**smx**<sup>®</sup>

## Simple Multitasking Executive

*smx is a hard-real-time multitasking kernel for embedded systems. It offers many features to permit using slower processors and less memory, thus reducing product cost. This is becoming increasingly important as the embedded industry shifts to SoC solutions and more consumer-oriented products.*

*smx* is delivered in pre-compiled library form with startup platforms. Source code is available separately. *smx* offers full support for the most popular C/C++ compilers and assemblers, and it is one of only a few kernels to provide a C++ class library (see the *smx++* brochure). *smx* is ROM'able for standalone environments. It supports several different processor types (see processor-specific *smx* brochures) and it provides many features to facilitate development for each processor type.

*smx* has been commercially available for over 12 years and has been used in hundreds of applications. During this time, it has steadily evolved into a better and better kernel. Today, *smx* is reliable, robust, and highly capable. It is an optimum choice for demanding, trouble-free use.

### The smx Concept

The *smx* concept is to keep things simple, but, at the same time, to offer a richness of features from which the application programmer can draw. Many RTOS kernels are too simple. While this may seem to make them easy to learn, the net

### FEATURES

- Unique, 3-level application structure:
  - background tasks
  - foreground tasks (lstr's)
  - isr's
- Almost no interrupt latency
- Fast task switching
- Preemptive, round-robin, and timeslice scheduling
- Multiple tasks per priority level
- Dynamic priorities
- Hooking of custom task entry and exit routines into scheduler
- Scheduler locking
- Small memory footprint
- Dynamic objects
- Protected heaps
- Fast and standard block pools
- Better stack sharing model than OSEK
- Indirect messaging via exchanges
- Message priorities and priority passing
- Client/Server messaging
- Pipes and buckets for easy I/O integration
- C++ support
- Timeouts on all task waits
- Cyclic and one-shot timers directly launch foreground tasks for tight timing control.
- Good selection of third-party tools supported
- Task-aware debugging
- Used in hundreds of applications

result is that the application ends up being more complex than it should be. It is fundamental that *what is not in the kernel ends up in the application*.

Unused *smx* functionality is not linked with the application. Hence, *smx*'s richness of features is not a memory burden, but rather, allows the application programmer to create a kernel customized to his needs.

Many RTOS kernels are exceedingly complex and difficult to use correctly. *smx* has aimed for the middle ground and has successfully achieved a good balance between simplicity and richness of features.

*smx* packs a great deal of capability into a relatively simple, easy-to-use API. Some of the main features of *smx* are listed in the sidebar. These are described in more detail below. The descriptions are grouped into the subjects of most importance to embedded system programmers:

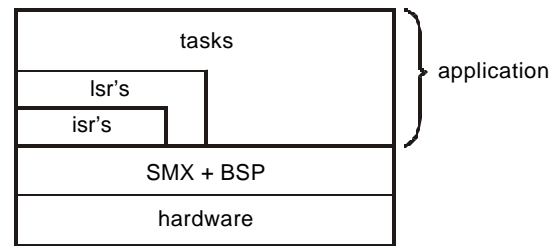
- Performance
- Memory Footprint
- Safety
- Ease of Use
- API

## Performance Features

### Foreground Tasks (lsr's)

As shown in the diagram, *smx* supports a three-level structure for application code. The levels are:

- (1) Normal Tasks (which run in the background)
- (2) Foreground Tasks, also called *link service routines* (lsr's)
- (3) Interrupt Service Routines (isr's)



*Link service routines* are so named because they link the interrupt-driven foreground to the task-implemented background. lsr's are unique to *smx*. A few other kernels and RTOS's (e.g. Nucleus and WinCE) have similar-appearing objects, but they do not work quite the same.

lsr's provide a natural way to write foreground (i.e. interrupt-driven) code. The *primary* interrupt service code (the isr) is kept short and sweet. Experts agree that isr's should be kept as short as possible in real-time systems, because this minimizes interrupt latency. The *secondary* interrupt service code is deferred, in the form of an lsr, until all interrupts have received primary service. lsr's do not disable interrupts, so they, themselves, can be interrupted to provide primary (isr) service to other interrupts or even to the *same* interrupt.

As noted, all interrupt service code that is not essential to the isr (i.e. secondary code) is moved into an associated lsr. The lsr is *invoked* from the isr, as needed (e.g. when all characters of a message have been received). isr's operate with interrupts *disabled* because isr's are non-reentrant. lsr's operate with interrupts *enabled* because each lsr invocation runs separately, in its turn. Hence, the lsr code need not be reentrant.

Like isr's, lsr's operate in the context of the current task. This means that they use its stack and registers. Because there is no stack switching, with its associated pushing and popping of registers, lsr switching time is negligible. Unlike isr's, lsr's can make *smx* calls. Hence an lsr can awaken a task, send a message, signal a semaphore, etc. Thus lsr's are, in effect, *foreground tasks*. Other properties of lsr's are discussed in appropriate sections that follow.

***smx* is not “like every other kernel.” It offers many unique features that are important for high-performance, low-cost embedded applications.**

## Scheduler Features

The scheduler is the most important part of any kernel. It determines whether or not to run a new task. If so, it *suspends* or *stops* the current task and *resumes* or *starts* the new task. The *smx* scheduler is a *preemptive* scheduler. It runs the longest-waiting task at the highest occupied priority level.

**Preemptive Scheduling** is the best algorithm for embedded systems. Despite this, many well-known RTOS's (e.g. WinCE, embedded NT, Linux, and PharLap ETS) utilize *priority time slicing*. Under this algorithm, when a higher priority task becomes ready to run, it must wait until the end of the current time slice to be dispatched. Hence response time is governed by the granularity of the time slice. However, if the granularity is set too fine, the processor spends too much time thrashing — i.e. interrupting the current task to find out if a higher priority task is waiting. This pretty much precludes hard real time response without using an over-kill processor.

*smx* uses preemptive scheduling. This means that, as soon as a higher priority task becomes ready to run, it preempts the current task and runs. For safety, *smx* does permit the current task to *lock* the scheduler if necessary (i.e. if it is in a *critical section* of code.) The programmer controls locking. This is discussed more, below.

**Priority Levels:** Some kernels (e.g. uC/OS) require each task to have a unique priority. This is limiting, because, within a group of equally important tasks, it is usually better for the task that has waited the longest, to run first. One task per priority level does not permit this. Allowing multiple tasks at the same priority level also permits round robin scheduling among those tasks. This is a good way to share resources equally among the lowest priority tasks in the

system. *smx* also permits time slicing among the lowest priority tasks — this is even more equal.

**Scheduler Locking:** *smx* allows the current task to lock the scheduler. Many kernels do not provide this feature. Why is it important? With the addition of locking, there are three ways to protect access to a resource: (1) disabling interrupts, (2) semaphores, and (3) locking.

The first method is the only way to protect a resource shared between isr's and other isr's, lsr's, or tasks. However, it causes interrupt latency and *should be used as little as possible*. Semaphores are the traditional method to protect resources shared between tasks. They can also be used by lsr's. Semaphores are resource-specific and do not add interrupt latency. However, they do cost a fair bit of processor overhead — typically on the order of 100 instructions to signal and test a semaphore. Hence using semaphores is inefficient for short, critical sections of code<sup>1</sup>.

This is where locking has an advantage. Locking + unlocking requires only about 10 instructions, if no preemption results, and about 50, if preemption does result. Obviously, processor overhead is less. Also, task latency<sup>2</sup> is less. Lock and unlock are very short operations, so if the critical section is short, the sum of these plus the critical section of code will be less than the code in the signal and test functions. (Scheduling cannot occur while *smx* functions execute.) Note that locking can be used between tasks, only.

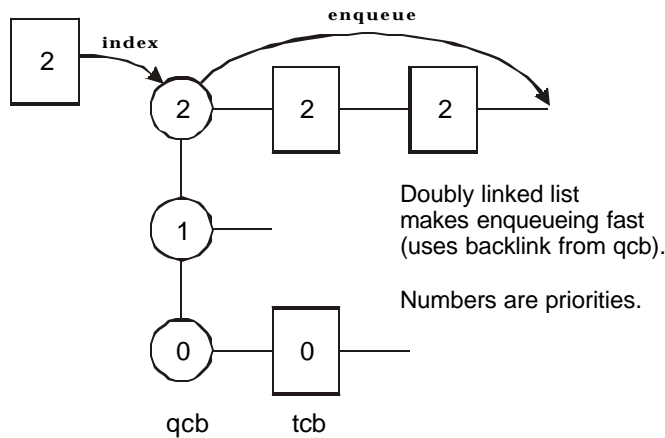
## Layered Ready Queue

*smx* is unique in having a *layered ready queue*. The ready queue is where ready-to-run tasks are enqueued. Nearly all kernels, other than *smx*, utilize a single, ordered ready queue. To enqueue a new task requires searching from the beginning of the queue until the last enqueued task, of equal priority, is found. Then the new task is enqueued

<sup>1</sup> A *critical section* is code that accesses a non-sharable resource or code that is non-reentrant.

<sup>2</sup> *Task latency* is the time that a task switch is prevented due to the scheduler not being allowed to run.

at that point. Obviously, if there are many ready tasks, this could take a long time. In most kernels interrupts must be disabled the whole time! Making this algorithm even worse is that low priority tasks take the longest to enqueue!



For *smx*, we observed that since *smx* tasks are permitted to share priority levels (which is not true for some kernels) the typical embedded system needs no more than 5 to 10 priority levels.<sup>3</sup> (How finely can you slice and dice the relative importance of each task?) Hence, why not have a separate queue for each priority level? This is how the *smx* ready queue is implemented. Each level is headed by a *queue control block* (qcb). The qcb's are contiguous in memory and in order by priority. Hence enqueueing a task is but a two-step process: (1) index to the correct qcb, based upon the priority of the new task, and (2) follow the backward link of the qcb to the end of the queue and link in the task. This fast, two-step process takes the same amount of time regardless of how many tasks are in the ready queue. There could be 10 or there could be 10,000. It would not make any difference!

Finding the next task to dispatch is also very fast because *smx* maintains an *rq\_top* pointer. *rq\_top* points at the tcb of the next task to run.

<sup>3</sup> However, *smx* does support up to 128 priority levels, if needed.

## Low Interrupt Latency

Isr's, which have been previously described, have a very beneficial effect upon *interrupt latency*. Interrupt latency is defined as the time from when an interrupt occurs until its isr starts running. This is possibly the most important single parameter in hard real time embedded systems. It is the sum of three terms:

$$\text{interrupt latency} = \text{processor latency} + \text{kernel latency} + \text{application latency}$$

The application programmer has control over only the last term. Kernel latency consists of the time that the kernel disables interrupts. All kernels, except *smx*, do this in order to protect internal kernel structures (e.g. the ready queue) while they are being changed. Because *smx* calls are not permitted from isr's and because any pending *smx* call must finish before an isr can run, it is not necessary to disable interrupts in *smx* calls, nor in 98% of the scheduler! In fact *smx* disables interrupts only to assure that isr's run without needless delays. As a consequence, worst-case *smx* interrupt latency is almost negligible — about the same as the processor, itself.

## Efficient isr's

*smx* also adds very little overhead to isr's. It requires adding an *enter\_isr* macro and an *exit\_isr* macro to each isr. The *enter\_isr* macro saves a few registers that the isr would probably save, anyway, and it increments the *srnest* counter. The *exit\_isr* macro decrements the *srnest* counter, tests whether to branch to the isr scheduler, and, if not, restores the saved registers and returns to the current task. That's it. For high priority isr's, that need not be linked to *smx*, even the small overhead of these macros may be omitted.

It is important to note that an isr operates using the current stack. Changing stacks would add overhead to isr's. This is the case, for example, with DOS and DOS extenders.

*smx* permits nested isr's. Some RTOS's (e.g. WinCE) do not. Nesting isr's is important for high performance, because without it, interrupt latency for high priority isr's can be considerably lengthened (since they need to wait for low priority isr's to finish).

***smx* was architected to encourage applications with large numbers of tasks. More tasks means more use of *smx* code and less use of unproven application code. This translates into faster project completion.**

## Small Memory Footprint

Although some memory devices have become very large and have achieved very low cost per bit (as proponents of inefficient RTOS's like to point out) other memory devices used in embedded systems have not. In particular, the move to SoC solutions is bringing back the need for highly efficient memory utilization because on-chip memory is much cheaper than off-chip memory. Whether implemented as an ASIC or an FPGA, the smaller the memory, the smaller the chip and the lower the cost. This is important in low-cost embedded products that may be produced in very large quantities. For such products, the company that chooses a sensible RTOS over a bloated RTOS will have the competitive advantage.

Efficient memory management is another area where *smx* excels. *smx* features which support this are discussed in the sections that follow.

### Stacks

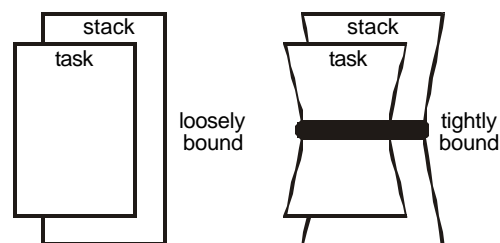
Ordinarily, every task requires its own stack. For performance reasons, isr's use the current task's stack. Hence, each task's stack must allow for the maximum isr nesting as well as the maximum subroutine nesting for the task, including auto variable space (which can be large for some routines). Although *smx* does allow stack size to be customized to each task, still the sum of RAM space required by all task stacks can be very large. In fact, it normally dominates other kernel RAM requirements.

In memory-constrained designs this can result in the unfortunate tradeoff of using fewer tasks than is optimum for the application. When this happens, much of the benefit of multitasking is

lost. This is because operations which could be handled by kernel calls become internal operations within tasks and therefore application code must be created to perform them. *smx* provides three solutions for this problem:

**Shared stacks:** *smx* incorporates a stack sharing feature which is similar to the OSEK BCC1 *one-shot execution model*. The latter allows the sharing of 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 are 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 is less restrictive, and it is easier (and less error-prone) to calculate the stack space required. Under *smx*, stacks may be either *tightly bound* or *loosely bound* to tasks. When a task is *stopped* it surrenders a loosely bound stack, but it keeps a tightly bound stack. When a task is *suspended*, it does not surrender its stack, regardless of how the stack is bound. Tasks may be either stopped or suspended when waiting for messages, signals, events, timeouts, etc. Hence, stopping vs. suspending a task introduces no constraints on what the task can do.



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 normal tasks, with tightly-bound stacks, and one-shot tasks, with loosely-bound 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.

Note that one-shot *smx* tasks can wait at semaphores, exchanges, etc. when stopped. OSEK one-shot tasks cannot do this. Also, to add normal tasks, OSEK requires changing to the ECC1 or 2 model. Neither of these permit stack sharing. With *smx*, there is only one model and there can be any number of either kind of task. As a result of these two differences, *smx* permits more optimal RAM vs functionality tradeoffs to be made, than OSEK.

**lsr's:** Functions, which might be performed by small tasks, can be performed by lsr's, instead. Examples would be functions invoked by timers and isr's. This is advantageous because lsr's do not require their own stacks.

**Dynamic tasks:** *smx* permits all objects, including tasks to be dynamically created and deleted. A stack is allocated from the *smx* heap when a task is created and freed to the heap when the task is deleted. Task creation is a fast operation, as compared to forking new processes in UNIX-like RTOS's. Hence deleting unneeded tasks to reduce memory required for stacks may be entirely acceptable in many designs.

## Heaps

As well as providing traditional block pools and fast block pools, *smx* provides heaps. Heaps are

normally frowned upon for embedded, real-time systems because block allocation is non-deterministic and can be slow as the heap becomes more fragmented. Also, severe heap fragmentation can lead to allocation failure and, potentially, to system failure. On the other hand, block pools waste memory because blocks often must be larger than necessary for most uses. Also, to make sure that the system does not run out of a given size block, more blocks must be allocated than are actually needed, most of the time. By offering both heaps and block pools, *smx* allows the designer to choose an optimum combination of methods for his application.

Quite a bit of memory is permanently allocated at startup, in embedded systems. Heaps are ideal for this purpose because each block can be exactly the right size. Also, fragmentation is not a problem because these blocks are never released. *smx* always starts searching at the first free block. Hence it does not waste time on blocks permanently allocated at startup.

## Messages

An *smx* message can be of the exact size required. Messages, like blocks, may be allocated from pools or from the *smx* heap. Heap messages are advantageous in situations where messages are normally small, but occasionally can be large. Rather than creating a message pool of large messages, which would waste memory, the occasional large message can be allocated from the heap and the thus messages in the message pool can be much smaller.

## Other Memory Efficiencies

*smx* object control blocks use memory as efficiently as possible. Nearly all *smx* objects are dynamic, meaning that they can be deleted, when not needed, thus reducing needed resources. *smx* requires very few global variables.

## Code Efficiency

Each *smx* function is in a separate module and interdependency between functions is minimal.

Since *smx* is linked directly with the application, only those functions actually used will be linked in. (This contrasts to kernels like those for Linux and QNX which are free-standing and hence must have all functions present.) For CISC processors, *smx* code footprint varies from about 8 to 30KB; for RISC processors, it is about 15 to 55KB.

## Safety

Concern about safety has come to be of equal importance to performance and size. Faced with greater system complexity and shorter development schedules, it is no longer a question of whether there will be bugs in delivered equipment. *This can be taken for granted.* Now it is a question of how many bugs, what consequences they will have, and how quickly they can be found and removed. A big question is: how safe is the operating system? Will a common programming error cause it to lock up? Will a small mistake or an unexpected event cause a task to cease running? Does the OS help the programmer to find errors like these? Is the OS designed such that it behaves in a reasonable manner or is it loaded with gotcha's?

As obvious as the right answers to these questions are, producing an OS with those right answers is not a simple exercise. As always in OS design, there are tradeoffs with performance and size. Code executes linearly — the more code there is, the longer it takes to execute. The more error checking performed, the larger the code. Processors may be faster and memory may be cheaper, but careful judgment is still necessary to decide what code to add and what code to omit. *smx* achieves a good balance between speed, size, and safety.

## Error Detection and Management

**Error Detection:** The *smx* error manager monitors over 50 error types. These include out-of-range parameters, broken queues, heap overflows, stack overflows, invalid control blocks, null pointer references, resource

exhaustion, and others. When an error occurs a unique *error service routine* (esr) is automatically invoked via a jump table. The esr records the error type, the time of occurrence, and the task or lsr i.d. in the *error buffer* (eb). It also increments a unique counter for the error type and loads the error i.d. into the *errnum* field in the task's TCB. esr's are user-extensible and can perform unique functions for specific error types.

**Error Buffer:** The error buffer (eb) is allocated from the *smx* heap. Its size is user-controlled and can be small or very large (up to thousands of records for 16-bit modes; unlimited for 32-bit modes). eb is viewed as an array of records under a debugger. It may be viewed symbolically with *smxProbe*. (See the *smxProbe* brochure for details.)

**Error Management:** *smx* supports both *point-of-call* and *centralized* error management. All *smx* calls return 0 if an error or a timeout has occurred. Hence point-of-call error management can be implemented as follows:

```
if (smx_call( ) )
    /* do normal action */
else
    /* do error or timeout action */
```

The error action code can access *ct->errnum* to determine what action to take. (*ct->errnum==0* implies a timeout.)

Centralized error management is performed by esr's, as discussed above. Rather than writing point-of-call error handling code, the user may customize esr's to deal with errors *centrally*. Although errors cannot be dealt with as effectively this way, the amount of code required is considerably less. The error manager is omitted from the non-test mode libraries to permit smaller and faster-executing code<sup>4</sup>. The error manager source code is included in the *smx* development kit. It can easily be extended for application error handling.

---

<sup>4</sup> This is not recommended, except in systems with very tight memory space.

## Task Timeouts

Any *smx* call, which causes a task to wait, can also start a timeout. Timeouts are for safety, not for timing. They are to insure that tasks do not wait indefinitely for an event. Hence, each task has a timeout register which records the time at which it should be resumed or restarted. All non-zero timeout registers, are periodically compared to the elapsed time (et) counter. If less than et, the corresponding task is resumed. This function is performed by the *timeout task*, which can be set to run at an appropriate priority. Hence the timeout mechanism is designed to create minimal processor overhead, which is appropriate for a safety feature.

## Safety Valves

Isr's can operate like *safety valves* and many *smx* applications have used them in this manner. If it is not feasible to use a fast enough processor to handle peak loads, Isr's can be used to smooth the peaks out. This is due to a unique Isr feature: unlike a task or an isr, an Isr can safely be invoked multiple times before running. Hence, an Isr can be interrupted again and again by isr's which invoke it or other Isr's. Each invocation can be passed a unique parameter such as a timestamp, a character, a buffer pointer, etc. which serves to make the invocation unique. (Note: This is not required.) When the peak load has passed, each invocation of the Isr executes, in order. Hence, Isr's preserve *temporal integrity* — i.e. events do not get out of order.

Despite the academic obsession with *deadlines* and *rate monotonic analyses*, temporal integrity is usually sufficient in real systems. The system will continue operating smoothly and not get out of whack. In fact, the natural filtering of peaks caused by Isr's may be just what the doctor ordered!

## Unblockable Isr's

Unlike tasks, which can be blocked by higher priority tasks or locked out by lower priority tasks, Isr's are not subject to delay by any tasks.

They are also immune to priority inversion because they cannot wait on events. Isr's are simple creatures which, except for interruption by isr's, are undeterred in doing their jobs. This lack of sophistication is often exactly what is needed for safety-critical and time-critical functions.

## Safer Messaging

Most kernels offer only rudimentary messaging in which pointers to blocks are passed via *message queues*. A message queue is similar to an *smx pipe*, except that it is wide enough to accommodate a pointer. The way this works is that a sending task gets a block from a block pool, puts a message into it, and loads a pointer to the block into a message queue. Sometime later, a receiving task gets the pointer from the message queue and processes the message. When done, it returns the block to its block pool, using the pointer it received.

This approach is inherently unsafe because there is no way for the receiving task to verify that it has received a correct pointer. The sending task may have altered the pointer such that it does not point to the beginning of the message, as it should, or the pointer may have been garbled in transmission. Hence, not only would wrong information be processed, but the block would not be correctly returned to its pool.

*smx* solves this problem by associating each message with a *message control block* (mcb). Message handles are passed from task to task via *exchanges*. (Exchanges enqueue mcb's and tcb's.) A message handle is a pointer to a message's mcb. This pointer is verifiable because it must point to a known structure with a known i.d. in a certain area of memory — namely an mcb. *smx* automatically verifies handles before passing them to receiving tasks.

Protected within the mcb is the actual message pointer. Neither the sending task nor the receiving task has any reason to alter this pointer. Both work with their own copies of it. Hence, it is very unlikely that a receiving task will process wrong data. Furthermore return of the message to its

pool is governed by the verifiable handle, not by the message pointer.

A messaging error undermines one of the goals of multitasking: that tasks be independent. Clearly, if a task's errors can propagate easily to other tasks, then the process of finding bugs becomes more difficult. In this case, a seldom used path in a sending task could produce an errant pointer that would rapidly bring down the whole system. Many programmers feel that C pointers are risky. Certainly, passing raw pointers from task to task is even more risky!

## Robust Heaps

*smx* heaps are designed to be more robust than those supplied with C compilers. Allocated blocks are separated by *heap control blocks* (*hcb*'s) rather than just next block pointers. An *hcb* contains a pointer to the next *hcb*, an *onr* field, and *fences* at the beginning and end. The *onr* field identifies which task owns the block. This is used to prevent memory leaks when tasks are deleted. (All owned blocks are automatically freed.) The fences are fixed bit patterns which are used to detect heap block overflows. They are checked whenever a heap block is freed and also may be periodically checked with *smx* *heapchk()* or *heapwalk()* calls. *smx* heap calls are *task safe*, which is not true of most compiler heap calls.

## Ease of Use

Ease of use ranks high with embedded system programmers. This is a direct result of increased complexity and increased time-to-market pressure. *smx* has many features to make the programmer's job easier.

## Natural API

A *natural* API is easier to learn and to use correctly. What is required for a natural API? Two important elements are *symmetry* and

*orthogonality*. These were primary considerations in the *smx* architecture. Relative to RTOS architecture, symmetry means that what can be done can be undone. For example, *smx* offers a delete function for nearly every create function; it offers a start function for every stop function, a resume function for every suspend function, etc. Orthogonality, relative to RTOS architecture, means that every *smx* function affecting a task can be performed in any task state. For example, an *smx* task may be deleted in any state. This sounds simple enough, but if a task is deleted while it is in the run state, it is actually deleting itself — not an easy thing to do! Interestingly enough, a task deleting itself actually turns out to be useful — thus emphasizing the importance of orthogonality.

What else? A small number of parameters per function is important. Some API's have a too many parameters per function. This makes it difficult to use a function properly. It is better to have two simpler functions. This has been done with *smx*. Most kernel API's are not as carefully crafted as *smx*.

**What is not in the kernel ends up in the application.**

## Powerful Messaging

Messaging is one of *smx*'s strongest points and goes well beyond that typically found in other kernels. *smx* message facilities are powerful and flexible. Messages may be any size and are passed by reference, *not by copying*. Message passing by copying is the only mode supported by many RTOS's (e.g. QNX). It is obviously much slower than passing by reference — especially for long messages.

**Exchanges:** *smx* messages are sent to and received from *exchanges*. An exchange is an *smx* object, which can enqueue any number of either messages or tasks, but not both at the same time. The use of exchanges has important advantages over the direct task-to-task messaging used by some kernels: (1) Anonymity — The sending

task need not know the identity of the receiving task. It simply sends to a known exchange. This makes it easy to swap receiving tasks without altering the sending task. Task *swapping* can be useful to handle different product versions or different installation configurations. Task swapping can even be done dynamically to handle changing circumstances, on the fly (e.g. changing from operating mode to test mode.)

(2) No limit on the number of waiting messages.  
(3) A message queue at an exchange creates a natural work queue for a server task that is receiving messages from the exchange. (4) A message can be used as a token to control access to a resource and to simultaneously provide needed information about that resource (e.g. the i/o port numbers needed to access it).

**Priorities:** *smx* messages have priorities. Exchanges can bypass high-priority messages around low-priority messages. A common example of this is to bypass an abort message around data messages. Another example is to allow more urgent work to be completed by a server task ahead of less urgent work. *smx* goes a step beyond this by also providing *priority pass exchanges*. A priority pass exchange changes the priority of the receiving task to that of the message. Hence a client task can assure that a server task will operate at the client's priority. It does so by assigning its priority to the messages it sends to the server's priority pass exchange. This is especially useful for server tasks that are managing resources. Being able to adjust their priorities permits them to operate as extensions of their client tasks. (This important, because using server tasks avoids the problem of *priority inversion* inherent in allowing client tasks to directly access resources.)

## Replies and Owners

As compared to the rudimentary messaging offered by most kernels (see Safer Messaging, above), *smx* messages have priorities, as discussed above, and also *reply* and *owner* attributes. The reply attribute tells a receiving task where to send its reply. This is important for most client/server implementations, since there

may be many possible clients. The owner attribute allows a message to be automatically freed if the task that owns it is deleted. This is important in dynamic task implementations such as may occur with C++.

## Easy I/O Integration

It is very difficult to create device drivers for some RTOS's (e.g. Embedded NT and Linux). New device drivers must be interfaced with the RTOS in a complex, poorly-documented manner. Then, testing is not well supported. As a consequence, creating device drivers for these OS's is programmer intensive.

In contrast with this, the only impact that *smx* has upon device drivers is that most isr's must be bracketed with *enter\_isr* and *exit\_isr* macros. Also, *smx* calls are not permitted within isr's. Instead, isr's must invoke lsr's which then make the *smx* calls. Other than this, *smx* imposes no constraints upon device driver implementation.

In addition, *smx* offers two objects to make it easier to interface device drivers to tasks. These are called *pipes* and *buckets*.

**Pipes:** It is customary to create *ring buffers* to hold incoming and outgoing character streams. The question, then, is how to interface these ring buffers with tasks — this is not simple because there can be access conflicts as well as other problems. *smx* provides an easy solution for this, called the *pipe*. A pipe is basically a ring buffer. It can be of any length. High-speed macros are available for isr's, lsr's, and tasks to put characters into and get characters out of pipes. In addition, *smx* calls are available to do the same thing, but they allow tasks to wait on full outgoing pipes or empty incoming pipes. Hence, pipes serve as a character conduit between serial device drivers and tasks.

**Buckets:** Ring buffers, or pipes, are not a good solution for high-speed pocket-sized communication because they necessitate an extra copy step per character. *smx* provides a unique object, called a *bucket*, to solve this problem. Incoming characters are loaded into a bucket, just

like a ring buffer. *smx* provides a macro for this. When a complete packet of characters, has been received (as determined by the receiving isr), an lsr is invoked to move the packet, by reference (i.e. no copy), from the bucket to a normal exchange where it can be processed as an *smx* message. Going the other direction, a packet is moved, by an lsr or a task, from a normal exchange to a bucket, as needed or as available. The bucket is emptied by a transmit isr using an *smx* macro. When empty, the spent packet is recycled.

## Versatile Timing

*smx* implements versatile timers. An *smx* timer is a dynamic object. When *started*, it is created and enqueued in the *timer queue(tq)*, for the number of ticks specified. When a timer times out, it invokes an lsr with a parameter. (These are specified when the timer is started.) If the timer is a *one-shot* timer, it self-destructs. If the timer is a *cyclic* timer, it immediately requeues itself in the timer queue, with no loss of ticks. (This is important to assure precise timing for taking samples, etc.) The fact that a timer directly invokes an lsr assures tight timing control because lsr's run ahead of all tasks. The lsr can do the work or it can start a task to do the work.

It is worth noting that timers are enqueued in tq by their differential time delay. As a consequence, only the first timer is decremented on each tick, until it times out. Then the next timer is decremented, etc. This results in minimal overhead per tick. Also, the overhead is the same regardless of how many timers are running.

## Scheduler Hooks

*smx* allows *exit routines* and *entry routines* to be hooked into the suspend and resume processes of the task scheduler. The hooked exit and entry routines are task-specific. This allows extending the context saved when a specific task is suspended and restoring it when that task is resumed. This is done in a manner that is transparent to the task. Examples of where this capability is important are: coprocessor registers,

memory bank registers, and other task-specific hardware. There is no penalty on tasks that do not have hooks--i.e. hooking is task-specific. *smx* is one of a very small number of kernels that offers scheduler hooks.

## Flexible Event Handling

**Counting Semaphores with Thresholds:** The *smx* semaphore is a normal counting semaphore with the addition of a threshold. When the threshold is 1, it operates like a normal counting semaphore — i.e. it will allow N users to access N resources if N is its initial value. Alternatively, with a threshold of 1, the semaphore might be used to cause a task to run once per event. Events will be stored when the task is too busy to handle them. When the threshold is a different value, T, the *smx* semaphore can be used to require T events before resuming a task. This is useful when T events must occur before a task can run. These could be either T occurrences of the same event or one occurrence, each, of T events.

**Event Queues:** Event queues are similar to semaphores. They also accept event signals. However, they allow multiple tasks to wait for various numbers of events. If no task is waiting, events are not saved.

**Event Tables:** Event tables allow tasks to wait for AND or OR combinations of multiple events.

## Protosystem

A single, integrated platform is provided for all versions of *smx*. This platform is called the *Protosystem*. It is a skeletal *smx* application, to which demo code and tasks can be added for other SMX<sup>5</sup> products (e.g. *smxFile*, *smxNet*, etc.). Full source code is provided for the Protosystem and for each demo. Demos provide helpful examples and useful code for the final application.

A single make file, or project file, serves to build the Protosystem for a particular combination of products. The combined Protosystem and demos

---

<sup>5</sup> "SMX" is the RTOS built upon the *smx* kernel.

show not only that the SMX products work together, but also how they work together.

To develop an application, it is necessary to exclude the demo modules and to add new modules for the application.

The Protosystem shows how to configure and use *smx*. It supplies startup code and drivers needed by *smx*. Depending upon the processor supported, it may also include other BSP code.

## Helpful Manuals

*smx* manuals are well written, complete, and accurate. And they are easy to use. The manuals are a convenient size (7" x 9"), which fits nicely alongside a keyboard, and they are spiral bound so they will lie flat or can even be folded backward. The manuals, shipped with *smx*, are as follows:

**Quick Start:** Introductory material and cookbook-style instructions (i.e. 1, 2, 3...) on the left page with sample code on the right page.

Topics are divided into the subject areas: "Installation and Getting Started," "Application Development," and "Writing *smx* Code." Topics are covered such as "What You Will Need," "Beginning a System," "Creating Tasks," and "Sending and Receiving Messages."

**User's Guide:** Tutorial manual with a full index. This manual presents the theory of *smx* in two main sections: "How To Use *smx*" and "Advanced Subjects." It covers topics such as "Getting Started," "Structuring Your System," and "Objects." We have received many compliments on this clearly written manual.

**Reference Manual:** Combines the standard reference manual format with a comprehensive, detailed glossary. Together, these two sections define *smx* 100%. Each *smx* call description explains all details of the call and has one or more relevant examples. Each glossary entry is a full

description of the topic — in some cases a page, or more in length.

We believe that having good manuals is one of the most important things which sets a commercial kernel apart from an in-house kernel. To see the *smx* manuals, visit our website at [www.smxinfo.com](http://www.smxinfo.com).

## Integration with Third Party Tools

Unfortunately, the integration of tools and utilities can be a major part of the application developer's job. *smx* helps to reduce this task by offering a broad array of supported tools, which are "glued" together by *smx* Protosystems. *smx* offers full support for the most popular C/C++ compilers and assemblers. Among those supported are:

- Borland
- DiabSDS
- CAD-UL
- Embedded Power
- MetroWerks
- Metaware
- Microsoft
- Paradigm
- U.S. Software

The *smxAware* product provides *smx* awareness for most of the above debuggers. (See the *smxAware* brochure.)

## API

*smx* is a full-featured kernel, providing over 100 function calls and macros. *smx* calls are written in C and are invoked like ordinary C functions. *smx* control blocks, buffers, and other objects are defined as C structures, C arrays, C pointers, etc. *smx* is fully compatible with ANSI C and C++.

As well as providing a carefully crafted C API, which follows, *smx* also offers a C++ API (see the *smx++* brochure).

## Task Management

*Task management is the process of creating and deleting tasks; starting and stopping them; and otherwise directly controlling them.*

**bump\_task** (task, new\_priority)

Changes task priority and requeues *task*.

**copro** (ON/OFF, task)

Enables/Disables coprocessor state saving for *task*.

**create\_task** (code\_ptr, priority, stack\_size)

Creates a task with *task\_main* code with the specified *priority*, and binds a stack from the heap of size *stack\_size* to it. If *stack\_size* == 0, no stack is bound.

**delete\_task** (task)

Deletes *task* and releases resources owned by it (including permanently bound stack).

**hook** (entry, exit, task)

Hooks exit and entry routines to *task* which run when *task* is suspended/resumed.

**IS\_LOCKED** (task)

Determines if *task* is locked.

**locate** (task)

Locates the queue which a task is in.

**LOCK** (task)

Locks *task* against preemption.

**resume** (task)

Dequeues *task* from any queue it may be in and puts it into the ready queue at the end of its priority level.

**STACK\_CHECK** (ON/OFF, task)

Enables/Disables stack checking for *task*.

**start** (task)

Puts *task* into the ready queue at the end of its priority level. Stops it first, if necessary.

**start\_new** (task, code\_ptr, priority)

Restarts a task with new code and priority. Clears HOOKED flag.

**start\_par** (task, par)

Starts *task* and passes *par* to it.

**stop** (task, timeout)

Dequeues *task*, releases its stack if not a permanently bound stack, and sets its timer to restart after *timeout*.

**suspend** (task, timeout)

Dequeues *task* and sets its timer to resume after *timeout*.

**unhook** (task)

Unhooks exit and entry routines from task.

**unlockx** (task)

**UNLOCK** (task)

Unlocks *task* so it can be preempted by higher priority tasks.

## Memory Management

*Memory management permits control over a heap and one or more dynamically allocated regions (dar's).*

**calloc** (num, size)

Allocates space for an array of *num* elements of *size* bytes from the heap.

**create\_dpool** (rxchg, num\_blks, blk\_size, darp)

Creates a block pool or a message pool in a dynamically allocated region (dar).

**create\_fbpool** (mem\_ptr, num, size)

Creates a fast block pool of *num* blocks of *size* bytes at *\*mem\_ptr* in the heap. Blocks are linked to *mem\_ptr*.

**create\_hpool** (num\_blks, blk\_size)

Creates a block pool in the heap.

**delete\_hpool** (pool)

Deletes any block pool or message pool that was allocated from the heap.

**find\_next** (last, task)

Finds the next block or message in a pool which is owned by *task*. Starts looking at *last* block.

**find\_pool** (blk\_or\_msg)

Finds the dar or heap pool to which *blk\_or\_msg* belongs.

**freex** (block)

Frees a block previously allocated from the heap.

**get\_block** (pool)

Gets a block from a dar or heap pool.

**get\_fblocks** (flp, num, size)

Gets *num* physically adjacent blocks from fast block pool linked to *flp*.

**heapchkx** ()

Checks heap integrity.

**heapinix** ()

Initializes the heap.

**heapsetx** (fill\_char)

Loads *fill\_char* into all unused bytes in the heap and checks heap integrity.

**heapwalkx** (heap\_entry)

Gathers information about the next block of the heap and puts it into the HEAPINFO structure.

**mallocx** (num\_bytes)

Allocates a block of at least *num\_bytes* from the heap.

**reallocx** (block, num\_bytes)

Reallocates an existing block larger or smaller from the heap.

**rel\_all\_blocks** (task)

Releases all blocks owned by *task*, except its stack block.

**rel\_block** (block)

Releases *block* back to its dar or heap pool.

**rel\_fbblock** (bp, flp)

Releases block back to a fast block pool.

**sort\_fblocks** (flp)

Sorts fast block pool to maximize contiguous blocks.

## Timing & Timers

*smx maintains real-time and elapsed-time clocks. It provides for wakeups, timeouts, and precision delays. Timers provide precise cyclic or one-shot timing.*

### **get\_etime** ()

Gets current elapsed time (ticks).

### **get\_stime** ()

Gets current system time (seconds).

### **read\_timer** (tmr, time\_left)

Returns time left for *tmr*.

### **sleepx** (time) or

### **sleep\_stop** (time)

Suspends or stops the current task until the specified system time.

### **SMX\_DELAY\_MSEC** (ms)

Delays for specified number of milliseconds using `count()`, to the accuracy of the tick.

### **start\_timer** (tmr, lsr, par, time, interval)

Creates and starts timer at specified *time* which runs *lsr* routine with *par* input parameter at fixed time *interval*. If *interval* == 0 timer runs once.

### **stop\_timer** (tmr, tlp)

Stops *tmr* and returns time left.

See `count()`, `count_stop()`, and `signalx()` for precision delays.

## Input and Output

*smx provides pipes and buckets for character i/o, and macros for interrupt management.*

### **create\_cx** (type)

Creates a bucket or pipe.

### **delete\_cx** (cx)

Deletes a bucket or pipe.

### **ENTER\_ISR** (), **enter\_isr**

Used to begin isr.

### **EXIT\_ISR** (), **exit\_isr**

Used to end isr. Binds interrupt service routine to the *smx* scheduler.

### **GET\_CHAR** (bucket)

Gets next character from *bucket*.

### **get\_msg** (cxchg)

Gets a message from a bucket or a pipe.

### **invoke** (lsr, par)

### **INVOKE** (lsr, par)

Invokes the link service routine, *lsr*, and passes parameter, *par*, to it.

### **num\_in\_pipe** (pipe)

Returns the number of characters in *pipe*.

### **pget\_char** (pipe, timeout) or

### **pget\_char\_stop** (pipe, timeout)

Gets the next character from *pipe*. Suspends current task or stops current task and waits if *pipe* is empty.

### **PGET\_CHAR** (char, pipe)

Gets *char* from *pipe*.

### **pput\_char** (char, pipe, timeout) or

### **pput\_char\_stop** (char, pipe, timeout)

Puts *char* into *pipe*. Suspends current task or stops current task and waits if *pipe* is full.

### **PPUT\_CHAR** (char, pipe)

Puts *char* into *pipe*.

### **PUT\_CHAR** (char, bucket)

Puts *char* into *bucket*.

### **put\_msg** (msg, size, cxchg)

Assigns a message block to a bucket or pipe, *cxchg*, to serve as a buffer of *size* bytes.

## Intertask Communication

*ITC encompasses the mechanisms by which tasks transfer data between themselves and by which they coordinate and synchronize with each other and with events. smx provides message exchanges, semaphores, and pipes.*

### **bump\_msg** (msg, new\_priority)

Changes message priority and requeues message.

### **create\_hmsg** (rxchg, blk\_size)

Creates a message in the heap and sends it to *rxchg*, unless *rxchg* is NULL.

### **create\_sema** (thres, limp)

Creates a semaphore with the specified threshold, *thres*, and one or more task priority levels.

### **create\_xchga** (type, limp)

Creates a message exchange of the specified type with one or

more task priority levels and one or more message priority levels.

### **delete\_hmsg** (msg)

Deletes a message in the heap.

### **delete\_sem** (sem)

Deletes a semaphore.

### **delete\_xchg** (xchg)

Deletes an exchange.

### **dq\_msg** (msg)

Removes *msg* from a message queue.

**locate** (msg)  
Locates the queue which a message is in.

**receive** (xchg, timeout)

**receive\_stop** (xchg, timeout)  
Gets a message from *xchg*. Suspends or stops current task for *timeout* if *xchg* is empty.

**rel\_all\_msgs** (task)  
Sends all messages owned by *task* back to their resource exchanges.

**sendx** (msg, xchg, reply)  
Sends a message to an exchange. Delivers *msg* to the top

waiting task, if any. Allows specifying an object (exchange, semaphore, etc.) to reply to.

**signalx** (sem)  
Signals a semaphore, *sem*. The top waiting task may be resumed.

**test** (sem, timeout)

**test\_stop** (sem, timeout)  
Suspends or stops the current task on semaphore, *sem*, until it is the top task and the semaphore's signal count exceeds its threshold.

## Event Management

*These calls permit counting events or monitoring AND or OR combinations of flags controlled by events.*

**count** (count, eventq, timeout) or  
**count\_stop** (count, eventq, timeout)  
Suspends or stops current task on event queue for *count* number of events.

**create\_eq** ()  
Creates an event queue.

**create\_et** (num\_slots)  
Creates an event table with *num\_slots*. (One slot is required per waiting task.)

**delete\_eq** (eq)  
Deletes an event queue.

**delete\_et** (et)  
Deletes an event table.

**reset\_flags** (et, flags)  
Resets specified flags (1 bits in *flags*) in event table.

**set\_flags** (et, flags)  
Sets specified flags (1 bits in *flags*) in event table. Resumes waiting tasks which match flags and clears their slots.

**signalx** (eq)  
Signals an event queue, *eq*. The top waiting task(s) may be resumed.

**test\_flags** (et, mask, timeout) or  
**test\_flags\_stop** (et, mask, timeout)  
Suspends or stops the current task on event table until the AND or OR of specified flags (1 bits in *mask*) is true.

## Miscellaneous

*Special purpose calls.*

**BUILD\_HT** (handle, name)  
Adds entry to handle table.

**clear\_q** (queue)  
Clears *queue*. Moves tasks to ready queue or messages to resource queues.

**create\_rq** (num\_levels)  
Creates the ready queue with the specified number of priority levels.

**ENTER\_SSR** ()  
Use to begin ansr. Makes a function into a system service routine (i.e. a restricted *smx* call).

**exit\_ssr** (retval)  
Use to end ansr. Makes a function into a system service routine (i.e. a restricted *smx* call).

**go\_smx** ()  
Initializes *smx* from information in the configuration table.

**LSRS\_OFF** ()  
Inhibits lsr's from running.

**lsrs\_on** ()  
Reenables lsr's and runs any that are waiting.

**qsize** (queue)  
Finds the number of objects in a queue.

**SMX\_DISPLAY\_ERROR** (errstr)  
Displays error string to screen, serial port, or other output device.

**unbuild\_ht** (handle)  
Removes entry from handle table.

## smx Error Types

00	OK	1D	INVALID_TIME
01	BAD_LIM	1E	INVALID_TMCB
02	BROKEN_Q	1F	LQ_OVERFLOW
03	CANT_CREATE_CBLOCKS	20	MSDOS_CALL
04	DOS_CRITICAL_ERROR	21	NHEAP_BLOCK_OVERFLOW
05	EMPTY_POOL	22	NULL_PTR
06	EVENT_TABLE_OVERFLOW	23	NULL_PTR_REF
07	FHEAP_BLOCK_OVERFLOW	24	NULL_TEST_FLAGS_MASK
09	IDLE_TIMEOUT	25	OP_NOT_ALLOWED
0A	INSUFF_DAR	26	OUT_OF_BCBS
0B	INSUFF_FHEAP	27	OUT_OF_BLOCKS
0C	INSUFF_NHEAP	28	OUT_OF_CXCBS
0D	INVALID_BCB	29	OUT_OF_MCBS
0E	INVALID_CB	2A	OUT_OF_PCBS
0F	INVALID_CXCB	2B	OUT_OF_QCBS
10	INVALID_ET	2C	OUT_OF_STACKS
11	INVALID_FHEAP_ALIGN	2D	OUT_OF_TCBS
12	INVALID_FHEAP_BLOCK	2E	OUT_OF_TMCBS
13	INVALID_MCB	2F	RQ_ERROR
14	INVALID_NHEAP_ALIGN	30	SIG_CTR_OVERFLOW
15	INVALID_NHEAP_BLOCK	31	STACK_OVERFLOW
16	INVALID_PARM	32	WAIT_NOT_ALLOWED
17	INVALID_PCB	33	WRONG_CBTYPE
18	INVALID_QCB	34	WRONG_TYPE_CXCHG
19	INVALID_SBCB	35	WRONG_TYPE_QUEUE
1A	INVALID_SP	36	WRONG_TYPE_QUEUE
1B	INVALID_SS	37	BLOCK_IN_USE
1C	INVALID_TCB		