

**smx<sup>®</sup>**

## 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 is portable to other processor families.***

*smx* has been commercially available for over 16 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 quick-start *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 and a table of detected error types.

*smx* offers full support for many 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 less, 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 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*. When first started, tasks are in the locked state to insure they will complete initialization, without interruption.

*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 *fast 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* fast block pools, *get* N contiguous blocks, *sort* a fast block free list, and *release* a fast block back to its pool. There is no protection from errors. Speed is maximized. Fast 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 *enter\_isr* and *exit\_isr* macros.

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

## Pipes

It is customary to create *ring buffers* to hold incoming and outgoing character streams. *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 macros 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.

## Buckets

Ring buffers, or pipes, are not a good solution for high-speed, packetized communication because they necessitate an extra copy step per character. *smx* provides an object, called a *bucket*, to solve this problem. Incoming characters are loaded into a bucket, just like a ring buffer. *smx* provides a macro for this. When a complete packet of characters, has been received (as determined by the receiving ISR), an LSR is invoked to move the packet, by reference (i.e. no copy), from the bucket to a normal exchange where it can be processed as an *smx* message. Going the other direction, a packet is moved, by an LSR or a task, from a normal exchange to a bucket. The bucket is emptied by a transmit ISR using an *smx* macro. When empty, the spent packet is recycled.

*smx* provides the necessary functions to *create* and *delete* pipes and buckets and the necessary functions and macros to *put* or *get* characters and messages to or from pipes and buckets. GET and PUT macros are safe for use from ISRs.

# 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 is 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 the *ticks* queue in *smx*, 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 Tables

Event tables 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 tables, 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*. Macros are provided to add (*build*) and remove (*unbuild*) entries from *ht* and to *get* a handle or a name from *ht*. Build and unbuild

are typically called after creating and deleting *smx* objects, respectively.

A macro to inhibit LSRs and a function to reenable 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, numeric conversion, time conversion, timing, and error display.

## *smx* Error Types

00	OK	28	OUT_OF_CXCBS
01	BAD_LIM	29	OUT_OF_MCBS
02	BROKEN_Q	2A	OUT_OF_PCBS
03	CANT_CREATE_CBLOCKS	2B	OUT_OF_QCBS
04	DOS_CRITICAL_ERROR	2C	OUT_OF_STACKS
05	EMPTY_POOL	2D	OUT_OF_TCBS
06	EVENT_TABLE_OVERFLOW	2E	OUT_OF_TMCBS
07	FHEAP_BLOCK_OVERFLOW	2F	RQ_ERROR
09	IDLE_TIMEOUT	30	SIG_CTR_OVERFLOW
0A	INSUFF_DAR	31	STACK_OVERFLOW
0B	INSUFF_FHEAP	32	WAIT_NOT_ALLOWED
0C	INSUFF_NHEAP	33	WRONG_CBTYPE
0D	INVALID_BCB	34	WRONG_TYPE_CXCHG
0E	INVALID_CB	35	WRONG_TYPE_QUEUE
0F	INVALID_CXCB	36	WRONG_TYPE_QUEUE
10	INVALID_ET	37	BLOCK_IN_USE
11	INVALID_FHEAP_ALIGN	38	TIMER_IN_USE
12	INVALID_FHEAP_BLOCK	39	HT_FULL
13	INVALID_MCB	3A	HOLDING
14	INVALID_NHEAP_ALIGN	3B	MESSAGE_NOT_SIZED
15	INVALID_NHEAP_BLOCK	3C	NOT_HOLDING
16	INVALID_PARM	3D	OBJECT_IN_USE
17	INVALID_PCB	3E	OBJECT_NOT_CREATED
18	INVALID_QCB	3F	WRONG_MCB_TYPE
19	INVALID_SBCB	40	USER_ABORT
1A	INVALID_SP	41	INIT_MODULES_FAIL
1B	INVALID_SS	42	HOST_HEAP_ALLOC_FAIL
1C	INVALID_TCB	43	CLIB_ABORT
1D	INVALID_TIME	44	INSUFF_SHEAP
1E	INVALID_TMCB	45	INVALID_SHEAP_ALIGN
1F	LQ_OVERFLOW	46	INVALID_SHEAP_BLOCK
20	MSDOS_CALL	47	SHEAP_BLOCK_OVERFLOW
21	NHEAP_BLOCK_OVERFLOW	48	BSS_NOT_OVERFLOW
22	NULL_PTR	49	INVALID_PRIORITY
23	NULL_PTR_REF	4A	OUT_OF_MUCBS
24	NULL_TEST_FLAGS_MASK	4B	INVALID_MUCB
25	OP_NOT_ALLOWED	4C	MUTEX_ALREADY_FREE
26	OUT_OF_BCBS	4D	MUTEX_NONOWNER_RELEASE
27	OUT_OF_BLOCKS	4E	LSRS_CANNOT_OWN_MUTEXES

# Complete smx API

## Task Management

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

- bump\_task** (task, new\_priority)  
Changes task priority and requeues *task*.
- copro** (ON/OFF, task)  
Enables/Disables coprocessor state saving for *task*.
- create\_task** (code\_ptr, priority, stack\_size)  
Creates a task with *task\_main* code with the specified *priority*, and binds a stack from the heap of size *stack\_size* to it. If *stack\_size* == 0, no stack is bound.
- delete\_task** (task)  
Deletes *task* and releases resources owned by it (including permanently bound stack).
- hook** (entry, exit, task)  
Hooks exit and entry routines to *task* which run when *task* is suspended/resumed.
- IS\_LOCKED** (task)  
Determines if *task* is locked.
- locate** (task)  
Locates the queue which a task is in.
- LOCK** (task)  
Locks *task* against preemption.
- resume** (task)  
Dequeues *task* from any queue it may be in and puts it into the ready queue at the end of its priority level.
- STACK\_CHECK** (ON/OFF, task)  
Enables/Disables stack checking for *task*.
- start** (task)  
Puts *task* into the ready queue at the end of its priority level. Stops it first, if necessary.
- start\_new** (task, code\_ptr, priority)  
Restarts a task with new code and priority. Clears HOOKED flag.
- start\_par** (task, par)  
Starts *task* and passes *par* to it.
- stop** (task, timeout)  
Dequeues *task*, releases its stack if not a permanently bound stack, and sets its timer to restart after *timeout*.
- suspend** (task, timeout)  
Dequeues *task* and sets its timer to resume after *timeout*.
- unhook** (task)  
Unhooks exit and entry routines from task.
- unlockx** (task)  
**UNLOCK** (task)  
Unlocks *task* so it can be preempted by higher priority tasks.

## Memory Management

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

- callocx** (num, size)  
Allocates space for an array of *num* elements of *size* bytes from the heap.
- create\_dpool** (rxchg, num\_blks, blk\_size, darp)  
Creates a block pool or a message pool in a dynamically allocated region (dar).
- create\_fbpool** (mem\_ptr, num, size)  
Creates a fast block pool of *num* blocks of *size* bytes at *\*mem\_ptr* in the heap. Blocks are linked to *mem\_ptr*.
- create\_hpool** (num\_blks, blk\_size)  
Creates a block pool in the heap.
- delete\_hpool** (pool)  
Deletes any block pool or message pool that was allocated from the heap.
- find\_next** (last, task)  
Finds the next block or message in a pool which is owned by *task*. Starts looking at *last* block.
- find\_pool** (blk\_or\_msg)  
Finds the dar or heap pool to which *blk\_or\_msg* belongs.
- freex** (block)  
Frees a block previously allocated from the heap.
- get\_block** (pool)  
Gets a block from a dar or heap pool.
- get\_fblocks** (flp, num, size)  
Gets *num* physically adjacent blocks from fast block pool linked to *flp*.
- heapchkx** ()  
Checks heap integrity.
- heapinix** ()  
Initializes the heap.
- heapsetx** (fill\_char)  
Loads *fill\_char* into all unused bytes in the heap and checks heap integrity.
- heapwalkx** (heap\_entry)  
Gathers information about the next block of the heap and puts it into the HEAPINFO structure.
- mallocx** (num\_bytes)  
Allocates a block of at least *num\_bytes* from the heap.
- reallocx** (block, num\_bytes)  
Reallocates an existing block larger or smaller from the heap.
- rel\_all\_blocks** (task)  
Releases all blocks owned by *task*, except its stack block.
- rel\_block** (block)  
Releases *block* back to its dar or heap pool.
- rel\_fblock** (bp, flp)  
Releases block back to a fast block pool.

**sort\_fblocks** (flp)

Sorts fast block pool to maximize contiguous blocks.

## 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.*

**get\_etime** ()

Gets current elapsed time (ticks).

**get\_stime** ()

Gets current system time (seconds).

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

Returns time left for *tmr*.

**sleepx** (time) or

**sleep\_stop** (time)

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

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

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

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

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

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

Stops *tmr* and returns time left.

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

## Input and Output

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

**create\_cx** (type)

Creates a bucket or pipe.

**delete\_cx** (cx)

Deletes a bucket or pipe.

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

Used to begin ISR.

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

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

**GET\_CHAR** (bucket)

Gets next character from *bucket*.

**get\_msg** (cxchg)

Gets a message from a bucket or a pipe.

**invoke** (lsr, par)

**INVOKE** (lsr, par)

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

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

Returns the number of characters in *pipe*.

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

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

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

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

Gets *char* from *pipe*.

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

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

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

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

Puts *char* into *pipe*.

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

Puts *char* into *bucket*.

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

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

## Intertask Communication

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

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

Changes message priority and requeues message.

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

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

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

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

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

Creates a message exchange of the specified type with one or more task priority levels and one or more message priority levels.

**delete\_hmsg** (msg)

Deletes a message in the heap.

**delete\_sem** (sem)

Deletes a semaphore.

**delete\_xchg** (xchg)

Deletes an exchange.

**dq\_msg** (msg)

Removes *msg* from a message queue.

**locate** (msg)

Locates the queue which a message is in.

**receive** (xchg, timeout)

**receive\_stop** (xchg, timeout)

Gets a message from *xchg*. Suspends or stops current task for *timeout* if *xchg* is empty.

**rel\_all\_msgs** (task)

Sends all messages owned by *task* back to their resource exchanges.

**sendx** (msg, xchg, reply)

Sends a message to an exchange. Delivers *msg* to the top

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

**signalx** (sem)

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

**test** (sem, timeout)

**test\_stop** (sem, timeout)

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

## Mutexes

**create\_mutex** (pi, ceil)

Creates mutex with or without priority inheritance and priority ceiling.

**clear\_mutex** (mtx)

Frees mutex and clears its wait queue.

**free\_mutex** (mtx)

Releases mutex from any task, regardless of nesting count.

**get\_mutex** (mtx, timeout)

Gets mutex, if free; increments nesting count if current task owns it; else waits.

**rel\_mutex** (mtx)

Releases mutex if current task owns it and nesting count is 1, else decrements nesting count.

## Event Management

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

**count** (count, eventq, timeout) or

**count\_stop** (count, eventq, timeout)

Suspends or stops current task on event queue for *count* number of events.

**create\_eq** ()

Creates an event queue.

**create\_et** (num\_slots)

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

**delete\_eq** (eq)

Deletes an event queue.

**delete\_et** (et)

Deletes an event table.

**reset\_flags** (et, flags)

Resets specified flags (1 bits in *flags*) in event table.

**set\_flags** (et, flags)

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

**signalx** (eq)

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

**test\_flags** (et, mask, timeout) or

**test\_flags\_stop** (et, mask, timeout)

Suspends or stops the current task on event table until the AND or OR of specified flags (1 bits in *mask*) is true.

## Miscellaneous

### *Special purpose calls.*

**BUILD\_HT** (handle, name)

Adds entry to handle table.

**clear\_q** (queue)

Clears *queue*. Moves tasks to ready queue or messages to resource queues.

**create\_rq** (num\_levels)

Creates the ready queue with the specified number of priority levels.

**ENTER\_SSR** ()

Use to begin an ssr. Makes a function into a system service routine (i.e. a restricted *smx* call).

**exit\_ssr** (retval)

Use to end an ssr. Makes a function into a system service routine (i.e. a restricted *smx* call).

**go\_smx** ()

Initializes *smx* from information in the configuration table.

**get\_handle**(name)

Gets the handle for a name.

**get\_name**(handle)

Gets the name for a handle.

**LSRS\_OFF** ()

Inhibits LSRs from running.

**lsrs\_on** ()

Reenables LSRs and runs any that are waiting.

**qsize** (queue)

Finds the number of objects in a queue.

**SMX\_DISPLAY\_ERROR** (errstr)

Displays error string to screen, serial port, or other output device.

**UNBUILD\_HT** (handle)

Removes entry from handle table.

\\server\smxd\lit\prod\smx.doc 12/28/05