

Stack Usage Checking

smx and smxAware Feature for v3.6.0

Author: David Moore

Introduction

Detecting and avoiding stack overflow is of primary importance in a multitasking system. It has been one of the most common problems smx users face, and one of the most difficult to detect, by the earlier mechanisms in smx. It has been the cause of a great deal of wasted time and frustration. Hence, it is important to give the user a reliable means to detect overflow, and it is even better to give him a good tool to check stack usages, so he is forewarned about stacks that are likely to overflow and need to be increased.

Prior to v3.6.0, all smx did was to write guard bands (32-bit patterns) at the end of the stacks and check them at task switches to ensure they were unchanged. Guard bands give no indication of stack usage, and from experience, we found they rarely ever detected stack overflows. If the stack grew only by individual pushes, this would be a reliable way to detect overflow. But it doesn't. The typical stack overflow is caused by a function allocating space for its autovariables. This is done by advancing the stack pointer by some offset (e.g. `sub sp, nnn`). Some routines, such as `printf()`, use a great deal of stack this way (1500 bytes or more), and this can cause the stack pointer to skip over the guard band into the next stack. If the autovariable that coincides with the guard band is not changed, the guard band remains intact and the overflow goes undetected. This is especially likely if the autovariables are large character buffers, of which only the first few bytes are used.

This paper discusses the new mechanisms for stack checking that are much more reliable at detecting stack overflow and also allow determining stack usage. Each task's stack usage is stored in its TCB, allowing for easy inspection in the debugger. Those using `smxAware` are able to display a simple bar graph of stack usages as a percentage of stack size.

This paper also discusses delayed release of stacks, which is a necessary change to avoid slowing down task scheduling with stack checking operations at the time of a task switch.

Terminology

“Top of stack” means the end the stack grows toward. “Bottom of stack” means the end the stack starts at. “Higher” means closer to the top. “Lower” means closer to the bottom. In this way, we discuss the stacks logically rather than in terms of addresses.

High-Water Mark

A new field has been added to the TCB to indicate a “high-water mark” (HWM) of stack usage. This stores the number of bytes of the stack that have been used by the task so far. Storing the number of bytes rather than storing a stack pointer value is good for two reasons: (1) It avoids problems in terminology that arise since the stack grows toward lower addresses rather than toward higher addresses; (2) It is easy for the user to understand, since it can be compared directly to the stack size. The user does not have to calculate it by subtracting addresses.

TCB.shwm is increased when smx determines that more of the stack has been used. Briefly, this is done by comparing against the current stack pointer and/or by scanning the stack. Details of updating the HWM are discussed below.

A HWM is task-specific and cumulative – i.e. it is valid even if the task has used many different stacks, since being created. A HWM is only increased and never reduced or cleared, so as a system runs, it will stabilize. Its accuracy will increase over time, as tasks run through all paths and interrupts occur at the deepest nesting levels in the code.

Stack Clearing and Scanning

Stack clearing makes it possible to determine stack usage by scanning from the top of the stack block to the bottom, until a location is found that no longer has the clear value. The clear value can be configured (see Clear Value for Stack Clearing below). Clearing and scanning are enabled by setting `STACK_ENHANCED_TESTING` in `conf.h`.

Stack Padding

Stack padding is extra bytes added to the top of the stack that the stack can grow into. Padding allows the system to keep running after overflow occurs, since the pad will be corrupted rather than the adjacent stack or other data. This increases the chances that the overflow will be detected. The user can then let the system run a while and then look at the stack usage graph to quickly determine optimal stack sizes. This saves the need to keep changing stack sizes in many places in his code during development. The pad can be set large during development and then disabled for release. Stack padding is enabled by setting `STACK_PAD_SIZE` in `conf.h`.

Configuration Constants

| | | |
|------------------------|------------|-------------------|
| STACK_ENHANCED_TESTING | 0 or 1 | |
| STACK_CLEAR_VALUE | 0x11111111 | (int-sized value) |
| STACK_PAD_SIZE | size | (0 disables) |

STACK_ENHANCED_TESTING enables stack scanning and clearing, to update the HWM (tcb.shwm and SHWM_VALID flag). This is done by process_next_stack(), which is called by the smxStackTask. Clearing and scanning are combined into a single option to minimize the number of configuration choices.

STACK_CLEAR_VALUE allows the user to specify the value to clear stacks to, when STACK_ENHANCED_TESTING == 1. This is discussed in Clear Value for Stack Clearing, below.

STACK_PAD_SIZE specifies how many extra bytes to allocate to every stack. This pad is put above the top of the stack (the end it grows toward). This setting is independent of the others because padding is useful to avoid overflow during development, regardless of whether stack testing is enabled. It adds RAM overhead, but very little run-time overhead.

These are all set in conf.h (and loaded into the cf table), so that it is not necessary to rebuild the smx library to change these settings. This permits library-only users to benefit from this added feature too. We considered also adding constants to xdef.h to allow the preprocessor to strip out this code, but it is only a small amount of extra code. Having fewer options makes things simpler.

New TCB Fields Summary

stp = stack top pointer

sbp = stack bottom pointer

ssz = stack size (usable space; not including guard bands)

shwm = stack high-water mark (number of bytes used; not including guard bands)

shwm directly compares against ssz. Percent stack used = shwm/ssz * 100.

Updating the HWM

One way to update the HWM is by comparison against the current stack pointer periodically as the system runs. This is done by the scheduler when it tests the new and old stacks during a task switch. The problem with this method is that it can easily miss an instant when the stack pointer is at an extreme. This is likely for rarely executed paths in the code, such as initialization. For example, a task might call a function once during initialization that uses many bytes of stack for autovariables. Unless the check had been

done during the brief time that this function ran, the extra stack usage would go undetected.

Another way to update the HWM is to initially clear stacks and then scan them while running. Stacks are scanned from top to bottom, until a non-clear value is encountered. If padding is enabled, the scan starts at the beginning of the pad.

It would seem that if tasks did not call functions during initialization that required more stack than the rest of the task, and if all other paths were exercised occasionally, the method of checking the stack pointer at task switches would be adequate, since the HWM would stabilize over time — the chances of catching the stack pointer at an extreme increases the more times those paths in the code run. However, it is hard to control how much stack is used by a task during initialization when it calls functions in third-party libraries, for example. For the case of catching the stack pointer during seldom-run paths in the code, it is not reasonable to have to let the system run a long time to get accurate stack usages. The program may fail long before it would have shown that stack overflow was the cause. When debugging, a problem such as stack overflow should be immediately apparent.

Stack scanning is the better method and it is almost 100% reliable¹. The stack has a memory of the overflow, so no lucky timing is required. The code only needs to run once through each path to see how much stack is used. (The only additional concern is usage by isr's/lsr's which also run on the stack and can run when the stack pointer is already at an extreme — but this applies when determining stack usage from the stack pointer, too.) We tried both methods on our Protosystem, and after letting it run a few minutes, we found that the usages reported by the stack pointer method were much lower than the actual usages reported by scanning the stacks.

Bound tasks' stacks are cleared by `create_task()`, and that stack is used for the life of the task, so it is never cleared again. Unbound tasks use shared stacks from the stack pool. Every time they stop (rather than suspend), their stack is cleared and released to the stack pool so the next time they start, they get a fresh, pre-cleared stack. The high reliability of this method comes at a cost of runtime overhead to clear and scan stacks. Also, RAM overhead is incurred for all of the stack pads, if padding is enabled.

Scanning and clearing stacks is a time consuming operation, so it is best done from a low-priority task (`smxStackTask`), rather than when the scheduler releases a stack or needs to get a new one. (See `Delayed Release of Stacks`.) Also it is best to do as little scanning as possible. A task's stack only needs to be scanned if it has run since the last time its stack was scanned. Tasks that have not run since the last scan have valid high-water marks. A new flag, `SHWM_VALID`, has been added to the TCB to indicate this.

¹ Stack scanning is not fully reliable because a function could skip over the whole pad if the pad is not big enough. Or the clear value could have been written at the overflow location(s), which would go undetected in the pad, but would likely damage an adjacent stack if the pad were not there. Or, it is possible a location could have changed and then changed back to the clear value before the check ran.

The scan task will scan only stacks with `SHWM_VALID == 0`. Being able to tell which stacks need to be scanned is especially important for `smxAware`, since it cannot run a routine on the target. If `smxAware` were to scan a stack, it would have to read and compare each dword of the stack via the JTAG/BDM/serial connection, which is very slow.

Whenever `TCB.shwm` is updated by scanning the stack, the `SHWM_VALID` flag in `TCB.flags` is set to 1 to indicate the HWM is known to be correct for this task. This flag is cleared by the scheduler for each task it starts or resumes. This flag is always 0 for `ct`. It will be 0 for all other tasks that have run since their stacks were scanned. If the user disables stack scanning, this flag will never be set because the results of updating `TCB.shwm` by comparison with the stack pointer may not be reliable, as discussed earlier. This way, `smxAware` will indicate it is unsure about the amount used for all stacks, and it can then scan all of the stacks to determine accurate HWM's.

The following is a summary of where these operations take place:

`STACK_ENHANCED_TESTING == 0`: (i.e for release)

1. The scheduler updates `TCB.shwm`, if necessary, in the `TM_TEST_STACK()` macro (`test_stack` for assembly), right before the stack switch, by comparing against the stack pointer. The macro has been changed to check `TCB.shwm` for overflow, in addition to checking the guard bands. This does not run very frequently so it is likely to miss an overflow, unless the system can run for a long time.
2. `EXIT_ISR` (future, if at all). This macro runs much more often than the number of task switches, and is asynchronous with respect to tasks, so it is much more likely to catch the stack pointer at a new high-water level. However, we are not going to implement this yet because this macro will be different for every CPU, and because it may slow down `EXIT_ISR` unacceptably.
3. Asynchronous `isr` (future, if at all): Instead of burdening all `isr`'s by adding the check to `EXIT_ISR()`, we could create a new `isr` that we hook to a timer interrupt that runs frequently and asynchronously with respect to the tick so that it is likely to interrupt at all places in the code if the system is run long enough. This `isr` would do the check and update of `TCB.shwm`. This would still be likely to miss the case where a task uses a lot of stack during initialization.

`STACK_ENHANCED_TESTING == 1`: (i.e. during debug/development)

1. Stacks are scanned and cleared. Stack pool stacks are cleared when returned to the stack pool, by `process_next_stack()`, called by `smxStackTask`.
2. `smxAware` can scan stacks for any tasks whose `SHWM_VALID` flag is 0 (and whose `STK_CHK` flag is 1) and it can update `TCB.shwm` and this flag.
3. `TM_TEST_STACK()` macro (`test_stack` for assembly scheduler) continues to update `tcB.shwm` based on `sp`, as above.

Regardless of how the constants are set, the `TM_TEST_STACK()/ test_stack` macro updates `TCB.shwm` since it is a short operation, and it could be a while before `smxStackTask` gets around to scanning the stacks again during times of heavier system loading. A little redundancy increases our chances of catching an overflow quickly.

Clear Value for Stack Clearing

The best value to clear the stacks with is one that does not occur in the application, so that values written to the stack can be seen by the scan or in a debugger's memory dump window. Although 0 might look nice in a memory dump, it is a poor choice since it is so common in an application. A value such as `0x11111111`, `0x12345678`, or `0xAAAAAAAA` might work well. Or a string such as "stak" might be nice in a memory dump. `0x55555555` is used for the guard bands, so in order for the guard bands to be visible in a memory dump, this value should not be used. We use `0x11111111` by default but the user can easily change this by setting `STACK_CLEAR_VALUE` in `conf.h`.

Guard Bands

The guard bands are still useful.

If `STACK_ENHANCED_TESTING == 0`, the top guard band is needed since it is the only means of checking for overflow. Even if `STACK_ENHANCED_TESTING = 1`, it is still useful to check the top guard band because it may be a while until `smxStackTask` checks this particular stack again. This might catch an overflow and if it does, the overflow generates the `STACK_OVERFLOW` error immediately.

Having a guard band at the bottom of the stack (where `sp` starts) is useful for detecting stack underflow. This is a rarely occurring error that results from imbalanced stack operations, such as too many pops vs. pushes or clearing too many parameter bytes off the stack after a function call. This might happen if we make changes to the scheduler, for example, or if the user writes task code in assembly language. Most likely an imbalance in a called routine will not cause an underflow because before that can happen, the code will pop the wrong value into the instruction pointer and the program will crash. It can happen under other circumstances too. For example, CodeWarrior for ColdFire v3 made an unusual optimization in stack handling that caused this error to occur, and other tools could too. Since it is a quick check it is useful to preserve it. Note that an underflow is more likely to hit a guard band than an overflow because the stack is likely to be imbalanced by only one or two slots.

The guard bands also allow the user to see the extents of each stack in the debugger's memory dump window.

Because of the benefits, and because checking the guard bands is a short operation, we will retain this code in all cases. Checking to see if we need to check the guard bands (i.e. if stack clearing/scanning is disabled) probably would take about as much code as simply checking them.

Specification of guard band locations:

1. Guard bands wrap the actual stack area, not the pad. They are in the same place regardless of whether padding is enabled.
2. `tcb.stp` points to the top guard band (the one the stack grows toward).
3. `tcb.sbp` points to the bottom guard band (the one where the stack pointer starts).

Originally the plan was to make `tcb.stp` point to the beginning of the block, so if padding were enabled, it would point to the pad. However, we realized that most of the time during execution, what is needed is a pointer to the guard band, not the pointer to the block. The only time the block pointer is needed is to free the stack, which is only done when a task stops and releases its stack. This doesn't happen for bound tasks unless they are deleted. The guard band checks happen much more frequently.

STK_CHK Flag

The `STK_CHK` flag in `TCB.cbtype` is an old feature of `smx` that allows disabling stack checking (pointers out of range and guard bands) during periods when the application switches to a foreign stack. Now it also inhibits the new operations of high-water mark updating, stack scanning, and stack clearing for each task that has this flag set. However, if the task is releasing its stack to the stack pool, its stack is scanned and cleared regardless of this flag's setting because stacks must be cleared before being returned to the stack pool. Note that scanning a stack does not add any extra time to the clearing operation (and may actually save time) since it saves us from clearing the bytes that are already cleared. We only clear from the point where the scan found the first changed value.

It is not necessary to skip stack scanning when `STK_CHK` is disabled because it does not matter whether a task is currently using a foreign stack or not; scanning is done to the task's actual stack, not the foreign stack (the `TCB` points to the actual stack). However, by disabling stack scanning when `STK_CHK` is disabled, we give the user a way to disable scanning on a task-by-task basis. He may wish to disable scanning for a task with a very large stack, for example. Clearing the `STK_CHK` flag is the only way to do that. Also it is consistent with the fact that the `TM_TEST_STACK()/test_stack` macro does not check guard bands when this flag is cleared (this behavior did not change). (If the user disagrees with this decision, it is a simple matter to change this behavior by simply removing the check of `STK_CHK` in an `if()` statement near the beginning of `process_next_stack().`)

Stack Overflow Error Reporting (STK_OVFL Flag)

Stack overflow should only be reported once for a task. Checking if `tcb.errnum == STACK_OVERFLOW` does not work because `tcb.errnum` is the most recent error caused by the task. If the task causes another error, `tcb.errnum` will be changed to that new error type, so `STACK_OVERFLOW` will be reported again. If the task keeps generating other errors, stack overflow will continue to be reported. We decided it was best to add another TCB flag and check it rather than `tcb.errnum`, to avoid this.

Delayed Release of Stacks (REL_STK Flag)

Initially while making these enhancements, we made the scheduler release shared stacks to the stack pool when a task stopped. We couldn't scan the stack because it could have been assigned to another task by the time we started scanning it. As a result, stacks could only be scanned when suspended, so they might not be scanned often enough for the HWM to be reliable.

The solution Ralph proposed was to delay the release of the stacks. This is the solution we chose. The scheduler was changed to set a new flag, `REL_STK`, instead of actually releasing the stack. `process_next_stack()`, called from `smxStackTask`, releases it (after scanning and clearing it). If the task is restarted before its stack can be processed and released, it will continue to use the same stack, just as it would if the stack were bound. See the section about `process_next_stack()`, below, for details of what it does.

Benefits of delayed release:

1. Stack clearing is done in a (low priority) task rather than in the scheduler, so scheduler performance is not impacted.
2. Scanning and clearing can be done in this one routine, which saves time. After scanning a stack it only needs to be cleared from the point where the scan stopped to the bottom of the stack.
3. If the task is resumed/restarted before the stack is scanned, cleared, and released, the stack is simply re-used by the task. This reduces the overhead of these operations for tasks that frequently stop and restart.
4. In the flyback case of the scheduler (where the new task has a stack and is about to start, but we re-enter the scheduler because an lsr has become ready), the task keeps the stack rather than needlessly clearing it and releasing it to the stack pool.
5. `tcb.stp` and `tcb.sbp` can be cleared. (`tcb.ssz` should not be cleared, so that `smxAware` will know how big the stack was and can show the bar to indicate percent used.) As the scheduler was initially modified, these pointers were still needed after it released the stack, in the macro that tests the stack. This was counter-intuitive. With delayed release, they are cleared when the stack is actually released.

6. Setting REL_STK rather than releasing the stack saves linking it into the free list, reducing code in the scheduler. Releasing the stack is done in the background.
7. smxAware can display accurate information for all tasks. Otherwise, it would not be able to scan stopped tasks because their stacks might have been already reassigned to other tasks. In this case, it would wrongly report the usage of such a task as the usage of another task. Stacks to be released will still be “connected” to the TCB so they can be scanned. Stacks that have been released have been already scanned and SHWM_VALID has been set, so smxAware does not need to scan them.
8. Consistency with handling of control blocks. Stacks in the stack pool are cleared and ready to use.

Problems with delayed release:

1. The system could run out of free stacks if process_next_stack() doesn't run often enough. The scheduler needed to be modified to handle this case. See the Complexities section below.
2. delete_task() cannot immediately free the TCB since that is what points to the stack. This is easily solved by adding a REL_TCB flag to the TCB. delete_task() sets it and process_next_stack() releases the TCB. It may be necessary to increase the number of TCB's by a couple (NUM_TASKS in conf.h) since some could be waiting to be released when new ones are needed.

In the case where the user has stack scanning disabled, we considered having the scheduler directly release stacks to the stack pool. However, there are benefits for delaying the release in this case too:

1. tcb.stp and tcb.sbp can be cleared when the stack is released. See benefit #5 above.
2. Simplicity. The scheduler runs the same either way. Typically scanning is enabled during debug/development and disabled for release, so this makes the debug and release builds run more closely the same way. It saves needing to have a check in the scheduler and extra code to directly release the stacks.
3. It saves some instructions in the scheduler since the stack is linked into the free list in the background rather than in the scheduler. See benefit #6 above.

Originally, the idea was to make a queue of tasks waiting to have their stacks released. However, we can't do that because a stopped task can be waiting on a semaphore, exchange, etc. so the links are needed for that queue. Instead, process_next_stack() just scans between tcbi and tcbx (where it last left off) until it finds a task with the REL_STK flag set and it will process and release its stack. When there are no more with REL_STK set, it will continue checking other tasks' stacks.

Delayed Release of TCB's (REL_TCB Flag)

Since stacks are not immediately released, `delete_task()` cannot immediately release the TCB either, since that is what points to the stack. Instead it sets a new flag, `REL_TCB`, and `process_next_stack()` releases the TCB if this flag is set, after it disconnects the stack from the TCB.

`process_next_stack()` / `smxStackTask`

This function performs the following operations:

1. It finds the next stack to process. Tasks that are waiting to release their stacks are processed first before tasks whose stacks need to be scanned. (If `STACK_ENHANCED_TESTING == 0`, they are the only tasks that are processed.) There is a special case we deal with in this loop: If a TCB is marked `REL_TCB` and it has a bound stack, both are released here since it is not necessary to do scanning, clearing, etc. Normally `delete_task()` directly releases the stack and TCB for bound tasks, but it can't do that for `ct` or for a task being processed by this function. We check for this and handle it in this loop because the code below this loop handles only stacks that are in use or shared stacks that are marked for release. Looping continues after this special case because we want to be sure to release a shared stack if any are marked for release, since the scheduler expects that this routine will free one if available.
2. If `STACK_ENHANCED_TESTING == 1`, it saves the TCB fields that are used for scanning and clearing in local variables. This ensures all are saved at a consistent state (remember this routine can be preempted and `stop()` and `delete_task()` can run on the task whose stack is currently being processed). Also saving the variables allows releasing the TCB at the top of this routine rather than at the end, which makes it available sooner.
3. If `REL_STK` is set, the stack is disconnected from the TCB (or the TCB is simply released if `REL_TCB` is set), by clearing `tcb.stp` and `tcb.sbp` and clearing the `REL_STK` flag in the TCB. The stack is not yet released to the stack pool. Doing this first avoids the need to `LOCK` the scan-and-clear operation; only this step of disconnecting it from the task needs to be locked. Otherwise, the task could be restarted while its stack is being scanned or cleared and remnant bytes could be left in the stack un-cleared. See the Complexities section below for more details. (`tcb.ssz` is not cleared so `smxAware` can display percentage of stack used. There is no need to clear `tcb.ssz` anyway, since all pool stacks are the same size so it will always have the same value.)

4. If `STACK_ENHANCED_TESTING == 1`:
 - 4a. It scans the stack for the first changed element (word/dword) and saves the number of bytes used in a temporary variable.
 - 4b. It clears the rest of the stack, from where the scan stopped, if `REL_STK` is set.
 - 4c. If the TCB has not been released, `tcb.shwm` is updated and the `SHWM_VALID` flag is set. `STACK_OVERFLOW` is reported `tcb.shwm > tcb.ssz`.
5. Releases stack to the stack pool if `REL_STK` is set.
6. Releases TCB if `REL_TCB` is set. (It is set by `delete_task()`.)

`process_next_stack` is called from the `smxStackTask`, which is a 0 priority task, so that these lengthy scanning and clearing operations are done during slack times. The reason for putting it into its own task instead of the idle task is because the scheduler can also force this task to run when a stack is needed for a new task it is about to start. In that case, we don't want to run whatever other code the user may have added to the idle task. We want the task to do only this stack processing. Note that this function must be called only from this one task, since it is non-reentrant. We can't just lock the whole function since we want other tasks to preempt it (so it runs only when they're not ready to run).

Note that the guard bands are not written by this function because the word/dword at the top of the stack is used to link to the next stack in the stack pool, so we can't write the top guard band until we take a stack out of the stack pool. It is simplest to write both guard bands at the same time, so we do so in the scheduler after we get a new stack.

Note: `smxStackTask` must have a bound stack to prevent the possibility that it can't run because there are no free stacks!

smxAware Display of Stack Usage

`smxAware`'s textual Stacks window lists each stack, with columns showing bytes used, stack size, and percentage used. It also indicates which stack usage checking method is used (scanning or checking the stack pointer).

The graphical displays include a Stack window that shows a color bar chart of stack usage. Bars are displayed horizontally and indicate percentage used. The bars are blue. At 80% usage, the bar turns to orange; at 100% they turn to red. It is immediately obvious from this display which stacks have overflowed or are close to overflowing. Stacks that have not been scanned since the last time their task ran are shown in dim colors to indicate that they are questionable. (These are stacks for which the `SHWM_VALID` flag is 0 in `TCB.flags`.) A future enhancement proposed for `smxAware` is to have it scan these questionable stacks while the user is looking at the display and then redisplay the bars after scanning them. Before each bar is text that shows the number of used bytes and

stack size (e.g. 732/1024). Showing the number of bytes used makes it easy to see what to set stack sizes to, as opposed to showing only the percentages.

Complexities

The HWM is not perfect, because stacks can grow depending on the timing of interrupt and function nesting and whether rarely-run paths in the code have run or not. Because of this, we don't overly complicate things to try to make the HWM perfect. For example, if, in some circumstances, the HWM is increased but not by enough, it will be corrected on the next scan or one soon after. The HWM's become more accurate the longer the system runs. What must be avoided is any case that might set it to a higher value than the actual task's stack usage.

Task Whose Stack is Being Scanned is Resumed/Restarted

If the task whose stack is being scanned is resumed/restarted during or right after the scan but before setting the HWM and clearing it, the stack could grow beyond the point the scan stopped. When `smxStackTask` is resumed, `process_next_stack()` will continue where it left off and it will set the HWM lower than the actual stack usage. We won't worry about this in the case where we are only scanning and not clearing the stacks. As explained above, the HWM is not perfect and it will be increased on the next scan of this stack.

However, in the case where a stack was to be cleared for release, any bytes above the point where the scan stopped would not get cleared. When this stack block was assigned to another task that uses less stack, these remnant bytes would be encountered by a later scan and it would appear that this other task used more stack than it really did. The programmer could waste a lot of time trying to determine why this task needs so much stack, when it really doesn't.

To solve this, we first considered adding another flag to the TCB: `SSCAN_VALID`. `process_next_stack()` would set it to 1 before scanning. The scheduler would set it to 0 in each task it starts or resumes. At the end of the scan, `process_next_stack()` would `LOCK()` itself and then check to see if `SSCAN_VALID` is still 1 in the TCB of the task whose stack it just scanned. If so, it knows that the task did not run again, so the results of the scan are valid. It would set `SHWM_VALID` to 1 and continue clearing and releasing the stack. If the flag was cleared, it would mean the task had run again during the scan and the results were invalid since the stack could have grown more. In this case, it would abort and scan the same stack the next time it ran. This seemed a bit complicated.

An alternative is to `LOCK()` the scan task, but it would be necessary to lock the whole scan/clear/release operation. This would prevent higher priority tasks from running for a long time.

Ralph thought of a good solution: If we disconnect the stack from the task and clear REL_STK before scanning, we don't need to LOCK the scan and clear, only the few lines to release the stack from the task. The key is that at this point, the stack is neither assigned to a task nor in the stack pool. This way scanning and clearing can be done without fear that the block will change. In the case that the task that owned the stack is restarted, it will get a new stack. It is ok to update the HWM after disconnecting the stack, since the HWM only grows larger. There is no race condition here. If the task ran again with a new stack while scanning the old stack, and if the HWM were increased (during a task switch), then when the scan of the old stack finished, the HWM would already be larger than what the scan found, so it would not be changed. If the scan finished and set the HWM and then the new stack was scanned and more was used than the old stack, the HWM would be increased. Disconnecting the stack from the TCB before scanning and clearing it is the solution we chose.

Task Whose Stack is Being Scanned is Deleted

In the original (v3.6.0) implementation, delete_task() would immediately release the stack and TCB for bound tasks. It would only set the flags for delayed release for unbound tasks (those with shared stacks). However, because process_next_stack() runs mostly unlocked, delete_task() could have immediately deleted the task whose stack was in the process of being scanned by process_next_stack(). This could result in an attempt to write the new HWM to the already freed TCB, which could have already been re-assigned to a new task. This problem was fixed in v3.6.1.

The initial fix was to change delete_task() to simply mark all stacks and TCB's for delayed release. This did not work well, though, since many tasks could be waiting to have their stacks and TCB's freed, since smxStackTask runs at minimum priority, and the system could run out of heap or TCB's.

The final fix was a compromise between these extremes: delete_task() now immediately releases the bound task's stack and TCB in most cases — all except the case where the task being deleted is the task whose stack is being processed by process_next_stack() or ct. In these 2 cases (and for tasks with shared stacks), it sets the flags for delayed release. This required adding an smx global, tproc, to indicate the task whose stack is currently being processed by process_next_stack().

A further benefit of this change is that smxKillTask is no longer necessary for a task to delete itself, since this is handled now by smxStackTask.

Out of Stacks

If the stack pool is empty and the scheduler needs a stack to start the top task, execution just loops back to the top of the task scheduler to look for the next-highest priority task to run. But before looping, the error service routine for OUT_OF_STACKS stops the new task for a tick, which removes it from the ready queue. This is important because it

allows same- and lower-priority tasks to run, which might release their stacks if allowed to run.

With delayed release of stacks, there could be stacks waiting to be released that could be immediately given to the new task, so we don't want to keep the new task waiting for lower priority tasks to run when there really are free stacks.

Ralph and I discussed this at length and considered several options. The scheduler can not directly call `process_next_stack()`, because this function may have been already running and it is non-reentrant.

One option would be to create an lsr to call this function. Then the scheduler could invoke the lsr to put it into the lsr queue and then branch to the lsr scheduler. The lsr would run and free a stack (if one was marked to be released) and then the task would get that stack. The lsr scheduler falls into the task scheduler so this all happens automatically. We considered having a timer run this lsr periodically too, rather than putting it into a task. But the problem of doing this in an lsr is that it can be a lengthy operation (due to scanning and clearing) and that would prevent other lsr's and high priority tasks from being able to run until this operation completed. For example, a higher priority task with a bound stack could become ready to run during the scan and we want it to immediately run and preempt the scan. It can't do that if the scan is done from an lsr.

A good solution is to call `process_next_stack()` in a new task, `smxStackTask`, that only calls this function (and `unlockx()`). Then, when the scheduler runs out of stacks, it moves `smxStackTask` to the top of the ready queue to precede the top task and then branches to the task scheduler. Since `smxStackTask` is the top task, it will run to free the stack. Note that the priority of `smxStackTask` would not be changed, so that after it ran, it would be enqueued at the end of its normal priority level. The scan/clear operation could be interrupted by an lsr and preempted by a higher priority task, and this is fine. Since the priority was not changed, `smxStackTask` is put back into the ready queue at its normal level if it is preempted. When the higher priority task finishes, the scheduler will again try to start the task that needed the stack, and, if necessary, `smxStackTask` will be moved ahead of it in the ready queue again. Besides being necessary to handle it this way, this has the benefit that `smxStackTask` will resume where it left off, so it will continue to make progress even if it is preempted. Normally, this task alternates with the idle task (it is created at priority 0) by bumping itself to the end of its level in the ready queue after every run. The reason for creating a new task rather than calling `process_next_stack()` from the idle task is because the user could have added other code to the idle task and we don't want to keep the scheduler waiting.

An additional complexity is determining when to report the `OUT_OF_STACKS` error. It should only be reported when there are no stacks in the stack pool and no tasks whose stacks are marked to be released (i.e. with `REL_STK` set). The `free_stack` global is 0 if there are no stacks in the stack pool, but there could be a stack marked to be released, in which case we're really not out of stacks. But the scheduler can't directly call

process_next_stack() (which would release the stack and set free_stack to point to it), because it is non-reentrant and could be currently running.

My first idea was to add a loop to the scheduler to scan TCB's to see if one has REL_STK set, ahead of the code that requeues smxStackTask at the top of the ready queue. If no TCB had REL_STK set, then the scheduler would report the error, and if one had it set, tnext would be set to point to it so process_next_stack() wouldn't have to scan to find it again. This won't work, though, because process_next_stack() may be in the process of scanning/clearing a stack it just disconnected from a task, and the task's REL_STK bit is no longer set, so we would report OUT_OF_STACKS wrongly when there really is an available stack (the one being scanned and cleared).

The scheduler needs a way to know that process_next_stack() ran and that it did not release a stack, so it can report OUT_OF_STACKS, but there is no direct coupling between the scheduler and this function since it is not directly called. What could happen is that a stack could be freed but then a higher priority task starts before we return to the task that needed a stack, so we reach the same path again. But we do not want to report an error since there was no out of stacks condition, since the stack was freed but given to another task.

To solve this, we added a global flag, need_stack, that is set before the scheduler bumps up smxStackTask. This flag is cleared in the other scheduler path that successfully gets a stack from the stack pool and in the flyback path since a new (higher priority) task may be ready and may or may not need a stack. If the scheduler path runs that needs a stack and need_stack is already set, it reports OUT_OF_STACKS.

The only remaining complexity with this is that at the time the scheduler bumps up smxStackTask to be top task, it could have been already running scanning a stack that doesn't need to be released (i.e. a bound stack or one for a task that has been suspended), so after process_next_stack() finishes, there is still no free stack available. The solution is to have smxStackTask call process_next_stack() again if need_stack == 1 && free_stack == 0. This way it will surely release a stack if there is one marked to be released.

Foreign Stack

If the user switches stacks or calls a library function that switches stacks, he should first turn off stack checking with STACK_CHECK(OFF), which clears the TCB STK_CHK flag. smx should issue a new warning type, SMXWARN_FOREIGN_STACK, to alert the user if the stack pointer is determined to be in a foreign stack when the scheduler does stack checking during a task switch. Ways to detect foreign stack vs. stack overflow:

1. ss != ct->ss (segmented x86 versions)
2. sp is outside of its bounds by some thresholds (e.g. more than 1/2 stack size below (logically) the heap or 3 times stack size above it (logically) means

foreign stack rather than stack underflow/overflow). Different thresholds make sense since underflow is unlikely to be more than a small amount out of bounds, while overflow could be quite a lot.

Decisions 10-3-03: (David & Ralph mtg)

1. Single option to enable scanning and clearing. If the user wants to just clear stacks, he can modify `process_next_stack()` not to do the scanning. Consider naming it `STACK_ENHANCED_TESTING`, which might enable additional features in the future.
2. The high-water mark is not perfect, so don't make things complicated to try to make it so. It can't be perfect because stacks can grow depending on the timing of interrupt and function nesting and whether rarely run paths in the code have run or not. As an example, don't worry about handling the case where the task whose stack is being scanned (to set the HWM) is resumed/restarted. The scan will be suspended at some point, and the stack may grow past that point. When the scan is resumed it will stop at or below where it left off, and it will report the HWM to be a little lower than it really is. It shouldn't set the `SHWM_VALID` flag in this case. We could avoid this problem with a new task flag or by making the routine into an `ssr`, but those add complexity and it isn't worth it. We only need to worry about this when clearing a stack to be released, and this is handled by disconnecting it from the TCB first.
3. Don't make `process_next_stack()` into an `ssr`. Just make it a regular function that is called from a task.
4. Don't enqueue TCB's for stopped tasks whose stacks need to be released, because each task may be enqueued on a semaphore, exchange, etc. In v4 there will be an additional pointer in the TCB that could be used to enqueue them on a "ReleaseStack" queue, but until then, we can just scan the TCB's to find the ones whose stacks need to be released.
5. Do delayed release even when scanning is disabled (i.e. when building for release). Main benefit is that we can clear `tcb.stp` and `tcb.sbp`. Also it is good for consistency to have it run the same way for debug as for release. It would be faster for the scheduler to release the stacks directly to the stack pool, when they don't need to be scanned/cleared, but the first 2 reasons are more important for simplicity.

Decisions 10-8-03: (David & Ralph mtg)

1. Put `process_next_stack()` in the `smx` library. Put in `xmisc.c` for now. Can't condition code out since it isn't in the Protosystem (or `SHARED`). Ralph says ok that it will add more code size. Can only check `cf.stk_enh_test` to know whether to do scanning and clearing, so run-time is saved but not code size. In the future, maybe we will add a config option to `xdef.h` to enable or disable this code.