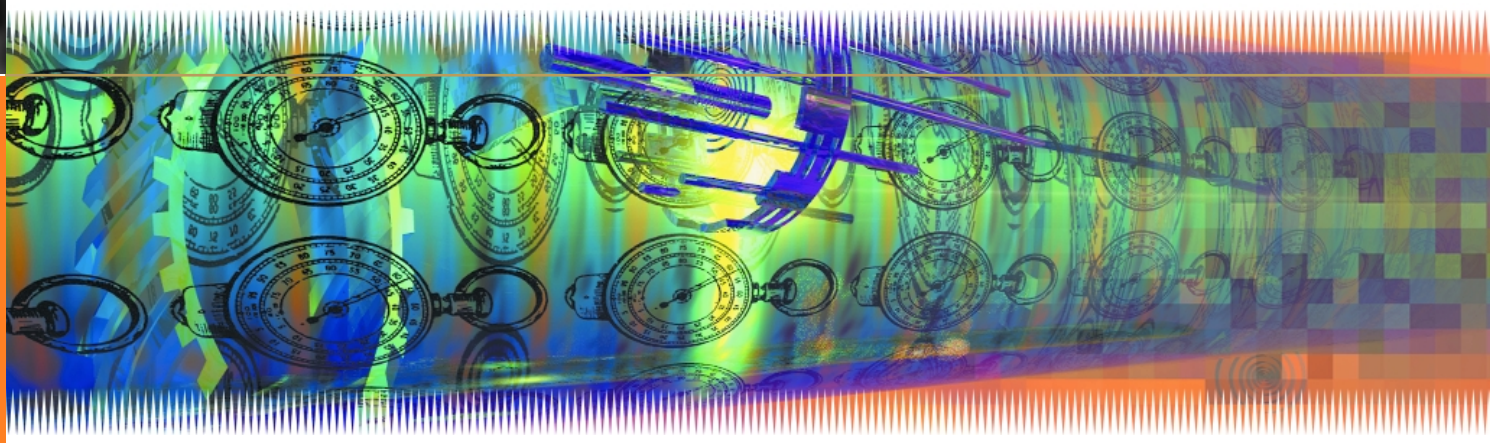
DAVID KALINSKY



Context Switch

From the humble infinite loop to the priority-based preemptive RTOS and beyond, scheduling options are everywhere to be found. This article offers a survey and comparison.

Many different kinds of task schedulers are available to software developers of embedded and real-time systems. They range from a simple cyclic executive that you can build "at home," to the many priority-based preemptive schedulers that are available commercially and beyond.

Table 1 shows a number of task schedulers, including the sorts of software tasks and hardware device interfaces they support. Depending on the nature of your application and your I/O requirements, you can choose the appropriate one from a wide spectrum of schedulers that will be described here.

The endless loop

For very simple embedded systems, the most basic way to write application software is as an endless loop. The activities programmed within the loop are executed in sequence. Branches and nested loops are okay, as long as when the code is done executing, it loops back to the beginning for another go-round.

For example (in pseudocode) :

```
DO FOREVER
   Request Input Device make a Measurement
   Wait for the Measurement to be ready
   Fetch the Value of the Measurement
   Process the Value of the Measurement
   IF Value is Reasonable
      THEN Prepare new Result using Value
      ELSE Result will be an Error Report
   Request Output Device deliver the Result
   Wait for the Result to be output
   Confirm that output is OK
END DO
```

This style of programming works well in some simple embedded systems, especially if the software can complete the sequence of code and loop around quickly enough. But in other embedded systems, this style of programming will result in performance that is too slow. Keep in mind that interrupts from hardware devices can't be handled in this

For some applications, the view of "real-time" taken by a basic cyclic executive is not precise enough. In more sophisticated applications, precision of timing is often more important than raw speed.

style of programming. Devices interact with software in the loop only when polled.

Basic cyclic executive

In more complex embedded systems, the idea of the endless loop can be extended. These systems have hundreds or thousands of lines of code, so software designers like to organize the code into separate units referred to as *tasks*. These tasks (sometimes also called *processes*) should be as independent as possible so that they deal with separate application issues and interact very little with one another.

In a software design using a basic cyclic executive, these tasks execute in standard sequence within an infinitely repeating loop, as shown in Figure 1. This is much like the endless loop design, but now dealing with large tasks.

These tasks can pass information to one another easily, by writing and reading shared data. That's because every task always runs to completion before another task begins running. So there's no danger of a task getting incomplete data from another task.

Here too, interrupts from hardware devices can't be handled. Devices must be polled, if they are to interact with tasks in the loop.

In some sense, this can be thought of as "real-time" task scheduling, if all of the software in the loop executes quickly and the loop can execute repeatedly at a very rapid rate.

Time-driven cyclic executive

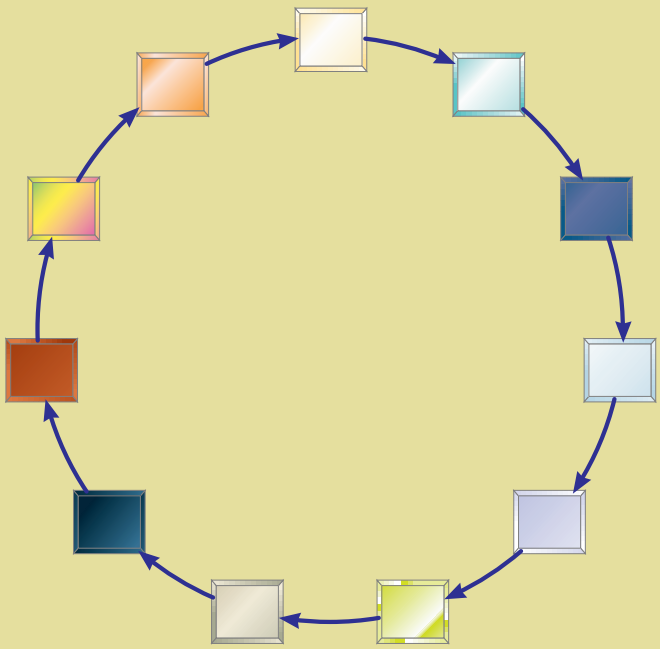
For some applications, the view of "real-time" taken by a basic cyclic executive is not precise enough. A basic cyclic executive tries to run its tasks as quickly and as often as possible. In more sophisticated applications, precision of timing is often more important than raw speed.

A time-driven cyclic executive can begin to address this requirement. In this scheme, one hardware timer interrupt is used to trigger the execution of all tasks. The tasks execute one after another, each one running to completion before the next one begins. For a time-driven cyclic executive to work correctly, the final task in the chain of tasks must complete its execution before the next timer interrupt arrives, as shown in Figure 2. The rate of hardware timer interrupts is the rate at which the tasks must execute.

TABLE 1 Some categories of task schedulers

Task scheduler type	Task execution	Device I/O
Endless loop	No tasks	Polled only
Basic cyclic executive	As often as possible	Polled only
Time-driven cyclic executive	Single frequency	Polled only
Multi-rate cyclic executive	Multiple frequencies, at higher precision	Polled only
Multi-rate executive with interrupts	Multiple frequencies, at higher precision	Polled and interrupt-driven
Priority-based preemptive scheduler	Periodic and non-periodic tasks	Polled and interrupt-driven
Deadline scheduler	Periodic and non-periodic tasks	Polled and interrupt-driven

FIGURE 1 Basic cyclic executive



SKELETON STOODIOS

FIGURE 2 Time-driven cyclic executive

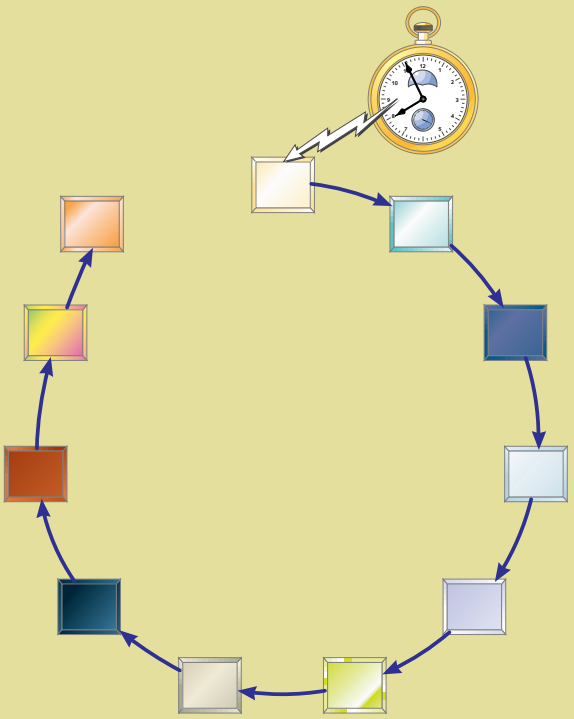
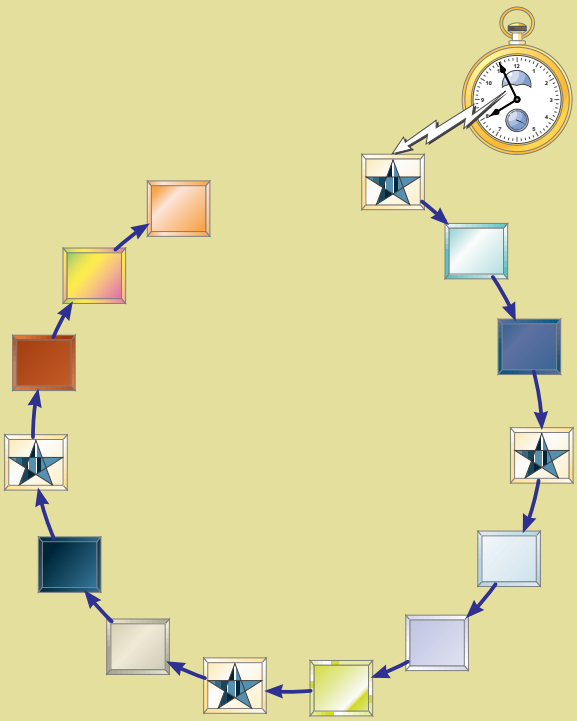


FIGURE 3 Multi-rate time-driven cyclic executive (two rates)



Although a hardware timer interrupt is involved here, tasks can still pass information to one another easily, by writing and reading shared data. Every task runs to completion before another task begins running. Interrupts from hardware devices (other than the timer) cannot be handled in this style of programming. Devices must be polled, if they are to interact with tasks.

Multi-rate cyclic executive

The time-driven cyclic executive assumes that all tasks need to run at the same rate of repetition. But in some applications, different tasks may need to run at different rates.

A modified time-driven cyclic executive, called the multi-rate cyclic executive, can handle this need reasonably well in cases where a higher rate is an integer multiple of the "base" rate.

The idea is simple. In a multi-rate cyclic executive, the base-rate tasks run once per timer interrupt, and a higher rate task runs a number of times per timer interrupt. That number is the integer multiple of the base rate. The repeated executions of the higher rate task should be as equally spaced as possible within the sequence of tasks following a timer interrupt.

Often the base-rate period is called the "major cycle," and higher rates identify so-called "minor cycles."

The example illustrated here shows a system of 10 tasks which execute at the base rate (for example, 10Hz, if the timer delivers 10 interrupts per second). In addition, an eleventh task, marked by a star, executes at 40Hz, four times the base rate. This is done by having the starred task appear four times in the chain of task execution which follows each timer interrupt, as shown in Figure 3.

Limitations of cyclic executives

Cyclic executives have been shown to solve a number of problems, while remaining fairly simple to implement. With the help of a hardware timer

FIGURE 4 A mid-task switch (manual preemption)

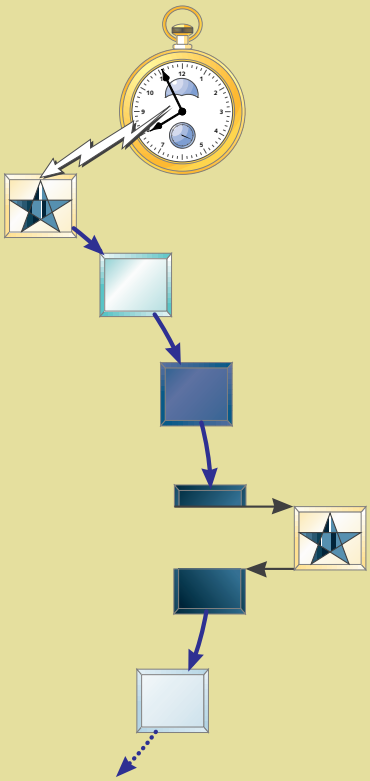
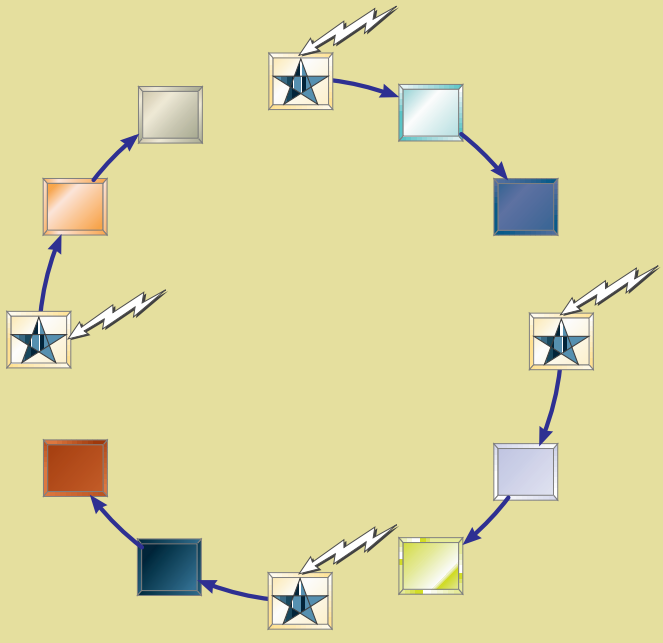


FIGURE 5 Multi-rate executive for periodic tasks



interrupt, they can run tasks at a regular rate. They can even run different tasks at different rates. Tasks can communicate with one another through shared data, without special concern about data integrity. Hardware devices (other than the timer) are polled, rather than interrupt driven.

The limitation that hardware devices must be polled when using a cyclic executive is often a serious one. If the device is not polled frequently enough, important transient occurrences might be missed. If the device is polled too frequently, much of the processor's power might be wasted. For these reasons, interrupt-driven peripheral devices are usually preferable for I/O.

Another objection to cyclic executives is that the timing of task execution can't be controlled precisely. Even when hardware timing is used to trigger the execution of a chain of tasks, only the first task in the chain has its start time determined precisely by hardware. The second task in the chain starts to run whenever the first ends, and so on. If these tasks contain code of varying processor loading such as data-dependent loops, all later tasks in the chain will run at times influenced by load on previous tasks.

Even if all tasks do not contain code of varying processor loading, timing of individual tasks is only approximate. This can be seen in the illustrated example of the multi-rate cyclic executive. In Figure 3, the starred task is required to execute at a rate of 40Hz. In other words, there should be precisely 25ms (or 25,000µs) between successive execution starts for the starred task. If the diagram is viewed as a circle where a complete circumference represents one 10Hz base period, then the starred task should execute at angles of precisely 0 degrees, 90 degrees, 180 degrees, and 270 degrees. But it does not. Sometimes it executes somewhat early, sometimes, a tad late. It all depends on when the pre-

vious task finished, and how long the following task will take. Remember, each task must run to completion and cannot be "interfered with" in mid-execution.

Some software designers have tried to solve these timing problems by actually counting machine cycles of the computer instructions to be executed by each task, in order to figure out precisely how long each task would take. Then the designer would determine exactly how much of a certain task could execute before a precisely timed task needed to run. This part of the task would be allowed to run, and then the precisely timed task would be inserted for execution, and the remainder of the delayed task would run later. Effectively, the task would be cut in two. See Figure 4.

This "solution" gives rise to several new problems:

- If the tasks involved in a mid-task switch share some data structures, those data could end up in an inconsistent state because of the mid-task switch. This could result in a numeric error in the outputs of either of the tasks involved.
- Every time software maintenance causes some code to be changed or added in the tasks which run before the mid-task switch, machine cycles need to be re-counted and task timings recalculated. A task might need to be cut apart differently for the mid-task switch in this new code situation.

In other words, this "solution" is an error-prone and excruciatingly tedious method of building software. Rather than a solution, this should be offered as an example of an attempt to use a cyclic executive beyond its realm of usefulness.

Cyclic executives should not be used in situations where timing requirements are so precise and critical that you would consider "surgically" cutting a task into two sections.

Multi-rate executive for periodic tasks

If all tasks are periodic, but at differing rates of execution, then a multi-rate executive can often be better than a cyclic executive. In such a scheduler, timer interrupts must be delivered at a rate that is the lowest common multiple of all the rates of the tasks. And at each timer interrupt (or "tick"), tasks can be made to execute.

For example, if tasks need to execute at 50Hz, 60Hz, and 100Hz, then timer interrupts must be delivered at a rate of 300Hz. The 100Hz task will be made to

execute on every third tick. The 60Hz task will be made to execute on every fifth tick. And the 50Hz task will be made to execute on every sixth tick.

If tasks do not need to be time-synchronized with each other, they could be executed at ticks that are offset from one another. For example, the three tasks above need not all be run at tick 0. Perhaps the 100Hz task would be run for the first time at tick 0, the 60Hz task at tick 1, and the 50Hz task at tick 2.

A simpler example is shown in Figure 5. Here we have only two rates, with the higher rate or "minor cycle" being four times the lower rate or "major cycle."

Every task must run to completion before another task begins running.

FIGURE 6 Interrupts can interfere with a task's data processing

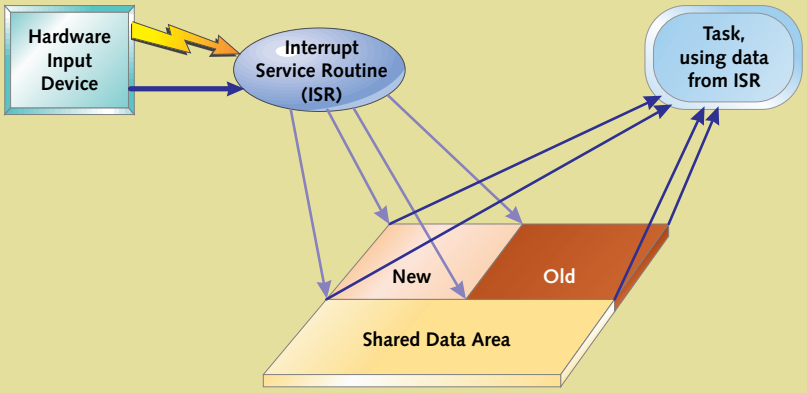
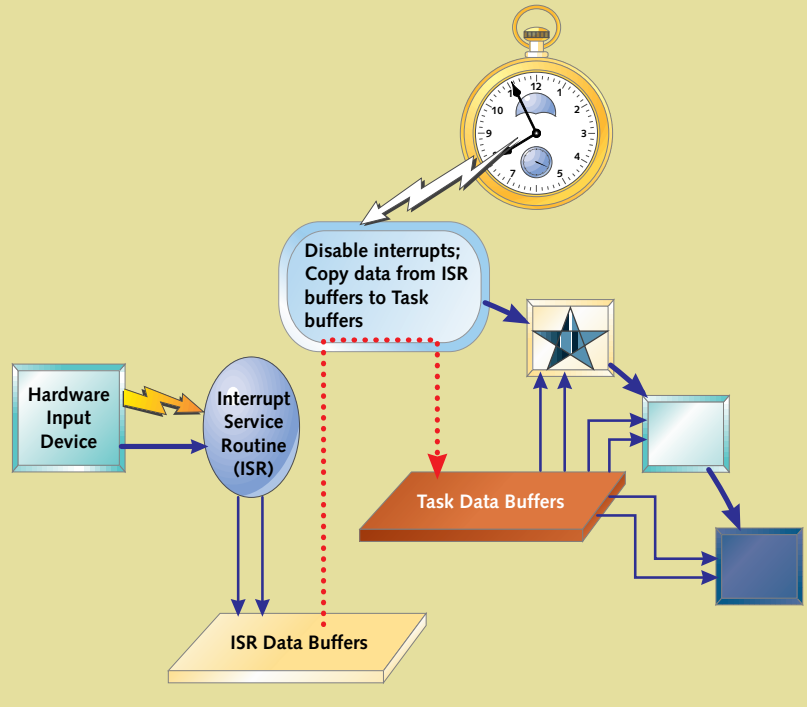


FIGURE 7 Multi-rate executive with interrupts



As with cyclic executives, tasks can pass information to one another easily, by writing and reading shared data. All hardware devices (other than the timer) must be polled.

Caution: adding interrupts

The restriction to polled hardware devices in all the previous types of schedulers is a serious one. Modern hardware I/O devices are typically interrupt driven. But interrupt-driven devices can cause problems of their own, if they are not handled properly in software.

Very often, if an interrupt service routine (ISR) tries to pass data to the very task it is interrupting, the task may not handle the passed data properly. For example, a task may begin processing some data and then an interrupt service routine might update the data, followed by the task reading the data again for purposes of further processing. The net result would be that part of the data processing in the task is done on an old value, and another part is done on a new value for the same data, resulting in possible inconsistent outputs from the task.

Another example is when a task and an interrupt service routine communicate through a shared data table. If an interrupt occurs and is serviced while the task is in the midst of reading from the table, the task might well read "old" data from part of the table and "new" data from other parts of the table. This combination of old and new data might lead to erroneous results. See Figure 6.

The integration of interrupt-driven software with task schedulers requires special care, particularly in terms of information exchange between ISRs and tasks.

Multi-rate executive with interrupts

One clever idea for avoiding the pitfalls we have seen when ISRs and tasks interact is to have the ISRs write their input data into one set of buffers, and

With a preemptive scheduler, switching between tasks can take place at any point within the execution of a task (even when the task isn't yet done executing its code).

FIGURE 8 Preemptive scheduling

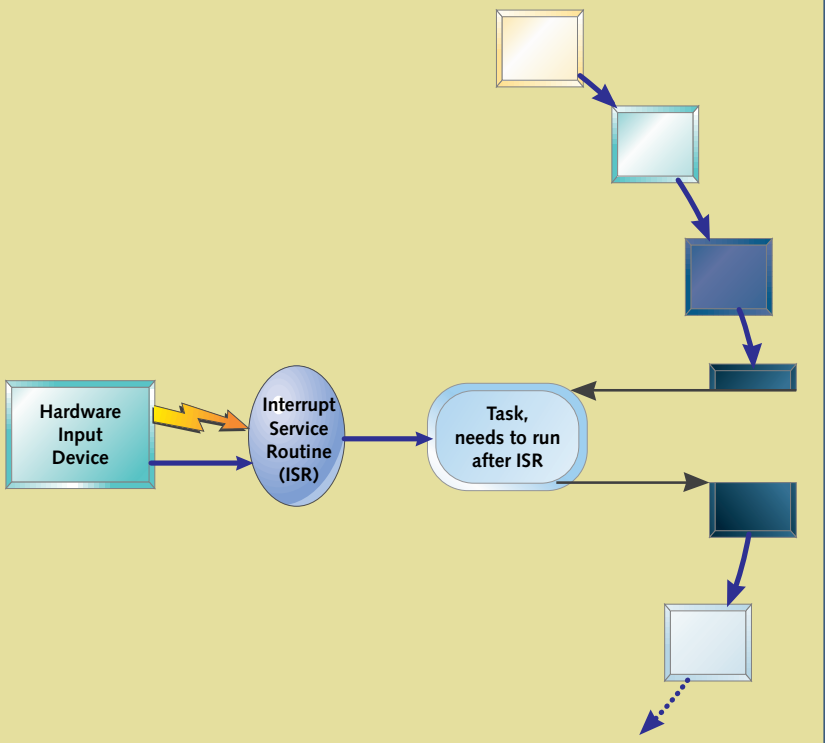
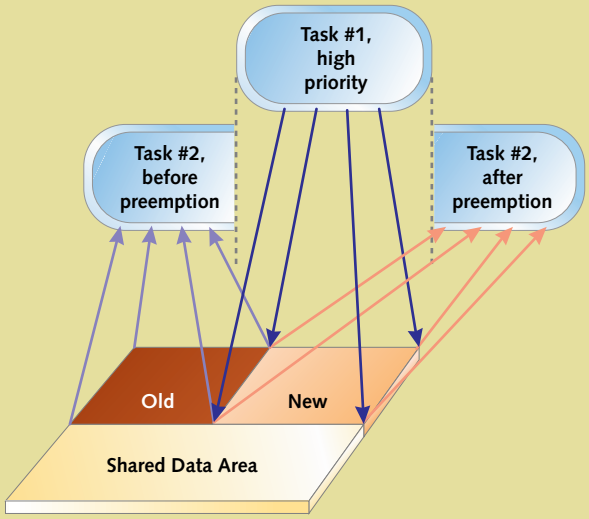


FIGURE 9 Problems in sharing data between preemptive tasks



the tasks use data from a completely separate set of buffers. At every clock tick (of the multi-rate executive for periodic tasks, for example), interrupts are turned off and input data are copied from the ISR buffers to the task buffers. Then interrupts are turned back on, and the tasks scheduled for that tick are permitted to execute. This is shown in Figure 7.

In this way, data can be transferred from ISRs to tasks without danger of inconsistent data (since interrupts are disabled during the data transfer). But interrupts are re-enabled and active while the actual application tasks are running. This technique works when all scheduled tasks finish running before the next clock tick.

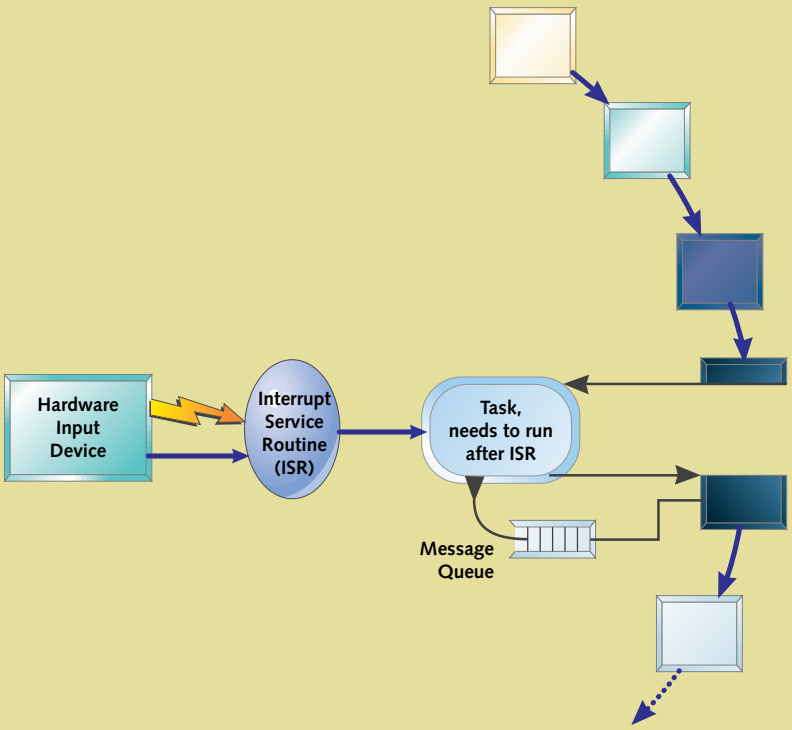
This kind of scheduler becomes quite complex, and should not be written as a weekend "garage" project. With this scheduler, every task runs to completion before another task begins running. As with previous kinds of executives, tasks can pass information to one another easily by writing and reading shared data.

Hardware devices are no longer restricted to polled only. They can be interrupt-driven. However, information delivered by an interrupt and acquired into software by an ISR is not immediately passed onward to a task for further processing. ISR data are transferred to task buffers only after the next timer interrupt. In some applications, this could be an unacceptable delay or complication.

Getting faster response: preemptive scheduling

The schedulers which have been surveyed so far are called *non-preemptive*, because switching between tasks only takes place when one task has fully completed its execution and another task wants to start its execution (from its beginning). Faster response can often be obtained by going over to a "preemptive" scheduler. With a preemptive scheduler, switching between tasks can take place at any point within the execution of a task (even when

FIGURE 10 Preemptive tasks can communicate via an RTOS's message queue



the task isn't yet done executing its code).

For example, when an interrupt occurs, its ISR might want to say something like "I don't care which task was executing before my interrupt, and I don't want to wait for the next timer tick. I would like Task 67 to begin executing right now!" A preemptive scheduler can do this, as shown in Figure 8.

However, a preemptive scheduler is orders of magnitude more complex than any non-preemptive scheduler. And such a scheduler gobbles up lots of RAM stacks, for storage of task "contexts" and other task status information. Such a scheduler could not be written in a month of Sundays as a "garage" project. Some commercially available RTOSes often have preemptive schedulers as their underlying "engines."

With a preemptive scheduler, hardware devices can be either polled or interrupt-driven. Information delivered by an interrupt and acquired into software by an ISR can be immediately passed onward to a task for further processing.

Caution: preemptive schedulers bring up new issues

Preemptive schedulers offer the software developer many benefits, beyond what can be achieved with simpler "home-made" schedulers. But their sophistication brings up new issues, of which a software developer must be aware.

One of these is the matter of which tasks may be preempted and which may not? The answer is to assign a priority number to each task. Tasks of higher priority can preempt tasks of lower priority. But tasks with lower priority cannot preempt tasks of higher priority. A preemptive scheduler needs to be told the priority of each task that it can schedule.

A second issue that a software developer must consider is: tasks that can be preempted, and which can preempt others, cannot be allowed to pass information to one another by writing and reading shared data. The simple methods for passing information between tasks that worked with non-preemptive schedulers no longer work with preemptive schedulers.

The problem in passing information between tasks by writing and reading shared data can be described as follows (see Figure 9): if one task preempts another while the second task is in the midst of reading from a shared data table, the second task might read "old" data from part of the table and "new" data from another part of the table after the first (preempting) task writes new data into the table. This combination of old and new data might lead to erroneous results.

This is, in fact, the same problem that occurs if an ISR and a task try to communicate by writing and reading shared data, as discussed earlier.

In order to help the software developer prevent these problems, an operating system that has a preemptive scheduler should provide mechanisms for passing information between tasks (and also to/from ISRs). The mechanisms provided vary in different operating systems. For example, most RTOSes provide a message queue mechanism and semaphores. RTOSes meeting the OSEK standard that is gaining popularity in automotive applications provide mechanisms called resources, events, alarms, and messages. In some RTOSes, the message queues are "global" entities (all tasks can perform all operations on them); in others, the message queues are "owned" entities (only the "owner" task can get the messages). Figure 10 shows an interrupt-triggered task delivering a message to a task it has preempted, using a message queue.

One of these mechanisms must be used every time information is to be passed between tasks, in order to ensure reliable delivery in a preemptible environment.

Deadline scheduling

Users of off-the-shelf, priority-based preemptive schedulers sometimes have the following objection: "where do I tell the scheduler what are the deadlines for my tasks, so that the scheduler will make sure they're met?" The fact of the matter is that you can't tell these schedulers about task deadlines. They don't want that kind of information. All they want to be told is each task's priority; they do all of their their task scheduling based on the priority numbers. The mapping between deadlines and priorities is not often straightforward. Rate monotonic analysis can be used in certain situations, but it's often downright impossible to be sure that tasks will meet their deadlines if you use a priority-based preemptive scheduler with fixed task priorities.

An alternative kind of preemptive task scheduler is called a *deadline scheduler*. This kind of scheduler tries to give execution time to the task that is most quickly approaching its deadline. This is typically done by the scheduler changing priorities of tasks on-the-fly as they approach their individual deadlines. The popular, commercially available off-the-shelf RTOSes don't offer deadline scheduling. But you can see how they work and build one of your own (see Ellison's book in References).

Spectrum of schedulers

This has been just a short introduction to the world of task schedulers. Depending on the nature of your application and your I/O requirements, you can choose from a wide spectrum of schedulers. They range from a simple cyclic executive that you can build "at home," to the many full-featured, priority-based preemptive schedulers available commercially, and to even more sophisticated schedulers. **esp**

David Kalinsky is the director of customer education at Enea OSE Systems, though he held the same title at another company, Aisys, when he wrote this survey article. David has been involved in the design of many embedded medical and aerospace systems over the years. He holds a PhD in nuclear physics from Yale. His e-mail address is david@enea.com.

References

Ball, Stuart. R. *Embedded Microprocessor Systems—Real World Design*. Boston: Newnes, 1996.

Laplante, P. A. *Real-Time Systems Design and Analysis—An Engineer's Handbook*. New York: IEEE Press, 1997.

Ellison, Karen S. *Developing Real-Time Embedded Software in a Market-Driven Company*. New York: John Wiley & Associates, 1994.