One-Shot Tasks

reduce RAM usage

by Ralph Moore
***smx* Architect**

Introduction

Many of the newer processors and SoCs have a significant amount of on-chip SRAM. Using on-chip SRAM is much faster than external SDRAM and, for some systems, may avoid the cost of external SDRAM and even having an external processor bus. Since SRAM uses much more chip space than NOR flash, optimizing on-chip SRAM usage is very important for both cost and performance.

In multitasking systems, each active task needs its own stack. Therefore, task stacks can require a major amount of RAM. To mitigate this problem, *smx* supports what are known as *one-shot tasks*. One-shot tasks run once from start to finish, then stop. When stopped, no information is carried forward to the next run. Thus, a one-shot task does not need a stack when it stops, and its stack is released back to the stack pool for use by another task. When a one-shot task is dispatched, the *smx* scheduler gives it a stack from the stack pool. Thus, a system with many one-shot tasks can operate with only a few stacks as long as only a few one-shot tasks need to run simultaneously.

Stopped *smx* tasks can wait at semaphores, mutexes, exchanges, etc – just like conventional *suspended* tasks, so there is no loss of flexibility. Starting a task takes the same length of time as resuming a task, so there is no loss of performance.

Theory

RAM Usage

Normally, it is necessary to have one stack per active task. There are two exceptions to this rule. One is the OSEK BCC1 *one-shot execution model*, which shares one stack between all tasks. It has serious limitations -- see the discussion at the end of this paper. The other exception is *link service routines* (LSRs). LSRs are another feature of smx that can be used to reduce RAM usage. They act like tasks, but run in the context of the current task and use its stack. For more information, see smx Link Service Routines.

In the general case, if one tries to share one stack among many tasks, a problem develops because, in a normal system, tasks will resume in a different order than they were suspended. Soon, most tasks will have fixed-size stack spaces that do not permit deeper nesting of functions nor handling of interrupts. Although various complex stratagems might be employed, the most practical solution is to allocate a stack per active task.

Copyright © 2005 by Micro Digital, Inc. All rights reserved. The information on this page is proprietary to Micro Digital, Inc. and may not be copied, stored, or transmitted, without written permission.

This being the case, how large must task stacks be? Unfortunately, even a lean kernel is likely to impose the need for about 200 bytes of stack. Added to this are the requirements of the tasks, themselves, including maximum function nesting and space for autovariables, which can easily add another 200-400 bytes. Some standard C library functions (e.g. sprintf( )) can add much more than this, just by themselves. This number does not even include saving a coprocessor state or making room for floating point emulation.

Finally, space is needed for the maximum possible nesting of interrupts — another 200 bytes, or so. It is true that interrupt service routines could have their own stacks. However, that adds delay at the most critical point of most embedded systems — namely, at the point of interrupt processing. Hence, although done in non-real time operating systems (e.g. MS-DOS) interrupt stacks are not a good idea for high-performance, embedded systems. Interrupt stacks also create other complexities and may not even save much memory.

Hence, we end up with 600 to 800 bytes. Stacks can easily go to 1000 bytes or more when a file system or networking is added. Thus, an embedded system with 20 tasks is likely to require about 20 KB of RAM.

Fine-Grained vs. Coarse-Grained Task Structure

Because of the significant memory required per stack, the tendency on the part of many programmers is to minimize the number of tasks. This has the unfortunate consequence of making each task more complex than is necessary. Large tasks defeat the purpose of a multitasking kernel, which is to simplify the programming job by dividing the application into simple tasks. Simple tasks achieve complex results via their interactions. Indeed, creating a few large, complex tasks is akin to creating *processes* in big multi-user systems, such as UNIX. This is not the correct way to use a high-performance, multitasking kernel.

An example will clarify this. Suppose we are developing a communication system. Usually it works well to have at least two tasks — a receive task and a send task. Suppose there are 10 identical channels. How many tasks should there be? I suggest 20, but many designers would struggle with only two. Why are 20 better? Because:

(1) The code is simpler — it only has to deal with one channel. Exactly the same receive code is used by all 10 receive tasks. Similarly for the send code. Thus to have 20 tasks does not add any more code than having two. In fact, the amount of code required may be much less, because it is simpler.
(2) More of what needs to be done is being done by proven kernel code. For example, proven kernel objects, such as semaphores and messages, are likely to be used more and untested application mechanisms, such as flags and buffers, are likely to be used less. Kernel code is orders of magnitude more reliable than brand-new application code.

Copyright © 2005 by Micro Digital, Inc. All rights reserved. The information on this page is proprietary to Micro Digital, Inc. and may not be copied, stored, or transmitted, without written permission.

(3) It is scalable — channels can be added or removed merely by creating or deleting tasks. Code need not be changed at all.
(4) It is dynamic — tasks can be created and deleted on the fly.
(5) The final design is easier to understand and to maintain by different programmers than the originators.
(6) Interfaces between sections of code are defined by the kernel API. Hence, they are more robust and better documented.
(7) The kernel adds extensive error checking and error reporting. These seldom get put into application code, yet they speed debugging.

So, in this simple example, there are abundant reasons to support a *fine-grained* task structure. When dealing with the needs of high-performance, high-reliability embedded systems, some rethinking of basic concepts is appropriate. First, we need a kernel (not an os) that is *mean and lean*. Then we need to learn how to best utilize that kernel to achieve the project goals. One-shot tasks are a step in that direction.

One-Shot Task Operation

So, let's get back to the subject of stacks. Wouldn't it be nice if at least some tasks could share stacks? Then there would not be so many stacks. This is where we introduce the concept of the *unbound* stack. What is an unbound stack? An unbound stack is a stack, which is not permanently bound to any particular task — i.e. it can be shared. How is this possible? It works as follows: Unbound stacks are all the same size and belong to a single *stack pool.* When a one-shot task is first dispatched (i.e. started running), unless it already has a permanently bound stack, it is loaned a stack from the stack pool. As long as the task is active (i.e. either running or suspended), it keeps the stack. Hence, the task and stack function normally when the task is active. However, when the one-shot task completes its function and is ready to wait for more work, it gives up the stack. We call this *stopping* the task as opposed to *suspending* it. In the latter case, the task would keep the stack, but in the former case, the task has no need for the stack so the task gives it up. (In this regard, it is important to know that a one-shot task is later restarted from the beginning rather than being resumed from where it left off.)

The stack goes back to the stack pool where it is available to be used by another task. Now the one-shot task is in what we call the *unbound mode*:

Copyright © 2005 by Micro Digital, Inc. All rights reserved. The information on this page is proprietary to Micro Digital, Inc. and may not be copied, stored, or transmitted, without written permission.

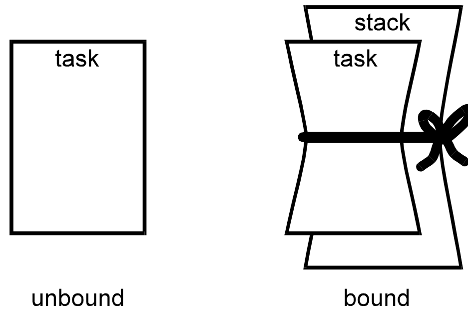

Figure 1: Task Modes

In this mode, the task has no state variables (e.g. register contents) and no auto variables worth remembering. This state is appropriate only if the task has completed all of its work and is ready to start over again, or to *restart*, as we call it. This necessitates a different code structure as shown in Figure 2.

Code Examples

a. Conventional Task:

```
void  atask_main (void)
{
   MCB_PTR msg:
   // initialize
   unlock (self);
   while (msg = receive (xchg, INF))
   {
      // process msg
   }
}
```

b. One-shot Task:

```
void  otask_main_init (void)
{
   // initialize
   // change task main function:
   task.fun = otask_main;
   receive_stop (xchg, INF);
}

void  otask_main (MCB_PTR msg)
{
   // unlock (self);
   // process msg
   receive_stop (xchg, INF);
}
```

Figure 2: Code for Convention and One-Shot Tasks

The conventional task is created with atask_main( ) as its main function:

```
TCP_PTR  atask;
atask = create_task(atask_main, NORM, 1000);
start(atask);
```

atask has normal priority and a permanently bound stack of 1000 bytes. When *atask* first starts, it initializes and unlocks itself (so it can be preempted by higher priority tasks), then goes into an infinite while loop. At the start of this loop, the task waits at *xchg* for a message. When a message is received, it is processed, then the task waits at *xchg* for another message. This process repeats for every new message received. In this particular example, *atask* never exits the while loop unless it is externally stopped or deleted.

Copyright © 2005 by Micro Digital, Inc. All rights reserved. The information on this page is proprietary to Micro Digital, Inc. and may not be copied, stored, or transmitted, without written permission.

The one-shot task has a different structure. It is created with otask_main_init( ) as its initial main function:

```
TCP_PTR  otask;
otask = create_task(otask_main_init, NORM, 0);
start(otask);
```

otask has normal priority and no permanently bound stack. When *otask* first runs, otask_main_init( ) does the initialization, then *otask's* main function is changed to otask_main( ) and *otask* stops on *xchg*. When a message is received, *otask* restarts, except this time with otask_main( ) as its main function. Note that the message handle, *msg*, is passed in as a parameter, not as a return value, as it is for *atask*. This is because, no code executes after a stop function (i.e. receive_stop().) *smx* handles passing in the *msg* parameter from receive_stop(). *otask* may or may not unlock itself (see discussion below). The message is processed and *otask* again stops on *xchg*. otask_main( ) runs, from the start, each time a new message is received.

Note that the results produced by *atask* and *otask* are the same, but their mechanisms are much different.

More Considerations

In the above discussion, it was noted that *otask* may or may not unlock itself. This is because, if *otask* is very short, it is probably desirable to leave it locked in order to prevent excessive task switching. This is because a locked task cannot be preempted by any other task (but ISRs and LSRs can run.) Such a task is called a *non-preemptible one-shot task*. The reason that smx tasks start locked and must be deliberately unlocked, is to assure that they cannot be preempted once they start running.

It is not a requirement that one-shot tasks be locked. They can be unlocked. However, then they can be preempted and will thus hold borrowed stacks longer. This is not a problem, but a bigger stack pool may be required. Ideally, a one-shot task is a *tiger task*, like an interrupt service routine. It is best to let it rip.

One-shot tasks can loop, if necessary, such as to process a series of packets. One-shot tasks can also suspend themselves, such as to receive a packet, then process it. Hence, a one shot task can act like a conventional task to process a burst of activity. However, once it finishes its business, it stops and must be restarted to run again. When restarted, it always runs from the beginning of its main code. (This is why first-time initialization code must be put into a separate function that is run only once.)

Stack RAM can also be saved by deleting tasks, when no longer needed and recreating them when needed. However, this adds overhead whereas stopping and starting tasks adds no overhead versus suspending and resuming them (except that a task may have to wait for a stack). Also, a deleted task cannot wait for an event. In addition, it loses its TCB and handle and thus can no longer be tracked by smxAware nor a debugger.

Copyright © 2005 by Micro Digital, Inc. All rights reserved. The information on this page is proprietary to Micro Digital, Inc. and may not be copied, stored, or transmitted, without written permission.
\\server\d\smxd\articles\oneshottasks.doc 12/22/05

Advantages of One Shot Tasks

What is the advantage of *otask*? Let's return to our example and look at the 10 send tasks. Suppose outgoing traffic is very light and a send task spends 90% of its time waiting at an exchange for another message to send. Why tie up a valuable stack all this time? Quite possibly, 2 stacks (or even 1 stack) would satisfy the needs of the 10 tasks. Hence, using one-shot tasks would, in this one area, give an 80% reduction in stack space!

Three advantages are gained: (1) potential cost reduction due to eliminating external RAM; (2) potential increase in performance due to using on-chip SRAM; (3) freedom to utilize a fine-grained task structure to achieve the optimum architecture for the application at hand. With reference to (2), note that putting stacks into on-chip SRAM has particularly great impact upon improving performance. This is especially true for multitasking systems because there is a strong tendency to use autovariables, which are stored in task stacks, in order to achieve subroutine reentrancy.

What about disadvantages: (1) performance? It turns out that getting and binding a free stack is an efficient process. It costs no more time than restoring registers from a bound stack. Hence, there is no performance hit for using one-shot tasks. (2) reduced flexibility? *smx* permits a one-shot task to wait, in the stopped condition or in the suspended condition, at all of the same objects (e.g. semaphores, exchanges, etc) as a conventional task. Hence, a one-shot task can do everything a conventional task can do. (3) run out of stacks? If no stack is available, *smx* stops the task being started, for a tick, then tries to dispatch it again. In such a situation, processing is probably stacked up due to limited processor bandwidth, anyway; so waiting for free-stacks does not degrade overall performance. At worst, a high-priority, one-shot task could miss a deadline. If this causes a problem, such a task can be given a permanently bound stack to avoid the problem. One-shot tasks are good for operations that are simple, infrequent, largely mutually exclusive (or can wait a tick or two for stacks), and finish in one pass. Although they are not a panacea for all embedded systems, they are good tools to have in your toolbox. One-shot tasks have been an integral part of *smx* from the start and many applications have benefited from them.

smx vs. OSEK Stack Sharing

The *smx* one-shot task stack-sharing feature is similar to the OSEK BCC1 *one-shot execution model*. The latter allows sharing a single stack among all tasks. The required stack space is determined by the sum of the maximum stack sizes of all priority levels. For example, if there were two priority 3 tasks requiring stacks of 200 and 300 bytes, then 300 would be added for level 3. This model requires that every task run to completion, except while preempted by a higher priority task.

The *smx* approach requires a bit more stack space but it is less restrictive. Also, it is easier and less error-prone to calculate the stack space required. Under *smx*, stacks may be either retained or surrendered when waiting for messages, signals, events, timeouts,

Copyright © 2005 by Micro Digital, Inc. All rights reserved. The information on this page is proprietary to Micro Digital, Inc. and may not be copied, stored, or transmitted, without written permission.

etc by using either the suspend or stop version of the *smx* call. Hence, stopping vs. suspending a task introduces no constraints on what the task can do. Also, when a task that surrendered its stack is restarted, it is given a new stack from the stack pool, and the task starts from the beginning of its code. (Hence, information from the old stack is not needed.) There is no performance penalty for *starting* a task with a new stack vs. *resuming* a task that already has a stack.

In contrast to OSEK BCC1, *smx* allows a mixture of conventional tasks, with permanent stacks, and one-shot tasks, with temporary stacks. If we make the rule, like OSEK BCC1, that one-shot tasks may not suspend themselves, then the number of stacks required in the stack pool is equal to the number of different priorities of one-shot tasks. For example, if there are 10 one-shot tasks having 3 different priorities, then only 3 stacks are required in the stack pool. Unlike OSEK, these stacks must all be the same size.

Note that one-shot *smx* tasks can wait at semaphores, exchanges, and other *smx* objects, when stopped. OSEK one-shot tasks cannot do this. Also, to add conventional tasks, OSEK requires changing to the ECC1 or ECC2 model, neither of which permits stack sharing. Hence, it is all or nothing with OSEK. With *smx* no such limitation exists -- there can be any number of either kinds of task. Not only that, one-shot tasks can behave like conventional tasks until they stop. Because of these differences, *smx* gives a programmer much greater flexibility in fitting an application into limited RAM space, than does OSEK.

For more information:

1. *smx User's Guide* – See Chapter 16, More on Tasks.
2. *smx Reference Manual* – See stop versions of smx services.

Copyright © 2005 by Micro Digital, Inc. All rights reserved. The information on this page is proprietary to Micro Digital, Inc. and may not be copied, stored, or transmitted, without written permission.