

# smx<sup>®</sup> Special Features

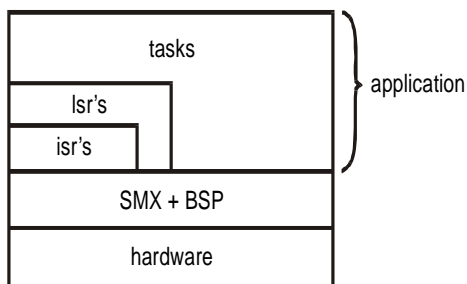
## for 32-bit embedded systems

*smx is a real-time multitasking kernel for hard real time embedded systems. It offers many features to permit using slower processors and less memory, thus reducing product cost. It also offers features to increase the safety and reliability of developed products and to reduce development time. These are summarized in the box to the right and discussed in detail in this document.*

### Performance Features

The features described herein not only permit using slower and less costly processors to perform the same functions, but also they permit handling rapidly occurring real-time events and performing background processing on the same processor. This avoids complex and costly dual processor designs.

#### Three-Level Structure



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

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

### FEATURES

#### High Performance

- Three-level structure for hard real-time: ISRs, LSRs, and tasks
- Very short interrupt latency
- Deferred interrupt processing
- Fast task switching
- Preemptive scheduling
- Scheduler locking
- Layered ready queue
- Timers directly launch LSRs for tight timing control

#### Small Size

- Efficient code
- Minimal RAM usage
- One-shot tasks permit stack sharing
- Dynamic objects

#### Reliability and Safety

- Comprehensive error detection and management
- Timeouts on all task waits
- Safe messaging via exchanges
- Mutexes for safe resource sharing
- Protected heaps

#### Ease of Use

- Processor-specific BSP
- Tool-specific project files
- Quick-start *Protosystem*
- Well-written manuals
- Task-aware debugging
- Graphical Analysis Tools
- C++ Support

*Link service routines* fill the gap between ISRs and tasks. They offload foreground processing from ISRs and they permit *smx* service calls to be made from the foreground. *smx* service calls are not permitted from ISRs. Instead, ISRs *invoke* LSRs. LSRs run after all ISRs have run and they run with interrupts enabled. Therefore, LSRs produce the following benefits:

- Short ISRs – reduces interrupt overruns and interrupt nesting.
- Low *smx* interrupt latency – reduces missed interrupts.
- Graceful overload handling – LSRs can be multiply invoked and execute in order
- Reduced reentrancy problems because LSRs cannot preempt each other nor *smx* service calls from tasks

These advantages make *smx* ideal for application with heavy interrupt loads and hard real-time requirements. For more information on LSRs, see the [\*Link Service Routines\*](#) white paper.

## Low Interrupt Latency

Interrupt latency is defined as the time from when an interrupt occurs until its ISR starts running. This is an important 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. Most kernels do this in order to protect internal structures (e.g. queues) while they are being changed.

Because *smx* services may not be called from ISRs and because any pending *smx* call must finish before an LSR can run, it is not necessary to disable interrupts in *smx* calls, nor in 98% of the *smx* scheduler. In fact *smx* disables interrupts only to assure that LSRs run without needless delays. Consequently, worst-case *smx* interrupt latency is almost negligible — it is about the same as the processor, itself.

## Deferred Interrupt Processing

In real-time systems, ISRs should be kept as short as possible to minimize interrupt latency caused by the ISRs themselves. Non-essential interrupt service code should be deferred. LSRs provide an excellent mechanism for this. They can be invoked to perform deferred interrupt processing, as well as calling *smx* services. Unlike a task, an LSR can be invoked many times, then run an equal number of times. When invoked, a unique parameter or pointer can be passed to the LSR. Thus, each invocation of an LSR can process unique information. LSRs permit handling peak loads of interrupts without loss of information.

*smx* adds very little overhead to ISRs. It requires adding an ISR enter macro, an LSR invoke macro, and an ISR exit macro to each ISR. The ISR enter macro saves a few registers and increments the *smnest* counter. The invoke LSR macro stores the LSR address and one parameter into the LSR cyclic queue. The ISR exit macro decrements the *smnest* counter and tests whether to call the LSR scheduler. If not, it restores the saved registers and returns to the interrupted task. High priority ISRs, which need not be linked to *smx*, may avoid even the small overhead of these macros.

Under *smx*, ISRs and LSRs operate using the current task's stack. This is faster because it avoids the stack switching time penalty, with its associated pushing and popping of registers. *smx* also permits nested ISRs. Some RTOSs do not. Nesting of ISRs can be important to prevent high priority interrupts from being blocked by low priority ISRs.

For more discussion of this topic, see the [\*Deferred Interrupt Processing\*](#) paper.

## Scheduler Features

The scheduler is the most important part of any multitasking kernel. It determines 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.

**Fast Task Switching:** There are two fundamental types of commercial kernels: *task-mode* kernels and

*process-mode* kernels. *smx* and other task-mode kernels operate in a single memory space and usually all software runs in *supervisor* mode. Normal task switching requires only pushing the current task's registers onto its stack, moving to the new task's TCB, popping its registers from its stack, and resuming it from where it ceased running.

Process-mode kernels (e.g. Linux and Windows XP) divide application code into isolated processes. One process cannot access the memory of another process or that of the kernel. This isolation is accomplished by using the Memory Management Unit (MMU) of the processor. In addition to normal task switching operations, switching from one process to another requires purging the MMU's look-aside buffer, switching to a different page table, and other functions. These extra steps cause process switching time to be much longer than task switching time. In addition, interprocess messages must be copied from one memory space to the other, which also reduces system performance.

Process mode is justified in environments that run independent applications, some of which may be of low quality or even hostile. Because of its success in such environments, there is a belief that isolated processes will increase the reliability of embedded applications. This supposition should be carefully analyzed, in any given instance. Reduced performance, increased complexity, and more difficult debugging are likely to have just the opposite effect for most embedded systems.

**Preemptive Scheduling** is the best method for embedded systems. Despite that, many well-known RTOSs (e.g. WinCE and Linux) 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. Thus, response time is governed by the granularity of the time slice. If the granularity is too coarse, response suffers. If the granularity is too fine, there is too much overhead interrupting the current task to test if a higher priority task is ready to run.

The *smx* scheduler is a *preemptive* scheduler. This means that, as soon as a higher priority task becomes ready to run, it preempts the current task and runs. There is no granularity to be concerned about. Hard

real time responsiveness is much easier to achieve with preemptive scheduling.

**Multiple Tasks per Priority Level:** Some kernels (e.g. uC/OS) require each task to have a unique priority. This is not optimum, because it usually is not possible to define priorities so precisely, and within a group of equally important tasks, it is usually better for the task that has waited the longest, to run first. Allowing multiple tasks at the same priority level also simplifies *round robin* scheduling among those tasks, which is a good way to share processor time among equally important tasks.

**Scheduler Locking:** *smx* allows the current task to *lock* the scheduler (which means to block it from dispatching a higher priority task.) Many kernels do not provide this feature. With the addition of locking, there are three main ways to protect access to a resource:

- (1) disabling interrupts,
- (2) using semaphores and mutexes and
- (3) locking.

The first method is the only way to protect a resource shared between an ISR and other ISRs, LSRs, or tasks. However, it increases interrupt latency and *should be used as little as possible*. Semaphores are the traditional method to protect resources shared between tasks. They do not cause interrupt latency, but they do increase processor overhead. Using semaphores is inefficient for protecting short, *critical sections*<sup>1</sup> of code.

Locking and unlocking require very little overhead (especially if no preemption occurs when unlocking.) In addition, *task latency*<sup>2</sup> is reduced. Hence, locking is good for short, critical sections of code. Note that locking affects only tasks. It has no effect upon ISRs and LSRs.

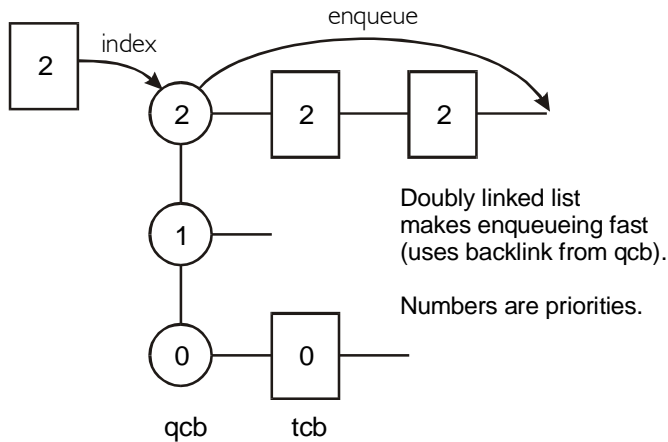
---

<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 delayed due to the scheduler not being allowed to run. This can occur due to a system service call running or due to the scheduler being locked.

## Layered Ready Queue

The *layered ready queue*<sup>3</sup> is unique to *smx* and of major benefit, if you wish to have a large number of tasks. Nearly all kernels, other than *smx*, utilize a



linear 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 after it. Obviously, if there are many ready tasks, this could take a long time. In most kernels, interrupts must be disabled for the entire enqueueing time! Making this algorithm even worse is that the lowest priority tasks take the longest to enqueue.

When designing *smx*, we observed that since *smx* tasks are permitted to share priority levels, the typical embedded system needed no more than 5 to 10 priority levels. (How finely can you determine the relative importance of a task?) Hence, *smx* has a separate ready queue for each priority level. Each level is headed by a *Queue Control Block (QCB)*. The QCBs are contiguous and in order by priority. Thus, enqueueing a task is a two-step process: (1) index to the correct QCB, using the priority of the 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 – 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.

<sup>3</sup> The ready queue is where ready-to-run tasks are enqueued.

## Timers Directly Launch LSRs

*smx* one-shot and cyclic timers directly invoke LSRs, instead of tasks. Hence, timer code runs at a higher priority than any task and it cannot be blocked by any task. This also assures that timer code cannot be blocked by task priority inversions. Even task locking has no effect upon it. The net result is reduced jitter and greater accuracy of timed operations (e.g. input sampling, pulse width modulation, stepper motor control, etc.)

## Small Memory Footprint

A small memory footprint is becoming important, again, due to the advent of SoCs, FPGAs, and 32-bit processors having significant on-chip memory. These are bringing back the need for highly efficient memory utilization because using only on-chip memory is cheaper and much faster than using off-chip memory. Using only on-chip memory not only reduces component cost, but also permits using slower processors, which produces 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. This has been the domain of 8 and 16-bit processors, but now is being challenged by 32-bit processors. SRAM requires about 8 times the chip space of flash memory. Hence, minimal RAM usage is more important than minimal flash usage, although both are important.

***smx* was architected to encourage applications with large numbers of tasks. More tasks means more use of *smx* code and less use of new unproven application code. Fewer bugs translate into faster project completion, lower development cost, and fewer headaches.**

## Stack RAM Reduction

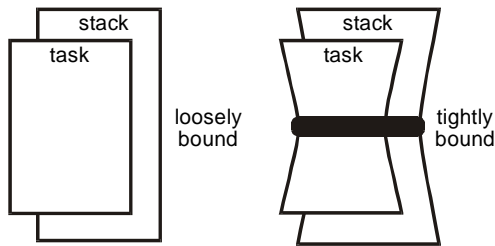
In general, every task requires its own stack. Depending upon the amount of function nesting, auto variable usage, ISR nesting, and other factors, task stacks can be quite large. Hence, systems with large numbers of tasks may require very large amounts of

RAM for their stacks. For best performance, stacks should be in fast RAM. This is especially true if autovariabes are used extensively to achieve reentrancy.

In RAM-constrained designs, this can result in the unfortunate tradeoff of using fewer tasks than is optimum for the application. When this happens, some of the benefit of multitasking is lost because operations that could be handled by kernel services between tasks become internal operations within tasks and therefore application code must be created to do them. *smx* provides three methods for reducing stack RAM:

(1) **One-Shot Tasks.** *smx* offers a unique *stack-sharing* capability for *one-shot* tasks. A one-shot task is a task that runs once, then stops. When it stops, it is done with the current pass and it has no information to carry forward. Hence, it does not need a stack to retain information for its next run. *smx* permits such tasks to be created and started without a stack. When dispatched, a one-shot task is given a temporary stack, called a *loosely bound* stack, from a *stack pool*. When the task stops, it releases the loosely bound stack back to the stack pool.

*smx* also supports normal tasks, which have *tightly-bound* stacks. These stacks 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, whereas a normal task retains its stack. Tasks may be either stopped or suspended when waiting for messages, signals, events, timeouts, etc. Thus, stopping vs. suspending a task introduces no constraints on what the task can do. It can, for

example, stop at a semaphore and wait for the next signal.

When a one-shot task is restarted, it is given a new stack from the stack pool, and it starts from the beginning of its code. There is no performance penalty for *starting* a task, with a new stack, versus *resuming* a task that already has a stack. If no stack is available, the one-shot task waits for the next tick.

One-shot tasks are good for functions that seldom run. Why tie up a large block of precious SRAM for a task that is idle almost all of the time? Examples are tasks that handle exceptional conditions or requests, infrequent operator input, infrequent downloads, etc. One-shot tasks are also good for mutually exclusive tasks that cannot run simultaneously (e.g. a task the starts up an engine versus a task that shuts down the same engine.)

As opposed to other mechanisms for sharing stack space (e.g. OSEK BCC1), the *smx* approach has two significant advantages:

- (1) Any mixture of one-shot and normal tasks may run simultaneously.
- (2) one-shot *smx* tasks can wait at semaphores, exchanges, etc. when stopped.

Because of these two differences, *smx* permits more optimal RAM vs. functionality tradeoffs to be made, than OSEK.

If we make the rule, like OSEK BCC1, that one-shot tasks may 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 are 10 one-shot tasks having 3 different priorities, then only 3 stacks are required in the stack pool.

(2) **LSRs:** Functions, which might be performed by small tasks, can be performed by LSRs, instead. Examples would be functions invoked by timers and ISRs. This saves stack space because LSRs do not require their own stacks.

(3) **Dynamic tasks:** *smx* permits all objects, including tasks, to be dynamically created and deleted. A stack

is allocated from the *smx* heap when a normal task is created and freed to the heap when the task is deleted. Task creation and deletion are fast operations in *smx* (although not as fast as starting and stopping one-shot tasks). Hence deleting unneeded tasks to reduce stack RAM usage may be acceptable in many systems. However, it is important to note that deleted tasks cannot directly wait for events, like stopped or suspended tasks. In addition, they drop off the radar relative to debugger awareness, whereas stopped tasks do not.

## Efficient Heap Usage

*smx* provides *heaps* as well *block pools*. Heaps are generally frowned upon for embedded, real-time systems because block allocation is non-deterministic — it becomes slower as heap fragmentation increases. 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, and more blocks must be allocated than actually needed, most of the time. By offering both heaps and block pools, *smx* allows the designer to choose an optimum combination for his application.

In embedded systems, many blocks are permanently allocated at startup. Heaps are good for these because each block can be exactly the right size. Fragmentation is not a problem because these blocks are never released. In addition, they cause no increase in allocation times because *smx* always starts searching at the first free block, which is after them.

## Exact Message Sizes

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 thus messages in the message pool can be much smaller.

## Other RAM 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 thus must have all functions present.) For CISC processors, *smx* code footprint varies from about 11 to 37 KB; for RISC processors, it is about 17 to 52KB.

## Safety & Reliability

Concern about RTOS safety and reliability is growing in importance due to greater system complexities and shorter development schedules. Embedded software is unlikely to be bug free in most new products. Now it is a question of how many bugs are in the code, what consequences they have, and how quickly they can be found and eliminated. This leads to the following RTOS concerns: How safe is it? 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 RTOS help the programmer to find errors? Is it designed to be *rugged* or is it *fragile*?

The right answers to these questions are obvious, but producing an RTOS with those right answers is difficult. As always in OS design, there are tradeoffs with performance and size. The more error checking that is performed, the larger and slower is the code. Processors may be faster and memory may be cheaper, but careful judgment is still necessary to decide what to add and what to omit.

We believe that we have achieved a good balance between speed, size, and safety for *smx*. The following text discusses the safety and reliability features built into it.

## Error Detection and Management

**Error Detection:** The *smx* error manager monitors about 80 error types (see chart in the *smx* data sheet). 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 invoked via a jump table. The ESR records the error type, the time of occurrence, and the task or LSR in which it occurred. This information is put into the *error buffer (eb)*. The ESR also increments a counter for the error type and, if the error occurred in a task, loads an error number into the *errnum* field in the task's TCB. ESRs are user-extensible and can be augmented to perform additional functions specific to an error type.

**Error Buffer:** The error buffer, *eb*, is allocated from the *smx* heap. Its size is user-controlled and can be small or large (e.g. up to thousands of records). *eb* can be viewed as an array of records via a debugger. It can also be viewed symbolically using *smxProbe* (see *smxProbe* data sheet for details.)

**Error Management:** *smx* supports both *point-of-call* and *centralized* error management. *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 */
```

Error action code can access *ct->errnum* to determine what action to take (*ct->errnum == 0* implies a timeout). Point-of-call error management provides the most precise error management because it is in the context of the error. However, too much of it may bloat and complicate the code. For this reason, centralized error management may be better for most errors.

Centralized error management is performed by ESRs, as discussed above. Rather than writing point-of-call error handling code, the user may customize ESRs to deal with errors, centrally.

Hence, one routine can deal with all errors of a single type or more than one type. Although errors cannot be handled as precisely, this way, the code required is much less. ESR source code is included in the *smx* development kit. It can easily be extended for application error handling.

## Task Timeouts

Every *smx* service call that causes a task to wait can be timed out. Timeouts are for safety, not for timing. They insure that tasks do not wait indefinitely for an event that failed to occur or was missed. Each task has a timeout register that records the time by which it should have been resumed or restarted. All non-zero task timeout registers are periodically compared to the *elapsed time (et)*. If less than *et*, the corresponding task is resumed or restarted. This operation is performed by the *timeout task*, which can be set to run at an appropriate priority and frequency. The timeout mechanism is designed to cause minimal processor overhead, as is appropriate for a safety feature.

## Safety Valves

LSRs naturally operate like *safety valves* and many *smx* applications have used them in this manner. If the processor is not fast enough to handle peak interrupt loads, LSRs can smooth out the peaks. This is due to a unique LSR feature that, unlike a task or an ISR, an LSR can be invoked multiple times before running. An LSR can be interrupted repeatedly by ISRs that invoke it or other LSRs. Each invocation can be passed a unique parameter such as a timestamp, a datum, a pointer, etc. which serves to make the invocation unique. When the peak load has passed, each invocation of the LSR executes, in order. Therefore, LSRs preserve *temporal integrity* — i.e. events do not get out of order.

Despite industry focus on *deadlines*, *rate monotonic analysis*, etc., maintaining temporal integrity may be sufficient for most systems. The control system will continue operating smoothly but may become a bit sluggish. This is not necessarily a bad thing since it tends to dampen over-shoot and ringing, and it could help to maintain control in extreme situations.

## Unblockable LSRs

Unlike tasks, which can be preempted by higher priority tasks or blocked by lower priority tasks, LSRs are not subject to delay by any tasks. They are immune to *priority inversion* because they cannot wait on events. LSRs are simple creatures, which, except for interruption by ISRs, are undeterred in

doing their jobs. This simplicity is often exactly what is needed for safety-critical and time-critical functions.

## Safe Messaging

Most kernels offer rudimentary messaging in which pointers to blocks are passed via *message queues*. 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.

There are a number of problems with this technique:

- (1) *Lack of safety*. The receiving task has no way to verify that it has received a valid pointer to the start of a message. The pointer might point anywhere, including the middle of the message. Hence, not only would wrong information be processed, but also information could be written anywhere in memory, and the block would not be correctly returned to its pool. These are common C pointer problems.
- (2) *No message priority* mechanism to permit more important messages to be processed first.
- (3) *Message size is not specified*. Some other method must be used to tell the receiving task the size of the message it has received.
- (4) *Reply address is not specified*. This creates a problem for server tasks to know where to send replies for client tasks
- (5) *No owner is specified*. 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++.
- (6) *No indication of what pool* to return a message block, when it is no longer needed.

Application code can compensate for these deficiencies, but so doing is error prone and reduces independence between tasks because receiving tasks must know more about the system aspects of the messages that they are processing.

*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 waiting MCBs or TCBs.) A message handle is a pointer to a message's MCB. This pointer is verifiable because it must point to a known structure with the MCB type in its *cbtype* field. *smx* 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 pool handle in the MCB, not by a message pointer.

Messaging errors undermine the goal of task independence. If each receiving task is able to at least verify that it has been given a valid message pointer, then propagation of errors is reduced and localizing errors to the task of origin is made easier. If not, 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. *Passing raw pointers from task to task, a common practice in other kernels, is even more risky!*

## Robust Heaps

*smx* heaps are designed to be more robust than those supplied with C compilers. *smx* heap calls are *task safe*, which is not true of most compiler heap calls. Also, allocated blocks are separated by *Heap Control Blocks (HCBs)* rather than by 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.)

A big problem with heaps is that a block overflow will usually damage the next block pointer, thus destroying the heap from that point up. The fences in *smx* HCBs are fixed bit patterns, like 0x55555555 that 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. Since HCBs are doubly linked, it is possible to work down from the top to save the rest of the heap.

## Stack Guard Bands

Similarly to heap fences, *smx* places known patterns, called *stack guard bands*, at the top and bottom of each stack. These are checked whenever a task is suspended or stopped, in order to detect stack overflow or underflow.

## Proven Code

A final comment is in order on safety and reliability. All commercial embedded software, like *smx*, that has been in wide use for many years and has had bugs fixed as they appeared, has a major advantage over new application code – it is *proven* code. Not only has it been tested once in one environment, but also it has been tested many, many times in all kinds of environments. It is about as close to bug free as is humanly possible. As such, it is considerably more valuable, line-by-line, than new code. Best results are achieved if embedded developers use proven code as much as possible and new code as little as possible. Hence, a full-featured kernel, like *smx*, is a better choice for safety and reliability than is a minimally featured kernel.

**Mechanisms to achieve safety and reliability are of less value if a kernel is difficult to use properly. *smx*'s ease of use is discussed next.**

## Ease of Use

Embedded systems are steadily increasing in complexity and projects are coming under increasing time-to-market pressure. It therefore is important to get off to a quick start and to have a kernel that is easy to understand and use. *smx* offers many features to make the programmer's job easier.

## Processor-Specific Support

*smx* reduces development schedule and cost by offering pre-integration with popular processors and tools. It is shipped with full support for specific processor evaluation boards and tool suites. This permits getting started very quickly and doing

software development while the custom board is being designed. See [www.smxrtos.com/eval](http://www.smxrtos.com/eval) for the current list of supported evaluation boards, processors, and tool suites.

The *smxBSP* package that is included with *smx* provides the processor-specific support for *smx* and other *smx* products and for application software. When the custom board is finally ready, it is necessary only to make small changes to *smxBSP* in order to move over to it. See the [smxBSP data sheet](#) for more information on the services that it provides and on porting it to other hardware.

For more information on specific processor support, see the *smx* data sheet for the processor family of interest:

- [smxARM](#). *smx* ARM kernel is targeted at all ARM7 and ARM9 processors.
- [smxCF](#). *smx* ColdFire kernel supports all ColdFire processors.
- [smxPPC](#). *smx* PowerPC kernel supports embedded PowerPC processors.
- [smx86](#). *smx* x86 kernel supports all x86 processors. It supports both real mode and protected mode.
- [psmx](#). *smx* portable kernel can easily be ported to other 32-bit processors.

## Tool Suite Support

*smx* offers in depth support for the following tool suites:

- CodeWarrior for ColdFire and PowerPC.
- IAR for ARM.
- Diab/VisionCLICK for PowerPC
- Paradigm C++, Borland C++, and Microsoft C++ for x86

Each *smx* delivery includes project files, or batch files, and help notes for the above compilers to build libraries for all delivered *smx* products and the *smx* Protosystem (see below). *smxAware* and the Graphical Analysis Tools support all of the above debuggers, including several x86 debuggers. In addition, Micro Digital provides *first line support* for the above tools. This means that if you have a problem getting started, our help team will help you

regardless of whether the problem is with *smx* or with the tools.

## 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 are added for other *smx* products (e.g. *smxFS*, *smxNet*, etc.), if present. Full source code is provided for the Protosystem and for each demo. Demos provide helpful examples and useful code to copy for the application.

A single project file, or make file, builds the Protosystem for a particular combination of *smx* products. Running the combined Protosystem and demos provide initial confidence tests that all delivered *smx* products are working correctly. To develop an application, it is necessary to exclude the demo modules and to add new modules for the application.

## Natural API

The *smx* API features symmetry, orthogonality, and limited parameters per function. These all help to make *smx* easier to use. See the *smx* data sheet for more information on this.

## Helpful Manuals

We believe that good manuals are one of the most important things that set a commercial kernel apart from an in-house kernel. *smx* manuals are complete, accurate, and well written. We make a strong effort to keep them up to date and written clearly. The manuals, shipped with *smx*, are as follows:

**Quick Start:** Introductory material and cookbook-style instructions 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*. The purpose of the Quick Start manual is to get started with a minimum of reading.

**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 and detailed glossary. Together, these two sections define *smx* completely. Each *smx* service 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 in length.

**Target Guide:** This manual discusses CPU and Tool issues for all 32-bit processors. It also documents the BSP API and other low-level API's. The BSP information discusses how to implement the BSP functions to support a new processor. (For information about porting to a new processor family, see the *smx Porting Guide*.)

**smx Porting Guide:** Complete documentation for *portable smx (psmx)*. Covers porting *psmx* to a new CPU family and/or a new compiler. To port to another CPU in a supported family, see the *smx Target Guide*, instead. To examine the *smx* manuals, ask for access to our restricted website.