# smx® Special Features

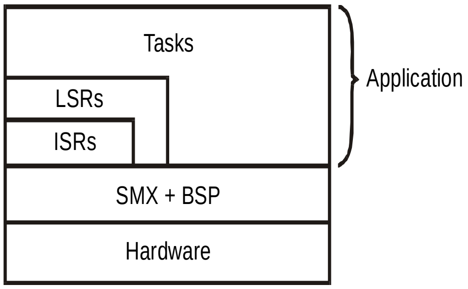
**smx is an advanced RTOS kernel. The Advanced Features section of the smx datasheet summarizes the advanced features of smx. This document presents more detailed information for the most important of these features and discusses their usage. See the smx User's Guide for more technical detail and for examples.**

## Contents

## High Performance

smx performance features permit using slower, less-costly processors. They also can avoid the need for a separate processor to handle foreground operations.

### Three-Level Structure



smx supports a three-level structure for application code, as shown above. These levels are:

- Interrupt Service Routines (ISRs)
- Link Service Routines (LSRs)
- Tasks

LSRs fill the gap between ISRs and tasks. LSRs are usually invoked from ISRs, and they run after all ISRs have run. LSRs are interruptible to allow more ISRs to run. They are ideal for applications with heavy interrupt loads due to the following:

- Deferred interrupt processing.
- Reduced interrupt latency.
- Graceful overload handling.
- Reduced reentrancy problems.

**Deferred interrupt processing.** It is general practice to keep ISRs as short as possible in order to minimize interrupt latency caused by the ISRs, themselves, and to avoid ISR reentrancy problems. With smx, interrupt service code that is less time-critical can be deferred to LSRs.

LSRs are invoked by ISRs when there is work for them to do, and they run after all pending ISRs have finished. Multiple LSRs can be enqueued in the LSR queue. The same LSR can even be enqueued more than once. LSRs perform deferred interrupt processing, including calling smx services. For more discussion, see the Deferred Interrupt Processing white paper.

**Interrupt latency** determines how frequently interrupts can occur without being missed. It is defined as the time from when an interrupt occurs until its ISR starts running and is the sum of three delays:

interrupt latency = processor latency + kernel latency + application latency

Kernel latency consists of the time that the kernel disables interrupts. Because ISRs do not make smx service calls, all smx services, LSRs, and 95% of the smx scheduler run with interrupts enabled. Consequently, smx interrupt latency is very small — about the same as the processor, itself.

**Graceful overload handling** is a feature of LSRs that is discussed in a section, below.

**Reentrancy problems** are reduced by deferring processing from ISRs to LSRs because LSRs cannot preempt each other.

**Relaxed task latency.** Critical functions that require very low-latency can be moved from tasks to LSRs. Then the required task latency can be relaxed. This permits more task design freedom.

## Scheduler Features

The scheduler is the most important part of a multitasking kernel. It decides whether or not to run a new task. To do so, it suspends or stops the current task and resumes or starts the new task.

**smx Scheduler:** smx has an advanced scheduler written in optimized C, using assembly functions for operations that C cannot perform. It runs exclusively using the System Stack with interrupts enabled about 95% of the time. SSR_EXIT() and ISR_EXIT() decide whether to call the prescheduler, which in turn decides whether to call the LSR and task schedulers. Bypass paths through the EXITs and prescheduler improve performance when the current task is being continued or exit is to another part of smx (e.g. an SSR or the scheduler interrupted by an ISR).

The LSR scheduler runs all LSRs in the LSR queue, and then returns to the prescheduler. LSRs can call SSRs and invoke other LSRs — all with interrupts enabled.

The task scheduler runs only when an operation has occurred that requires a task switch. It tests for stack overflow, stops or suspends the current task, and then starts or resumes the top task in the ready queue. In the dispatch process, it handles the difficulties related to stack sharing and scanning. (The benefits of these are discussed in sections below.) It also handles exceptional conditions such as a damaged or empty ready queue.

Prior to actually dispatching the next task, the scheduler checks the LSR queue to see if an LSR has become ready to run due to an interrupt that occurred while the scheduler was running. If so, the scheduler does an *LSR flyback*, which will run the LSR(s) and then start the task scheduling process all over, if a higher priority task has become ready to run due to an LSR running.

The scheduler also implements task profiling if enabled, task switching time measurements if enabled, and autostop due to a task running through the closing brace of its main function or returning from its main function.

The scheduler assembly functions and the prescheduler comprise about 100 lines of assembly code, making the scheduler portable to other processor architectures, with minimal effort.

**Fast Task Switching** encourages dividing an application into many tasks, thus allowing the kernel to do more of the work of coordinating operations via its intertask communication mechanisms. This simplifies application development by allowing it to be implemented with small, simple tasks that are easier to write and debug.

There are two fundamental types of commercial RTOS kernels: *task-mode* kernels and *process-mode* kernels.

smx is a task-mode kernel. All software runs in the same memory space and in the *supervisor* or *protected* mode of the processor. Task switching requires saving the current task’s non-volatile registers in its Register Save Area (RSA), moving to the new task’s TCB, restoring the new task's saved registers from its RSA, then resuming the new task from where it left off. All of this is implemented very efficiently.

Process-mode kernels separate the kernel and the application into isolated processes. Isolation requires a processor with a Memory Management Unit (MMU). In addition to the above normal task switching operations, switching from a task in one process to a task in another process requires purging the MMU look-aside buffer, switching to a different page table, reloading caches, and possibly other operations. These extra steps place a large time overhead on task switching. Messaging also encounters increased overhead due to the need to copy messages from one process's memory space to another's. Process mode is normally used in environments that run independent applications, some of which may not be reliable or trustworthy.

A task-mode kernel, such as smx, is the better choice for hard-real-time embedded systems.

**Preemptive Scheduling,** as used by smx, is the best method for embedded system task switching. As soon as a higher priority task is ready to run, it preempts the current task and runs. There is no time-slice granularity involved, as with some kernels.

The problem with time-slice scheduling is that a ready task must wait until the end of the current time slice to be dispatched. Thus, task response time is governed by the granularity of the time slice. If the granularity is too coarse, task response time is too slow. If the granularity is too fine, there is too much overhead from interrupting the current task to determine if a higher priority task is ready to run.

**Multiple Tasks per Priority Level:** smx allows multiple tasks to share a priority level. Within a group of equally important tasks, the task that has waited longest will run first. This is a more natural scheduling method as compared to requiring each task to have a different priority.

Allowing multiple tasks at the same priority level also simplifies round-robin scheduling, and priority promotion (see mutex discussion) works better if tasks can share priority levels.

**Scheduler Locking:** smx allows the current task to lock the scheduler, in order to protect itself from preemption. Locking is useful for short sections of critical code and for accessing global variables because its overhead is small. ISRs and LSRs are not blocked from running so there is no foreground impact.

Locking is also useful to prevent unnecessary task switches. For example:

```
smx_TaskLock();            /* lock current task */
smx_SemSignal(semA);
smx_SemTest(semB, tmo);    /* unlock it */
```

Without the lock, a higher priority task waiting at semA will immediately preempt this task, when it signals semA. When the higher-priority task is done, this task will resume only to suspend itself on semB — a wasted task switch. With the lock, the higher priority task becomes ready, but does not run. This task then suspends itself on semB, which automatically unlocks this task, and the higher priority task runs.

The smx locking mechanism supports nesting. This is important because a function, which does locking, may call a sub-function, which also does locking. If the sub-function left the current task unlocked, due to lack of lock nesting, then the function would be unprotected, thereafter.

## Layered Ready Queue

The layered ready queue[1] supports a large number of tasks, with minimal overhead. Enqueueing a new task in a linear ready queue, as used by most kernels, requires searching from the beginning of the ready queue until the last enqueued task of equal priority is found, after which the new task is enqueued. Obviously, if there are a large number of tasks in the ready queue, this will take significant time.

In most kernels, interrupts are disabled for the entire enqueueing time, because ISRs can make kernel calls, which result in enqueueing new tasks in the ready queue. This can greatly increase interrupt latency, which is one of the most important things in an embedded system. Making matters even worse is that the lowest priority tasks take the longest to enqueue. Hence, the least important tasks are slowing down interrupt response the most.

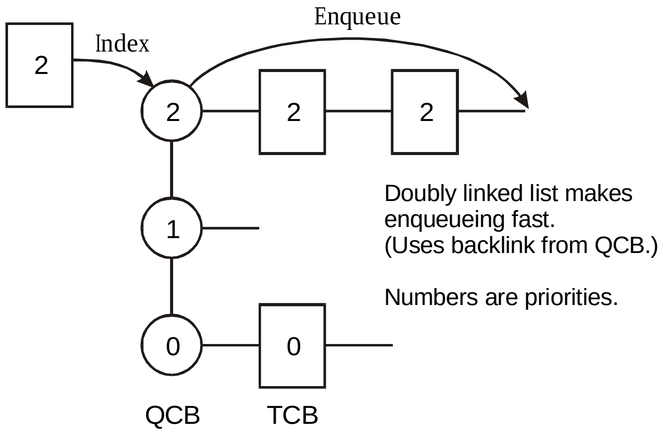
When designing smx, we observed that since smx tasks are permitted to share priority levels, embedded systems usually need no more than about 10 priority levels. Therefore, smx has a separate ready queue level for each priority level. Each level is headed by a

[1] The ready queue is where ready-to-run tasks are enqueued.

ready queue control block (RQCB). The RQCBs are contiguous in memory and in order by priority. Enqueueing a task is thus a two-step process:

(1) index to the correct RQCB, using the priority of the task, and

(2) follow the backward link of the RQCB to the end of the queue and link the TCB to it:



Doubly linked list makes enqueueing fast.
(Uses backlink from QCB.)

Numbers are priorities.

This two-step process is fast and it takes the same amount of time regardless of how many tasks are in the ready queue — 10 or 10,000 makes no difference. Finding the next task to dispatch is also fast because smx maintains a pointer to the TCB of the next task to run (which is the first task in the highest occupied rq level).

## Fast Heap

The new smx heap is a high-performance, configurable heap, with debug and safety features. It has some similarities to GPOS heaps, in that it is a *bin-type* heap. But, it is simpler, and it has an implementation appropriate for embedded systems. Its architecture is governed by the following embedded system objectives:

(1) Works with very small to large amounts of RAM.
(2) High performance.
(3) Deterministic behavior.
(4) Small code size.
(5) Ruggedness.
(6) Strong debug support.

Embedded system heaps can vary in size from kilobytes to megabytes. Hence, efficiency, performance, and adaptability are important. At the low end, efficient memory utilization is most important; at the high end, performance is most important. In general, all embedded systems need fast, deterministic block allocations. Code must be as small and efficient as possible, especially for low-end systems. Requirements 5 & 6 are discussed in appropriate sections that follow.

Most RTOSs have *linear* heaps, which require searching from the first free chunk until a big-enough free chunk is found, in order to allocate a block of the desired size. When a heap becomes highly fragmented this can require searching through hundreds of chunks.

The new smx heap has a two dimensional structure:

(1) Physical structure

(2) Logical structure

The *physical structure* is the usual linear structure with all chunks (*inuse* and *free*) linked together, in physical order. The *logical structure* consists of *heap bins*. A heap bin holds one chunk size (*small bin*) or a range of chunk sizes (*large bin*). The bins are configured by a size array and can be reconfigured merely by adding, removing, or changing sizes.

A bin heap normally starts with a small bin array (SBA). This might consist, for example, of bins for chunk sizes 24, 32, 40, ... bytes up to any desired limit. The SBA is accessed simply by converting chunk size to the bin index. As long as bins are kept full, allocation is nearly as fast as a block pool. This is very attractive for object-oriented languages such as C++, which are heavy heap users.

Above the SBA is the *upper array* of large and small bins culminating in the top bin. It takes all sizes above its limit (e.g. 2048 bytes and up). There can be any number of bins in the upper bin array. Access to the correct upper bin is via a binary search algorithm — still very fast. Within a large bin the best-fit chunk is taken after N tries, where N is a configuration option.

Several services and mechanisms are provided to keep bins full — see the smx Heap section in the smx datasheet. To the degree that this is achieved, the heap is very fast and deterministic.

### Timers Directly Launch LSRs

smx one-shot, cyclic, and pulse timers directly invoke LSRs, instead of starting or resuming tasks. Hence, timer code runs at a higher priority than any task and it cannot be blocked by any task (i.e. priority inversion is not possible between a task and an LSR). Even task locking does not prevent LSRs from running. The net result is reduced timer jitter and greater accuracy of timed operations. This is important for precise sampling and smooth control.

### No-Copy Block I/O

Through a process we call *block migration*, smx allows any block to be made into a message, which then can be propagated to tasks via exchanges. This brings the full exchange messaging capabilities of smx to bear on I/O blocks.

smxBase provides interrupt-safe block pools for use from ISRs, so that an ISR can obtain an input block from a base pool and fill it with incoming data. When full, the ISR invokes an LSR and passes the block pointer to it. The LSR *makes* the block into an smx message and sends the message to a message exchange, where a task waits to process it.

The message may be partially processed by the first task, then sent up to the next layer of the software stack via another exchange, and so forth. When the last task is done with the message, it simply releases it, and the data block automatically goes back to the correct base pool. This entire process requires no copying of the message, hence it is efficient and fast.

The reverse process can be used for block output: A message is obtained by a high-level task, partially filled, and then passed down to the next software level via an exchange. The task waiting at that exchange adds more information (e.g. a header) and passes the message down to the next level, etc.

The lowest level of the software stack (which could be a task or an LSR) *unmakes* the message into a bare block, loads its block pointer and pool handle into ISR global locations, and starts the output process. The ISR outputs all of the data, then releases the block back to the pool it came from. Like input, this entire process requires no copying of the message.

smx block migration provides considerable flexibility in that blocks can come from anywhere. If they are either base or smx pool blocks they will automatically be released back their correct pools. Whatever thread (ISR, LSR, or task) is releasing a block need not know where it came from. If the block is not from a pool, the pool parameter is NULL, and no action is taken to release the block to a pool.

## Small Memory Footprint

Small memory footprint is important for SoCs, because using only on-chip memory is cheaper and faster than using additional off-chip memory. This also allows using simpler processors, without MMUs, thus producing further cost and power savings. In addition, the smaller the memory, the smaller the chip, and the lower its cost. These savings are particularly important in low-cost embedded products that are produced in large quantities (e.g. *Things*).

SRAM requires about 6 times the chip space of flash memory. Hence, minimal RAM usage is more important than minimal flash usage, although both are important.

### Stack RAM Reduction

In general, every active task requires its own stack. (An *active task* is one that is running, ready to run, or waiting for an event.) For best performance, stacks should be in fast RAM. This is especially true if auto variables are used extensively in routines called by tasks in order to achieve reentrancy.

Depending upon the amount of function nesting, the number of function parameters, and the number of auto variables, task stacks can be quite large. Hence, systems with large numbers of tasks can require large amounts of RAM for their stacks.

In RAM-constrained designs, this tends to result in the unfortunate tradeoff of using fewer tasks than is optimum for the application. When this happens, many benefits of multitasking, such as ease of coding, are lost because operations that could be handled by kernel services become internal operations within tasks. Application code must be created to perform these operations that could otherwise be handled by existing smx services. In addition these operations are hidden from debug tools, such as smxAware.

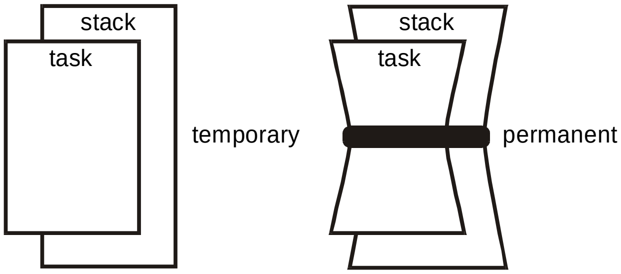
### Minimizing Stack RAM

**System Stack (SS)** is used for initialization, ISRs, LSRs, the scheduler, and the error manager. This leaves only the stack requirement of each task, itself, which greatly reduces the amount of RAM required per task and makes it easier to fine-tune task stack sizes, since the unpredictability of interrupts is removed. Of further benefit, SS can normally fit into on-chip SRAM, even if task stacks cannot, which helps to improve ISR, LSR, and scheduler performance.

**LSRs** can perform functions that might otherwise be performed by small tasks. Examples are functions invoked by timers and ISRs. Using LSRs saves task stack space because they run in SS.

**One-Shot Tasks** offer a unique stack-sharing capability for smx. A *one-shot* task is a task that runs once, and then stops. When it stops, it has no information to carry over to its next run and thus does not need a stack. smx permits one-shot tasks to release their stacks for use by other tasks, while they wait for events.

One-shot tasks are created and started without a stack. When dispatched, a one-shot task is given a temporary stack from the stack pool. When the task stops, it releases the stack back to the stack pool. smx also supports normal tasks, which have permanent stacks that are bound to the tasks when they are created.



While running, there is no operational difference between one-shot tasks and normal tasks. Both can be preempted and both can be suspended while waiting for events. Both retain their stacks, when preempted or suspended. However, when stopped, a one-shot task releases its stack back to the stack pool, whereas a normal task retains its stack.

For every normal wait service, smx provides a stop version — see smx API section in the smx datasheet. Thus, stopping vs. suspending a task introduces no constraints on what a one-shot task can do vs. a normal task. It can, for example, stop at an exchange to wait for a message, just like a normal task can suspend at the same or different exchange to wait for a message.

When a one-shot task is dispatched after being created or stopped, it is given a new stack from the stack pool, and it starts from the beginning of its main code. There is very little performance penalty for starting a task with a new stack versus resuming a task that already has a stack. (Getting a stack is balanced by not needing to restore registers from it.) If no stack is available, the scheduler passes over the task until one is available. During the wait, the task retains its position in the ready queue.

Compared to other mechanisms for sharing stack space (e.g. OSEK BCC1), the smx approach has three advantages:

(1) Any mixture of one-shot and normal tasks can run simultaneously.
(2) One-shot tasks can wait at semaphores, exchanges, etc. when they are stopped.
(3) One-shot tasks can run in any order.

If one-shot tasks do not suspend themselves (i.e. they can wait only in the stopped state), 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 were 20 one-shot tasks having 3 different priorities, then only 3 stacks would be required in the stack pool.

**Using one-shot tasks:** One-shot tasks are good as helper tasks that do some simple thing then stop. They are also good for tasks that seldom run. Why tie up a large block of SRAM for a task that seldom runs? Examples are tasks that handle exceptions, infrequent operator requests, infrequent downloads, etc. One-shot tasks are also good for mutually exclusive operations which cannot happen simultaneously (e.g. the task that starts an engine versus the task that stops the engine), for sequential operations, or for state machines.

As one gets accustomed to using one-shot tasks, the applications for them seem to multiply. Since they do not require allocation of permanent stack space it is nice to do many little things with many little one-shot tasks, rather than doing them with a few complicated tasks. Coding and debugging is much easier for little tasks.

Performance can be scaled to available RAM simply by reducing the number of stacks in the stack pool. Then one-shot tasks may need to wait for stacks. If so, performance will be reduced, but the system will continue to run correctly. With a one-shot task one gets a full task at a discounted RAM price.

**Deleting tasks:** Like most kernels, smx permits tasks to be deleted. A stack is allocated from the smx heap when a bound task is created and freed to the heap when a bound task is deleted. Task creation and deletion are relatively fast operations (though not as fast as starting and stopping one-shot tasks).

Deleting unneeded tasks is one way to reduce stack RAM usage. The downside is that deleted tasks cannot wait for events. Hence, it can be more complicated to restart them than to restart one-shot tasks. In addition, deleted tasks drop off the radar relative to debugger awareness, whereas one-shot tasks do not. For these reasons, one-shot tasks are likely to be a better solution in restricted memory systems.

### Efficient Memory Allocation

smx provides both a heap and block pools. Heaps are popular because of their flexibility and programmer preference. The new smx heap is fast and deterministic if bins are full. However, block pools are still faster and more deterministic. In addition heaps can fail due to *external fragmentation* — i.e. too many small free chunks separated by inuse chunks.

Block pools are safer, but they also waste memory, in this case due to *internal fragmentation* — i.e. blocks often are larger than necessary, and block pools often have more blocks than needed. But this is more controllable than heap fragmentation and also block pools are simpler.

Offering both a heap and block pools permits using an optimum combination for a given application.

### Sharing Dynamic Control Blocks

In a kernel, each object (e.g. semaphore, task, message, etc.) requires a *control block* to manage it. Control blocks contain information about the object. smx control blocks are dynamically allocated from control block pools. For example, when a task is created, its task control block (TCB) is obtained from the TCB pool. If the task is later deleted, its TCB is returned to the TCB pool and can be reused. This contrasts with most kernels for which all control blocks are statically allocated at link time and cannot be shared.

For an smx data block, a block control block (BCB) is allocated from the BCB pool and linked to a data block allocated from a data block pool. Both are returned when the smx block is no longer needed. The same is true of smx messages, which use message control blocks (MCBs) linked to data blocks. In a dynamic situation (e.g. receiving packets over a communication link), dynamically allocated control blocks can result in significant memory savings and flexibility vs. statically allocated control blocks.

## Debug Aids

Multitasking problems can be difficult to debug. Simple, free kernels provide almost no help. This can be costly in lost development time and missed schedules. Generally speaking, we spend most of our time testing and debugging. Hence the debug aids provided by an RTOS kernel are very important.

Bugs are a serious problem whether malware exploits them or not. However, the threat of malware increases the importance of fixing bugs and reducing other weaknesses that might be exploited. It is sobering to realize that embedded system programmers may have less time to fix bugs than malware writers have to exploit them!

### Debugger Support

Of course a debugger is the first line of attack on bugs. We have done a lot of debugging with smx and we have made many improvements to make debugging an easier and more pleasant experience.

smx does extensive error checking, which can be a strong ally in the war on bugs. When an error is detected, smx sends an error messages to the console[2] identifying the error type, such as “Out of TCBs”. Many minor problems like this are easily found and fixed by keeping an eye on the console or a watch on smx_errno.

smx employs many features to make debugging with a debugger easier. Among these are: smx control blocks are defined as structs with meaningful short names. Clicking on an smx object name in the

[2] Typically a terminal emulator running on a computer connected to a serial output from the board under test.

debugger watch window opens the struct display, revealing its fields and showing their values.

Most objects have user-assigned names, which makes identification easier — especially when tracing links. For most objects, names are stored in the fourth field of their control blocks. Fields with fixed values, such as cbtype or state, are defined with enums so field values are easily understood, for example SMX_CB_TASK or SMX_WAIT.

Link fields are defined as control block type pointers, when possible, so that clicking on a link field in one control block reveals the fields of the control block to which it points. Often it is helpful to trace through a queue by repetitively clicking forward links.

smx uses short names to facilitate viewing fields, variables, and values in reasonably narrow columns to conserve display space. This allows keeping the watch window reasonably narrow, even when tracing queues. Source code for smx is limited to 80 columns so that the source code window is also fairly narrow. This allows displaying at least one or two additional windows, such as local variables, registers, disassembly, memory, call stack, and notes on the center display.

Generally speaking, the more data seen at once, the greater the likelihood of spotting anomalies — i.e. things that do not look right. These often lead to finding and fixing bugs. The human brain is good at spotting anomalies, such as a wolf amongst the sheep.

The **smxAware** debugger plug-in is a graphical and textual kernel-awareness tool that shows system-level views. It is opened from the debugger menu. It allows accessing: smx objects, text event log, and graphical displays including:

- Event Timeline showing all tasks, ISRs, and LSRs and their interactions,
- Profile bars for tasks, ISRs, and LSRs
- Stack and Memory usage bar graphs, and
- Memory Map Overview, which shows what is where in memory.

See the smxAware datasheet for details.

## esmx (examples for smx)

esmx can be linked into the Protosystem by uncommenting #define SMX_ESMX in the master preinclude file. It has examples that show a wide range of usages for smx objects. For example, etask() has 7 normal task demos and 6 one-shot task demos. Unlike the simplified demos in the manuals, these demos are complete, and they compile and run. Thus, the all-important details are there.

Extreme misery in debugatory can be avoided by picking a close example, stepping through it to learn its secrets, and then modifying it to do what you need.

## Event Logging

If enabled, events are logged into the event buffer (EVB) for later viewing by smxAware, or via the debugger. EVB is a cyclic buffer of variable-size event records. Records are created by logging macros inserted into smx code. The types of events that are logged are:

- Task start, stop, suspend, and resume
- SSR calls
- LSR entry, exit, and invoke
- ISR entry and exit
- Errors
- User events

Logging can be selectively enabled or disabled by event type and also by one of eight SSR groups, defined by the user.

Each record is precisely time-stamped. EVB can be uploaded and viewed through smxAware as a graphical timeline or as an event log. The timeline provides a convenient graphical view, which can be zoomed in or out. The event log is useful for support — it is used to record what was happening in a system when an error occurred.

Recording errors in EVB is helpful because it allows seeing them in the context of other system and user events, which helps to find their causes. Errors appear as red dots on timeline graphs.

Seven user event log macros are provided to log from 0 to 6 parameters per event. User event macros can be placed, like printf()'s, anywhere in the code. Events appear in the smxAware timeline or event log, relative to system events. User events appear as white tick marks within task/LSR/ISR bars, similar to other events. Mousing over them with the Details button on shows the parameters logged.

## Stack Overflow Detection

**Stack overflow** is a common problem in multitasking systems. It can damage other tasks' stacks, the heap, data buffers, etc. Often the culprit is not the task to experience problems. Thus it can be difficult to debug stack overflow and it can waste precious debug time.

The smx scheduler checks the stack pointer and the stack high-water mark when suspending or stopping a task. If either indicates an overflow, the error manager is called, which logs the problem and sends an "smx STK OVFL" message to the console.

**Stack padding** is useful during development. A configuration option allows placing stack pads above all stacks of user-selectable size. Stack pads absorb stack overflows thus preventing damage, which allows the system to continue running.

smxAware has a **stack usage window**, which shows stack usage per task and which stacks have overflowed into their stack pads. This makes spotting insufficient stack sizes easy and enlarging them while you still have hair left.

## Task Stack Sizing

Multitasking requires multiple stacks and stacks are one of the biggest RAM users in a system, so fine-tuning stack sizes is important to save memory. smx maintains a high-water mark for each task to indicate maximum stack usage by the task.

Stack scanning is the most reliable way to measure stack usage — a task need not be stopped at peak stack usage in order to see that there is a problem. Stack scanning is done from the idle task in order to not take time away from important tasks. The stack high-water mark is maintained in each task's control block. Stack usage can be viewed graphically in smxAware, or it can be seen in the debugger watch window by comparing the shwm field to the ssz field in a task's TCB.

As previously noted, the use of the System Stack for everything except tasks makes task stack sizing much easier to do and less risky.

## Heap Debugging

Heap debugging is another challenging endeavor. Particularly problematic are block overflows that damage heap links, thus causing the system to experience exceptions, such as attempting to access non-existent memory. Heap leaks, due to tasks forgetting to free blocks, are also difficult to find.

In order to maximize memory efficiency, inuse chunks have only a forward link and a backward link + flags, producing a total overhead of 8 bytes. Looking at them is not helpful, other than to see whether they have been damaged.

The smx heap provides a *debug mode*. When the debug mode is on, block allocations produce *debug chunks* instead of *inuse chunks*. A debug chunk consists of a heap debug control block (HDCB) followed by fences, followed by the allocated data block, followed by more fences. (Fences are words with a fixed pattern.) The number of fences and the fence pattern are configuration options. Fences are traps — enough fences will keep the overflow inside the chunk, thus preventing damage to the heap, and allowing the system to keep running, so you can catch the bug causing the overflow.

Obviously, a debug chunk can be much larger than an inuse chunk. This is why selective use is permitted — debug mode can be turned on only while suspected tasks or functions run. Thus debug chunks can be used even in tightly memory-constrained systems.

An HDCB contains the block owner (task or LSR that allocated the block), time of allocation, and size. These are helpful to track down forgetful tasks that cause memory leaks.

Sometimes, the only way to find a heap problem is to manually scan the heap using the memory window. This is a mind-numbing experience, fraught with human error. To reduce eye and brain strain, heap fill mode can be turned on. In this mode, the top chunk is filled with a unique pattern, during heap initialization. Thereafter, data blocks are filled with a unique pattern, when allocated, and with another unique pattern, when freed. This helps a lot to separate data from metadata and free from inuse. It is also easy to see the top chunk calving and becoming ever smaller. Although this may not be helpful, it is interesting to watch.

## Precise Profiling

If profiling is enabled, precise run times are recorded for each task, all ISRs combined, and all LSRs combined. Overhead is calculated as the remaining time per frame. Resolution is one tick timer clock, which depends upon hardware and BSP code, but is typically one to several instruction clocks. Hence, even infrequent, small tasks will accumulate run time counts (RTCs).

Task profiling is implemented by RTC functions built into smx that record the tick timer count when a task starts running and the count when it stops running. The differential count is then added into the RTC field in the task's TCB. Similar functions accumulate RTCs for combined ISRs and combined LSRs.

The profile frame size can vary from one tick to hundreds of seconds. Small frames are typically useful during debug, whereas long frames may be more useful for systems in the field, in order to accumulate continuous operational data without overloading storage and communication facilities.

At the end of a frame, all RTCs are copied into an RTC array and cleared. The array has a row per RTC and a column per frame. Each column constitutes a sample. When full, each new sample overwrites the oldest sample. Profile data is displayed via smxAware in graphical form, or it can be viewed in a debugger watch window. Coarse profile values displayed on the console are calculated from RTCs each second.

Run time counts show where processor bandwidth is being used and can help to spot problems such as excessive overhead or tasks hogging the processor.

RTCs can also be used for run-time limiting. For example, a system-monitoring task could monitor RTC frames and make task priority adjustments to achieve more balanced operation. It could even suspend overly-active tasks for one or more frames. This could be a means to defend against denial-of-service attacks or to stop tasks that are in infinite loops.

### Precise Time Measurements

smxBase services are available to measure precise times between a reference point and one or more end points. This is helpful for system optimization. Results are typically stored in an array. Resolution is the same as for profiling. Hence, very precise measurements can be made for function execution times, response times, switching times, interrupt latencies, etc. The maximum time measurement is limited to one tick period.

Use of time measurement services is good for monitoring times in running systems. For example, it is used in smx_HeapMalloc() to accumulate best and worst allocation times. During debug, smxAware provides tools for precise measurements of operations over short to long periods. This may be the easiest way to get quick time measurements.

Elapsed time (etime) can be used for longer time measurements. Resolution for etime is one tick. One tick resolution, which is offered by many RTOS kernels, is not adequate for modern processors, some of which can execute one million instructions in a single tick!

## Safety, Security, and Reliability

Concern about SS&R issues is growing due to deployment of Internet of Things, proliferation of embedded systems, and greater exposure to malware and other threats, such as high-energy particles. A kernel can either be a big help or a big hindrance in dealing with these matters.

Kernel misuse can be accidental (a bug) or intentional (malware). Either way, it is important to reduce the ways in which it can happen. If the kernel is able to detect and recover from misuse, it can help in developing safe, secure, and reliable systems. On the other hand, if the kernel has many serious vulnerabilities, it may be impossible to build a safe, secure, and reliable system using it.

smx already has many protections in place, and more are being added. The following sections discuss current features and how they can help to develop safe, secure, and reliable systems.

### Error Detection and Management

**Error Detection** is the starting point and smx monitors about 70 error types. These include invalid parameters for services, broken queues, heap overflow, stack overflow, invalid control blocks, null pointer references, and resource exhaustion. These are all checked frequently. smx provides three ways to deal with detected errors:

**Local Error Management** smx services return FALSE or NULL, if an error or a timeout has occurred. For reliable operation, calls to smx services should be implemented as follows:

```
if (smx_Call())
        /* do normal action */
else
        /* use SMX_ERR to fix problem */
```

SMX_ERR is macro that tests the err field in the current task's TCB. If it is 1, a timeout has occurred and the most appropriate action might be to try again; otherwise, an error has been detected and SMX_ERR is the error number; it should be used to determine what to do.

Local error management provides the most precise error management because it is done in the context of the error. However, too much local error management bloats the code and makes it difficult to understand.

**Central Error Management.** smx provides a central error manager, smx_EM(), to reduce the need for local error management. When an error is detected, smx_EM() is automatically called. It records the error number in smx_errno, increments a 32-bit global error counter, and increments an 8-bit counter for the error type. If the error occurred in the current task, the error number is loaded into its err field. Information regarding the error may also be recorded in the error buffer (EB) and in the event buffer (EVB), and an error message is queued for output to the console.

These actions may be adequate for most errors in most systems. Sadly a project may be out of time to do better.

**Error Hook.** smx_EMHook() is called from smx_EM() to provide a middle ground between overly encumbered local code and non-specific central code. It allows the user to add specific error handling code for certain error types.

For example a switch can be made on the error type. Then code for that error type has access to the current task and other variables to effect a recovery or to provide a better record of the error, such as using user event macros to log more information about it into EVB. This code could also send reports back to a central location for analysis.

**Safe Error Manager.** An error manager must be designed to not cause an error, by itself. Such an error would occur under rare circumstances, which could be extremely hard to duplicate. And such an error would be totally unexpected. smx_EM() does the following to avoid this:

It runs under the System Stack so that it cannot cause a task stack overflow, while processing another error. (One would not normally think to include extra stack space for error handling when tuning task stack sizes, and the extra RAM requirement per stack would probably not be welcome, either.)

Error message pointers are enqueued for later console output by the idle task to avoid taking time from critical tasks.

Interrupts are enabled so that critical interrupts will not be ignored due to unrelated error processing.

**Error Buffer (EB)** is a cyclic buffer of error records. Each error record contains the error type (errno), the time of occurrence, and the task or LSR in which it occurred. The error buffer size can be adjusted from a few records to hundreds of records, as needed. EB can be viewed as an array of records in a debugger or symbolically in smxAware.

**Stack Overflow.** If a task cannot be suspended because its Register Save Area (RSA) would be overwritten, the task is automatically restarted. If damage has or will occur outside of the stack, a system exit is invoked. It, in turn, may cause a system reboot, as determined by application code.

## Task Timeouts

Every smx service that causes a task to wait requires a timeout to be specified. Timeouts are primarily for safety, but can also be used for accurate timing. For safety, they ensure that tasks do not wait indefinitely for an event that has failed to occur. If no wait is desired, NO_WAIT can be specified. If no timeout is desired INF can be specified. However, requiring timeouts on all task waits can aid debugging and increase reliability.

Each task has a timeout register that records the time by which it should be resumed or restarted. The soonest timeout is compared to etime by the smx_TimeoutLSR(). If less than or equal to etime, the corresponding task is resumed or restarted. The timeout LSR can be invoked every tick, if timeouts are being used for accurate timing, or after many ticks if timeouts are only being used for safety. Either way, the timeout mechanism has been designed to be efficient and add very little overhead.

## Graceful Overload Handling by LSRs

Peak interrupt loads are difficult, if not impossible, to predict and they are likely to occur at the worst possible times. In such situations, LSRs provide a safety mechanism. This is because an LSR can be invoked many times before running and it can be interrupted and invoked more times while running. Each invocation can be passed a unique parameter such as a timestamp, a reading, or a pointer. When the peak load has passed, LSRs run in the order invoked, thus preserving the order of events.

This behavior achieves graceful degradation under stressful situations. The system slows but does not break. Deadlines may be missed, but order is not lost. In this circumstance, a control system may become sluggish but continue operating. The resulting dampening might be beneficial to reduce physical stress on machinery or to give an operator a chance to shut the system down. In a data acquisition system, data will not be lost.

## The Unstoppable LSR

Unlike tasks, which can be preempted by higher priority tasks or blocked by lower priority tasks, LSRs are not subject to delay by any task. They are immune to priority inversion because they cannot wait on events. They are simple creatures, which, except for interruption by ISRs, are undeterred in their jobs. The unstoppable LSR may be exactly what is needed for safety-critical and time-critical functions. For more information on LSRs, see the Link Service Routines white paper.

## Safe Messaging

Most kernels offer rudimentary messaging in which pointers to messages or small messages are passed via what are called *message queues*. smx pipes can be used to do the same thing, so this kind of messaging is supported by smx. However, it has many problems with regard to reliability and efficiency:

- The receiving task cannot verify that it has received a valid pointer to the start of a message. The pointer could point anywhere, which could result in processing wrong data or writing to a wrong memory area.
- Message size is not specified. Some other method must be used to tell the receiving task the size of the message it has received.
- No message priority exists to permit more important messages to be processed first.
- No indication is given of where to return a used message when it is no longer needed.
- An owner is not specified, so there is no automatic means to avoid memory leaks when owner tasks are deleted.
- A reply address is not specified for servers to know where to send replies to clients.
- Passing messages directly through pipes requires copying them in and copying them out.

Application code can compensate for these deficiencies, but doing so requires new code to be written and debugged. It also reduces independence between tasks because receiving tasks must know more about the messages they are receiving.

When messages and their parameters get separated, the potential for errors increases. An smx message consists of a message block, holding the actual message, which is linked to a message control block (MCB), holding the message parameters. This is consistent with good programming practice.

A message's handle is a pointer to its MCB. A correct handle is verifiable because it must be in the range of the MCB pool and it must point to a structure with the MCB type in its cbtype field. Hence if a message handle is damaged, it is not likely to be accepted and used.

MCBs increase the independence between tasks and also their tolerance to change. For example, an smx receiving task need not know the message size. It simply loads the MCB size field into its local counter. When the counter reaches 0 the message is empty and can be released back to its pool. Thus, messages can vary in size, without changing the code.

Using bundled parameters also helps to avoid miscommunication between task authors and is more adaptable to change. Some parameters, such as the owner, are used by smx to increase reliability. For example a task cannot send or release a message that it does not own. This avoids the problem of a task accidentally releasing a message that it sent to another task.

Embedded within the MCB is the message block pointer. Neither the sending task nor the receiving task has any reason to alter this pointer, which is loaded by smx[3]. Both work with their own block pointers. Hence, it is unlikely that a receiving task will process wrong data due to receiving a bad message block pointer. Furthermore, return of the message block to its pool is governed by the pool handle in the MCB, not by a message pointer, so this is safer, too.

[3] In this kind of discussion, it is assumed that smx code is more reliable than new application code because smx code has been proven by use in many projects.

## Message Exchanges

smx messages are sent to and received from message exchanges. An *exchange* is an smx object that enqueues either messages or tasks, whichever is waiting for the other. The use of exchanges has important advantages over direct task-to-task messaging, used in some kernels:

- Anonymous receivers. The receiving task’s identity is not hard-coded into the sending task’s code. The sender simply sends its messages to a known exchange. Thus, it is easy to swap receiving tasks without altering sending task's code. This can be useful for handling different product versions, different installation configurations, going from startup mode to operating mode, or for other reasons.
- No limit on the number of waiting messages at an exchange.
- Work queues. A message queue at an exchange is a work queue for a server task that is receiving messages from that exchange. These messages could be coming from many client tasks.
- Token messages. A message can be used as a *token* to control access to a resource and, at the same time provide information to access the resource (e.g. its port number).
- Broadcasting, multicasting, and distributed message assembly are supported.

## Message Priorities

smx messages have priorities. Exchanges permit servicing high-priority messages ahead of low-priority messages. This allows more urgent work to be completed 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 received message. This allows a client task to pass a priority to a server task via the messages that it sends and thus control the priorities at which they are processed. For example, a 911 exchange could give priority to fire over lost dog.

This is especially useful for *resource server tasks*. Adjusting resource server task priorities permits them to operate as extensions of their client tasks. This is preferable to client tasks directly accessing resources, because then delays and priority inversions can occur. In this approach, the main task goes on with its work, knowing that the server task will access the resource at a later time. When done with its work, the main task might wait at another exchange or semaphore for a reply from the server task, which tells it if the resource operation was completed properly.

## Safe Block Pools

Unlike the bare block pools provided by smxBase, smx provides safe block pools. smxBase pools are useful for ISRs and drivers, but they lack important safety features. This is ok in low-level, non-multitasking code which generally is simple and requires high performance, but it is not good in tasks.

An smx block consists of a data block linked to a block control block (BCB). smx blocks are manipulated via their handles, which are pointers to their BCBs. smx blocks have the following advantages over bare blocks:

- Block pool services are task-safe.
- Blocks are automatically freed to their correct pools.
- A block is automatically freed if its owner task is deleted.
- Pool information can be obtained via a block’s handle.
- smx block pools are more easily created and deleted than base block pools.

For example, to release a block, it is necessary only to know its handle — smx knows its pool and other particulars about it (e.g. its size). This prevents releasing a block to the wrong pool, which can be nasty.

As with smx messages, a bare block can be *made* into an smx block and an smx block can be *unmade* into a bare block. Hence, for those who prefer message queues to exchanges, bare blocks can be made into smx blocks and the smx block handles can be passed via a pipe. This improves the safety of queue messaging.

## Reliable Heap

**Self healing**. Heaps are fragile structures due to their high concentrations of pointers that are embedded in the data. These pointers present a large target for block overflows, wild pointers, and energetic particles that cause bit flips. Since embedded systems are often deployed in remote places and are expected to run for extended periods, some amount of *self-healing* is desirable

Hence, the new smx heap incorporates a scanning function that is called from idle. This function scans only a few nodes forward per pass so other tasks can run. It reports errors found and fixes them if it can. A companion function can be called to scan backward through the heap in order to fix broken forward links. Heap self healing is a work in progress.

**Heap recovery**. Another common problem in heaps, especially if memory is limited, is not being able to allocate a large block, after running for a long time. smx_HeapRecover() is provided to search for adjacent free chunks that can be merged to form a big-enough free chunk and thus recover.

## Dynamically Allocated Regions

smx is shipped with two DARs:

**SDAR** is used for smx objects, such as control blocks. It is typically only a few kilobytes in size, and can usually be put into on-chip SRAM to improve performance.

**ADAR** is used for the heap, stack pool, user-defined block pools, and other dynamic objects. It normally is large and is located in external SDRAM, if present.

Isolating smx objects from application objects improves reliability and helps debugging by reducing damage to smx objects due to application code bugs. This is a pernicious problem that cuts the ground out from under one, because one assumes that the kernel is running properly. Wrong assumptions are the hardest problems to fix.

## Avoiding Unbounded Priority Inversion

This occurs when a higher priority task is waiting for an object owned by a lower priority task and the lower priority task is preempted by one or more mid-priority tasks. The mid-priority task(s) may run for any length of time, thus causing unbounded priority inversion for the high priority task. This can cause high-priority tasks to miss deadlines causing system failures.

The traditional solution to this problem is for tasks to wait on mutexes, which control access to the shared resources. Most RTOSs, including smx, offer *priority inheritance* to prevent unbounded inversions. With this, when a high-priority task waits at a mutex, the current owner's priority is boosted to the same priority. This prevents mid-priority tasks from running.

smx mutexes also implement *priority propagation*, which means that if the mutex owner is waiting on another mutex, the owner of that mutex is also promoted. Priority promotion propagates to all mutexes linked together, in this way. Hence the owner of the final mutex, and all owners in between, will be promoted to the high priority. Many kernels do not offer this.

smx mutexes also implement *staggered priority demotion*. This means that when a task releases a mutex, its priority is demoted to the highest priority of any task waiting for its other owned mutexes. When the last mutex is released, the task reverts back to its normal priority.

These features ensure that smx mutexes work reliably in complex multi-mutex situations.

## Deadlock Prevention

smx mutexes also offer *ceiling protocol*, which is a simpler way to avoid unbounded priority inversion. With it, when a task obtains a mutex, its priority is immediately raised to the ceiling priority of the mutex. The ceiling priority of a mutex is set to the priority of its highest-priority potential owner. Thus only tasks of higher priority can run and these tasks do not use the mutex.

Among the group of tasks using a mutex, if other mutexes are also used, then the highest priority of any user of any mutex in the group of tasks becomes the ceiling priority of all mutexes in the group. Then, if any user gets one mutex in the group, no other user can run. This makes *mutex deadlocks* impossible, because there can be only one owner, at a time, of any mutex in the group. Hence, ceiling protocol provides a simple solution for two vexing problems of mutex usage.

One downside of ceiling protocol is that it is not automatic. Hence, if a higher-priority user is added or if the priority of a current user is raised above the ceiling, the ceiling will fail to protect against unbounded priority inversion for that user. To guard against this, priority inheritance may be enabled along with ceiling protocol. Then users above the ceiling will be protected. In fact, the ceiling may be set at a mid-level, if so desired. This could be used to prevent deadlocks among lower-priority users that share a group of mutexes while not preventing higher-priority users that use one mutex in the group from running.

# Other Special Features

## Power Management

The smx_PowerDown(sleep_mode) service provides a power-down framework, at the RTOS level. It is called from the idle task, when there is no useful work to do. It in turn calls the sb_PowerDown(sleep_mode) function, which does the actual power down process and time recording. This is hardware and application dependent and thus must be user-implemented.

When power is restored, control returns to the sb function, which must determine time lost, initialize the tick timer counter, and return ticks lost to smx_SysPowerDown(), which then performs *tick recovery*. This process handles events which timed out during the power-off period. It does this in an efficient manner that preserves the proper order of timed events. Execution time for tick recovery is dependent upon the number of events that timed-out, not upon the number of ticks lost.

Operation is largely transparent to the application, if sb_PowerDown() is able to accurately determine the time lost.

## Easy to Use API

The smx API has a high degree of symmetry and orthogonality. By *symmetry* we mean that what can be done can be undone. For example, smx offers a delete function for nearly every create function, a start function for every stop function, and a resume function for every suspend function.

By *orthogonality*, we mean that kernel services operating upon tasks do not depend upon the task's state. For example, deleting an smx task has the same result, regardless of its state. This sounds easy, but consider if the task is in the run state, then it is actually deleting itself, which is not easy to do. However task self-deletion is actually useful, underlining the importance of orthogonality.

Limiting the number of parameters per function is also important for ease of use. Nearly all smx functions have three or fewer parameters. Functions with many parameters can be difficult to use correctly — there being problems remembering what each does, their order, etc. In addition, some combinations may not be legal and others may be untested.

As much as possible, smx avoids restrictions on the use of its services. Restrictions are easily forgotten, leading to problems. For example, most kernels go into an undefined state if a task tries to delete itself or if the idle task is accidentally deleted. We try to avoid Achilles heels, like these, because they can happen during normal operation and usually have bad consequences.

See the smx API section of the smx datasheet for details.

## C++ Support

smx maintains a *this* pointer in the TCB of each task. When a task switch occurs, the new task's this pointer is loaded into the global this pointer.

Another C++ requirement stems from global objects. A C++ compiler generates *initializers* for global objects. Initializers run from the boot code before main() is called. In order to support initializers, all smx objects are auto-created and initialized as needed. For example, when the first task is created (possibly by an initializer), the TCB pool and stack pool are automatically created, first. Since the stack pool comes from ADAR, it is initialized before the stack pool.

**smx++** is an optional product that goes a step further by providing smx base objects from which C++ application objects can be derived or which can be used as member objects in them. See the smx++ datasheet for more information.