

**smx[®]**

Simple Multitasking Executive

smx is a hard real time multitasking kernel for embedded systems. It can be used alone or with other components of the SMX RTOS. It supports ARM, ColdFire, PowerPC, x86, and it is portable to other processor families.

smx has been commercially available for over 20 years and has been used in over 500 applications. During this time, it has evolved into a reliable, robust, and capable kernel. It is the optimum choice for demanding, trouble-free embedded systems. smx is delivered with the *Protosystem*, which contains sample code and a foundation to build on.

smx has many unique features to reduce product cost, provide safe and reliable products, and to assure quick project starts and on-time deliveries. See the *smx Features* data sheet for discussion of these features. The discussion in this data sheet provides an overview of smx and an introduction to its API. See the end of this data sheet for a summary of the smx API.

smx offers full support for popular tool suites. smxAware, which is available separately, provides kernel-aware debugging and graphical analysis tools that work with many popular debuggers. smxBSP, which is included with smx, provides turnkey processor support for many processor types. smx is ROM'able. It is one of the few kernels to provide a C++ class library, smx++.

The smx Concept

The smx concept is to keep its API simple, but to also offer a richness of features from which the application programmer can draw. Some RTOS

kernels are too simple. While this may make them easy to learn, the net result is that the application ends up being more complex than it should be. This is because *what is not in the kernel ends up in the application*. Unused smx functionality is not linked into the code. Hence, smx's richness of features is not a memory burden, but rather, allows the application programmer to customize smx to his needs. Some RTOS kernels are too complex, which makes them difficult to use correctly. smx has aimed for the middle ground and has successfully achieved a good balance between simplicity and richness of features. smx packs considerable capability into its natural, easy-to-use API.

Natural APIs are easier to learn and to use correctly. Two important elements of such APIs are *symmetry* and *orthogonality*. These were primary concerns in the smx architecture. In an RTOS, symmetry means that what can be done can be undone. For example, smx offers a *delete* function for nearly every *create* function; it offers a *start* function for every *stop* function, a *resume* function for every *suspend* function, etc. Orthogonality, in an RTOS, means that kernel functions operating on tasks do not depend upon which task or the task's state. For example, deleting an smx task has the same result, regardless of its state. This sounds easy enough, but consider that if the task is in the run state, it is actually deleting itself! (Interestingly, a task deleting itself is actually useful — thus reinforcing 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 or five, at the most. However, some APIs have 10 or more parameters per function. Such functions are difficult to understand and to use properly.

Each of the following sections provides a brief discussion of a functional area of the smx API.

Task Management

smx permits *creating* and *deleting* tasks. Tasks may be created with or without stacks. The former are called *normal tasks*; the latter are called *one-shot tasks*. One-shot tasks can share stacks, thus saving RAM. (See *smx Features* for more information on this.) Once created, tasks can be directly *started* or *stopped* and *resumed* or *suspended*. Tasks can be *bumped* to the end of the current priority level or to a new priority level. Tasks may be *locked* against preemption and then *unlocked*. Tasks can be created to start locked or unlocked.

smx allows *exit routines* and *entry routines* to be hooked into the suspend and resume processes of the task scheduler. The hooked exit and entry routines are task-specific. They allow extending the context saved when a task is suspended and restoring it when the task is resumed, in a manner transparent to the task. Examples include saving and restoring: coprocessor registers, memory bank registers, and global variables.

Memory Management

smx supports two types of block pools. The first type, the *raw block pool*, is allocated from the heap and is primarily used by smx to allocate control block pools and by smx++ to allocate classes of objects. It provides raw blocks linked into free lists. Functions are provided to *create* raw block pools, *get* N contiguous blocks, *sort* a raw block free list, and *release* a raw block back to its pool. There is no protection from errors. Speed is maximized. Raw blocks are appropriate for proven code such as smx. They can be used for application code, but something safer may be more appropriate.

smx also offers *normal block pools*, in which each block is controlled by a *Block Control Block (BCB)*. A BCB contains a pointer to the start of the data block, the block's current owner, and the pool it came from. Functions are provided to *create* block pools

from either a *DAR* or the heap. A *DAR* is a *Dynamically Allocated Region* of memory to which the programmer has assigned beginning and ending addresses. Heap pools can be *deleted*, but not *DAR* pools. Functions are also provided to *get* and *release* blocks. Blocks owned by a task are automatically released if it is deleted, thus preventing memory leaks.

smx also supports a heap. *malloc* and *free* functions are provided as well as *initialize*, *realloc*, *check*, and *set* functions. The smx heap is safe for multitasking and provides protection against overflows and leaks. See *smx Features* for more discussion.

Timing and Timers

smx maintains *elapsed time* in ticks and *system time* in seconds. Elapsed time is used for task timeouts, and system time is used for task *sleep*. Functions are provided to *get* either time.

smx implements versatile timers. When *started*, an smx timer is created and enqueued in the *timer queue(tq)*, for the number of ticks specified. When a timer times out, it invokes an LSR with a specified parameter. (An LSR is a service routine that is in between an ISR and a task. See *smx Features* for a detailed description.) If the timer is a *one-shot* timer, it self-destructs. If the timer is a *cyclic* timer, it immediately requeues itself in the timer queue, with no loss of ticks. (Hence, there is no cumulative error.) Directly invoking LSRs assures low jitter because LSRs run ahead of all tasks. Due to the way timers are enqueued in tq, there is no operational overhead after the first timer. Hence, a large number of timers can be used in a system.

Input and Output

Easy I/O Integration

It is difficult to create device drivers for many OSs (e.g. Linux) because they must be linked with the kernel, separately from the application. In addition, they must emulate a file I/O interface to the application. This is not required by smx. Under smx, device drivers are part of the application and can have an appropriate interface for what they do. The only

impact that smx has upon device drivers is that for ISRs: (1) smx calls are not permitted by ISRs. Instead, they must *invoke* LSRs to make smx calls, later. (2) ISRs invoking LSRs must be bracketed with *ISR enter* and *ISR exit* macros.

Pipes

It is customary to create *ring buffers* to hold incoming and outgoing character streams. *Pipes* provide a method to interface ring buffers to tasks — this is not simple because there can be access conflicts between ISRs and tasks. A pipe contains a ring buffer. It can be of any length. High-speed functions are available for ISRs, LSRs, and tasks to put characters into and get characters out of pipes. In addition, smx calls are available to do the same thing, but they allow tasks to wait on full outgoing pipes or empty incoming pipes. The pipe macros and functions are interrupt-safe. Hence, pipes can serve as character conduits between serial device drivers and tasks.

smx provides the necessary functions to *create* and *delete* pipes and the necessary functions to *put* or *get* characters to or from them.

Intertask Communication

Messages

smx provides functions to *create* a DAR message pool or to *create* or *delete* individual messages from the heap. A message consists of a *Message Control Block (MCB)* and a data block for the actual message. The MCB contains a pointer to the start of the data block and message control information. Use of MCBs provides safe messaging (see *smx Features* for details.)

Message Exchanges

smx messages are *sent* to and *received* from *exchanges*. An exchange is an smx object that enqueues either messages or tasks, whichever is more abundant. The use of exchanges has important advantages over the direct task-to-task messaging used by many kernels:

- *Anonymous receivers*. The receiving task's identity is not hard-coded into the sending task's code. The sender simply sends to a

known exchange. Thus, it easy to *swap* receiving tasks without altering sending tasks. Task swapping can be useful for handling different product versions or different installation configurations. It can even be done dynamically to handle changing circumstances (e.g. changing from startup mode to operating mode.)

- *No limit on the number of waiting messages*.
- *Work queues*. A message queue at an exchange is a natural work queue for a server task that is receiving messages from the exchange.
- *Token messages*. A message can be used as a *token* to control access to a resource and to simultaneously provide needed information about that resource (e.g. i/o port numbers needed to access it).
- *Multiple message queues* are easily implemented via multiple exchanges. Rather than searching through a single queue for the right message type, as some kernels do, a task simply checks for messages at the appropriate exchange.

Three types of exchanges can be *created*: *normal*, *priority pass* (see below), and *resource* (used for message pools.) Exchanges can be *deleted*.

Message Priorities

smx messages have priorities. Exchanges can bypass high-priority messages around 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 message. This allows a client task to pass its priority to a server task via the message it sent. Doing this is especially useful for server tasks that are accessing resources. Being able to adjust their priorities permits them to operate as extensions of their client tasks. This is preferable to the client tasks directly accessing resources, because then priority inversions can occur.

Semaphores

smx provides *counting* and *binary* semaphores. The classical use of a counting semaphore is to control

access to N resources. The internal count is started at N and decremented each time a task *tests* the semaphore. The first N tests pass, but each subsequent test requires a *signal* before it will pass. Hence, only N tasks can be running, at once. smx has added an internal *threshold* to enable a counting semaphore to also count off multiple events (signals) per action. Each signal increments an internal counter. Each time the count reaches an internal threshold the first waiting task is allowed to resume.

Binary semaphores are used in *producer/consumer* activities. A binary semaphore can have values of 0 and 1, only. A *signal* changes it to 1; a *test* changes it to 0. As a consequence, the producer can signal the semaphore any number of times, but the first time the consumer tests it, it will change back to 0. Thus, the consumer can then accept all items the producer has produced, without retesting the semaphore each time. When done it will test the semaphore and wait for the next signal(s).

Mutexes

For systems requiring greater safety than can be provided by semaphores, smx offers *mutexes*. Mutexes are safer than semaphores for the following reasons:

- Nested testing by the same owner is permitted, without task lockup.
- Spurious releases have no effect, if not owned. Can be released only by the owner.
- Priority promotion of the owner is automatic.

The smx mutex implementation is very powerful and complete. It provides both *priority ceiling* and *priority promotion inheritance*. For the latter, priority propagation to other mutexes and owners is supported. For more information, see the smx Mutex Tech Note.

Event Management

Event Queues

An *event queue* (*eq*) permits tasks to simultaneously wait for specified numbers of *events* (signals). Tasks are enqueued differentially so that only the counter of

the first task is decremented with each *signal*. This minimizes overhead, thus allowing any number of tasks to wait for events of a give type. Functions are provided to *create* and *delete* event queues, *count* a specified number of events, and *signal* events. An example of an event queue is *smx_TicksEQ*, which permits a task to wait for a specified number of ticks.

If no task is waiting, events are not saved. This is unlike the counting semaphore, which records all events.

Event Flags Groups

Event flags groups permit waiting on multiple events. They allow tasks to wait for AND or OR combinations of up to 31 flags. Functions are provided to *create* and *delete* event flags groups, to *set* and *reset* flags and to *test* flags. When testing flags a task provides a mask indicating which flags to test and whether to AND or OR them together. Multiple tasks can wait on the same or different combinations of flags. When a combination occurs, all tasks waiting on it are resumed. Before waiting again, the flags usually must be reset, depending upon the application.

Miscellaneous

All smx objects are identified by *handles*. These are 32-bit unsigned integers that are returned by create functions. smx has a *handle table*, *ht*, to convert handles to ASCII names and vice versa. It is primarily used by smxAware. Create/Delete functions can add/remove entries to/from *ht*. Functions are provided to get a handle or a name from *ht*.

A macro to inhibit LSRs and a function to reenble them is also provided. These are needed to protect resources that are shared between LSRs and tasks. Also included in miscellaneous functions are queue management.

The following is a summary of the smx API. For full details, please download the smx Reference Manual from www.smxrtos.com.

smx API

Task Management

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

smx_TaskBump(task, pri)
smx_TaskCreate(code, pri, stack_size, flags, name)
smx_TaskDelete(*task)
smx_TaskHook(task, entry, exit)
smx_TaskLocate(task)
smx_TaskLock()
smx_TaskLockClear()
smx_TaskResume(task)
smx_TaskSetCopro(task, *state)
smx_TaskSetStackCheck(task, *state)
smx_TaskSleep(time)
smx_TaskSleepStop(time)
smx_TaskStart(task)
smx_TaskStartNew(task, par, code, pri)
smx_TaskStartPar(task, par)
smx_TaskStop(task, timeout)
smx_TaskSuspend(task, timeout)
smx_TaskUnhook(task)
smx_TaskUnlock()
smx_TaskUnlockQuick()

Memory Management

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

smx_BlockCreatePoolDAR(num_blks, blk_size, darp, name)
smx_BlockCreatePoolHeap(num_blks, blk_size, name)
smx_BlockDeletePoolHeap(*p)
smx_BlockFindNext(last, task)
smx_BlockFindPool(block)
smx_BlockGet(pool)
smx_BlockRelease(block)
smx_BlockReleaseAll(task)

smx_HeapCalloc(num, size)
smx_HeapCheck()
smx_HeapFree(*block)
smx_HeapInit()
smx_HeapMalloc(num_bytes)
smx_HeapRealloc(*block, num_bytes)
smx_HeapSet(fillval)
smx_HeapWalk(*heapentry)

smx_RawBlockCreatePool(mem_ptr, num, size)
smx_RawBlockGet(*flp, num, size, clr_sz)
smx_RawBlockRelease(bp, *flp, clr_sz)
smx_RawBlockSortPool(*flp)

Timing and Timers

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

smx_SysEtimeGet()

smx_SysStimeGet()

smx_SysStimeSet()

smx_TimerRead(tmr, *time_left)

smx_TimerStart(*tmr, time, interval, lsr, par, name)

smx_TimerStop(tmr, *time_left)

Input and Output

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

smx_ISR_ENTER()

smx_ISR_EXIT()

smx_LSR_INVOKE(lsr, par)

smx_LSRInvokeF(lsr, par)

smx_PipeCreate(*ppb, width, length, name)

smx_PipeDelete(*pipe)

smx_PipeGet8(pipe, *b)

smx_PipeGet(pipe, *pdst)

smx_PipeGetWait(pipe, *pdst, tmo)

smx_PipeGetWaitStop(pipe, *pdst, tmo)

smx_PipePurge(pipe)

smx_PipePut8(pipe, b)

smx_PipePut(pipe, *psrc)

smx_PipePutWait(pipe, *psrc, tmo)

smx_PipePutWaitStop(pipe, *psrc, tmo)

smx_PipeResume(pipe)

smx_PipeResume_F(pipe)

smx_PipeStatus(pipe, *ppss)

Intertask Communication

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

smx_MsgBump(msg, pri)

smx_MsgCreateHeap(blk_size, rxchg)

smx_MsgCreatePoolDAR(num_blks, blk_size, darp, rxchg, name)

smx_MsgDeleteHeap(*msg)

smx_MsgDequeue(msg)

smx_MsgFindNext(last, task)

smx_MsgFindPool(msg)

smx_MsgLocate(msg)

smx_MsgReceive(xchg, timeout)

smx_MsgReceiveStop(xchg, timeout)

smx_MsgReleaseAll(task)

smx_MsgSend(msg, xchg, *reply)

smx_MsgXchgCreate(type, *limp, name)
smx_MsgXchgDelete(*x)

smx_Pipe: See Input and Output section.

smx_SemCreate(thres, *limp, name)
smx_SemDelete(*sem)
smx_SemSignal(sem)
smx_SemTest(sem, timeout)
smx_SemTestStop(sem, timeout)

Mutexes

Mutexes protect resources and critical sections.

smx_MutexClear(mtx)
smx_MutexCreate(pi, ceiling, name)
smx_MutexDelete(*mtx)
smx_MutexFree(mtx)
smx_MutexGet(mtx, timeout)
smx_MutexRelease(mtx)

Event Management

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

smx_EventFlagsCreate(num_slots, name)
smx_EventFlagsDelete(*ef)
smx_EventFlagsReset(ef, nflags)
smx_EventFlagsSet(ef, nflags)
smx_EventFlagsTest(ef, mask, timeout)
smx_EventFlagsTestStop(ef, mask, timeout)

smx_EventQueueCount(eq, count, timeout)
smx_EventQueueCountStop(eq, count, timeout)
smx_EventQueueCreate(name)
smx_EventQueueDelete(*eq)
smx_EventQueueSignal(eq)

Miscellaneous

Special purpose calls.

smx_HTAdd(h, name)
smx_HTDelete(h)
smx_HTGetHandle(name)
smx_HTGetName(h)
smx_HTInit()
smx_LSRsOff()
smx_LSRsOn()
smx_QueueClear(queue)
smx_QueueSize(queue)
smx_SysPseudoHandleCreate()