ThreadX® Modules Ease Memory Constraints in Small Embedded Systems

Concern over physical size, power consumption and battery life place constraints on memory in deeply embedded systems. These constraints can be eased by using techniques commonly found in large systems but not often seen in small RTOSes

Introduction

Embedded systems have always found themselves challenged with regard to the amount of memory available for software and data. More is always desirable, based on the benefits to functionality, capacity, and redundancy, yet at the same time undesirable because of the impact to size, power, heat and cost. Over time, various compromises, and advances in technology, have changed the picture somewhat, but there has always been this tension that will likely always be a limiting factor. After all, Parkinson's Law tells us that memory demand will expand to exhaust available capacity. More memory leads to more demand; hence, we'll always want to put more into a system, no matter what the memory limit might be.

To meet system functionality requirements, while remaining within the limits of memory availability, embedded developers from time to time borrow big system features to better handle certain situations. Again glancing back to the mainframe days of the 1970s, available memory could be used more efficiently if it were "time shared" among various pieces of independent code. We called these code sections "overlays," and they were loaded into memory on demand from mass storage, over-laying previously resident code. Of course, this introduced significant overhead, but it was a very effective approach for mainframe systems where such overhead was relatively harmless. With this method, you could introduce virtually unlimited functionality, structured as a set of overlays, without expanding physical memory at all.

A modern technique for functional expansion has worked its way into everyday life through the "App." An App, or Application, is structured in a manner that allows it to be loaded onto a running system. When a system must be updated without removal from service, an App can be downloaded into the system, adding to the memory-resident code that can be run at any time.

Both the Overlay and App can be leveraged to give embedded systems a boost in meeting the constraints of a memory system that's just not as big as developers want. In fact, they can be blended in a manner that suits embedded systems quite well, and provides some measure of relief from memory constraints.

Modern Embedded Systems Are Still Memory-Constrained

Many of today's embedded systems used in consumer, medical and industrial applications face memory constraints. These systems run with a real-time operating system (RTOS), and often with relatively small amounts of memory.

RTOSes come in all sizes and flavors, from the large (256KB – 1MB), like Wind River's VxWorks®, to the super-compact (under 20KB), like Express Logic's ThreadX®. Large RTOSes offer many features adapted from desktop systems that are typically not available in the super-small RTOSes because such features generally require a larger amount of code, and result in a slower real-time response. The small RTOS generally operates as a library of services, linked with the application into a single executable file (see Figure-1). The application references the services it needs through an API. A small RTOS can provide these services with low overhead, and in a small memory footprint, often less than 20KB.
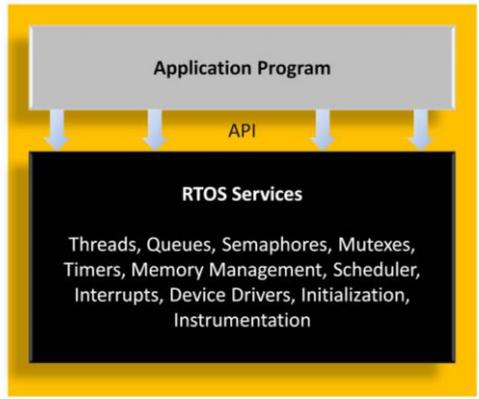

Figure-1

This single executable file is efficient in both time and space since all of the code is statically linked. But it lacks flexibility since any changes to the application or RTOS require re-building, re-linking, and a re-download/flash of the new image in its entirety.

In contrast, desktop operating systems such as Windows and Linux, and large RTOSes, such as VxWorks, have two-piece "OS/Application" architectures (see Figure-2) where target memory is allocated for the OS, and additional memory is allocated for each application that is to be run with the OS, usually as needed.
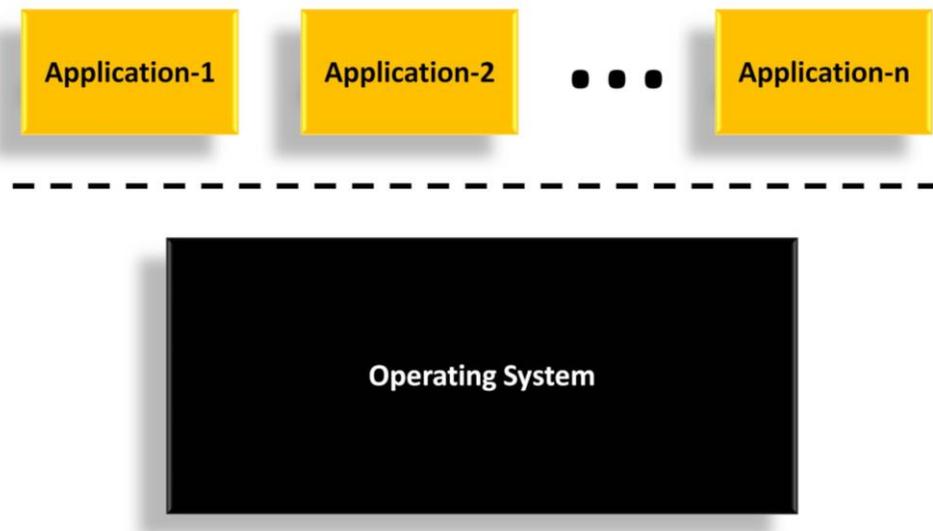

Figure-2

In this architecture, there is a resident kernel, containing all the OS services available to applications, all linked into a distinct, memory-resident executable. This kernel executable boots the system and runs continuously, providing a foundation for applications which are loaded and run in dynamically-allocated memory. Often in these systems, a virtual memory architecture provides demand paging to and from mass storage on desktop systems or multi-user separation in embedded systems. Virtual memory also can enable use of non-contiguous memory fragments that otherwise might grow over time and reduce or exhaust available memory.

Applications that are not linked with the RTOS must use a different means of accessing RTOS services from the one used by functions within a statically linked executable. This alternate access is necessary because the application is developed independently of a particular hardware environment, and simply cannot predict the memory address of the OS service routine they wish to use. In most cases, these applications access the required OS service routines via a "trap" mechanism that does not require the calling program to be linked with the service it calls (See Figure-3).
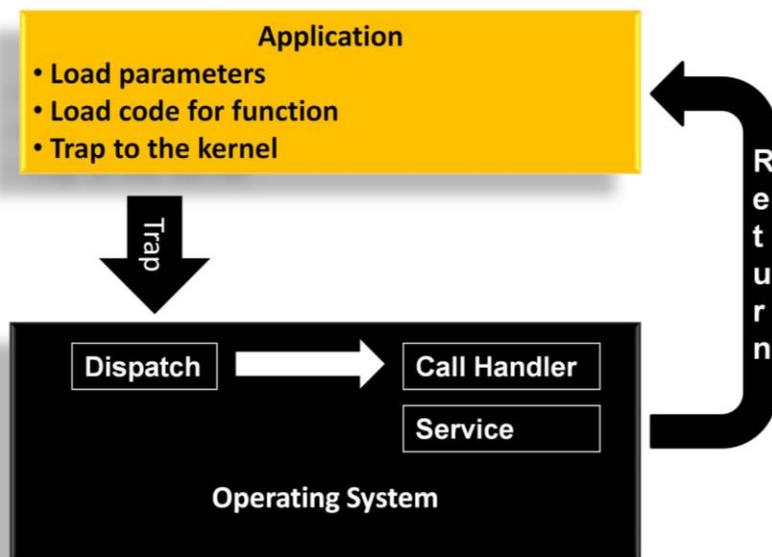
Figure-3

A "trap" is a software interrupt that can be used to catch errors before they propagate or to stop the system if it's impossible to perform a requested operation. A divide by zero or the loading of unaligned data illustrate possible illegal operations that might create a trap and activate a trap handler. Similarly, an intentional trap can be generated through an instruction such as a "software interrupt" or "swi" instruction, and used to communicate between non-linked program elements.

The trap process requires interrupt servicing, processing, a function call, and then the same in reverse. For a desktop or large RTOS, this overhead is insignificant given the relatively lengthy response time expected of such systems. But, for a small RTOS used in hard real-time, deeply embedded applications, low overhead response is essential.

Bringing Dynamic Memory Use to the Small RTOS
To provide the ability to add functionality dynamically, a "Module" structure can be used (See Figure-4). A Module is a collection of one or more application threads, not linked with the kernel, but instead built as a separate executable that can be loaded into target memory when needed, and overwritten when no longer needed – just like an overlay. This approach produces a "time shared memory" capability that enables system functionality to exceed physical memory. Such Modules can be downloaded into memory via a wireless network or local mass storage, like an App.
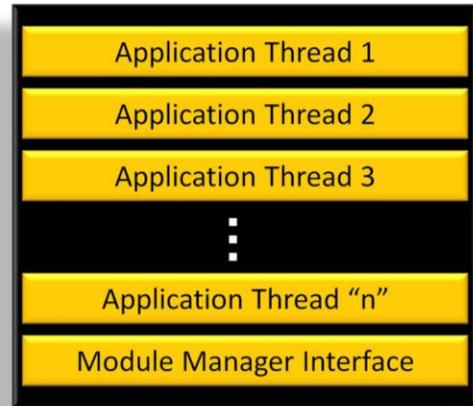

Figure-4

The Modules use kernel services via an interface with a "Module Manager," an agent within the kernel image that loads and initializes a Module as well as fielding all Module requests for RTOS services. The Module Manager is responsible for allocating memory for a Module, loading it via a communications or storage interface, and keeping track of its location in memory. The Module Manager also informs the Module of the addresses of the RTOS routines the Module's application threads need to access, avoiding the need for a trap mechanism, and providing a very efficient calling interface. The Module Manager also can overwrite one Module with another, if vacant memory is not sufficient to hold the new Module – much like the overlays of old. Threads within Modules make RTOS service calls exactly as they would make calls if the service function were directly linked with the application. In the Module, however, these calls are handled by an interface component that communicates with the Module Manager. The trap mechanism is avoided, enabling a low overhead service call interface (See Figure-5).
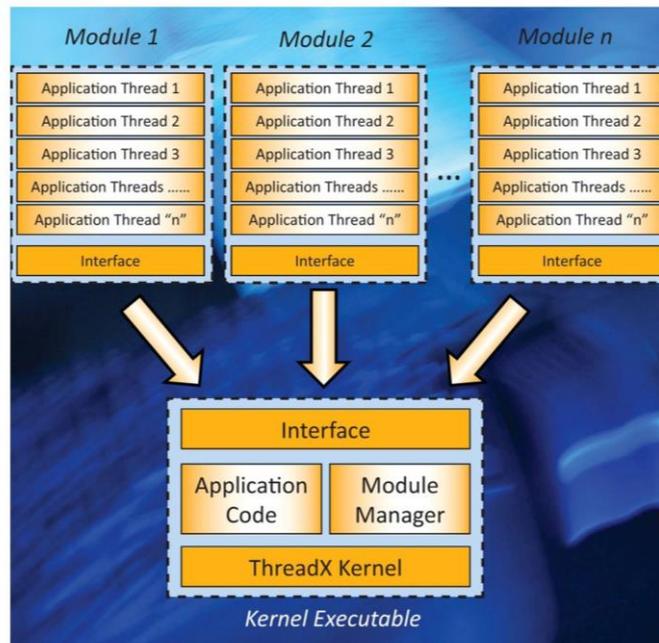

Figure-5

Application threads still can be linked with the kernel and reside with the kernel in target memory as part of the RTOS's executable image. This option enables the system designer to balance the need for minimum overhead with memory conservation, using the more efficient, but more memory hungry approach only where absolutely necessary.

Securing ThreadX Modules With Memory Protection
Modules also provide a convenient means for memory to be guarded against inadvertent, unintended access. They also enable developers to distinguish between "more trusted" and "less trusted" Modules, applying the appropriate level of protection as demanded by each Module. Once a Module's code has been proven to be "trusted," for example, it can be given access to the entire system, including hardware control registers, other threads, and the RTOS, without risk. This trust requires extensive testing and perhaps certification as well. For less critical code, a more practical approach might be to allow that code to access only the memory within its own Module and nothing outside.
Despite testing, there is risk of accidental stack or memory corruption due to an erroneously calculated pointer or array limit, or stack overflow. These faults can be catastrophic and difficult to find, especially if only one portion of the development team is familiar with the offending Module. The extreme difficulty in tracing to the source of such errors makes it all the more important to avoid them. This protects the rest of memory, including other threads and the RTOS, from faults within the un-trusted Module code. While all code is tested, a 100% assurance of correctness is expensive and time consuming. It may be more cost-effective, depending on the code's function in the system, to simply isolate it from other code just in case it "misbehaves."

Using the system's MMU or MPU, memory boundary registers can be set to constrain an un-trusted Module's code to accesses within a given memory region (See Figure-6).
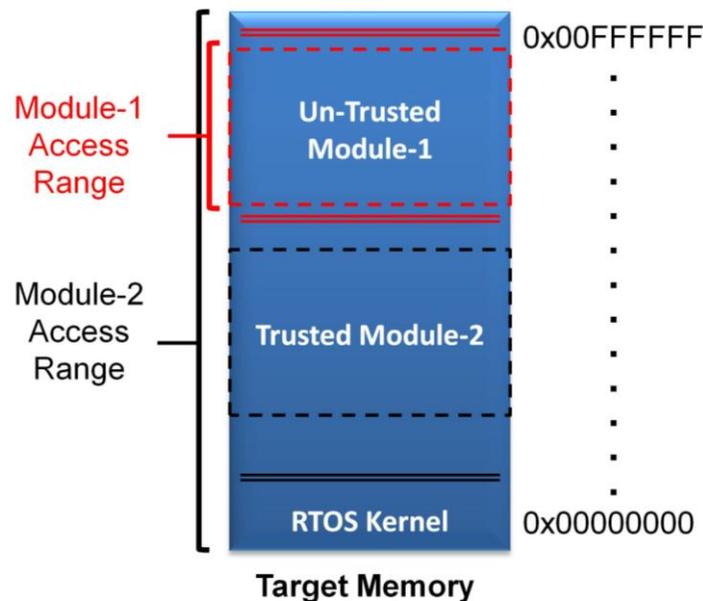

Figure-6

With boundary register settings, any attempt to access memory locations outside the specified range will result in a trap. In that event, the trap handler can take appropriate action – terminate the offending thread, restart it, alert the system, or halt the system, all depending on the criticality of the offending code. The trap also provides excellent debugging information, as it identifies the code making the errant access, allowing it to be fixed.

Summary

ThreadX Modules enable the RTOS to dynamically load and run additional application threads beyond those linked with the kernel. Applications gain increased functionality without the cost of an increased footprint or additional memory, and while retaining an efficient service call interface. This technique also provides on-demand reconfiguration and updates for deployed systems. ThreadX Module technology ideally suits situations where application code size exceeds available memory, when new Modules need to be added after a product is deployed, or when partial firmware updates are required.

Another advantage of downloading separate application Modules is that each Module can be more readily developed by its own team or individual programmer. Each team can then focus on one aspect of a product's functionality, without having to be concerned with other details.