Helping you avoid stack overflow crashes!

One of the toughest (and unfortunately common) problems in embedded systems is stack overflows and the collateral corruption that it can cause. As a result, we have spent considerable effort inventing creative ways our customers can deal with this 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
- StackX stack depth 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:

"With experience, one learns the standard, scientific way to compute the proper size for a stack: Pick a size at random and hope."

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. Figure 1 shows an example thread stack with the 0xEF data pattern.

![Figure 1: Example thread stack with 0xEF data pattern](image)

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.

Kernel Awareness and ThreadX 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, with its
ThreadX kernel awareness for a Cortex-M3 target. Figure 2 contains an illustration that
shows the information related to the “thread” object.
The key fields in this display are the *Stack Ptr*, *Stack Start*, *Stack End*, *Stack Size*, and *Stack Usage* columns. Of course 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.

As mentioned previously, 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. Figure 3 shows the memory dump of this stack area.

![Memory dump for thread 5 stack area](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 occurs, 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, 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 space used. Figure 4 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.
The previous techniques are very useful to empirically attempt to calculate stack usage, and help the developer allocate enough memory for each stack. However, there is one major problem with analyzing stack usage based upon examination of the stack pointer and/or the erosion of the default stack pattern in memory. The problem is that the accuracy of the analysis is dependent on the test suite generating the worst-case stack usage. If the test does not cause worst-case stack usage, the analysis won’t be able to calculate the actual worst-case stack usage.

In order to address this problem, Express Logic offers StackX, a static analysis tool that examines and analyzes an application’s executable file, builds a graphical call tree that shows the nested function call possibilities, and calculates the worst-case stack depth throughout the call tree. Since StackX runs on the host and simply analyzes the ELF file produced by the development tools to determine stack usage, no code or test suite is required to run on the target. In addition, no run-time overhead is introduced on the
target either, because the stack size is already guaranteed to be correct before any execution on the target.

In the previous example, we have calculated that thread 5 only has used 136 bytes, which is correct. However, running StackX on the same ELF file used in the previous discussion, the worst-case stack size of thread 5 is 152 bytes, as shown in Figure 5.

![Figure 5: Worst-case stack size of thread 5 is 152 bytes](image)

From this information, it is important to cross check the static analysis from StackX with the empirical approaches discussed previously. In fact, the call tree generated by StackX provides a good target for the run-time test suite, which then helps ensure the accuracy of the empirical stack analysis methods.
Currently, StackX is available for ARM targets and supports ELF files generated by all popular development tools.

**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, consider Figure 6, which shows the trace of thread 5’s execution in the trace buffer:

![Figure 6: Trace showing the execution of thread 5 in the trace buffer](image-url)
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 appears in Figure 7.

![TraceX Thread Stack Usage](image)

**Figure 7: Thread stack usage**

As illustrated in Figure 7, 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!