smx Philosophy

by Ralph Moore

January 2014

Starting with the v4.0 release in October 2010 to the recent v4.2 release, smx has been undergoing a steady evolution. The following is a discussion of the philosophy behind that evolution.

What is not in the kernel ends up in the application

Many people think that a simple kernel is good because there is less to learn. The flip side of this is that there is more code to write, debug, and test. For example, if one needs a special type of semaphore and the kernel does not have it, then one must create code to perform what the missing semaphore would do. Avoiding extra work, like this, favors a kernel with a rich set of features over a simple kernel. It certainly is easier to learn the features of a well-written kernel than to reinvent them.

In this sense, I differ strongly from the erroneous common wisdom that "all kernels are equivalent and the less the better." So you would rather drive a clunker than a Ferrari? I don't think so!

The more tasks the better

This is another area where I differ from the beliefs of some people. It is much easier to design, code, and debug a simple task than a complicated task. Yet some people believe that the number of tasks should be minimized. I see this quite differently for the following reasons:

- It is easier to solve a complex problem by breaking it into smaller pieces.
- Keeping tasks simpler makes it easier to understand and debug them.
- Using more tasks places more reliance upon proven kernel services to handle the difficult parts of a design vs. using new, untested application code.
- More use of kernel services enforces more design standardization, because there are a limited number of ways to do things.
- Using more tasks, and hence more kernel mechanisms, make relationships more apparent and easier to debug.
- Using more kernel functions permits taking advantage of task debug features and smxAware to reveal complex relationships.
- Problems can often be fixed by changing task parameters, such as a task's priority, without rewriting code.
- Greater modularity of the final design.
- Ability to reconfigure easily by replacing tasks.
- Greater reuse of code, since simple tasks are likely to be more general.

Copyright © 2014 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.

Thus, smx is specifically designed to make using a large number of tasks practical.

Minimizing task RAM usage

Static RAM requires about 6 times as much chip space as SDRAM and flash memory, per bit. This makes on-chip SRAM very dear. Many processors cannot support external RAM, and for those that do, operating tasks out of on-chip SRAM generally provides much better performance.

In order to support large numbers of tasks, it is important to minimize RAM usage per task. A task's stack is usually its main use of RAM — often stacks require 500, 1000, or even more bytes. smx reduces task stack usage by providing a *system stack* and by providing special tasks, called *one-shot* tasks. If there is no system stack, each task stack must allow for maximum ISR and LSR nesting on top of maximum scheduler plus error manager depth. For smx, ISRs, LSRs, the scheduler, and the error manager all run under the system stack. As a consequence, smx task stacks may require only 200 bytes, each.

In addition, one-shot tasks are able to share stacks. One-shot tasks can wait upon all of the same events as normal tasks, without consuming stacks. They are ideal for simple functions that do something, then wait. Since there is no information to bring over from one run to another, they do not need a stack, while waiting. One-shot tasks are given a stack when they start and give it up when they stop. While running they behave like normal tasks — they can be preempted, suspended, etc. A group of mutually-exclusive, one-shot tasks needs only one stack. If the stack pool is empty, a one-shot task waits in the ready queue, while other tasks run. One-shot tasks need not be mutually exclusive — fewer stacks results in less memory for lower performance, which may be perfectly acceptable if the tasks are doing non-time-critical operations.

Maximizing performance

Performance is the dominant factor for selection of features and algorithms. smx is intended to be a race car, not a family sedan. This may seem out of keeping with other objectives stated in this paper, but it is not. The point is that there are good ways and bad ways to do things. Generally, simple solutions are faster and easier to use. So, seeking high performance is not necessarily at odds with other objectives. In fact, it often enables other objectives since it provides the performance margin to achieve them.

Providing multiple ways to do things

Many people think there should be only one way to do each thing. I disagree. This does not fit the wide range of requirements that exist in the embedded software universe. Nor does it allow for different preferences among programmers. My philosophy is to provide different ways, within reason, to accomplish needed functionality. Hopefully, one of the ways will be a good fit and appeal to a programmer as the way to go. With modern tool suites, functions that are not used are not linked in, so there is no good reason not to give programmers many options to solve their problems. My hope is that smx users will study

Copyright © 2014 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 manuals and the examples and experiment with different options in order to achieve really good solutions to their problems.

Trading performance for functionality and reliability

Using a middling ARM9 processor running at 400 MHz, smx can perform over 250,000 task switches per second! (Measured with two tasks alternating at the same priority level, by using smx_TaskBump().) This is far more than almost any imaginable application could need. Hence, it makes sense to use unneeded cycles to achieve more functionality, reduce debug time, and improve reliability. The latest smx release (v4.2) is doing this by provide optional features such as event logging, precise profiling, stack tracing, more error checking, etc.

Bombardment from space

I think we are not paying enough attention to threats posed by cosmic rays, electrical storms caused by the Sun, lightning, and other natural phenomena. These can damage critical variables stored in RAM and flash. They will become a bigger problem as feature sizes and operating voltages continue to decline. I often wonder how existing systems can work so reliably with current feature sizes.

As billions more systems go into operation, with even smaller feature sizes, there are going to be more bit flips with tragic consequences. smx is moving in the direction of doing something about this. I think the solution consists of lots of little things such as defining variables so that they have no undefined states, using codes such as Hamming code to catch single bit errors, redundancy of key variables, etc. In the future, more of these techniques will be incorporated into smx, as well as frequent automatic testing and fixing of queues, control blocks, etc.

Extensive error checking

I see an RTOS kernel as being like a network of freeways. There may be thousands of origins and destinations, but most traffic flows through kernel pathways. Hence the kernel is in an optimum position to serve as traffic cop. smx has always offered extensive parameter and other testing. I think this is very important not only to aid debugging, but also to improve system reliability, in the field. smx is progressing toward greater and greater error checking and is beginning to add error recovery features. An important aspect of this is to implement error checking and handling features so that they are efficient enough to leave in the final code. This way, application software can build upon the base provided by the kernel in order to provide error recovery systems that meet the needs of safety-critical applications.

Fool-proofing

The dictionary definition of *fool-proofing* is: *impervious to misuse*; in other words it is the opposite of error-prone. It does not mean that users are fools! This subject is given very little attention in discussions of reliability that I have seen, yet I believe it is vital for something as complex and difficult to use as a real-time multitasking kernel. What

Copyright © 2014 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.

difference does it make that a kernel has been certified, if users are frequently misusing it? Yet I am not aware of any standards on this subject.

Making smx as fool-proof as we can is a dominant goal of its architecture. To give a few simple examples:

- Most kernels go into undefined modes of operation if a task goes through the last curly brace of its main function. This does not bode well for reliable operation! Under smx, such a task goes into a dormant state, from which it can be restarted. It can even return a parameter to itself, such as an error code, before it goes dormant.
- The smx base block pool, which is the lowest-level pool, allows a block to be released via a pointer to anywhere in the block. The pointer is automatically corrected to point to the beginning of the block, before it is used. Most kernels do not do this. I hate to think what happens when a rarely-used path forgets to correct a block pointer before releasing its block.
- The smx block pool provides even more protection by finding the pool automatically, so the user cannot return a block to the wrong pool. If a seldom-used path releases a small block to a big block pool, some really bad things will happen.

Generally, the solution is to make operations more automatic and more intuitive.

Greater use of subroutines

When smx was first written in 1989, processors were much slower; hence smx had a lot of parallel code to achieve good performance. This led to maintenance problems in that some fixes did not get made to all parallel paths and the parallel paths tended to diverge, over time, such that one developed a bug, but others did not. Modern processors are about two orders of magnitude faster; hence using common subroutines is now the better approach. This leads to greater reliability and, in the case of smx, a 40% code-size reduction. smx code size will continue to be reduced, with multicore support in sight.

Simple and elegant mechanisms

Once I decide upon a set of features that I think are useful, I then try to find the simplest way to implement those features — factoring and refactoring and even redefining the features (something most projects have no time to do). My feeling is that if something is too complicated, it is not right for a kernel — a kernel should be simple and elegant. Unfortunately, given the inherent complexity of kernels, this lofty ideal cannot always be achieved. However, I keep trying.

An illustration of the drive for simplicity is the smx LSR, which has been part of smx from the beginning. Most kernels allow certain services (e.g. semaphore signals) to be called from ISRs. As a result, critical sections of the kernel such as enqueueing and dequeueing must be protected by disabling interrupts. I had a fear that I might forget to protect some obscure critical section of smx code. Hence smx does not permit service

Copyright © 2014 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.

calls from ISRs. These other kernels also have some kind of call-back mechanism, with queuing, to handle resuming a higher priority task made ready by a service call from an ISR. Again, this strikes me as error-prone.

smx has a simpler solution: ISRs can only invoke LSRs and perform a few simple pipe and base functions. LSRs run after all ISRs are done, thus providing a natural deferred interrupt processing mechanism. LSRs run in the order invoked and cannot preempt each other. They can call SSRs, which implement smx services, but they cannot preempt SSRs, nor can they wait. This simple, fool-proof mechanism avoids the need to disable interrupts to protect critical sections of smx. In fact, SSRs, LSRs, and most of the smx scheduler run with interrupts enabled.

Simplicity is also achieved by providing simple services which can be easily combined to solve complex problems. For example, most smx service calls require 2 or fewer parameters. More parameters mean more difficulty in using a service correctly. smx has about 120 carefully chosen services. Some kernels have hundreds. It is difficult to understand and use so many services correctly.

Building in ruggedness

We need not only correct software, but also rugged software — software that stands up to whatever nature throws at it. LSRs provide an example of this. It is possible in many systems that, under a barrage of interrupts, processor capacity may be exceeded. This is the once-in-a-blue-moon scenario that we all fear, since it is so hard to duplicate in order to find the problem causing a system to crash.

Because LSRs offload ISRs of all but minimal processing responsibility, it is probable that ISRs will be able to keep up, but LSR processing will fall behind. Once the ISR burst is over, the processor will process each LSR in order, as fast as it can, with no interference from tasks. A unique feature of LSRs vs. tasks is that the same LSR can be invoked multiple times, each time with a different parameter value. Hence, all processing will get done, and *temporal integrity* will be preserved. In a control system I think this may result in sluggish behavior, but not instability leading to failure. In a data acquisition system, no data will be lost, nor will time-stamps be compromised (because they, too, are handled by LSRs).

Minimizing interrupt latency

I have always felt that minimal interrupt latency is very important for embedded system kernels. Some processors disable all interrupts while an ISR is running. Hence, even high-priority, non-kernel interrupts are impacted. Other processors disable only interrupts at the current level and below, but still some interrupts may be missed. Enabling interrupts part way through a long ISR is not advisable because ISRs are usually non-reentrant. In fact, different ISRs may compete for the same global variables, making this an even worse idea. For this reason, most kernels provide some form of *deferred interrupt processing*.

Copyright © 2014 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.

For smx, LSRs perform this role. As previously explained, due to the LSR mechanism, SSRs, LSRs, and most of the smx scheduler run with interrupts enabled. As a consequence, smx interrupt latency is not impacted by complex service calls and it is very small. For some kernels, interrupt latency depends upon the complexity of service calls — e.g. how many items are in a queue. This is highly undesirable because a high-level operation that is introduced late in a project may cause a low-level operation, such as servicing an interrupt, to randomly start malfunctioning. This is the kind of feedback we do not need!

Another, often overlooked, consideration is how often interrupts are disabled. Obviously if interrupts are seldom disabled, then ISRs are more likely to not be delayed, except by higher priority ISRs. This should create smoother operation and reduce hiccups. The fact that smx seldom disables interrupts helps to achieve this goal.

Providing a good kernel debug experience

Given that over half of development time is spent doing debugging and testing, it only makes sense that the kernel should make this as pleasant and productive as possible. Success in finding bugs depends upon noticing anomalies — little things that grab ones attention as not being quite right. Inability to see these clearly can easily result in going in wrong directions and wasting time. Modern debuggers have greatly improved the debug experience; creating a good kernel debug experience is equally important.

Most commercial kernels offer kernel-aware plug-ins, task profiling, and stack overflow detection, which are helpful, and we have put great effort into making smxAware excel in these areas.

However, I think more attention needs to be given to low-level debugging, such as using enums so that field values have names (e.g. SMX_READY) instead of numbers. Also, defining links as specific control block pointer types. For example, when looking at an smx semaphore control block, clicking on its forward link brings up the full TCB of the first waiting task. Clicking on its forward link brings up the full TCB of the next waiting task, and so on. These kinds of little features provide the better visibility that can lead to quickly fixing bugs. For example, one might notice that the tasks are not in the expected order because one task has the wrong priority.

Proven software

An important advantage that a commercial kernel offers over in-house software is that it is in use in many other projects. A project is likely to not use some paths in a given body of software. This becomes a problem when a new feature exercises untested paths — naturally one thinks that the new feature is at fault. Use in multiple projects makes this unlikely to happen with commercial software.

Obviously, generic kernels offer less proven code — another disadvantage for them.

Copyright © 2014 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.

Good manuals and good examples are crucial to success

Good manuals and examples are a final piece to the puzzle. Misuse of kernel features can lead to serious problems. Yet few of us have the time (or the patience) to do the reading we should. One normally reads the manual only when nothing else works. I think one solution to this problem is to provide numerous good examples that actually run. These can be cut and pasted into application code and modified to fit. smx comes with a library of such examples called *esmx*.

esmx can be plugged into any smx evaluation kit or shipment. One can look through the examples, pick one of interest, set a breakpoint, start the system running, then step through the example. An "esmx Debug Tour" is provided to show how to do this effectively. I think stepping through examples may be one of the best ways to learn a kernel, because one can see what actually happens when various services run. Actual task sequences can frequently be surprising, and misconceptions can be quickly cleared up. There is nothing like reality to learn a lesson!

The smx User's Guide presents smx subject by subject with few back or forward references. It starts with Under the Hood and Introductions to Objects and Tasks to help the reader get oriented; from there on, you read as you need. There is a major section on development, which covers Structure, Method, Coding, and Debugging. This presents a system that I call "skeletal development," which is basically a top-down design approach using tasks and kernel services. The essence of this approach is to define threads (ISRs, LSRs, and tasks) and to get their interactions right before filling in operational code.

Operational code is emulated with delays (and possibly visual effects). The skeleton runs from the very beginning and as more and more threads and services are added, it approximates the final system behavior more and more closely. Thus, costly errors can be eliminated before much code is written. This approach is good for both managers and programmers. Seeing the system running at an early stage builds confidence of both.

Slots for operational code are bounded by time allocations (delays) and kernel services. Hence, operational code programmers are given boundaries for their creations. If they stay within the boundaries, all should be well. Of course, as code becomes available, it can be popped in. I think this method will result in better use of kernel resources, less time spent reworking operational code, and fewer integration problems. It thus permits completing the project more quickly and producing a better final product.

The smx Reference Manual provides brief descriptions concerning how to use each service correctly and it includes an extensive glossary to help understand unfamiliar terms.

Summary

This paper explains what factors have influenced the recent evolution of smx and discusses their relative importance. Designing a good RTOS kernel requires achieving a good balance, which I think I have done with smx. However, the evolution continues.

Copyright © 2014 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.