

When an Irresistible Force Meets an Immovable Object

By John A. Carbone, VP Marketing, Express Logic, Inc.

ARM® processor-based Systems on a Chip (SoCs) abound in the areas of consumer electronics, mobile devices, and wireless communications. The ARM IP licensing model is amenable to the design of SoCs that incorporate an ARM core, along with companion technology such as mobile baseband radio, WiFi® or Bluetooth®. The ARM low-power, high-performance technology adds further appeal to SoC designers, leading to its widespread adoption. SoCs generally incorporate limited memory, and the popular ARM Cortex™-M microcontroller series upon which many SoCs are based, incorporate limited memory on-chip, making for an overall modest amount of memory available for an application. While additional memory, external to the Cortex-M3, might be configured either on SoC or elsewhere, that memory is not accessed as rapidly as on-chip flash. This makes it desirable to fit critical real-time sensitive code in the limited on-chip space. This compact memory model contrasts with the fiercely competitive demand for more and more functionality in the devices using these SoCs, bringing us to the irresistible force (feature demand) meeting the immovable object (limited memory).

To meet the demand for ever-increasing system functionality while remaining within the limits of memory availability, embedded developers might want to borrow some big system features to make better use of available physical memory. For instance, mainframe and desktop systems routinely “time share” available memory among various applications, or even portions of a single application. Applications can be loaded into memory on demand from mass storage, over-laying previously resident code. Such an “overlay” technique was common among mainframes and minicomputers where memory was too pricey to have in abundance, and demand-paged virtual memory hadn’t yet been introduced. Both overlays and demand-paging involve copying code from mass storage to physical memory as needed, whether a page at a time, or a block of code at a time. These approaches introduce significant overhead, but are very effective for mainframe and desktop systems where such overhead is relatively harmless – i.e., they’re not real-time systems. With this method, developers could introduce virtually unlimited functionality without expanding physical memory to the extent necessary to contain all the code at one time. These techniques today are the norm in all PCs and even in some large embedded systems as well.

Another 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 via wireless transfer onto a running system, sort of like a “network overlay.” In the current use mode, apps remain in memory once loaded, but if memory is full and the user desires another app, an existing app can be manually removed to make room for the new one. Also, when a system must be updated without removal from service, an entire OS image can be downloaded into the system, again over a wireless connection.

The overlay, demand paging and app technologies work well in PCs and large systems where mass storage is plentiful and where real-time responsiveness is minimal. However, such is not the case in the world of the SoC. Even so, these technologies can be leveraged to give real-time SoC-based, embedded systems a boost in meeting the demands of a memory-constrained 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, providing a measure of relief from memory constraints. While desktop OSes today support demand-paged virtual memory, dynamically downloadable apps or overlays, and many other technologies that help use memory, such technologies are only found in the larger RTOSes and not those commonly used in memory-limited SoCs. Indeed, not all RTOSes are alike in their memory footprint and real-time responsiveness, since all applications are not alike in their needs in these areas.

Accordingly, RTOSes come in all sizes and flavors, from the large (256KB – 1MB), like Wind River’s VxWorks®, to the super-compact (under 10KB), like Express Logic’s ThreadX®. Some may also consider using Linux, but it demands even more memory than the largest RTOS. Linux and the large RTOSes offer many features adapted from desktop systems that are typically not available in super-small RTOSes because such features require a larger amount of code and result in a slower real-time response. The small RTOS generally operates as a library of services, where only those services actually used by the application are linked with the application into a single executable file (see Figure 1). The application references the services it needs through an API, making direct function calls that are extremely efficient. A small

RTOS can provide these services with low overhead and in a small memory footprint that's often less than 20KB, and in extreme cases even less than 4KB, depending on the services actually used by the application.

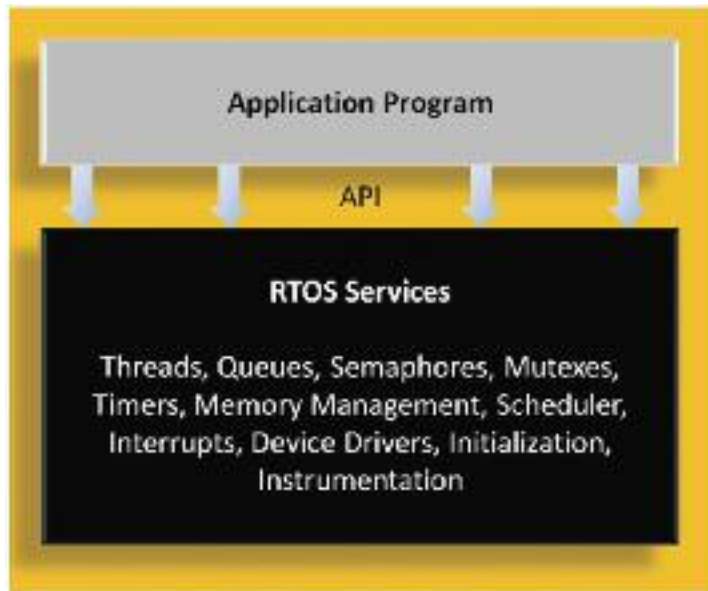


Figure 1

Because all of the code is statically linked, this single executable file is efficient in both time and space. 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.

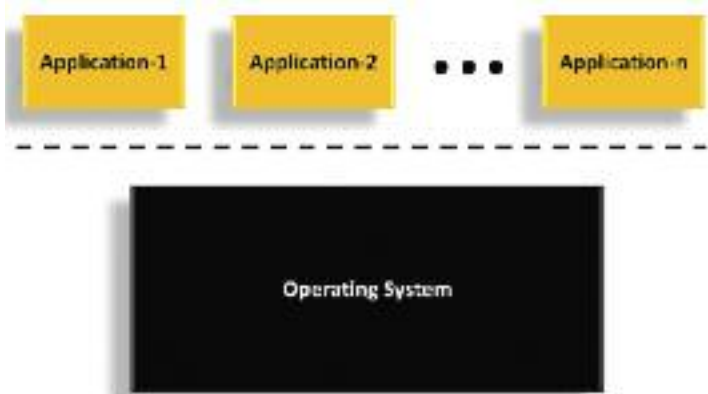


Figure 2

In this architecture, there is a resident kernel, containing all the OS services available to applications, whether used by any one application or not, all linked into a distinct, memory-resident kernel 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,

the virtual memory architecture provides demand paging to and from mass storage (routinely 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, whether or not demand paging is available.

Unfortunately, RTOSes employing this type of architecture must provide a mechanism for applications to call upon OS services in the absence of a direct function call, since the service functions are not linked with the application. Often, a trap mechanism is used, whereby the requesting application intentionally performs an operation that results in a processor exception or interrupt, and the OS interprets the exception as a request for service. The OS, then, causes the service function to be performed, and communicates back to the requesting application with the results. This involves the system's interrupt structure, ISR code, and a handler for this type of exception. In addition to expanding total code footprint, it also introduces a less efficient mechanism for accessing OS services. Both of these are shortcomings that SoC designers would like to avoid. Yet, they also would like to be able to add functionality dynamically, as do the large systems that employ this mechanism.

How to Enrich the SoC

To add functionality dynamically, but still avoid the overhead of a typical trap interface, ThreadX uses an "Application Module" technology. An application module (see Figure 3) is a collection of one or more application threads, not linked with the kernel, but built as a separate executable that can be loaded into target memory when needed, and overwritten when no longer needed – just like an overlay or an application in a large RTOS. 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 wired or wireless network (like an app), or from local mass storage (like an overlay).

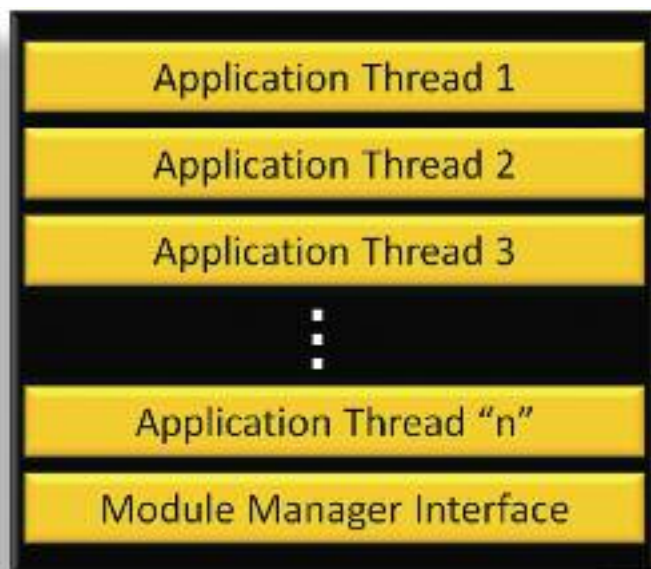


Figure 3

To reduce the large overhead of a trap interface, application modules use kernel services via a customized interface with a “module manager,” an agent within the ThreadX kernel that loads and initializes a module as well as fielding all module requests for RTOS services. The module manager allocates memory for a module, loading it via a communications or storage interface, and keeps track of its location in memory. As part of a “registration” process, the module manager informs the newly loaded module of the addresses of the RTOS routines that the module’s application threads need to access. This registration is in effect a dynamic linking that enables direct function call interface between applications and ThreadX services. 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 if the service function were directly linked with the application. In the module, however, these calls are handled by an interface component that is configured by the module manager. The trap mechanism is avoided, enabling a low overhead service call interface (see Figure 4). While not quite as efficient as a directly-linked function call interface, the module interface is far more efficient than a trap, and perfectly suitable for real-time systems demanding low overhead.

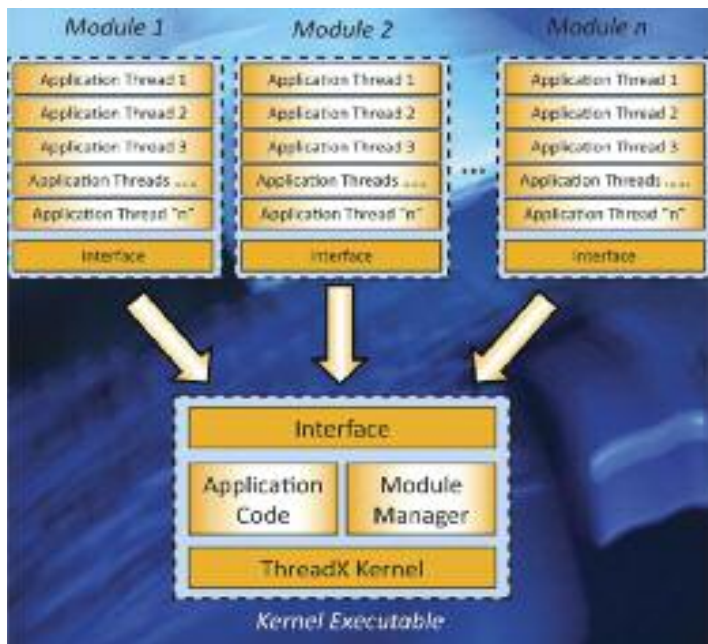


Figure 4

If desired, 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.

Summary

Downloadable application modules enable ThreadX dynamically to load and run additional application threads beyond those linked with the kernel. Applications gain increased functionality while retaining an efficient service call interface. This technique also

provides on-demand reconfiguration and updates for deployed systems.

Downloadable application 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.

END

Tools for Software Testing..

Continued from page 27

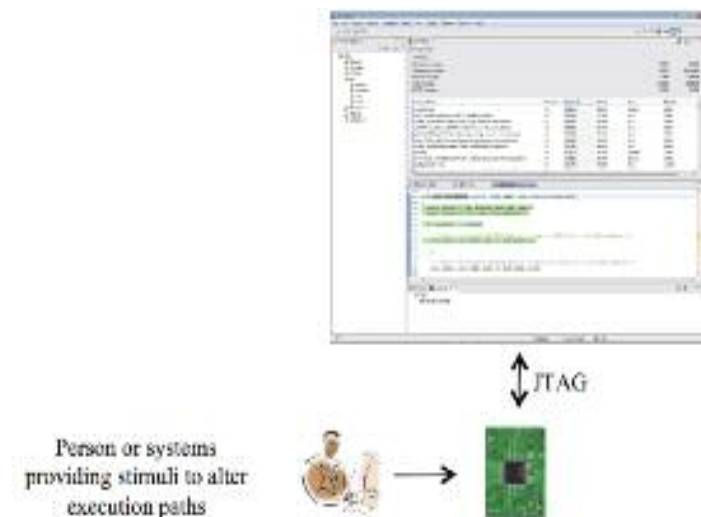


Figure 10: Atollic TrueANALYZER performs in-target code coverage analysis.

The fact that the code coverage analysis is performed as the application run in the target system is critically important. There is no guarantee that tests conducted in a simulated environment, say on a PC, will faithfully emulate the interaction of all actors in the outside world with the target system.

Summary

Software quality is getting an increased focus in the embedded industry and now deeply integrated tools are available for in-target test automation and test quality measurement of software running in ARM processor-based boards.

Being integrated in the Atollic TrueSTUDIO C/C++ IDE for ARM® development, the Atollic TrueVERIFIER and Atollic TrueANALYZER products enable ARM software developers to use a uniquely powerful tool suite for delivering ARM processor-based embedded projects with a much higher software quality.

More information on Atollic and the Atollic products are available at www.atollic.com.



eSOL Component Real-time OS Platform

INTEGRATED SOFTWARE PLATFORM

EMBEDDED QUALITY, VERIFIED RELIABILITY

– Backed by scalable OS and 30 years proven track record –

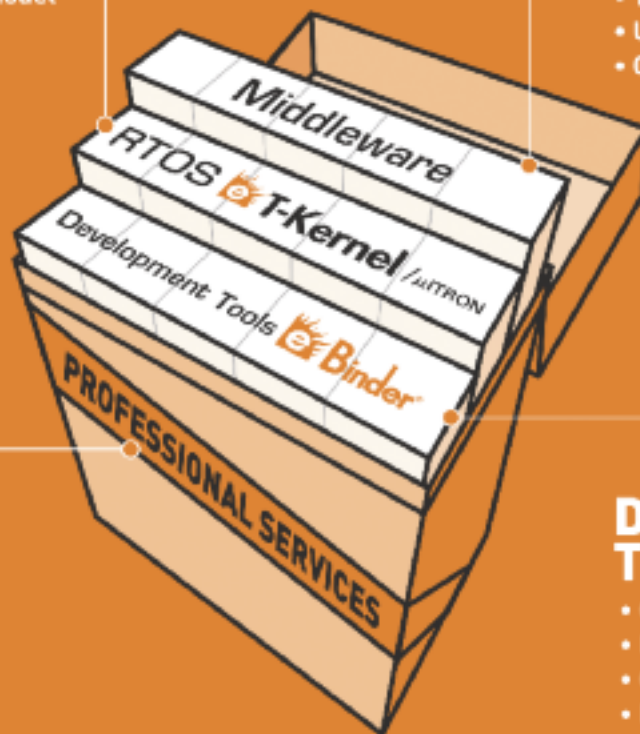
Real-time Operating System

- Scalable profiles
- Process model/Thread model
- POSIX-compliant
- Memory protection
- Multi-core support

Middleware

- File system
- TCP/IP protocol stack
- USB host/device stack
- GUI development tool

ALL-IN-ONE PACKAGE



Professional Services

- Customer support
- Porting
- Customization
- Training

Development Tools

- Configuration tool
- Build tool
- C/C++ Compiler
- Multiprogramming tool
- System analysis tool

Supported Architecture: ARM7, ARM9/9E, ARM11, ARM11 MPCore, Cortex A-8, Cortex-A9, Cortex A-9 MPCore, MIPS, SH

The Proven Solution for Embedded Software Development

LOOKING TO ACCELERATE YOUR COMPETITIVE EDGE? eCROS eliminates the time-consuming need to build software platforms, so you can focus on what really matters: application development. This all-in-one suite combines the RTOS, middleware, and development tools with comprehensive professional services. Our scalable RTOS in particular enables you to reuse software assets for effective product line development. Our Memory Protection and unique Memory Partitioning technology assures system reliability and high quality.

Backed by 30+ years of expertise—and proven track record in car navigation systems, consumer electronics, factory automation devices, and much more—eCROS is the best way to develop high-quality application software and maximize reliability of your system.

web search: or go to www.esol.co.jp



eSOL is a member of the ARM Connected Community



eSOL Co., Ltd. Embedded Products Division Tel: +81 3-5302-1360 E-mail: ep-info@esol.co.jp

Embedded Excellence™