smx®

Superior Multitasking Executive

smx is an advanced RTOS kernel, which offers unique features to speed application development and to make debugging easier. It packs rich functionality into a relatively small size and it offers high performance and safety.

smx has been commercially available for over 25 years and has been used in hundreds of applications. During that time, it has evolved into a reliable, robust, and capable kernel, and it continues to evolve to match the rising expectations for embedded software. The v4.3 release is the latest step in this evolution.

New v4.3 Features

New high-performance heap. It is a bin type heap, which overcomes the poor performance and indeterminacy of normal RTOS heaps. It uses algorithms similar to GPOS heaps, but tailored to the needs of embedded systems. Allocation performance approaches that of block pools.

Easy heap configuration via a single constants array. Heap consists of a small bins array that permits access by size, an upper bins array consisting of large and small bins, and a top bin for all larger chunks. The heap can easily be tuned to a specific application, such as one using object-oriented programming.

smx_HeapPeek(), smx_HeapSet(), smx_HeapBinPeek(), smxHeapBinSeed(), and smx_HeapChunkPeek() permit sophisticated heap monitoring and control.

Debug chunks help to find heap problems. They provide time and owner information and fences around data blocks. They can be freely mixed with inuse chunks, which have less overhead.

Self-healing heap is provided by automatic heap scan and heap fix functions, which run continuously during idle time.

Contents

smx_HeapExtend() permits extending the heap, while running, to addition space which can adjacent or not.

smx_TimerDup(), smx_TimerPeek(), smx_TimerReset(), and smx_TimerStartAbs() added to allow more flexibility in timer usage.

Pulse timer allows easily generating pulses for pulse width modulation (PWM), pulse period modulation

(PPM), and frequency modulation (FM), using the new smx_TimerSetPulse().

smx_SysPowerDown(mode) provides structure at the RTOS level to put the system into a specified power saving mode, and then performs tick recovery when power is restored. The latter is done in an efficient manner, which preserves correct timeout relationships between LSRs and tasks.

Error management improvement to make local error handling easier.

Cortex-M4 FPU support has been added for automatic state saving, to improve task switching time for tasks using the FPU.

smx API

The smx API provides simple services with few parameters that are easy to use. It minimizes restrictions on services, which helps to minimize usage errors. All functions for an smx object sort together, giving a clear view of what operations are possible for the object. The following sections summarize the smx API per object type.

Task Management

smx_TaskBump(task, pri)
smx_TaskCreate(fun, pri, stack_size, flags, name)
smx_TaskDelete(*task)
smx_TaskHook(task, entry, exit)
smx_TaskLocate(task)
smx_TaskLock()
smx_TaskLockClear()
smx_TaskPeek(task, par)
smx_TaskResume(task)
smx_TaskSetStackCheck(task, ON/OFF)
smx_TaskSleep(time)
smx_TaskSleepStop(time)
smx_TaskStart(task)
smx_TaskStartNew(task, par, pri, fun)
smx_TaskStartPar(task, par)
smx_TaskStop(task, tmo)
smx_TaskSuspend(task, tmo)
smx_TaskUnhook(task)
smx_TaskUnlock()
smx_TaskUnlockQuick()

Tasks may be created with or without permanent stacks. If stack_size is > 0, a stack is allocated from the heap. If stack_size == 0 a stack is allocated from the stack pool when the task is dispatched. The former are called *bound* tasks; the latter are called *unbound* or *one-shot* tasks. One-shot tasks share stacks from a stack pool; this reduces RAM needed for task stacks. Stop functions provided by smx allow one-shot tasks to give up their stacks while waiting for events. When the event occurs the task starts over and gets a new stack.

Once created, a task needs to be started in order to run the first time. After that, it can be suspended or stopped, then resumed or restarted. A task can be restarted as is or restarted with a new parameter and even new code and priority. A task can be bumped to the end of its current priority level or to the end of a new priority level. Hooking exit and entry routines into the scheduler allows extended task context to be saved and restored transparently when a task is suspended and resumed. This is useful for FPU and coprocessor registers.

Tasks may be locked against preemption and then unlocked, which is useful for short sections of non-reentrant code and for reducing unnecessary task switches. Tasks can sleep for long periods or be suspended for times accurate to a tick. Additional utility functions permit locating where a task is waiting, peeking at its parameters, and controlling the checking of its stack.

smxBase Memory Management

sb_DARInit(dar, pi, sz, fill, fillval)
sb_DARAlloc(dar, sz, align)
sb_DARFreeLast(dar)

sb_BlockPoolCreate(p, pool, num, size, name)
sb_BlockPoolCreateDAR(dar, pool, num, size, align, name)
sb_BlockPoolDelete(pool)
sb_BlockPoolPeek(pool, par)

sb_BlockGet(pool, clrsz)
sb_BlockRel(pool, bp, clrsz)

smxBase provides memory management services to non-task software and to smx, itself.

Dynamic memory can be divided into regions, called dynamic allocation regions (DARs). This has been done primarily to separate smx dynamic variables in SDAR from application dynamic variables in ADAR.

DARs are managed by smxBase. The DAR initialization function allows creating a new DAR.

The dar parameter is the handle of a statically-declared DAR control block, pi points to the starting address of the DAR block, and sz is its size. If fill is true, fillval will be written throughout the DAR.

Blocks are permanently allocated from DARs with specified sizes and alignments. These blocks are normally quite large and used for block pools, stack pool, heap, etc. The last allocated DAR block may be freed in the event that creating the object using it must be aborted.

Base block pool functions are implemented in smxBase (shown with sb_ prefixes). Base pools contain *bare* blocks for use by software such as ISRs and drivers. The sb_BlockPoolCreate() function creates a pool of num blocks of size bytes starting at p, which can point anywhere in RAM. sb_BlockPoolCreateDAR() creates a pool in the specified DAR with the specified alignment.

A pool can be deleted, but it is up to the user to reuse the memory left behind. A peek function returns pool information. Interrupt-safe block get and release services, with optional clearing are provided. These allow getting and releasing blocks from ISRs, drivers, etc.

smx Memory Management

smx_BlockPoolCreate(p, num, size, name)
smx_BlockPoolCreateDAR(dar, num, size, align, name)
smx_BlockPoolDelete(*pool)
smx_BlockPoolPeek(pool, par)

smx_BlockGet(pool, *bpp, clrsz)
smx_BlockMake(pool, bp)
smx_BlockPeek(blk, par)
smx_BlockRel(blk, clrsz)
smx_BlockRelAll(task)
smx_BlockUnmake(*pool, blk)

smx provides memory management functions to tasks that are *task-safe* (i.e. allow preemption).

Unlike base block pools, smx block pools are task-safe, have a more automatic API, and provide better protection from programming mistakes. Each block is linked to a Block Control Block (BCB), which contains a pointer to the start of the data block, the block's current owner, and the pool it came from.

Like base block pools, smx block pools may be created anywhere in memory or in a DAR. They also can be deleted, but it is up to the user to reuse the memory left behind. Services are provided to get and release blocks, with optional clearing. An smx block can be made from any bare block (e.g. a base block, heap block, or a static block) and can be unmade into a bare block. Blocks owned by a task are automatically released if it is deleted, thus reducing memory leaks. Peek services allow looking at block and block pool parameters.

Heap

smx_HeapBinPeek(binno, par)
smx_HeapBinSeed(num, bsz)
smx_HeapCalloc(num, size)
smx_HeapChunkPeek(cp, par)
smx_HeapExtend(xsz, xp)
smx_HeapFix(fp, num)
smx_HeapFree(bp)
smx_HeapGetCP(bp)
smx_HeapInit(sz, hp)
smx_HeapMalloc(sz)
smx_HeapPeek(par)
smx_HeapRealloc(bp, bsz)
smx_HeapRecover(sz, num)
smx_HeapScan(tp, num)
smx_HeapSet(par, val)

Heaps are normally not recommended for embedded systems due to poor, non-deterministic performance. However, the new smx heap is different. If properly managed, it can achieve allocation times that are deterministic and fast.

The smx heap provides the normal heap functions, calloc(), free(), malloc(), and realloc(). Macros and functions are provided to map these to the above, equivalent smx services. The smx heap services are task safe and provide numerous other advantages (see discussion in the New v4.3 Features, above).

Messaging

smx_MsgBump(msg, pri)
smx_MsgGet(pool, *bpp, clrsz)
smx_MsgMake(pool, bp)
smx_MsgPeek(msg, par)
smx_MsgReceive(xchg, *bpp, tmo)
smx_MsgReceiveStop(xchg, *bpp, tmo)
smx_MsgRel(msg, clrsz)
smx_MsgRelAll(task)
smx_MsgSend(msg, xchg)
smx_MsgSendPR(msg, xchg, pri, reply)
smx_MsgUnmake(pool, msg)

smx_MsgXchgClear(xchg)
smx_MsgXchgCreate(mode, name)
smx_MsgXchgDelete(*xchg)
smx_MsgXchgPeek(xchg, par)

smx messaging is exchange-based rather than pipe-based (aka *message queue* based) as with other RTOSs, although smx also provides complete pipe support (see the Pipes section).

Exchange messaging permits more sophisticated and reliable operation than passing bare block pointers via pipes. An smx message consists of a message control block (MCB) linked to a data block, which contains the actual message. The MCB contains the message block pointer, type, priority, owner, reply index, and return pool. Bundling this information with each message permits more automatic and reliable message handling. For example, when a task is deleted, all messages it owns are automatically returned to their pools. This reduces memory leaks.

MsgGet() gets a data block from the specified block pool and links it to an MCB from the MCB pool. It also loads a pointer to the data block into bpp, does an optional clear of clrsz bytes, and returns the message handle, msg. When no longer needed, MsgRel() releases a message back to its block and MCB pools.

MsgMake() makes a message from any bare block pointed to by bp; its pool, if any, is stored in the MCB. This service, following sb_BlockGet(), is the basis for high-speed, no-copy data input. When the last task is done with the message, MsgUnmake() reverses the process and releases the data block to its correct pool.

Reversing the process is the basis for high-speed, no-copy output. In this case, a message is obtained with smx_MsgGet(), converted to a bare block by smx_MsgUnmake(), then released to its pool by sb_BlockRel().

smx messages are sent to and received from exchanges. The use of exchanges has important advantages over direct task-to-task messaging, such as anonymous receivers — i.e. the receiver identity need not be hard coded into the sender. This increases system flexibility and task independence.

An exchange is an smx object, which enqueues either messages or tasks, whichever is waiting for the other. XchgCreate() allows creating one of three exchange modes: normal, priority-pass, or broadcast. The first two have priority queues; the third has FIFO queues. An exchange can be deleted, when no longer needed.

A priority-pass exchange changes the priority of the receiving task to that of the message it has just received. Thus a client task can pass its priority via a message to a server task. Server tasks (e.g. a print server) are a good way to avoid resource conflicts and to free high-priority tasks for more important work.

A broadcast exchange permits broadcasting messages to many tasks simultaneously. Each task receives a proxy message; the sender retains the original message. Multicasting is also supported.

Utility services are provided to bump a message's priority, peek at its parameters, release all messages owned by a task, and clear an exchange of messages or tasks. Also peek is provided for exchanges.

Pipes

smx_PipeClear(pipe)
smx_PipeCreate(pbuf, width, length, name)
smx_PipeDelete(*pipe)
smx_PipeGet8(pipe, pdst)
smx_PipeGet8M(pipe, pdst, lim)
smx_PipeGet(pipe, pdst)
smx_PipeGetWait(pipe, pdst, tmo)
smx_PipeGetWaitStop(pipe, pdst, tmo)
smx_PipePut8(pipe, byte)
smx_PipePut8M(pipe, psrc, lim)
smx_PipePut(pipe, psrc)
smx_PipePutWait(pipe, psrc, tmo)
smx_PipePutWaitStop(pipe, psrc, tmo)
smx_PipeResume(pipe)
smx_PipeStatus(pipe, *ppss)

smx pipe services consist of a mixture of SSRs for use from tasks and functions for use from ISRs. Both can be used from LSRs. Pipes handle serial byte and packet streams for task-to-task communication and I/O (ISR to LSR or task). Packets may be from 1 to 127 bytes. Pipes are viewed as having *widths* corresponding to packet sizes. Put functions put bytes or packets into pipes, and get functions get them out. For puts, psrc points to the source of the next byte or packet. For gets, pdst point to the next destination for a byte or packet.

The PipeCreate() function accepts a pointer to a pipe buffer and creates a pipe of the specified width and length. The pipe buffer can be anywhere in RAM. Pipes can be deleted, when no longer needed.

For inter-task communication, the PipePutWait() and PipeGetWait() services are SSRs intended for use from tasks. They provide synchronization between tasks for serial transfers of packets. If a pipe is full, the putting task will wait until the getting task gets a packet. Conversely, if the pipe is empty, the getting task will wait until the putting task puts a packet. LSRs can also use these services, but cannot wait. Stop versions are provided for one-shot tasks.

For I/O, PipePut8() transfers a byte to a pipe; PipeGet8() gets a byte from a pipe and puts it at pdst. The Put8M() and Get8M() versions transfer multiple bytes, up to the specified limit, lim. The PipePut() and PipeGet() versions transfer packets. All of these functions are intended for use from ISRs and can safely interrupt complementary pipe SSRs being used from tasks or LSRs.

Since the I/O functions are not SSRs, they cannot resume a task waiting at the other end of a pipe. After a number of bytes or packets have been transferred to or from a pipe, an ISR should invoke an LSR to call PipeResume(). This will resume the first task waiting at the other end of the pipe.

Pipe utility functions are provided to clear pipes and to get pipe status.

Note that smx pipes normally operate between a task at one end and a task, LSR, or ISR at the other end. However, smx does allow multiple tasks to wait on the same pipe, in priority order, either to put or to get bytes or packets. This permits pipes to be used as *message queues*.

Semaphores

smx_SemClear(sem)
smx_SemCreate(mode, lim, name)
smx_SemDelete(*sem)
smx_SemPeek(sem, par)
smx_SemSignal(sem)
smx_SemTest(sem, tmo)
smx_SemTestStop(sem, tmo)

smx provides 6 types of semaphores:

- Binary Resource (lim = 1) controls access to a single resource.
- Multiple Resource (lim = N) controls access to N resources (e.g. blocks in a block pool).
- Binary Event (lim = 1) records that one or more events have occurred.
- Multiple Event (lim = 0) counts all events; the classical counting semaphore.
- Threshold (lim = T) fires every T events.
- Gate (lim = 1) resumes or restarts all waiting tasks on one event.

A semaphore is created with the mode and lim determining its type. It can be deleted when no longer needed. SemTest() is used to test a semaphore. SemTestStop() is provided for one-shot tasks. SemSignal() is used to signal a semaphore, when a resource has been released or an event has occurred. Tasks wait at a semaphore in priority order, except for the gate semaphore.

The classical use of a resource 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 subsequent tests suspend (or stop) the testing task on the semaphore until it is signaled by another task that has released its resource. Hence, only N tasks can use the resources, at once.

A binary event semaphore is used in producer / consumer transactions. It can have only two states: 0 and 1. A signal changes it to 1; additional signals have no effect upon it. A producer may signal the semaphore many number of times. When the consumer tests the semaphore, it will pass and clear its 1 count to 0. The consumer accepts all items the producer has produced without retesting the semaphore. When done the consumer might do other work before testing the semaphore again.

A multiple event semaphore records all events that have occurred. No events are lost, even when no task is waiting (unlike an event queue). This is useful for counting quantities such as revolutions of a wheel or items on a conveyer belt.

A threshold semaphore enables counting multiple events per action. Each signal increments an internal counter. When the count reaches the threshold, the first waiting task is resumed and the count is reduced by the threshold. This can be used, for example, to determine that all slave tasks are done.

A gate semaphore passes all waiting tasks on one signal. It is useful to restart all slave tasks at once or to resume run-time limited tasks that have reached their run-time limits.

Clearing a semaphore releases all waiting tasks with FALSE return values and restores the semaphore to its initial state. SemPeek() allows obtaining information about a semaphore.

Mutexes

smx_MutexClear(mtx)
smx_MutexCreate(pi, ceiling, name)
smx_MutexDelete(*mtx)
smx_MutexFree(mtx)
smx_MutexGet(mtx, tmo)
smx_MutexGetStop(mtx, tmo)
smx_MutexRel(mtx)

Mutexes are safer for resource protection than binary resource semaphores for the following reasons:

- Nested testing by the same owner is permitted, without causing the owner to stall.
- A mutex can be released only by its owner. Attempted releases by non owners have no effect.
- Priority promotion of the owner, when a higher-priority task waits, avoids unbounded priority inversion.

A mutex with priority inheritance (pi) promotion is created by specifying pi = 1. A mutex with ceiling priority promotion is created by specifying a ceiling > 0. Mutexes can be created with both.

If a mutex is free, MutexGet() passes; otherwise the calling task is suspended. If priority inheritance is enabled, the mutex owner's priority will be raised to that of the suspended task, if higher. Furthermore, if this task is waiting on another mutex, the higher priority will propagate to its owner and so on through all such linked mutexes. This is called *priority propagation*. Not all RTOSs do this.

When a task gets a mutex, if *priority ceiling* is enabled, the task's priority is immediately raised to the ceiling. This is a simpler way to avoid unbounded priority inversion and it also avoids *mutex deadlocks*, which occur if two tasks wait for mutexes already owned by each other. Deadlocks are avoided by giving all mutexes that are shared by the same tasks, the same ceiling. Then, the first task to get one of the mutexes blocks the other tasks from running.

A mixture of ceiling and inheritance can be useful in some circumstances (see smx Special Features).

When a mutex with pi or ceiling is released, *staggered priority demotion* occurs. This means that the releasing task's priority is reduced to the highest level necessitated by pi or the ceiling of mutexes that it still owns. (This, of course, is what would be expected.)

Mutexes have an internal nesting count. Each time the owner gets a mutex, its nesting count is incremented; each time the owner releases the mutex, its nesting count is decremented. A mutex is not released until its nesting count reaches 0. Nesting is necessary because called functions may get and release the same mutex. Sometimes, mutex dependencies can be hidden in libraries. In such cases, using semaphores for resource protection can result in task stalls — i.e. a task waiting for a semaphore that it already owns.

MutexFree() allows freeing a mutex by a non-owner, regardless of its nesting count. MutexClear() does the same and resumes all waiting tasks with FALSE. These are provided for special operations, such as mutex delete and system recovery and should not be used for normal operation. Mutexes can be deleted when no longer needed.

Event Queues

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

An event queue permits multiple tasks to wait for differing counts of an event. An example of an event queue is smx_TicksEQ, which permits tasks to wait for specified numbers of ticks, with tick accuracy. Event queues are useful for counting other events such as revolutions, pulses, etc.

Services are provided to create and delete event queues, when no longer needed. EventQueueCount() allows a task to wait for a count of events, with a timeout. A Stop version is provided for one-shot tasks. A task or LSR can signal an event queue. Tasks are enqueued differentially so that only the counter of the first task is decremented by a signal. This minimizes overhead, thus permitting large numbers of tasks to wait for events of a given type.

If no task is waiting, signals are ignored. This is unlike the multiple event semaphore, which counts all signals. Clearing an event queue, resumes all waiting tasks with FALSE.

Event Groups

smx_EventGroupClear(eg, imask)
smx_EventGroupCreate(imask, name)
smx_EventGroupDelete(*eg)
smx_EventGroupPeek(eg, par)

smx_EventFlagsPulse(eg, smask)
smx_EventFlagsSet(eg, smask, pcmask)
smx_EventFlagsTest(eg, tmask, pcmask, tmo)
smx_EventFlagsTestStop(eg, tmask, pcmask, tmo)

Event groups permit tasks to wait for logical combinations of up to 16 flags. The combinations supported are AND, OR, and AND/OR (e.g. AB + CD). Flags can be individually set or reset. Event groups are useful for systems which are monitoring multiple flags and for state-machine operation.

EventGroupCreate() creates an event group with 16 internal flags and initializes the flags to imask. EventGroupClear() resumes all waiting tasks with 0 return values. A peek service is provided to look at event group parameters. An event group can be deleted, when no longer needed.

EventFlagsSet() first clears flags in the preclear mask, pcmask, then sets flags in the set mask, smask. This permits mutually-exclusive flags, such as M and ~M, which are useful for modes and states. If new flags are set, all waiting tasks are tested for matches and resumed if true. EventFlagsPulse() temporarily sets unset flags in smask, tests waiting tasks for matches and resumes tasks if true.

In EventFlagsTest() the 18-bit tmask specifies the flag combination to test. Bit 16 true is AND; bit 17 true is AND/OR; neither true is OR. For AND/OR, AND terms are separated by 0 bits. For example: MA + nMB is represented by 11011b. If the test condition is met, the test passes immediately and the current task continues (or restarts). If not, it is suspended in the eg wait queue and tmask and pcmask are stored in its TCB.

When a match occurs, due to EventFlagsSet() or Pulse(), pcmask is used to determine which matching flags, if any, to clear. This is called *post clear* or *automatic clear*. The alternative is *manual clear* using EventFlagsSet(0, clear_mask). pcmask permits clearing event flags while not clearing mode flags. In the above example, pcmask = 01001b would clear both A and B, but neither M nor nM.

Multiple tasks can wait at the same event group on any combinations of flags. For each FlagSet() or FlagPulse(), all waiting tasks are checked for matches and resumed or restarted, if found. A combined post-clear flag is accumulated for all flags causing matches and those flags are cleared at the end. Thus flags are set and reset only once, to avoid potential race conditions.

EventGroupPeek() allows peeking at event group parameters.

Timers

smx_TimerDup(*tmrbp, tmr, name)
smx_TimerPeek(tmr, par)
smx_TimerReset(tmr, tlp)
smx_TimerSetLSR(tmr, lsr, opt, par)
smx_TimerSetPulse(tmr, period, width)
smx_TimerStart(*tp, delay, period, lsr, name)
smx_TimerStartAbs(*tp, time, period, lsr, name)
smx_TimerStop(tmr, tlp)

TimerStart() creates a timer control block (TMCB) and enqueues it in the timer queue (tq) for the number of ticks specified by the delay parameter. The timer handle is put into tmrbp.

When a timer times out, it invokes the specified LSR. This provides a low-jitter timer vs. starting a task. If the interval parameter is 0, the timer is a *one-shot timer*, which is deleted when it times out. If the interval parameter is not 0, the timer is a cyclic or pulse timer, which keeps running. Each time such a timer times out, it is immediately requeued to ensure no loss of ticks.

When in tq, each TMCB stores a differential count from the timer ahead of it. Thus, only the count in the first timer is decremented. This results in low overhead, which permits using a large number of timers in a system.

TimerStartAbs() operates the same as TimerStart(), except that it accepts an absolute time from system start instead of a delay from now. It is useful to start coordinated timers from a common base time.

TimerDup() can be used to create a duplicate timer, tmr, with a different handle and name, but otherwise the same settings. It is enqueued after tmr with a

differential count of 0. This is another way to have a common base time.

A pulse timer can be created from a cyclic timer, while it is running using TimerSetPulse(), then controlled by it. This service allows smoothly changing width and period together. Software pulse timers reduce the need for hardware timers. A Pulse timer allows easily generating pulses for pulse width modulation (PWM), pulse period modulation (PPM), and frequency modulation (FM).

The LSR invoked at timeout, its parameter option, and its parameter can be changed with TimerSetLSR(), while the timer is running. Available options are to return: par, pulse state, etime at timeout, or number of timeouts since timer start.

TimerStop() loads time left to the next timeout into tlp, then deletes the timer. This is commonly used to stop a one-shot timer that is timing an event when the event occurs. TimerPeek() returns information concerning a running timer.

ISR and LSR Control

smx_ISR_ENTER()
smx_ISR_EXIT()
smx_LSR_INVOKE(lsr, par)
smx_LSRInvoke(lsr, par)
smx_LSRsOff()
smx_LSRsOn()

The only impact that smx has upon ISRs is:

(1) smx service calls are not permitted from ISRs. Instead, ISRs must invoke LSRs to make them.
(2) Most ISRs begin with ISR_ENTER() and end with ISR_EXIT() macros.

ISR_ENTER() saves volatile registers, switches to the system stack (SS), and increments a nesting counter. Different processors may require more or fewer operations. ISRs can be nested. ISR_EXIT() reverses the ISR_ENTER() actions and returns to the interrupted task, unless its ISR is nested, an LSR is ready, or a task switch may be needed. In this case, it switches to the pre-scheduler. These macros are written in assembly language and optimized for speed.

The INVOKE() macro is used by an ISR to invoke an LSR to perform deferred interrupt processing. Invoking an LSR simply stores the LSR address and a parameter in the LSR queue, lq. The same LSR can be enqueued multiple times with the same or different parameters. This helps to buffer transient high interrupt loads and maintain order. ISRs and the LSRs that they invoke run in the system stack, SS, and do not increase the load on task stacks.

smx_LSRInvoke() is used to invoke LSRs from tasks. This is convenient for starting or emulating I/O operations from tasks. Disabling interrupts is normally used to protect globals shared with tasks and LSRs from interrupts. When it is only necessary to protect globals shared between tasks and LSRs, LSRsOff() inhibits LSRs, and LSRsOn() reenables them.

System

smx_SysEtimeGet()
smx_SysStimeGet()
smx_SysStimeSet()
smx_SysPowerDown(mode)
smx_SysPseudoHandleCreate()
smx_SysWhatIs(h)

etime is elapsed time in ticks, and stime is system time in seconds. Elapsed time is used for task timeouts; system time is used for task sleeps. Functions are provided to get etime and stime and to set stime to the integer equivalent of time elapsed from a fixed starting date and time.

smx_SysPowerDown(mode) provides a structure for power down operations. When there is no useful work to do, it calls sb_PowerDown(mode), which preserves the tick counter count and performs the actual power down operations.

When power is restored, sb_PowerDown() determines clocks lost, starts the tick counter, and returns the number of ticks lost to smx_SysPowerDown(), which performs tick recovery in a manner that preserves the proper order of timer, task, and TicksEQ timeouts. The time to perform tick recovery is generally very short. It depends only upon the number of timeouts and not upon the length of time that power was off. Operation is transparent to the application, if sb_PowerDown() is able to accurately determine the time lost.

Handle Table

smx_HTAdd(h, name)
smx_HTDelete(h)
smx_HTGetHandle(name)
smx_HTGetName(h)

smx_HTInit()

All smx objects are identified by handles. Most smx control blocks contain the names of smx objects. Pseudo handles can be added to HT for ISRs, LSRs, and other objects without names, so that smxAware can display them by name. Functions are provided to get a handle or a name from HT.

Other smxBase Services

sb_TMInit()
sb_TMStart(pts)
sb_TMEnd(ts, ptm)

sb_MsgDisplay()
sb_MsgOutputConst(mtype, mp)
sb_MsgOutputVar(mtype, mp)

The TM functions permit precise time measurements, to a tick timer clock, from one start point to one or more end points. The degree of precision may be as low as one instruction time, depending upon the clock rate selected for the tick timer.

Messages to the console may be enqueued, by pointer, in the output message queue, or copied to the output message buffer for later display by a lower-priority task (usually idle). Error messages are handled this way. This permits using polled UART drivers without impacting system performance.

Ease of Use

This important smx goal is supported by the following features.

Processor Support

SMX is shipped with support for your processor evaluation board and tool suite. This reduces front-end time and gets your project rolling sooner. In addition, well-written BSP notes detailing interrupt settings, memory mapping, and configuration information for all supported peripherals are included.

smxBSP is the board support package included with SMX. See the smxBSP datasheet for a summary of BSP services. Normally, only small changes to the smxBSP code are needed to support the actual project hardware.

smxBase includes smxBSP and provides additional services used by smx and other SMX modules. See the smxBase User's Guide, for details.

Tool Support

SMX deliveries include project files or makefiles to build the libraries for delivered SMX products and the Protosystem. Detailed tool information is provided in the SMX Target Guide, and we provide first-line support for the tools. Hence, if you have a problem, our team will help you, regardless of whether the problem is with SMX or with the tools.

smxAware

smxAware shows what is going on at the task level. It is integrated with the debugger. System and application events can be viewed in graphical and textual form, and smx objects can be examined, in detail. Memory usage, stack usage, and profiling charts help with system tuning. See the smxAware datasheet.

Protosystem

SMX RTOS comes with a framework application called the *Protosystem*. The Protosystem contains core application code for smx and demos for the SMX modules included in your release. Full source code is provided. An included project file or makefile, builds it. Running the Protosystem and demos provides an initial confidence test that the release is working correctly.

The Protosystem provides the framework to start your application development. Demos are replaced with application tasks, objects, and code. Unfinished operating code is simulated with delays. smxAware provides system visibility before most code even exists!

Developing a working skeleton, while other team members fill in detailed task code, helps to solve integration and structure problems up front. Once defined, tasks can be allocated processor and RAM quotas. If developers can meet their quotas, final integration should be smooth as glass.

Evaluation Kits

See www.smxrtos.com/eval for free evaluation kits for many popular processor evaluation boards Each incorporates BSP, Base, and Protosystem source code with an smx library. Some include libraries and demo code for other SMX modules. Contact our sales department to request a custom evaluation with the SMX modules you need.

Because most smx configuration constants can be set from the application, it is possible to increase the numbers of tasks, semaphores, messages, and other smx objects, to increase stack and other sizes, to enable or disable profiling and stack scanning, and other things. Hence it is possible to do useful work with an smx evaluation kit. You do not need to have smx source code.

Using an evaluation kit gives hands-on experience with smx, and your processor and development tools. Having picked smx, many developers start product development with eval kits, while licensing details are being worked out.

Accurate Manuals

Good manuals are one of the most important things that sets a commercial kernel apart from an in-house or free kernel. The smx manuals have been substantially rewritten in v4.

smx User's Guide: A tutorial manual, which presents the theory of smx in four main sections: Introduction, Services, Development, and Advanced Topics. The first section provides orientation. Subsequent sections can be read as needed. The development section provides information on how to structure a multitasking application, using the skeleton methodology, coding, and debugging.

smx Reference Manual: Provides smx service call descriptions and a comprehensive glossary. Service call descriptions are complete and accurate. Each explains all details of the call, including parameters, return values, what the call does, side-effects, and errors, and gives usage examples.

Advanced Features

The advanced features of smx are listed here, by category, for convenience in comparing smx to other RTOSs. They are explained more fully in the smx Special Features data sheet and in the smx User's Guide.

Performance

Short Interrupt Latency permits handling high-frequency interrupts. smx disables interrupts only briefly in a few places, not in system calls.

Fast Task Switching allows applications to use large numbers tasks. For a 400MHz ARM9, smx achieves up to 250,000 task switches per second.

Layered Ready Queue is unique to smx. Enqueueing and dequeueing times are very fast and independent of the number of tasks in rq.

New High-Performance Heap is a bin type heap, which overcomes the poor performance and indeterminacy of normal RTOS heaps.

Link Service Routines (LSRs) are a unique feature of smx, which perform deferred interrupt processing and reduce interrupt latency.

Timers Invoke LSRs Directly, which results in low timer jitter because LSRs cannot be blocked by tasks.

System Stack can be located in on-chip SRAM to improve ISR, LSR, and scheduler performance.

smx Control Blocks and dynamic variables in SDAR improves performance if SDAR can be put into fast on-chip RAM.

Cache-Line Alignment of smx Control Blocks improves performance of smx operations if control blocks are in external memory.

No-Copy Block I/O from bare blocks to messages and back improves software stack performance.

Efficient Memory Usage

System Stack used for initialization, ISRs, LSRs, scheduler, and error manager results in much smaller task stacks.

One-Shot Tasks allow sharing stacks from a stack pool and do not consume stacks while waiting for events.

Dynamically Allocated Regions (DARs) for one-time allocations avoid mixing data and control. Allocated blocks can be aligned for performance.

- System DAR (SDAR) is used for smx objects such as control blocks.
- Application DAR (ADAR) is used for the heap, stack pool, and dynamic application objects.

Safety, Security, & Reliability

Extensive Error Checking. All smx service parameters and numerous other things such as stack overflow, broken queues, and out of resources are continuously monitored, recorded, and reported.

Error Manager records each error in the current task and in global variables.

Error Manager Hook permits easily adding user error-specific code to the error manager.

Local Error Handling is supported by 0 return value if a service fails and the SMX_ERR macro to determine what happened, on a task-specific basis.

Error Manager Runs in System Stack to avoid task stack overflows, while processing other errors.

Error Buffer records information for each error.

Event Buffer records error events in the context of system and application events. Errors are then displayed by smxAware relative to task switches, etc.

Error Messages are enqueued for later display by idle task, when there is no high-priority work to do.

Stack Overflow is caught by scheduler checking of stack pointers and stack high water marks when tasks are suspended or stopped.

Stack High Water Mark is maintained in each task's TCB by scanning stacks during idle time. It can be viewed graphically in smxAware.

Timeouts on All System Service Waits can be used to break deadlocks and to recover from event failures.

Graceful Interrupt Overload Handling due to LSR safe buffering of peak interrupt loads.

Unstoppable LSRs guarantee critical operations cannot be blocked by tasks.

Self-Healing Heap is provided by automatic heap scan and heap fix functions, which run continuously from idle task.

Heap Recovery Function attempts to merge enough free chunks to form a big-enough chunk for a failed allocation.

Heap Extend Function provides another way to recover from a failed allocation.

Heap High Water Mark helps for sizing the heap and monitoring it for memory leaks.

Exchange Messaging assures reliable message transfers using protected pointers and safety checking.

Mutex Priority Inheritance promotes the priority of a mutex owner to that of the highest priority waiting task to avoid unbounded priority inversion.

Mutex Ceiling Priority provides a simpler way to avoid unbounded priority inversion and also prevents mutex deadlocks.

SDAR isolates smx objects from application objects, such as pools, stacks, etc., which are in ADAR.

Debug Aids

smxAware graphical and textual kernel awareness tool shows the system-level view.

System Event Logging into the event buffer (EVB) for later display by smxAware aids visualization of system operation.

User Event Macros can be placed in application code and permit storing up to 6 variables for later viewing via smxAware.

Stack Pads allow operation to continue despite stack overflows. Easily removed for release.

Precise Profiling of tasks, ISRs, and LSRs gives exact measures of how code is performing. Resolution is to the tick timer clock.

Precise Multi-Path Time Measurements of execution times, response times, interrupt latencies, etc. help to improve code performance.

Use of Enums, Object Names, and Control Block Pointer Types for Links makes debugging easier.

Debug Chunks in the Heap store helpful information per chunk and have fences around data blocks to catch block overflows.

New Heap Functions aid diagnosis of heap problems.

Ease of Use

Heap Configuration via a Single Constants Array allows the heap to be easily tuned to a specific application.

smx_SysPowerDown(mode) provides structure at the RTOS level to put the system into a specified power saving mode, then performs accurate tick recovery when power is restored.

Message Exchanges provide a simple, flexible way to exchange messages.

Message Priorities allow higher priority messages to bypass lower priority messages at exchanges.

Pass Exchanges cause server tasks to operate at message priorities set by client tasks.

Message Broadcast Exchanges simplify sending messages to multiple receivers.

Multicasting and Proxy Messages enable powerful new operations.

Message Make allows making a message from any block, thus permitting no-copy processing.

Message Unmake allows unmaking a message into its original block.

Event Groups allow AND, OR, and AND/OR combinations of flags. Selective pre-clear supports mutually-exclusive flags. Selective post-clear clears used event flags, while preserving mode flags.

Hooked Exit & Entry Routines transparently save and restore extended task context to support FPUs, other coprocessors, and global objects.

Pipes with Variable Widths from 1 to 127 bytes can be used to pass bytes, pointers, or small data packets for both I/O and intertask communication.

Accurate Task Timeouts permit resolution as low as one tick, with low overhead. This simplifies code by allowing task timeouts to be used for accurate timing.

Pulse Timer allows generating pulses for pulse width modulation (PWM), pulse period modulation (PPM), and frequency modulation (FM).

smx_SysPowerDown(mode) provides structure at the RTOS level to put the system into a specified power saving mode, then performs tick recovery when power is restored.

Additional References

1. smx Special Features.
2. smxAware datasheet.
3. smx++ datasheet.
4. smxBSP datasheet.

smx manuals are available for review and may be downloaded from www.smxrtos.com.

