WHAT MAKES A GOOD RTOS

Disclaimer

Although all care has been taken to obtain correct information and accurate test results, Real-Time Consult and Real-Time 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 Real-Time Consult and Real-Time Magazine have been advised of the possibility of such damages.

http://www.realtime-info.be

E-mail: info@realtime-info.be

EVALUATION REPORT LICENSE

This is a legal agreement between Mr Huang Wang, HH tech., USTC Phys. East Campus,HeFei,AnHui,P.R.China, HeFei, ANHui China, 230026, China, representing yourself and/or your company HH tech. and the company REAL-TIME CONSULT.

1. **GRANT**. Subject to the provisions contained herein, REAL-TIME CONSULT hereby grants you a non-exclusive license to use its accompanying proprietary evaluation report for projects where you or your company are involved as major contractor or subcontractor. You are not entitled to support or telephone assistance in connection with this license.
2. **PRODUCT**. REAL-TIME CONSULT shall furnish the evaluation report to you electronically via Internet. This license does not grant you any right to any enhancement or update to the document.
3. **TITLE**. Title, ownership rights, and intellectual property rights in and to the document shall remain in REAL-TIME CONSULT and/or its suppliers or evaluated product manufacturers. The copyright laws of Belgium and all international copyright treaties protect the documents.
4. **CONTENT**. Title, ownership rights, and an intellectual property right in and to the content accessed through the document is the property of the applicable content owner and may be protected by applicable copyright or other law. This License gives you no rights to such content.
5. **YOU CAN NOT**:
   - You can not, make (or allow anyone else make) copies, whether digital, printed, photographic or others, except for backup reasons. The number of copies should be limited to 2. The copies should be exact replicates of the original (in paper or electronic format) with all copyright notices and logos.
   - You can not, place (or allow anyone else place) the evaluation report on an electronic board or other form of on line service without authorisation.
6. **INDEMNIFICATION**. You agree to indemnify and hold harmless REAL-TIME CONSULT against any damages or liability of any kind arising from any use of this product other than the permitted uses specified in this agreement.
7. **DISCLAIMER OF WARRANTY**. All documents published by REAL-TIME CONSULT on the World Wide Web Server or by any other means are provided "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. This disclaimer of warranty constitutes an essential part of the agreement.
8. **LIMITATION OF LIABILITY**. Neither REAL-TIME CONSULT nor any of its directors, employees, partners or agents shall, under any circumstances, be liable to any person for any special, incidental, indirect or consequential damages, including, without limitation, damages resulting from use of OR RELIANCE ON the INFORMATION presented, loss of profits or revenues or costs of replacement goods, even if informed in advance of the possibility of such damages.
9. **ACCURACY OF INFORMATION**. Every effort has been made to ensure the accuracy of the information presented herein. However REAL-TIME CONSULT assumes no responsibility for the accuracy of the information. Product information is subject to change without notice. Changes, if any, will be incorporated in new editions of these publications. REAL-TIME CONSULT may make improvements and/or changes in the products and/or the programs described in these publications at any time without notice. Mention of non-REAL-TIME CONSULT products or services is for information purposes only and constitutes neither an endorsement nor a recommendation.
10. **JURISDICTION**. In case of any problems, the court of BRUSSELS-BELGIUM will have exclusive jurisdiction.

Agreed by Mr Huang Wang on Sep 24 1999

1 Introduction

What is a real-time operating system (RTOS)? There are a lot of misconceptions on the topic of real time. Following reflects the opinion of Real-time Consult on what makes a good RTOS.

We will examine techniques that can be found in general purpose operating systems (GPOS) and explain why they can or cannot be used in real-time operating systems (RTOS).

2 Real-time systems & real-time operating systems

What is a real-time system? Different definitions of real-time systems exist. Here we give just a few:

- Real-time computing is computing where the system correctness depends not only on the correctness of the logical result of the computation but also on the result delivery time.
- DIN44300: The Real-time operating mode is the operating mode of a computer system in which the programs for the processing of data arriving from the outside are permanently ready, so that their results will be available within predetermined periods of time. The arrival times of the data may be randomly distributed or be already a priori determined depending on the different applications.
- Koymans, Kuiper, Zijlstra – 1988: A Real-Time System is an interactional System that maintains an on-going relationship with an asynchronous environment, i.e. an environment that progresses irrespective of the RTS, in an uncooperative manner.
- Real-time (software) (IEEE 610.12 - 1990): Pertaining a system or mode of operation in which computation is performed during the actual time that an external process occurs, in order that the computation results may be used to control, monitor, or respond in a timely manner to the external process.
- Martin Timmerman: A real-time system responds in a (timely) predictable way to unpredictable external stimuli arrivals.

To build a predictable system, all its components (hardware & software) should allow realizing this requirement. Traffic on a bus for example should take place in a way that all events could be managed within the prescribed time limit. An RTOS should have all the features necessary to be a good building block for a RT system.

However one should not forget that a good RTOS is only a building block. Using it in a wrongly designed system may lead to a malfunctioning RT system. A good RTOS can be defined as one that has a bounded (predictable) behavior under all circumstances of system load (simultaneous interrupts and thread execution).

In a RT system, each individual deadline should be met. There exist different categories of real-time systems :

- hard real-time: missing a deadline has catastrophic results for the system;
- firm real-time: missing a deadline has a non acceptable quality reduction as a consequence;
- soft real-time: deadlines may be missed and can be recovered from. The reduction of system quality is acceptable;
- non real-time: no deadlines have to be met.

A transactional system is NOT a RT system. Indeed, performance is defined in statistical terms such like x transaction/s average should be sustained. If however, the requirement for such a system looks like x transaction/s average with a maximum of y fractions of a second for each transaction, then we have a RT system constraint due to the maximum time limit imposed on the transaction!

3 System architecture

3.1 OS structures

3.1.1 Monolithic operating system

The OS is a piece of software that can be designed in different ways. 20 years ago, the OS was just one piece of code composed of different modules. One module calls another in one or more ways. This is called a monolithic OS. This type of OS has the problem to be difficult to debug. If one has to change one module, then the impact on other modules may be great. If one fixes a bug in one module, other bugs in other modules may show up.

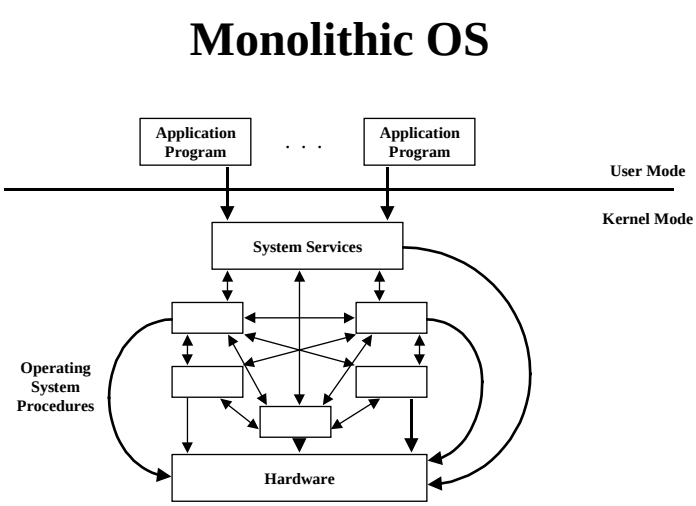

Figure 1 Monolithic operating system

The more modules, the more interconnections between modules, the more chaotic the software becomes due to the multiple interconnections. This is sometimes referred to as spaghetti software. With this design, it is almost impossible to distribute the OS in a way or another on multiple processors.

3.1.2 Layered operating system

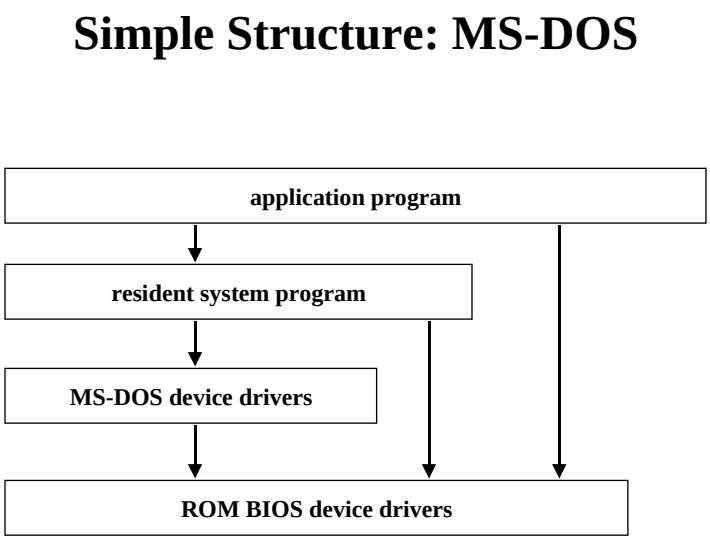

Figure 2 MS-DOS system architecture 1

A better approach is to use a “layered structure” inside the OS, like the well-known OSI layers. However OS technology, for performance reasons, is not that orthogonal with its layers as the OSI technology. In OSI you may NOT skip one layer. You may therefore easily replace one layer without effect on the others. This is not the same in OS technology. A “system call” goes directly to each individual layer. In RTOS one wants even to go directly to the hardware. The OS software is in most cases as chaotic as in the previous “monolithic” approach.

To clarify this statement the next figures show the MS-DOS approach. Figure 2 gives the impression of a well-organized OS.

However by redrawing it like in Figure 3 we see how things are really organized. It is indeed a layered structure, but several shortcuts exist: an application can access the BIOS or even the hardware directly.

Simple Structure: MS-DOS (2)

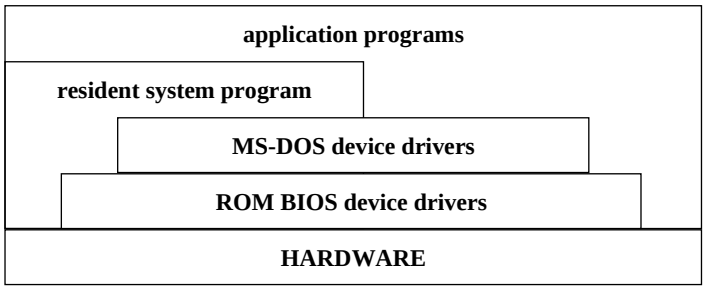

Figure 3 MS-DOS system architecture 2

RTOS have fundamentally been designed in this way for a long time.

3.1.3 Client-server operating system

A new approach has been observed in the last 5 years: Client-Server technology in OS. As demonstrated in Figure 4, the fundamental idea is to limit the basics of the OS to the strict minimum (a scheduler and some synchronization primitive). The other functionality is on another level, implemented as system threads or tasks. A lot of these “server” tasks are responsible for different functions or “system calls”. The applications are clients requesting services from the server via system calls.

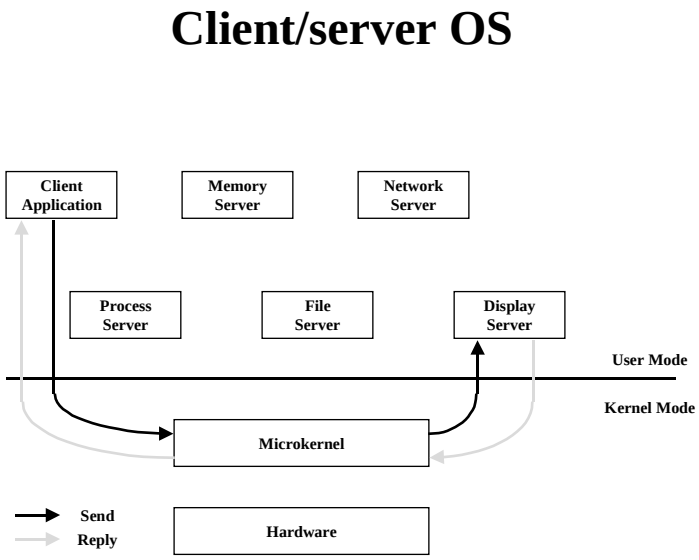

Figure 4 Client/Server operating system

This way of doing makes it a lot easier to the OS vendor to sell a “scalable” OS (with less or more functions). It is more easily debugged (each “object” remains small). Distributing over multiple processors is simple. Replacing one module does not have a “bug snow ball effect”. One module crash does not necessarily crash the whole system, meaning it is an investment in a more robust environment. It is also more conceivable to implement redundancy in the OS in this way. Dynamic loading and unloading of modules becomes a possibility.

The major problem of this model is the overhead due to memory protection. The server processes have to be protected. Every time a service is requested, the system has to switch from the application’s memory space to the server’s memory space. The time it takes to switch from one process to another one will increase when the processes are protected from each other. On the other hand, if no protection is offered, a bug in the application might affect the system processes, which could compromise the stability of the system.

3.2 Process – Thread – Task model

The first RTOS, more than 20 years ago was produced by DEC for the PDP family of machines. A multitasking concept is essential if one wants to develop a good real-time application. Indeed, an

application has to be capable of responding in a predictable way (definition of real-time) to multiple simultaneous external events (arriving in an uncontrolled way). If you only use one processor to do so, you have to introduce a sort of pseudo parallelism, called multitasking. Rate Monotonic Scheduling (RMS) theories then are helping today to compute in advance the processor power you need to deal with all these simultaneous events in time.

We have one application running on a system. This application is subdivided in multiple tasks.

In the UNIX (or POSIX) world, people are talking from the very beginning about different processes in the system. In this complex systems, the context or environment for each process is very heavy, (Processor, I/O, MMU, FPP, etc.) and therefore switching from one process to another is time consuming. Two reasons changed this approach. First, producing complex, distributed software has a need for multitasking approach which is too heavy to implement with the process concept. Second, the POSIX effort wanting to put the RT and non RT world in the same basket. The notion of thread was therefore invented. It is a “sub process” or a “light weight process”. It inherits the context of a process but uses only a subset of it so that switching between threads can be done more rapidly. Also between threads, there are no security aspects because they really belong to the same environment or process.

The net result today is that we can say in a RT environment: a process is an application subdivided in tasks or threads.

3.3 Scheduling & Priorities & Interrupts

In a multitasking environment, one has to “schedule” from one task (or thread) to another. If more than one thread wants to use the processor simultaneously, one needs an algorithm to decide which thread will run first. A deadline driven scheduling mechanism would be ideal. However, the current state of technology does not permit for this. A replacement can be found in pre-emptive priority scheduling, taking into account the existence of theories like RMS to give you a decision rule which priority level to assign to each thread. Pre-emption should be used all the time to assure that a high priority event can be dealt with before any other lower priority event. For this we do not only need a pre-emptive priority scheduling mechanism, but also the interrupt handling following different simultaneous interrupts, should be handled in a pre-emptive way.

Furthermore, we have to notice that each OS needs to disable the interrupts from time to time to execute critical code that should not be interrupted. The number of lines of code executed should be limited to a minimum to have minimum interrupt latency, but more essential, should be bound under all circumstances.

There are different reasons to have a “lot” of priority levels provided in the RTOS. The first one comes from what we discussed in paragraph 3.1 and 3.2. In a client-server OS environment, the system itself can be viewed as one or more server applications subdivided in threads. Therefore it is necessary that a number of high priority levels can be devoted to system processes and threads.

The second one comes form RMS scheduling theory. In a complex application with a large number of threads, it is essential to be able to put all the real-time threads on a different priority level. The non real-time threads can be put on one level (lower than the real-time ones) and may run in a round-robin fashion. A level 0 priority or lowest level is necessary to implement the necessary idle monitor to measure the available processor power.

4 Basic system facilities

4.1 Tasking Model

4.1.1 General

To understand the reasons of this study we need to take a look at the different phases of the system development methodology and observe where the characteristics of the RTOS emerge.

Four fundamental development phases come to light:

- Analysis: determines WHAT the system or software has to do;
- Design: HOW the system or software will satisfy these requirements;
- Implementation: DO IT, i.e. implement the system or software;
- Maintenance: USE IT, i.e. use the system or software.

In a waterfall model, one supposes that all these phases are consecutive. In practice, this is never possible. Most developments end up being chaotic, where all the phases are executed simultaneously. Adopting a pragmatic approach, the methodology used should just be a framework to guide the making of the correct documents, doing the adequate reviews and audits at the appropriate time.

In real-time systems, both hardware and software are dealt with in what is called today a co-design process. In such a process, the phases can be refined as follows:

- Feasibility study: how much effort will it take to build the required system;
- System analyses (SA): WHAT is the system going to do: draft refined or detailed requirements;
- System architectural design (SAD): HOW will we realize the requirements by defining subsystems working in (real) parallel;
- Subsystem software analyses (SSA): WHAT is one subsystem going to do;
- Subsystem software architectural design (SSAD): similar to system architecture design, but here we define a pseudo-parallelism in a multitasking model;
- Software detailed design (SDD): design all the tasks in the system by subdividing them in modules
- Implementation: code writing, debugging, testing, integration of the subsystem;
- System integration: integrating all subsystems;
- System delivery.

As seen 2 important steps are concerned with architectural design. In both these steps, the OS as a building block is an important factor. In the SSAD – the multitasking capabilities of the OS is important. In the SAD, the capability of supporting multiple processor architectures, interconnected in different ways, is important.

In this paragraph, we area concentrating on the multitasking model (SSAD).

We should be capable of doing SSAD without knowing the RTOS used. However, actual commercial RTOS vendors have made choices and one has to live with the possibilities and limitations of the actual products.

All products are different in the choices made. This means that a SSAD will largely depend on the RTOS chosen. This also means that porting the application to another RTOS environment is just an illusion, even if the RTOS is POSIX compliant.

4.1.2 Categories

4.1.2.1 Model

As stated in 3.2, today we may have different models for multitasking. Basically, on RTOS will consider the application for the system as a (non-defined) process, which is subdivided in tasks. Others may use a POSIX like model with processes subdivided in threads. Nevertheless, each process should be considered as a different application. Not too much communication should exist between these applications and in most cases they are of different nature: hard real-time, soft real-time and non real-time.

The use of one or another model depends on the system application. Under all circumstances, an application should first be subdivided in at least 3 subsystems: the hard real-time part, the soft real-time part and the non real-time part. Of course, the system might be as simple as to having only one of these.

If the system is simple, (just hard or soft RT), then one does not need a process/thread model. Just having tasks in the system is enough.

If however the system is complex and has at least 2 or 3 subsystems with different RT behavior, then a process/thread model is a solution.

One should remember that a simple system with just a task model would have better RT response compared to a more complex system with a process/thread model.

4.1.2.2 Priority levels

For the hard real-time part of the system one has to find a method to make the system predictable in all circumstances. The best solution should be to use a deadline driven scheduler. However we need an extra 10 years before we could think about using such technology in a commercial environment.

For the time being, we have to stick to pre-emptive priority driven schedulers. This introduces a problem: how can we know that the system will react in a predictable way under all possible circumstances?

Liu and Layland started in 1971 to work at an answer for this question. They invented what is now called Rate Monotonic Scheduling (RMS). It is a formal (mathematical) method to prove what should be the condition to have a predictable system. You need a fixed (pre-emptive) priority scheduling for implementing the theory. One can, by example demonstrate, that a variable priority scheme is probably a better solution, but nobody has yet come up with a mathematical foundation of how to do so and to be sure that it will work under all circumstances.

When using RMS, you need a different priority level for each real-time task. If the system is complex, you might need a lot of priority levels. We therefore state that at least 128 available (fixed) priority levels is a must.

4.1.2.3 Bounded dispatch time

When the system is not loaded, there will be only one thread waiting in the ready state to be executed. With higher loads, multiple threads might be in the ready list. The dispatch time should be independent of the number of threads in the list.

In good RTS design, the list is organized (taking into account the thread priorities) when an element is added to the list so that when a dispatch occurs, the first thread in the list can be taken.

4.1.2.4 Max. number of tasks (threads, process)

A task, thread or process may be considered as on OS object. Each object in the OS needs some memory space for the object definition. The more complexity the object, the more attributes it will have, the bigger the definition space will be. If there is for example an MMU in the system, the mapping tables are extra attributes for the task and more system space is needed for all this.

This definition space may be part of the system or part of the task. If it is part of the system, then in most cases the RTOS would like to make a reservation of the maximum space it would allocate to these tables. In this case a maximum number of tasks which may coexist in the system is then a system parameter. Another approach could be a total dynamic allocation of this space. The maximum number of tasks is then only limited by the available memory in the system shared among object tables, code, etc.

4.1.2.5 Scheduling policies

The scheduler is one of the basic parts of an OS. It has the function to “switch” from one task, thread or process to another. As this activity is considered overhead, it should be done as quickly as possible.

To understand scheduling one should know that a task has different states. At least 3 states are needed to make an OS run smoothly: running, blocked and ready.

A task is running if it is using one processor to execute the task code. If it has to wait for another system resource, then the task is blocked (waiting for I/O, memory, etc..) once the missing resources have been allocated to the task, the tasks wants to run. As different tasks probably want to run simultaneously and only as many tasks can run as available processors, one needs a “waiting for run” queue. A task being in this queue is called ready. The queue is called the “ready list. In a symmetrical multiprocessor system, there is only one queue for all processors. In other architectures, you have one queue per processor.

If there is more than one task in the ready queue, you need a decision-making algorithm: which task can use the processor first. This is also called the scheduling policy.

There are probably as much policies as engineers inventing them. Therefore we have to limit ourselves to the one’s really useful in RT systems.

In all RT systems, we need a deadline driven scheduling policy. However this is still under development and not commercially available today.

A pre-emptive priority scheduling policy is a minimum requirement. You cannot develop a (hard) predictable system without it! If you apply RMS, then each task should have a different priority level.

In more complex systems, only part of the system is hard real-time, other parts will be soft or non-real-time. The soft real-time parts should be designed like the hard RT part, but taking into account that not always all the needed processor power will be available. The same scheduling policy is applicable. In the non-RT part, then a more general purpose OS (GPOS) approach may be desirable. In GPOS systems, the philosophy is “maximum usage of all system resource”. This philosophy is contradictory with the RT requirement of being predictable.

If one wants to give each task a equal share of the processor, a round robin scheme is more appropriate.

Process states & transitions

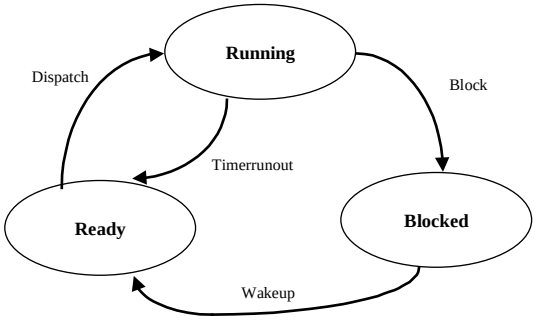

Figure 5 Process states & transitions

The non real-time part in a complex system should therefore be capable of using a RRS. Most RTOS implement this when you put more than one task on the same priority level. Other RTOS have a RRS explicitly defined for some ranges of priorities.

Conclusion: an RTOS should always support pre-emptive priority scheduling. For complex applications, where for some parts of the system, a more GPOS philosophy is needed, RRS or some other mechanisms might be useful.

4.1.2.6 Number of documented states

In the previous paragraph, we mentioned a minimum of three states for a task. There is however no limit to the maximum number of states. Indeed, the blocked state can be subdivided in a number of blocked states specifying the reason of blocking. (waiting for I/O, waiting for semaphore, waiting for message send, waiting for a memory block…). Making a diagram mentioning all these states is a good graphical representation of what the OS is capable of. However this is never done in a systematic way.

Such diagrams might be very useful for debugging purposes. By animating the diagram during a reply, one could see how the system evolves during time. Such a tool is not available today.

4.1.2.7 Min. RAM required per task

Memory footprint is an important issue in embedded system despite the cost reduction of silicon and disk memory today. The size of the OS is important, the system space necessary to run the OS with all the objects defined (see 4.1.2.4). A task needs RAM to run for the changing parts of the task control block (the task object definition) and for stack and heap to be capable of executing the program (which might be in ROM or RAM). It is the RTOS vendor's choice to allocate a minimum number of RAM for this. It is also an indication of the vendor of how big he sees the minimum application wherefore the RTOS is intended.

4.1.2.8 Max. addressable memory space

Each task can address a certain memory space. Each vendor may have a different memory model depending if he relies on X86 segments or not. This depends largely on the product history of the RTOS. If it was initially developed for the flat address space of the MOTOROLA processors or for the segmented X86 INTEL series. The limitation of 64K segments in a 8086 processor is an important limitation for modern software. RT systems needing to be backward compatible with this type of hardware is a burden to the designer. Vice versa, it should be carefully observed that this limitation is not imposed by the RTOS, making new designs possible outside the 64K space per module.

4.2 Memory

Each program needs to reside in memory in order to be executed. If we accept that it is not wise to develop auto-modifying code – this program may be in ROM.

Each program using a modular approach will use subroutines and therefore needs a stack, which should be in RAM. A program makes no sense if it does not manipulate some input data to produce some output. These “variables” also have to reside in RAM.

The fundamental requirement about memory in a real-time system is that the access time to it should be bound (= predictable). As a direct consequence, the use of virtual memory is prohibited for real-time processes. This is the reason why systems providing a virtual memory mechanism should have the ability to “lock” the process into main memory so that swapping does not occur. (Swapping is a mechanism that cannot be made predictable!)

Second, if paging is supported, the associative map for the pages should be part of the process context and therefore be completely loaded in the processor or MMU. In the other case, the system is based on a statistical phenomenon which is non-acceptable in hard real-time.

As a result, the whole story is to know if you are going to use static or dynamic memory allocation in your design.

Combined memory protection & dynamic relocation

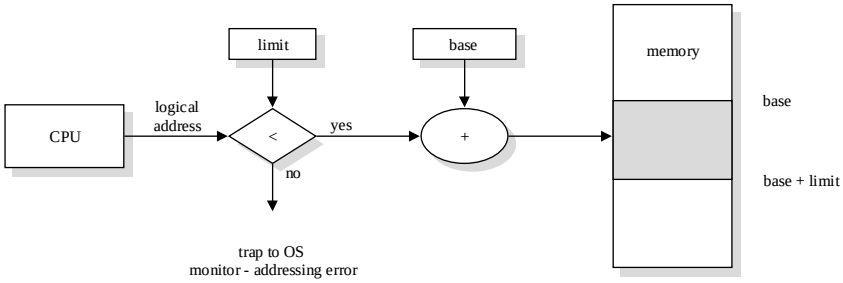

Figure 6 Memory protection & dynamic relocation

Static memory allocation means that all memory is allocated to each process or thread when the system starts up. In this case, you never have to ask for memory during the execution of the process. This however may be very costly. When hard real-time is not required, you can imagine to use a dynamic allocation mechanism. This means that a process, during runtime, is asking the system for a memory block of a certain size to house one or another data structure. (You never ask memory for a piece of program in a hard RT-system). In this approach, the designer should know what to do if the memory block doesn't become available in time. Some RTOS support a timeout function on a memory request. You ask the OS for memory within a prescribed time limit. The task is waiting for the memory as long as the timeout is not expired. This feature may significant reduce the application code.

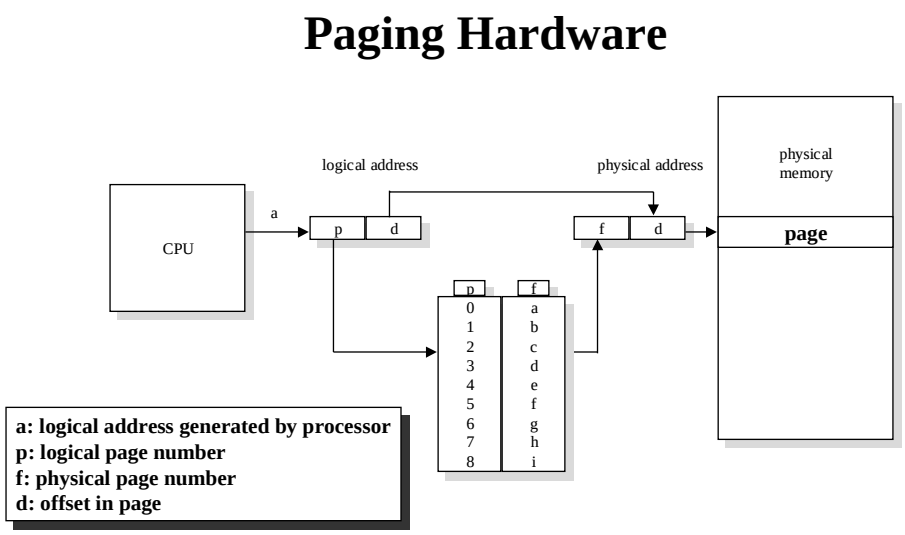

Figure 7 Paging hardware

In some circumstances, it may not be acceptable that a hardware failure corrupts data in memory. Then the use of a hardware protection mechanism is indicated.

Figure 6 shows that memory relocation is possible with a change of the base address and that a protection mechanism is simple. Each logical address has to be in the range [base, base+limit] otherwise there is a memory protection failure.

This means that the hardware will check if the access to a certain block in memory is allowed to this process. This hardware protection mechanism can be found in the processor or in a MMU. Today's MMUs however do much more than just protecting memory blocks. They allow also for address translation, something we do not need in RT because we use (cross) compilers generating PIC code (Position Independent Code).

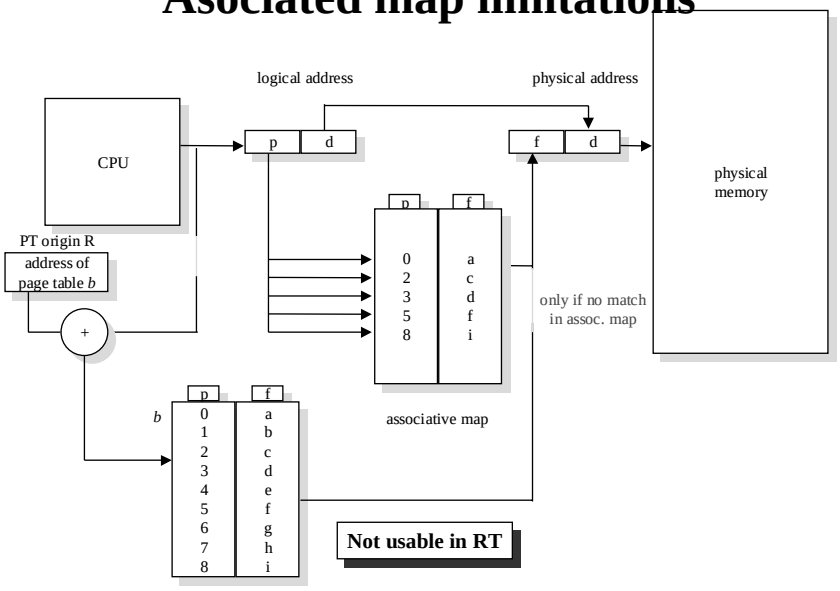

Figure 8 Associated map limitations

A potential drawback of a paged-MMU mechanism is the page size. Indeed in GPOS the page-size is often 2K and the paging mechanism (see Figure 7) is connected to the memory translation mechanism (logical – physical addressing). This introduces the need of an associative map in the MMU. This associative map has a limited size and a sort of caching between the page tables in memory and this associative map is used to deal with this (see Figure 8). As stated before, this mechanism introduces unpredictability.

Suppose we use a paged MMU for memory protection reasons, then the protection is connected to the pages and so to the page size. To make it predictable one can imagine making the associative map part of the task context. However if the page-size is limited to 2 K, the number of pages needed for the whole program and data area is too high to fit in the associative map. One needs therefore a much larger page-size.

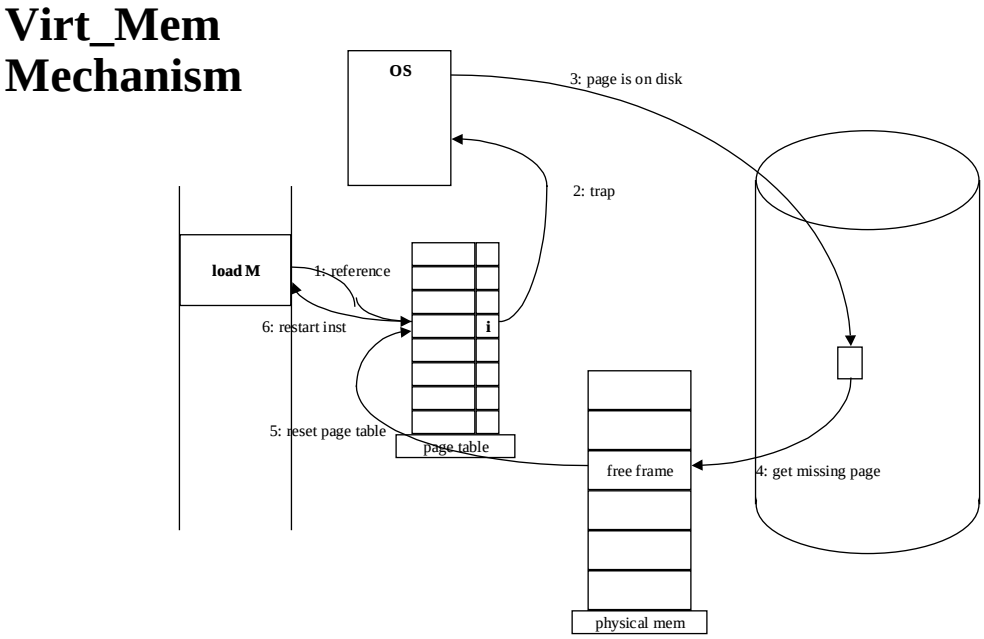

Figure 9 Virtual memory mechanism

To avoid this problem, one can use a segmented approach. Segments have variable sizes and can be much larger than 2 K, however another problem arises here. If segments, with variable sizes are allocated and de-allocated, external memory fragmentation will be the result. To clean up memory, some OS use compaction of garbage collection. However, this can again not be used in a RT environment. Indeed during the compaction procedure, the tasks that are displaced cannot run. This system becomes unpredictable. This is the major problem if object orientation is used. Therefore C++ and JAVA are wrong candidates for hard real-time tasks today as long as the compaction problem is not solved.

Virtual memory (Figure 9) is another technique that cannot be made predictable, and should therefore not be used in real-time systems.

A simple solution is to allocate all memory for all objects you need during the system life and never de-allocate them. Another solution resides in allocating and de-allocate always fixed size blocks of memory (introducing internal fragmentation = never using some parts of memory internal to the blocks).

From the previous discussion we see that RT memory management and GP memory management have different objectives and mechanisms. It is difficult to join them.

In simple systems which is or hard or soft or non real-time, the choice is simple. In complex systems where both hard, soft and non real-time functionality's are required, a good but expensive solution is to run each subsystem on a different processor. However, today, the available cheap processor performance is so high that one might want to put on the same platform all functionality (hard, soft & non real-time).

In HRT you use static memory allocation. In SRT you may accept dynamic memory allocation – no virtual memory – no compaction. In non-RT you may want virtual memory and compaction.

If we accept that the non-RT task are put on a much lower priority than the SRT and than the HRT, then the question still may be: how pre-emptable is a virtual memory mechanism and the compaction mechanism. This can only be detected by testing the available RTOS.

Each RTOS is different in the memory allocation possibilities. Indeed, one RTOS is targeted towards one type of system (HRT, SRT, non-RT) or toward a mix of all.

A target system may not need a memory protection scheme. However, during system development, it might be interesting to have MMU support around. Indeed, having memory protection is a very nice debugging tool. It helps the designer programmer to debug the system easy. A stack-heap overflow may be debugged in seconds with a MMU based system, where in a non-protected system you might need one week to find the error. Also, it might be that the bug is there but that you never detected it during tests, except 3 years after the system is delivered…

4.3 Interrupts

4.3.1 General

Remark: in this paper we use the word interrupt for hardware interrupt. A software interrupt, together with a hardware interrupt and other vectoring mechanism provided by the processor are called exception handling (this is the MOTOROLA naming).

A RT System is supposed to react on external events within a prescribed time limit called a deadline. All these external events are translated via the hardware in one or more bit transitions somewhere.

A first method to detect the occurrence of the event is to poll from the task the “event bit” from time to time. If the system has only one external event, and has nothing to do besides waiting for that event, then this polling mechanism is the most efficient way to go. However, in a multitasking environment, the system has to deal with more than one event and cannot afford doing busy waiting. Therefore, in an OS one has decided to make the detection of the event via an interrupt with an associated interrupt service routine (ISR). This ISR maybe stand alone or part of a device driver structure depending on the RTOS device driver model.

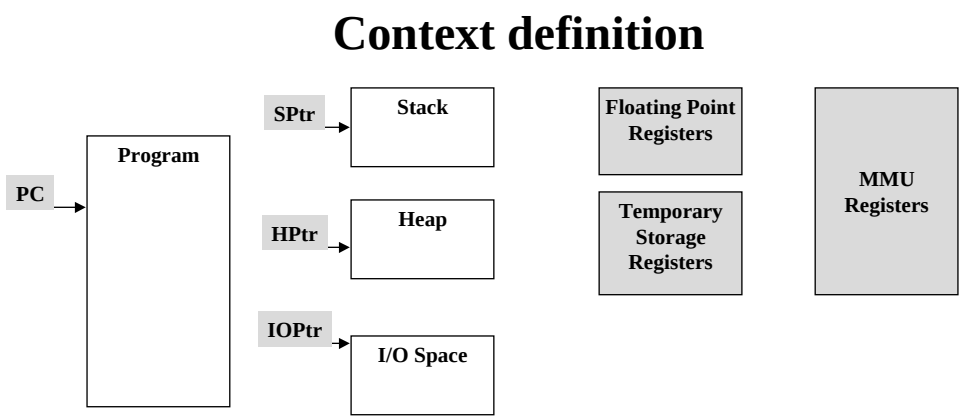

Figure 10 Context definition

Once you accept to deal with more than one event, you deal with more than one (simultaneous) interrupt. Here again, you have a problem with predictability. What happens if a higher order interrupt shows up when a lower level ISR is busy. To have a quality RTOS, the lower level ISR must be pre-emptable.

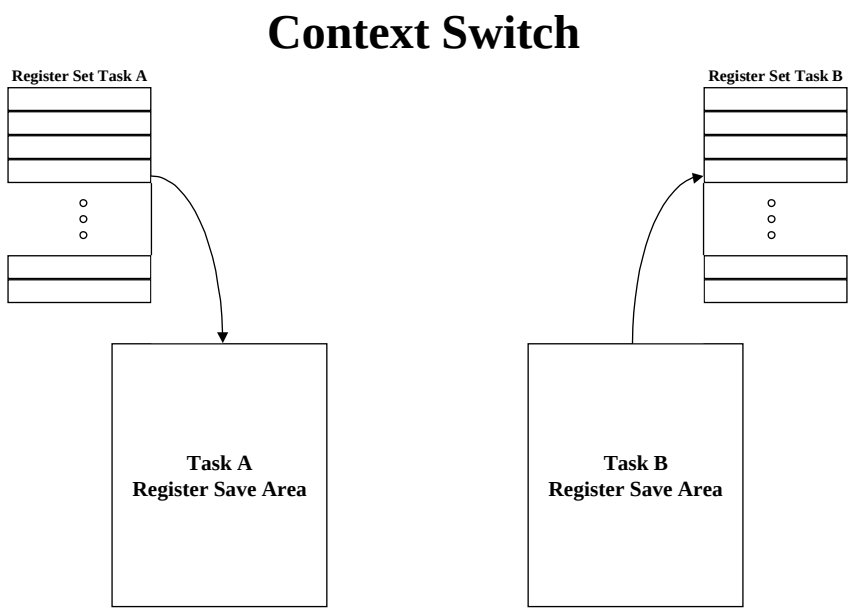

Figure 11 Context switch

Here a performance & design issue shows up. Indeed, first the RT system must be predictable, so there should be a bound time between the interrupt and the handling of the event. Second, some events need faster treatment than others (shorter deadlines).

The most straightforward method of dealing with events is to let them detect via an interrupt and then to start a task or thread dealing with the event. This means that you need certain system calls to be executed from the ISR level and therefore you need a minimum of context (see Figure 10). The time you need, taking into account simultaneous interrupts, to go from interrupt to start task is an important system metric and should be defined (see Figure 12). Tests are needed for this because the vendors never publish comparable figures on these issues.

Context Switch Timing

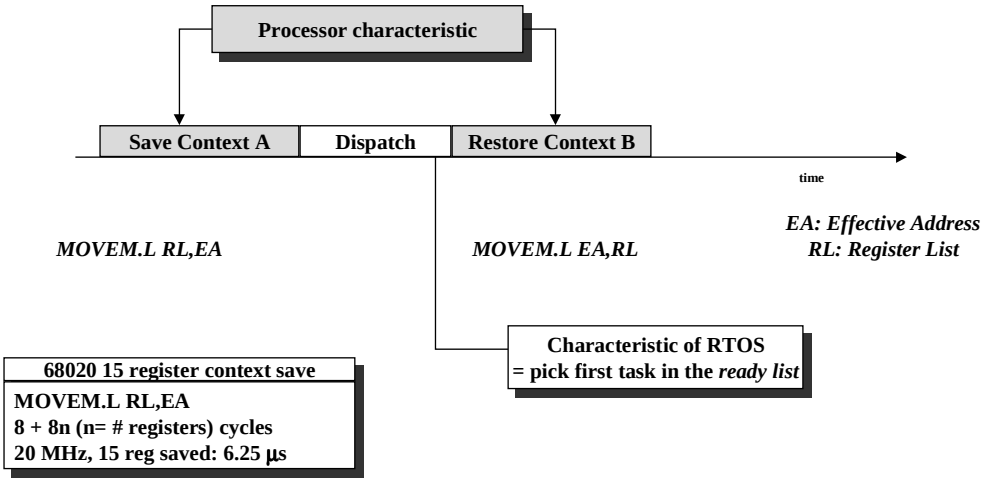

Figure 12 Context switch timing

Under some circumstances, one cannot accept the previous latencies because the design requires shorter deadlines than the ones you obtain with the previous technique. This can be solved by executing event code or in the ISR or in the 0 level ISR or background mode, or in the device driver. This means that there are different levels where the event code can be executed. Good design requests for the shortest possible ISRs and device driver, but you may decide to overrule this.

The different RTOS may differ in the model they have to deal with this. The simplest model is one with just a task level and an ISR level. The complex model is one with an ISR, a background ISR, a device driver and a task level. Depending on where an event is dealt with, the event handling metrics will be different (Figure 13). Important here is the context definition for each level. It is the simplest for the ISR and the most complex for the process or task level. Context switching times largely depend on the context definition and the processor speed. It is the RTOS together with the compiler, which defines the context definition on each level.

Interrupt to task run

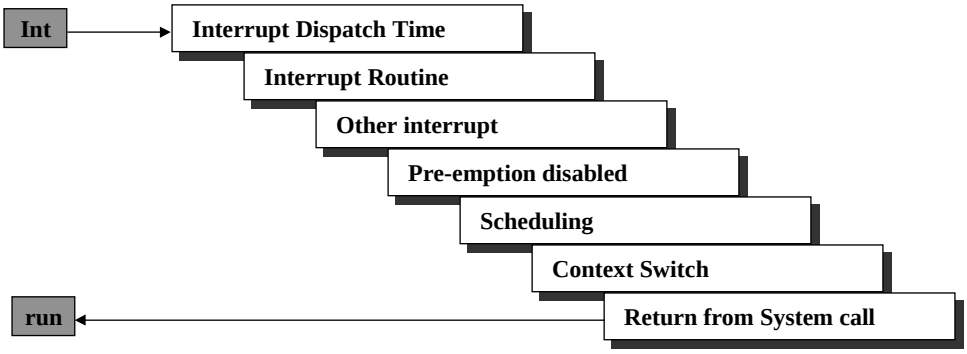

Figure 13 Interrupt-to-task run

The evaluation report should clarify which model is used by the RTOS. Figure 13 shows the different elements which are part of the total interrupt to task run time.

- Interrupt dispatch time: the time the hardware needs to bring the interrupt to the processor.
- Interrupt routine: the execution time of the ISR.
- Other interrupt: the time needed for managing each simultaneous pending interrupt.
- Pre-emption disabled: time needed to execute critical code during which no preemption may happen.
- Scheduling: time needed to make the decision on which thread to run.
- Context switch: time to switch from one context to another.
- Return from system call: extra time needed when the interrupt occurred during the execution of a system call.

The maximum time needed to go from interrupt to run is the sum of all the potential maximums of these different latencies. The fact that more than one simultaneous interrupt is to be dealt with is important. Therefore one should always try to determine during design how many simultaneous interrupts might occur!

The test shall measure the maximum of these times under all load circumstances.

5 API richness

5.1 General

5.1.1 Purpose

The purpose of this section is to go in some detail through all the available system calls of the OS commonly named the API or Application Program Interface.

Every system call is a software interrupt (SWI). The classical vectoring system of each processor is used to implement the service. The number of vectors is a limitation of the processor. The processor may have enough software vectors to implement all the system calls, but in complex OS this is rarely the case. Therefore one or another data passing mechanism will be used by the OS around the SWI to be capable of using the same vector for all or part of the system calls and also to be capable of rendering status data once the system call is executed.

The classical approach is that the program continues execution just after the SWI instruction. There, every programmer should implement error handling.

If assembler is used, the programmer will explicitly see the SWI and he will need to now the details of the data-structures and the passing mechanism of this data to the OS. When using a high level language, the SWI system calls will be implemented as procedure calls in that language. One speaks about a C or ADA or whatever interface to the OS.

The format of these procedure calls is defined by the interface library builder and can yes or no correspond to a standard like POSIX. (See 5.1.2).

The fundamental issue we want to discuss about the API is its richness. Indeed the number of system calls may be very limited. The minimum functionality you need in a RT multitasking environment is a scheduler and one synchronization primitive. All other synchronization and communication primitives or system calls can be built on that. The question however is, what part of the software is going to do the job, the application or the OS? The more system calls you have and the more complex they are, the fewer lines of codes the application will have. Even more, the code executed in the OS is certainly more efficiently executed (and better debugged) in the OS than the same code in the application. So we are interested how many system calls are supported and how complex they are. This ultimate aim is to reduce application maintenance as much as possible due to limited application code length.

Of course, a very simple application can probably do with just a few system calls. A very complex one should have a wide choice of system calls.

5.1.2 POSIX

POSIX defines amongst other things the syntax of the library calls that execute the SWI for the OS interface. This means that if everybody uses the POSIX interface, on the application side, in C, all application code should be the same. However it is not as simple as that as we will see. On the implementation side, all RTOSs are different. Each product will have other performance characteristics for the “same” system call.

POSIX is very new and there are a lot of legacy applications around. And even more: POSIX does not necessary have the best system calls for efficient implementation of certain portions of code. Therefore lots of vendors do not only support POSIX but support also a second set of non-POSIX system calls. And

this is not against the POSIX rules. Both POSIX and non-POSIX compliant calls may co-exist in the same system. This is one of the first reasons why we do not believe that POSIX compliance is an essential issue today.

Another issue is that it took a very long time to standardize and the standard has been cut in different pieces: the ones everybody can agree on immediately, the ones it is difficult to agree on and the ones they will never agree on…. Today, some parts of the standard are finalized, others are still in draft form. Therefore, different products comply with different "draft revisions" of the standard which makes the actual usefulness of being compliant problematic.

Real-Time Systems range from very small-embedded systems like modems to very complex hybrid systems like a satellite ground station. Today's RTOS are not scalable enough to deal with both the small and large systems. A particular RTOS aims at a particular target application size. The historical consequence is that a small-embedded system RTOS uses a totally different API compared to a RT-UNIX like RTOS. To deal with this, the POSIX committee invented profiles. This is stated in POSIX 1003.13. This means that if one talks about POSIX compliance, he should mention the profile. However nobody is doing so. Another point is that these profiles do not cover all systems we have today.

Features versus Profiles

Feature	Minimal	Control	Dedicated	Multi
Process Primitives	+	+	+	+
Process Environment	+	+	+	+
Files & Dir	-	+	-	+
IO	+	+	+	+
Device Specific	-	+	-	+
Data Base	-	-	-	+
Binary Semaphores	+	+	+	+
Memory Locking	i	i	+	+
Mapped Files	-	+	-	+
Shared Memory	i	i	+	+
Process Priority Sched	-	-	+	+
Real-time Signals	+	+	+	+
Clocks & Timers	+	+	+	+
IPC Msg Passing	+	+	+	+
Synchronised IO	+	+	+	+
Asynchronous IO	-	+	+	+
Prioritised IO	-	-	+	+
RT Files	-	+	-	+
Threads	+	+	+	c

i: implicit
c: configurable

Figure 14 Features versus profiles

The fundamental conclusion is that POSIX compliance is not a fundamental issue. POSIX has one advantage however: today we talk less or more the same language if we talk API's. This was not the case 10 years ago.

Problems with POSIX .13

	POSIX AEP			
	Minimal	Controler	Dedicated	Multi
Files & Dir	no	yes	no	yes
RAM Disk	no	yes	no	yes
Standard File System	no	yes	no	yes
RT File System	no	yes	no	yes
Mapped Files	no	yes	no	yes
Network	no	no	no	yes

Figure 15 Problems with POSIX 1003.13

Portability of a RT application is a myth. Doing analysis and design of a RT application in a methodological way with RT-modeling techniques can only solve it. Producing new code starting from this documentation is easier than to try to port a POSIX application from one system to another.

(RTConsult) Extended POSIX .13

	Extended POSIX .13							
	Minimal	Controler			Dedicated			Multi
		Embedded	Connected	Data Acquisition	Embedded	Connected	Data Acquisition	
Files & Dir	no	**yes**	**yes**	**yes**	**no**	**yes**	**yes**	yes
RAM Disk	no	**yes**	**yes**	**yes**	**no**	**yes**	**yes**	yes
Standard File System	no	**no**	**no**	**yes**	**no**	**no**	**yes**	yes
RT File System	no	**no**	**no**	**yes**	**no**	**no**	**yes**	yes
Mapped Files	no	**no**	**no**	**yes**	**no**	**no**	**yes**	yes
Network	no	**no**	**yes**	**opt**	**no**	**yes**	**opt**	yes

Figure 16 Extended POSIX 1003.13 by RTConsult

5.2 Categories

5.2.1 Task management

In paragraph 3.2 we explained the difference between task, threads and processes. The most complete model will support processes (= applications) subdivided in threads. The threads are the units scheduled (and executed in the system). Threads can be started, stopped and resumed. A process (when subdivided in threads) is not started nor stopped because a process is more a sort of context.

When using RMS, one has no intention to change task priority levels during execution. You therefore don't really need this feature. However, in some application areas it might be interesting to do so. Priority inheritance for example is an automatic mechanism changing the priority of a task to avoid priority inversion. It is therefore an investment for future systems to have the possibility to dynamically change the priority of a task. Important here is to know which object may do so (another task, an ISR, etc) and under what circumstances.

5.2.2 Clock and timer

Most RT systems work with relative time today. Something happens BEFORE or AFTER some other event. In a fully event driven system you don't need a ticker since there is no time slicing. However if you want to time stamp some events or if you want to introduce systems calls like "wait for one second" you need a clock and or a timer.

RT Synchronization today is done by blocking (or waiting) for an event. Absolute time is not used. This might change in the future. Indeed, you can also synchronize things by using the absolute time. This is what humans do if they decided to start a meeting at 9:00 am. Having a precise absolute time clock in the system is therefore something what we should look for in the future. Some systems need it already today (especially in space applications). The precision of the software absolute clock is in most cases not enough for these applications. A precise hardware clock should be provided, which in turn, the RTOS should support.

5.2.3 Memory management

We discussed memory management in detail in section 4.2. RTOS vendors use, (in a non-POSIX environment) different names for similar objects and the same names for different objects.

Naming in RTOS

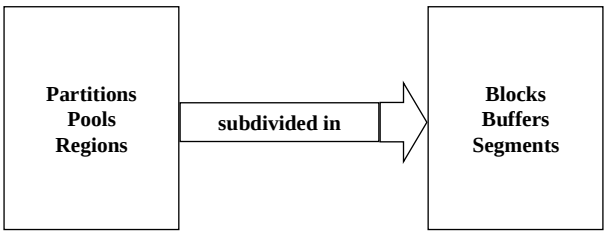

Figure 17 Naming in RTOSs

To make a good RT-design, one should know if the RTOS implements fixed or variable size blocks (buffers, segments) and if these structures are protected by some MMU mechanism.

The design should take care not to run into a fragmented memory when using dynamic allocation and de-allocation.

An important feature is the fact that the allocation of memory may be time limited or not. Indeed, if the application needs some memory to fulfil the deadline, then this memory should be available immediately when asked, or at the latest within some milliseconds. Some vendors support such a timeout as an attribute of the system call. This is again very important in reducing the application code and thus the system maintenance. Indeed, if no such feature is implemented, the RTOS will return a “not available memory” error code after the allocation request. If the system can wait for some time, the application should then take care in re-asking for that memory until “too late”. This induces a lot of complexity in the application code.

This example shows again clearly that richness of the OS calls is very important in reducing the number of application lines of code. This reduces maintenance and bug probability. The system will be more reliable!

5.2.4 Interrupt handling

A RT designer has to write his own interrupt routines (and device drivers). These modules are part of the OS. Therefore they are difficult to debug and a mistake in these leads to a major disaster. Some RTOS are trying to limit this potential disaster by not allowing the vector table to be changed by the programmer. Others just don’t care. It is difficult to state what the best solution is. The first one introduces some protection against programmer errors but introduce some extra indirect jumps and therefore overhead and interrupt handling performance reduction. The second one is fast but needs more care from the programmer.

Another issue is to look what system calls can be performed starting form an ISR.

5.2.5 Synchronization and exclusion objects:

Synchronization and exclusion is needed so that the threads can execute some critical code. The objects can also be used to make sure that some threads are executed one after the other. A variety of objects are available:

- semaphores: synchronization and exclusion;
- mutexes: exclusion;
- conditional variables (in conjunction with mutexes): exclusion depending on a condition;
- event flags: synchronization on multiple events (can contain high level logic);
- signals: asynchronous event processing and exception handling.

The rules when to use what objects are very simple:

- How do you realize the requirements by producing the minimum amount of code.
- What construct can I use to be flexible in the acceptance of design changes.

How to apply these rules is not the purpose of this document. However, if an RTOS does only support one or another primitive or object, then you are not even capable of applying the rules. Again, our statement is that the richer the RTOS API, the better. At least you will have the choice during design.

5.2.6 Communication and message passing

Communication and message passing is a form of synchronization where data exchange happens. Examples are:

- queues: multiple messages;
- mailboxes: single message.

The same rules and remarks as stated in the previous paragraph are applicable. However, when data is exchanged, an extra problem arises: is the data structure completely copied from the sender thread space to the receiver thread space or is just the pointer passed?

In most RTOS, passing the full data structure is not done for performance reasons, so a pointer goes from the sender to the receiver.

Here we have another design issue: are you sure that in all circumstances the pointer is still valid when the thread is using it? This document is not the place to go into these details, but the reader should be aware of the problem.

Another issue is: “does the object accept single or multiple senders and receivers”?

By stating all these issues, one thing becomes clear: each RTOS behaves differently with apparently the same objects. It is our aim to understand the behavior by testing and to publish the results in the evaluation document.

5.2.7 Waiting list length

As stated, synchronizing means blocking or waiting on a synchronization object. A very important feature in a good RTOS is that the temporal behavior of the system does not depend on the length of these waiting lists.

Good RTOS design means that for each thread that starts pending on an object, the list is reorganized at that moment, so that the time it takes to release the object is independent of the queue list length.

6 Development methodology

6.1 Introduction

It is important that an RTOS provides the users with an efficient way to develop applications. The availability of good and efficient tools plays an important role in the development process, but there is more. Different design philosophies exist, each with their own perks and pitfalls. Operating systems can use different configurations of *host* and *target*. The *host* is the machine on which the application is developed, while the *target* is the machine on which the application executes.

6.2 Host = Target

In this configuration, the host and target are on the same machine. The RTOS has its own development environment (compilers, debuggers) and its own command shell. In such configuration, there are no connection problems between host and target. However, the development environment is sometimes of lesser quality, since the vendor of the OS often does not have sufficient resources to develop both the operating system and the development environment. Furthermore, the RTOS does not have all the features available in a general purpose operating system (GPOS) which facilitate development (e.g. source code control system, backup tools, …).

6.3 Host ≠ Target

In this case, host and target are two different machines linked together (e.g. serial link, LAN, bus, ..) for communication. The host is a machine with a proven GPOS, which is often more suitable as a host than a machine with a dedicated RTOS. This situation often allows for a better and more complete development environment, since all the all the features of the host can be used.

The drawback of this configuration is in debugging. The debugger is on the host, while the application is executed on the target. So called debug agents are residing on the target to communicate the debug information to the host.

When host and target are different machines, the development environment should provide simulators that allow developers to execute a prototype of their application on the host by simulating the target. The target can be simulated in 2 ways: by simulating the target microprocessor or by simulating the target RTOS (API).

6.4 Hybrid solutions

The hybrid solutions are trying to combine the best of both worlds. The host and target are on the same physical machine, but they run on different operating systems communicating with each other in some way (e.g. by shared memory). The host OS is the rich and proven GPOS, while the target OS is the dedicated RTOS.

In this situation, the application can be developed using all the tools available in the GPOS, and since the RTOS and the target application code run on the same hardware, communication between the two is not a big issue.

In reality however, these hybrid solutions have their own set of problems. The same hardware recourses are shared by 2 operating systems (the GPOS and the RTOS), sometimes keeping the RTOS from

exhibiting predictable real-time behavior under all circumstances. The other way around, the RTOS monopolizing the processor could keep the GPOS from performing its housekeeping chores, compromising the stability of the whole system. Multiprocessor architectures should improve things.

7 Conclusion

The purpose of this document has been to explain the reasons of testing commercial RTOS by first defining what we think are essential features for a good RTOS.

An RTOS, being a building block of a RTSystem should have in all circumstances a predictable behavior. This means a behavior that is bounded in time independent of the load in the system and the length of the queues in the system.

The purpose of the evaluation project is to detect by analyzing the documentation and by executing extensive tests if all the requirements for "a good RTOS" are met.