA	Microkernel Assessment by Fault-injection AnaLysis and Design Aid
MMU	Memory Management Unit
POSIX	Portable Operating Systems Interface
RAM	Random Access Memory
RMA	Rate Monotonic Analysis
RMS	Rate Monotonic Scheduling
RTOS	Real-Time Operating System
SFI	Software Fault Isolation
SSA	System Safety Assessment
SVA	Software Vulnerability Analysis
TCB	Task Control Block

EXECUTIVE SUMMARY

Commercial off-the-shelf (COTS) real-time operating systems (RTOS) provide a variety of services to application software within a system. As RTOS services and capabilities grow in complexity, it is clear that they have an increased influence on the overall system performance and, as such, should have consideration in the overall system safety assessment (SSA). This report addresses some aspects of using COTS RTOS software that may affect safety in aviation systems.

Historically, aviation-based computing systems have used a federated design approach, which can effectively isolate functions with respect to system criticality. However, in more recent years manufacturers are integrating many of these functions into single computing systems with possibly different levels of criticality. RTOSs have become the central computing resource to manage these functions, and for this reason, RTOSs in integrated modular avionics (IMA) require a high level of scrutiny. The RTOS and the associated partitioning, both spatially and temporally, of such IMA systems is important to maintain effective software level separation. The challenge is to design a partitioning solution that enables the exchange of information between partitioned functions and controlled access to other shared resources (such as input and output devices) while keeping the partitioned functions largely autonomous and unaffected by other functions.

In IMA and non-IMA systems, RTOS system performance and associated determinism is key to system safety in higher software levels. Complex central processing units that offer memory caching systems and memory management partitioning need additional RTOS considerations and perhaps alternate RTOS design approaches.

Specified and unspecified RTOS features or elements can pose potential safety hazards. Some of these safety-related considerations with respect to several associated RTOS attributes and features are data consistency, dead code, tasking, scheduling, memory and input and output, and queuing. This list is not all-inclusive, but it does represent a set of potential issues and approaches to consider. It should not be used as a checklist, but rather as a means for understanding some COTS RTOS attributes that could need analysis and verification activities to meet the robustness guidance of DO-178B.

Software level assignments of level A, B, C, or D from the system safety assessment will direct the rigor of analysis of safety vulnerability that the RTOS is asked to tolerate. Some of the elements or attributes will have little or no consideration for level D software; however, as the software level increases, the software vulnerability with respect to safety also increases.

These and other functional class concerns can create a basis for a software vulnerability analysis (SVA). With RTOSs as the center of the computing resource in many aviation systems, it follows that the RTOS should be carefully analyzed for its own vulnerability with respect to safety. From any SSA it is apparent that the RTOS software itself needs to be developed and verified at a software level of safety associated with the system software levels assigned to its applications. But beyond this, the COTS RTOS design and attributes should not compromise safety through its features, application programming interfaces (APIs), and the development environment itself.

The report also makes a recommendation for stress and robustness testing of a typical COTS RTOS.

Several approaches to RTOS safety implications are discussed with a detailed look at testing methods in general and a specific look at wrappers. This report describes techniques that may be used for improving system safety via proper COTS RTOS analysis. It offers several test strategies that may be considered to verify the effectiveness of these safety assurance techniques.

1. SCOPE AND INTENT.

The purpose of this report is to investigate some aspects of using commercial off-the-shelf (COTS) real-time operating system (RTOS) software that may affect safety in aviation systems. There is a trend to use commercially available RTOSs in aviation systems because of perceived cost and timesavings associated with using readily available COTS components. Because of the complexity and unknown integrity of many RTOSs, there are a number of concerns regarding potential aircraft safety effects. The Federal Aviation Administration (FAA) is sponsoring research in the area of COTS software to identify technical and safety issues regarding their use, as well as to identify areas to be addressed in future certification policy and guidance. A previous COTS avionics software study [COTS SW] identified that COTS RTOSs are a potential focus area for COTS in aviation software applications. This follow-on work provides an in-depth study into the considerations of using COTS RTOSs in aviation systems.

COTS RTOS, for the purpose of this study, is meant to be available RTOS software that is sold by vendors through public offerings. The RTOS must be available to the general public and have at least minimal existing distribution. The RTOS may or may not have supporting DO-178B [RTCA SC167] software life-cycle data available. However, it is important to note that the findings of this report can be applied to in-house-developed RTOSs.

2. INTRODUCTION.

The basic focus of the research was to identify what characteristics of the technology can be used to support protection and partitioning rationale such that empirical data obtained can be applied to protection and partitioning approaches proposed by applicants. In particular, real-time operating systems are being designed to provide the protection and partitioning functions for the applications executing on the RTOS and computer hardware. The RTOS is the "glue" between the application, hardware resources, system services, and input/output (I/O) devices. Without an understanding of the underlying concepts, approval of systems with protection and partitioning features of such RTOSs can be difficult and inconsistent.

This research will be accomplished in two phases. This report summarizes the phase that focuses on COTS RTOS characteristics. The next phase will consider architectural strategies for COTS RTOS integration with software applications in aviation systems, with an emphasis on RTOSs and the complex hardware with which they must interface.

3. BACKGROUND.

Airframe and engine manufacturers, in part due to perceived cost and schedule reductions, see commercially available RTOSs as candidates for use in airborne-embedded software systems.

COTS RTOSs have been approved as part of aviation systems that have no safety impact and those with only minor safety impact on aircraft performance and operations. However, once aviation system applicants propose to use them on systems with more severe safety impacts (major, hazardous, severe-major, or catastrophic failure conditions), the considerations of this report are applicable.

Software professionals have pursued the reuse model established in the hardware arena for using COTS hardware components when building a system. Traditional hardware designs can be fabricated from subassemblies and other components. Software designers have not been as effective in establishing a reuse of COTS software components. Nevertheless, software component reuse is still sought as a means for increasing software development productivity and reducing development costs and schedule times.

This report takes a detailed look into the safety and certification issues of using a COTS RTOS in aviation applications with potential safety impacts. RTOS attributes are detailed and their safety-related properties are discussed as well as considerations to address when integrating a COTS RTOS with an airborne software application(s) in an aviation system.

4. ANALYSIS OF SAFETY-RELATED ISSUES IN COTS RTOS.

In real-time systems, correctness of operation depends not only on the right results being generated but also on the results being produced within time constraints. Timing requirements are an integral part of the design and implementation of a real-time system. In real-time systems that employ an RTOS, the correct operation of the system is dependent on the services provided by the RTOS. An RTOS must respond in a predictable way to unpredictable external events. Additionally, an RTOS should have the necessary features to effectively implement the real-time system (i.e., it must be an effective building block for the system) and to support safety features of the system.

This section analyzes concepts related to the use of COTS RTOSs in the context of the aviation systems that may impact aircraft safety. It reviews common functional characteristics that RTOSs should possess when they are used in safety-related systems. In particular, this section summarizes the various time and space (memory) partitioning features and other protection mechanisms used by RTOSs and documents predictability issues related to these features and mechanisms. Further, RTOS features that may be susceptible to failures or that may cause concerns for safety are also analyzed. Lastly, robustness-benchmarking techniques are presented and approaches for handling DO-178B compliance are discussed.

4.1 CHARACTERISTICS OF RTOSs USED IN AVIATION APPLICATIONS.

4.1.1 Predictable Timing.

As RTOS services and capabilities grow in complexity, it is clear that they have an increased influence on the overall system performance and, as such, should have consideration in the overall system safety assessment (SSA). An RTOS typically contains the following features to support deterministic timing [Timmerman 98]:

- Multithreaded and pre-emptible execution.
- Thread priority assignment.
- Predictable thread synchronization mechanisms.

- Priority inheritance mechanism, to prevent *priority inversion*, in which a lower priority task may get executed instead of a higher priority task that is ready [Tindell 00].

- Known and predictable timing behavior (e.g., interrupt latencies, task switch latencies, and driver latencies).

The above features provide the infrastructure needed to support predictable execution times for tasks in a real-time system. In safety-related, real-time systems where a failure can result in a catastrophic, hazardous, or major failure, not only should the RTOS itself be safe but it should also promote the safety of the entire system by providing features that minimize the ways that tasks can adversely affect each other. In regards to safety, a critical feature that an RTOS should have is the support for partitioning of resources, in both space and time, and other protection mechanisms between the multiple applications executing on the same central processing unit (CPU) or sharing common computer resources (such as I/O devices). Partitioning has become increasingly important in the context of aviation systems, because more and more functionality is being consolidated onto single computing platforms.

4.1.2 Federated Versus Integrated Systems.

Aviation systems have traditionally used a federated architecture, in which many distinct computer systems are assigned to distinct control functions in the aircraft, and communicate with each other only using directed or broadcast data buses. These systems are largely de-coupled and only communicate as needed to perform their designated functions. One advantage of the federated architecture is that it provides inherent fault-containment and isolation, since faults cannot easily propagate from functions that are physically located in separate units. However, the federated system approach has its disadvantages in terms of the number of systems and components needed to produce, certify, and maintain. There are often many components of different types, increasing the cost of maintenance and upgrades. The configuration of components can result in an aircraft architecture where it is very difficult to analyze, verify, and validate. It may require constructing a very sophisticated systems integration laboratory and high-fidelity simulation to even approach verification and validation of all the aircraft systems' dependencies and interaction. However, at the system level, a federated system that only performs a limited set of functions can often be more easily verified and validated than a highly complex, highly integrated aircraft system with many functions. The federated approach also creates obstacles for improvements in functional or safety procedures, because adopting any new system-level solutions may require making changes to each of the variously affected subsystems, which is costly. An alternative to a federated architecture is integrated modular avionics (IMA), in which multiple functions, with possibly different levels of criticality, are incorporated on a single physical platform. One researcher [Rushby 99] views that the use of a small number of generic types of components will facilitate the analysis of safety. In federated architectures, there are many subsystems, of different kind, and the possibility that any one can fail is larger than if there are fewer components. Thus, fewer redundant general-purpose units might be better than several specific purpose units.

4.1.3 Partitioning.

A smaller number of computing systems means that larger numbers of functions need to share the same computing resource, which potentially introduces a whole new set of potential failure conditions that the system must address. When executing functions on the same computer hardware, it is necessary to protect functions from having adverse effects on one another. One way to provide this protection is to partition or isolate the functions of the system from one another. Otherwise, one function may interfere with another, causing it to no longer perform its intended function. Also, software functions of different software levels may be executing on the same computer and sharing computer resources. Partitioning and other protection mechanisms are valid means of protecting higher-level software functions from adverse effects of lower-level (less assured) software functions.

Without partitioning, an alternative would be to assign the software level of all functions to the highest level of the system's functions. However, developers typically want to reduce cost and time constraints by developing the less critical functions to a lower level, thereby needing to implement partitioning and other protection mechanisms to allow that level reduction. Even without functions of higher and lower levels, the need for partitioning still exists in some systems, because functions that may independently be safe may not be safe when integrated with other functions sharing computer resources.

The intent of partitioning is to control any additional hazards or failure conditions that may be introduced when multiple aviation functions are sharing the computer processors, memory, and other system resources. Therefore, the software level of the partitioning and protection mechanisms must be at the highest level of classification the functions being protected and potentially higher if the failure of multiple functions introduces more severe failure conditions than the functions by themselves. Partitioning provides fault containment between functions that allows multiple functions to execute on the same computer and in the same system. This can facilitate safety analysis and increase safety assurance, if properly implemented and verified. Ideally, partitioning and other protection mechanisms should produce a virtual impression that each function has control over its own computing system and system resources. Protection schemes should address both the space (memory) and time (CPU throughput) domains, which are described in the following subsections.[1]

4.1.3.1 Spatial Partitioning.

Spatial partitioning seeks to prevent a function in one partition from overwriting or corrupting the data space (i.e., memory) of a function in another partition. Memory protection can be achieved by two mechanisms.

- First, the hardware-based method, which consists of using a memory management unit (MMU), can be used to perform checks whenever memory is being accessed in order to prevent unauthorized access to certain memory locations. Many times, MMUs are COTS

[1] The terms partitioning and protection are often used interchangeably in the aviation community. In short, partitioning is a method of protection.

hardware devices that are provided with the microprocessor. The RTOS kernel can control portions of the MMU. Hardware-based spatial partitioning is the most prevalent form of spatial partitioning. It has a one-time cost of designing, implementing, and certifying the partitioning mechanism of the kernel and its supporting hardware. The MMUs available in most commercial processors today are overly complex and raise certification concerns.

- A second choice would be to use software fault isolation (SFI), which consists of adding logical checks in the code at each memory access point. By examining the machine code of the software in a partition, it is possible to determine the destinations of some memory references and to statically check whether they are accurate. Indirect memory references cannot be checked statically, so instructions are added to the program to check the contents of the address register at runtime, immediately prior to its use. The SFI technique imposes some overhead cost by adding code to the program. It also requires an additional analysis and certification cost on every project. However, it is possible to automate much of the check procedure and to qualify a tool or toolset that can be used on multiple projects.

A related concern in spatial partitioning is to save the status information of a task prior to switching execution to another task. The RTOS usually saves registers in the stack whenever it performs a context switch, potentially mixing data of different tasks of perhaps different assurance levels on the stack. However, it is important to save all registers, without ignoring the less frequently referenced ones. The content of any register that has not been saved may get modified by another task, which may constitute a failure condition by corrupting or overwriting the stack data of another task.

4.1.3.2 Temporal Partitioning.

Temporal partitioning ensures that each function has sufficient processing time to complete its operation. Temporal partitioning is closely related to schedulability of tasks in a multitasking real-time system; hence, both temporal partitioning and multitasking scheduling have similar challenges. The techniques used for scheduling tasks in real-time systems can thus be used to enforce temporal partitioning. However, in the case of safety-critical systems, there is added emphasis on using proper schedule enforcement mechanisms in order to prevent a task from overrunning its schedule, monopolizing the CPU, crashing the system, or comparable problems.

Task scheduling techniques can be subdivided into two classes—static and dynamic.

- With static scheduling, the list of tasks is executed under a fixed cyclic schedule and each task receives a fixed slice of execution time in a cycle. The sequence of the execution of these tasks is decided at design time, based on the constraint of satisfying all task deadlines, and it is not flexible. One advantage of static scheduling is that it is very easy to prove that task deadlines will be met with the proposed schedule. However, static scheduling techniques can have potentially long response times to external events, such as interrupts, which can only be serviced when the corresponding interrupt service routine is scheduled to run. In addition, tasks are scheduled for their entire duration (period), even if they have nothing or very little to process.

- With dynamic scheduling, there is no predefined schedule, but priorities are assigned to tasks at design time so that a higher priority task may pre-empt a lower priority task that is executing. At any given time, the highest priority task that is ready to execute is allowed to execute. Under certain circumstances, priority assignment techniques such as Rate-Monotonic Scheduling (RMS) and Deadline-Monotonic Scheduling (DMS) [Liu 73, Tindell 00] can be shown to guarantee that tasks meet their deadlines under all circumstances. RMS assigns priorities to tasks monotonically, based on the period of the task—the smaller the period, the higher the priority of the task. RMS assumes that task deadline equals task period. DMS is a generalization of RMS, with the assumption of task deadline being less than or equal to its period. Priorities are assigned monotonically, based on the deadline of the task—the shorter the deadline, the higher the priority.

Hardware interrupt timers, or watchdog timers, are sometimes under the control of the RTOS kernel and may be used as a mechanism to prevent a task from overrunning its schedule.

4.1.3.3 Interpartition Communication.

If the goal of partitioning were to simply keep one partitioned function isolated from another, then it would be a relatively straightforward problem to address. However, in reality, partitioned functions may need to communicate, and they may require access to other shared resources (such as I/O devices, queues, and buffers) at the same time. Hence, the challenge is to design a partitioning solution that enables the exchange of information between partitioned functions (e.g., interpartition communication) and controls access to other shared resources (such as I/O devices) while keeping the partitioned functions largely autonomous and unaffected by other functions. Interpartition communication and sharing of I/O devices influences both the space and time aspects of partitioning and protection mechanisms.

In the case of interpartition communication, the space can be partitioned by using the kernel as a trusted intermediary between partitioned functions, where the kernel copies data from one memory space to another. Another way is to reserve separate memory space for communication between each pair of partitioned functions. For communication between partitioned functions and devices, there are several approaches for space partitioning. For processors with memory-mapped I/O, in which a device is only accessed by one partitioned function, the device may be mapped into the memory space of that particular partition and mechanisms such as MMU and SFI can be used. If access to a device is shared between partitioned functions, special device management partitions can be implemented that control several devices and are trusted to keep them separate.

Interpartition communication and communication with I/O devices can also affect the time-partitioning model in use. For instance, if a servicing partition fails to respond, another partition might wait indefinitely for the data or service from the servicing partition. Therefore, the RTOS should provide alternative mechanisms to prevent this from happening. Communication between partitioned functions and devices should also be managed by the RTOS (e.g., using device management partitions), because a partitioned function may hold a device indefinitely. The use of other cross-partition interference, such as locks and semaphores, should also be limited.

4.1.3.4 Cache Memory and Partitioning.

An area that deserves particular attention is the use of cache memory in a partitioned computer platform environment. A cache is typically small-size, high-speed memory, or a hierarchical set of memory, that resides between the CPU and the main memory. A cache is an intermediate memory storage location, which has rapid access times. The role of the cache is to match the fast speed of the processor to the slower speed of the main memory. Cache memory contains a copy of the most frequently accessed memory locations, which can reduce the overall memory access time. The use of cache can lead to nondeterministic execution time for functions, depending on how much of the data needed by the function is available in cache. This behavior may be further aggravated by the fact that cache is typically a shared resource among partitioned functions, which may lead to cross-interference among partitioned functions in the time domain, and violate the partitioning protection. Depending on the state in which the cache memory is left by a function in a partition, the execution time of the next function scheduled to execute may vary. Even though the execution time of a function is nondeterministic due to cache, it is still bounded by the worst case of having all accesses directly to and from main memory. Since worst-case analysis is crucial in safety-critical, real-time systems, timing analysis and scheduling to tasks should address protection of the partitioned functions considering the presence and use of cache memory. Interference of cache in the spatial domain can be controlled using memory protection mechanisms such as MMU and SFI. However, the use of cache introduces an additional concern of maintaining cache coherency. Cache coherency keeps consistent multiple copies of a single variable, so that the cache swapping is valid at all times. Changing a datum only in cache or main memory, without reflecting it in its copied version, may result in inconsistent or erroneous behavior. Techniques for preserving cache coherency should be verified, and the overhead of additional timing or interactions with the CPU should be accurately analyzed and addressed in worst-case scenarios.

4.1.3.5 Additional Partitioning Considerations.

Static scheduling of tasks may not be appropriate if partitioned functions frequently need to wait for data from other partitioned functions or devices, since the function that is waiting for service may waste CPU time. Also, with static scheduling, if a critical interrupt is sent to a partitioned function that is not actively running, servicing of that interrupt will have to wait until that partition function is scheduled, thus response time may suffer. In such cases, dynamic scheduling may be more flexible, although more difficult to verify. However, one should address how other partitioned functions may interfere with timing of a partition, and also address the worst-case overhead of context switching by the kernel to ensure performance and timing requirements can be met. For instance, a faulty partitioned function may repeatedly issue a request for CPU time, which may produce significant overhead and interfere with the execution time of a given function. In this case, it is important to establish hard maximum quotas of time allocation for a partitioned function, and quota for interrupts and invocation of kernel functions can be deducted from the quota of the issuing partitioned function.

Regardless of the mechanism chosen for partitioning, the operating system is an essential component in the implementation of the mechanism. In a traditional operating system, all operating system services may be accessible to all applications, which make the safety analysis of the operating system more difficult. Rushby deals with the complexities of partitioning in space and time and other protection mechanisms by proposing alternative operating system architecture [Rushby 99]. The proposed alternative suggests an arrangement that allocates operating system services separately within each partition. Critical applications may use a minimal set of services, whose robustness may be easier to verify, while less critical applications may employ something closer to full-fledged commercial operating system services. In other words, operating system services may be allocated within the confines of certain partitions, leaving mainly the kernel as the only common resource between partitions.

4.2 RTOS SPECIFIC FAILURES WITH POTENTIAL SAFETY IMPACT.

This section analyzes aspects of an RTOS that may be the most susceptible to failures or that may cause concerns for safety. When developing an RTOS for the aviation domain, RTOS developers or users should document any failure or safety concerns, the severity, and their approach for addressing the problem. For example, one safety requirement is to prevent run-time errors that could compromise the continued safe operation of the system. Some of the potential failure conditions associated with this requirement are erroneous data, improper implementation of the RTOS requirements specification, and incorrect calculations or operations performed by the RTOS.

Each of these failure conditions can be further decomposed into areas of concern with respect to software vulnerability based on the RTOS function used. A software vulnerability analysis (SVA) can identify areas of potential anomalies, which can be provided as input not only to a robustness or stress-test plan, but also to a system functional hazard analysis or SSA. How an SVA is conducted is up to the RTOS developer or applicant, but table 1 identifies the areas of concern with regards to RTOSs and can be used as a basis for establishment of an SVA.

It is not possible to perform a vulnerability analysis without referring to a specific RTOS implementation. Two RTOSs that offer the same feature may implement the feature in vastly different manners, and the vulnerability analysis results will depend on the implementation of the feature.

A review of a recent paper [Kleindermacher 02], along with a study of a representative example of RTOSs, reveals the following areas of concern, as listed in table 1. The table does not represent an exhaustive list of RTOS concerns with respect to an SVA, and for any given RTOS, the areas of concern will differ.

TABLE 1. RTOS AREAS OF CONCERN BY FUNCTIONAL CLASS

Number	Functional Class	Concern	Description
D1	Data consistency	Data corruption or loss within the RTOS by the RTOS itself	Data, which is visible to the RTOS, is corrupted or “lost” by the RTOS.
D2	Data consistency	Input data corruption or loss by the RTOS	The RTOS incorrectly handles input data or loses it by storing it incorrectly, or incorrect data values are assigned to data variables or returned as results.
D3	Data consistency	Erroneous data or results caused by incorrect calculations or operations by the RTOS	Incorrect data values assigned to data variables or returned as results.
D4	Data consistency	Abnormal parameters	Calculations performed by the math library functions may return unpredictable small numbers if the values passed as parameters are abnormal.
C1	Inclusion of deactivated code or dead code	Inclusion of deactivated code	Unused functions may be loaded with the application even though they are never called. This activity can also be dependent on a linker or loader that is used to link the executable code into the executable image and/or load the image into the target computer memory. Unintended activation of this code may have unknown effects, typically leading to system failure.
C2	Inclusion of deactivated code or dead code	Generation of dead code	Additional software is generated by the compiler or linker, which is not verified during requirements-based testing or coverage analyses. This is especially a concern for Level A applications where the applicant needs to “account” for executable object code that is not traceable to source code; it can result in dead code, and compiler generated code can result in code that is not exercised during requirements-based test, nor is it included in structural coverage analysis which is typically performed at the source code level. Compiler- or linker-generated object code is not exempt from satisfying these objectives for compliance to requirements and robustness for Levels A-D and for low-level requirements for Levels A-C.
T1	Tasking	Task terminates or is deleted	The task runs to completion or is deleted by another task. If the programming model requires a task to run forever, in a never-ending loop, then the API call to delete the task should be removed.
T2	Tasking	Kernel’s storage area overflow	A central storage area in the kernel, which holds task control blocks and other kernel objects, may run out of space due to a malicious task that constantly allocates new kernel objects that may, in turn, affect execution of other tasks. A quota system should be implemented to protect other tasks in the system.

TABLE 1. RTOS AREAS OF CONCERN BY FUNCTIONAL CLASS (Continued)

Number	Functional Class	Concern	Description
T3	Tasking	Task stack size is exceeded	The task stack is overwritten leading to unpredictable system behavior and stack data corruption.
S1	Scheduling	Corrupted task control blocks (TCB)	TCB's may be corrupted, which compromises the scheduling operations of an RTOS. Scheduling information data should be protected from access from user software applications.
S2	Scheduling	Excessive task blocking through priority inversion	A user task of high priority may be excessively blocked by a low-priority task because they share a common resource and an intermediate task pre-empts the low-priority task.
S3	Scheduling	Deadlock	If two tasks both require the same two resources but they are scheduled in an incorrect sequence, then they may cause a deadlock by blocking each other.
S4	Scheduling	Tasks spawns additional tasks that starve CPU resources	New tasks spawned by an existing task may affect the schedulability of all tasks in the system. User applications should not be allowed to spawn new tasks at their own will.
S5	Scheduling	Corruption in task priority assignment	Increasing or decreasing the priorities of tasks in the system may lead to the task set not being schedulable or the system not responding in a timely manner. The ability to change the priority of a task should be limited to special cases, such as to prevent the occurrence of priority inversion.
S6	Scheduling	Service calls with unbounded execution times	Schedulability of tasks is impacted if there are kernel service calls that have unbounded execution time. The execution time of a task that makes such service calls may itself be affected, as well as accounting for the kernel's overhead while switching between tasks. Kernel service calls should have bounded execution time regardless of system load conditions.
M1	Memory and I/O device access	Fragmentation of heap memory space	Allocation, de-allocation, and the release of memory from the heap may lead to fragments of free memory, which complicates future allocations and may compromise timing analysis, making it unpredictable. Dynamic memory allocation, de-allocation, and "garbage collection" should be very limited and controlled.
M2	Memory and I/O device access	An incorrect pointer referencing/de-referencing	An incorrect reference to an object, such as a semaphore, may be passed to the kernel via a service call, which can have disastrous results. The kernel should check validity of pointer references.
M3	Memory and I/O device access	Data overwrite	Data is written beyond its allocated boundaries and overwrites and corrupts adjacent data of other functions in memory.

TABLE 1. RTOS AREAS OF CONCERN BY FUNCTIONAL CLASS (Continued)

Number	Functional Class	Concern	Description
M4	Memory and I/O device access	Compromised cache coherency	Increased access time occurs due to cache misses. This occurs when needed data is not available in cache and data must be accessed from other typically slower memory. Data loss due to missed memory updates.
M5	Memory and I/O device access	Memory may be locked or unavailable	The MMU page tables may be incorrectly configured or corrupted such that access to a region of memory is prevented.
M6	Memory and I/O device access	Unauthorized access to critical system devices	Unauthorized access to I/O devices may lead to improper functioning of the system. The kernel must implement mandatory access control to all critical devices.
M7	Memory and I/O device access	Resources not monitored	Proper allocations and usage of resources are to be monitored, otherwise resource could be deadlocked
Q1	Queuing	Task queue overflow	May experience loss of information or change in scheduler performance. May result in missed schedule deadlines and incorrect task sequencing.
Q2	Queuing	Message queue overflow	Messages may be missed, lost, or delayed if the queue is not properly sized or messages are not consumed promptly unless this is protected.
Q3	Queuing	Kernel work queue overflow	The work queue is used to queue kernel work that must be deferred because the kernel is already engaged by another request and the queue is full. Kernel work deferred to the work queue must originate from an interrupt service routine. The work queue may overflow if the interrupt rate is too high for the kernel to process tasks within the allotted time frame.
I1	Interrupts and Exceptions	Interrupts during atomic operations, such as task switching	Certain operations that work on global data must complete before subsequent operations can be invoked by another task of execution. An interrupt arriving during this period may cause operations that modify or use a partially modified structure, or the interrupt may be lost if interrupts are masked during critical code execution.
I2	Interrupts and Exceptions	No interrupt handler	No interrupt handler has been defined for an interrupt. A default interrupt handler should be provided by the RTOS if the user has specified none.
I3	Interrupts and Exceptions	No exception handler	No exception handler has been defined for an exception raised by a task. A default exception handler should be provided to suspend the task and save the state of the task at the point of exception.
I4	Interrupts and Exceptions	Signal is raised without a corresponding handler	A signal may be sent by a task to another task or by the hardware under defined exception conditions.
I5	Interrupts and Exceptions	Improper protection of supervisor task	Supervisor task that is invoked, due to an exception, runs in an unprotected address space that may be corrupted.

4.3 ROBUSTNESS BENCHMARKING TECHNIQUES.

Various methods have been developed over the years to assess the robustness of COTS operating systems. The following description summarizes some of the methods.

- An early method, called CRASHME [Carrette], operates by writing random data values to memory. Several tasks are then spawned to execute those random bytes as concurrent programs. The method relies on pure chance of the execution of the tasks with random data causing the system to crash.

- Similarly, the Fuzz approach [Miller 90, Miller 98] also relies on random data injection, but it tests specific operating system elements and interfaces (as opposed to the completely random approach of CRASHME). Fuzz compares the quality of open-source operating systems versus commercial operating systems. The results concluded that open-source operating systems were more robust than commercial ones, based on robustness measures.

- The Ballista work [Koopman 00] is similar to Fuzz, except that operating system function calls are used instead of application level software, as well as combinations of valid and invalid data. The Ballista robustness testing system tests the exception handling capabilities of application programming interfaces (APIs) of portable operating systems interface (POSIX) -compliant operating systems. The basic idea in Ballista is to focus on the data types of the system calls and not the actual calls. This makes the definition of the test cases to be carried out very simply. For each data type some test values are defined, representing common values as well as boundary values. For every system call, all the combinations of the test values are used to produce the test cases. Each test case is executed one at a time and after the execution the result is interpreted according to the CRASH (catastrophic, restart, abort, silent, hindering) severity scale. Tests using Ballista were conducted on 15 POSIX operating system versions and identified many instances of exceptional conditions being handled in a nonrobust manner; some leading to complete system crashes.

- Another tool, called MAFALDA (Microkernel Assessment by Fault-injection AnaLysis and Design Aid) [Fabre 00] gathers information on the failure modes of microkernels and helps to integrate them into safety-critical systems using wrappers. MAFALDA classifies the failure modes of the microkernel by using both classical software fault injection and parameter fault injection, like Ballista.

To date, no evidence has been shown that these have been used for certification activities in civil aviation applications. However, some of these approaches could possibly be used to help augment robustness testing.

5. STUDY OF RTOS SAFETY ASSURANCE TECHNIQUES.

DO-178B section 6.4.2.2 requires robustness testing for Levels A, B, C, and D software. As such, to meet the robustness requirement for a highly integrated and complex RTOS, an SVA could be conducted and corresponding robustness and stress testing could be developed to meet the robustness guidance in DO-178B. This is particularly significant for COTS RTOSs since aspects of the design and verification data for a COTS RTOS may not be available.

An RTOS SVA is not a requirement per DO-178B, and one could offer that any application using a COTS RTOS that meets the objectives of DO-178B clearly is in compliance. The basis for recommending a COTS RTOS vulnerability analysis is that the highly integrated nature of today's COTS RTOSs coupled with complex microprocessor architectures of modern processing systems may well indeed influence the overall system safety, particularly in the time, space, and resource domains.

The following sections (1) describe the techniques that may be used for improving system safety of COTS RTOS, (2) describe the test strategies that may be considered to verify the effectiveness of these safety assurance techniques, and (3) investigate the fault-containment techniques to protect against the effects of unintentional functions and failures in an RTOS.

5.1 TECHNIQUES FOR IMPROVING SYSTEM SAFETY AND PROTECTION FROM FAILURES IN AN RTOS.

Several techniques exist to help system safety with respect to COTS RTOS-based products. Some techniques are used by the end user of the RTOS (i.e., the system integrator), while other techniques would be implemented as part of the COTS RTOS development. The techniques can be grouped into the following general categories, based on the how they seek to improve system safety:

- Prevent the presence of defects in the RTOS (i.e., fault avoidance), which can be accomplished by proper design assurance.
- Analyze and test the COTS RTOS and remove any defects if present.
- Protect against remaining defects in the COTS using wrappers or other similar techniques.

A combination of techniques is often employed for increased safety assurance of the end product. The first group of techniques (fault avoidance) usually depends on the efforts of the COTS developer. Stringent development practices and adherence to software development guidelines and standards (drafted by various safety-related industries) can help to prevent defects and obtain acceptance of the COTS product in a particular target domain. Extensive studies in the field of software engineering have focused on software development processes, and in the case of safety-critical systems, the most important consideration is to incorporate safety as an integral part of the design and development process; i.e., it should not be an afterthought. In the end, in the case of the safety-critical applications, the applicant must be able to present sufficient evidence (or provide supporting material to help the system and software developer and COTS user present the evidence) of compliance to a particular standard for the end-user application domain.

5.2 TEST METHODS.

Some design and verification techniques used on the COTS RTOS are key to its relative system safety affects. Access to COTS development data is essential to properly assess the development and verification activities especially for higher software levels; unfortunately, many COTS products lack this data. However, a survey conducted for this research project revealed that aviation systems manufacturers striving to develop the most critical systems (i.e., Levels A and B) are seeking vendors who have considered DO-178B guidance in their COTS RTOS development. As such, the DO-178B software life-cycle data can be made available to the user as part of the service agreement. Independent and supplemental testing is still required and a variety of techniques can be used.

Methods used to test software in general also apply for testing of COTS RTOS. The purpose of testing is to detect faults in the component under test, i.e., to identify discrepancies between the specifications of the RTOS and its actual behavior. The vendor must perform the necessary component testing of the RTOS to verify its compliance to all the requirements specifications, which includes those related to safety. The vendor should also test the RTOS with representative software applications executing on the RTOS, and to demonstrate RTOS robustness, should execute rogue application testing where the rogue application(s) attempts to violate the partitioning and other protection mechanisms offered by the RTOS. The certification applicant using the COTS RTOS must ensure that the RTOS is properly integrated into the final product, such that appropriate software integration testing, hardware-software integration testing, and system-level testing are performed.

Testing of an RTOS consists of subjecting the RTOS to a variety of test cases. Clearly no set of test procedures can achieve 100% test coverage in a practical amount of time. Combinatorial explosion is a term often used to refer to the unbounded increase in the number of test cases that results from choosing different combinations of input values for each test case. Even for moderately complex software, the number of test cases required to adequately test the RTOS may be impractical. Test cases are thus chosen in a manner such that test cases exercise different aspects of the RTOS to maximize the test coverage. Defects found during testing can be fixed by the user, if source code is available, or with support from the vendor. Alternatively, test results can provide information that can be used to appropriately design wrappers or other schemes for shielding the system and software applications from the RTOS defects. Testing of an RTOS can be subdivided into the following subcategories, based on how test cases are selected:

- Stand-alone testing, including:

  - Requirements based (black box)
  - Structural based (white box)
  - Random testing (black box)
  - Error seeding or fault injection based (white box)
  - Equivalence class and boundary testing

- Testing of an RTOS in a target system with actual or representative software applications integrated, comprised of:
  - Requirements-based testing (functions as intended)
  - Robustness or stress testing (has no unintended functions, side effects, or anomalous behavior)

Some of the testing methods require knowledge of the source code (white-box testing), while others make no assumption about the inner workings of the unit under test (black-box testing), which is an important consideration in the case of COTS. Each of these techniques is discussed in detail below.

5.2.1 Stand-Alone Testing of an RTOS.

Stand-alone testing of the RTOS is equivalent to software component testing; i.e., the RTOS is tested in isolation from other components of the target system. Conceptually, both requirements- and structural-based approaches subdivide the input domain into a number of logical subsets, according to some criteria, and select a number of elements from each subdomain as test cases. The criterion employed for subdividing the input domain distinguishes the two approaches, which are further described below.

5.2.1.1 Requirements-Based Testing.

In requirements-based approaches, input data are selected from partitioned sets that effectively test the functionality specified in the requirements specification of the RTOS. Hence, partitioning of tests is based on selecting inputs that invoke a particular aspect of the RTOS's functionality. Testing involves the observation of the output states, given the inputs, and thus, no analysis of the internal structure of the RTOS is attempted. Therefore, requirements-based test approaches are a type of black-box testing. Requirements-based approaches can use the equivalence class techniques discussed below.

An example of requirements-based testing of RTOSs is the Ballista project [Koopman 00], which tests only the exception-handling capabilities in a POSIX-compliant operating system. In the Ballista project, subdividing the input domain into subdomains, with each subdomain containing one API function call, derives test cases. A test case consists of the name of the function call and a single-test tuple that are passed as parameters. The general Ballista approach is to not only test the requirements but also test the robustness of a single call for a single-test tuple, and then iterate this process for multiple test cases such that each has different combinations of valid and invalid test values. The Ballista testing methodology involves automatically generating sets of exceptional parameter values to be used in calling software applications. Tests performed on over 15 operating systems concluded that none of the RTOSs tested displayed a high level of robustness.

5.2.1.2 Structure-Based Testing.

Structure-based testing approaches are a form of white-box testing. The basis for the subdivision of the domain is not the functional specification of the system or what the system should do, but what the underlying code and structure of the RTOS is itself. Structure-based approaches devise subdomain partitions that attempt to provide coverage by exercising necessary elements of the code that constitute the RTOS. Under the guidance of DO-178B, structural verification is conducted on software Levels A, B, and C. The approach encouraged by DO-178B is to conduct a structural coverage analysis based on the systems functional requirements already tested. These elements for analysis relate to structural elements and architecture of the program such as statements, edges, paths, or the data flow characteristics of the program. A basic requirement is for each program element to be executed at least once, which results in complete structural coverage and associated analysis per DO-178B. It is important to note that structure-based testing alone does very little towards meeting DO-178B and that it is really the analysis that is the mechanism for discovering structural deficiencies. The discoveries during this analysis and testing also include revealing dead code and improper use of deactivated code. The structural coverage analysis can also point out shortcomings in the requirements-based test cases or procedures. With respect to the guidance of DO-178B, what this implies for software Levels A, B, or C is that the COTS RTOS must either be accompanied by a structural coverage analysis or the applicants must conduct the analysis by themselves. In the latter case, availability of the COTS RTOS source code is required, which is not always possible with COTS RTOS.

Particularly acute problems that can occur with structure-based approaches are that there is no guarantee of coverage unless the structural coverage analysis is conducted. It also suffers from combinatorial test case explosion if the code lacks effective design structure. Further discussion regarding structural coverage analysis is available in [RTCA SC190] Frequently Asked Questions 42, 43, 44, and Discussion Paper 3 of DO-248.

5.2.1.3 Random and Statistical-Based Testing.

Statistical-based techniques rely on the assumption of random and statistical nature of defects to exercise a program with the aim of causing it to fail. Two approaches that fall under this category are random testing and error seeding (a.k.a., fault injection-based testing). (Error seeding is discussed in more detail in section 5.2.1.4.) This type of testing can be correlated to robustness testing with respect to DO-178B; however, for higher level of criticalities, it lacks a completeness and coverage argument.

Random testing is a black-box testing approach in which test cases are derived at random. This avoids possible bias in only testing known features of the software. Random testing is simple and quite easy to automate. One problem with random testing is that it is possible that, at the end, only a section of the software may have been tested. In addition, random testing is not repeatable, so it is hard to take credit for certification purposes. Random testing is, thus, best used to complement the other testing techniques described in this report. The CRASHME approach is a representative example of random testing of robustness of an RTOS.

5.2.1.4 Error Seeding or Fault Injection-Based Testing.

Error seeding is a unique technique for testing in which some carefully devised known defects are injected (seeded) into the code that is to be tested. Test cases are then applied to the program containing the known defects and possibly a number of unknown defects. Assuming that the seeded defects are typical defects, it can be argued that the ratio of the known defects found during testing and the total of the known defects is the same as the ratio of the unknown defects found and the total of the unknown defects. This allows for statistically estimation of the number of remaining defects in a program. This is a way of building confidence that the test cases used are somehow valid in uncovering the various types of defects. Unfortunately, there is no guarantee that all defects will be revealed. The effectiveness of this technique is heavily dependent on the knowledge of the types of defects in the system and of the test cases that can uncover them. In addition, this technique is a type of white-box testing that requires availability of source code, which may not be available with COTS components.

5.2.1.5 Equivalence Class and Boundary Testing.

With equivalence class techniques, each input condition is partitioned into sets of valid and invalid classes called equivalence classes. These are, in turn, used to generate test cases by selecting representative values of valid and invalid elements from each class. In this approach, one can reasonably assume (but not with 100% certainty) that a test of a representative value of each class is equivalent to a test of any other value. Testing of boundary values can also be conducted at this level.

The requirements-based approach of dividing the test domain into equivalence classes is, in general, ineffective in testing combinations of input circumstances. However, most hard to detect faults are due to a combination of, or a sequence of, inputs to the system. Another major issue is that of the rapid proliferation of test cases needed for adequate coverage (combinatorial explosion). Another difficulty is in determining the granularity level of the equivalence classes. In most typical situations, it is hard to predict the correct level of granularity of the equivalence classes prior to testing. Requirements-based testing is also unable to find nonfunctional failures (e.g., task schedule or task queue overruns, failure from presence of dead code, etc.).

5.2.2 Testing of an RTOS in a Target System.

The RTOS must be tested in the target environment, integrated with the actual or representative software applications that will run in the target system. This testing is referred to as requirements-based integration testing and requirements-based hardware software integration testing in which applications are gradually integrated and tested in the RTOS platform. It also provides for the inclusion of all the applications in the target system (system testing) and culminates with more rigorous, robustness testing where rogue applications may attempt to violate the RTOS protection mechanisms and invalid data and worst-case interrupts and events are introduced to determine how robust the system is. The process of incrementally adding applications (modules and partitioned functions) makes it easier to detect faults during integration testing. System and robustness testing are still needed after integration testing because, while the interaction between all different modules may have been tested during integration testing, task loading and contention of RTOS resources may not have been effectively tested.

In general, stand-alone testing of the RTOS is likely to cover the overall functioning of the operating system in more breadth and probably more rigorously check each functional feature. On the other hand, integration and system testing using other applications, tests the RTOS in the confines of the target environment. Integration testing on a target system may only test a narrower portion of the system in more depth, but it is likely to exercise a greater number of test cases in the subset of the code that is going to be more heavily used. While unit testing focuses mainly on the internal properties of the component being tested, integration and system testing can uncover inconsistencies in the interaction among the units. Performing integration and system testing only, without stand-alone testing of the RTOS, may leave untested unused code in the RTOS, which may lead to potential safety hazards during operation.

System-level testing is subdivided into different classes of testing, depending on which aspect of the system needs to be verified, such as functional (requirements-based) testing, robustness testing, and stress testing.

- Functional testing uncovers the differences between functional requirements and functional behavior of the system and demonstrates that the system satisfies "functional" requirements.

- Robustness testing extends the boundaries set by functional tests, by subjecting the system to unconventional conditions, such as various fault scenarios and invalid inputs, in order to try to crash the system. The motivation behind applying robustness testing comes from an observation that most system failures occur during unusual scenarios that are easily overlooked, or hard to conceive, during unit testing.

- Stress testing subjects the system and software to the extremes of real-time workloads, large data volumes, repetitive operations, and operations for extended periods of time. The purpose of stress testing, also called load testing, is to measure characteristics such as response time and memory utilization under data and transaction loads, which is particularly crucial in the case of RTOS. For instance, the task switching time in an RTOS may be dependent of the number of tasks in the ready queue. Thus, the worst-case condition needs to be evaluated. Benchmarking techniques can be very helpful when trying to measure key performance criteria of an RTOS. Important features to be measured are task switching time, pre-emption time, interrupt latency, semaphore shuffling time, deadlock breaking time, memory allocation/deallocation time, and message passing time. Adequate instrumentation to perform the measurements is an issue in this type of testing.

5.3 COTS FAULT-CONTAINMENT TECHNIQUES.

A variety of fault-containment techniques can be used; this report focuses only on those techniques for COTS RTOS software. Fault-containment techniques complement the efforts of fault prevention and fault-testing mechanisms that were described above. Fault-containment techniques are needed when faults cannot be removed or, as a precautionary measure, when one is not sure whether the component can be considered fault-free. Fault-containment strategies are particularly useful when dealing with COTS software, since source code may not be available, and the only alternative may be to isolate, or contain, specific faulty behaviors. The approach, therefore, is to use untrustworthy COTS software in a way to provide assurance by other means that the COTS software will not impact the safety, functionality, and performance of the entire system.

The partitioning methods discussed in section 4.1.3 can essentially be viewed as fault-containment techniques for application-level faults (not the RTOS level). Partitioning is a system approach for fault containment of application errors, which requires RTOS-level support. Similarly, other software and hardware features that can contribute to the process of fault containment are value monitors and timing monitors [Jaffe 00]. Value monitors correspond to additional logic blocks whose purpose is to check the validity of results generated from the subsystem being monitored. Value monitors can be viewed as additional safety checks that can be quickly identified when results do not make sense (i.e., negative altitude information being generated). The key idea is that value monitors should use a relatively simple logic (that is easier to validate) in order to test the validity of results generated from a more complex subsystem. Timing monitors (such as watchdog timers and heartbeat and activity monitors) are devices used to monitor things such as task schedule overruns and nonresponsiveness of tasks. Timing monitors can generate interrupts to signal the violation of a timing schedule.

Besides protecting the system against application-level faults, it is necessary to protect against faults in the COTS RTOS itself. Fault containment in COTS RTOS is typically achieved by the use of wrappers. A wrapper (a.k.a., invocation filter or mini-API) is middleware used between the software applications and the RTOS API. A wrapper can prevent the invocation of unwanted features in the RTOS. Wrappers can be designed at different levels of complexity. Some wrappers may bypass most API calls and concentrate on intercepting specific API calls that are deemed as being problematical. Wrappers can also intercept and perform logical checks into APIs to ensure that the user function is properly calling the API. This would be useful, for example, in RTOSs that lack proper exception handling, such as those tested using Ballista. As an alternate use, wrappers can complement the original API by implementing additional features desired by the application but are not provided in the original COTS software. Many COTS RTOS vendors and applicants suggested that wrappers are the fault-containment technique of choice and, as such, wrappers are discussed below in more detail.

5.3.1 The Role of Wrappers in COTS RTOS-Based, Safety-Critical Systems.

A software wrapper is a software layer used to protect, isolate, or interface to another component. Wrappers are viable candidates to protect COTS components within a system, without modification to the COTS component. Wrappers can be used to enhance a wrapped COTS component functionality, thus allowing it to meet all the targeted system requirements. In addition, wrappers can be used to mask COTS functionality that is not used in the new system implementation.

5.3.2 Wrappers—Interface Abnormalities.

Wrappers can be used to address three major issues with respect to employing COTS components in general:

- Consistency of operation of a COTS component is insufficient or not established by adequate evidence.
- Specification of the COTS component is incorrect or incomplete.
- COTS components are to be employed in a different context from that of the original design.

Wrappers can take action in response to detection of an abnormal circumstance at the RTOS interface (i.e., either input or output). Wrappers may be used to mitigate certain input conditions for which the COTS RTOS is known (or suspected) to produce anomalous behavior in the system. Input conditions that should be addressed include the following [Popov 01]:

- Inputs outside the domain intended, by system designers, for the COTS.
- Inputs outside the domain where the system designers consider the COTS trustworthy.
- Inputs in a domain for which the COTS is known to produce anomalous behavior.
- Inputs that are illegal per the COTS specification.
- Inputs determined to be erroneously generated by the system.
- Inputs generated by the system that are illegal outputs of the system, per the system specification.

These input conditions are determined through knowledge of the system design, the COTS item, and the operating modes of the system. Wrappers can also be used to mitigate certain COTS component output conditions, which are known to produce anomalous behavior. Output conditions that should be addressed include the following [Popov 01]:

- Outputs that are illegal for the COTS component, relative to its specification.
- Outputs that are determined to be erroneous.
- Outputs that are in a domain considered risky for the system.
- Outputs indicative of a COTS component, which used internal functions that are not trusted.

5.3.3 Wrappers—Responses to Interface Abnormalities.

The wrapper response is specified as part of the system design. Actions may range from simply reporting the abnormal circumstance to providing alternate functionality. Popov suggests wrapper actions may include the following:

- Report exceptions, erroneous outputs, erroneous inputs, anomalous behavior, errors detected.
- Substitute safe or default parameters or outputs or move the system to a safe state.
- Redirect action to alternate or default function such as a backup or simplified version.
- Retry previous actions that produced the current abnormality.

It has been suggested that the third most frequent cause of COTS design errors are discovered in the error-handling portions of the code [Ghosh 99]. For COTS, wrappers provide a good mechanism to address these "bugs," as they become known, through input and output screening.

5.3.4 Wrappers in a Kernel.

A loadable kernel module is sometimes the basis of wrapper implementations for COTS RTOSs. This approach has been the focus of recent research and development in the security domain, for mission critical systems [Fraser 99]. The [Fraser 99] research is summarized as follows:

1. Wrappers are run in kernel mode, executing in kernel space, with kernel protections.
2. The wrapper intercepts some or all of the system calls made by the wrapped application.
3. The wrapped application's interaction with the operating system and other processes is completely controlled by the wrapper, without context-switch overhead.

This approach has been demonstrated for FreeBSD and Solaris operating systems and may be applicable to any operating system that supports dynamically loadable kernel modules.

The development of wrappers for COTS operating systems is related to the development of operating system extensions, since both rely on kernel interfaces. Prototypes of SLIC [Ghormley 98], a system for efficiently inserting trusted extension code in existing operating system kernels, have been demonstrated for Solaris and Linux that interpose extensions on the kernel interfaces by modifying jump tables or by binary-patching kernel routines. Note that binary patching is a practice not condoned in DO-178B. The approach requires a well-defined interface to capture events and is limited in that new functionality can only be implemented in terms of existing functionality provided by the COTS operating system. This also requires a very detailed visibility into the internal workings of the RTOS; therefore, it is unlikely to be of much support for COTS RTOSs where this visibility is very limited.

Arguments associated with system service calls may not explicitly enumerate all the data necessary to implement effective wrappers. Access to additional data structures may be required, including global structures, kernel structures, and user mode structures. Utility functions to provide this access pose an additional complication in wrapper development and may require access to RTOS source code.

5.3.5 COTS RTOS Hardening Via Wrappers.

The hardening of COTS RTOSs, through the use of wrappers, is the subject of recent research efforts. COTS RTOSs are being incorporated in systems that must be highly reliable, secure, and safe. Wrappers can be used to modify the influence of RTOS behavior on the system but may require internal visibility of the RTOS. If an RTOS fault response causes an undesirable system response, then a wrapper may possibly be used to alter its response to one that the system is able to handle; or alternatively, initiate a fault mitigation strategy within the wrapper. In general, wrappers used in COTS software are limited in use, with respect to COTS RTOS. It is difficult to conceive how a wrapper could completely isolate the RTOS, considering that the services the RTOS provides may prohibit the effective integration of a wrapper, particularly on the hardware services side of the RTOS. Because of the pervasive role of the RTOS in controlling the entire system operation and the software applications that are executing on the RTOS, there not only needs to be wrappers between the RTOS and the APIs but also between the RTOS and the functions executing on it. Wrappers may be useful for checking input and output parameter boundary values (input/output screening and data validity checking), but they cannot, for instance, protect against inconsistencies in global data variables. It is not uncommon for the COTS RTOS vendor to include some built-in wrapper functionality in the COTS package, thus an additional wrapper layer by the integrator may be redundant at times. COTS RTOSs to be used in a safety-related application may reveal that a wrapper in itself is not sufficient. In this case, access to source code seems essential. A survey of the aviation safety community shows that designers of systems targeted towards safety-related applications have only used COTS RTOSs that have the source code available. Many times these RTOSs are reverse engineered in an attempt to achieve the acceptance of the aviation safety community guidance.

5.3.6 Wrappers and Software Assurance Levels.

For safety-critical systems, the wrapper software development is treated as any other critical component and is to be developed under the certification objectives and guidance of DO-178B. This means the certification of the wrapper must be obtained at the level of criticality appropriate for its function. As an interface between the COTS application program and the system RTOS, the wrapper may require certification at the level of the RTOS, which will be comparable to the highest criticality in the system; however, certain implementations may allow the wrapper software to be treated as application code, which is properly isolated in space and time by the RTOS architecture. In this circumstance, the criticality level of the wrapper can be the same as that of the application, which may be lower than the highest criticality in the system. In this case, certification required for the wrapper would then be according to the lower level of criticality.

The complexity of wrapper software can range from simple parameter passing to very complex kernel operations. The certification effort will vary according to the complexity and criticality level of the wrapper. Careful consideration of the wrapper development and certification effort is required to determine the cost-effectiveness of the wrapper implementation, with due consideration of the benefits of reusing the COTS and existing COTS certification evidence. The safety-critical system designer must do a careful benefit analysis of using COTS and the development cost of the wrappers required to meet system requirements. The CPU overhead of implementing wrappers has been evaluated in a number of research prototype projects. The overhead penalty of a given wrapper is dependent on the complexity and nature of the interface with the kernel. Typical overheads reported in the literature range from 2% to 15%.

As with all software, wrapper software is subject to design errors. The probability that design errors exist must be minimized through development processes such as described in DO-178B. Wrapper software must undergo verification testing appropriate to the criticality level. Functional testing with the COTS RTOS should include all the black box testing used to verify COTS functionality and to identify the COTS fault conditions addressed by the wrapper. Proper operation must be shown for all cases. Wrappers should be developed to the highest level of the software they are protecting.

6. DEVELOPING AN RTOS SVA AND STRESS TEST PLAN (A CASE STUDY).

This section describes how a software vulnerability analysis and potential stress test plan for an RTOS in a safety-critical environment could be developed. A specific RTOS that is being used by multiple aviation manufacturers was used as a baseline to represent the typical features of an RTOS. This particular RTOS is a real-time, pre-emptive, multitasking kernel designed for real-time, critical-embedded applications. The test plan is further augmented by considering testing additional features commonly found in other RTOSs. The stress test plan is developed considering the safety and protection mechanisms employed by RTOSs. The test plan describes a set of tests that can be used to verify some of the safety-enabling features of the RTOSs. Obviously, as mentioned in section 4.2, certain tests are dependent on how a particular feature is implemented in an RTOS. The test plan presented here is rather generic in nature and can be used as a baseline for evaluating multiple RTOSs.

The start of any case study requires understanding the features of the RTOS under study. In the specific RTOS considered, task-handling methods, memory management methods, and interrupting handling methods are basic system facilities. Partitioning and/or other protection mechanisms are also considered. The following facilities and features are under review.

- Task-Handling Method
    - Task model
    - Scheduling policies
    - Priority levels
    - Maximum number of tasks
    - Critical section involved in task switch
    - Minimum random access memory (RAM) required per task

- Memory Management Method

  - Memory management model
  - Maximum addressable memory space
  - Memory protection method (including cache)
  - MMU support

- Interrupting Handling Method

  - Interrupting handling model
  - Is interrupting nesting enabled? If it is, how many nest layers are enabled?
  - Minimum RAM required by interrupt
  - Context switch section and timing
  - Communication method between interrupt and tasks

Once the features are understood, the SVA analysis can commence. The result of which could possibly provide a basis for the robustness and stress test plan, input into the SSA, and possibly affect the overall system design, such as providing system design and architecture considerations via wrappers.

Conducting the SVA is an activity that requires a detailed understanding of the RTOS features and potential functional areas of concern as noted in section 4.2. The SVA itself is not presented here, however, having a base of expertise in the RTOS via the vendor or other users of the RTOS makes for a very effective approach. The detailed stress test plan offered is summarized in the table in appendix A.

7. CONCLUSIONS AND RECOMMENDATIONS.

This report supports the idea that a separate RTOS SVA and the resultant development of appropriate robustness and stress tests may be a vehicle to be used to effectively assess certain safety implications of COTS RTOSs to meet the robustness objective of DO-178B. It also supports the use of this analysis as needed input for the SSA.

Once the RTOS safety implications are understood, the following actions can be taken.

1. Prevent the presence of defects in the RTOS (i.e., fault avoidance), which can be done by proper design assurance.

2. Analyze and test the COTS RTOS and remove any defects if present.

3. Protect against remaining defects in the COTS using wrappers or other similar techniques.

8. BIBLIOGRAPHY.

[Timmerman 98] Timmerman, M., Beneden, B.V., and Uhres, L., “Windows NT Real-Time Extensions, Better or Worse?”, *Real-Time Magazine*, pp. 11-19, March 1998.

[Tindell 00] Tindell, K., “Deadline Monotonic Analysis,” Embedded Systems Programming, June 2000; http://www.embedded.com/2000/0006/0006feat1.htm.

[Rushby 99] Rushby, J., “Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance,” SRI International, March 1999. Work Sponsored by FAA Technical Center, NASA Langley Research Center, DARPA and NSA; http://www.csl.sri.com/users/rushby/papers/faaversion.pdf.

[Kleindermacher 02] Kleindermacher, D., and Griglock, M., “Real-Time Operating System Requirements for Use in Safety Critical Systems,” *Proceedings for the Embedded Systems Conference*, San Francisco, CA, 2002.

[Liu 73] Liu, C. L. and Layland, J.W., “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment,” *Journal of the ACM*, 20(1):46-61, January 1973.

[Carrette] Carrette, G., “CRASHME: Random Input Testing,” http://people.delphi.com/gjc/crashme.html.

[Miller 90] Miller, B., Fredriksen, F., and So, B., “An Empirical Study of the Reliability of Operating System Utilities,” *Communications of the ACM*, Vol. 33, pp. 32-44, December 1990.

[Miller 98] Miller, B., Koski, D., et al., “Fuzz Revisited: A Re-Examination of the Reliability of Unix Utilities and Services,” Computer Science Technical Report 1268, University of Wisconsin-Madison, May 1998.

[Koopman 00] Koopman, P. and DeVale, J., “The Exception Handling Effectiveness of POSIX Operating Systems,” *IEEE Transactions on Software Engineering*, Vol. 26, No. 9, pp. 837-848, September 2000; http://www-2.cs.cmu.edu/~koopman/ballista/tse2000/ tse2000.pdf.

[Fabre 00] Fabre, J.-C., Rodriguez, M., et al., “Building COTS Microkernel-Based Systems Using MAFALDA,” in *Proceedings of 2000 Pacific Rim International Symposium in Dependable Computing*, pp. 85-92, December 2000.

[Jaffe 00] Jaffe, M.S., “Architectural Approaches to Limiting the Criticality of Commercial-Off-The-Shelf (or Other Re-Used Software) Components in Avionics Systems,” *Aviation Today*, September 18, 2000; http://www.aviationtoday.com/reports/ arch_methods.htm.

[Popov 01] Popov, P., Riddle, S., et al., “On Systematic Design of Protectors for Employing OTS Items,” University of Newcastle Technical Report No. CS-TR-730, April 2001. Centre for Software Reliability, UK; http://www.csr.ncl.ac.uk/dots/pubs/ euromicro-tr.doc.pdf.

[Fraser 99] Fraser, T., Badger, L., and Feldman, M., “Hardening COTS Software With Generic Software Wrappers,” *1999 IEEE Symposium on Security and Privacy*; http://citeseer.nj.nec.com/fraser99hardening.html.

[Ghormley 98] Ghormley, D., Rodrigues, S., et al., “SLIC: An Extensibility System for Commodity Operating Systems,” *Proceedings of the USENIX 1998 Annual Technical Conference*, New Orleans, Louisiana, 1998; http://citeseer.nj.nec.com/ghormley98slic.html.

[Ghosh 99] Ghosh, A., Schmid, M., “An Approach to Testing COTS Software for Robustness to Operating System Extension and Errors,” *Proceedings of the Tenth International Symposium on Software Reliability Engineering*, pp. 166-174, 1999; http://www.cs.nps.navy.mil/people/faculty/bmichael/sw4540/article7-ghosh.pdf.

[RTCA SC167] DO-178B, “ Final Report for Clarification of DO-178B ‘Software Considerations in Airborne Systems and Equipment Certification,’” Special Committee 190, RTCA, Inc., 1992.

[RTCA SC190] DO-248B, Final Report, Special Committee 190, 2001.

[ARP4754] Aerospace Recommended Practice 4754 Certification Considerations for Highly Integrated or Complex Aircraft Systems, 1996.

[COTS SW] United Technologies Research Center, Krodel, J., “Commercial Off-The-Shelf (COTS) Avionics Software Study,” DOT/FAA/AR-01/26, May 2001.

9. RELATED DOCUMENTS.

Leveson, N., *Safeware: System Safety and Computers*, Addison-Wesley, 1995.

United Technologies Research Center, Thornton, R., “Review of Pending Guidance and Industry Findings on Commercial Off-The-Shelf (COTS) Electronics in Airborne Systems,” DOT/FAA/AR-01/41, August 2001.

AEEC, “Avionics Application Software Standard Interface,” January 1997.

10. GLOSSARY.

Combinatorial explosion – The condition of massive amounts of test cases that develop when testing all the combinations of paths that occur in a software system.

Deactivated code – Executable object code (or data) which, by design, is either (a) not intended to be executed (code) or used (data), for example, a part of a previously developed software component, or (b) is only executed (code) or used (data) in certain configurations of the target computer environment, for example, code that is enabled by a hardware pin selection or software-programmed options. [RTCA SC167]

Dead code – Executable object code (or data) which, as a result of a design error, cannot be executed (code) or used (data) in an operational configuration of the target computer environment and is not traceable to a system or software requirement. An exception is embedded identifiers. [RTCA SC167]

Deadlock – A situation where two or more tasks are forever suspended attempting to obtain two or more shared resources (e.g., semaphores). Since each task has the semaphore that the other needs, the tasks could suspend on the semaphores forever.

Priority inheritance – A mechanism for avoiding priority inversion by temporarily boosting the priority of a task using a semaphore, while it owns the semaphore, to the maximum priority of all tasks that also uses the same semaphore.

Priority inversion – A lower-priority task may get executed instead of a higher-priority task that is ready. Priority inversion occurs when a higher-priority task is suspended on a semaphore that a lower-priority task has, and the low-priority task, in turn, gets pre-empted by a middle-priority task. The middle-priority task gets to execute before the high-priority task.

Robustness testing **–** A method of verification to demonstrate that software can continue to operate correctly despite invalid inputs. [RTCA SC167]

Stress testing – A method of testing that subjects the system to the extremes of real-world workloads, large data volumes, repetitive operations, and operations for extended periods of time.

APPENDIX A—SAMPLE TEST PLAN FOR COTS RTOS

Some test cases are dependent on information about the target environment; however for this test plan, no system in particular was targeted. Lower level testing is also offered and visibility of the source code would be of benefit.

It should be noted that the test cases presented are not comprehensive, and specific features of a particular RTOS will require additional or modified testing. Also, although the most common RTOSs features are considered for testing, some test cases listed may not apply at all to some RTOSs that lack a particular feature being tested. Whenever there is a correspondence, the test cases presented are cross-referenced to the vulnerabilities described in section 4.2, for a specific RTOS considered in that section.

Test cases are presented. Some have correspondence to the RTOS vulnerable features described in section 4.2.

Test No.	Test Name	Test Purpose	Test/Verification Activities	Related Vulnerability Aspect (Section 4.2)
1	Memory protection: Memory management unit (MMU) support and operation	Processors often provide an MMU to enforce memory protection. The purpose of this test is to check whether the MMU is used by the RTOS and whether applications are prevented from accessing unauthorized address space.	Verify whether the processor contains an MMU and, if so, whether the RTOS used it. Run tasks, which attempt to perform, read and write operation outside their designated address range and verify whether exceptions are raised by the system.	M5
2	Dynamic memory allocation operation	Check if allocation of dynamic memory pool is performed correctly.	Create a dynamic memory pool, print the dynamic memory pool's information: ● Allocate some memory larger than the minimum number of bytes allocation from this dynamic memory pool; print the dynamic memory pool's information. ● Allocate some memory smaller than the minimum number of bytes allocation from this dynamic memory pool or larger than the maximum memory available from the memory pool; see if the error messages occur. ● Create another dynamic memory pool, print the number of the established memory pools and obtain the memory pool pointers and print the memories' information, de-allocate that allocated memory, delete the memory pools.	M1
3	Dynamic memory allocation quota	Check if the RTOS has resource protection mechanisms to prevent a malicious task from consuming resources unlimitedly.	● Create a task that requests memory in an infinite loop, while another critical task requests little memory (create objects, etc.) periodically. It should be checked if the previous task would corrupt the critical task. ● A task repeatedly creates several kinds of kernel objects, such as semaphore, subtask, and so on. Each object has a control block itself. It can be checked if the control blocks consume all memory available and take the system down.	M1

Test No.	Test Name	Test Purpose	Test/Verification Activities	Related Vulnerability Aspect (Section 4.2)
4	Automatic resource reclamation; prevention of memory leaks	When applications terminate, for whatever reason, any associated resources should be reclaimed unless they are intentionally kept as backup; otherwise the system may eventually run out of resources. The purpose of this test is to check whether resources are properly reclaimed after use.	In this test, a task creates subtasks in an infinite loop. The subtasks just request memory and finishes without freeing up memory. It should be checked if the application would eventually run out of memory.	M1
5	Stack overflow protection	To check whether the RTOS provides facilities to protect against stack overflow.	In this test, a task can call some functions to overflow its stack. It can then be checked if the kernel will suspend the task, or the task will corrupt the whole system.	T3
6	Kernel's storage area overflow	Verify whether the RTOS establishes quotas for usage of the control block storage area, to prevent one task from consuming the entire space and preventing other tasks from starving for this resource.	Create a condition in which one low-priority task repeatedly consumes additional kernel storage area resources. Verify whether such procedure affects the operation of a high-priority task that periodically allocates and releases such kernel space.	T2
7	Critical resource monitor	The fault detection capabilities of an RTOS are enhanced by proactively verifying the integrity of system elements. Such verification can be as simple as mechanisms for monitoring critical components periodically or on demand. The purpose of this test is to check how effective these mechanisms are.	Design a critical resource monitor using facilities the RTOS provides. Evaluate whether these facilities are adequate.	M7
8	Protection against improper invocation of service calls	Check whether the kernel protects itself from applications that pass illegal parameters to service calls.	A series of test cases shall be designed that pass illegal values to the parameters of the various system calls. It should be verified whether these test cases are handled properly by the system.	M2

Test No.	Test Name	Test Purpose	Test/Verification Activities	Related Vulnerability Aspect (Section 4.2)
9	Fault-tolerance: Fault detection, isolation, and recovery	When a task fails, the kernel should be able to load a recovery agent that runs in "supervisor" mode. The kernel should also log the event that preceded the failure. The purpose of the test set is to check whether such operations are properly executed.	The RTOS provides application tasks with several facilities that improve diagnosis of system problems. The test should show if those fundamental features could enable the system designer to build in fault tolerance.	I5
10	Determinism of execution times of operations with objects such as mailboxes, queues, pipes, mutexes, semaphores, event groups, signals and timers	All kernel services must have predictable execution time, since they affect the real-time operation of the system. Tests should be designed to measure the throughput for performing operations using the referred RTOS objects. Measure whether the timing for obtaining and releasing semaphores, and sending and receiving messages, among other operations, is predictable with respect to parameters such as the number of tasks suspended or the size of the message.	A series of tests should be designed to measure the timing of access to various objects, with respect to number of messages, tasks, etc. In the case of semaphores, for instance, the following tests can be performed: • Measure the time to acquire synchronization in a no-contention/contention/not available case. In this test, the time it takes to acquire a binary semaphore will be tested. • Measure the time to release synchronization in a no-contention/contention case. In this test, the time it takes the system to release a binary semaphore and schedule a higher-priority task that was thereby released is measured. A test cycle starts with only one task pending on the semaphore that is about to be released. Then, the number of tasks is increased one by one until there are several hundred tasks of different priority pending on the semaphore, and a new test cycle can start. The idea is to investigate whether or not the time to release the semaphore (and schedule the released task) is proportional with the number of tasks waiting for the semaphore.	S6

Test No.	Test Name	Test Purpose	Test/Verification Activities	Related Vulnerability Aspect (Section 4.2)
11	Task switching latency	Task switch latency is the time interval between the last instruction of the currently running task before giving up the processor (voluntary or involuntary) and the first instruction in the next ready task. The purpose of this test is to: ● Verify whether or not the time the operating system needs to perform a task switch is dependent on the number of tasks in the ready-queue. ● Measure the overhead caused by memory management.	Measure the task switch latency for tasks running in the same process and having the same priority level, using FIFO scheduling. In this test series, a number of tasks of equal priority are created. As soon as anyone of these tasks becomes active, it voluntarily releases the processor to let another task of the same priority run. This test is executed with different number of tasks. The metric measured is the task switch latency, i.e., the time it takes to switch from one task to another.	S6
12	Task switching latency under heavy load conditions	Verify whether or not the system's performance is affected when it is loaded with a large number of objects.	Examples of test cases: ● Run the task switch latency test (no. 11) while the system is loaded with the maximum number of semaphores that can be created without running out of memory. ● Run the task switch latency test while the system is loaded with the maximum number of tasks that can be created without running out of memory. Verify the impact of task-switching latencies on the scheduling of tasks.	S6
13	Task scheduling technique	Verify whether the task scheduling method in implemented correctly.	First, the scheduling method used by the RTOS needs to be identified (RMS, round-robin, etc.). Then tests should be designed to verify whether task scheduling is performed in accordance to the scheduling method in use. The nature of the tests will depend on the scheduling method in use, but typically involve defining tasks with different execution times, deadlines, and priority levels, loading these task onto the system, and verifying how they get scheduled by the RTOS.	S6

Test No.	Test Name	Test Purpose	Test/Verification Activities	Related Vulnerability Aspect (Section 4.2)
14	Priority inversion	The priority inheritance mechanism is absolutely crucial for an RTOS, and its operation should be verified.	A test case should be designed to create a situation with several tasks where the priority inversion problem occurs: a high-priority task wants to acquire a binary semaphore that is owned by a low-priority task. A medium priority task keeps the low-priority task from running and releasing the binary semaphore so that it cannot be acquired by the high-priority task. In this test, the time it takes for the highest priority task to acquire the binary semaphore needs to be measured. This time includes the time it takes to boost the priority of the lowest priority task, have it release the binary semaphore, and switch back to the highest priority task so it can acquire the binary semaphore. In this test, a higher-priority task is suspended on a semaphore that a lower priority task has. Then the low-priority task is pre-empted by a middle-priority task. It should be checked if the low-priority task will execute and release the semaphore and the high-priority task gets it to run again.	S2
15	Protection of task schedule from creation of new tasks	Verify how the system protects the execution of a critical task from a malicious task that spawns new tasks that consume additional CPU time. Ideally, the RTOS should assign a quota that guarantees minimum CPU time to a partition, independently of events in other partitions.	In this test, two tasks can be created, Task B being a critical task while Task A is not. During the test, Task A spawns new tasks. It should be checked if the run-time characteristics of Task B are affected by Task A.	S4
16	Corruption of task control blocks	Verify protection of task control blocks from access via application tasks	Such condition should be effectively prevented with proper configuration and operation of the MMU. This test is directly related to Test 1.	S1
17	Interrupt, Exception and Signal-handling capability	Verify whether a proper handler is defined for each type of interrupt, exception, and signal that can be raised in the system.	Test cases are dependent partly on the hardware platform where the RTOS runs. These test cases will attempt to trigger each of the potential hardware interrupts, exceptions, and signals and verify whether the corresponding ISR gets invoked.	I2, I3, I4

Test No.	Test Name	Test Purpose	Test/Verification Activities	Related Vulnerability Aspect (Section 4.2)
18	Simultaneous interrupts and interrupt nesting	• Determine how long the system needs to respond to two simultaneously occurring interrupts. • Determine if the interrupt handling is prioritized and that interrupts can be nested.	• High- and low-priority interrupts occur simultaneously. Measure the latency to service both interrupts. This test measures the time it takes the system to respond to two simultaneously occurring interrupts. • Low-priority interrupts occurs first, followed shortly by the high priority interrupt. The high-priority interrupt occurs before the system has finished servicing the first interrupt. Measure the latency to start servicing the high-priority interrupt and verify that interrupt handling is prioritized.	M7
19	Interrupts during atomic operations	Interrupts during operations, such as task switching, should be disabled in order to prevent leaving the RTOS in an inconsistent state.	Develop a test case that triggers a high-priority interrupt while the kernel is in the process of switching tasks. Verify that the interrupt processing is delayed until the task switching activity is completed.	I1
20	Protection of supervisor task	Verify whether supervisor task runs in a protected area that cannot be corrupted by other applications.	Typically, the processor provides a user mode and a supervisor mode operation, where supervisor mode is reserved for kernel and fault recovery operations. Verify whether such is the case.	I5
21	Deadlock prevention and recovery	Verify what mechanisms are used by the RTOS to handle deadlocks. The RTOS may implement mechanisms to prevent deadlocks or correct deadlocks via time-outs, for example.	In this test, two tasks and two semaphores are created. The first task has the second semaphore and the second task has the first semaphore. Then the second task attempts to obtain the second semaphore and the first task attempts to obtain the first semaphore. This will check if the system can check out the deadlock.	S3
22	Inclusion of deactivated code and dead code	Verify inadvertent execution of dead code and deactivated code.	This condition is hard to test, and it is more related to testing in the target system, rather than unit-testing the RTOS. Adequate testing may require reverse engineering of object code and check for any conditions that may cause the "idle" code to be activated and test for such condition.	C1, C2
23	Protection against modifications in task priority assignment	Verify how the RTOS prevents a task from inadvertently or maliciously modifying its priority level.	Create a task in which it attempts to raise or lower its priority (typically through a system call); verify how the RTOS handles such attempts.	S5

Test No.	Test Name	Test Purpose	Test/Verification Activities	Related Vulnerability Aspect (Section 4.2)
24	Task, message, and kernel work queue overflow	Verify whether the RTOS gracefully handles the attempts to overflow the various activity queues.	Test cases: • Create a task that, in turn, creates multiple tasks ready to run in a recursive mode. This should cause the task queue to overflow. • Create a task that sends messages to another task in an infinite loop. The receiving task fails to consume the message. This should cause the message queue to overflow. • Create a task that has long execution time and in of highest priority; in the mean time, repeatedly generate interrupt requests from an external source that will create a backlog in the kernel work queue.	Q1, Q2, Q3
25	Unauthorized access to critical system devices	Verify whether the RTOS prevents applications from accessing I/O devices that they are not authorized to.	Verify what mechanism is used to protect access to critical system devices. If devices are memory-mapped, this protection may be via MMU, and Test 1 can be replicated to test this condition.	M6