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 normal 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 © 2005-2012 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.

smx uses the System Stack for ISRs, LSRs, the scheduler, and the error manager. This saves on the order of 200 bytes per task.

Hence, we end up with about 500 bytes per stack. Stacks can easily go to 1000 bytes or more when a file system or networking is added. Thus, an embedded system with 20 tasks may require 10 to 20 KB of RAM for task stacks, alone.

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 really 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 should be 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 much more reliable than brand-new application code.
(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.

Copyright © 2005-2012 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.

(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 many 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 *lean and mean*. 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) 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 its 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 the *unbound mode*:

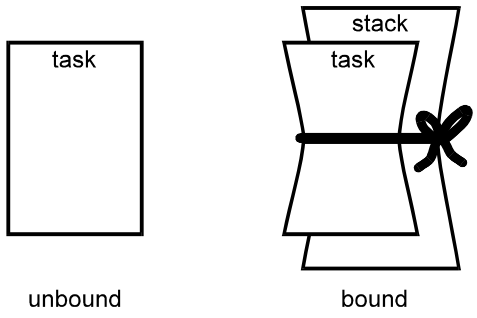

Figure 1: Task Modes

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

Copyright © 2005-2012 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.
12/21/12

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[1].

Code Examples

a. Normal Task:

```
void  atask_main (void)
{
  MCB_PTR msg;
  /* initialize atask */
  while (msg = smx_MsgReceive (xchg, INF))
  {
    /* process msg */
  }
}
```

b. One-shot Task:

```
void  otask_main_init (void)
{
  /* initialize atask */
  task.fun = otask_main;
  smx_MsgReceiveStop (xchg, INF);
}

void  otask_main (MCB_PTR msg)
{
  /* process msg */
  smx_MsgReceiveStop (xchg, INF);
}
```

Figure 2: Code for Normal and One-Shot Tasks

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

```
TCP_PTR  atask;
atask = smx_TaskCreate(atask_main, PRI_NORM, 1000, NO_FLAGS, "atask");
smx_TaskStart(atask);
```

atask has normal priority and a permanently bound stack of 1000 bytes. When *atask* first starts, it initializes itself, 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.

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

```
TCP_PTR  otask;
otask = smx_TaskCreate(otask_main_init, PRI_NORM, 0, NO_FLAGS, "otask");
smx_TaskStart(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. smx_MsgReceiveStop().) smx handles passing in the

[1] Note: This code is simplified to convey the basic concepts being discussed.

Copyright © 2005-2012 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.

msg parameter from smx_ReceiveStop(). 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 different.

More Considerations

If *otask* is very short, it may be desirable for it to run locked in order to prevent excessive task switching. This is because a locked task cannot be preempted by any other task (yet ISRs and LSRs can still run.) Such a task is called a *non-preemptible one-shot task*. To run otask locked, it is created it as follows:

```
        atask = smx_TaskCreate(otask_main_init, PRI_NORM, 0, SMX_FL_LOCK, "atask");
```

and then it is not unlocked, when it runs.

It is not a requirement that one-shot tasks be locked. They can be created 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 ISR. It may work 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 normal 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.

Advantages of One Shot Tasks

What is the advantage of *otask*? Let’s consider an example of 10 send tasks sending to 10 different ports. Suppose outgoing traffic is 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 send tasks. Hence, using one-shot tasks would, in this one case, give an 80 to 90% reduction in stack space.

Copyright © 2005-2012 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.

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 many variables are auto variables or parameters, for reentrancy, and they often are stored in task stacks.

What about potential disadvantages:

(1) Performance? Starting a one-shot task and resuming a normal task take about the same time — getting a stack is balanced by not having to restore registers.
(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 normal task. Hence, a one-shot task can do everything a normal task can do.
(3) Run out of stacks? Should the stack pool run out of stacks, the smx scheduler will skip to the next ready task that can run. Each time the scheduler is entered it will try again to run the stalled one-shot task (assuming that it is still the top task). Eventually, a stack will become available and the task will run. If this performance hit is not acceptable, either the stack pool size can be increased, or the one-shot task can be converted to a normal task with a tightly bound stack.

One-shot tasks are good for operations that are simple, infrequent, 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 a good tool 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 OSEK BCC1 automotive standard incorporates a different stack sharing mechanism. For it, there is one big stack shared by multiple tasks. The major disadvantage of this approach is that tasks must be resumed in the reverse order from which they were suspended. Compared to OSEK BCC1 the smx one-shot task has three significant advantages:

(1) One-shot tasks can resume or restart in any order.

(2) One-shot tasks can wait at semaphores, exchanges, etc. when stopped.

Copyright © 2005-2012 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.

12/21/12

(3) Any mixture of one-shot and normal tasks can run simultaneously.

The OSEK 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.

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, so total smx RAM usage may be somewhat higher.

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 normal 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. In addition, one-shot tasks can behave like normal tasks until they stop. Because of these differences, smx provides much greater flexibility to fit an application into limited RAM space, than does OSEK.

For More Information

1. *smx User's Guide* – See Chapter 16, One-Shot Tasks.
2. *smx Reference Manual* – See the stop versions of smx services.

Copyright © 2005-2012 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.