

OSE 4.5.1 ON A PPC PLATFORM

© Copyright Dedicated Systems Experts NV. 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 NV, Bergensesteenweg 421 B12, B-1600 St-Pieters-Leeuw, Belgium.

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

Date: **April 19, 2004**

Doc: **EVA-2.9-TST-OSE-PPC-01**

Issue: **1**

1	Introduction	6
1.1	Purpose and scope	6
1.2	Document issue: the 2.9 framework	6
1.3	Related documents	7
2	Results summary	8
2.1	Product under test.....	8
2.2	Test result.....	8
2.2.1	Positive points	8
2.2.2	Negative points.....	8
2.2.3	Ratings	8
3	Introduction	9
3.1	Product under test.....	9
3.1.1	Software	9
3.1.2	Hardware	9
3.2	Introduction	9
4	Installation and BSP	10
4.1	Installation.....	10
4.1.1	Installation on Host.....	10
4.1.2	Installation on target.....	10
4.2	BSP	12
5	Test Results	13
5.1	Calibration system test (CAL)	13
5.1.1	Tracing overhead (CAL-P-TRC).....	13
5.1.2	CPU power (CAL-P-CPU).....	13
5.2	Clock tests (CLK)	15
5.2.1	Operating system clock setting (CLK-B-CFG)	15
5.2.2	Clock tick processing duration (CLK-P-DUR)	15
5.3	Thread tests (THR)	18
5.3.1	Thread creation behaviour (THR-B-NEW)	19
5.3.2	Round robin behavior (THR-B-RR).....	19
5.3.3	Thread switch latency between same priority threads (THR-P-SLS).....	20
5.3.4	Thread creation and deletion time (THR-P-NEW).....	22
5.4	Semaphore tests (SEM).....	27
5.4.1	Semaphore locking test mechanism (SEM-B-LCK)	27
5.4.2	Semaphore releasing mechanism (SEM-B-REL).....	27
5.4.3	Time needed to create and delete a semaphore (SEM-P-NEW).....	27
5.4.4	Test acquire-release timings: contention case (SEM-P-ARN).....	30
5.4.5	Test acquire-release timings: contention case (SEM-P-ARC)	32
5.5	Mutex tests (MUT).....	35
5.5.1	Priority inversion avoidance mechanism (MUT-B-ARC)	35
5.5.2	Mutex acquire-release timings: contention case (MUT-P-ARC)	35

Date: **April 19, 2004**

Doc: **EVA-2.9-TST-OSE-PPC-01**

Issue: **1**

5.6	Interrupt tests (IRQ)	36
5.6.1	Simultaneous interrupt priority handling (IRQ_B_SIM)	36
5.6.2	Interrupt latency (IRQ_P_LAT).....	37
5.6.3	Interrupt dispatch latency (IRQ_P_DLT)	38
5.6.4	Interrupt to thread latency (IRQ_P_TLT).....	39
5.6.5	Maximum sustained interrupt frequency (IRQ_S_SUS).....	40
5.7	Memory tests.....	41
5.7.1	Memory leak test (MEM_B_LEK).....	41
6	Support.....	42
7	Appendix A: Vendor comments	43
8	Appendix B: Acronyms	44
9	Appendix C: Document revision history.....	45
9.1	Issue 1.0 (April 19, 2004).....	45

2 Results summary

2.1 Product under test

OSE 4.5.1 from ENEA Data (Sweden) on a PowerPC platform.

2.2 Test result

"RT-VALIDATED": if you take care with the internals of the OS in your design.

2.2.1 Positive points









- Predictable and fast performance!
- Very good micro-kernel architecture and nice concepts on message passing!
- Ideally suited for distributed fail-safe systems!

2.2.2 Negative points

- Documentation has to be seriously improved, especially because the RTOS concept is somewhat different compared with traditional RTOS.
- No user friendly method to configure the operating system, no tool present to do this.
- Minor bugs found in the BSP and in POLO.

2.2.3 Ratings

For a description of the ratings, see [Doc. 3]. The first four ratings are the ones given in the theoretical evaluation (independent of the platform used for the tests) which can be found in [Doc. 5].

RTOS Architecture	0		10
OS Documentation	0		10
OS Configuration	0		10
Internet Components	0		10
Development Tools	0		10
Installation and BSP	0		10
Test results	0		10
Support	0		10

5.3 Thread tests (THR)

These tests are used to measure the performance of the scheduler.

Due to the design of OSE, which is based on a true micro-kernel, everything except the scheduler and signaler runs outside the kernel. The heap for instance, is handled by a separate thread. Another main concept of OSE is the signaling system: this means that a lot of system calls have a blocking behavior:

- The system call sends a signal to the related process handling this call
- The signal is handled at the priority of the receiver, in the mean time the sender waits for the answer and blocks.
- A signal is send to the sender with the results of the call.

Of course, using such "blocking" calls can jeopardize the real-time performance of the calling thread. Therefore two header files define all API calls in OSE:

- `ose.h`: the one containing all system calls
- `ose_i.h`: the one containing the non-blocking calls only (which may be used in the interrupt handlers).

The threads with stringent real-time requirements should use only the calls defined in the non-blocking header file. Of course none of the threads at the same or higher priority should use them, and the threads handling the system calls should be at a lower priority level.

All this to say that we needed to adapt the settings of the operating system to run these tests: when creating threads, this is done with a blocking call causing other threads to activate. We solved this by setting the OSE system daemon at the highest priority level, so the blocking call does not activate lower priority threads: indeed the system daemon handles the call and send the acknowledge back. Remark that this is done only to run our generic test-suite! This is something you shouldn't do in a real design!

☹ The main problem here is that this blocking nature of these calls is not clearly shown in the manuals! In the section where the documentation is discussed you will notice that such important hidden features has a major impact on the points given on documentation. For the discussion about the documentation and architectural inside of OSE we reference the reader to the qualitative evaluation of this product done in [Doc. 5], which can also be downloaded on our website.

A similar problem appears when ending a thread by a C-language return statement. In OSE, threads are not expected to end themselves in this way. When the main entry function of a thread returns, an error handling mechanism is started in OSE. The programmer can attach an error handler to the kernel and detect the "OSE_EPROCESS_ENDED" error. If however, the error handler is not installed or does not acknowledge that the error was handled, the kernel will go into an endless loop causing a system crash!

Of course, one can argue that in a normal real-time system a thread should not end so if it does than this is an error situation. However this is not always the case: typically a network server service will create threads for each connection made to it.

Again, this "feature" is a bit hidden in the documentation and easily overlooked!

The thread test behaved well and stable.

5.3.1 Thread creation behaviour (THR-B-NEW)

This will test the behavior of creating threads. Does the operating system behave as it should for a real-time operating system?

In our opinion, a thread created with a higher priority than the creating thread should activate immediately. A thread created with a lower priority than the creating thread should surely not activate until any higher priority threads have finished their job.

In OSE, thread creation and deletion are handled by the system daemon (ose_sysd) at the priority of this system daemon: these API calls can block. This can cause unexpected behavior if you use the OSE operating system wrongly!

To be able to run our standard generic test code, we needed to increment the OSE system daemon to the highest possible priority. This you shouldn't do in a real system! In a future version of the framework we are thinking to adapt these tests to resolve the need of incrementing the system daemon.

Our workaround by putting the system daemon on the highest priority is not a good idea for a real system: the system daemon is used also for other tasks that can consume time and this is better not done on the highest priority level!

Therefore we suggest the system developers using OSE not to use dynamic thread creation at higher priority level threads.

Remark that in OSE does not provides an explicit system call to yield the processor. However this can be done by setting the priority of the thread, which puts the thread at the tail of the ready FIFO for its priority. So we used the "set_pri(get_pri(current_process()))" construction to yield the processor.

5.3.1.1 Test results

Test	result
Test succeeded	YES
Lower priority not activated?	OK
Same priority at tail?	OK
Yielding works?	YES (by setting it's priority again)
Higher priority activated?	OK

5.3.2 Round robin behavior (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!

In OSE, there is no round-robin scheduling between processes using priority scheduling: each thread runs until a blocking call. Therefore this test could not be done.

Round robin scheduling exists in OSE when a thread is running in the background priority class.

5.3.2.1 Test results

Test	result
Test succeeded	NO (no such feature in OSE)
Time slice following this test	

5.3.3 Thread switch latency between same priority threads (THR-P-SLS)

This test measures the time to switch between threads of the same priority. Therefore the voluntary yield processor to other thread system call is used (in OSE this is done by setting the priority, as shown in the code extract: `set_pri(get_pri(current_process()))`).

Besides the 35 μ s spike (first clock tick after thread creation) and the clock ticks (5 μ s), the thread switch latency is very stable.

Once the number of threads becomes large enough, the thread context is swapped out the cache on each switch! Therefore the thread switch time increases from 0.8 μ s to 3.5 μ s (factor 4.4). This thread switch latency does not increase further with more threads: we tested this with 254 threads, which had also an average switch latency of 3.5 μ s (diagram not shown in the document).

☺ The thread switch latency does not depend on the number of ready threads in the ready queue (besides memory caching effects): this is a good thing!

5.3.3.1 Test results

Test	result
Test succeeded	YES

Test	Sample qty	Avg	Max	Min
Thread switch latency, 2 threads	16383	0.8 μ s	13.9 μ s	0.8 μ s
Thread switch latency, 10 threads	16379	0.8 μ s	16.2 μ s	0.8 μ s
Thread switch latency, 128 threads	16320	3.4 μ s	27.6 μ s	2.6 μ s

5.3.3.2 Diagrams

