# smx vs. FreeRTOS

by Ralph Moore

This paper compares the commercial RTOS kernel **smx** to the generic free RTOS kernel **FreeRTOS**. Both are real-time, multitasking kernels intended for use in embedded systems. The material in this paper is organized into sections, which first present FreeRTOS features, then present the corresponding smx features, thus giving side by side comparisons. This comparison is based upon what is documented in the manuals and available for all ports, other than processors which are too small.

## Task Scheduling

Both kernels implement preemptive task scheduling, which is the best method for real-time embedded systems. Both also support cooperative and time slice scheduling. Task switching times are an important characteristic of RTOS kernels.

**FreeRTOS** task switching time is fast for the Cortex-M port.

**smx** uses special algorithms to achieve very fast task switching — up to 290,000 task switches per second on a 400 MHz ARM9. The et2 example in the examples for smx (esmx) package can be used to measure task switching rate on any supported processor.

## Breaking the Last Brace

**FreeRTOS**: If a task runs through the last } of its main function, in a release build, results are unpredictable. This is an unfortunate Achilles heel.

**smx**: Running through the last } of a task's main function is a normal operation called "autostop". The task simply goes into a dormant state from which it can be restarted by another task. While in the dormant or "stopped" state, the task does not consume a stack, thus saving RAM. Tasks of this type are called "one-shot tasks". They are useful for simple, infrequent operations. A task can even return a parameter to itself when autostopping.

## One-Shot Tasks

**FreeRTOS**: No support.

**smx** has taken the one-shot task concept a step further by implementing stop versions of all smx services that can wait for an event. For example, for smx_SemTest() there is smx_SemTestStop(). The latter permits a task to wait for a semaphore, in the stopped state, without consuming a stack. When the semaphore is signaled, the task is restarted and smx gives it a stack from its stack pool. Doing so takes about the same time as resuming a task, since there are no registers to reload. If no stack is available, the task remains in the ready queue and the smx scheduler simply skips over it until a stack is available. Hence, many one-shot tasks can share a few stacks in a pool. When a one-shot task is running, it behaves exactly like a normal task, except that it does not have an internal infinite loop — it just runs straight through. A system can have any mixture of normal and one-shot tasks. Use of one-shot tasks allows breaking an application into many small tasks, which are easy to write and debug, and do not add excessive RAM overhead.

## System Stack

**FreeRTOS**: Support is available for Cortex-M processors, which implement system stack support in the hardware. It is available for some other ports, but the kind of support and coverage per processor is not specified.

**smx** implements a system stack for all processors it supports. It uses the system stack for all smx ISRs, LSRs, the smx scheduler, and the smx error manager. When there is no system stack, the stack overhead of these must be added to every task stack. This can be substantial. For example, if the typical ISR requires 50 bytes for autovariables and to save the registers it uses, and if ISRs could nest up to 10 deep, then 500 bytes must be added to every task stack. If RAM is dear, this discourages using as many tasks as are needed for an optimum system solution.

smx does not have this problem. For it, task stacks can be on the order of 100 to 200 bytes, depending upon nesting of subroutines called by the task main function. Using a system stack also makes the process of tuning task stacks less hazardous. When ISRs use task stacks, there is a possibility that a small change in one ISR could produce a rare stack overflow due to maximum nesting of ISRs. This rare event could escape testing when tuning stack sizes and show up in the field, where it is difficult to find and expensive to fix. Hence a system stack not only saves RAM, but it also improves safety and reliability.

## Interrupt Handling

**FreeRTOS** has special ISR versions of kernel services that can be used from ISRs. There are two disadvantages to this: (1) Service calls add significant overhead to ISR execution times. (2) Many internal kernel operations, such as enqueueing and dequeueing tasks, must be protected from interrupts by disabling them. These increase interrupt latency. There is also the risk of subtle bugs in the kernel, due to not disabling interrupts, when they should be.

**smx** uses a simpler system, which adds minimal interrupt latency and is inherently safer. It consists of the following: (1) ISRs are not permitted to make service calls; they can only invoke link service routines (LSRs), which is an interrupt-safe process. (2) LSRs run after all ISRs are done and can perform any no-wait smx service calls. Interrupted smx service routines (SSRs) are allowed to complete before LSRs can run. The result is that all SSRs and LSRs run with interrupts fully enabled, and only a few small sections of the smx scheduler require interrupts to be disabled.

## Deferred Interrupt Processing

The ideal in embedded systems is to defer interrupt processing as much as possible in order to keep ISRs short and thus minimize interrupt disabled times.

**FreeRTOS** offers a somewhat cumbersome way of doing this. When an ISR version of a service runs, it determines if it is making a higher-priority task ready, and if so, it sets a xHigherPriorityTaskWoken flag ISR variable. At the end of the ISR, portEND_SWITCHING_ISR(xHigherPriorityTaskWoken) is called to do the actual task switch and to return to the new task instead of the old one. It is not clear how much load this puts on the ISR — I have not studied the code. However, the main downside of this approach is that deferred interrupt processing is done by a task, which may be preempted by higher priority tasks, and which may be subject to unbounded priority inversions.

**smx** provides a simpler and more deterministic approach. Deferred processing is performed by LSRs. LSRs run ahead of all tasks, and thus cannot be blocked by tasks, nor are they subject to priority inversions. LSRs run in the order invoked and therefore preserve the correct sequence of real-time events. Since they can make all non-wait service calls they operate like small tasks. In addition, the same LSR can be invoked multiple times, each with a different parameter (e.g. value or a pointer). This is valuable in overload situations. LSRs are similar to Linux tasklets, but simpler, and tasklets cannot be multiply invoked.

## Mutexes

**FreeRTOS** offers a standard mutex with **priority inheritance**. Recursive getting and giving of a mutex by the mutex owner is supported. This is an expected feature of any mutex. It is not clear why FreeRTOS has non-recursive mutexes. Recursive, alone, should be sufficient. It is not clear if FreeRTOS supports priority promotion and staggered priority demotion, which are described below.

**smx** offers standard mutexes with both **priority inheritance** and **ceiling protocol.** To handle the situation where the owner of one mutex is waiting at another mutex, **priority propagation** to the owner of the second mutex, and beyond, is supported. Each task maintains a list of mutexes that it owns. This allows **staggered priority demotion**, such that when one mutex is released, the task's priority will be demoted to that of the highest priority waiting task at or ceiling of any of its mutexes. Also, the mutexes will be automatically released if the task is deleted.

Ceiling protocol provides a simpler method of raising owner priority. It is also effective at preventing mutex **deadlocks**. When a task becomes an owner, its priority is simply raised to that of the mutex's ceiling. All mutexes shared between a group of tasks are given the same ceiling. Hence if a task gains one mutex in the group, it gains all. smx allows both schemes to be used together, such that ceiling might be used to a mid-priority level and inheritance above.

## Semaphores

**FreeRTOS** offers counting and binary event semaphores and resource semaphores. A basic API is provided for them.

**smx** adds **threshold** and **gate** semaphores. A threshold semaphore is a counting semaphore that counts signals up to the specified threshold, then resumes the top task and reduces the count by the threshold. This is useful for taking an action every Nth event, waiting for N slave tasks to report back, etc. A gate semaphore is more-or-less the opposite — it resumes all waiting tasks on a single signal. It is like opening the gate of a corral so all the horses can run out.

## Event Groups

**FreeRTOS** offers **event bits**, which are similar to event groups. There can be either 8 or 24 bits per group and a task can test for AND or OR combinations of specified bits. The API offers the usual create, delete, set, wait, clear, and get functions. It also offers an unusual sync function that appears to be useful for rendezvous operations.

**smx** offers **event groups**, which are groups of event flags. There can be 16 flags per group and a task can test for AND, OR, or AND/OR combinations of flags. In addition, a pre-clear mask is provided for the set operation to permit mutually exclusive flags. These are useful for modes. AND/OR tests can be used to implement mode-dependent testing, such as: MA + nMB where M = mode M, nM = not mode M, A = event A, and B = event B. Thus, in mode M the task waits for event A, but when not in mode M, the task waits for event B. This permits implementing state machines, where tasks correspond to states. smx also provides the usual create, delete, set, test, and clear, as well as pulse and peek functions.

## Timers

Software timers are useful for repetitive operations, timing actions, timeouts, etc.

**FreeRTOS** provides both one-shot and cyclic timers. A timer is created by allocating and initializing a timer control block, which contains the timer period, type, call back function pointer, and ID. The callback function is called from the **timer service task** when the timer expires. Timer commands are loaded into a timer queue and can timeout if the timer queue is full. Timer processing can be delayed due to the priority of the service task. It does not appear that this implementation is adequate for accurate time delays — even less so when coupled with time-slice task scheduling.

**smx** uses **volatile timers**, which are created on-the-fly when a timer is started. A timer control block (TMCB) is allocated from the TMCB pool, initialized, and linked into the timer queue. (This design is possible because TMCBs come from a fast block pool, not from the heap.) Each TMCB maintains a differential count from the timer ahead of it, so the KeepTimeLSR need only decrement the first TMCB count. KeepTimeLSR is invoked from TickISR and thus runs once per tick. When a timer times out, it is immediately requeued, if cyclic, and then its LSR is invoked with the specified parameter. One-shot timer TMCBs are freed automatically. Volatile timers are flexible — initial delay, cycle time, LSR, and its parameter are specified by timer start. Invoking LSRs, which run ahead of tasks, instead of callback functions running from a timer task means that timing will be more crisp and accurate — a primary requirement in many systems.

## Memory Management

**FreeRTOS** provides 4 kinds of heaps. The first two have very limited functionality and are intended for minimal systems. The third makes compiler heaps thread-safe. The fourth provides thread safety, first-fit block allocation, and block free with consolidation of free blocks.

**smx** divides dynamic memory into two regions: **SDAR** for system objects and **ADAR** for application objects. More DARs can be added, if desired. SDAR is normally small and can be put into on-chip SRAM to improve smx performance. ADAR includes application block pools, task stacks, and the heap. It is normally large and located in external memory. Separation of SDAR and ADAR avoids applications overwriting smx objects due to block and stack overflows. Blocks allocated from DARs can be aligned for best performance or to minimize size.

**smx** provides two types of block pools: **base pools** for use by ISRs, drivers, and initialization code and **smx pools** for use by tasks for **blocks** and **messages**. Compared to heaps, block pools provide deterministic, fast block allocations. Base pools are lightweight, interrupt-safe, and provide minimal safety. smx pools are middleweight, task-safe, and provide more safety.

**smx** also provides a heap, which has safety features not found in compiler heaps nor the other FreeRTOS heaps. Each allocated data block is separated by a **heap control block** (HCB), which provides forward and backward links, owner ID, and a fence. Backward links permit heap repair if an HCB is damaged. Fences help to detect damage. Owner ID permits automatic freeing of heap blocks when a task is deleted. Allocated blocks are aligned on cache-friendly, 16-byte boundaries, which improves performance.

## Control Blocks

All kernels use control blocks. A control block enables a kernel to control the object that it represents. For example a task control block (TCB) contains information necessary to control the task it is associated with.

FreeRTOS uses dynamic control blocks, like smx, which is rare for RTOS kernels. Unlike smx, FreeRTOS gets its control blocks from its heap. Unfortunately, this brings the deficiencies of heaps (indeterminacy, slow allocation, failure due to fragmentation, etc) to bear upon kernel object creation. Another downside, is that control blocks are mixed with application stacks and blocks. Hence, application bugs causing stack and block overflows can damage kernel control blocks. When a control block is compromised, kernel behavior becomes unpredictable. This is obviously undesirable when trying to debug  application code and can result in considerable wasted time. You would like for the kernel to be solid so you can focus on application bugs. The MPU version provides greater safety, but very few processors have an MPU.

**smx** gets its control blocks from control block pools in SDAR (see Memory Management section, above). In general, there is one pool per control block type. Getting control blocks is deterministic and very fast. Pools are created dynamically, as needed, and are aligned on cache-line boundaries. Most smx control blocks are 16 bytes in size or a multiple of 16 (e.g. the TCB is 80 bytes) so they are cache-line aligned within their pools. This makes a big performance difference for operation from external memory for processors with data caches. Putting smx control blocks in SDAR makes them immune to stack and block overflows in the heap, or elsewhere in ADAR. In addition, since smx control blocks are small, SDAR is small, and it frequently can be located in on-chip SRAM, resulting in even better performance, for most processors.

## Messaging

**FreeRTOS** provides **message queue** messaging. It provides the usual services, such as create, delete, send, receive, peek, as well as ISR versions for I/O. There is a SendToFront version to effectively achieve a two-priority system, but the higher priority messages end up in LIFO order. If what you really want is two message priority levels, you are not going to get it from this. The main downside to message queues is that data must be copied in and then copied out. This is a problem with all message queues. For large messages, it is best to transfer block pointers via message queues. Another downside, is that a message queue has a limited length. It is possible that it will fill up and a sending task or ISR cannot wait, so data will be lost.

**smx pipes** operate the same as message queues and the API is similar. For I/O, smx offers get and put functions, which are ISR-safe and do not involve smx services. When an input or output pipe needs to be serviced, an ISR invokes an LSR, as explained previously, to either handle it or to wake up a task to handle it. smx permits increased safety for task to task pipes by passing handles to smx blocks rather than raw block pointers.

smx supports a second type of messaging, **exchange messaging,** which permits tasks to send messages to exchanges and to receive messages from exchanges. Messages are data blocks controlled by message control blocks (MCBs). The latter contain message priority, reply pointer, owner, block pool pointer, message block pointer, and exchange links. There are three types of exchanges: **normal**, **priority-pass**, and **broadcast**. When a message is sent to a normal exchange, it is delivered to the top waiting task or enqueued at the exchange, if no task is waiting. When a task receives a message from an exchange it gets the top waiting message, or it is enqueued at the exchange, if no message is waiting. A priority-pass exchange operates the same way and also passes the priority of the message to the receiving task. A broadcast exchange passes a message handle and block pointer to every waiting task and any that arrive later. Exchange messaging is ideal for client/server applications. There are no queue length limits; messages can be prioritized; server priority can be adjusted to message importance; and servers can reply to unidentified clients.

Exchange messaging has many inherent safety features:  Ownership passes from sender to exchange to receiver. A sender cannot send or release a message that it does not own. Message release is automatic — the user need not know where a message block came from in order for it to get back to the correct pool. Message operations are performed via message handles, which are tested, rather than via untested block pointers, which could point anywhere. Message block pointers are protected inside of MCBs.

The latest release of smx has added the capabilities to make blocks into messages and to unmake messages into blocks. These permit no-copy input, no-copy propagation through a stack, and no-copy output from a stack. Another new feature, proxy messages, adds enticing possibilities.

## Profiling

Profiling allows determining exactly how much processor time is used by each task, LSRs, ISRs, and kernel overhead.

**FreeRTOS**: Trace macros can be placed in the code to gather trace information. This can be viewed using an external tool. Information on profiling is sketchy.

**smx** provides built-in trace macros and functions to capture run time counts, and built-in code to save trace information for later viewing. Task profiling is accurate to one clock of the hardware timer used to generate the tick. For some processors, this is single instruction accuracy; for others it may be as many as 30 instructions. Run-time counts (RTCs) are kept for each task and for LSRs combined and ISRs combined. The RTC for the idle task is, of course, idle time — an important thing to know. Uncounted clocks during a profile frame are attributed to overhead — also important to know. Frame length is specified by the user; it can vary from one tick up to several minutes. Profiles are recorded in a buffer, which can save from a few up to hundreds or thousands of frames, as determined by the user. These, in turn can be uploaded to smxAware for individual frame and composite frame viewing. Since profiling is very precise and profile periods can be long, it is possible to pick up tasks that seldom run or run for only brief periods. Profiling helps to correlate undesirable system behavior with certain tasks hogging the processor.

## Error Management

Error detection, reporting, and management are vital for quick debugging and reliable operation.

**FreeRTOS** reports if a service fails, but it gives little or no reason why. It does contain asserts, which are helpful during debug, but not available in release versions. It also offers stack overflow monitoring.

**smx** monitors about 70 error types. All smx service parameters are range and type checked, stack overflow is frequently monitored, and numerous other potential problems such as broken queues, excess locks, heap block overflows, insufficient DAR, insufficient heap, etc. are monitored. When an error occurs, the smx error manager (EM) is invoked. As noted previously, EM runs in the system stack, so that it cannot cause a stack overflow while handling another error (i.e. a double fault). EM records the error number in the current TCB. It also sets a global error number, increments a global error counter, increments a global error type counter, logs the error both in the error buffer and the event buffer and queues an error message for future console output by the idle task. Undesired operations can be disabled.

EM saves hours and hours of frustrating debugging time. Global error variables are useful for monitoring system health, in the field. EM also stops seriously damaged tasks and halts or reboots the system if an irrecoverable error occurs. smx_EMHook() permits customizing EM operation to the requirements of an application.

## Stack Management

**FreeRTOS** provides a service to obtain a task's stack high-water mark. This is determined by loading a fixed pattern into each stack, when a task is created. Stack overflow detection is implemented by testing sp when suspending a task and by checking the top 20 bytes of the stack for the pattern. The MPU version provides greater stack checking, but very few processors have an MPU.

**smx** does stack scanning automatically from the idle task and records high-water marks in each stack owner's TCB. Free stacks in the stack pool are also scanned, and their high-water marks are stored in last owner TCBs. As a consequence of frequent scanning, smxAware is able, at all times, to upload and display stack usage graphs for tasks in its stack usage window. Also smx has a configuration option to specify the size of a pad to be added to every stack, so that during development a system can keep running even after a stack overflow has occurred. This makes it easy to tune stack sizes. The stack usage display in smxAware is color coded to use orange and red to alert the user of near and actual overflow, respectively.

The smx scheduler also checks a task's stack pointer, when it is suspended or stopped. When smx detects that the stack of a task that is about to be suspended has overflowed into the register save area (RSA) above the stack, the task is restarted, because it cannot be suspended. If the stack high-water mark indicates probable overflow above the RSA, an exit routine is called that the user can implement to halt or reboot the system.

## C++ Support

**FreeRTOS**: Provides only extern "C" statements in header files to permit its use with C++ compilers.

**smx++** provides a C++ API for smx. smx services are grouped into classes, such as smx_Task and smx_Sem. Then smx services are available as methods, constructors, and destructors of the classes. The C++ new and delete operators are overloaded with base pools. As a consequence object construction and destruction is very fast and does not result in a fragmented heap. smx++ objects reference smx control blocks, and consequently are very small (only 8 bytes).The smx scheduler automatically handles the unique *this* pointer for each C++ task. C and C++ tasks can be freely mixed in the same system and are differentiated only by the *this* pointer. As a consequence, projects can use: C only, C++ only, or a mixture of C and C++. The latter enables programmers to work in the language they prefer or which is most suitable for the task at hand. The overall smx++ implementation is efficient and does not result in significant performance nor memory overhead.

## RTOS-Aware Debugger Plug-in

**FreeRTOS**: A third-party tool from Percepio is offered and free basic state-viewing plug-ins are available for several debuggers.

**smxAware** is a premier kernel-aware debugger plug-in for smx and SMX middleware. When a system is stopped at breakpoint, the contents of the smx event buffer (EVB) are uploaded to smxAware. EVB size is specified by the user and can be very large if plenty of RAM is available. Event logging is designed to be efficient so that performance is minimally impacted. With smxAware, it is possible to view an event timeline showing when threads (ISRs, LSRs, and tasks) ran and what service calls they made. This gives an overview that is invaluable to understand what is happening in a system. The view is easily zoomed in and out, and precise times are shown. The same information is also available in log form. (Customers often send us logs for support.) All smx object control blocks, queues, and other diagnostic information can be displayed. Graphical displays are available for stack usage by task, processor usage by task, memory usage of DARs, heap, and stack pool, and profile frames. smxAware Live is a remote monitoring version that allows viewing the graphical displays of a system in the field.

## Conclusion

Free is fine for hobbyists, but not for commercial projects. The cost of good commercial tools is minmal compared to the cost of project overruns, canceled projects, and unhappy customers. Good tools are needed to do good work and to deliver projects on time. Kernel selection should be based upon what a kernel can do, not what it costs — cost, alone, is penny-wise and pound-foolish.

## Disclaimer

This comparison is based upon published FreeRTOS manuals and some other published materials. The FreeRTOS code has not been studied, nor tested. It has come to my attention that some ports of FreeRTOS offer additional features, not documented in its manuals. To undertake a comparsion of all features in all ports would be a prohibitive task, besides which, what is important is those features that are common to all ports (other than processors too small to use them). Having features available only in some ports undermines the concept of being able to move an application to a different processor due to using the same RTOS.

Ralph Moore
smx Architect
March 25, 2014