Technical Paper :

"New Directions in RTOS Kernels"

OVERVIEW

Real-**T**ime **O**perating **S**ystems ("**RTOS**s") are nowadays a technology that is pretty stable and mature. The basic features and characteristics of RTOSs are well-understood by the embedded software development community that uses them. However, this mature technology continues to advance and change --- although much more in an evolutionary than a revolutionary manner. This paper will focus on some of the directions in which RTOSs continue to change and evolve. In order to keep this discussion at a practical level, it will be restricted to trends that are already underfoot, and to features that we might reasonably expect to appear in RTOSs within a decade.

These days, RTOSs are quite large and complex pieces of software that provide a wide variety of services. These services together form an “abstraction layer” that allows embedded application programmers to do many useful things by simply making requests to the RTOS. Services include network communications, file systems management, distributed systems management, redundancy management, and dynamic loading of application software. The most basic services reside in a part of the RTOS called the “**kernel**”. See Figure 1. I would like to focus on kernel services in this article. All of the more sophisticated services (networking, file systems, etc.) are built on top of the kernel and rely upon it for their basic service needs. Application software developers also rely on RTOS kernels for basic services such as task scheduling and inter-task communications.

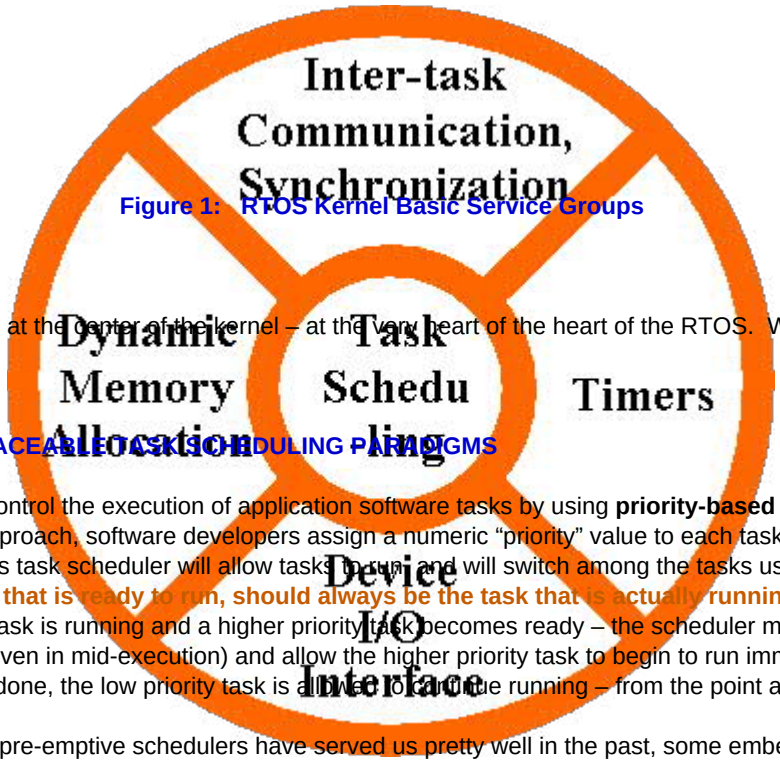

Figure 1: RTOS Kernel Basic Service Groups

The task scheduler is at the center of the kernel – at the very heart of the heart of the RTOS. We begin our foray into the future there.

MODULARLY REPLACEABLE TASK SCHEDULING PARADIGMS

Most RTOSs today control the execution of application software tasks by using **priority-based pre-emptive** scheduling. In this approach, software developers assign a numeric "priority" value to each task in their application software. The RTOS's task scheduler will allow tasks to run, and will switch among the tasks using the rule that **"The highest priority task that is ready to run, should always be the task that is actually running"**. In other words, if a relatively low priority task is running and a higher priority task becomes ready – the scheduler must immediately stop the low priority task (even in mid-execution) and allow the higher priority task to begin to run immediately. When the higher priority task is done, the low priority task is allowed to continue running – from the point at which it was stopped.

While priority-based pre-emptive schedulers have served us pretty well in the past, some embedded developers bring up objections to them. For example, low priority tasks may suffer "**starvation**" (a total lack of opportunity to run) under a priority-based pre-emptive scheduler --- if higher priority tasks have sufficient work to consume all available processor time. Or even when tasks do not totally "starve", a priority-based pre-emptive scheduler might cause low priority tasks to run only after long delays. These delays could be so long that the outputs of the task would be useless by then. Such delays are called "**schedulability problems**". Another objection that can be heard is ***"Where do I tell the scheduler about my real-time deadlines??"***. In fact, there is no way to tell a priority-based pre-emptive scheduler about software deadlines. Priority-based pre-emptive schedulers can ensure that application software deadlines will be met only in very limited cases. [You can learn about these cases in a subject area called "Rate Monotonic Scheduling Theory".]

To answer the above objections, several alternatives to priority-based pre-emptive scheduling have begun to appear on the RTOS horizon. The first is called "**Deadline Scheduling**". In this approach, the RTOS kernel's task scheduler is provided with information about task deadlines, and it temporarily raises the priorities of tasks as they approach their deadlines if they have not yet run. In this way, the deadline scheduler urgently strives to get tasks to run before they miss their deadlines, by pre-emptively "borrowing" running time from tasks that normally have higher priorities. [Hopefully, this will not cause the "lender" tasks to then miss their own time deadlines.]

Deadline schedulers are of interest in DSP and multi-media applications where crisp timing and consistency of task execution timing are most critical.

There are a number of ways for an RTOS to do deadline scheduling. The simplest is called **Earliest Deadline First** ("**EDF**") scheduling. In EDF scheduling, the task that is nearest to its deadline (and has not yet run) will be allowed to run first. Thus, an EDF scheduler views task deadlines as more important than task priorities.

A somewhat more complex deadline scheduler is the "**Least Laxity**" (or "**LL**") scheduler. It takes into account both a task's deadline and its processing load, in order to handle situations such that seen in Figure 2.

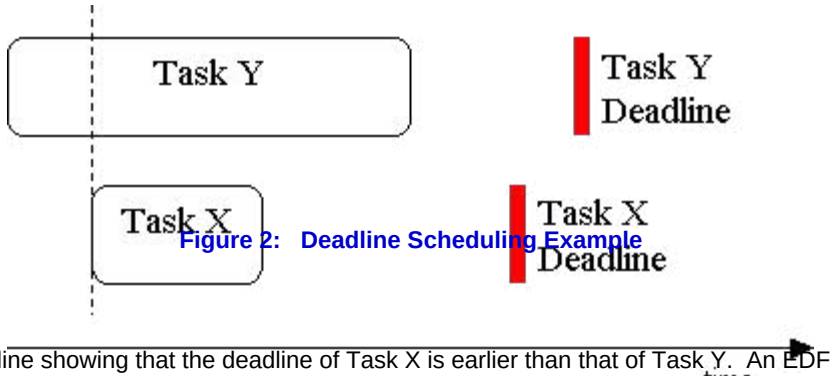

Figure 2: Deadline Scheduling Example

Figure 2 is a timeline showing that the deadline of Task X is earlier than that of Task Y. An EDF deadline scheduler would allow Task X to run before Task Y, even if Task Y normally has higher priority. However, doing so could well cause Task Y to miss its deadline. So perhaps an "**LL**" scheduler would be better?

An "**LL**" scheduler evaluates the urgency of tasks using a value called "**Laxity**", where:

Laxity == (Task Deadline – Task Execution Time).

Laxity is the amount of time that would be left before a task's deadline if it ran to completion immediately. In some sense, it is the amount of time that the scheduler can "play with" before causing the task to fail to meet its deadline. When the "LL" scheduler has evaluated the Laxity for all tasks, it finds the task with the smallest current value of Laxity – and that is the task that needs to be scheduled to run next.

Thus, a Least Laxity deadline scheduler takes into account both deadline and processing load. "LL" scheduling, while excellent for highly time-critical tasks, might be overkill for less time-sensitive tasks. And so there is a third interesting variant of deadline scheduling, called "**Maximum Urgency First**" (or "**MUF**") scheduling. It is really a mixture of some "LL" deadline scheduling, with some traditional priority-based pre-emptive scheduling.

In "**MUF**" scheduling, high-priority time-critical tasks are scheduled with "LL" deadline scheduling, while within the same scheduler other (lower-priority) tasks are scheduled by good old-fashioned priority-based pre-emption. It provides crisp deadline scheduling for time-critical software, while supporting the majority of tasks with low-overhead traditional pre-emptive scheduling.

To summarize our foray into the realm of deadline scheduling, we have seen three interesting deadline scheduling strategies:

- **EDF** Earliest Deadline First scheduling
- **LL** Least Laxity scheduling
- **MUF** Maximum Urgency First scheduling.

However, a word of caution must be added here: The failure modes of a deadline scheduler may be unacceptable for your application. So, please approach deadline schedulers with caution. In other words: ***"When a deadline scheduler works well, it works very well. But when a deadline scheduler works badly, it can work very badly."*** For example, when an overloaded deadline scheduler makes a valiant attempt to get one task to meet its deadline, it may as a result cause one or more higher priority tasks to miss their deadlines. Thus, a very careful CPU loading analysis should be done before bringing a deadline scheduler into your project. [The math for this is part of "Rate Monotonic Scheduling Theory".]

But deadline scheduling is not the only future alternative to today's priority-based preemptive scheduling. While it' s good at improving schedulability for time-critical tasks, it does not address the problem of task starvation. Superior management of task starvation concerns is offered by futuristic "Partition Schedulers".

When using a "**Partition Scheduler**", tasks are collected together into groups known as "partitions" (sometimes also known as "processes" or "blocks"). Typically, a "partition" will consist of a number of tasks that work together to implement a major embedded application, such as data acquisition or motion control. See Figure 3. For the partition scheduler, each partition is assigned one or more windows of execution time in a repeating time line. Within each time window along the time line, only the tasks in the partition assigned to that window can run. All tasks in all other partitions are not allowed to run. Tasks in the active partition for the window are typically scheduled according to conventional task scheduling, such as priority-based pre-emption within the window. Thus when a certain application partition's window is active, the group of tasks in that partition has guaranteed access to the processor's CPU, as a

group. Another application cannot run and take processing time from this application during its window. Thus, starvation of access to the processor is prevented, at the level of groups of tasks.

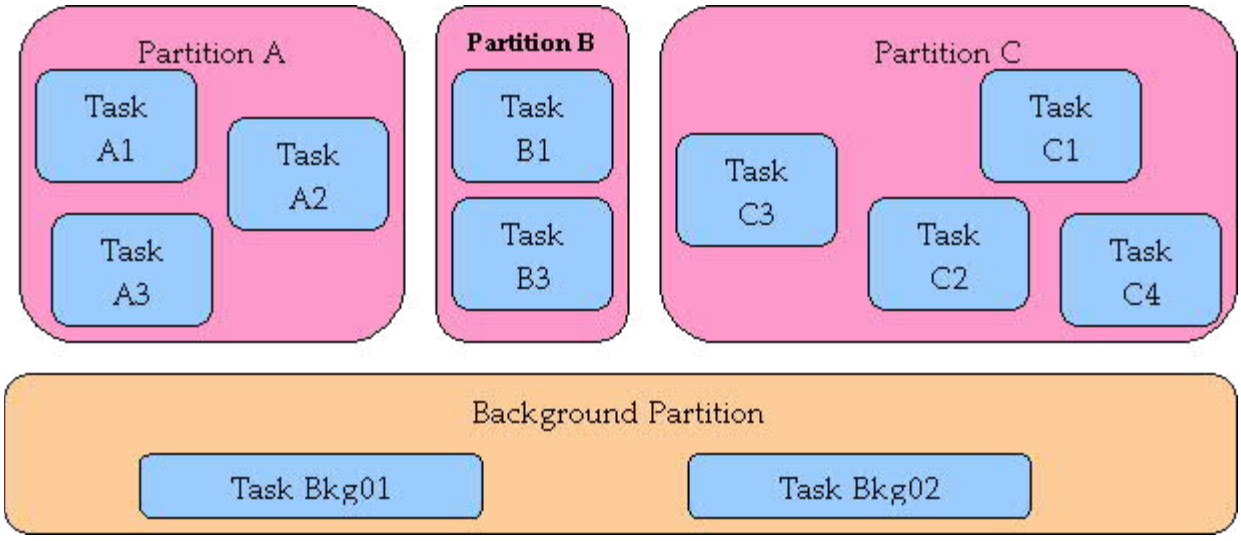

Figure 3: Partition Scheduler Partitions Example

In addition to the partitions that contain collections of CPU-critical tasks, a "background partition" can be added as well, as shown near the bottom of Figure 3. When there are no tasks ready to run within the currently active partition, the partition scheduler can run background tasks in the background partition instead. Tasks in the background partition may or may not run from time to time. They certainly cannot have hard real-time deadlines. And, unlike the other critical partitions, the "background partition" might actually still suffer from CPU-access starvation.

So now we've added Partition Schedulers to our list of interesting schedulers for future RTOSs. Which will be the best scheduler for our ideal future RTOS? A sensible answer to this question would be ***"Well, there probably isn't one single best scheduler for all embedded systems. Wouldn't it be better for the ideal future RTOS to offer us a variety of available task schedulers, from which we could choose the best scheduler for each of our projects?"*** And so, the prediction is that future RTOSs will allow embedded developers to choose from a modularly interchangeable repertoire of task scheduler options.

The modularly replaceable scheduler repertoire could range from today's priority-based pre-emptive scheduling, through various deadline schedulers and partition schedulers, and beyond. Perhaps we can even imagine combination schedulers using, for example, partition scheduling at the task group level, and MUF deadline scheduling for tasks within the partitions. Scheduler selection or replacement would be via mouse-click in an RTOS configuration tool.

THE FUTURE OF INTER-TASK COMMUNICATION: ASYNCHRONOUS MESSAGE PASSING

Referring back to Figure 1, the second main section of an RTOS kernel is "Inter-task Communication and Synchronization" – the crown above the Task Scheduler. These mechanisms are necessary in pre-emptive multitasking, because without them the tasks could communicate corrupted information or otherwise interfere with one another. Today's RTOSs offer a varied menagerie of inter-task communication and synchronization mechanisms, including message queues, mailboxes, pipes, semaphores, mutexes, event flags, signals, condition variables, and many more.

A clear trend as RTOSs advance into the future, is the increased focus on message passing for intertask communication, and a gradual de-emphasis of the other mechanisms we've had to deal with in the past. Message passing is simple and intuitive, as well as being a conceptual "gateway" to multi-core and distributed multi-processor embedded systems. It is the preferred technique for communication between tasks on different processors or on

different embedded boards, and it also works well in traditional single-processor environments.

Asynchronous message passing is a loosely-coupled approach to data transfer from task to task, where a task sending a message does not wait for any information from the receiver task -- and thus can not fail even if the receiver task has failed or becomes inaccessible. It is a conceptual "gateway" into the world of fault-tolerant embedded system design.

The asynchronous message passing approach avoids synchronization issues inherent in mechanisms such as semaphores, which can create long delays in a multiprocessor environment as tasks wait for distant semaphores to be locked or unlocked. It also avoids more traditional semaphore pitfalls such as unbounded priority inversions, ease of deadlocking, and difficulty in debugging. Mutexes, which are often viewed as a solution to the problems inherent in semaphores, work by changing priorities of tasks – and so they too are inappropriate in multi-core or multi-processor environments. They just don't work when mutual exclusion is needed between tasks on different processors. Many embedded software designers are wary of mutexes even in single-processor environments because of the "behind-the-scenes" task priority changes that RTOSs use to implement the mutexes. Hence, asynchronous message passing can be expected to dominate inter-task communication the future.

The simplest kind of asynchronous message passing model is called **asynchronous "direct" message passing**, in which tasks send messages "directly" to one another (via the RTOS), without the need for application software to be aware of queues, mailboxes, or other intervening RTOS mechanisms. This is shown in Figure 4. It is conceptually simple and intuitive, and less prone to design errors and bugs than other mechanisms.

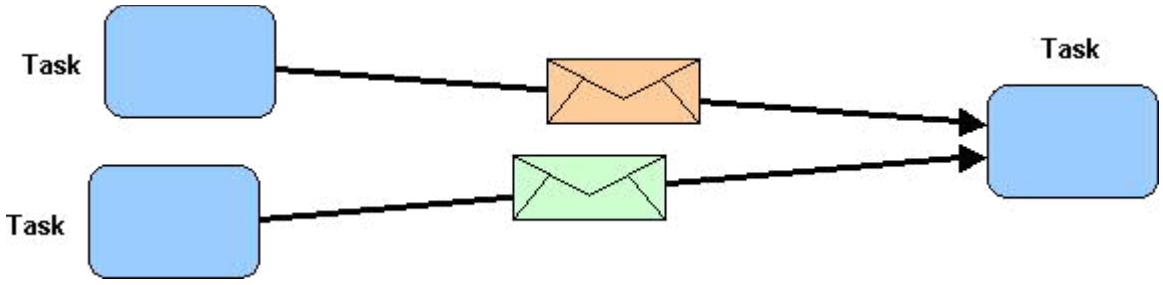

Figure 4: Asynchronous Direct Message Passing for Inter-task Communication

TIMER SERVICES

The third main section of an RTOS kernel shown in Figure 1 is Timers. Most RTOSs today offer both "relative timers" that work in units of **ticks**, and "absolute timers" that work with **calendar date and time**. For each kind of timer, today's RTOSs provide a "**task delay**" service, and often also a "**task alert**" service based on a signaling mechanism such as event flags.

Future additional timer services will focus on the fundamental requirements of real-time software: meeting time **deadlines**. They will cooperate intimately with task schedulers (deadline schedulers or other sorts of schedulers) to help determine whether tasks (or chains of tasks) have met or missed their real-time deadlines.

Other future timer services will address task "**liveness**" issues, to monitor whether tasks are continuing to execute regularly as expected --- or whether they're suffering from violations of liveness such deadlock, lockout or starvation.

DYNAMIC MEMORY ALLOCATION

In the fourth main section of RTOS kernel services, most RTOSs today offer tasks the ability to "borrow" chunks of RAM memory for temporary use. Some RTOSs provide memory allocation from "heaps", that tend to suffer from memory fragmentation. Other RTOSs also offer alternative memory allocators, sometimes called "pools", that are non-fragmenting.

But it is still possible to run out of memory ("memory starvation") with either of these approaches. One solution is to organize tasks into groups, as in Figure 3. Each group can be assigned its own memory "heap" or "pool", so that no matter what happens within a group's RAM memory allocation area, other groups will not be endangered by memory starvation.

An additional dangerous situation can occur when tasks are created dynamically "on the fly", for instance when a task creates a number of new tasks, that can themselves go ahead and create even more new tasks. This "population explosion" of tasks could quickly consume all of the memory in a heap or pool. But a future RTOS could protect against this by doing memory budget management within each pool. For example, if a task created 2 more tasks, the 3 tasks together would have the same total memory allocation "budget" as the initial creator task had. Perhaps each of the 2 new tasks would receive a memory budget of 1/3 of the initial task's budget, and at the same time the initial task's budget would be reduced to 1/3 of its original value. In this way, other tasks in the same group and in other groups, would not have less memory available to them after new tasks were created. Of course, the future RTOS would have responsibility for managing this RAM memory budgeting.

DEVICE I/O INTERFACE

In this final section of RTOS kernel services, the Device I/O Interface, today's kernels often provide both a standard application programmer's interface ("API") between tasks and device driver software, and a supervision facility for organizing and managing large numbers of diverse device drivers.

However today's device driver APIs and supervisors are "standard" only within a specific RTOS. Different RTOSs provide different, incompatible "standards". There is no industry-wide "**Gold Standard**" for device driver APIs and supervisor features across most RTOSs. And unfortunately, it is expected that this sad state of affairs will continue for the foreseeable future. This is part of a wider RTOS API standardization issue, for which there is also glacially slow progress toward an industry-wide "**Gold Standard**".

CONCLUSION

In the near future, we will witness much wider availability of a plethora of task scheduling options, including traditional priority-based preemptive schedulers, deadline schedulers of several varieties, and partition schedulers. Some future RTOSs may offer modularly interchangeable repertoires of these sorts of task scheduler options.

In the realm of intertask communication and synchronization, there will be a trend toward greater focus on a single main mechanism, rather than a growing cornucopia of intertask communication and synchronization mechanisms. Semaphores and mutexes will shrink in popularity, while asynchronous direct message passing will grow in popularity. This trend will be driven by the growing complexity of embedded software, and the growing use of distributed and multi-core embedded system designs.

Future RTOSs will have enhanced timer services to help determine whether tasks (or chains of tasks) have met or missed their real-time deadlines, as well as to monitor task "liveness". Their dynamic memory allocators will have the ability to systematically avoid memory starvation of tasks.

Perhaps we will meet at a professional conference in year 2020, to see how many of these predictions will have come to pass by then ?

This paper has been a brief introduction to some advanced RTOS kernel features that we may encounter in the future. They answer some software design needs that are already apparent today.

END.

A condensed version of this material appears in the November 2004 issue of Elektronikpraxis magazine (Germany).