BY PHILIP KOOPMAN JR.

ht Tasking

Cooperative tasking techniques can often meet an embedded system's multitasking requirements and give better performance.

FitzGraphics

To most embedded developers, multitasking means using a preemptive multitasker and a complex, expensive (in terms of software cost, memory size, and run-time overhead) piece of software. While embedded real-time systems are best written with a multitasking approach, preemptive multitasking is often overkill. A preemptive multitasker is a very generalized tool; users may not want to pay the costs that always accompany generalized solutions to their very specific problems.

A less widely understood approach to real-time design is cooperative multitasking. Judicious use of cooperative tasking techniques can often meet an embedded system's multitasking requirements, while giving better performance and a simpler software environment than a preemptive multitasker.

In my work with stack-based processors at Harris Semiconductor Inc., I've investigated the different approaches to multitasking and explored the different tradeoffs among complexity, cost, and speed. As a result, I'm convinced that the ease of implementation and efficiency of cooperative multitasking are widely underestimated.

WHY MULTITASKING?

Multitasking is used in embedded systems for a variety of purposes. Keeping separate system functions resident in separate tasks helps reduce the complexity of each portion of the system. Separate tasks also help modularize the work of writing the system by forcing the creation of well-defined interfaces between modules and programmers.

Tasks can also be used to distinguish different activity classes for a system. For example, routines that only need to be invoked occasionally can be included in a task run in response to an external stimulus, such as an interrupt or a timer pulse. Tasks that always run but are not time-critical can be assigned lower priorities than other tasks.

On larger systems with multiple processors, breaking a software system into tasks provides natural boundaries for process migration. Setting parallelism granularity at the task level allows each processor to be assigned to one or more active tasks, greatly reducing the software's complexity. In the best of all possible worlds, we have one processor per task. Unfortunately, not all systems have the luxury of multiple processors.

Heavyweig

Heavyweight Tasking

The problem comes when many tasks contend for use of a single processor. This processor must be shared among tasks, which can involve significant overhead.

PREEMPTIVE TASKING

Forth has a long tradition of providing many support levels for multitasking. While cooperative multitasking is a well-known technique in Forth circles, the technique needn't be limited to that language. Indeed, with the recent appearance of C compilers for stack processors like the RTX, the approach becomes particularly attractive for C programmers as well. However, stack processors do alter some of the design tradeoffs.

The preemptive method is usually what is meant by "multitasking" in conventional computing terminology. In a preemptive multitasker, some predefined event, such as a timer pulse, periodically halts the executing program and transfers control to an executive program or kernel. The kernel saves the complete state of the processor (all registers and status bits), including the program counter and stack contents. Once the old task state is saved, the kernel uses some scheme to select another task for execution. The newly selected task has its state loaded into the processor and execution is resumed on the new task.

The advantage to full preemptive multitasking is that it is almost trans-

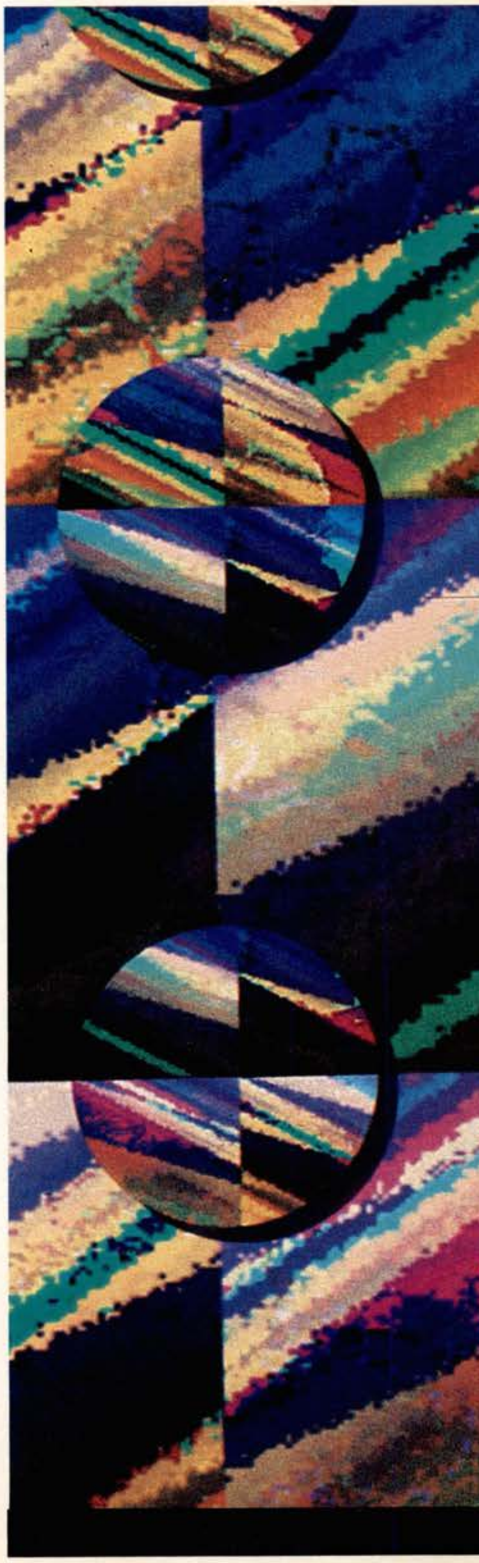


Table 1
Pause subroutine code for a lightweight multitasker written for the RTX 2000

RTX 2000 instruction	Comment
R>	Transfer return address of calling task (task restart address) to data stack.
0 U!	Store that restart address to task area word 0.
1 U@	Fetch link value from task area word 1.
UBR!	Store that link value as the new value of the user base register.
0 U@	Fetch the restart address of the new task from the new task area.
>R EXIT	Transfer the restart address from the data stack to the return stack, then perform a subroutine return (equivalent to a jump to the restart address).

Features

1. Each task uses the user base register to point to a 32-word task area used for scratch storage.
2. The pause subroutine code takes six words of memory shared among all tasks.
3. Each invocation of pause code takes 10 clock cycles, including the subroutine call instruction from the calling task.

parent to the programmer, and it is a powerful and automatic way to ensure that no problems occur during the transitions between tasks. Unfortunately, the cost for this transparency and power is very high. The machine's entire state must be saved to guarantee a correct restart when the task is resumed.

In contrast to RISC and most CISC processors, however, most stack machines automatically track the number of active elements on the stack. In practice, the stacks tend to be fairly shallow. However, whatever processor architecture you're using, the cost for a preemptive multitasker context switch is high.

An additional problem with preemptive multitasking is that certain code sequences, known as critical regions, must not be interrupted by a task switch. Typically, these sequences deal with access to resources shared among tasks or time-critical code.

To prevent a preemptive multitasker from interrupting these sequences, a flag must be used to notify the executive that the task is in a critical region. Because task switches are forbidden within these regions, task-switching latency increases dramatically. Alternatively, semaphores or other synchronization methods may be used to notify other tasks when a data structure is in use in case the task is interrupted. Either way, a considerable run-time overhead and programming effort is involved in preventing a preemption from causing problems.

HEAVYWEIGHT COOPERATION

In some systems, the number of shared data references and critical regions executed will have a much greater impact on efficiency than the number of task switches made. Or, you may be dealing with a relatively simple system where it's hard to justify the generality and complexity of a preemptive multitasker. In these cases, it may actually be desirable to do away with the preemptive tasker and implement cooperative tasking.

In cooperative tasking, the task decides when it is ready to give up control of the processor. This decision is typically made by each task. The task periodically invokes a routine that queries a timer to see whether the system desires a task switch. This technique is called "pausing."

In a simplified but oft-used case, a task switch is performed on every pause. When it is time for a task switch, the full processor state is saved in a manner similar to that used for preemptive tasking. The term "heavyweight" is used because the complete machine state must be saved to guarantee correct operation. The term "cooperative" is used because the task relinquishes control of the system by executing pause statements sprinkled throughout its code rather than relying on an external preemption mechanism.

The advantage to using the cooperative tasking method is that neither synchronization variables nor critical-region flags are necessary. Since task switching only occurs when the programmer requires it, a task cannot be shut down by the kernel at an inopportune moment. This fact can substantially reduce your run-time overhead and code complexity. Further, much of the code in the kernel used for synchronization and data-structure sharing can be eliminated, reducing memory costs.

The disadvantage of using the cooperative tasking method is that it places more of a burden on the programmer for correct operation. Programmers are responsible for ensuring that periodic checks for task switching are placed in appropriate sections of the code. If they place these checks wisely, the result will be good performance with small task-switching latencies. If programmers are not so wise, some task-switching latencies may be undesirably long.

In cooperative tasking, the task decides when it is completed and ready to give up control of the processor to other tasks.

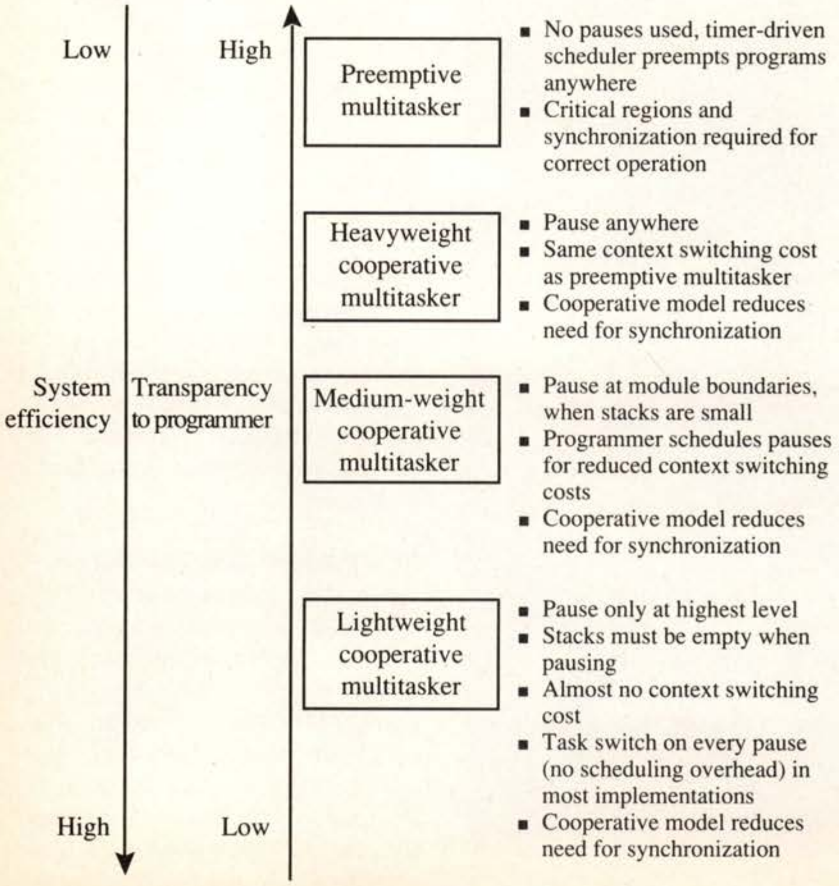
DEALING WITH LATENCY

The issue of task-switching latency can be addressed in three ways. First, we can use cooperative tasking in applications where task-switching latency is not extremely critical or the tasks are quite simple and rely on the programmer's skills. Obviously this approach has limited applicability.

The second way to reduce this problem is to move all time-critical code to interrupt-service routines. For example, a task that is activated to read data from an input port before the data is overrun can be replaced by an interrupt-service routine that places the data into a first-in-first-out buffer and a task that processes data from that buffer. This strategy has the effect of desensitizing the system to task-switching latency. Similar methods must often be used with preemptive tasking models as well because of their problems with latency caused by the critical regions.

Finally, you can treat an unacceptably long latency as a soft error. This solution can be accomplished by using a watchdog timer on either the development system or target system. When the maximum permissible task-switching latency is exceeded, a warm start of the task can be performed, along with a suitable message to the programmer if he or she is in a development mode.

Figure 1
Characteristics of tasking models.



Low
High
System efficiency
Transparency to programmer
High
Low
Preemptive multitasker
No pauses used, timer-driven scheduler preempts programs anywhere
Critical regions and synchronization required for correct operation
Heavyweight cooperative multitasker
Pause anywhere
Same context switching cost as preemptive multitasker
Cooperative model reduces need for synchronization
Medium-weight cooperative multitasker
Pause at module boundaries, when stacks are small
Programmer schedules pauses for reduced context switching costs
Cooperative model reduces need for synchronization
Lightweight cooperative multitasker
Pause only at highest level
Stacks must be empty when pausing
Almost no context switching cost
Task switch on every pause (no scheduling overhead) in most implementations
Cooperative model reduces need for synchronization

Heavyweight Tasking

HEAVYWEIGHT COSTS?

The heavyweight cooperative tasking model is the model traditionally included in Forth programming environments. It is preferred to the preemptive tasking model because of its simplicity and the direct control it gives programmers.

Heavyweight tasking is very inexpensive when executing Forth on a conventional processor, because a Forth virtual machine running on a register-based CPU typically uses only two or three registers and leaves the other resources idle. (Forth requires two stack pointers and, in some implementations, will keep the top data stack element in a register.) Machine code sequences in a Forth program may use any number of registers for efficiency, but they are not preserved across subroutine boundaries and need not be saved when pausing.

Thus, Forth kernels on conventional machines trade off a little bit of runtime speed (by not making optimal use of registers) in return for very fast context-switching times. Since the cost of context switching is quite low, most Forth systems running on conventional processors use heavyweight cooperative multitasking.

Stack-based processors usually contain some sort of stack-buffer hardware on-chip. The use of this stack buffer significantly increases the execution speed of Forth programs over that possible with conventional processors. Unfortunately, its use by Forth and other languages also increases the state that must be saved from the CPU on a context switch. The amount of state that must be saved is typically equal to a small register file on a conventional CPU.

Medium-weight cooperative tasking is an cost-reducing embellishment of heavyweight tasking.

Of course, techniques such as partitioning the hardware stack buffer into multiple areas for high priority tasks can eliminate saving stack contents to memory in a significant number of applications. Still, the general case of a very large number of low priority tasks on a single processor does require several words of data to be saved from the CPU on many context switches. Therefore, methods other than heavyweight tasking that have reduced context-switching costs can be attractive in some situations.

MEDIUM-WEIGHT TASKING

Heavyweight cooperative tasking eliminates the overhead and complexity of dealing with synchronization and critical regions by restricting the times at which switching can occur. However, it still pays a reasonably high overhead for saving and restoring context on a task switch. Therefore, medium-weight cooperative tasking is an embellishment of heavyweight tasking that reduces the cost of context switching.

Stack-based programs, especially those programs written in Forth, tend to be quite modular. This modularity often results in routines that run in less time than the desired interval between task switches. Further, the amount of information transferred on the stack between independent modules is often very small.

Since context switching on stack machines primarily consists of saving the active stack elements, the cost of cooperative multitasking can be significantly reduced by simply choosing to place pauses at module boundaries. This method reduces the average number of stack elements that need to be saved on pauses and task-switching overhead.

No compiler analysis is required, since the user chooses where the pauses go. The user cannot make a serious mistake, because the stack hardware automatically tracks the depth of the stacks and saves all the required elements. Medium-weight cooperative tasking modifies a program that uses the heavyweight tasking model so that on the average it has fewer elements on the stack to save on context switches.

A LIGHTWEIGHT EXAMPLE

What if pauses are placed only where the stacks are guaranteed (by the programmer) to be empty? In this case, the cost of a context switch can be much longer, since only the program counter and perhaps one or two other registers need be saved. This method is called lightweight cooperative tasking.

Heavyweight Tasking

An even further refinement of lightweight cooperative tasking is possible. If the size of the routines invoked between pauses is chosen appropriately, no timer is needed to decide when to change tasks. A task change can be performed on every pause. Since the cost of task changing is so low in a lightweight tasking model, the last piece of tasking overhead is removed and overall tasking costs are reduced even further. Further, the code required to implement the tasker is drastically reduced, resulting in a greatly simplified system software environment.

Table 1 shows PAUSE code for a lightweight multitasker implemented on the Harris RTX 2000 stack processor. This multitasker assumes a round-robin scheduling policy with equally weighted task priorities. It traverses a circular linked list for task scheduling and switches tasks on every PAUSE command. Each task has a 32-word control area and scratchpad space in memory accessed through the user base register.

Offset 0 from the user base register contains the restart address for the task, which was set the last time the task was paused. Word offset 1 from the user base register contains a pointer to the control area for the next task. To start a new task, the program suspending execution performs a subroutine call to the pause code, the user base register is set to the value for the next task in the linked list, and the program counter is set to the saved restart address for that next task. In this example, the total time to process a task switch is 10 clock cycles, and the process uses seven instructions, including the subroutine call to the pause code.

It should be noted that even if an application does not appear to be suitable for lightweight tasking at first, it may be simple to adapt the technique to the situation. For example, if one task takes

too long between pauses, that task might be broken into several subtasks that do fit within the time constraints.

EXPLORING THE TRADEOFFS

One method to break up a task is to implement a finite state machine and execute a pause on every state change. I once used this technique on a proprietary microcoded graphics adapter that used a bit-sliced architecture. The system requirements were to sample a host interface every 50 microseconds; sample a bit tablet, joystick, and button box every few milliseconds; draw vectors, arcs, and shaded polygons on a storage tube display; and refresh several hundred dynamic vectors without flicker—all without any timers available. I accomplished this feat in about 2,000 words of microcode, using the various lightweight tasking techniques.

Now that we've looked at the differ-

One method that can be used to break up a task is to implement a finite state machine and then execute a pause on every single state change.

Heavyweight Tasking

ent tasking models and their characteristics, it's time to consider a selection process. How do you pick the model that's right for your application? Figure 1 summarizes the characteristics of the various tasking models.

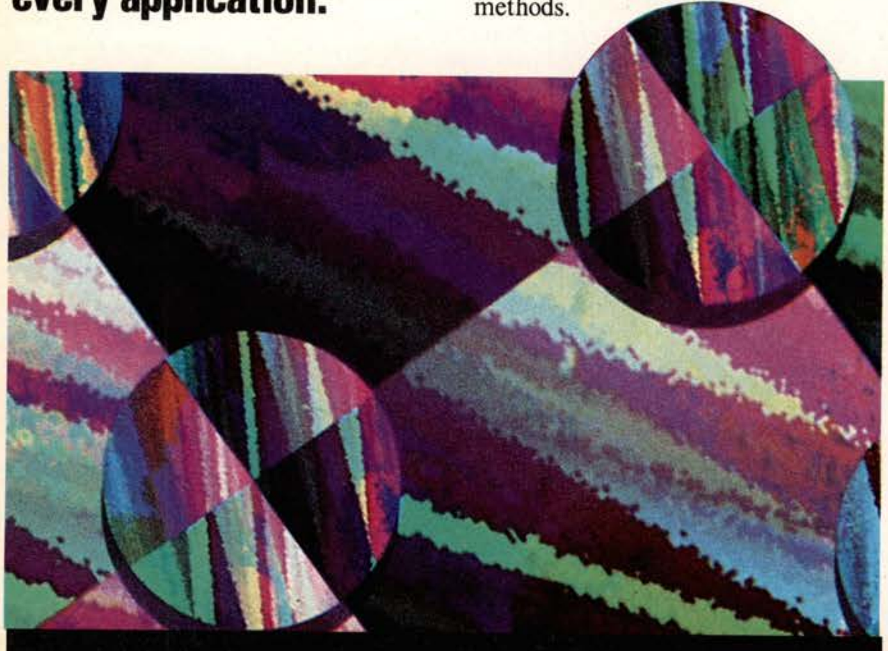
The driving force for the different types of tasking is the placement of pause statements. Preemptive tasking models do not use pauses. Heavyweight tasking models allow pauses anywhere in a program. Medium-weight and lightweight tasking models place restrictions on pause statements to simplify the tasker.

Lightweight tasking is simple, small, and fast, but may not be ideal for every application.

As tasking models progress from fully preemptive to lightweight cooperative tasking, transparency of the tasking model decreases, because programmers must spend more effort explicitly managing tasks. On the other hand, as transparency decreases, system efficiency increases because of reduced overhead and complexity for task synchronization and context switching.

The choice between preemptive and cooperative models should be made based on the run-time overhead and programming effort required to support synchronization for the preemptive model versus the effort required to insert appropriate pauses into the code for the cooperative model. For those applications where the costs of a preemptive multitasker or heavyweight cooperative multitasker can be supported, they should be used, since they reduce overall programmer effort.

In general, the more frequent synchronizations and critical regions become and the more tightly constrained target system's resouces become, the more attractive cooperative tasking becomes. For applications where task switching must be extremely fast, medium- and lightweight multitaskers are appropriate. While they require investment in terms of program organization and placement of pause statements, they can reward the programmer with superior performance over other methods.



SUMMING IT ALL UP

This discussion has been restricted to the costs of task synchronization and context switching. The method used to select the next task to be executed also involves a variety of tradeoffs between speed and providing such features as priority-based scheduling. Most scheduling methods can be combined with any tasking method, although in practice, the simpler scheduling methods are more often associated with the simpler tasking methods.

The fact that lightweight tasking is simple, small, and fast does not make it ideal for every application. As we move down the hierarchy from preemptive multitasking to lightweight cooperative tasking, we find a tradeoff at every step between the burden placed on the programmer for correct system operation and a resulting increase in system speed and simplicity. The weight of this burden depends on system requirements, type of program, and language used for implementation. Many Forth programmers using stack-based architectures find that a medium- or lightweight tasking model meet most of their needs. Obviously, you should choose the model that best meets yours.

These methods for stack machines and Forth programs on conventional machines can be adapted for use by CISC and RISC machines executing programs in any language. However, in some cases the adaptation may prove awkward, because it is much more difficult to analyze whether registers have active values than whether a stack is empty. Compilers could be adapted to provide this information, but such techniques are generally not available to application developers.

Philip Koopman Jr. is a senior scientist at Harris Semiconductor Inc. and the author of Stack Computers: The New Wave *(Chichester, W. Sussex, U.K.: Ellis Horwood Ltd., 1989) He received his Masters in computer and systems engineering from Rensselaer Polytechnic Institute, Troy, N.Y., and a Ph.D. in computer engineering from Carnegie Mellon University, Pittsburgh, Pa. He may be reached via usenet at koopman @greyhound.ece.cmu.edu.*

VOL. 3 NO. 4 > APRIL 1990

EMBEDDED SYSTEMS PROGRAMMING

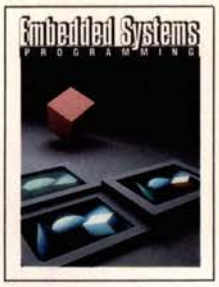
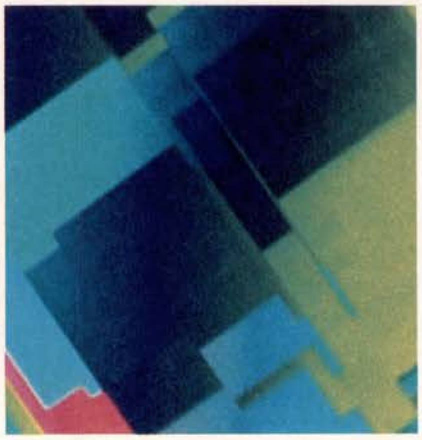

On the cover: We hold this truth to be self-evident, that not all tasks are created equally. Photograph by David Bishop.

Table of Contents


COOPERATIVE MULTITASKING **PAGE 54**

HEAVYWEIGHT TASKING **PAGE 42**

REENTRANT FLOATING **PAGE 62**