WINDOWS CE 5.0

Disclaimer

Although all care has been taken to obtain correct information and accurate test results, Dedicated Systems Experts and Dedicated Systems Magazine cannot be liable for any incidental or consequential damages (including damages for loss of business, profits or the like) arising out of the use of the information provided in this report, even if Dedicated Systems Experts and Dedicated Systems Magazine have been advised of the possibility of such damages.

http://www.dedicated-systems.com

E-mail: info@dedicated-systems.com

EVALUATION REPORT LICENSE

DOCUMENT CHANGE LOG

Issue No.	Revised Issue Date	Para's / Pages Affected	Reason for Change
1	August 24, 2004	All	Initial Issue

1 Introduction

1.1 Purpose and scope

This paper presents the qualitative evaluation results of the Windows CE 5.0 operating system. Test results of the Windows CE operating system on a X86 platform can be found also on the website: “Ce 5.0 on a X86 platform.” [Doc. 5].

The layout and the content of this report follow the one depicted in “The evaluation test report definition.” [Doc. 3] and “The OS evaluation template.” [Doc. 4]. See section 1.3 of this document for more detailed references of these documents. Therefore these documents have to be seen as an integral part of this report.

Due to the tightly coupling between these documents, the framework version of “The OS evaluation template.” has to match the framework version of this evaluation report (which is 2.9). More information about the versioning of documents, tests and the relations between both can be found in “The evaluation framework.” see [Doc. 1] in section 1.3 of this document.

1.2 Document issue: the 2.9 framework

This document shows the results in the scope of the evaluation framework 2.9.

1.3 Related documents

These are documents that are closely related to this document. They can all be downloaded using following link:
http://www.dedicated-systems.com/encyc/buyersguide/rtos/evaluations

Doc. 1	The evaluation framework. This document presents the evaluation framework. It also indicates which documents are available, and how their name giving, numbering and versioning are related. This document is the base document of the evaluation framework. EVA-2.9-GEN-01	Issue: 1	Date: April 19, 2004
Doc. 2	What is a good RTOS? This document presents the criteria that Dedicated Systems Experts use to give an operating system the label “Real-Time”. The evaluation tests are based upon the criteria defined in this document. EVA-2.9-GEN-02	Issue: TBD	Date: TBD
Doc. 3	The evaluation test report definition. This document presents the different tests issued in this report together with the flowcharts and the generic pseudo code for each test. Test labels are all defined in this document. EVA-2.9-GEN-03	Issue: 1	Date: April 19,2004
Doc. 4	The OS evaluation template. This document presents the layout used for all reports in a certain framework. EVA-2.9-GEN-04	Issue: 1	Date: April 19, 2004
Doc. 5	OSE 4.5.1 on a PPC platform. This document presents the layout used for all reports in a certain framework. EVA-2.9-OS-OSE-PPC-01	Issue: 1	Date: April 19, 2004

2 Results summary

2.1 Product evaluated

Windows CE 5.0 Microsoft Corporation, Inc.(x86 platform)

2.2 Theoretical evaluation conclusion

2.2.1 Positive points

– Extensive platform support

– Generally good real-time performance

2.2.2 Negative points

- Documentation insufficient for such a complex system

2.2.3 Ratings

For a description of the ratings, see [Doc. 3].

RTOS Architecture	0	8	10
OS Documentation	0	6	10
OS Configuration	0	7	10
Internet Components	0	9	10
Development Tools	0	8	10

3 Introduction

3.1 Product evaluated

3.1.1 RTOS

Windows CE 5.0 Microsoft Corporation, Inc.(x86 platform)

3.2 Introduction

The Windows CE RTOS provides multiple built-in configurations (Tiny Kernel, Enterprise Terminal, VOIP,….), together with lots of development technology (SDK's, BSP's,..) to customize these configurations. In addition to understand and test the developments made, Windows CE inserted some remote tools. The Windows CE 5.0 RTOS provides real-time application support for time-critical systems.

3.3 Supported CPU

Following CPU are supported by Windows CE 5.0:

– An Emulator

– ARMV4T, ARMV4I, ARMV4

– SH3 , SH4

– MIPSIV_FP, MIPSII_FP, MIPSIV, MIPSII, MIPS16

– X86

4 Technical evaluation

This is the qualitative approach of the evaluation. As such the scores are given in respect of the experiences gathered for the product evaluated, and by definition they aren't based on objective criteria. Therefore this part of the evaluation is not used to differentiate between the three validation logos.

Remark: although the technical evaluation is done in this report, the quantitative test results of this operating system on a certain platform can be found in other reports.

4.1 OS Architecture

RTOS Architecture	0	▨	▨	▨	▨	▨	▨	▨	8			10

Windows CE .NET has a modular architecture and is highly scalable. The virtual memory and privilege level protection make the system robust.

Windows CE.NET is a priority based pre-emptive real-time operating system with protections between processes and a protected kernel.

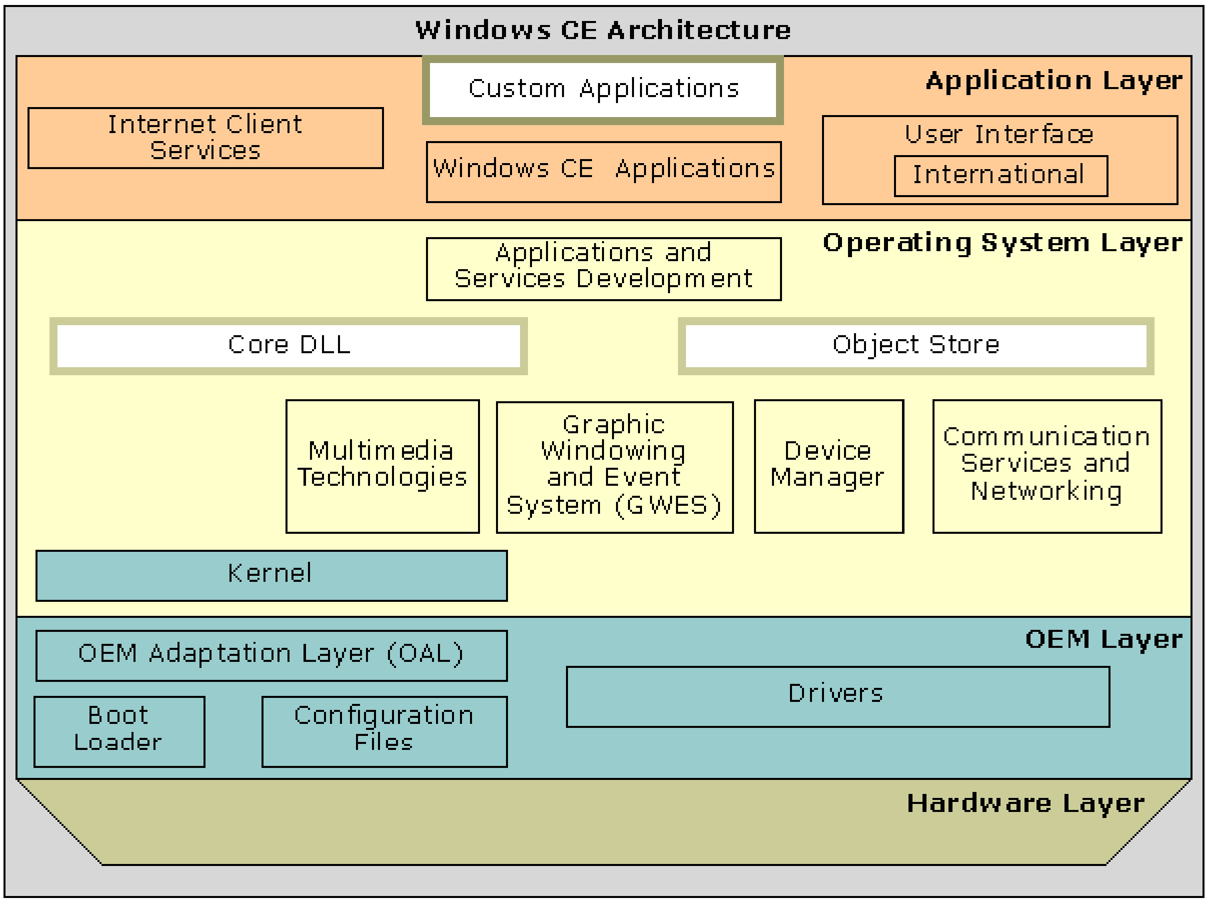
For the people that have still doubts about it, we can confirm that Windows CE 5.0 has all the features to make it a valid real-time operating system (this statement is true since Windows CE version 3.0).

It is highly modular, with a relative simple kernel and where optional components run as separated processes. By using separate processes for optional modules the OS becomes more reliable.

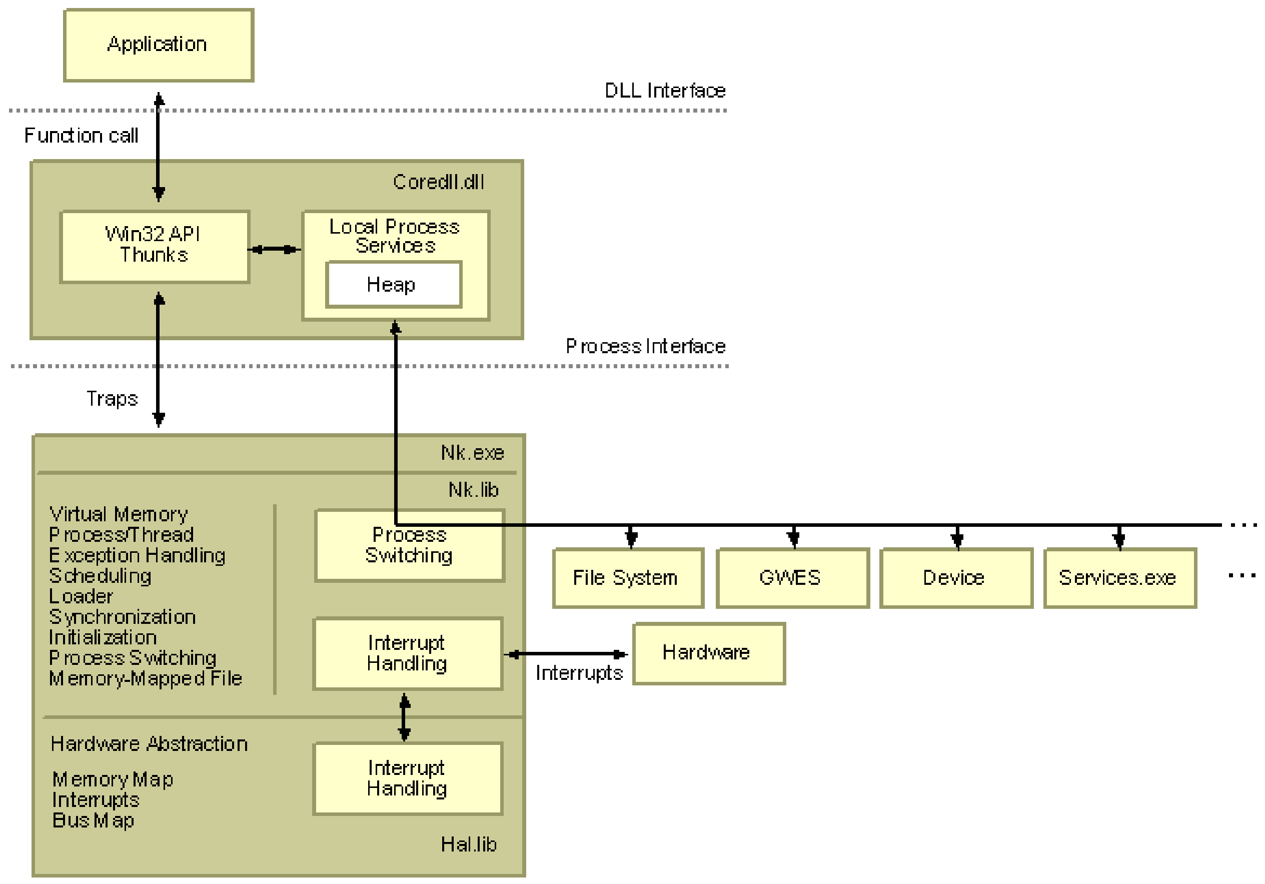
System calls are handled in the kernel (by a software trap interface) en redirected to the correct process if needed.

The main advantages of Windows CE are the interoperability with the general purpose windows environment (with technologies like DCOM) and the ease to make windowing applications just like in the general purpose windows operating system. This gives you a large developer base for the non real-time part of your system. The real-time part is better designed by developers with a good understanding of real-time behaviour.

Although it has a modular design so that a minimal configuration can suit small systems, it is to complex and has a to large footprint to be a good solution for deeply embedded systems.



CE Architecture



Redirection of function calls to optional modules

4.1.1 Task Handling Method

4.1.1.1 Processes and threads

Windows CE is a classical pre-emptive multitasking system with support for both processes and threads within a process. CE uses a priority based scheduler to decide which thread that is ready to run it will activate. Remark that priority number 0 is highest priority. Threads with the same priority are scheduled using a fair round-robin time slice mechanism. This time-slice quantum can be set for each thread separately.

The number of running processes in windows CE is limited to 32. The reason for this can be located in the partitioning of the virtual memory which is explained further in this document.

The number of running threads within a process is only limited by the available system resources (RAM).

4.1.1.2 Thread priorities

Windows CE uses 255 priority levels which is high enough to accommodate complex real-time systems.

Compared with other RTOS, Windows CE is build to be as compatible as possible with a General Purpose operating system. Nevertheless different changes were done in the kernel to make CE suitable for real-time systems:

– To be compatible with GPOS Windows, the lower 8 priorities are the same as used in the normal Windows SetThreadPriority calls. Thus ported windows applications to CE will not affect the real-time behaviour of the system.
– The 248 highest priority levels can only be set by the specific CeSetThreadPriority call.

An OEM can protect the top most 248 priority levels, so that only the lowest 8 levels are available for applications. This makes it possible to add third-party software without compromising real-time performance.

Microsoft recommends using the priority levels ranges as follows:

– 0 through 96:	Reserved for real-time above drivers
– 97 through 152:	Used by the default Windows CE–based device drivers
– 153 through 247:	Reserved for real-time below drivers
– 248 through 255:	Mapped to non-real-time priorities

Lucky the system designer can adapt this to his needs: the developer is free to use any priority to fit his application.

Microsoft did a good job to document the default priority of all system and driver threads. These default priorities can be adapted by changing some register settings: well done!

It is a pity that it isn't possible to create a thread with a predefined priority level: each thread is created with the default priority level (THREAD_PRIORITY_NORMAL: 251). Therefore you will need to change the priority of the thread after its creation.

Remark that interrupts run as a thread and as such they use the same priority levels (see the interrupts section further in this document).

4.1.1.3 Fibers

Windows CE supports also "Fibers".

Fibers are also thread of executions but NOT managed by the kernel. Each fiber runs in the thread context of the thread that created it.

So fibers are only useful to create your own scheduler within a thread.

This can be maybe used for some exotic applications or for legacy systems, but normally it is not a good idea to use this in real-time systems. Remark that Microsoft disapproves the use of it in its documentation where they state "In general, fibers do not provide advantages over a well-designed multithreaded application".

	OS under evaluation

	OS under evaluation
Supported memory models	☺ - Processes protected between each other, with multiple threads in one process. - Flat memory model: no protection between user processes and kernel.
Thread/process priority levels	☺ 256 levels
Max. number of processes	32 processes
Max. number of threads	The maximum number of *threads* in a process is only limited by the amount of memory available.
Scheduling policies	Priority based scheduling Round-robin between threads at the same priority level, with adjustable time-slice (quantum). When the quantum is set to zero, the thread runs to completion.
Number of documented thread states	5 states (running, suspended, sleeping, blocked, and terminated).
Number of undocumented thread states	

4.1.2 Memory Architecture

4.1.2.1 Boot rom

It is possible to configure Windows CE as such, that it can be executed from place in ROM. This saves both booting time (no need to decompress/copy programs into RAM) and RAM.

This is no limitation, booting can be done by flash card, compressed ROM and so on. Of course, in such case the program is not executed in place.

4.1.2.2 Physical memory

Windows CE can manage up to 512MB of physical memory (being it device memory or RAM). Microsoft build-in this limitation on purpose so that CE can run on a lot of embedded processors without compatibility problems. Some of these processors can only manage 512MB physical memory.

Remark that depending on the platform used, the same physical memory can be re-mapped in multiple regions depending on cache settings and privileges for each region. Thus the addressing region can be much larger than the physical 512MB.

The RAM on a Windows CE–based device is divided into two areas: the object store and the program memory.

– The object store resembles a permanent, virtual RAM disk. Data in the object store is retained when you suspend or perform a soft reset operation on the system (if the device has a backup power supply for the RAM). When operation resumes, the system looks for a previously created object store in RAM and uses it, if one is found. Devices that do not have battery-backed RAM can use the hive-based registry to preserve data during multiple boot processes.

– The program memory consists of the remaining RAM. Program memory works like the RAM in personal computers — it stores the heaps and stacks for the applications that are running.

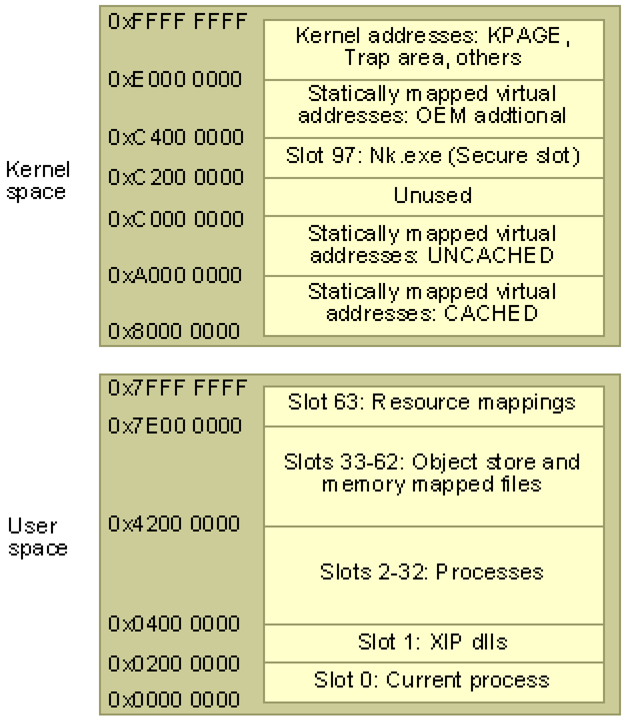
Dynamic moving memory allocation from object store RAM to program RAM is possible to meet RAM requirements for an application. In the test report 'EVA-2.9-TST-CE-x86-0x.doc' you can see a memory test that tests this. We did a memory tests in a way that the RAM was consumed very fast: the RAM moved until the bottom 64Kb was reached.

4.1.2.3 Virtual memory

When Windows CE OS starts, it creates a single 4-gigabyte (GB) virtual address space. The address space is then divided into two sections: kernel and user space depending on the highest address bit. This is the same as in the general purposes Windows operating systems.

The user space is further divided in 64 slots of 32 MB from which 32 are reserved for the different processes (this makes the 32 processes limit).

All the processes share the virtual address space! However processes do not have access to other processes. This 32 MB virtual address space is to be used for the program/data/heap and so one. It is possible to have access to more virtual memory than this 32MB limit by giving it access rights for the process. This can be done by making some memory accessible from the process (VirtualAlloc) or by using memory mapped files.


0xFFFF FFFF
Kernel addresses: KPAGE, Trap area, others
0xE000 0000
Statically mapped virtual addresses: OEM addtional
0xC400 0000
Slot 97: Nk.exe (Secure slot)
0xC200 0000
Unused
0xC000 0000
Statically mapped virtual addresses: UNCACHED
0xA000 0000
Statically mapped virtual addresses: CACHED
0x8000 0000
Kernel space
0x7FFF FFFF
Slot 63: Resource mappings
0x7E00 0000
Slots 33-62: Object store and memory mapped files
0x4200 0000
Slots 2-32: Processes
0x0400 0000
Slot 1: XIP dlls
0x0200 0000
Slot 0: Current process
0x0000 0000
User space

Windows CE Virtual address space allocations

4.1.2.4 Paging

Windows CE implements a paged virtual memory management system similar to other Microsoft Windows–based desktop platforms. The page size depends on the platform, but if possible 4,096 bytes (4 KB) is used.

Important in real-time systems is that it is possible to disable paging. In this mode, a module is fully loaded into memory before it is run. So paging faults can not occur during the run of the application.

4.1.2.5 Memory protection

Under normal configurations the MMU is used to protect processes from each other and to protect the kernel space. This is the same as currently used in most General Purposes operating systems.

This protection helps a lot for making reliable systems. However, for some situations the performance penalty of the protection could be too much. In such cases it is possible to configure CE to use no protection between processes and the kernel.

4.1.2.6 Heap

The heap in windows CE is optimized for small objects. However, just like in any real-time system long-time usage can lead to a fragmented heap causing extra lookup delays and even failing allocation requests.

You can add your own heaps in a process. This can then be used for instance for fixed size allocations (preventing fragmentation).

	OS under evaluation
MMU support	Yes (or NOT, can be set in boot loader)
Physical Page Size	1Kb or 4Kb along 64 Kb regions
Swapping/Demand Paging	Supported, but can be disabled to achieve real-time performance.
Virtual memory	YES
Memory protection models	☺ Full virtual memory protection. Each process runs in its own virtual memory space. This can be disabled for performance reasons.

4.1.3 Interrupt Handling

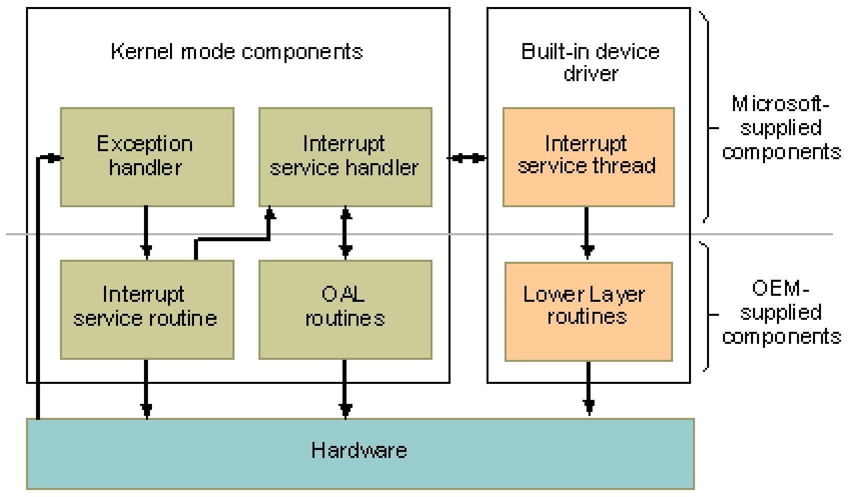
Interrupt handling in windows CE is a bit different than more tradition RTOS. Microsoft uses the same terminology as we do for the interrupt handling:

- The Interrupt Servicing Routine (ISR): low level routine in the kernel with limited system calls.
- The Interrupt Servicing Thread (IST): thread at application level handling the interrupt with access to all system calls.

The difference is that in Windows CE, the ISR is meant to be implemented in the OAL (OEM Abstraction Layer) just for handling the lowest level of the interrupt (interrupt controller). It is not easy to add quickly an ISR… it is even impossible to do this outside the OEM initialization code!

Therefore device drivers will in general run their interrupt handling in the IST. Interrupt service threads act just like any other thread: so they use the same priority system! This means that the system designer can even give an IST a lower priority than some application thread.

This system is shown in the figure below.



Interrupt handling in Windows CE 5.0

This system has its pros and contras. The main negative point is the extra delay until the IST is activated. As an advantage it makes interrupt handlers simpler (all system calls available) and gives the system designer a high degree of freedom. Another advantage is that the hardware interrupt levels do not affect the IST priorities.

If however for some real-time constraints shorter latencies are required, it is possible to make a custom ISR by adapting the OAL (in fact by adapting the BSP). As said before: this is not an easy task and can not be used by a device driver for a card that can be used on different platforms.

	OS under evaluation
Handling	☺ Nested and prioritized CE uses a thread interrupt model to encourage the use of threads to handle most of the interrupt service work. OEMs can access a kernel ISR also to perform a minimal amount of work. Windows CE also provides installable ISRs that can be installed.
Context	The ISR runs in a special context and uses virtual addresses statically mapped by the OEM.

	OS under evaluation
Context	The ISR runs in a special context and uses virtual addresses statically mapped by the OEM. The IST is a normal application thread and has its own context and priority like any other thread in the system.
Stack	The IST is a normal application thread and has its own stack.
Interrupt-to-task communication	Only an event can be used from within the ISR to signal the IST. No other API is accessible from within the ISR. OEM can create a shared memory region by statically mapping a memory region into the ISR's address space.

4.1.3.1 System timer

The OAL has to implement the system timer interrupt (in fact that is the only required ISR in the kernel) that should activate each millisecond. This interrupt has to check if any time-out is reached and if so signal a "reschedule" event to the kernel. In the other case a "NOP" is signalled to the kernel and no rescheduling occurs.

New in windows CE 5.0 is the ability to slow down the timer interrupt if no threads are running: this is useful for saving power (important for handheld devices).

So the clock timer runs never at thread level and is compact. As long that no time-outs are detected the delay will be small: increment the milliseconds timer and compare it with the reschedule time set by the kernel.

4.1.3.2 Hi resolution timers

An OEM can add hi resolution timers for more accurate timing or performance analysis. This is not obligatory (as this will depend on the used hardware platform).

These timers can only be read from an application: they are not used by the kernel in scheduling threads and calculating time-outs.

4.1.4 Synchronisation mechanisms

In windows CE, the synchronisation mechanisms can be divided in two distinctive categories:

– Protection mechanisms against simultaneous access: critical sections, mutexes and semaphores.

– Communication mechanisms: events and message queues.

These two categories are so fundamentally different implemented that you should not use protection mechanisms as a communication mean. This is not a negative point, at the contrary: see the sections below.

Main difference with other RTOS is the use of generic wait functions (like WaitForSingleObject) independent of the object type (mutex, semaphore, events but also process and threads). This is the same like in the General Purpose Windows Operating System. As an advantage it is possible to wait on multiple objects (until one of them is signalled). For the experienced real-time systems developer (with other RTOS) this system is a bit strange.

4.1.4.1 Protection mechanisms

Three mechanisms are foreseen in Windows CE to do this:

– Critical sections: Can only be used within a process

– Counting semaphore: can also be used between processes

– Mutex: like the counting semaphore but without counting

Fundamental here is that Windows CE uses priority inheritance to avoid priority inversion cases! Even better: it is always there, it is impossible to disable priority inheritance!

This is a feature that we didn’t find in any other RTOS we evaluated before. In our opinion having priority inheritance always enabled is how it should be done in all RTOS! Remember that the system crash of the Mars Pathfinder was caused by a mutex used in its defaults settings: without priority inheritance enabled!

Of course, some people will argue that this causes the protection mechanisms to be a bit slower. This is true in the average case, but in real-time systems the average is of no importance: the worst-case is the only time we are interested in. Priority inheritance does exactly improve worst-case situations!

In our opinion it doesn’t make sense to disable priority inversion whatsoever, just like it is useless to disable the brakes of a car…

One remark here: priority inheritance works only at one level. When using a second protection mechanism while the priority is already increased will not cause a second inheritance. This is not a main problem: using a second protection while already in a protected area is just an example of bad design and should never happen in any well build real-time system.

4.1.4.2 Communication mechanisms

Following communication mechanisms are foreseen in Windows CE:

– Events: to signal that something occurred, can be used between processes.

– Message queues: to pass data between threads (not usable between processes).

Not much to explain here: these mechanisms behave like in most other RTOS.

If there is need to pass data between processes, then shared memory can be used to do so.

4.1.4.3 Interlocked mechanisms

Windows CE has also some InterLockedXxx API calls available to perform atomic operations like incrementing and decrementing a counter.

4.2 API richness

Windows CE 5.0 uses a subset of the Win32 API. This API provides the most commonly used kernel features.

It should be noted to the reader that the tables below only make an inventory of the kernel related APIs. The Win32 API includes a very large amount of interfaces that provide features that are beyond the scope of this study.

The tables below make an inventory of basic real-time objects and features present in the POSIX and OS proprietary interfaces.

While interpreting these results, the reader should keep in mind that these tables cover a strictly defined set of system calls only. As it is very hard to compare the different API for the different operating systems, no points are given for this section. The reader should use this section to check if the API calls he needs for his application are available or not.

In general, the more system calls are available, the easier it is for the programmer to find the correct call for it's application, without writing an own library with for instance wrapper calls…

4.2.1 Task Management

Thread management	YES
Get stack size	✓
Set stack size	✓
Get stack address	✓
Set stack address	-
Get thread state	✓
Set thread state	✓
Get TCB	✓[1]
Set TCB	✓
Get priority	✓
Set priority	✓
Get thread ID	✓
Thread state change handler	-
Get current stack pointer	✓

[1] This structure is called a thread context in Windows CE

Thread management	YES
Set thread CPU usage	-
Set scheduling mechanism	✓
Lock thread in memory	-[2]
Disable scheduling	✓

4.2.2 Clock and Timer

Clock	YES
Get time of day	✓
Set time of day	✓
Get resolution	✓
Set resolution	✓
Adjust time	✓
Read counter register	✓
Automatically adjust time	✓

Interval timer	YES
Timer expires on an absolute date	-
Timer expires on a relative date	✓
Timer expires cyclical	✓
Get remaining time	-
Get number of overruns	✓
Connect user routine	✓

4.2.3 Memory Management

Fixed block size partition	NO
Set partition size	-
Get partition size	-

[2] All threads can be locked in memory at configuration time. Executable modules can be loaded and locked into memory with the "LoadDriver" API

Fixed block size partition	NO
Set memory block size	-
Get memory block size	-
Specify partition location	-
Get memory block – blocking	-
Get memory block - non blocking	-
Get memory block - with timeout	-
Release memory block	-
Extend partition	-
Get number of free memory blocks	-
Lock/unlock partition in memory	-

Non-fixed block size pool	YES
Set pool size	✓
Get pool size	-
Make new pool	✓
Get memory block size	✓
Get memory block – blocking	-
Get memory block - non blocking	✓
Get memory block - with timeout	-
Release memory block	✓
Extend pool	✓[3]
Extend block	✓
Get remaining free bytes	-
Lock/unlock pool in memory	✓
Lock/unlock block in memory	✓

[3] The heap cannot be extended with an API, but the system automatically extends it when necessary.

4.2.4 Interrupt Handling

Interrupt handling	YES
Attach interrupt handler	✓
Detach interrupt handler	✓
Wait for interrupt – blocking	-
Wait for interrupt - with timeout	-
Raise interrupt	-
Disable/Enable hardware interrupts	✓
Mask/Unmask a hardware interrupt	-
Interrupt sharing	✓

4.2.5 Synchronization and Exclusion Objects

Counting semaphore	YES
Get maximum count	-
Set maximum count	✓
Set initial value	✓
Share between processes	✓
Wait - blocking	✓
Wait - non blocking	✓
Wait - with timeout	✓
Post	✓
Post – Broadcast	✓
Get status (value)	-

Binary semaphore	NO[4]
Set initial value	-
Share between processes	-

[4] Binary semaphores are not explicitly provided, but can easily be simulated by a counting semaphore with maximum count of one.

Binary semaphore	NO[4]
Wait - blocking	-
Wait - non blocking	-
Wait - with timeout	-
Post	-
Get status	-

Mutex	YES
Set initial value	✓
Share between processes	✓
Priority inversion avoidance mechanism	✓
Recursive getting	✓
Thread deletion safety	-
Wait – blocking	✓
Wait - non blocking	✓
Wait - with timeout	✓
Release	✓
Get status	-
Get owner's thread ID	-
Get blocked thread ID	-

Conditional variable	NO
Pend non blocking	-
Pend with timeout	-
Pend in fifo / priority order	-
Broadcast	-
Priority inversion	-

Event flags	YES
Set one at a time	✓
Set multiple	-

Event flags	YES
Pend on one	✓
Pend on multiple	✓
Pend with OR conditions	✓
Pend with AND conditions	✓
Pend with AND and OR conditions	-
Pend with timeout	✓

POSIX signals	NO
Install signal handler	-
Detach signal handler	-
Mask/unmask signals	-
Identify sender	-
Set destination ID	-
Set signal ID	-
Get signal ID	-
Signal thread	-
Queued signals	-

4.2.6 Communication and Message Passing Objects

Queue	YES
Set maximum size of message	-
Get maximum size of message	✓
Set size of queue	✓
Get size of queue	✓
Get number of messages in queue	✓
Share between processes	✓
Receive – blocking	✓
Receive – non blocking	✓
Receive – with timeout	✓

Queue	**YES**
Send - with ACK	✓
Send - with priority	✓
Send – OOB (out of band)	-
Send - with timeout	✓
Send – broadcast	-
Timestamp	✓
Notify	-

Mailbox	**NO**[5]
Set maximum message size	-
Get maximum message size	-
Share between processes	-
Send - with ACK	-
Send - with timeout	-
Send – broadcast	-
Receive – blocking	-
Receive – non blocking	-
Receive – with timeout	-
Get status	-

[5] A mailbox is in fact nothing more than a message queue that can store no more than one message. It is included for the sake of completeness, but most operating systems do not explicitly support it anymore.

4.3 Documentation

OS Documentation	0	▨▨▨▨▨▨ 6 □□□□	10

The documentation does not give a clear and structured overview. It is also lacking in-depth information about the inner workings of the system. Microsoft offers paid premier support for OEMs. Online support can be freely obtained from an extensive online knowledge base on the MSDN website.

Windows CE 5.0 comes with an online documentation set. This documentation set contains a lot of information, but it is not presented in a very structured way. The documentation can easily be used as a reference, but is less appropriate as a tutorial. Newcomers will have a hard time acquiring an overview of the system if this is the only documentation they have at their disposal. It also lacks in-depth information about the inner workings of the system. Documenting the APIs and available features is not enough to provide the reader with a sufficient understanding of a complex system like Windows CE 5.0.

Microsoft does provide an extensive knowledge base on its MSDN website, which can be freely consulted. Premier support for OEMs is available against payment.

4.4 OS Configuration

OS Configuration	0	▨	▨	▨	▨	▨	▨	7			10

4.4.1 OS boot options

The boot loader follows a semi-chronological order of tasks, but there are some options you can set yourself. It is possible to relocate the run-time image in RAM, the initial place is in flash. It is possible to enable MMU and caches, disabling them is possible by adjusting the config.bib file. Windows CE can obtain an IP address from a DHCP server, the alternative is to assign static IP addresses. For debugging purposes you can add a passive KITL connection which won't disturb the OS.

All the tasks of the boot loader are supported by libraries which make it easier to develop your boot loader.

There are 2 types of binary images that will be used by the boot loader: .bin file and the .nb0 file.

- The most common format for a CE image is the .bin file, it is optimized to minimize the amount of data that needs to be transferred to the device.
- The .nb0 file is useful to place the boot loader image on the target device, the format is a raw binary image of the boot loader. Most of the time the board manufacturer provides a program to do this, but alternatively you can do the same thing through a JTAG connection.

4.4.2 OS configuration options

The next step is to use the platform builder to create, customize and configure a platform. Configuring the platform to your requirements is a complicated and intricate process.

Although the platform builder integrated development environment (IDE) includes wizards for creating platforms and components, most of the configuration work will happen through manually editing registry files, manipulating a set of environment variables and modifying various other configuration scripts. This makes the configuration process a difficult task to newcomers.

Some useful options in the build process are : CE Target Control support, Eboot Space in Memory, Full Kernel Mode, KITL connections. Just checking the boxes in the build options tab will enable or disable the options.

We were under the impression that Platform Builder 5.0 has made some progress compared to previous versions in terms of ease of configuration.

4.4.3 BSPs

For each kind of platform windows CE provides a BSP, it is possible to create new ones yourself, clone a BSP or modify a BSP. The option to create global drivers for all BSPs is an advantage.

The following BSPs, which have been reworked to improve their quality, are included in this release:

- Samsung SMDK-2410
- AMD DBAU1000
- AMD DBAU1100
- AMD DBAU1500
- NEC Solution Gear 2 Vr4131
- NEC Solution Gear 2 Vr5500

It is possible to create new BSP's, add boot a loader or drivers to the BSP,… All this can be done with the wizards of the Microsoft Platform builder IDE tools:

- Clone an existing BSP
- Create a new BSP
- Use your new BSP
- Export a BSP for use by OEM Customers
- Import third party BSP into the catalog

The difficulty in this process is that you need to configure you run-time image configuration files so that they use the new BSP. By changing these files you can easily make mistakes and create the possibility that the build process is disturbed by your changes. Errors will occur but they often don't give enough information to know what's the real problem, searching after these problems is a complicated and time consuming process. Reinstallation is most of the time the best solution, which is not a real problem because your workspace files will stay intact.

4.4.4 Device drivers

Many Windows CE 5.0 device drivers have been reviewed and reworked to improve their quality.

Windows CE Device Drivers are trusted modules, but don't need to run in kernel mode. Many services are provided by the OS to develop an interface that suites the device.

There are 3 kinds of processes that load drivers : (ref : Platform builder help) '

Process	***Drivers***
File System (FileSys.exe)	*FileSys.exe loads file system drivers. For more information, see File Systems.*

Device Manager (Device.exe)	*Device.exe loads audio drivers, battery drivers, keyboard drivers, mouse drivers, NDIS drivers, notification LED drivers, serial drivers, PC Card drivers, USB drivers, and any other driver that exposes the stream interface. Device.exe loads most of its drivers with ActivateDeviceEx, and these drivers expose a stream interface. For more information, see Device Manager.*
Graphics, Windowing, and Events Subsystem (GWES.exe)	*GWES.exe loads a device driver if GWES is the only client of a driver. Device drivers loaded by GWES present a standard set of functionality for all similar devices. Drivers that GWES loads may expose the stream interface, but they also may expose different interfaces. This makes accessing the drivers much faster. GWES loads display drivers, printer drivers, and touch screen drivers. For more information, see Shell and User Interface Overview.*

4.5 Internet components

Internet components	0	▨▨▨▨▨▨▨▨▨□	10

Windows CE 5.0 has very extensive Internet support

The internet support is divided in modules and components. These can be found in a structured way among the other catalog items, just clicking to add to your OS design. Adding catalog items is pretty straightforward, but if you use design templates who already have many components then is the customization much more complicated then just removing and adding the components. Most of the time the components are linked and can't be removed before the dependencies of that component are removed too.

The most supporting OS Template for internet components is The Enterprise Web pad design. This configuration supports internet-based applications, including communications and networking functionality's. For more information about the security issues that can affect this template design, see ms-help://MS.WindowsCE.500/wceosdev5/html/wce50conEnterpriseWebPadConfiguration.htm

Windows CE 5.0 comes with an extensive set of Internet products and tools. It includes, among others:

– An HTTP server to post information. The server supports active server pages, ISAPI extensions and filters.

– A web browser for viewing information. The browser is a miniature version of Internet Explorer. It supports frames, tables and Javascript, as well as JPEG, static and animated GIF and WAV files.

– A telnet server to remotely administer devices, or to administer devices that do not have displays.

– Networking protocol support for communicating across the internet/intranet.

Many other tools and utilities are available from third-party vendors.

4.6 Development tools

Development tools	0								8			10

Platform Builder 5.0 has improved over its predecessor in terms of user friendliness. However, the product still contains some unpleasant bugs.

A complete integrated development environment is available on the (Windows based) host, while the real-time Windows CE .NET application runs on the target. Platform builder is available for the platform developers, while other (independent) tools can be used for application development. Platform developers can export an SDK for the application developers.

Microsoft provides development tools to cater the need of two categories of developers: the platform developers and the application developers.

Platform developers use an integrated development environment called Platform Builder on the Windows based host, while the target runs Windows CE 5.0 with the (real-time) applications. Aside from this platform development tool, Microsoft also provides the Embedded Visual Tools for application development.

The Platform Builder can be used to create a custom SDK (Software Development Kit) based on the Windows CE 5.0 OS to allow developers to write applications that run on the target platform. An SDK is a set of library, header, and Help files that developers use to write applications for a specific platform. The SDK is used in conjunction with the Embedded Visual Tools to create, debug and run custom applications.

😐 Platform Builder 5.0 (PB) has a few new features that make configuring a Windows CE 5.0 image somewhat easier. A new platform wizard that assists you while creating a new platform is one example. Still, we encountered some very annoying bugs in the tool. Quite often the PB crashes when one disconnects the host from the target. On one occasion, the crash apparently destroyed some key files on our host that rendered it inoperable. Our platform builder needed to be reinstalled for this reason.

PB now uses dirs and sources to sysgen your platform, the makefiles also still exist but are less current.

The table below makes an inventory of the most commonly used tools and features available in PB 5.0.

	Present (Yes/No)	Integrated in IDE	Standalone	Command based	GUI based
Editor					
Color highlight	Yes	✓			✓
Integrated help	Yes	✓			✓

	Present (Yes/No)	Integrated in IDE	Standalone	Command based	GUI based
Automatic code layout	No				
Compiler					
C	YES	✓	✓	✓	✓
C++	YES	✓	✓	✓	✓
Ada	No				
Assembler	Yes	✓	✓	✓	✓
Linker					
Incremental	Yes	✓	✓	✓	✓
Symbol table generation	Yes	✓	✓	✓	✓
Development tools					
Profiler	Yes		✓	✓	
Project management	Yes	✓			✓
Source code control	Yes	✓			✓
Revision control	Yes	✓			✓
Debugger					
Symbolic debugger	Yes	✓			✓
Thread sensitive debugging	Yes	✓			✓
Mixed source and disassembly	Yes	✓			✓
Variable inspect	Yes	✓			✓
Structure inspect	Yes	✓			✓
Memory inspect	Yes	✓			✓
Register inspect	Yes	✓			✓
Target connection					
JTAG	TBD				

	Present (Yes/No)	Integrated in IDE	Standalone	Command based	GUI based
BDM	TBD				
Serial (via ROM-Monitor or app?)	Yes				✓
Network (via ROM-Monitor or app?)	Yes				✓
System analysis tool					
Tracing	Yes	✓	✓		✓
Thread information	Yes		✓		✓
Interrupt information	Yes		✓		✓
Loader					
TFTP boot loader	Yes	✓	✓		✓
Serial boot loader	Yes	✓	✓		✓
Load separate modules	Yes	✓			✓

5 Appendix A: Vendor comments

6 Appendix B: Acronyms

Acronym	Explanation
API	Application Programmers Interface: calls used to call code from a library or system.
BSP	Board Support Package: all code and device drivers to get the OS running on a certain board
DSP	Digital Signal Processor
FIFO	First In First Out: a queuing rule
GPOS	General Purpose Operating System
GUI	Graphical User Interface
IDE	Integrated Development Environment (GUI tool used to develop and debug applications)
IRQ	Interrupt Request
ISR	Interrupt Servicing Routine
MMU	Memory Management Unit
OS	Operating System
PCI	Peripheral Component Interconnect: bus to connect devices, used in all PCs!
PIC	Programmable Interrupt Controller
PMC	PCI Mezzanine Card
PrPMC	Processor PMC: a PMC with the processor
RTOS	Real-Time Operating System
SDK	Software Development Kit
SoC	System on a Chip

7 Appendix C: Document revision history

7.1 Issue 1.0 (April 19, 2004)

Initial version

EVALUATION REPORT DEFINITION

http://www.dedicated-systems.com

Email: info@dedicated-systems.com

DOCUMENT CHANGE LOG

Issue No.	Revised Issue Date	Para's / Pages Affected	Reason for Change
1	April 29, 2004	All	Initial Issue

1 Introduction

1.1 Purpose and scope

This paper explains the evaluation tests done and how the results are presented in the RTOS evaluation report. It explains how framework 2.9 describes the tested real-time OS and shows how detailed the results are. This document is an important companion document in reading and understanding the evaluation reports.

The evaluation report itself shows the test results obtained for a particular RTOS. The executed tests are explained here. Due to the tightly coupling between these documents, the issue of this document has to match the issue of the evaluation report. Information on how we handle versions of documents and tests can be found in “The evaluation framework.”, see section 1.3 of this document.

This document defines and explains all tests executed in the framework 2.9. Generic pseudo “C” code for these tests is available in another document which can be obtained on request. The results of the test will indicate if a system may or may not be qualified as “real-time”. For each test, objective criteria are set to determine if a result is real-time or not. (real-time is used here in the sense of “predictable response time and behavior”

1.2 Document issue: the 2.9 framework

Framework 2.9 is a serious revision of the previous framework to achieve:

– Better readability
– More objective qualification criteria

These are the main changes between the current (2.9) and the previous (2.5) framework:

– Test labeling is completely changed to improve the readability of the evaluation reports.
  Remark that in appendix, a lookup table is added to find the relation between the test labels used in the previous framework and the current framework.
– Separation of the RTOS memory model and the test labeling, again to improve readability.
– Generic pseudo code is made for each test.
  This code is based on the “C” programming language and uses macros for the RTOS dependent system calls. (Published in a separate document).
– Added a new test category to test the RTOS “Behavior”.
  These tests check if the operating system acts as expected in a certain scenario we define. These scenarios correspond to classical scenarios one would use in an application.
– Added some objective criteria to differentiate between “real-time” and “non real-time”.

Generic pseudo code is now available for download on request. Third parties can therefore more easily repeat the tests presented here. However, these third parties should also be aware of the fact that one needs extensive and costly measuring equipment to do the timing measurements in order for these measurements not to impact on the behavior of the system!

1.3 Related documents

These are documents that are closely related to this document. They can all be downloaded using following link:
http://www.dedicated-systems.com/encyc/buyersguide/rtos/evaluations

Doc. 1 The evaluation framework.
This document presents the evaluation framework. It also indicates which documents are available, and how their name giving, numbering and versioning are related. This document is the base document of the evaluation framework.
EVA-2.9-GEN-01 Issue: 1 Date: April 29, 2004

Doc. 2 What is a good RTOS?
This document presents the criteria that Dedicated Systems Experts use to give an operating system the label “Real-Time”. The evaluation tests are based upon the criteria defined in this document.
EVA-2.9-GEN-02 Issue: 1 Date: TBD

2 The evaluation report

2.1 Introduction

2.1.1 The evaluation framework

The evaluation framework is introduced in [Doc. 1]. This document also describes the aim of the evaluation reports. Our concept of real-time is introduced there. More ideas and concepts used in this document are explained in the paper "What is a good RTOS?", see [Doc. 2] in section 1.3 of this document.

The tests are designed in order to verify if:

– A system behaves like expected

– The system response is predictable in all circumstances (worst case time needed?)

– A system does not have buggy behavior or instable behavior.

Depending on the results of the tests a system is labeled "RT-VALIDATED", "VALIDATED" or "DID NOT QUALIFY". Logos are available for each of the qualifications.

2.1.2 The two evaluation tracks

Our evaluation is divided into two tracks:

- The first track is a qualitative study called the technical evaluation. In this approach, we focus on the system architecture of the operating system and/or the architecture of the platform.

- The second track is referred as the practical evaluation and is a quantitative approach. This second track is detailed in depth in this document.

2.1.2.1 Qualitative track: The technical evaluation

The two technical evaluations closely related with a test report are:

– The operating system evaluation

– The platform evaluation.

Both evaluations are merely theoretical. The content and layout of these reports are given in two other documents, but these are not mandatory to understand the evaluation reports in this case.

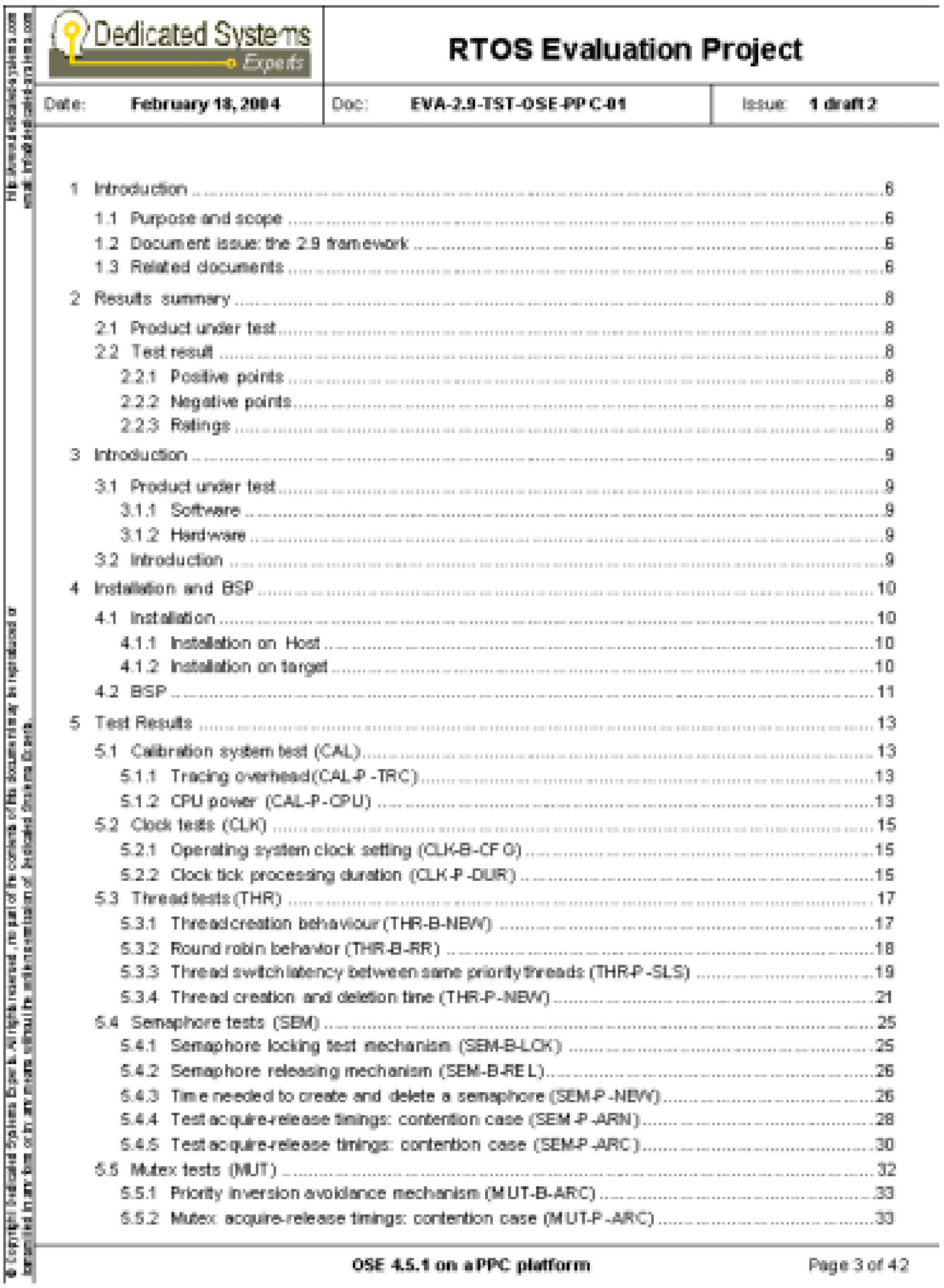
2.1.2.2 Quantitative track: The practical evaluation

The practical evaluation measures specific real-time features of the operating system. It does not certify that a set of application threads will meet their deadlines; rate monotonic scheduling and similar methods can be used for this purpose. The purpose is to assert that a RTOS is suitable or not for real-time applications. If the behavior of an application needs to be predictable, then the underlying software, i.e. the RTOS, needs to have all the features necessary to meet these requirements. In general, all the system calls and operations of an operating system should exhibit predictable behavior. This implies that the execution time has to be bounded, independent of the workload of the system.

The purpose of our test suite is not only to measure the throughput and the responsiveness of the RTOS, but also to test its determinism and behavior, much more important than performance figures.

2.2 The evaluation report layout

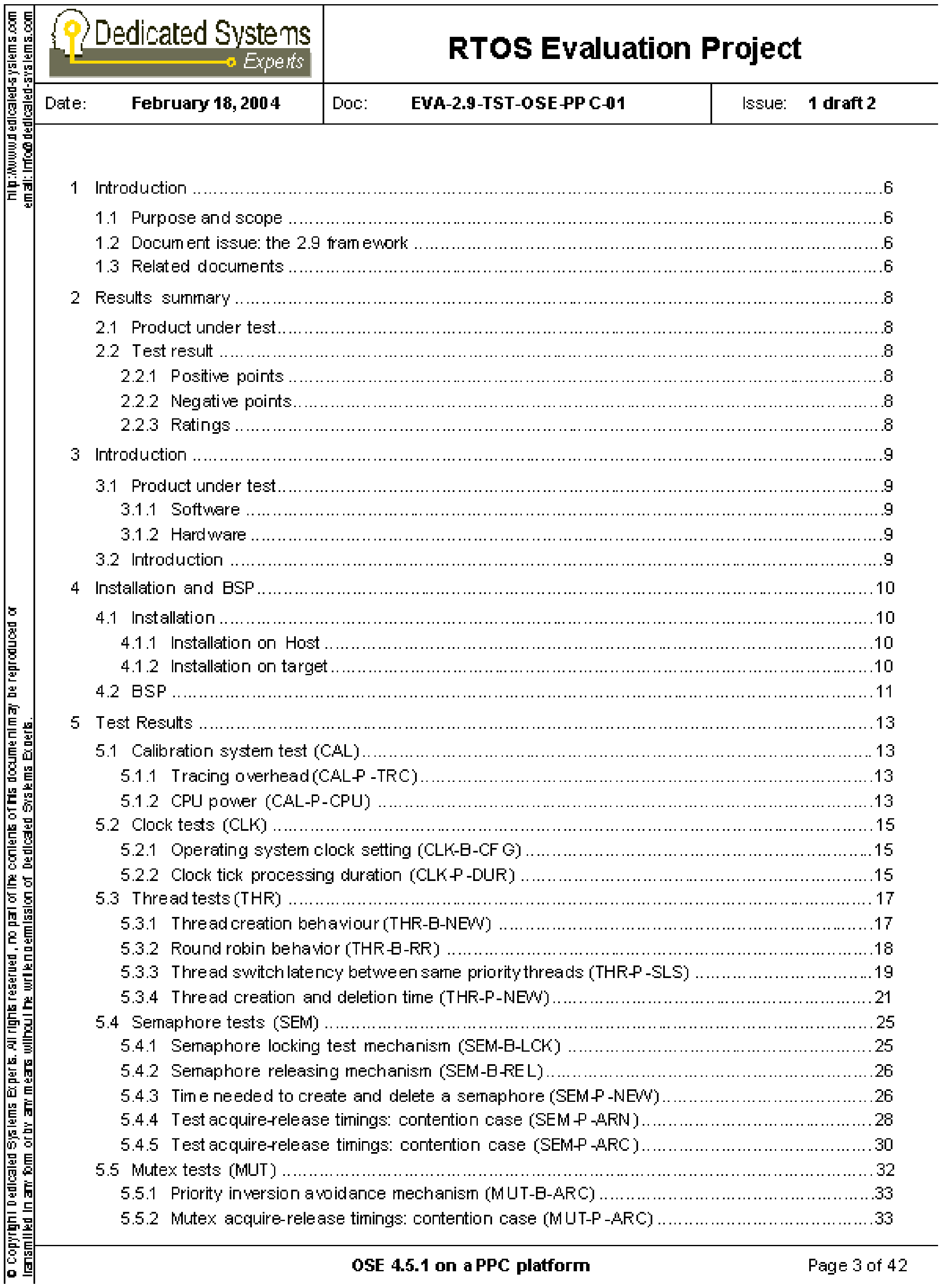
In the figure below a sample of the table of contents of an evaluation report is shown.



RTOS Evaluation Project

Date: February 18, 2004 | Doc: EVA-2.9-TST-OSE-PPC-01 | Issue: 1 draft 2

OSE 4.5.1 on a PPC platform   Page 3 of 42



RTOS Evaluation Project

Date: February 18, 2004 | Doc: EVA-2.9-TST-OSE-PPC-01 | Issue: 1 draft 2

http://www.dedicated-systems.com
email: info@dedicated-systems.com

© Copyright Dedicated Systems Experts. All rights reserved, no part of the contents of this document may be reproduced or transmitted in any form or by any means without the written permission of Dedicated Systems Experts.

OSE 4.5.1 on a PPC platform   Page 3 of 42

Figure 1 Content table of example test report

2.3 Measurement method

2.3.1 Tracing PCI access cycles

An absolute time reference is required to measure time intervals. Most operating systems include software timers that could be used as a measurement tool. But timers in different operating systems do not necessarily have the same resolution or precision to perform accurate measurements. The use of software timers adds unpredictable overhead to the results because the measurements are performed by the operating system while it executes the test. The use of it may also change the behavior of the system. Furthermore, the measurements based on the software timers are synchronous with the system clock and this is not what we are looking for. Instead of a software timer, DS-Experts uses an external hardware device: a PCI bus analyzer.

During the execution of a test, tracing data is written at a valid PCI bus address before and after the evaluated system operation. Writing the trace to a double word aligned address on a 33MHz PCI bus takes on most platforms six to seven bus-cycles, i.e. 180ns to 210ns (this largely depends on the platform used). The trace is specific to its position in the code, making it possible to identify and follow the execution path of the test during the analysis of the results.

The PCI bus analyzer stores the data written on the PCI bus into its local memory and timestamps it. When the test is completed, the data is downloaded from the PCI bus analyzer to a PC where it can be further processed.

Tracing is done in the test code by issuing *TraceWriteXxx()* calls. There are four types of traces written on the PCI bus:

– A trace to indicate the start of a timing measurement: *TraceWriteBefore*.

– A trace to indicate the end of a timing measurement: *TraceWriteAfter*.

– A trace to check the behavior of a test (state analysis): *TraceWriteData*.

– A trace to indicate an error condition in the generic to RTOS dependent library: *TraceWriteError*

Figure 2 shows the generic performance measurement using the two timing traces: the time difference between the traces “before” and “after” written on the PCI bus gives the execution time for the system operation. As the collected data marks the start and the end of a system operation, it can be used to calculate the desired time intervals. Each test loops until the trace buffer of the PCI analyzer is full (currently this is 32K trace samples or about 16K time samples).

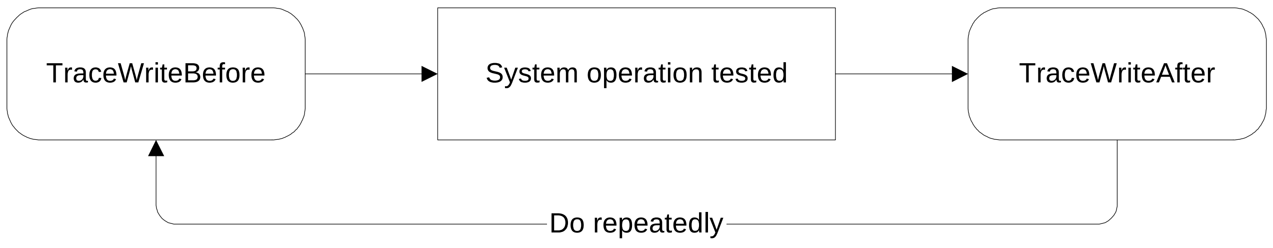


Figure 2 generic performance measurements

2.3.2 State analysis

The state traces written during any test performed have a specific identification indicating the state transition that will take place.

Therefore it is possible to do a state analysis on the captured trace: does the system behave as expected? Dedicated Systems Experts designed a tool to verify the captured trace files and to detect anomalies in the expected behavior. This tool improves the verification of a system: such errors would be difficult to notice without. Remark that the trace files can indeed be as large as ten thousands of samples making it almost impossible by hand to detect a fault that only occurs once in a while.

Sometimes multiple valid state changes are possible depending on the features of the system being tested. So the state checker does not only check for a valid trace, it can also determine if the trace is the one expected for a real-time system or not.

Furthermore, the tests are designed to detect invalid state changes and generate an error trace when such a condition is met.

2.3.3 Statistical analysis

The captured trace file, once validated by the state checker, is passed through a statistical program, which generates the diagrams and the “minimum, average and maximum” duration of a certain action.

These values and diagrams are shown in the test evaluation reports.

2.3.4 Generating interrupts

For the interrupt testing series, we use a PCI bus exerciser to generate interrupts at programmable intervals on the PCI bus. To test simultaneous interrupt behavior, two such exercisers are used.

It is important to note that the interrupts are generated by the firmware in the exerciser, this means completely independent of the system under test. This is an extremely important requirement otherwise interrupt generation will always be linked in a way or another to the operating system clock.

2.4 System configuration parameters

Here we discuss the parameters that remain constant during all tests. They depend on the system configuration. The most important setting concerns the protection model used by the operating system.

2.4.1 Memory Protection model

Memory configurations of an operating system can be classified into the following protection models each being an enhancement of the precedent (see figure 3):

– **No memory protection model**: flat memory layout without memory protection between the threads in the system. This model does not require an MMU.

– **System/User memory protection model**: the system address space is protected from the user address space. User processes and system processes run in a common virtual address space. This model requires an MMU.

– **User/user protection model**: adds protection between user processes to the system/user model. This model requires an MMU.

Independent of the memory protection model, it is also possible to have virtual addressing by the MMU. In this case the physical addresses are mapped onto some virtual address space.

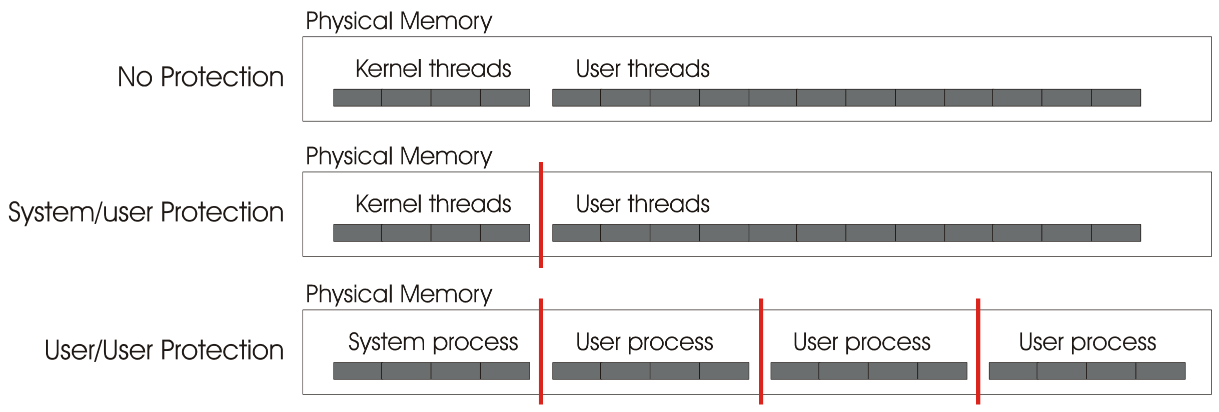


Figure 3 Memory models

Besides the memory protection model, most CPU's have also the notion of privileged CPU instructions that can only be called in some processor mode. This is used to restrict some functionality outside the kernel.

If an OS supports more than one model, some or all tests are redone for the different configurations. In such case, there will be multiple "test results" chapters in the evaluation report: one for each configuration. The same protection configuration is used for all tests published in one chapter.

3 Testing overview

This section of the document describes in detail how tests are identified, how they are compiled and the generic system calls used in the generic test code.

3.1 Naming of the test series

The naming of the different test series is based on multiple identifiers separated with an underscore. We can express this in the Backus-Naur Form (BNF), a formal meta-syntax used to express context-free grammars. A short introduction is given here below.

3.1.1 The Backus-Naur Form

A definition of the symbols used in the BNF notation is shown here:

– “HELLO”	“”	indicates a string exactly as it should appear
– “A” | “B”	|	indicates a choice (or)
– <name>	<>	indicates a definition (type), with the name: name
– <space>::= “ ”	::=	indicates an equality (the definition <space> is equal “ ”)
– # comment	#	indicates start of comment (until end of line)

We give an example of a BNF syntax definition and possible valid syntax:

```
<bit>                 ::=  “0” | “1”
<binary value>        ::=  <bit> | <bit><binary value>        # recursive example!
```

Valid syntax for <binary value> are then: “001101”, “1”, “110”, ...

3.1.2 Test identifiers

Here we define the BNF used for the test identifiers:

```
<test identifier>        ::=  <main test id> | <main test id> “_” <optional parameters>
<main test id>           ::=  <tested object name> “_” <test class> “_” <test name>
<tested object name>     ::=  # one of the entries as shown in the table below
<test class>             ::=  # one of the entries as shown in the table below
<test name>              ::=  # defined in a table for each tested object name
<optional parameters>    ::=  <parameter> | <parameter> “_” <optional parameters>
<parameter>              ::=  # for each test, the description of the optional parameters
```

An example would be “THR_P_NEW”: this would be a performance (P) test on threads (THR), more specific it would test the creation (NEW) time of a thread.

All tests are grouped by object type. We believe that this is the most logical way of ordering the tests.

In the tables below, we define the valid <tested object name> and <test class> used in this document.

<test object name>	description: test related to
CAL	Calibration test: test used in order to calibrate the platform
CLK	Operating system's internal clock
THR	Threads (in same user space)
SEM	Semaphores
MUT	Mutex = mutual exclusion semaphore. In the scope of the evaluations we differentiate the mutex from a semaphore when a priority inheritance mechanism is involved.
IRQ	Interrupts

<test class>	description
B	Test Behavioral issues
P	Test the Performance of something
S	Stress testing: test the behavior under heavy load conditions

3.1.3 Diagram identifiers

The test result diagram will have the same identifier, but may have an extra parameter if multiple measurements are performed for the same test.

Here we define the name giving for each diagram:

```
<diagram identifier>     ::=  <test identifier> | < test identifier > "_" <measurement type>
<measurement type>       ::=  # for some tests a measurement type may be defined
```

An example would be "THR_P_NEW_DEL": this would be a performance (P) test on threads (THR), more specific it would test the creation/deletion (NEW) time of a thread: results shown in the diagram are the ones of the deletion (DEL) time.

3.1.4 Source code identifier

The source code related with a certain test has the same file name as the test identifier. If the test has optional parameters, the same source code will be compiled (by use of a make tool) in different test executables depending on the optional parameters.

We can illustrate this with an example:

- Say we perform a test with name "X_Y_Z"
- Say that this test has one parameter, a loop quantity "PAR_LOOP"
- Say that we perform this test with the "PAR_LOOP" parameter set to the values 1, 10 and 128 respectively

The generic source code file would have the name “X_Y_Z.c”. An example of such code file is shown below:

```
void TestEntry(void)
{
   int iLoopCounter = PAR_LOOP;

   while (iLoopCounter > 0)
   {
      DoTheTest();
      iLoopCounter--;
   }
}
```

It is then possible to change the parameter by using compiler command line parameters. All “C” compilers support a command parameter to add a preprocessor macro name, like shown here: “-Dname=value”. This is then used in the make file to differentiate the compilation for different parameter settings.

An example of the make description file for the test above could be:

```
X_Y_Z_1:    X_Y_Z.c
            $(CC) -DPAR_LOOP=1   -o X_Y_Z_1   X_Y_Z.c

X_Y_Z_10:   X_Y_Z.c
            $(CC) -DPAR_LOOP=10  -o X_Y_Z_10  X_Y_Z.c

X_Y_Z_128:  X_Y_Z.c
            $(CC) -DPAR_LOOP=128 -o X_Y_Z_128 X_Y_Z.c
```

The test executables generated in the directory would then be

```
> ls
X_Y_Z_1
X_Y_Z_10
X_Y_Z_128
```

The results of these tests would also have these labels in the test report.

This approach makes it easy to add extra tests with other parameter settings without writing any code. It guaranties that the same code is used again. If these scenarios would be stored in different source files, it would be almost impossible to guarantee consistent code.

The same system may be used for issuing a test in different scenarios. Then the parameter can generate different executables by using the #if, #else, #elif and #endif preprocessor constructions.

We can illustrate this again with an example:

– Say we perform a test with name “X_Y_Z”
– Say that this test has different scenarios set by the “SCENARIO” parameter
– Say that we perform this test with two scenarios: “SC1” and “SC2”

The generic source code file would have the name “X_Y_Z.c”. An example of such code file is shown below:

```
/* test scenarios: if invalid used, compiler errors will occur */
#if   SCENARIO==1
# define DoTheTest               Test1
#elif SCENARIO==2
# define DoTheTest               Test2
#endif

void TestEntry(void)
{
   /* init depends on test scenario */
   DoTheTest();
}

void Test1(void)
{
   …
}

void Test2(void)
{
   …
}
```

It is then possible to change the scenario by using again some compiler command line parameters. This is used in the make file to differentiate the compilation for the different scenarios.

An example of a make description file for the test above could be:

```
X_Y_Z_SC1:  X_Y_Z.c
            $(CC) –DSCENARIO=1   -o X_Y_Z_SC1   X_Y_Z.c

X_Y_Z_SC2:  X_Y_Z.c
            $(CC) –DSCENARIO=2   -o X_Y_Z_SC2   X_Y_Z.c
```

The test executables generated in the directory would be

```
> ls
X_Y_Z_SC1
X_Y_Z_SC2
```

The results of these tests would also have these labels in the test report.

Using scenarios makes only sense when the code differences between each scenario are minimal. Otherwise another test name will be used instead.

3.2 Coding style

A coding style is used to clearly identify functions, type definitions and variables and to describe the layout of bracketing and indenting. This should considerably enhance the readability of the code.

Remark that we use strict ANSI C for coding. As a consequence, comments are always written using the "/**/" construct and not the "//" construct.

3.2.1 Identifiers

3.2.1.1 Functions

Functions are written using concatenated words starting with an uppercase for each word, the rest of the characters are written in lower case.

Remark that some of the functions used may also be macro definitions. There is no syntax difference between the two types.

Example:

```
/* write a state trace */
TraceWriteData(5);
```

3.2.1.2 Type definitions

These are written just like the functions as defined before, but a character "t" is added in front of definition.

Example:

```
/* data needed for os dependent part of mutex */
typedef struct tOsMutexData
{
   int *ipMutex;    /* the mutex */
} tOsMutexData;
```

3.2.1.3 Variable declarations

Written just like the functions defined before, but prefix are added in front to indicate the type of the variable. The examples below show how this is done.

Examples:

```
int     iCounter;                          /* an integer */
int    *ipCounter;                         /* a pointer to an integer */
int     iaCounters[10];                    /* an array of integers */
int    *ipaCounters[10];                   /* an array of pointers to integers */

tData  *tpData;                            /* pointer to a type defined structure */

char    cChar;                             /* a character */
unsigned int uiCounter;                    /* an unsigned int */
void   *vpDummy;                           /* a void pointer */
```

So the prefix does not only explains the type, but also how it is used (as pointer, array etc …)

3.2.1.4 Pre-processor constants

These are written in all uppercase, where each word is separated by an underscore. Prefixes are used to show the origin of the constant.

Examples:

```
TRACE_BUFFER_SIZE                              /* size of the trace buffer in samples */
PAR_QTY_LOOP                                   /* number of loops to perform */
```

3.2.2 Bracing styles

In the code the bracing style as shown in the example will always be used.

Examples:

```
/* bracing in a if/else scenario */
if (iX != 0)
{
   Function1();
}
else
{
   Function2();
}

/* bracing in a switch/case scenario */
switch(iX)
{
case 1:
   Function1();

case 2:
   Function2();

default:
   FunctionDefault();
}
```

3.2.3 Indenting

Braces are used to separate code blocks. Each deeper level, the code will be indent with three white spaces.

So no tabs will be used (as this depends on the editor) and a fixed font has to be used (so that each letter has the same width).

3.2.4 Code blocks and comments

Comments will be placed before the code block or code lines it comments. There will be a blank line before the comment and no blank line after the comment.

A code block (separated by braces) can be used to comment a larger part of the code.

Example:

```
/* handle this */
HandleThis();

/* this block of code does something */
{
   /* do the initialisation */
   InitIt();

   /* now do the real thing */
   DoIt();
}
```

A comment may be put on the same line where an identifier is defined.

3.3 Libraries

As we want to re-use the same generic test code again, independent of the platform under test, libraries are made.

These libraries abstract the tracing system and the operating system. As such the API used in the generic code remains the same, but the implementation may differ.

The different generic API calls used are explained in this section.

3.3.1 Tracing API

All tracing API calls use the "Trace" prefix.

Following constants are defined:

– *TRACE_BUFFER_SIZE*
  Number of samples that can be stored in the trace buffer.

Following API calls are defined:

– *void TraceInitialise(void)*
  This call will initialise the tracing system, it has to be called before any other Trace API call.

– *void TraceCleanUp(void)*
  This call will clean up the tracing system first initialised with TraceInitialise. It has to be called at the end of the test.

– *void TraceWriteData(int iState)*
  This call has to be used for tracing state data. This data can then be analysed by the state analyser to detect invalid behaviour.

  – *int iState*
    Number used to identify the current state.

– *void TraceWriteBefore(int iTimer)*
  This call has to be used for starting a time measurement. Multiple such calls may be used for multiple time measurements at the same time. Therefore the iTimer parameter is used.

  – *int iTimer*
    The identifier of the timer to start.

– *void TraceWriteAfter(int iTimer)*
  This call has to be used to stop a time measurement which was started with the TraceWriteBefore call. Multiple such calls may be used for multiple time measurements at the same time. Therefore the iTimer parameter is used.

  – *int iTimer*
    The identifier of the timer to stop.

– *void TraceWriteError(int iErrno)*
  This call is used to signal an erroneous condition whenever detected. It is also used within the libraries. The trace file analyzer will always detect such traces and declare the test result as invalid.

  – *int iErrno*

An identifier signalling the error type.

3.3.2 Interrupt generating API

Interrupts are generated by PCI boards (P-Drive) containing their own processor. On the P-Drive some software is running that may generate PCI interrupts. The test application starts the interrupt generation process by sending a command to the P-Drive (using a PCI mailbox register).

In the current test environment, it is possible to activate two P-Drives for testing interrupt prioritisation.

All the interrupt generating API calls use “IrqGenX” as prefix, where X can be 1 or 2 for the two P-drives used:

- *void IrqGen1Initialise(void)*
  This will initialise the PCI card for generating interrupts.
- *void IrqGen1Start(void)*
  Will start the interrupt generating process.
- *void IrqGen1Disable(void)*
  Will stop the interrupt generating process.
- *void IrqGen1AckInterrupt(void)*
  Will acknowledge the interrupt and clear the interrupt source.
- *int IrqGen1GetVector(void)*
  This will return the interrupt vector in the operating system used for this device.

Above calls exist also with the “IrqGen2” prefix.

3.3.3 Generic operating system API

These API calls abstract the operating system used, otherwise it would not be possible to write portable generic testing code.

All operating system API calls start with the “Os” prefix followed by the object related to the call. For instance, all thread related calls shall have the prefix “OsThread”.

As not only the API calls may differ between the different operating systems, but also the object data, the operating system object data will be hidden in a type definition with the name “tOsObjData”.

To avoid complex handling in the library that would delay the effective system-call and generate measurement overhead, most of these API calls will be macros. In general, the complex system calls (like creating and starting a thread) will be split in two parts:

- An “OsObjCreate” call, that will be used to set-up an operating system dependent structure. This call will be handled by a real function and may take some time.
- An “OsObjXxx” call, that will be a macro using the operating system dependent structure already set-up by the previous call so the extra overhead will be minimum.

Both the “OsObjCreate” and “OsObjDelete” calls are used only for this purpose and are never measured in the scope of this framework.

In the next sections the API calls for each object type are explained in detail.

3.3.3.1 Threads

All API calls concerned with this library use the “OsThread” prefix

Following data structure is defined:

– *tOsThreadData*

  A data structure containing all operating system dependent data for accessing threads and the related data. This is used to hide the operating system internals.

Following constants are defined:

– *OS_PRIORITY_HIGHEST*

  Highest priority an application thread can use in the system.

– *OS_PRIORITY_HIGH*

– *OS_PRIORITY_MIDDLE*

– *OS_PRIORITY_LOW*

– *OS_PRIORITY_LOWEST*

  Lowest priority an application thread can use in the system.

  Remark that the exact priority is of no importance, only the RELATIVE priority is relevant.

Following API calls are defined:

– *int OsPriorityIncrement(int iPriority)*

  This call is used to increment the priority: increment means in this context to higher the priority level, which is not the same as setting the priority variable to a higher number!

  – *return: int*

    The incremented priority.

  – *int iPriority*

    The priority to increment.

– *int OsPriorityDecrement(int iPriority)*

  This call is used to decrement the priority.

  – *return: int*

    The decremented priority.

  – *int iPriority*

    The priority to decrement.

– *void OsThreadSetMyPriority(int iPriority)*

  This call is used to change the priority of the current active thread.

  – *int iPriority*

    The new priority of the calling thread after executing this system call.

– *tOsThreadData *OsThreadCreate(int iPriority, void (*Function)(void*), void *vpArgument)*

  This call is used to create the operating system dependent data for starting, stopping the thread. Remark that in scope of this framework priority based scheduling will always be used.

  – *return: tOsThreadData**

    The operating system dependent data.

- *int iPriority*

  The default priority that the thread of execution will have when started.
- *void (*Function)(void*)*

  The entry function called when the thread starts.
- *void *vpArgument*

  The argument passed to the entry function of the thread when the thread starts. Remark that some operating systems do not have an ability to pass arguments to a thread. In such case a more complex library will be needed to pass arguments to a starting thread. Without arguments our generic tests cannot run.

- *void OsThreadStart(tOsThreadData *tpThread)*

  This call is used to start the execution thread already initialised by the OsThreadCreate call.

  - *tOsThreadData* tpThread*

    The operating system dependent thread data for the thread to start.

- *void OsThreadStop(tOsThreadData *tpThread)*

  This call is used to stop the execution thread started by the OsThreadStart call.

  - *tOsThreadData* tpThread*

    The operating system dependent thread data for the thread to stop.

- *void OsThreadDelete(tOsThreadData *tpThread)*

  This call is used to clean up the structure allocated and initialised by the OsThreadCreate call.

  - *tOsThreadData* tpThread*

    The operating system dependent thread data to clean up. After this call, the thread data will be invalid.

- *void OsThreadYield (void)*

  This call is used to yield the CPU to another ready thread at the same priority level (if any).

- *void OsThreadSleepSeconds(int iSeconds)*

  This call is used to delay the current active thread for some seconds (non-busy blocking wait).

  - *int iSeconds*

    The number of seconds the executing thread should wait.

- *void OsThreadSleepOneTick(void)*

  This call is used to delay the current active thread until next clock tick (block until next OS tick).

3.3.3.2 Semaphores

All API calls concerned with this library use the “OsSem” prefix

Following data structure is defined:

- *tOsSemData*

  Data structure containing operating system dependent data for accessing semaphores and related data. This structure is used to hide operating system internals.

Following API calls are defined:

- *tOsSemData *OsSemCreate(unsigned int uiInitialCount)*

  This call is used to create the operating system dependent data.

- *return: tOsSemData**
  Operating system dependent data.
- *unsigned int uiInitialCount*
  Initial count of the semaphore.

– *void OsSemInit(tOsSemData *tpSemaphore)*
  This call is used to initialise the semaphore with its initial value.
  - *tOsSemData *tpSemaphore*
    Operating system dependent data.

– *void OsSemP(tOsSemData *tpSemaphore)*
  This call is used to try and take the semaphore.
  - *tOsSemData *tpSemaphore*
    Operating system dependent data.

– *void OsSemV(tOsSemData *tpSemaphore)*
  This call is used to release the semaphore.
  - *tOsSemData *tpSemaphore*
    Operating system dependent data.

– *void OsSemDestroy (tOsSemData *tpSemaphore)*
  This call is used to destroy the operating system semaphore initialised by the OsSemInit call.
  - *tOsSemData *tpSemaphore*
    The semaphore to destroy.

– *void OsSemDelete (tOsSemData *tpSemaphore)*
  This call is used to clean up the structure allocated and initialised by the OsSemCreate call.
  - *tOsSemData *tpSemaphore*
    The operating system dependent semaphore data to clean up. After this call, the semaphore data will be invalid.

3.3.3.3 Mutex

All API calls concerned with this library use the “OsMutex” prefix

Following data structure is defined:

– *tOsMutexData*
  Data structure containing all operating system dependent data for accessing mutexes and related data. This is used to hide operating system internals.

Following API calls are defined:

– *tOsMutexData *OsMutexCreate(void)*
  This call is used to create the operating system dependent data to use a mutex.
  - *return: tOsMutexData**
    Operating system dependent data.

– *void OsMutexInit(tOsMutexData *tpMutex)*
  This call is used to initialise the operating system mutex object.

- *tOsMutexData *tpMutex*
  Operating system dependent data.

- *void OsMutexP(tOsMutexData *tpMutex)*
  This call is used to try and take the mutex.
  - *tOsMutexData *tpMutex*
    The mutex to take.
- *void OsMutexV(tOsMutexData *tpMutex)*
  This call is used to release the mutex.
  - *tOsMutexData *tpMutex*
    The mutex to release.
- *void OsMutexDestroy (tOsMutexData *tpMutex)*
  This call is used to destroy the operating system mutex initialised by the OsMutexInit call.
  - *tOsMutexData *tpMutex*
    The mutex to destroy.
- *void OsMutexDelete (tOsMutexData *tpMutex)*
  This call is used to clean up the structure allocated and initialised by the OsMutexCreate call.
  - *tOsMutexData *tpMutex*
    The operating system dependent mutex data to clean up. After this call, the mutex data will be invalid.

3.3.3.4 Interrupts

Remark that for some operating systems it is not possible to use the generic interrupt handling test code. In such case specific custom interrupt test software is written. These tests still follow the structure of the generic tests.

All API calls concerned with this library use the “OsIrq” prefix.

Following data structure is defined:

- *tOsIrqData*
  Data structure containing OS specific data.

Following API calls are defined:

- *tOsIrqData* OsIrqCreate(int vector, void(*isr)(void*), void *arg)*
  This API call will create and set-up the tOsIrqData structure.
  - *Int vector*
    Interrupt vector to connect interrupt handler on (use the IrqGenXGetVector() call to get this from the interrupt generating library).
  - *void(*isr)(void*)*
    The interrupt handler to be called from the interrupt.
  - *void *arg*
    Argument to be passed to the interrupt handler.
- *void OsIrqDelete(tOsIrqData* Irq)*

This API call will clean-up and free the tOsIrqData structure.

  - *tOsIrqData* Irq*

    Data structure containing OS specific data.

- *void OsIrqEnable(tOsIrqData* Irq)*

  API call used to enable the interrupt.

  - *tOsIrqData* Irq*

    Data structure containing OS specific data.

- *void OsIrqDisable(tOsIrqData* Irq)*

  API call to disable the interrupt.

  - *tOsIrqData* Irq*

    Data structure containing OS specific data.

4 The tests described

4.1 Calibration system test (CAL)

These tests are used to calibrate the tracing overhead in comparison with the processing power of the platform. This is important to understand the accuracy of the measurements done in scope of this report.

Here the minimum time between two traces is measured, which shows the time duration of taking a sample. The results in the evaluation report are the trace timestamps decremented with this value.

To detect how much the impact is of the tracing system in comparison with CPU performance a second test is done. In this test, the loop time is measured and compared with the tracing overhead.

In short:

<test name>	description
P_TRC	Measure the tracing overhead
P_CPU	Measure the CPU performance in comparison with the tracing overhead

4.1.1 Tracing overhead (CAL_P_TRC)

This test will calibrate the tracing system overhead. The result found will not only be used to have an idea of the overhead but also to depict the accuracy of the measurements. If the trace delay is stable and known, then the measurement accuracy will be better than with an unstable delay. Therefore, the test loop will disable interrupts during the two traces to avoid any influence from the platform on the time measurements. For this the critical section operating system calls are used.

In the rest of the report, the tracing overhead will be subtracted from the results obtained.

4.1.1.1 Test Parameters

None

4.1.1.2 Measurements done

Following times are measured during the test:

– Tracing overhead.

4.1.1.3 Test results

Results will be shown in a table as shown below:

Test	result
Average tracing overhead	In nsec
Minimum tracing overhead	In nsec
Maximum tracing overhead	In nsec

Test	**result**
Tracing accuracy	In nsec
Critical section primitive present?	YES or NO

4.1.2 CPU power (CAL_P_CPU)

This test will calibrate the CPU performance and the memory bandwidth of the platform being used. This test does measurements with the same code in 2 cases: cached (loop) or not cached (un-looped) code and data. As such the effects of the cache can be calculated and performance of platforms can be compared with our standard platform (Pentium MMX 200 MHz platform).

4.1.2.1 Test Parameters

None

4.1.2.2 Measurements done

– Duration of CPU test loop (cached and not cached)

– Duration of memory test loop (cached and not cached)

4.1.2.3 Test results

Caching effect:

Test	**no cache**	**cached**	**cache effect**
CPU test duration			
MEM test duration			
Average caching effect (CPU and MEM)			

The same test on our standard platform (Pentium MMX 200 MHz):

Test	**no cache**	**cached**	**cache effect**
CPU test duration	401.9 us	270.8 us	1.48
MEM test duration	5.442 ms	1.512 ms	3.60
Average caching effect (CPU and MEM)			2.54

Performance compared with our standard platform, larger values mean faster:

Test	**no cache**	**cached**
CPU performance factor		
MEM performance factor		

4.2 Clock tests (CLK)

The clock test measures the time an operating system needs to handle its clock interrupt. Some platforms (like the x86 motherboard) have the clock interrupt on the highest system interrupt. This is certainly not the best choice for real-time systems. In such a case, a clock interrupt showing up during a testcycle will cause a delay in the measurement for that cycle. If any test takes multiple operating system clock cycles to finish, than these spikes will be seen in the test results.

This is the reason why this test is the first run in our test bench. The table below shows the different tests done:

<test name>	description
B_CFG	Test the internal clock period setting
P_DUR	Test the clock interrupt processing duration

4.2.1 Operating system clock setting (CLK_B_CFG)

This will test the setting of the clock tick in the operating system. In our tests we use the default setting from the OS vendor. This test verifies the setting, or detects the clock tick timing if it is not settable.

4.2.1.1 Test Parameters

None

4.2.1.2 Measurements done

None

4.2.1.3 Test results

These test results only show the clock tick time. They are not used to validate a system as being real-time. If the test would not behave as expected (e.g. sleep behavior problem) than the clock time is measured by other means.

Results will be shown in a table as shown below:

Test	result
Test succeeded	YES or NO
Tested clock period	Time measured in msec
Clock period adaptable	YES or NO

4.2.2 Clock tick processing duration (CLK_P_DUR)

This will test the clock tick processing duration in the kernel. The test can be parameterized to detect any impact of the system load on the clock processing time.

This test will have only one application thread running. This application performs busy loops. The time needed for each busy loop is measured and expected to always be the same. However, if during the loop a clock interrupt occurs, the busy loop will be interrupted for some time. The differences between the longer busy loop duration and the normal busy loop duration can only be caused by the clock interrupt, as al other interrupts are turned off during this test.

The test results are extremely important, as the clock interrupt will disturb all other measurements done in this framework.

It may be possible to disable the clock interrupt for some operating systems and platforms. Of course, then some other OS features may cease functioning such like “perform any delay” or “waiting with timeout”. Tests of these are then impossibleest.

4.2.2.1 Test Parameters

None

4.2.2.2 Measurements done

– Clock duration time

4.2.2.3 Test results

These test results show the clock tick processing duration. The test results are shown in a diagram together with a table containing the normal busy loop time and the loop time when a clock interrupt occurred.

The loop counter used in the test loop has to be chosen so that:

– Enough loop samples including a clock interrupt are gathered (some hundreds)

– Time can be measured accurately.

With long loops, more samples are generated, but the timing accuracy of the samples becomes less (timing is done with a 16-bit exponential timing system: limited bit size of mantissa!).

Test	result
CLOCK_LOOP_COUNTER	The value set for the test loop.
Normal busy loop time	Time in µs of the busy loop when parameter above is used for this test.
Busy loop time with clock interrupt	Time in µs of the busy loop when an clock irq occurred.
Clock interrupt duration	Difference between the two values above.

Diagrams:

– Clock duration time

4.3 Thread tests (THR)

Thread tests shows the behavior of the scheduler. What is the thread switch latency? Does it depend on system load? What's the time to create/delete a thread? Are the priorities handled well? These are some of the questions we solve with the Thread tests.

The table below shows the different tests done:

<test name>	description
B_NEW	Thread creation behavior: when creating a thread with lower, same or higher priorities how are these scheduled?
B_RR	Test checks the Round Robin scheduling of threads at the same priority level.
P_SLS	Thread switch latency performance between same priority threads.
P_NEW	Thread creation and deletion time.

4.3.1 Thread creation behaviour (THR_B_NEW)

This will test the thread creation behavior. Does the operating system behave as a real-time operating system should behave?

This test checks following issues when generating a thread:

– When creating a thread with a lower priority than the creating thread, following rule applies: "The new thread shall not run while the creating thread is active".

– When creating a thread with the same priority than the creating thread, following rule applies: "The new thread should not be activated immediately; it should be put at the tail of the ready thread queue".

– When yielding the processor, another thread in the same priority FIFO queue (if any) shall run.

– When creating a thread with a higher priority than the creating thread, following rule applies: "The new thread shall preempt the creating thread and become active immediately.

– When a thread lowers its priority below the priority of another thread that is in the ready-to-run state, then the thread shall be preempted and the highest priority ready-to-run thread shall be activated.

If these tests fail, most other thread tests will not be able to run.

4.3.1.1 Test Parameters

None

4.3.1.2 Measurements done

None

4.3.1.3 Test results

If the test fails, then the operating system will NOT receive the RT-VALIDATED logo!

The data trace of this test should be:

```
/* normal behavior */
STATE_START
STATE_YIELDING
STATE_MIDDLE_ACTIVE
STATE_HIGH_ACTIVE
STATE_LOW_ACTIVE
STATE_END
```

Also no error trace may be generated, otherwise the test invalidates the OS as being RT!

Results will be shown in a table as shown below:

Test	result
Test succeeded	YES or NO
Lower priority not activated?	OK or FAILED
Same priority at tail?	OK or FAILED
Yielding works?	YES or NO
Higher priority activated?	OK or FAILED

Diagrams: none

4.3.2 Round robin behaviour (THR_B_RR)

This test checks if the scheduler uses a fair round robin mechanism when threads are having the same priority and all are in the ready-to-run state!

All threads are in the ready-to-run state so on each operating system clock tick (end of time slice) the round robin mechanism should schedule the next thread in the ready queue and store the current thread at the end of the FIFO queue.

Remark that not all operating systems use round robin scheduling between threads running at the same priority. As this feature is not needed for guarantying real-time behavior, an OS without this feature can still receive the “RT-VALIDATED” logo.

4.3.2.1 Test Parameters

– PAR_THREADS_QTY: number of threads that will be used in this test, the number of threads that have to be scheduled in the FIFO
  In normal circumstances, this test is run only with PAR_THREADS_QTY set to 10. If anomalies are detected, other values than 10 may be used trying to understand the bad behavior.

4.3.2.2 Measurements done

None

4.3.2.3 Test results

If the operating system uses a fair round robin system for scheduling ready-to-run threads of the same priority, then each clock tick a trace will show up. Also the order of the trace is important: the thread with ID PAR_THREADS_QTY will be the first followed by decreasing thread IDs until zero is reached. Then it should restart the same scenario all over again.

When there is no fair scheduling it can occur that a thread will loop endlessly. After some time the main thread will higher its priority to stop the test in all cases.

Results will be shown in a table as shown below:

Test	result
Test succeeded	YES or NO
Time slice following this test	Time slice in ms

Diagrams: none because it deals with behavior

4.3.3 Thread switch latency between same priority threads (THR_P_SLS)

This test will measure the time to switch between threads of the same priority. Therefore the "voluntary_yield_processor_to_other_thread" system call is used. If the THR_B_NEW test fails, then this test cannot be run.

A number of threads at the same priority will be generated. Each thread will yield another thread. These threads are handled in round-robin mode (if supported by the OS).

As for a voluntary yield, only the ready-to-run queue of the same priority level has to be checked, this test may have better results than the switch latency test for different priorities.

The aim here is to check the FIFO queuing mechanism: this should not depend on the number of threads in the FIFO: otherwise the scheduler is has not a predictable behavior.

4.3.3.1 Test Parameters

– PAR_THREADS_QTY: number of threads that will be used in this test, the number of threads that have to be scheduled in the FIFO
  This test is run with PAR_THREADS_QTY set to:
  – 2
  – 10
  – 128

4.3.3.2 Measurements done

Test	Sample qty	Avg	Max	Min
Thread switch latency, 2 threads				
Thread switch latency, 10 threads				
Thread switch latency, 128 threads				

4.3.3.3 Test results

If the test fails, then the operating system will NOT receive the RT-VALIDATED logo!

Results will be shown in a table as shown below:

Test	result
Test succeeded	YES or NO

Diagrams:

- Thread switch latency between two threads.
- Thread switch latency between ten threads.
- Thread switch latency between 128 threads.

Remark that for the OS to be predictable, the number of threads in the ready queue may not have an impact on the switch latency measured.

4.3.4 Thread creation and deletion time (THR_P_NEW)

This will test the time to create a thread and the time to delete a thread. Different scenarios are possible, from which the following are tested here:

- Scenario “NER” (NEver Run): The created thread has a lower priority than the creating thread and is deleted before it had any change to run: in this test no thread switch occurs.
- Scenario “RTE” (Run and TErminate): The created thread has a higher priority than the creating thread and activates. The created thread immediately terminates itself (thread does nothing).
- Scenario “RNT” (Run, but does Not Terminate): The same scenario as above, but the created thread does not terminate (it lowers it’s priority when it is activated).

In the scenarios “RTE” and “RTN”, the creation time is the duration from the system call creating the thread to the time when the created thread activates. For the “NER” scenario the creation time is the duration of the system call.

Remark that this test cannot run if the THR_B_NEW test failed!

4.3.4.1 Test Parameters

- SCENARIO: test scenario selected
  - SC1: NER, Never run
  - SC2: RTE, Run and terminate
  - SC3: RNT, Run, but do not terminate

4.3.4.2 Measurements done

Test	Sample qty	Avg	Max	Min
Thread creation, never run				
Thread deletion, never run				
Thread creation, run and terminate				
Thread deletion, run and terminate				
Thread creation, run and block				
Thread deletion, run and block				

4.3.4.3 Test results

If the test fails, then the operating system will NOT receive the RT-VALIDATED logo!

Results will be shown in a table as shown below:

Test	result

Test	result
Test succeeded	YES or NO

Diagrams:

- Thread creation time in the “NER” scenario (duration of system call).
- Thread deletion time in the “NER” scenario.
- Thread creation time in the “RTE” scenario (duration of system call start to activated thread).
- Thread deletion time in the “RTE” scenario.
- Thread creation time in the “RNT” scenario (duration of system call start to activated thread).
- Thread deletion time in the “RNT” scenario.

4.4 Semaphore tests (SEM)

This is testing the performance and the behavior of a counting semaphore. The counting semaphore is a system object that protects for simultaneous accesses to some device or resource. This is well known as the Dijkstra paradigm. A semaphore object has:

– A semaphore counter

– A P() function which tries to acquire the semaphore (from the Dutch language “Probeer” = Try). If the counter is zero, this system call will block the calling thread until the semaphore is released by another thread. If the semaphore counter is not zero, the counter will decrement and the thread continues.

– A V() function which releases the semaphore (from the Dutch language “Vrij” = Release). This will increment the semaphore counter.

Remark that in scope of this test, only protection semaphores between threads belonging to the same process are tested.

In most operating systems, there exists system calls for a simpler version of the counting semaphore object: where the counter can only be zero or one (acquired or free). In the next section we discuss such an object that we will call the “MUTEX” (MUTual EXclusive semaphore).

The table below shows the different tests done for this object:

<test name>	description
B-LCK	Semaphore protection behavior.
B-REL	Verifies that blocked thread with highest priority activates on a release operation.
P-NEW	Semaphore creation and deletion time.
P-ARC	Acquire and release time in contention case
P-ARN	Acquire and release time in no contention case

4.4.1 Semaphore locking test mechanism (SEM_B_LCK)

This will test if the counting semaphore locking mechanism works as expected. The P() call should block only when the count is zero. The V() call should increment the semaphore counter. In the case the semaphore counter is zero, the V() call should cause a rescheduling in the kernel: indeed blocked threads may be activated.

The aim of this test is to detect the good behavior of the the counter functions.

4.4.1.1 Test Parameters

None

4.4.1.2 Measurements done

Only state behavior is measured.

4.4.1.3 Test results

If the test fails, then the operating system will NOT receive the RT-VALIDATED logo! In case where the test fails in a way that the semaphore cannot guaranty some protection over shared objects, then the operating system will receive the DID_NOT_QUALIFY logo.

The data trace of this test should be:

```
/* normal behavior */
STATE_HIGH_ACTIVE
STATE_LOW_ACTIVE
STATE_HIGH_ACTIVE
STATE_LOW_ACTIVE
STATE_HIGH_ACTIVE
```

Also no error trace may be generated, otherwise the test invalidates the OS as being RT!

Results will be shown in a table as shown below:

Test	result
Test succeeded	YES or NO
Maximum semaphore value?	Not tested: from documentation
Rescheduling on free?	OK or FAILED

4.4.2 Semaphore releasing mechanism (SEM-B-REL)

This test verifies that the highest priority thread being blocked on a semaphore will be released by the release operation. This should be independent of the order of the acquisitions taking place.

Therefore the test is run in two scenarios:

– Where the highest priority thread acquires first the semaphore (called the scenario "HI")

– Where the lowest priority thread acquires first the semaphore (called the scenario "LOW")

The release result of both scenarios should be the same: the highest priority thread should be activated upon the release.

4.4.2.1 Test Parameters

- SCENARIO: test scenario selected
  - 1: "HI" Highest priority thread acquires first.
  - 2: "LOW" Lowest priority thread acquires first.

4.4.2.2 Measurements done

Only state behavior is measured

4.4.2.3 Test results

If the test fails, then the operating system will NOT receive the RT-VALIDATED logo! However, the semaphore can still be used to protect shared data and devices.

The data trace of this test should be:

```
/* behavior in "HI" */                /* behavior in "LOW" */
STATE_LOW_ACTIVE                      STATE_LOW_ACTIVE
STATE_HIGH_ACTIVE                     STATE_MIDDLE_ACTIVE
STATE_MIDDLE_ACTIVE                   STATE_HIGH_ACTIVE
STATE_LOW_ACTIVE                      STATE_LOW_ACTIVE
STATE_HIGH_ACTIVE                     STATE_HIGH_ACTIVE
STATE_LOW_ACTIVE                      STATE_LOW_ACTIVE
STATE_MIDDLE_ACTIVE                   STATE_MIDDLE_ACTIVE
STATE_LOW_ACTIVE                      STATE_LOW_ACTIVE
```

Results will be shown in a table as shown below:

Test	result
Test succeeded	YES or NO

4.4.3 Time needed to create and delete a semaphore (SEM_P_NEW)

This will test the time needed to create a semaphore and the time to delete it. The deletion time is checked in two cases:

– Where the semaphore is used between the creation and deletion (the “USE” scenario).

– Where the semaphore is not used between the creation and deletion (the “DUM” or dummy scenario).

For a good real-time operating system it is expected that there is no difference between the two scenarios. If a difference is detected, then this probably means that the operating system handles some initializations on the semaphore on its first use (making the first use slower, which is not desirable in a RT system).

4.4.3.1 Test Parameters

– SCENARIO: test scenario selected
  – 1: “USE” semaphore used.
  – 2: “DUM” semaphore not used: thus a dummy semaphore.

4.4.3.2 Measurements done

Following times are measured during the test:

Test	Sample qty	Avg	Max	Min
Semaphore creation time, used				
Semaphore deletion time, used				
Semaphore creation time, never used				
Semaphore deletion time, never used				

4.4.3.3 Test results

If the test fails, then the operating system will NOT receive the RT-VALIDATED logo!

Results will be shown in a table as shown below:

Test	result
Test succeeded	YES or NO

Diagrams:

– Semaphore creation time, used.

– Semaphore deletion time, used.

– Semaphore creation time, never used.

– Semaphore deletion time, never used

4.4.4 Test acquire-release timings: no contention case (SEM_P_ARN)

This tests the acquisition and release time in the no contention case. As in this test case the semaphore does not block nor causes any rescheduling (thread switch), the duration of the system call should be very short.

In fact, the OS will only need to increase or decrease the semaphore counter in an atomic way.

4.4.4.1 Test Parameters

None

4.4.4.2 Measurements done

Following times are measured:

Test	Sample qty	Avg	Max	Min
Semaphore acquisition time, no contention				
Semaphore release time, no contention				

4.4.4.3 Test results

If the test fails, then the operating system will NOT receive the RT-VALIDATED logo!

Results will be shown in a table as shown below:

Test	result
Test succeeded	YES or NO

Diagrams:

– Semaphore acquisition time, no contention.

– Semaphore release time, no contention.

4.4.5 Test acquire-release timings: contention case (SEM_P_ARC)

This is used to test the time needed to acquire and release a semaphore depending on the number of threads blocked waiting for the semaphore. It measures only the time in the contention case: this means when the acquisition and release system call causes a rescheduling to occur.

The aim of this test is to verify if the number of blocked threads waiting for the semaphore has an impact on these timings. So this will answer the question: “how much time the operating system needs to find out the next thread to schedule?”.

This test is very important to detect how predictable the operating system is depending on the number of blocked threads in the wait for semaphore list. When a good search algorithm is used the impact should be small. This test also figures out where most time is spend in ordering the threads:

– During acquisition: when the thread gets in the blocked state.
– During release: when the thread gets in the read-to-run state.

This test will be done with a number of threads equal to the number of priorities available in the operating system with a maximum of.

Remark that we are testing the contention case: so each measurement will always include the thread switch latency!

4.4.5.1 Test Parameters

– None

4.4.5.2 Measurements done

Following times are measured:

Test	Sample qty	Avg	Max	Min
Semaphore acquisition time, contented				
Semaphore release time, contented				

4.4.5.3 Test results

If the test fails, then the operating system will NOT receive the RT-VALIDATED logo!

Results will be shown in a table as shown below:

Test	result
Test succeeded	YES or NO
Max number of threads pending	128 or number of priority levels if less than 128

Diagrams:

– Semaphore acquisition time, contended.

– Semaphore release time, contended.

Most of the time, a zoom-in diagram will be shown to see the influence on the number of blocked threads.

4.5 Mutex tests (MUT)

Here the performance and the behavior of the mutual exclusive semaphore are tested.

Although the mutual exclusive semaphore (further called mutex) could be the same as the counting semaphore where the count is one, this is not the aim of this test to copy the precious described tests. In the scope of the framework, this test will look into the details of a mutex system object that avoids priority inversion.

Details about the priority inversion situation can be found in [Doc. 2].

Different mechanisms exists to avoid a priority inversion scenario, most RTOS use one of these:

– Priority inheritance: the blocking thread will inherit the priority of the blocked thread.
– Priority ceiling: the priority of the blocking thread will be set to a high fixed (ceiling) priority.

In scope of this framework, it does not matter how the operating system avoids priority inversion. It only detects if such a system actually does prevent the priority inversion.

If the operating system does not has such a mechanism, then this section will be skipped (tests will not be done).

The tests in this section will also detect how much time it takes to deal with the priority inversion avoidance mechanism.

The table below shows the different tests done for this object:

<test name>	**description**
B_ARC	Acquisition en release behavior in the contention case with priority inversion. Does the system call really avoid the priority inversion case?
P_ARC	Acquire and release time in contention case with priority inversion.

4.5.1 Priority inversion avoidance mechanism (MUT-B-ARC)

This test will determine if the system call under test prevents the priority inversion case. Therefore the test will artificially create a priority inversion.

The flow chart, with the expected execution path is shown in the figure below (execution path in light green). The important steps in the execution flow are:

- 1: The main test execution thread, which runs at low priority will create two other threads.
  - A high priority thread, that will start execute immediately. This thread will block on the semaphore “high”.
  - A middle priority thread, with the priority between the creating and the high level thread. Also this thread will block, now on the semaphore “middle”.
- 2: The main thread will acquire the mutex (so the critical section lock starts there).
- 3: The main thread will release the semaphore “high” so the high priority thread activates. This simulates an external event in a real system.
- 4: The high level thread also acquires the mutex: as the mutex is taken, the high priority thread blocks, and the low level priority activates again.
- 5: This is the crucial point: the low level thread activates the middle level thread (by releasing the middle semaphore). If priority inversion protection is enabled, then the middle level thread will NOT activate! Instead, the low-level priority thread has inherited the high level priority of the blocked high level thread (or received the mutex ceiling priority). As this priority is now higher than the middle priority, the low level thread continues!
- 6: The low level thread comes at the end of the critical section and releases the mutex. At that moment the priority of the thread is restored and the operating system will schedule the high level thread when the lock is released.

The rest of the flow is straightforward.

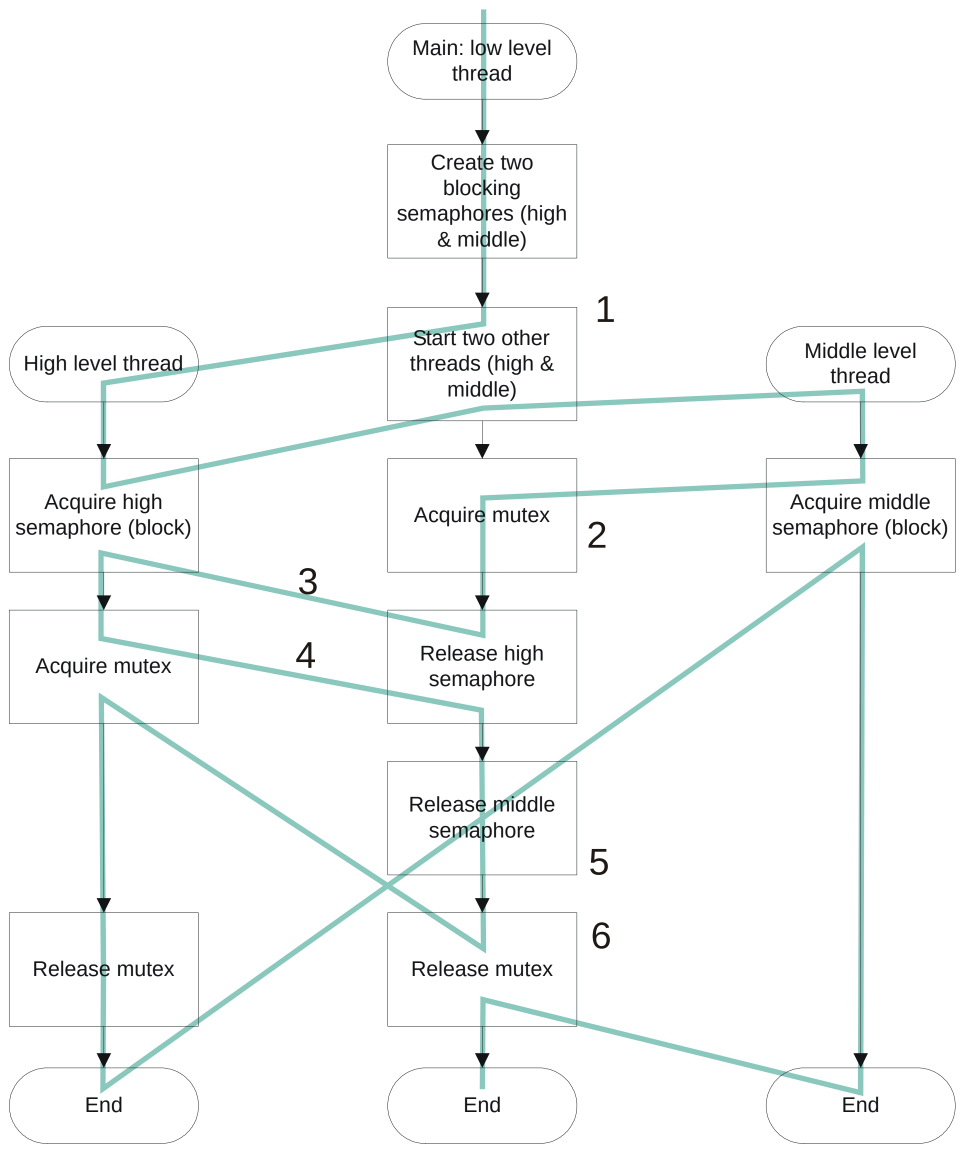


Figure: Priority inversion avoidance

When there is no priority inversion avoidance mechanism, the flow of the test will be as shown in the following diagram. In that case the middle level thread will be activated first!

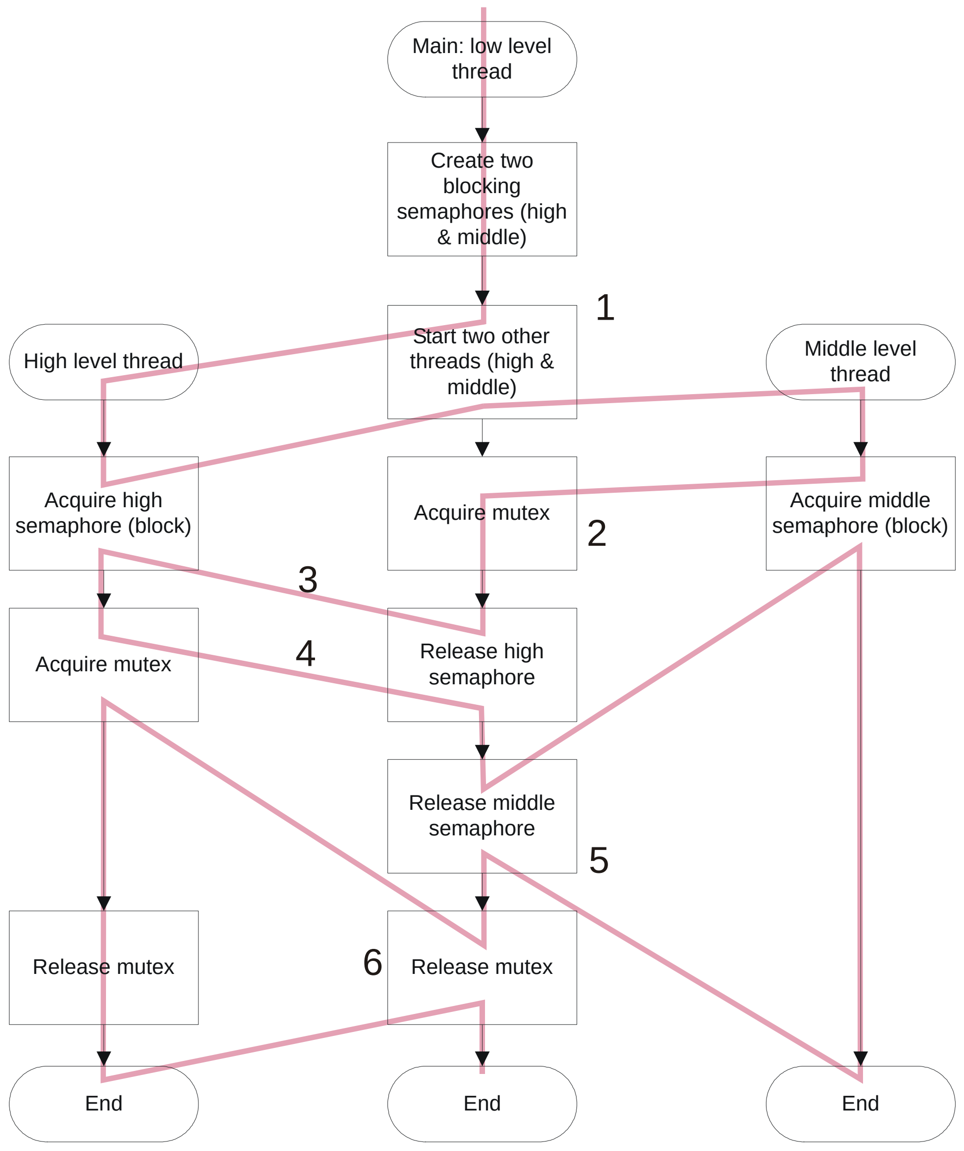
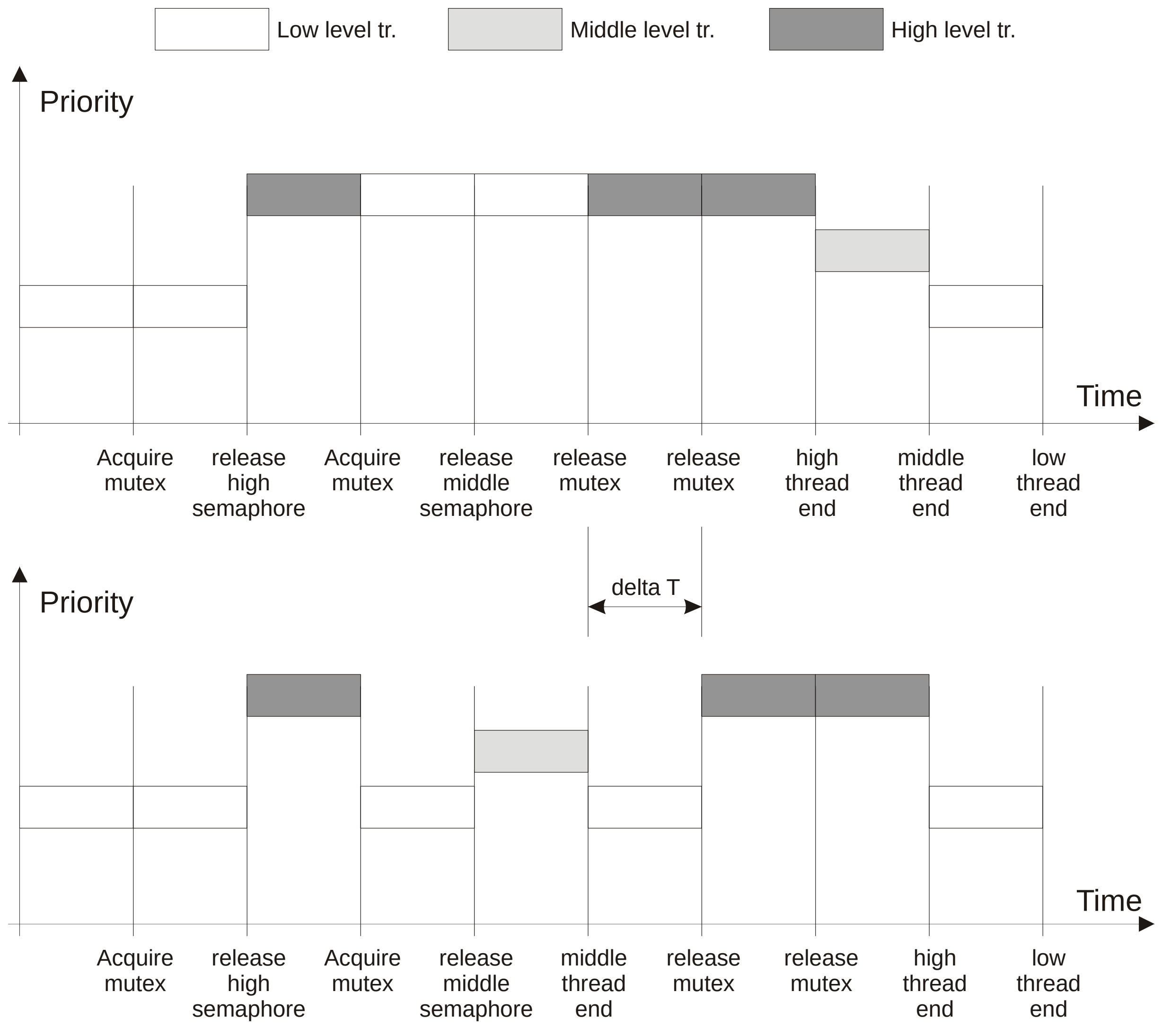


Figure: No priority inversion avoidance

In the figure below, the timing difference between the two scenarios is shown. It is clear that priority inversion causes an extra delay (delta T) for the high level thread. As a result, the system becomes less predictable. Do not forget that in a real live situations the middle priority thread can be active for a long time!



Timing diagrams

Therefore, an operating system that does not have any system call to avoid a priority inversion case will not receive the RT-VALIDATED logo!

4.5.1.1 Test Parameters

None

4.5.1.2 Measurements done

None

4.5.1.3 Test results

If the test fails, then the operating system will NOT receive the RT-VALIDATED logo! However, it still can receive the VALIDATED logo if the locking of the mutex is usable for disabling mutual access to some part of the code.

The data trace of this test should be:

```
/* RT-VALIDATED */                    /* VALIDATED */
STATE_LOW_ACTIVE                      STATE_LOW_ACTIVE
STATE_HIGH_ACTIVE                     STATE_MIDDLE_ACTIVE
STATE_LOW_ACTIVE                      STATE_LOW_ACTIVE
STATE_LOW_ACTIVE                      STATE_MIDDLE_ACTIVE
STATE_HIGH_ACTIVE                     STATE_LOW_ACTIVE
STATE_MIDDLE_ACTIVE                   STATE_HIGH_ACTIVE
STATE_LOW_ACTIVE                      STATE_LOW_ACTIVE
```

Results will be shown in a table as shown below:

Test	result
Priority inversion avoidance system call present	YES or NO
System call used	
Test succeeded	YES or NO
Priority inversion avoided	YES or NO
Mechanism used if any?	INHERITENCE or CEILING

4.5.2 Mutex acquire-release timings: contention case (MUT_P_ARC)

This is the same test as above, but performed in a loop. In this case, the time is measured to acquire and release the mutex in the priority inversion case.

4.5.2.1 Test Parameters

None

4.5.2.2 Measurements done

Following times are measured:

Test	Sample qty	Avg	Max	Min
Mutex acquisition time, contended				
Mutex release time, contended				

4.5.2.3 Test results

If the test fails, then the operating system will NOT receive the RT-VALIDATED logo!

Results will be shown in a table as shown below:

Test	result
Test succeeded	YES or NO

Diagrams:

- Mutex acquisition time, contended.
- Mutex release time, contended.

4.6 Interrupt tests (IRQ)

Here the performance of the interrupt handling in the operating system and hardware is tested.

In a real-time system, interrupt handling is a major part of the system: indeed such systems are typically event driven. The stress tests, which check how stable the interrupt latency is shows how much the kernel uses a critical section with disabling interrupts. Real-Time operating systems do this as less and as short as possible.

The table below shows the different tests done for this object:

<test name>	description
B_SIM	Behavior of nested interrupts: do they prioritize, or are they handled in a FIFO way.
P_LAT	Interrupt latency (from interrupt to interrupt handler), hardware and operating system delay combined.
P_DLT	Interrupt dispatch latency (from interrupt handler to interrupted thread) when no rescheduling occurs.
S_SUS	Maximum sustained interrupt frequency the system can handle without loosing interrupts.

4.6.1 Simultaneous interrupt priority handling (IRQ_B_SIM)

This test verifies if simultaneous interrupts are handled prioritized. It answers the question if a lower priority interrupt can be pre-empted by a higher-level interrupt.

Just like thread priorities, prioritization of interrupts makes higher level interrupts more predictable. Remark that it is not always possible to change the priority of a certain interrupt: this depends largely on the platform used.

Prioritization behaviour can easy tested by starting the interrupt generation of one device in the interrupt handler of the other device. This is done in two scenarios, in one of the two scenarios the interrupt handler will be interrupted by the other. In the other scenario the interrupt handler won't be interrupted. If this is the case, then prioritization occurs.

4.6.1.1 Test Parameters

- SCENARIO: test scenario selected
  - 1: Interrupt handler A will initiate interrupt B.
  - 2: Interrupt handler B will initiate interrupt A.

4.6.1.2 Measurements done

Only state change is measured

4.6.1.3 Test results

If the test fails, then the operating system will NOT receive the RT-VALIDATED logo!

Results will be shown in a table as shown below:

Test	result
Test succeeded	YES or NO
Interrupt pre-emption existing following the documentation?	-
Lower level interrupt pre-empted by higher level interrupt?	YES or NO
Higher-level interrupt not pre-empted by lower level interrupt?	YES or NO
If not priority based: which mechanism is used?	LIFO or FIFO

4.6.2 Interrupt latency (IRQ_P_LAT)

This measures the time it takes to switch from a running thread to an interrupt handler. So it only measures the software latency.

The latency caused by the hardware: from interrupt line going high, until the processor starts the exception vector, is not measured here.

4.6.2.1 Test Parameters

None

4.6.2.2 Measurements done

Following times are measured:

Test	Sample qty	Avg	Max	Min
Dispatch latency from interrupt handler				

4.6.2.3 Test results

Diagrams:

– Dispatch latency from interrupt handler.

4.6.3 Interrupt dispatch latency (IRQ_P_DLT)

This measures the time it takes to switch from the interrupt handler back to the interrupted thread.

The total overhead on an interrupted thread is both the interrupt latency and the dispatch latency. Of course also the duration of the interrupt handling itself has to be added.

4.6.3.1 Test Parameters

None

4.6.3.2 Measurements done

Following times are measured:

Test	Sample qty	Avg	Max	Min
Dispatch latency				

4.6.3.3 Test results

Diagrams:

– Dispatch latency.

4.6.4 Interrupt to thread latency (IRQ_P_TLT)

4.6.4.1 Test results

This measures the time it takes to switch from the interrupt handler to the thread that is activated (by using a semaphore if this can be provided by the OS) from the interrupt handler.

This can largely depend on the operating system under test. In the generic code the semaphore is used to do this. However, in Linux for instance, there is no simple way to do this. Most drivers in Linux will block the calling process and activate it again when data is available (interrupt occurred).

Most RTOS do provide such a mechanism.

4.6.4.2 Test results

Test	Sample qty	Avg	Max	Min
Latency from interrupt to activated thread				

4.6.4.3 Diagrams

Diagrams:

– Latency from interrupt to activated thread.

4.6.5 Maximum sustained interrupt frequency (IRQ_S_SUS)

This test measures the probability an interrupt is missed: is the interrupt handling duration stable and predictable?

The test is done on different levels, depending on the RTOS and the results of each test:

– 100 000 interrupts, initial phase: each test takes only some seconds.

– 1 000 000 interrupts, second phase based on the results from the first phase. This test still takes less than a minute and gives already accurate results.

– 1 000 000 000 interrupts, takes some hours: to verify stability.

On some operating system the worst-case interrupt latency is so large, that it is impossible to do the test with a billion interrupts. In such case, a smaller number of interrupts will be used.

4.6.5.1 Test results

Shown in a table as below. Remark that this table is just an example.

Interrupt period	#interrupts generated	#interrupts serviced	#interrupts lost
20 µs	100 000	100 000	0
20 µs	1 000 000	999 982	18
25 µs	1 000 000	1 000 000	0
25 µs	1 000 000 000	1 000 000 000	0

4.7 Memory tests (MEM)

This test will check if there are memory leaks in the operating system. It will create and delete in a loop different type of operating system objects (threads, semaphores, mutex, …).

The table below shows the different tests on this subject:

<test name>	description
B_LEK	Test if there are memory leaks in the OS.

4.7.1 Memory leak test (MEM_B_LEK)

This test continuously create/remove objects in the operating system (threads, semaphores, mutexes, …).

4.7.1.1 Test Parameters

– None

4.7.1.2 Measurements done

Nothing: memory consumption is checked after a large number of test loops.

4.7.1.3 Test results

Results will be shown in a table as shown below:

Test	result
Test succeeded	YES or NO
Test duration (how long we let the endless loop run)	
Number of main test loops done	

5 Appendix A: Document revision history

5.1 Issue 1.0 (April 29, 2004)

Initial version