FRPort Tips

When using FRPort, your application is running on smx, not FreeRTOS (FR). One reason to change to smx is in order to use the security features of SecureSMX. These are highly dependent upon smx and cannot be used on FreeRTOS. These tips are intended to help you to adjust to this change.

1. **FRPort converts FreeRTOS functions into smx functions.** smx functions do not generate FreeRTOS control blocks; they generate smx control blocks. Hence the FreeRTOS handle points at an smx control block, not a FreeRTOS control block. If any of your code accesses FR control block fields directly, it must be changed to use an appropriate FR function, which is mapped to an smx function such as an smx Peek function. In any event, it is bad practice to access RTOS control block fields directly because they may change from version to version of an RTOS.
2. **Key smx Variables**: It is recommended that you put the following at the top of the debugger Watch window to monitor what is happening:
   2.1. smx_srnest – nesting level. If not 0, system code, LSR, or ISR is running.
   2.2. smx_sched – scheduler command: TEST, STOP, SUSP, NOP
   2.3. smx_errno – smx error. smx does testing of service call parameters and other things.
   2.4. smx_ct – current task.
   2.5. smx_rq – ready queue is layered by priority.
3. **smxAware**: Click smxAware in the main menu and select:
   3.1. Text shows all smx control blocks and their fields, queues, diagnostics, and config settings.
   3.2. Graph shows a graphical timeline of all tasks, ISRs, LSRs, system calls, errors, etc.
   3.3. Event Buffer is a log of events, which is useful to diagnose problems.
   3.4. Memory Usage shows usage of various memory resources.
   3.5. Memory Map is a graphic diagram showing locations of system structures in memory and memory dumps of any areas.
4. **LSRs** are a unique feature of smx used for deferred interrupt processing. They are invoked from ISRs and timers. They run after all ISRs have finished and before any tasks run; hence they cannot be blocked by tasks. LSRs operate like mini-tasks that have priority over all tasks.
5. **STATIC/DYNAMIC** does not exist in smx. The STATIC and DYNAMIC versions of an FR create function are implemented with the same smx create function. All smx control blocks come from control block pools. Rather than defining static control blocks, it is only necessary to specify how many of each control block is needed by your system. smx services check that handle parameters are inside of corresponding pool limits as well as checking that control block types are correct. This helps with debugging as well as making code more resistant to hacking. Other RAM, such as task stacks are allocated from a heap or statically, as appropriate.
6. **ISRs:** IRQs above a certain priority level are RTOS-independent and require no modification. RTOS-dependent ISRs must be modified for smx to add smx_ISR_ENTER()

at the start and smx_ISR_EXIT() at the end. These macros are defined in xarmm.h. Also, set (in XBASE\barmm.h) SB_ARMM_DISABLE_WITH_BASEPRI to 1 and SB_ARMM_BASEPRI_VALUE to the top RTOS-dependent priority

7. **FromISR() Functions:** smx does not implement special ISR system service functions. Instead, ISRs invoke LSRs, which can make all normal non-wait system service calls. Performing potentially long system services from ISRs is not a good practice. Keeping ISRs short results in better performance, reliability, and security. All ISR code that is not essential to re-enabling the interrupt causing the ISR should be moved to an smx LSR. FromISR() functions are implemented, for convenience, but are the same as non-ISR functions. This way you can move whole sections of code from an ISR to LSR without modifying the calls. See TIMx_IRQHandler() in IntQueueTimer.c for examples of how to move code from ISRs to LSRs. It is strongly recommended that you search your entire codebase for FromISR to find all uses to be sure none are left in ISRs. Otherwise, you may spend a lot of time debugging subtle problems due to corruption of system data structures.
8. **LSRs** run in the order invoked by ISRs or timers, and all LSRs run before task-mode is resumed. Hence, for tasks it does not matter if a service call was made in an ISR or in an LSR – the result is the same. Since LSRs maintain temporal integrity, making service calls within ISRs vs. within LSRs should also not matter. An exception might be making a get-value service call in order to decide whether to perform a function that must be performed in the ISR. FromISR functions that just return values should be safe in ISRs.
9. **Deferred-interrupt processing** is best moved from tasks to LSRs. This results in less system jitter because LSRs cannot be blocked by higher-priority tasks and task priority inversions.
10. **Interrupt Response Times:** LSRs are interruptible, unless interrupts are disabled in them. The smx task scheduler is over 90% interruptible and has special "LSR flybacks" to assure that all invoked LSRs run before the next task runs. Tasks are interruptible, unless interrupts are disabled in them. System services routines (SSRs) are almost 100% interruptible.
11. **Task Local Storage:** only one 32-bit word is implemented in smx TCB.
12. **xQueueGenericReset()** implemented with smx_PipeClear() resumes all waiting tasks.
13. **Co-Routines** are not implemented in smx. However, smx implements *one-shot tasks*, which share stacks from a stack pool and can do anything a normal task can do. A one-shot task requires a full TCB of about 100 bytes but is able to wait on smx objects without a stack, thus saving 500-1000 bytes for a normal task stack.
14. **Timer queue**: smx allows any number of timers to be enqueued in the timer queue, tq. Hence timer function timeouts are not needed and are ignored. Timers are enqueued by differential timeout counts resulting in fast timeout handling. (Only, the differential count of the first timer in tq is decremented each tick). Timeouts are handled by an LSR invoked from the tick ISR.
15. **Timers**: smx uses smx_KeepTimeLSR(), invoked from the smx_TickISR(), rather than a task to manage timers. Timers are enqueued, by expiration time, with no limit on the number of timers. When a timer expires, it invokes an LSR, which is executes the same code as the callback function from an FR timer. This function will run sooner and with less jitter than from a task, since there is less overhead.

16. **Timers:** smx implements dynamic timers rather than the static timers used by FR. smx timers obtain Timer Control Blocks (TMCBs) from a TMCB pool. When a timer is not in use, its TMCB is returned to the pool. Normally smx creates and starts a timer with a single start timer function. To implement FR static timers, smx timers are parked at the end of smx_tq with infinite timeouts, until they are started. When started or restarted, a timer is moved to its timeout-relative position in smx_tq and its active flag is set. When a timer times out or is stopped, it is parked at the end of smx_tq and its active flag is reset.
17. **Timer ID:** The LSR parameter is used for this. The parameter can be used to identify which timer invoked an LSR or to provide a parameter to an LSR to tell it what to do. The same is true for LSRs invoked from ISRs.
18. **Queues:** The term *queue* in smx means a linked list of waiting objects such as tasks, messages, or timers. A data queue is called a *pipe*. Pipes, semaphores, and mutexes are separate objects, each with their own control blocks and functions.
19. **Mutexes:** All mutexes are recursive.
20. **xQueueCreateStatic():** smx requires size = (length + 1)*width.
21. **Debugging with Control Blocks**: FRPort converts FreeRTOS functions into smx functions. smx functions do not generate FreeRTOS control blocks; they generate smx control blocks. Hence the FreeRTOS handle points at an smx control block, not a FreeRTOS control block. In order to see the smx control block, do the following:

```
TimerHandle_t  t1;
TMRCB_PTR    t1x;

 t1 = xTimerCreate("t1", 10, pdFALSE, (void*)1, (TimerCallbackFunction_t)ti00_LSR1);
 t1x = (TMRCB_PTR)t1;
```

   Then t1x can be used to look at the smx control block. It does not have the same structure nor field names, but the field names are similar and it fairly easy to find the desired information. During debugging, it is also possible to step into smx functions in order to see related smx control blocks.

   It may seem a little awkward, at first, to use FreeRTOS functions with smx control blocks, but you will get used to it quickly and it gives you a good introduction to smx.

ISR Changes

FreeRTOS has special versions of system services such as xSemaphoreTakeFromISR(), which perform all of the functions of the normal service, except the actual task switch, which is done after all ISRs finish.

smx has a different approach. Only one smx service[1], smx_LSR_INVOKE() is permitted from ISRs. This function loads the LSR handle and one parameter into the LSR queue, lq. After all ISRs have finished, LSRs are run in the order invoked, ahead of all tasks.

[1] There are some low-level pipe services implemented in xpipe.c such as smx_PipePut8() and smx_PipeGet8() that are intended for usage from ISRs in order to transfer data to and from pipes rapidly. See the smx Reference Manual for more information.

Hence, to port to smx some ISR modifications are necessary. However, they must be done, anyway, because the FreeRTOS ISR mechanism is not secure. LSRs can call any smx service with no wait. Hence, a FromISR() service call can be replaced with invoke of an LSR that calls the equivalent smx service with the same parameters. Normally the service will resume a task that performs processing required by the interrupt. If the amount of code is small, it could be moved to the LSR and the task omitted. This should result in performance improvement since LSRs have far less overhead than tasks and they cannot be blocked by priority inversions.