Multitasking Mysteries Revealed

Multitasking lets you allocate processing time among several tasks. This article provides the inside scoop on multitasking and gives you pointers on when it's appropriate for your embedded application and when it's not.

Revolutions are commonplace in life, and the embedded software world is no exception. It wasn't long ago when most embedded software was written in assembly language. Eventually, the migration to higher-level languages took hold; in fact, a vast majority of today's embedded applications are written in the C programming language. Quite an impressive revolution in only the last decade!

Along this same theme, use of multitasking is dramatically rising. This new revolution is due in part to necessity and partly because it is now in vogue. Although there are compelling reasons to use a multitasking operating environment, such environments are not always necessary and may even be counterproductive in certain situations. In addition, several multitasking pitfalls may have a severe impact on today's embedded software.

This article takes the mystery out of multitasking by examining when multitasking isn't necessary, the benefits of multitasking, and various pitfalls to watch out for.
Isolating the processor allocation logic makes it easier to predict and adjust the system's run-time behavior.

WHAT IS MULTITASKING?

Multitasking is simply a technique to allocate processing time among various duties the software must perform. This typically involves a division of the software into pieces, commonly called tasks, and creating a run-time environment that provides each task with its own virtual processor. (Basically, a virtual processor consists of a virtual set of processor resources.) In addition to the normal register set, virtual processor resources also include a program counter, a stack memory area, and a stack pointer. Only the executing task uses the physical processor resources. Figure I shows the virtual processor resources for multitasking on a Motorola 68000 microprocessor.

The multitasking run-time environment controls task execution. When a higher-priority task needs to execute, the currently running task's registers are saved in memory and the new high-priority task's registers are recovered from memory. The process of swapping the execution of tasks is commonly called context switching.

This transfer of processor control to another task is invisible to the application software. Of course, this invisibility might be the most fundamental benefit of multitasking. Instead of embedding the processor allocation logic inside the application software, it's done externally by the run-time environment. This arrangement isolates the processor allocation logic, making it easier to predict and adjust the run-time behavior of the system.

Multitasking is becoming mundane in the PC and workstation world. Most of us don't think about the underlying multitasking environment that allows us to use our word processor and print a spreadsheet at the same time (except when it crashes!). Just the opposite is true in the embedded multitasking arena. Due to challenging hardware and software constraints, the use of multitasking in embedded products requires considerable attention.

WHEN IS IT OVERKILL?

Some embedded applications probably won't benefit from multitasking. Typically, embedded applications less than 32K aren't candidates for serious multitasking. In addition, embedded applications that have only one software purpose might not benefit from multitasking except that its use could ease future product enhancements.

Deciding whether or not to use multitasking should be viewed exclusively from a product development perspective. If it makes sense, go ahead and use it. Otherwise, when no clear reason to use multitasking exists, don't use it, regardless of how "neat" it might seem. For example, many of us automatically lean toward an interrupt driven approach to I/O. However, many real-world situations exist in which a more primitive, polled approach may not only be better but may also be the only workable solution. In other words, as in all product design decisions, be objective and choose the design that makes the most sense.

BENEFITS OF MULTITASKING

The proper use of multitasking provides a variety of benefits—but before we discuss them, we should define some characteristics of the "kinds" of multitasking. For the remainder of this article, our multitasking environment will assume a preemptive, priority-based tasking model.

A task's importance or priority is determined by the developer. Most comprehensive multitasking environments allow multiple tasks to have the same priority. Changing a task's priority during run-time is also widely supported. Scheduling tasks for execution is usually simple and straightforward. If multiple tasks are ready at the same time, the highest priority task is executed first. Ready tasks that have the same priority are executed in the order they became ready. This round-robin scheduling is an important technique in dividing processing time between equally important tasks.
Preemption is the act of temporarily suspending an executing task in favor of a higher-priority task. This occurs in two flavors, namely solicited and unsolicited. Solicited preemption occurs when a task makes a run-time service request that causes a higher-priority task to become ready and temporarily preempt its execution. Unsolicited preemption is asynchronous and invisible to the executing task; it occurs when an interrupt handling routine makes a higher-priority task ready. In either case, control returns to the preempted task after the higher-priority task completes its processing.

After briefly laying the groundwork, we are now ready to get back to the discussion of multitasking benefits.

**Responsiveness**

Before multitasking became popular, most embedded applications allocated processing time with a simple control loop, usually from within the C main function. This approach is still used in very small or simple applications; however, in large and/or complex applications, the response time to external events is directly affected by the entire control loop processing. In general, the worst-case response time to an external event is on the order of one pass through all of the processing in the control loop. Listing 1 shows a typical control loop method for an application performing high-priority serial input. Notice that the worst-case response time occurs when the input happens immediately after the check in the control loop. In this event, the input is left undetected until all of the other items in the control loop have been processed. The worst case response time is equal to the worst-case processing of one pass through the control loop. Even worse, the application’s response time to any external event changes whenever the control loop is modified or anything is called from the control loop. This situation makes the application unstable and extremely difficult to maintain and improve on.

Multitasking provides fast and deterministic response time to high-priority events. In multitasking, a higher-priority task can preempt an already executing lower-priority task. As a result, the worst-case response time is basically the time required to perform a context switch. Not only is this case deterministic, it’s also fast. A context switch has three principal aspects: saving the executing task’s environment, finding the next task to execute, and restoring that task’s environment. The assembly language segment in Listing 2 is an example of context switching on a Motorola 68000 target.

The example in Listing 2 assumes that upon entry into _save_task_context, a 68000 interrupt record is at the top of the stack. The example also assumes that the variable _next_task is set up somewhere in the program to point at the control block of the next task to be executed. This pointing occurs as a result of a system service call made from an interrupt service routine. Finally, assume the task’s stack pointer is saved in the first long word of its control block. Although primitive, this example provides a reasonable lower bound for task context switches on a 68000 processor. Using the Software...
Development Systems' 68000 simulator with zero-wait states selected, execution of the example context switch code required 278 clock cycles, or 17.4 μs, on a 16MHz 68000.

A fast and deterministic response time allows application developers to concentrate on specific requirements of each application task without worrying about what effect each task might have on other system response times. Furthermore, future modification of the program is much easier because the developer doesn't have to worry about affecting existing responsiveness with changes in unrelated areas.

**THROUGHPUT VS. OVERHEAD**

Possible workaround to the control-loop response time problem is to simply add more polling. Listing 3 shows an additional two polling calls to check for the presence of high-priority serial input. This option improves the responsiveness, but still doesn't guarantee a constant response time and does nothing to ease the future modification of the application. Worse yet, the processor is now performing more unnecessary processing because of this extra polling. All of this extra polling in the application's control loop reduces the total throughput of the system.

An interesting point regarding overhead should be made. Most of us would assume that multitasking increases overhead and has a negative impact on total system throughput. But in some cases, multitasking actually reduces overhead by eliminating all of the redundant polling that occurs in non-multitasking processor allocation. The overhead of multitasking is a function of the time required for context switching. If the context switch time is less than the specific polling processing, the multitasking environment provides a solution with the potential of less overhead and more throughput.

**EASE OF DEVELOPMENT**

In non-multitasking environments, each engineer should have intimate knowledge of the run-time behavior and requirements of the complete system. Each engineer must have this knowledge because the processor allocation logic is sprinkled throughout the application. Small embedded applications (usually less than 32K) are often handled by one or two engineers. As the application increases in size and complexity, the number of engineers involved increases as well. It is almost impossible for each engineer on a larger project to know the exact processing and response time requirements of all other parts of the application.

Multitasking, on the other hand, frees each engineer from the worries associated with processor allocation and allows them to concentrate on his or her specific piece of the application software. In addition, multitasking forces the application to be broken up into clearly defined pieces. This breaking up alone might make multitasking an advantageous paradigm for many embedded applications.

**PORTABILITY AND MAINTENANCE**

Most multitasking operating environments provide a layer of abstraction between the underlying processor and the application. Processor-specific activities like context switching and interrupt processing are inherently handled by the multitasking environment. This makes the application more processor-independent and therefore portable to other processors. In fact, even porting an application to another multitasking environment is frequently quite easy, provided it has similar functionality.

As mentioned previously, multitasking removes the processor allocation logic from the application-specific software. Therefore, making improvements or repairs to some code within one task should not have much (if any) impact on the rest of the system. Conversely, in single-threaded or control-loop environments, a small change in any part of the application software could disrupt the timing to the extent that the system no longer functions.

**COMMON PITFALLS**

To this point, we have primarily touted the virtues of multitasking. Well, multitasking isn't all a bed of roses. The biggest problem with multitasking is its

```
LISTING 2
Context switching on a Motorola 68000 target.

/save_task_context:                ; Assume that interrupt stack frame
   move.l  d0,d1/a0:a6,(a7)        ; is present at the top of the stack
   orl.w  #$0700,sr                ; Lockout interrupts
   move.l  _current_task,a0       ; Get current task pointer
   move.l  a7,(a0)                 ; Save task's stack pointer
   clr.l _current_task            ; Clear current task pointer
   move.l  _system_stack,a7       ; Switch to the system stack
   jmp _scheduler                 ; Return to the scheduler

......

/_scheduler:

   move.l  _next_task,d0          ; Pickup the next task to execute
   beq _scheduler                 ; If nothing to do, just keep checking!
   jmp _restore_task_context; Otherwise, transfer control to task

......

/_restore_task_context:

   orl.w  #$0700,sr                ; Lockout interrupts
   move.l  d0, _current_task       ; Setup current task pointer
   move.l  d0,a0                   ; Prepare to use task pointer
   move.l  (a0),a7                 ; Switch to task's stack
   move.l  (a7)+,(d0-a7)           ; Recover task's registers
   rte                              ; Recover task's SR and return to it!
```
misuse, which translates into excessive memory usage, unwanted run-time behavior, and excessive processor overhead. Debugging multitasking systems has some pitfalls of its own. However, with a little care and planning, most of these pitfalls can be avoided.

**MEMORY PITFALLS**

Multitasking run-time environments almost always require more memory than simple control loop run-time environments. This requirement makes sense because a multitasking environment offers a richer set of run-time services. Many full-featured real-time kernels for the Motorola 68000 family have instruction areas ranging in size from 5K to 15K. Watch out for environments that don’t scale down towards the 5K size—in addition to the memory issue, larger kernel instruction areas might also indicate slower performance and more overhead.

Multitasking environments require more memory for their data. Data memory is needed for the kernel’s own stack, task control blocks, task stacks, and other objects, such as queues, mailboxes, and semaphores. Typical multitasking kernels for the 68000 require 1K for the system stack, less than 50 bytes for task control blocks, and a minimum task stack size of 100 bytes. Of course, the specific size of each task stack is determined by what happens inside the task.

Each task stack must be large enough to accommodate the worst-case function call nesting and local variable allocation. For example, a task that doesn’t have any function calls or local variables may require a stack size of 100 bytes. However, another task in the same application might have significant function call nesting and large local data structures. Such a task might require 2K for its stack. The basic points here are to avoid designing an application with excessive tasks and to make sure that each task’s stack can accommodate the worst case function call nesting and local variables.

**PRIORITY PITFALLS**

The first warning about tasks is to avoid as many of them as possible! This might sound funny, but it’s good advice. Recall that each task requires its own stack for context management and its own high-level function call usage. In addition to the extra stack memory, each new task introduces more run-time complexity, making it harder to predict what and when things will happen. For example, describing in detail the run-time interaction of a five task system is often much easier than for a 200-task system. The moral, of course, is to avoid overly complicated run-time behavior.

Selecting task priorities is one of the most important aspects of using multitasking. Unfortunately, priority selection is often overlooked or not given proper consideration in many embedded systems. Assigning priorities based on how important the associated task seems, rather than looking at what your selection means during run-time, is sometimes very tempting. Misuse of task priorities can starve other tasks, create priority inversion, drain processing power, and make the system’s run-time behavior difficult to predict.

A basic rule of multitasking systems is that lower priority tasks don’t execute until no higher-priority tasks are ready. If higher-priority tasks are always running, lower-priority tasks never execute. This condition is called starvation. Most starvation problems are detected early in integration, but complicated systems (those with many tasks and priorities) may have some hidden starvation conditions. The first step to prevent starvation is to reduce the number of tasks and different task priorities. This will make the system easier to understand and therefore easier to reduce the risk of an unforeseen starvation condition. Another possible solution is to gradually raise a starving task’s priority inside one of the more frequently executing high-priority tasks. Although a little trickier, this approach is reasonable when starvation conditions are unavoidable.

**LISTING 3**

Polling calls to check for the presence of high-priority input.

```c
main()
{
    /*Look for high-priority serial input. */
    poll_for_serial_input();

    /*Some other program task "a." */
    program_task_a();

    /*Look for high-priority serial input again. */
    poll_for_serial_input();

    /*Some other program task "b." */
    program_task_b();

    /*Look for high-priority serial input. */
    poll_for_serial_input();

    /*Some other program task "c." */
    program_task_c();
}
```

**LISTING 4**

Code segments for tasks_1, 2, and 3.

```c
task_1_b_and_c()
{
    /*Loop forever. */
    while(1)
    {
        /*Wait for message from queue. */
        my_queue_poll();

        /*Send message to next task's queue. */
        send_to_next_task();
    }
}
```
Another unpleasant situation that arises from having too many tasks of different priorities is known as priority inversion. Priority inversion occurs when a higher-priority task is blocked while waiting for a resource held by a lower-priority task. Of course, in some situations two tasks of different priorities must share a common resource. If only these two tasks are active, the priority inversion is a function of how long the lower-priority task keeps the resource. This condition is both normal and deterministic. However, if tasks with priorities that fall between the two previously mentioned tasks are activated, then the time of the system’s priority inversion condition becomes less predictable. Priority inversion is bad enough, but unpredictable priority inversion is unacceptable.

Several possible solutions to this problem exist. First, the run-time environment can temporarily raise the priority of the task holding the resource to that of the highest priority task waiting for the resource. This approach will alleviate the unpredictable priority situation; however, there is a cost. The run-time environment in this case must examine the task holding the resource each time a task is blocked on the resource. If the running task’s priority is lower than that of the suspending task, the priority must be raised to that new level. This solution also requires a linear search of the task suspension list when the lower-priority task finally releases the resource. All of this processing requires time and in the real-time embedded world, there just isn’t enough of it! Worse yet, this extra overhead is incurred on all resource management calls—even those without the potential of priority inversion.

Because the run-time environment can’t afford any extra overhead, the best technique to handle unpredictable priority inversion is to prevent it in the application. Typically this is accomplished through a careful selection of tasks, shared resources, and, most importantly, task priorities. The application may also temporarily raise task priorities to avoid undeterministic priority inversion.

EXCESSIVE OVERHEAD

One of the most overlooked methods of reducing overhead in multitasking environments is to reduce the number of context switches. The first technique one can use is to limit the number of tasks. The fewer tasks there are in the system, the fewer the number of context switches. Although reducing the number of tasks is a good start, it’s the task priorities that require close consideration.

As mentioned previously, a context switch occurs when the execution of a running task is suspended in favor of the execution of another task. This situation occurs whenever a higher-priority task becomes ready during the execution of a lower-priority task. It’s important to realize that higher-priority tasks can become ready as a result of external events or by the actions of an executing task.

To illustrate the effects task priorities have on context switch overhead, assume a three task environment, with tasks named task_1, task_2, and task_3. Assume that all of the tasks are suspended, waiting for a message. When task_1 receives a message, it forwards it to task_2. Task_2 then forwards the message to task_3. When task_3 receives the message, it just throws the message away. After all of the tasks receive and process the message they go back and wait for another one. Listing 4 shows a small C code segment for each of these tasks.

Notice how the amount of processing required to perform this sequence of events is greatly affected by the individual priority of each task. If all of the tasks have the same priority, a single context switch occurs between their execution. However, if task_2 is higher priority than task_1, and task_3 is higher priority than task_2, the number of context switches doubles. Why? Because the act of sending a message to a higher-pri-
ority task causes a context switch. In this simple example, the overhead doubled and probably would account for as much as 60% of the processing time. Figure 2 illustrates the difference in run-time behavior relative to different task priorities.

This example is extremely simple, but it does effectively illustrate what can happen if task priority selection isn’t given the proper amount of consideration. Remember to carefully select task priorities, and only make a task higher priority when you really want its execution to pre-empt another. If you aren’t careful, you may accidentally use multitasking to transform your new 33MHz processor into a 16MHz part!

**DEBUGGING**

Another common pitfall in using multitasking is in the area of debugging. Debugging a multitasking application is complicated because the same application function may be executed from a variety of different tasks. Understanding multitasking is all that’s necessary. In fact, this problem is being addressed through advances in source-level debuggers. Many of today’s compiler tool vendors are offering task-aware extensions to their products that provide such things as task-aware breakpoints and other multitasking information.

**BUY VS. BUILD**

Assuming that multitasking is beneficial to your application, the next question is whether to buy a kernel or build it yourself. Building the environment yourself allows you to tailor it to your specific needs. However, most custom environments require significant development time, are more expensive, less functional, and less portable than commercial alternatives.

Buying the right commercial multitasking product could be a better approach. Generally speaking, a commercial product may cost the equivalent of about three to four weeks’ salary. Using a commercial product allows your team to concentrate on the actual application instead of the run-time environment, which helps get your product to market faster. Because commercial multitasking products have to support many different processor families to stay in business, your software investment is protected.

Never underestimate the positive effects of customer feedback. Commercial kernel vendors deal with a variety of customers daily. Because of their inherently unique perspective, the comments and suggestions of customers help to mold commercial kernels into even better products. Today’s commercial kernels represent, to some degree, the comments and suggestions of thousands of users. It is impossible to have this kind of natural evolution with in-house kernels.

Having said these things about commercial products, you should also realize that a vast difference exists between kernel vendors and their products. Some kernel vendors are still charging run-time royalties. At some point in the near future, this will seem as silly as compiler vendors charging royalties for the object code they generate. The functionality of commercial multitasking environment also varies greatly. You should review in detail the functional capabilities, as well as the application interfaces, before any purchase. Some multitasking environments might be too big or slow for your application. Some products just might not have the functionality you need. The amount of difference between vendors is really quite amazing and should be closely studied.

Even if you aren’t yet using multitasking (a significant number of embedded applications aren’t), don’t be surprised if it’s soon a consideration—keep this article handy.

Thanks to Software Development Systems and Motorola for providing the tools and microprocessor, respectively, used for the examples in this article.

Bill Lamie is the author of ThreadX® and the president of Express Logic, Inc. He may be reached at blamie@expresslogic.com.