Detecting and Avoiding Stack Overflow in IoT/Embedded Systems

ThreadX RTOS users have five powerful tools that are able to help embedded applications avoid this system-crippling problem.

One of the toughest (and unfortunately common) problems in embedded systems is stack overflow and the collateral corruption or crash that it can cause. Often, the consequences of a stack overflow manifest themselves far removed from the cause of the overflow itself, making the cause that much more difficult to identify and fix. As a result, we have spent considerable effort inventing creative ways our customers can deal with this potential problem. ThreadX developers have an array of tools at their disposal to detect and even avoid stack overflow problems. These tools and techniques not only help developers avoid stack overflow due to inadequate stack memory allocation, they also help minimize RAM wasted by allocating excessive memory for thread stacks “just to be safe.” The following tools and techniques are discussed in this white paper:

- Manual stack inspection
- Kernel awareness and ThreadX stack analysis
- ThreadX run-time stack analysis
- IAR EWARM stack usage analysis
- TraceX stack analysis

Overview

In the C programming language, the stack—a region of memory in which local variables are located and function arguments are passed—is allocated by the programmer, with the amount of memory allocated based on factors such as machine architecture, OS, application design, and amount of memory available. If the program should require more memory for its stack than has been allocated, the stack overflows—without warning in most cases—which can corrupt other memory areas and often results in a program malfunction or even a crash. Such problems are very difficult to trace back to the stack overflow, causing programmers to expend considerable time and energy to find the underlying cause of the problem that the application exhibits. As a result, they tend to over-allocate stack memory as a precaution, “just to be safe.”

Traditionally, deciding how much memory to allocate for the stack has been a trial and error process. As widely respected industry commentator and consultant, Jack Ganssle, has observed:

In an RTOS, there is a separate stack for each thread, and each thread might have drastically different stack size needs. Making things even more challenging, stack overflows often affect a somewhat unrelated memory area – global variables, allocated memory, or another thread’s stack – and thus the subsequent problem does not manifest itself until much later than when the overflow occurred.

**Manual Stack Inspection**

The most obvious and basic technique to prevent stack overflows is to manually inspect the stack memory region and stack pointers for potential overflow. To facilitate this, ThreadX places a 0xEF data pattern throughout each thread’s stack. The idea here is to run the thread through its validation tests and then review all of the thread stacks. The non-0xEF byte closest to the start of the stack represents the high-water mark of that thread’s stack usage. Of course, if there are no remaining 0xEF data patterns in a thread’s stack, there is a high-probability that a stack overflow has occurred. The following figure shows an example thread stack with the 0xEF data pattern:

In addition to detection of stack overflow, manual stack inspection can be used to tune the stack size. For example, if a large area of unused stack space is found, it may indicate that the size for that thread’s stack is excessive. Of course, this analysis assumes that the test suite is exercising the worst case call tree depth for each thread. Note also that every thread stack must always have a minimum amount of unused memory in order to save its context if an interrupt occurs at the highest point of stack used. The exact amount needed varies by architecture, but is defined in each ThreadX port’s `readme_threadx.txt` file.
ThreadX Kernel Awareness and Stack Analysis
Most major embedded development tools provide what is called ThreadX kernel awareness. Such awareness provides a single-click, system level view of ThreadX resources. Most of the ThreadX-aware debuggers also provide thread stack analysis, which effectively automates the manual inspection technique described above. The following screen shot shows an example of IAR’s Embedded Workbench for ARM (EWARM), with its ThreadX kernel awareness for a Cortex-M3 target. Specifically, this illustration shows the information related to the “thread” object:
The key column is the **Stack Usage** column. The difference between the **Stack Size** and the **Stack Usage** column yields the remaining stack size. For example, in the example shown thread 5 has a stack size of 512 bytes, while its current usage is 136 bytes. This means that there are 376 free bytes on thread 5’s stack.

This information is obtained by the debugger automatically examining the thread’s stack memory for the 0xEF data pattern. The stack memory for thread 5 ranges from address 0x20001a58 through 0x20001c57. The figure below shows the memory dump of this stack area:

![Memory Dump](image)

Manual inspection of the thread’s stack memory area shows that the lowest address not having the 0xEF data pattern is 0x20001bd0. Subtracting this from the ending stack address of 0x20001c57 yields the reported used stack size of 136 bytes. Of course, the IAR debugger takes care of all this manual work, providing this valuable information via a simple mouse click.

**ThreadX Run-time Stack Checking**
The automated debugger stack checking is a powerful feature, but it does have some shortcomings. One such shortcoming is that the stack overflow detection is still made by the developer. If the developer doesn’t see the overflow, the problem may go undetected. Also, there is no mechanism to stop the system immediately when the overflow occurs. This is where ThreadX run-time stack checking comes into play.
By default, the run-time stack checking in ThreadX is disabled. To enable run-time stack checking, simply build the ThreadX library with `TX_ENABLE_STACK_CHECKING` defined. With stack checking enabled, ThreadX examines the stack of every thread being scheduled prior to execution. In addition, every suspending thread’s stack is analyzed. If an overflow condition exists, ThreadX immediately calls the internal ThreadX default stack error handler `_tx_thread_stack_error_handler`.

Alternatively, the application may register its own stack error handler by supplying a callback function to `tx_thread_stack_error_notify`.

In addition to detecting overflow conditions, ThreadX run-time stack checking keeps track of the high water mark of stack usage. This address subtracted from the ending address yields the amount of stack spaced used. The following figure shows the thread control block for thread 5. The last structure member (named `tx_thread_stack_highest_ptr`) contains the highest used address of the thread’s stack. In this example, it matches the same value derived by the IAR kernel awareness.

The principal advantage of ThreadX run-time stack checking is that overflow detection occurs closer to the point where the overflow occurred and it does not rely on a developer spotting the overflow condition. It also provides the high water mark so that stack size tuning is possible. Finally, there are some environments that simply don’t have automated debugger stack analysis, making ThreadX run-time checking the only option.
IAR EWARM Stack Usage Analysis

Another useful tool for determining proper stack memory use and allocation is IAR EWARM’s Stack Usage Analysis. Under the right circumstances, the IAR EWARM linker can accurately calculate the maximum stack usage for each call graph, starting from the program start, interrupt functions, tasks etc. (each function that is not called from another function, in other words, a root). If you enable stack usage analysis, a stack usage chapter will be added to the linker map file, listing for each call graph root the particular call chain which results in the maximum stack depth.

In general, the compiler will generate this information for each C function, but if there are indirect calls (calls using function pointers) in your application, you must supply a list of possible functions that can be called from each calling function. If you use a stack usage control file, you can also supply stack usage information for functions in modules that do not have stack usage information.

Result of an Analysis—The Map File Contents

When stack usage analysis is enabled, the linker map file contains a stack usage chapter with a summary of the stack usage for each call graph root category, and lists the call chain that results in the maximum stack depth for each call graph root. This is an example of what the stack usage chapter in the map file might look like:

```plaintext
*** STACK USAGE ***
Call Graph Root Category Max Use Total Use
------------------------------- ------- -------
interrupt 104 136
Program entry 168 168

Program entry
"_ iar_program_start": 0x000085ac
Maximum call chain 168 bytes

  " iar_program_start" 0
  " cmain" 0
  "main" 8
  "printf" 24
  "_PrintfTiny" 56
  " _Prout" 16
  "putchar" 16
  " _write" 0
  " _fwrite" 0
  " iar_sh_stdout" 24
  " iar_get_ttio" 24
  " iar_lookup_ttioh" 0

interrupt
"FaultHandler": 0x00008434
Maximum call chain 32 bytes
"FaultHandler" 32

interrupt
"IRQHandler": 0x00008424
Maximum call chain 104 bytes
"IRQHandler" 24
"do_something" in suexample.o [1] 80
```
The summary contains the depth of the deepest call chain in each category as well as the sum of the depths of the deepest call chains in that category. Each call graph root belongs to a call graph root category to enable convenient calculations in check that directives.

Call Graph Log
To help you interpret the results of the stack usage analysis, there is a log output option that produces a simple text representation of the call graph (--log call_graph).

Example output:
Program entry:
0 __iar_program_start [168]
 0 __cmain [168]
 0 __iar_data_init3 [16]
  8 __iar_zero_init3 [8]
  16 - [0]
  8 __iar_copy_init3 [8]
  16 - [0]
 0 __low_level_init [0]
 0 main [168]
 8 printf [160]
  32 _PrintfTiny [136]
  88 _Prout []80
  104 putchar [64]
  120 __write []48
  120 __dwrite []48
   120 __iar_sh_stdio [48]
    144 __iar_get_tti [24]
     168 __iar_lookup_tti [0]
    120 __iar_sh_write [24]
     144 - [0]
  88 __aeabi_uidiv []0
  88 __aeabi_idiv0 []0
  88 strlen []0
 0 exit []8
 0 _exit []8
  0 __exit []8
 0 __iar_close_tti [8]
   0 __iar_lookup_tti [0] ***
 0 __exit []8 ***

Each line consists of the following information:
- The stack usage at the point of call of the function
- The name of the function, or a single '-' to indicate usage in a function at a point with no function call (typically in a leaf function)
- The stack usage along the deepest call chain from that point. If no such value could be calculated, "[---]" is output instead. "***" marks functions that have already been shown.

ThreadX Stack Usage Analysis
Here is a linker map chapter for Stack Usage Analysis performed on the ThreadX standard demo application, “demo_threadx”
*** STACK USAGE ***
Call Graph Root Category | Max Use | Total Use
------------------------ | -------- | --------
Program entry            | 300      | 300      
thread                   | 276      | 1544     
Uncalled function        | 260      | 264      

Program entry
"__iar_program_start": 0x000029cd

Maximum call chain 300 bytes

"__iar_program_start" 0
"__cmain" 0
"main" 8
"_tx_initialize_kernel_enter" 8
"tx_application_define" 32
"_tx_byte_allocate" 32
"_tx_thread_system_suspend" 16
"_tx_thread_system_return" 204

thread
"thread_0_entry": 0x00000655

Maximum call chain 260 bytes

"thread_0_entry" 8
"_tx_event_flags_set" 40
"_tx_thread_system_preempt_check" 8
"_tx_thread_system_return" 204

thread
"thread_1_entry": 0x00000679

Maximum call chain 252 bytes

"thread_1_entry" 8
"_tx_queue_send" 24
"_tx_thread_system_suspend" 16
"_tx_thread_system_return" 204

thread
"thread_2_entry": 0x000006a5

Maximum call chain 252 bytes

"thread_2_entry" 8
"_tx_queue_receive" 24
"_tx_thread_system_suspend" 16
"_tx_thread_system_return" 204

thread
"thread_3_and_4_entry": 0x000006d9

Maximum call chain 244 bytes

"thread_3_and_4_entry" 8
"_tx_semaphore_get" 16
"_tx_thread_system_suspend" 16
"_tx_thread_system_return" 204

thread
"thread_5_entry": 0x00000719

Maximum call chain 260 bytes

"thread_5_entry" 16
"_tx_event_flags_get" 24
"_tx_thread_system_suspend" 16
"_tx_thread_system_return" 204

thread
"thread_6_and_7_entry": 0x00000745

Maximum call chain 276 bytes

"thread_6_and_7_entry" 8
"_tx_mutex_put" 24
"_tx_mutex_priority_change" 24
"_tx_thread_system_suspend" 16
"_tx_thread_system_return" 204

Uncalled function
"SysTick_Handler": 0x00001929

Maximum call chain 4 bytes

"SysTick_Handler" 4

Uncalled function
"_tx_thread_time_slice": 0x00002409

Maximum call chain 0 bytes

"_tx_thread_time_slice" 0

Uncalled function
"_tx_timer_expiration_process": 0x000022ed

Maximum call chain 260 bytes
"_tx_timerExpirationProcess" 24
"_tx_threadTimeout" 8
"_tx_eventFlagsCleanUp" 16
"_tx_threadSystemResume" 8
"_tx_threadSystemReturn" 204

The following functions were excluded from stack usage calculations:

"_tx_mutexThreadRelease": 0x000024e3
"_tx_threadSchedule": 0x000019dd
"_tx_threadShellEntry": 0x000027a9

IAR EWARM Call Graph Output for demo_threadx

thread:
0 thread_0_entry [260]
  8 _tx_eventFlagsSet [252]
    48 _tx_threadSystemPreemptCheck [212]
      56 _tx_threadSystemReturn [204]
        260 - [0]
    48 _tx_threadSystemResume [212]
      56 _tx_threadSystemReturn [204] ***
      56 _tx_timerSystemDeactivate [4]
        60 - [0]
  8 _tx_threadSleep [228]
  16 _tx_threadSystemSuspend [220]
    32 _tx_threadSystemReturn [204] ***
    32 _tx_timerSystemActivate [4]
      36 - [0]

thread:
0 thread_1_entry [252]
  8 _tx_queueSend [244]
    32 _tx_threadSystemResume [212] ***
    32 _tx_threadSystemSuspend [220] ***

thread:
0 thread_2_entry [252]
  8 _tx_queueReceive [244]
    32 _tx_threadSystemResume [212] ***
    32 _tx_threadSystemSuspend [220] ***

thread:
0 thread_3_and_4_entry [244]
  8 _tx_semaphoreGet [236]
    24 _tx_threadSystemSuspend [220] ***
  8 _tx_semaphorePut [228]
    24 _tx_threadSystemResume [212] ***
  8 _tx_threadSleep [228] ***
thread:
  0 thread_5_entry [260]
    16 _tx_event_flags_get [244]
    40 _tx_thread_system_suspend [220] ***

thread:
  0 thread_6_and_7_entry [276]
    8 _tx_mutex_get [260]
    24 _tx_mutex_priority_change [244]
    48 _tx_thread_system_resume [212] ***
    48 _tx_thread_system_suspend [220] ***
    24 _tx_thread_system_suspend [220] ***
    8 _tx_mutex_put [268]
    32 _tx_mutex_prioritize [236]
    56 _tx_thread_system_preempt_check [212] ***
    32 _tx_mutex_priority_change [244] ***
    32 _tx_thread_system_preempt_check [212] ***
    32 _tx_thread_system_resume [212] ***
    8 _tx_mutex_priority_change [244] ***
    32 _tx_thread_system_resume [212] ***
    32 _tx_thread_system_resume [212] ***
    32 _tx_thread_system_resume [212] ***
    8 _tx_thread_system_preempt_check [212] ***
    24 _tx_thread_system_suspend [220] ***

Uncalled:
  0 SysTick_Handler [4]
    4 - [0]

Uncalled:
  0 _tx_thread_schedule [---] (no info)

Uncalled:
  0 _tx_mutex_thread_release [-excluded-]
    8 _tx_mutex_put [268] ***

Uncalled:
  0 _tx_thread_shell_entry [---]
    8 Indirect call [---]
    8 _tx_thread_system_suspend [220] ***

Uncalled:
  0 _tx_thread_time_slice [0]

Uncalled:
  0 _tx_timer_expiration_process [260]
    24 _tx_thread_timeout [236]
    32 _tx_queue_cleanup [220]
    40 _tx_thread_system_resume [212] ***
    32 _tx_semaphore_cleanup [220]
    40 _tx_thread_system_resume [212] ***
    32 _tx_mutex_cleanup [220]
    40 _tx_thread_system_resume [212] ***
    32 _tx_block_pool_cleanup [220]
    40 _tx_thread_system_resume [212] ***
32 _tx_byte_pool_cleanup [220]
40 _tx_thread_system_resume [212] ***
32 _tx_event_flags_cleanup [228]
48 _tx_thread_system_resume [212] ***
32 _tx_thread_system_resume [212] ***
24 _tx_timer_system_activate [4] ***

Program entry:
0 __iar_program_start [300]
0 __cmain [300]
0 __iar_data_init3 [12]
8 __iar_zero_init3 [0]
8 __iar_copy_init3 [4]
12 - [0]
0 __low_level_init [0]
0 main [300]
8 _tx_initialize_kernel_enter [292]
16 _tx_initialize_high_level [24]
24 _tx_thread_initialize [16]
32 __iar_Memset [8]
32 __iar_Memset_word [8]
40 - [0]
24 _tx_timer_initialize [16]
32 __iar_Memset [8] ***
16 _tx_initialize_low_level [0]
16 _tx_thread_schedule [---] (no info) ***
16 tx_application_define [284]
48 _tx_block_allocate [236]
64 _tx_thread_system_suspend [220] ***
48 _tx_block_pool_create [32]
72 __iar_Memset [8] ***
48 _tx_block_release [220]
56 _tx_thread_system_resume [212] ***
48 _tx_byte_allocate [252]
80 _tx_byte_pool_search [28]
108 - [0]
80 _tx_thread_system_suspend [220] ***
48 _tx_byte_pool_create [32]
72 __iar_Memset [8] ***
48 _tx_event_flags_create [24]
64 __iar_Memset [8] ***
48 _tx_mutex_create [24]
64 __iar_Memset [8] ***
48 _tx_queue_create [32]
72 __iar_Memset [8] ***
48 _tx_semaphore_create [24]
64 __iar_Memset [8] ***
48 _tx_thread_create [244]
80 __iar_Memset [8] ***
80 _tx_thread_stack_build [0]
80 _tx_thread_system_preempt_check [212] ***
80 _tx_thread_system_resume [212] ***
48 _tx_thread_create [244] ***
48 _tx_thread_create [244] ***
48 _tx_thread_create [244] ***
48 _tx_thread_create [244] ***
48 _tx_thread_create [244] ***
0 exit [8]
0 __exit [8]
8 - [0]
TraceX Stack Analysis

Another stack analysis tool available to ThreadX users is TraceX. Although the main purpose of TraceX is to provide a system level, graphical view of what the application is doing, TraceX also analyzes the stack usage for each thread represented in the trace buffer. TraceX does not provide a worst-case stack size for the entire thread execution, but only the worst case stack usage within the captured trace. For example, the following trace shows thread 5’s execution in the trace buffer:

Event number 184 is thread 5’s call to `tx_event_flag_get`, which in turn suspends as shown by event 185 and the subsequent execution of other threads. The stack analysis of this trace buffer, selected by View -> Thread Stack Usage:
The TraceX view of thread stack usage shows that thread 5 has a minimal available stack of 416 bytes (or used stack of 96 bytes). The reason TraceX shows less stack used than the other methods is that the stack sampled in the trace buffer does not include the stack required to save the thread’s context. However, it still provides a useful cross checking of the stack usage for thread execution captured within the trace buffer.

**Summary**

ThreadX users must still deal with stack overflow issues as well as attempting to ascertain the minimal amount of stack space required for each thread. However, ThreadX users have unparalleled stack analysis tools at their disposal – eliminating much of the guesswork and hope!