LYNX
SOFTWARE TECHNOLOGIES

How to Choose a Real-Time Operating System (RTOS)

Challenges in Choosing an RTOS

Choosing an RTOS is not as simple as choosing a car. We know about cars; we know their strengths and weaknesses and we intuitively understand compromises like performance vs practicality or luxury vs price. We see all kinds of vehicles on the road, so the range of available cars is obvious. When choosing an RTOS, the middle ground is crowded—dozens of general-purpose RTOSes with broadly similar characteristics compete. They all have a scheduler, services, libraries, middleware, technical support, and graphical tools. Any one of them could genuinely do a good job and so choosing between them is a mixture of quantitative metrics (like features and price) and qualitative measures (like past-experience, personal-preference, and reputation).

Talking to a car dealer, however, means that the conversation will be all about cars. What's more, if you talk to a Ford dealer, he will point you to a Ford; talk to a Fiat dealer and you'll get a different answer. More important, however, than a vendor's bias regarding a brand of car—or RTOS—is the limiting nature of the scope of the discussion. Changing markets have delivered innovative transportation options, but a car dealer will never suggest the latest motorbike, an Uber, or even the bus—yet one of those options may be the best fit for your requirements. The same is true when choosing an RTOS.

Design Your System Architecture First

This article covers several platform options, including commercial RTOSes, Linux, Hypervisors, unikernels, containers, and separation kernels. It goes into some detail regarding the trade-offs of key decisions made by RTOS vendors when building their RTOS and how those decisions affect the choices you have. It goes further to shed light on little known niche RTOSes—RTOSes that are 100 times smaller than usual—and alternatives that provide RTOS-like scheduling and separation with better memory efficiency and wider guest flexibility than an RTOS. These options, however, are dependent on the software architecture of your system. It is important to design your system architecture first—before choosing an RTOS—because an RTOS limits architectural freedom. The smallest RTOSes, as well as the most efficient and flexible alternatives, are not centered around a general-purpose RTOS, but more innovative approaches like single-stack RTOSes and hypervisors. Architecture must be considered first to avoid accidentally eliminating these innovative approaches.

How Big Should an RTOS Be?

RAM components in embedded devices cost money and consume electricity, so minimizing memory use is desirable for high volume and battery powered devices. RTOS memory footprints range in size from as little as 8K up to 2MB — that is, about 8192 to 2,097,152* bytes, or equivalent to approximately 3 to 700 pages of plain printed text. Whether this is small or not depends on your perspective. A modern PC's RAM is thousands of times larger, but the Sengled model E11-G13 smart lightbulb has only 12K of RAM.

A good RTOS should be small and fast, so it's tempting to think smaller is better, and assuming identical configurations, you'd be right. Recompile LynxOS® with a newer gcc version and it will be slightly smaller and slightly faster, i.e., slightly better. But, RTOSes are rarely overweight and so comparing one versus another purely on size is naive.

*These are not round numbers because 1K = 2^10 = 1024 and 1M = 2^20 = 1048576

How Big Should an RTOS Be? (cont'd)

Just as robot-control is different from WiFi-routing, so there are different RTOSes built with features and design compromises to suit each ecosystem. There is no free lunch when coding, every instruction and variable consumes RAM. For example, the printf (print formatted) function from the standard library of the popular C programming language, alone is 8K. For comparison, Zephyr is a small open source RTOS, its entire minimum configuration is 8K. For that you get threading, interrupts and memory allocation, but if you need Bluetooth communication, that doubles the footprint to 16K. This is perfect for tiny Internet of Things (IoT) devices that Zephyr is aimed at. General purpose RTOSes such as Integrity, LynxOS, QNX and VxWorks are larger. By comparison, the default configuration of LynxOS-178® is 1.4MB. For this you get a POSIX® RTOS with thread and process support, floating point, a filesystem, USB, networking, optional bash shell, and of course printf.

In short, how big your RTOS should be depends on your requirements. Expect a general purpose RTOS with lots of features to be about 1.5MB, whereas a minimal specialist RTOS like Zephyr would be around 16KB. Yes, RTOSes can be tiny, but this is not necessarily better; each RTOS is built as small as possible with the features it needs to satisfy its intended purpose. The inclusion—or not—of these features has a far bigger impact on RTOS size than clever optimizations, for e.g.

RTOS Scalability

A highly scalable RTOS maximizes the range of use-cases it is suited to, hopefully allowing you to reuse it on multiple projects, large and small. General purpose RTOSes do a good job of being scalable. They are split into roughly a hundred components and have tools to scale pieces in and out while safely managing dependencies to get down to, typically, about 100K. Scalability can be achieved in different ways. RTOSes from the 1990s tended to guard their source code closely and were delivered as binary-only libraries. This made their scalability relatively coarse, pulling in a whole object file should one function be needed. Improved scalability can be achieved if the RTOS source code is provided, then the compiler and linker can do a finer grain job of including just the components necessary and of optimizing across modules. The Nucleus and ThreadX RTOSes have always included their source code, VxWorks joined them in the 2000s.

Key RTOS Design Choices

Some key design choices have an outsized effect on the size and features of an RTOS. There are pros and cons of each that may be wise or foolish depending on the device being built.

1. Dynamic Memory

If RAM is plentiful the RTOS can manage the excess as a pool (called the heap) and allow applications to allocate and free RAM for their own purposes. This gives flexibility to applications; they can ask for memory as they need it and recycle it after use. But applications can also lose track of memory—causing a memory leak or security vulnerability—and there is the question of what to do if you run out of memory or it becomes fragmented. Safety critical systems in avionics and automotive avoid dynamic memory.

2. Dynamic Linking

If an RTOS is compiled with standard binary interfaces (like DLLs in Windows) and a linker is built into the deployed RTOS, then new functions can be loaded and executed, and then unloaded and replaced—all without rebooting. This is a very efficient development method compared with reloading the entire RTOS every time (over a slow serial connection). VxWorks's popularity took off in the 1990s largely because of this feature. A drawback of this feature is that code and data need to use relocatable 32-bit addresses. This is because modules don't know where in memory they will be loaded, so they need to be able to call-to and access-to data anywhere in the full 32-bit address range. Full range addressing is most flexible and usually wise, but applications less than 16MB in size can fit within the range of the more compact PowerPC relative addressing mode so may prefer to forego dynamic linking in favor of run-time performance.

Key RTOS Design Choices (cont'd)

3. Shell Access
A command line interface to the RTOS is very convenient for developing, configuring and debugging the system on the fly. Without one, every time any RTOS configuration change or application change is made, the entire RTOS or application must be recompiled, redeployed and rebooted. However, a shell requires both a command interpreter, as well as the functions to do what the commands request (display memory etc.), built into the RTOS. A second drawback is that the shell presents a security vulnerability. A famous use of an RTOS shell was when NASA fixed a priority inversion bug on the Mars Pathfinder spacecraft remotely from Earth.

4. Memory Management Unit (MMU) Support
An MMU is hardware that the RTOS can make use of to protect itself from poorly coded applications and to protect applications from over-writing each other. The MMU is a powerful mechanism that allows a protected tasking model—called the process model—to be implemented. This model is used on Windows, Linux and Macs. The problem with processes is that they impose a delay—called a context switch—every time an RTOS system call or process switch is made. Delays run counter to an RTOS's goal to be real-time, so adding MMU support is a compromise of speed vs safety.

RTOS Without a Scheduler

Most RTOSes use a hardware timer to generate a system clock tick that is caught by the RTOS kernel. The kernel manages the list of tasks in the system and based on task priorities, chooses which to run by making a context switch. This approach was suitable for many scenarios, but in the 1990s, with the OSEK standard for automotive RTOSes, another approach with even lower overheads was discovered. If all the tasks and their priorities are known ahead of time then static scheduling is possible—that is, the order of task execution is decided at compile time. The key innovation is that instead of context switching between tasks, simple function calls are used. This eliminates the scheduler entirely and combines every task's stack into a shared stack. This is called a single-stack implementation and is very suited to tiny static embedded systems that are too small to use 1) – 4) above and that want to be even smaller than Zephyr. The RTA-OS from ETAS is an RTOS that uses this single-stack approach.

Choosing a Multi-Core RTOS (Or No RTOS?)

In the embedded world, the principle challenge of multi-core processors (MCPs) is how to efficiently run software on all cores while maintaining the predictable responsiveness that RTOSes are known for. Solving this problem becomes increasingly important as the availability of new single-core processors (SCPs) diminishes. While multi-core aware (SMP) RTOSes are one option, they may not be the best choice. Virtualization and scalability innovations from Enterprise cloud-servers present new design options for embedded projects. Heterogeneous systems running different RTOSes simultaneously, or an efficient mix of "just enough" RTOS and bare-metal to provide maximum flexibility while avoiding vendor lock-in become possible. The question "Which RTOS Do You Need" is more and more becoming, "Do You Need an RTOS?"

Symmetric Multiprocessing (SMP) RTOS

The classic approach to run an application on a multi-core processor (MCP) is to run a Symmetric Multiprocessing (SMP) OS. SMP OSes are aware of all cores and schedule processes across the cores in order to balance the load. SMP OSes are one way to solve the problem of how to efficiently use the computing power of MCPs and have been the common approach used for decades by Windows, Linux and MacOS PCs. SMP adoption has been slower in the embedded market. It is currently supported by many (though not all) RTOSes. Wind River introduced SMP with VxWorks v6.6 in 2007 and Mentor Graphics added SMP support to Nucleus v3 in 2010. Lynx released its SMP support in LynxOS® v7 in 2015, but neither LynxOS-178® nor FreeRTOS support SMP.

Symmetric Multiprocessing (SMP) RTOS (cont'd)

SMP support is not an obvious feature for RTOSes. Many embedded systems are small and simply do not need the extra compute power it offers. SMP itself does not make your RTOS bigger, but it works best with applications that are already large and compute-power-hungry, such as image processing and artificial intelligence (AI). Such large embedded systems have substantial overlap with Linux and—unless determinism is crucial—may be better implemented on Linux. Implementing SMP edges RTOSes away from their traditional embedded market and into tough competition with open source and low-cost Linux. This creates a business problem for RTOS vendors: Should they invest in SMP if a large proportion of their SMP users will switch to Linux anyway?

Linux

Linux is not real-time, although it is an excellent choice for many other embedded systems such as WiFi routers, set-top boxes, smart home devices, etc. The Linux kernel is impressive as the largest open source software project in the history of computing and is immensely versatile, scalable and has support for practically every device driver and protocol you can imagine. Lynx MOSA.ic™ includes Buildroot, a simple, efficient and easy-to-use tool to generate embedded Linux systems through cross-compilation. A Real-Time Operating System (RTOS) is different because it is deterministic. This means it is predictable and can be relied upon to respond within x number of µSecs every single time. Not just 99.999% of the time, but 100% of the time, regardless of extraneous factors such as CPU load, number of users, available free memory, etc.

A common misconception is that RTOSes are fast. They can appear fast, but slow systems can also be real-time as long they are predictable. The thing that makes an RTOS real-time is that it is built to be responsive, so any part of the system can be interrupted enabling it to rapidly respond to an event. Disabling of interrupts and long—must run to completion—processing tasks are avoided. Linux, by comparison, is built for throughput. In the past the entire Linux network stack disabled interrupts and for years Linux had a big kernel lock. The Linux PREEMPT_RT real-time patch adds more pre-emption points to the Linux kernel, making it more responsive. The problem is it is impossible to know for sure that you didn't miss a long code path somewhere, or add a driver lacking pre-emption points that creates a long code path. So, a discussion about Linux and real-time leads to the question; how real-time, and hard vs soft real-time? These questions, however, miss the point, as real-time means predictable 100% of the time—not more of the time.

Hypervisors

Virtualization is an elegant way to unlock the power of an MCP. Under this approach, a hypervisor is used to split the MCP into a number of virtual machines (VMs). This is more flexible than SMP, as it allows consolidation of different stand-alone OSes onto a single MCP. LynxOS-178®, Linux, Windows, and even bare-metal and competing RTOSes can all be run on an MCP to achieve hardware and software consolidation and re-use of legacy applications. Hypervisors can also time slice a core between multiple OSes, allowing multiple OSes to co-exist on one core. If implemented correctly, hypervisors can provide excellent isolation between guests, improving both security and safety. These benefits and others have led to a "land grab" with RTOS vendors who have rushed to implement a hypervisor in order to control the target hardware and virtualize their competitors (rather than be virtualized).

Containers

Running a hypervisor on an MCP gives you the best of all worlds, but the main drawback is that running multiple OSes side-by-side consumes a large amount of RAM. The core problem is duplication—each OS includes its own full set of services and libraries, and despite those components doing equivalent things, re-using them between OSes is not possible for 2 reasons:

1. Hypervisor separation—by deliberate design—prevents it
2. The OSes are not built to be virtualized (so make no allowance for compatibility or re-use with other OSes)

Containers (cont'd)

Containers are one approach to solving this problem and are proving successful in the Enterprise and cloud-server arenas. A container is a lightweight Virtual Machine (VM) that can choose to re-use some of the underlying OSs services and libraries while over-riding others. This is a tremendous win because it eliminates most of the dependency and configuration difficulties when installing complex software. Such software is packaged up into a container so that it takes its dependencies and configuration with it.

Containers started as a Linux technology, but the concept has spread; Microsoft has Windows-Server containers and Wind River has VxWorks Portable Deterministic Containers (VxPOD), the only RTOS container implementation. Containers share the kernel of the underlying OS, so Windows-Server containers can run only Windows applications and VxWorks containers can run only VxWorks applications. This is the main problem with containers for embedded systems, they lock you into one vendor. Both the underlying RTOS providing the container infrastructure and all containers must be the same RTOS. This is not a problem for cloud-servers (since they all run Linux, and want Linux flavored containers), but will restrict your options when building an embedded device.

Unikernels

A unikernel is a special componentized OS linked directly with the application that only provides the OS components the application actually uses. It is a custom build of the OS, that uniquely contains just the OS services that it's one application needs. Since a unikernel hosts just a single application, MMU protection from other applications is unnecessary. Unikernels therefore dispense with the MMU and use a single-address space instead. There are 2 substantial benefits to this approach:

1. Only the OS components used by the application are included, drastically† reducing the size overhead vs a full OS
2. The MMU context switch between user-space and application-space is eliminated, speeding up the application

Unikernels are even smaller and more efficient than containers and are gaining momentum in Enterprise IT because they allow higher VM density on cloud servers. Unikernels are also attractive because with "less-OS", they improve security by reducing the attack surface. MirageOS is an example unikernel implementation.

There is conceptual overlap between source-code RTOSs are unikernels:

- Both link directly with the application and pull-in only OS components used by the application to improve scalability
- Both use a single-address space to avoid the overhead of MMU context switches

In this sense, RTOSes are already unikernels. But what makes unikernels compelling is that they enable RTOS-like size and efficiency for the vast library of full-featured Linux applications running on—defacto-to-standard—Linux servers. It would be interesting to see a POSIX® RTOS implementation of a unikernel. Such a theoretical RTOS would be able to run Linux applications with not only the size advantage of an RTOS, but with RTOS real-time responsiveness too. As of the time of writing (October 2019), no such unikernel POSIX® RTOS exists.

Unikernels are a great fit when used with virtualization because the hypervisor provides isolation between VMs, "replacing" the security lost by dropping MMU protection. RTOSes, particularly MMU-less RTOSes, are a good fit for virtualization for the same reason.

Separation Kernels

LynxSecure®—the foundation of Lynx MOSA.ic™—is a separation kernel as described by John Rushby in his 1981 paper Design and Verification of Secure Systems. The Muen Separation Kernel is another implementation, and the world's first open source separation kernel. LynxSecure® was built in 2005 to separate DoD

† Unikernels can be as small as 4% the size of the equivalent code bases using a traditional OS

Separation Kernels (cont'd)

applications with different security classifications running on the same hardware. Memory, CPU cores, and devices are all securely isolated, so that VMs are strictly restricted to only the hardware resources allocated to them. In use and in marketing material separation kernels appear similar to embedded hypervisors, but the key difference is that a separation kernel does just separation. It contains no device drivers, no user model, no shell access and no dynamic memory. These ancillary tasks are all pushed up into guest software running in the virtual machines. At boot, LynxSecure®'s job is to setup the virtual machines and then get out of the way. Once a LynxSecure® system is running, all that is left of LynxSecure®‡ is a collection of handlers that run when trigged by hardware events.

LynxSecure® is a cleanly separated and distinct piece of software. It contains no part of our RTOS, LynxOS-178®, and can be purchased separately. This is not the case with most embedded hypervisors, that are actually built by adding hypervisor functionality to an existing RTOS. This is a convenient approach because it allows the hypervisor to piggyback on the vendor's existing catalogue of RTOS BSPs, mitigating the cost of a BSP, but results in a less secure and less elegant design. Separation kernels are small, so lend themselves well to security scrutiny, and cost less to certify to aviation safety, industrial safety or automotive safety standards.

Design Choices Determine Architecture

There is much more to building an embedded device than just choosing an RTOS, and much more to choosing an RTOS than just its size. The design choices covered here make a huge impact on the software architecture of your embedded system. There are exciting and innovative options at every level and opportunities to build a better product in your market in whatever dimension you choose—be it size, price, flexibility, features, maintainability, security, safety or performance. There are myriad commercial products as well as open source RTOSes, OSes, hypervisors and separation kernels to choose from. Be sure to understand the complete picture—including design considerations and innovative approaches—before narrowing your scope. A tremendous amount is at stake when choosing your next embedded software platform. No vendor—Lynx included—can offer every option, nor does every combination of technology fit together. The matrix of CPU support, RTOSes and features is sparse and ever-changing and will hamper your ability to find the best technology for your project. Take your time and carefully weigh up all aspects before making a decision.

‡ LynxSecure® can optionally include a scheduler to schedule VMs on CPU cores