Preemptive Virtues

by
Ralph Moore
President, Micro Digital Inc.

Multitasking is fact of life for most embedded systems developers. We have little choice in the matter: few systems these days are so simple that you can dispense with using a task model in your design. Once you've decided to use multitasking, however, the choices start to appear. But which approach to multitasking will you choose?

We have three basic methods of multitasking. The simplest is the round-robin scheme, a variation of a loop where each task has its turn and runs until finished. Slightly more sophisticated is the time-slice approach, where each task is allowed to run for a fixed time and is then suspended until the next cycle. Finally, we have the preemptive model, where the most important ready task runs until it is finished or preempted by a more important task. The first two methods repetitively cycle through all of the tasks. The third method does not.

Which of these methods is best for a real-time system? I think the preemptive-model method is the best. Other articles have discussed the first two methods under the heading of "cooperative" multitasking. (See "Heavyweight Tasking," by Philip Koopman, Jr., Embedded Systems Programming, April 1990, pp. 42-52 and "Cooperative Multitasking," by Jack Woehr, Embedded Systems Programming, April 1990, pp. 54-61.)

Real-time systems are preemptive by their very nature, which is why they use interrupts extensively. For simple systems, it is often possible to do all processing using interrupt-service routines (ISRs). This approach is the easiest. However, it is not usually sufficient-especially since new designs are increasing in complexity.

Most systems have a mix of frequent interrupts, which require little processing, and less frequent interrupts, which require more processing. Since external events are usually asynchronous, all interrupts caused by them may occur at once. This situation causes a temporary overload of the processor and can cause subsequent, frequent interrupts to be missed - resulting in system malfunction. The problem is often solved by moving extensive processing into background functions. (For purposes of this article, we define background to be code that is not highly time-critical and that usually runs with interrupts enabled. We define foreground as highly time-critical code that usually runs with interrupts disabled.) Using this approach, ISR code is held to a minimum so that interrupts can be reenabled as soon as possible. (ISRs are usually nonreentrant for speed, so it is customary to not allow them to be interrupted.)

Moving extensive processing into the background solves a major problem for most systems. However, it introduces the question of how to communicate between the foreground ISRs and the background functions. A common approach is for each ISR to set a flag when a background service is required. These service flags are monitored by a background code loop (variously called an idle loop or a super loop.) The background loop calls functions as their flags are set.

This approach is basically round-robin scheduling. It works fine if each function has adequate time to complete before being called again. Background functions are usually non-reentrant (again, for speed. They can also use non-sharable resources). Furthermore, some functions may need to complete within specified time constraints. These factors often lead to the problem we encountered in the foreground -- a temporary processor overload resulting in malfunction.

The usual solution to processor overloading is to modify the loop so that some flags are tested more often than others. Also, long functions may be broken into fragments so that more urgent functions can run in between the fragments. These performance problems typically occur late in a project and are solved by quick fixes. Under such circumstances. Well-crafted code degenerates into "spaghetti code." Spaghetti code is code that has lost its integrity. It has become subverted by a set of requirements and has lost its clean, simple structure. Worse, it is not easily extensible and may be understood only by its originator. Unfortunately, this type of subversion is common. (For the past generation of microprocessor software, it may be the norm!)

It is my theory that spaghetti code arises from a mismatch between foreground and background. The foreground is characterized by two important attributes: priority and preemption. A round-robin or time-sliced background lack these attributes, and that is where the problem lies. Clumsy attempts to fix the background lead to spaghetti code.

Preemption is a natural phenomenon -- as are priorities. We even run our own lives this way. For example, you are programming and the phone rings. What do you do? You answer it! We are so accustomed to such interruptions, we scarcely pay any attention to them. Software that allows preemption is bound to handle real-world events better. Here again, programmers who respond to interrupts and crises are more effective than programmers who refuse to be interrupted.

Let's look at how a real-time, multitasking kernel averts the dreaded Spaghetti Software Syndrome. To start with, we have the concept of a task. A task is not code; it is a portion of work. In software, the task is translated into a task-control block (TCB), which contains information about the task, a stack for the task's local variables, and associated code, which often is reentrant so that it can be used by other tasks. A TCB typically includes a forward link, a backward link, a priority, a control block type, a return value, a stack handler, a stack pointer, and a function pointer. Forward and backward links are pointers used to link the task into various queues (only the pointers change; nothing really moves). The fun field points at the code used by the task. (Code is just a list of instructions telling the processor how to accomplish the task.) The other fields are not important here. Every kernel has its own TCB format, this one is just an example. (This and subsequent examples are based upon smx, a simple multitasking executive.)

Interstate Transfers

The next topic to consider is what task states are allowed.

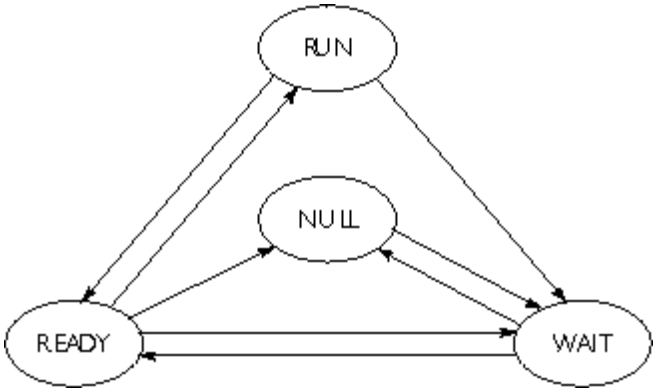

Figure 1 is an example of task states and task-state transitions. This diagram is representative of most kernels. NULL means that the task has not been created yet or has been deleted. WAIT means it is waiting for an event. READY means it is ready to run. RUN means it is actually running. In a single-processor system, only one task at a time can be in the RUN state. Most of the time, this task will be the highest-priority task that is ready to run.

If you have ever watched streams of ants, you have some idea of how multitasking systems work -- organized chaos! Individual ants run into other ants going the opposite direction. They get turned around, and go off in the wrong direction. Other ants go off in circles. Yet, somehow, the job of bringing food home to the nest gets done.

Like an ant, a task steadfastly pursues its own little mission. As with an ant colony, the mechanism for overall system control is not obvious to an observer. It is indeed a minor miracle resulting from the following:

- Usually running the highest priority task when it is ready
- Communication from task to task
- Communication from foreground service routines to tasks.

To use the first scheme, the scheduler (which is part of the kernel) must occasionally gain control. We have two mechanisms to accomplish this task. The first is when the current task (the one in the RUN state) makes a kernel call. Following such a call, control passes to the scheduler. The kernel call may have suspended the current task (put it into the WAIT state), caused a higher priority task to become ready, or neither. The scheduler sorts this information and decides whether to continue the current task or run another.

The second way the scheduler gains control is when an interrupt occurs. Following an ISR, control usually passes to the scheduler. The ISR may have caused a higher priority task to become ready. Again, the scheduler decides what to do. It gives the system an adaptive nature determined by priorities, which is a valuable characteristic because it makes the system behave in a reasonable manner (the most important task runs first). It also makes the system relatively easy to design and tune by changing priorities.

The remaining two means of system control involve objects provided by the kernel for intertask communication. Most kernels provide semaphores and message exchanges (also called "mail boxes"). These mail boxes are usable from foreground service routines as well as from tasks.

Some tasks and service routines send messages to exchanges. Other tasks receive messages from exchanges and process them. If an exchange has no messages, a task attempting to receive a message will be suspended on the exchange and enter the WAIT state. The next-highest priority task will then run. When a message is sent to the exchange, the waiting task will be resumed and enter the READY state. If this task is of higher priority than the current task, which could be any task at this time, it will run. Messages may pile up at exchanges during periods of peak activity. This situation is not a problem. These messages form a work stream similar to units on an assembly line moving from one workstation (task) to the next.

Semaphores operate in a similar manner, except that no messages are involved. A task may wait at a semaphore. Another task may signal the semaphore causing the waiting task to resume. Semaphores provide a means for one task to jog another.

Tasking in Practice

So much for theory. What does the code look like? It's pretty straightforward. To create a task we simply give the task a name and assign it code, such as a C function, and a priority:

```
TCB_PTR atask;
atask = create_task( atask_main, PRIORITY );
```

atask is a pointer to the task's TCB.

From here on, *atask* identifies the task. For example, to put the task into the ready queue and make it ready to run:

```
start( atask );
```

The *create_task()* and *start()* are kernel calls. The task's code is a C function:

```
void atask_main( void )
{
/*initialization */
/* operation */
}
```

The following code could be reentrant and shared by many tasks:

```
TCB_PTR task[N]
for {i = 0; i < N; i++}
    task[i] = create_task( atask_main, PRIORITY );
```

These tasks, *task[0], task[1],...,task[N],* share the atask_main code. How do tasks exchange messages? First, create a message:

```
MCB_PTR amsg;
amsg = create_nmsg( 100 );
```

This code takes 100 bytes from the near heap and allocates a message control block (MCB) to handle it. We need an exchange to send the message to:

```
XCHG_PTR anxchg;
anxchg = create_xchg( NXCHG, 0, 0 );
```

This code creates a normal exchange with one task level and one message level. Now, *taskA* loads information into the message and sends it to *anxchg*:

```
void taskA_main()
{
/* fill amsg */
send( amsg, anxchg );
}
```

Then *taskB* receives the message and processes it:

```
void taskB_main()
{
MCB_PTR m;
if (m = receive( anxchg, SEC ))
    /* process m */
else /* alternative action */
}
```

This task waits up to one second for a message to arrive at *anxchg*. If no message arrives, then it takes alternative action.

The main problem I've observed in newcomers to preemptive multitasking is their tendency to overly control. For example, time-slicing seems to be a security blanket that is hard to give up. Firing off signals to semaphores, where other tasks may or may not be waiting, seems a difficult act of faith. Sending messages blindly to exchanges seems overly adventuresome. A strong need for determinism and sequential ordering exists. Then comes the dawn -- the programmer realizes that complex, real systems aren't very deterministic. They are best controlled by nonsequential means! Once you have cleared these conceptual barriers you have a fascinating opportunity to design real-time systems in a new and, hopefully, better way.

Ralph Moore is a veteran of 15 years in microprocessor applications and the architect of smx. He is President of Micro Digital, Inc.