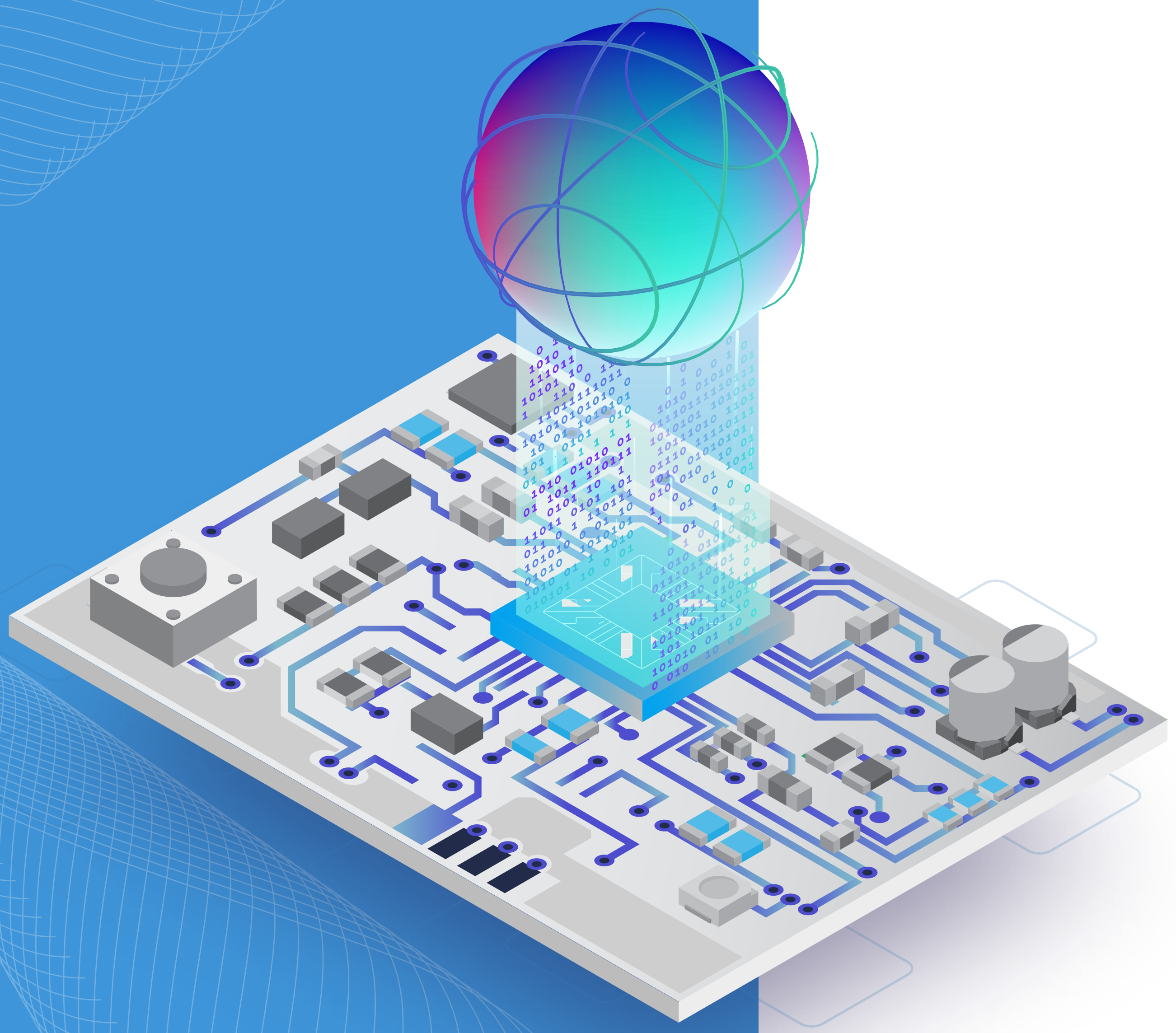
Ultimate Guide to Real-time Operating Systems (RTOS)

01/

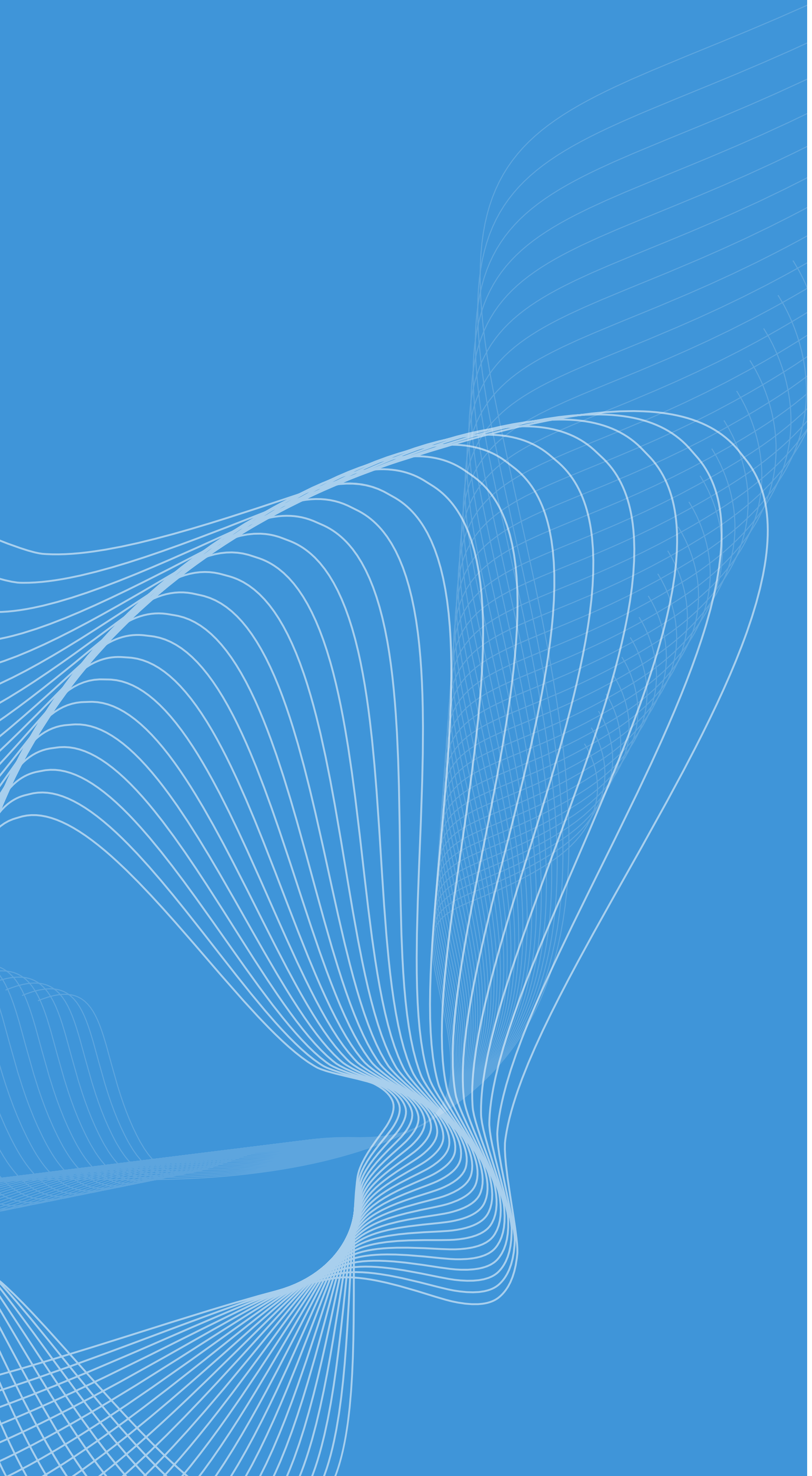
What is an RTOS?

A real-time operating system (RTOS) must be fast and responsive, schedule tasks and manage limited resources, and ensure functions are isolated and free of interference from other functions. In this section, you'll learn more about what is a real-time OS and the pros and cons of two types of RTOS architectures: monolithic and microkernel.



RTOS definition

The main responsibility of an operating system is to manage hardware resources and activities in the system: scheduling application programs, writing files to disk, sending data across a network, and so on. When the OS must handle multiple events concurrently and ensure that the system responds to those events within predictable time limits, it is called a real-time operating system, or RTOS.



Why RTOS for embedded systems?

Many embedded systems require real-time behavior, and due to hardware resource constraints, performance and efficiency are top priorities. An RTOS provides the rigorous resource management and scheduling required to meet the demands of applications– with multitasking, threads, priority-driven preemptive scheduling, and fast context-switching — all essential features of an embedded real-time system.

An RTOS typically has a small footprint and is optimized for performance, however each RTOS must be customized with capabilities needed for the hardware and system it supports. From a bare-bones kernel configuration managing a small number of tasks to a full-functionality RTOS managing hundreds of tasks and subsystems including graphics, networking, filesystem, audio and more – an RTOS should flexibly scale to address system requirements and resources.

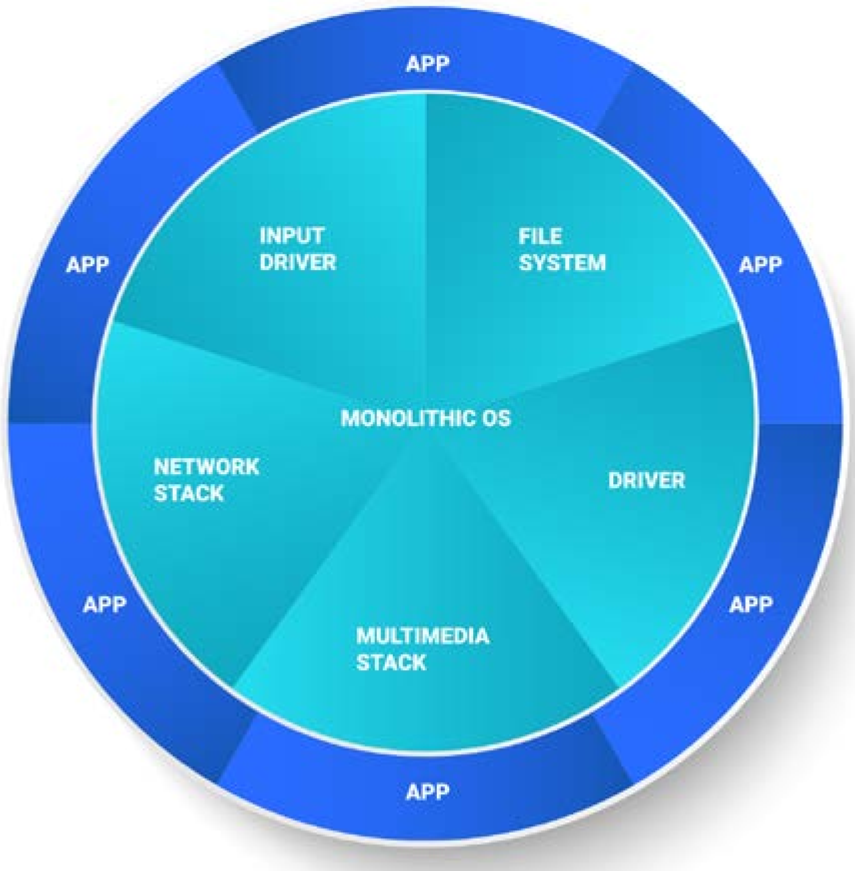
RTOS architectures

RTOS architecture affects the reliability of an embedded system and its ability to recover from faults. There are two RTOS architectures: monolithic and microkernel.

What is a monolithic RTOS?

Monolithic means one huge stone. By definition, a monolithic kernel runs all operating system components in the kernel space. For instance, a monolithic RTOS includes device drivers, file management, networking, and graphics stack as part of the kernel space. Applications, however, run in the user space. Although running user applications as memory-protected processes protects a monolithic kernel from errant user code, a single programming error in a file system, protocol stack or driver can crash the system. In addition, any change to a driver or system file requires OS modification and recompiling.

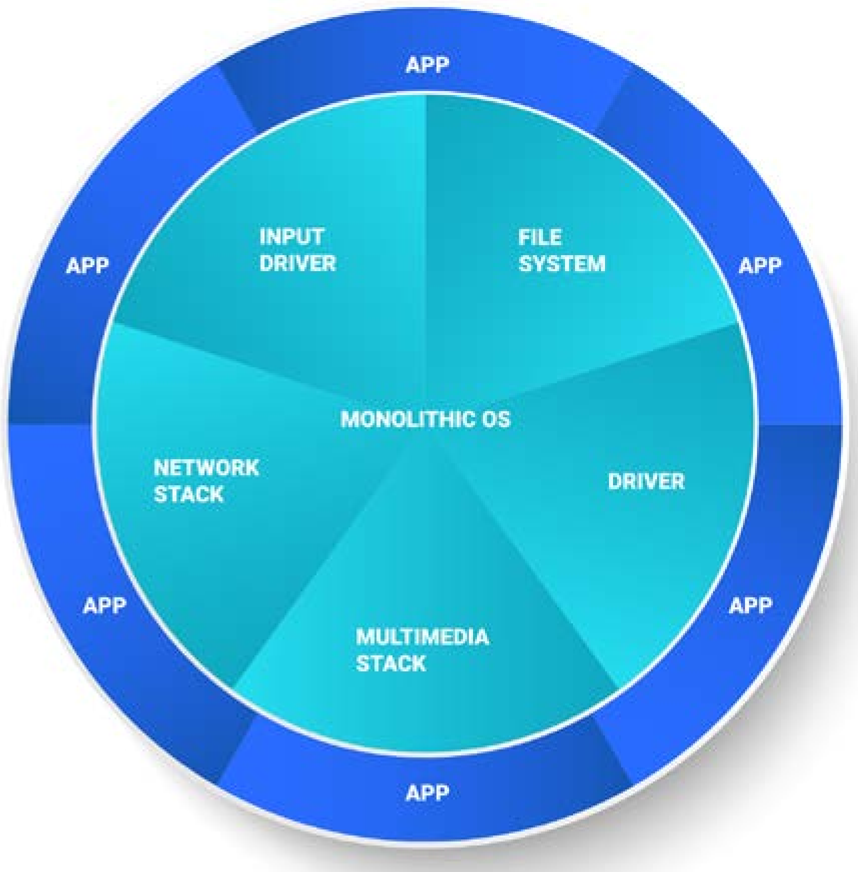
In a monolithic OS, a single programming error in a file system, protocol stack or driver can crash the whole system.



Monolithic OS architecture diagram

Advantages and disadvantages of a monolithic RTOS (e.g. RTLinux)

Monolithic RTOS	
Advantages	**Disadvantages**
Process scheduling, memory management and file management all run as a single large process in the same address space, which can improve performance. The entire OS is contained in a single static binary file, which may run faster and more reliably than dynamically linked libraries.	The failure of any service can crash the OS. Adding or removing a service requires modifying and recompiling the OS. The OS's kernel services represent a large attack surface - if a service is compromised it can make the whole system vulnerable. Larger footprint. Difficult to debug and maintain.



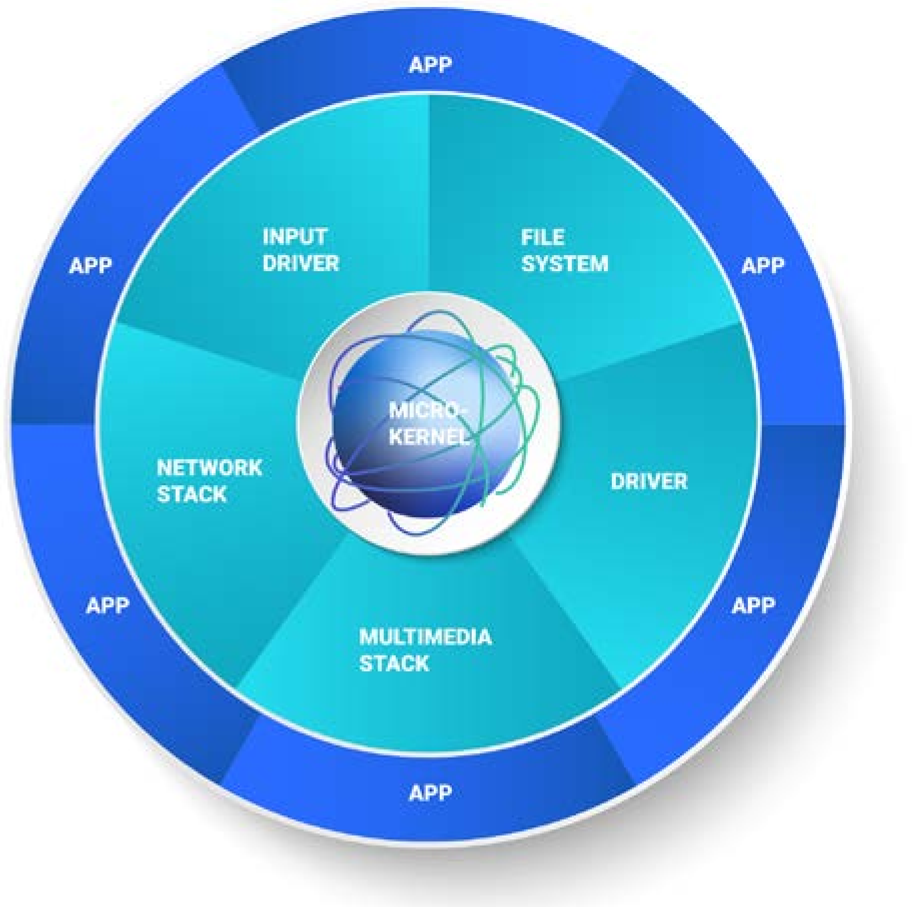
Monolithic OS architecture diagram

What is a microkernel RTOS?

A microkernel RTOS is structured with a tiny kernel that provides minimal services. The microkernel works with a team of optional cooperating processes that run outside kernel space (in the user space), which provides higher-level OS functionality. The microkernel itself lacks file systems and many other services normally expected of an OS. A microkernel RTOS embodies a fundamental innovation in the approach to delivering OS functionality: modularity is the key, and the small size is a side effect.

In a microkernel, only the core RTOS kernel is granted access to the entire system, which improves reliability and security. The microkernel protects and allocates memory for other processes and provides task switching. All other components, including drivers and system-level components, are each contained within their own isolated process space.

Isolation prevents errors in a component from affecting other parts of the system – the only thing that a component can crash is itself. Such crashes can be easily detected, and the faulty component can be restarted hot – while the system is still running – so quickly that the restart has no effect on performance.



Microkernel OS architecture diagram

In a microkernel RTOS, isolation prevents errors in a component from affecting the rest of the system – the only thing a component can crash is itself.

During code development, the isolation of all processes has two significant benefits:

01

Bugs are found earlier in development and are easily traced to a line of code in the faulty process.

In comparison, latent bugs can remain from driver development for a monolithic OS, even after deployment, because stray pointers or other bugs do not cause an easily identified process crash.

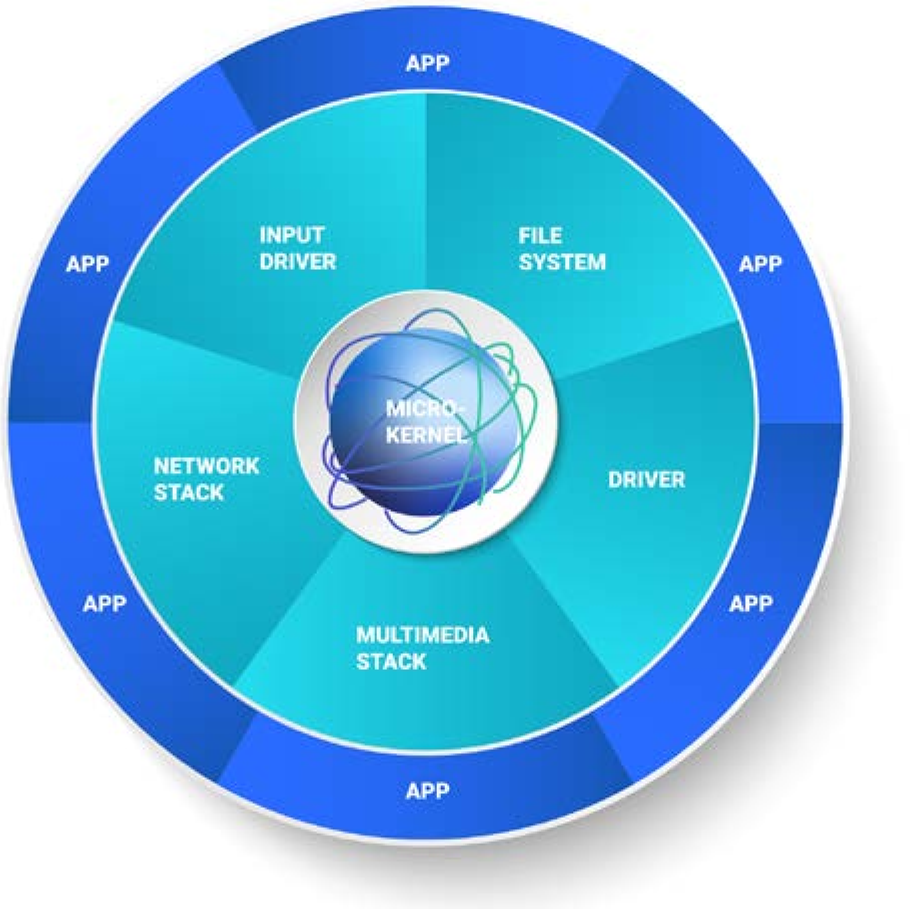
02

Drivers are treated like application processes, making them far easier to write and debug.

You don't need to be a device driver specialist or a kernel debugger to write a driver for a microkernel.

Advantages and disadvantages of a microkernel RTOS (e.g. QNX® Neutrino® RTOS)

Monolithic RTOS	
Advantages	**Disadvantages**
Fault isolation and recovery for high availability. Restart a failed system service dynamically without impacts to the kernel (no system reboot). Easy expansion – develop device drivers and OS extensions without a kernel guru and without recompiling. Easier to debug. Small footprint. Less code running in kernel space reduces attack surface and increases security.	Requires more context switching, which can increase overhead.

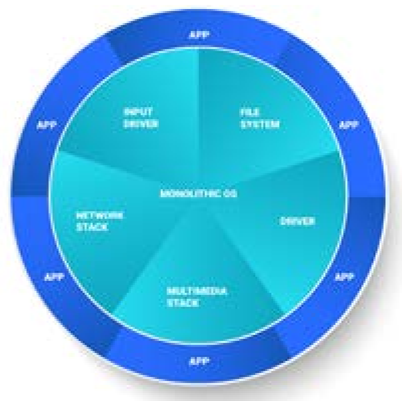


Microkernel OS architecture diagram

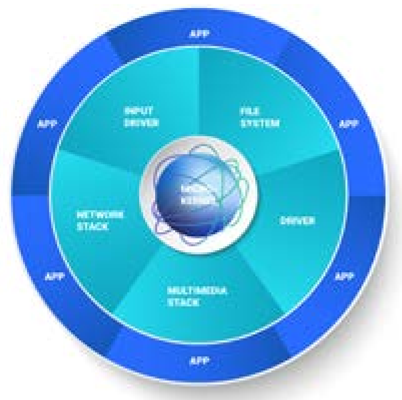
Monolithic kernel vs microkernel, a comparison

Three big differences stand out when comparing a monolithic kernel versus a microkernel OS architecture:

Category	Microkernel	Monolithic
Performance	Slightly slower performance due to higher number of context switches.	Better performance due to smaller number of context switches.
System Updates	New drivers and OS services updates can be performed with no changes to the kernel and thus do not require an OS reboot.	New drivers and OS services updates will require an OS rebuild and reboot.
Power Fail Recovery	Can restart any service individually, without interrupting the kernel.	Recovery of a failed service requires an OS reboot.
Qualification and Certification	Easier and less costly to qualify and certify the kernel. Most system updates do not require a full qualification and certification cycle but rather are limited to the updated service or driver.	Difficult and more costly to qualify and certify. System updates require a full qualification and certification cycle as the OS is generally rebuilt.
Maintenance	Easier and less time consuming to maintain and troubleshoot deployed systems. Users can update, troubleshoot and reboot a service without requiring an entire OS reboot.	Challenging and more time consuming to maintain and troubleshoot. Users require an OS reboot when performing most system updates, troubleshooting steps or a service reboot.



VS



QNX Neutrino RTOS is a commercial microkernel RTOS

Since 1980, thousands of companies have deployed and trusted QNX real-time technology to ensure the best combination of performance, security and reliability in the world's most mission-critical systems. At the core of this offering is **QNX Neutrino Realtime Operating System (RTOS)**, a full-featured and robust RTOS designed to enable the next-generation of products for automotive, medical, transportation, military and industrial embedded systems.

The microkernel design and modular architecture enable BlackBerry® QNX® customers to create highly optimized and reliable systems with low total cost of ownership. With the QNX Neutrino RTOS, embedded systems designers can create compelling, safe, and secure devices built on a highly reliable RTOS serving as the foundation that helps guard against system malfunctions, malware and security breaches.

What to expect in an RTOS

02/

An RTOS delivers functionality that helps embedded systems developers deliver safe, secure and reliable products. In this section, you'll learn about key RTOS functionality to look for or to consider building into your own RTOS – such as spatial and temporal separation, adaptive partitioning, preemptive priority-based scheduling, and system determinism.

Spatial separation

Spatial or hardware separation, also called spatial isolation, provides each process with its own private address space. Embedded systems require the isolation of software components to ensure freedom from interference in hardware (spatial) and time (temporal). A memory management unit (MMU) provides spatial separation by mapping physical memory to virtual memory and protects parts of the physical address space from unwanted access.

An MMU allows an RTOS to use a process model – each task (process) is allocated its own virtual address space – for much greater reliability. In comparison, without an MMU, code, data and the kernel itself would compete for and share the same memory space – a less reliable approach.

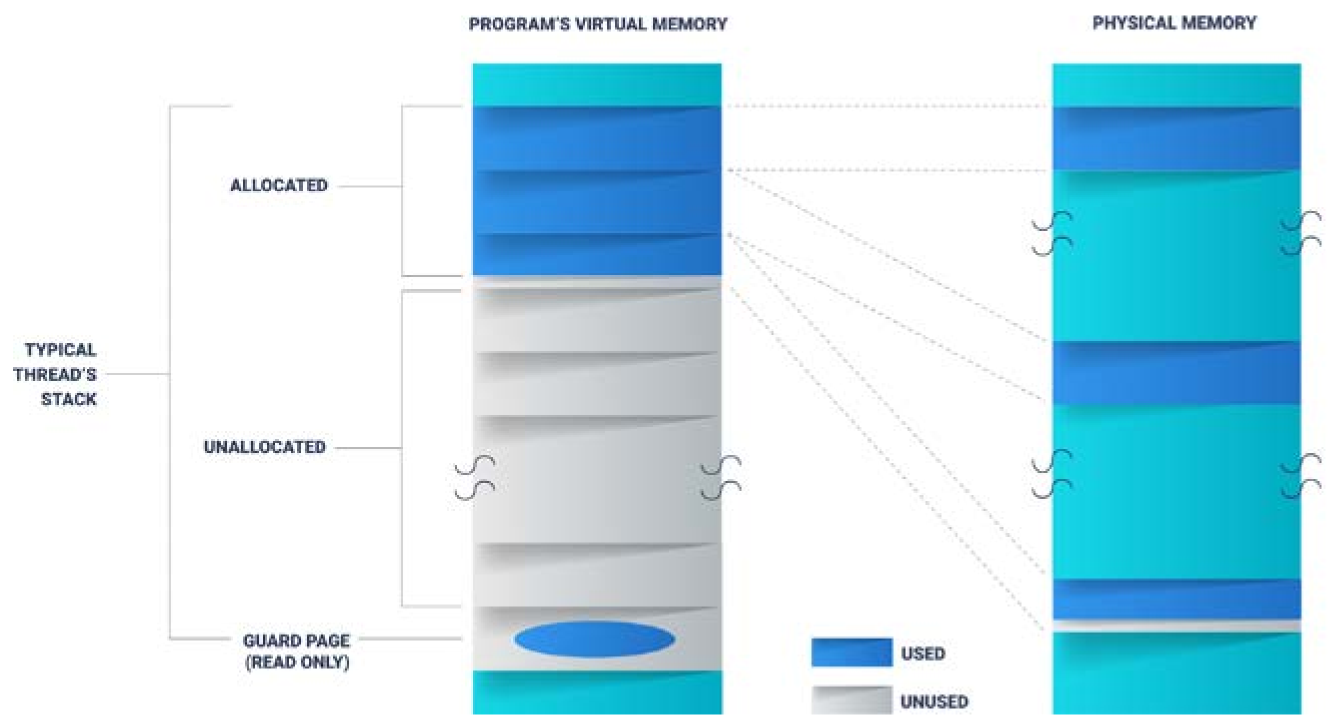


Figure 1:

The QNX OS provides spatial separation through the use of the memory management unit (MMU). The operating system memory management unit maps the user program's virtual memory address to a physical memory address, providing full memory protection.

Temporal separation

Temporal separation, also called temporal isolation, allows each process to run without depending on the timing of another unrelated system sharing the same hardware or software resources. RTOS scheduling provides temporal separation by ensuring process threads run when they are supposed to, and there are always enough CPU computing cycles to go around.

By partitioning resources, scheduling algorithms deliver temporal separation between tasks with different levels of criticality – ensuring the higher priority task gets the resources it needs. Tasks generally include both periodic (regular) tasks and aperiodic (irregular) tasks. The most common ways to provide temporal separation are static and adaptive partitioning.

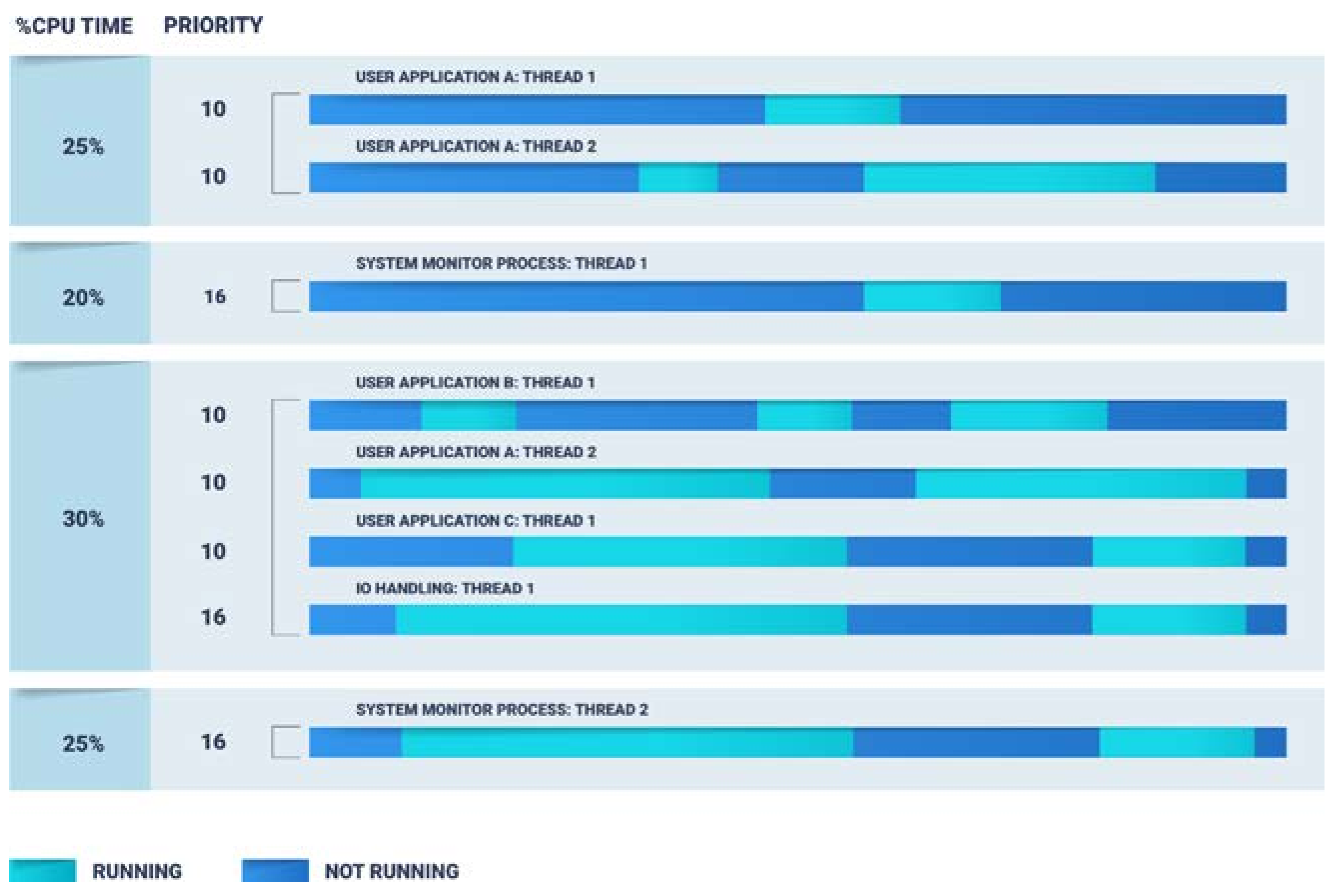


Figure 2:

In the QNX RTOS you can assign a priority to each thread and use adaptive partitioning to guarantee CPU availability for critical threads, ensuring they can run as soon as they are ready.

Interprocess communication (IPC)

Interprocess communication adds an additional layer of isolation between address spaces. Developers need to pass data between processes or tasks. In the QNX RTOS, interprocess communication maps POSIX calls to messages, in addition to the hardware separation afforded by the MMU.

A message-based approach to interprocess communication provides a high level of architectural decoupling as required by safety standards such as ISO 26262. Although message passing is the primary form of interprocess communication in the QNX OS, other forms of IPC are also available. Message passing is part of the POSIX standard and all BlackBerry QNX solutions are POSIX-compliant.

Static versus adaptive partitioning

Partitioning prevents processes (or threads) from monopolizing CPU cycles needed by others. You don't want one application – whether defective or malicious – to corrupt another application or prevent it from running. To address this issue, some systems use virtual walls, called partitions, around a set of applications to ensure that each partition is given an engineered set of resources. The primary resource considered is CPU time but can also include any shared resource, such as memory or file space.

An RTOS can enforce CPU partition budgets via static or adaptive partitioning, or another partitioning scheme.

- **Static partitioning** divides tasks into groups (partitions) and allocates a percentage of CPU time to each partition. No task can consume more than the amount allocated to its partition. When partitions don't need their full allocation, CPU cycles are left unused and interrupts have to wait until the partition runs, which can cause unacceptable latency.
- **Adaptive partitioning** provides a dynamic scheduling algorithm that allows the system designer to reserve CPU cycles for a process or group of processes and to dynamically reassign CPU cycles from partitions of lower need to partitions with higher need. The result is a faster, more efficient and responsive system that guarantees time for important tasks, with minimal unused CPU cycles.

The QNX Neutrino RTOS uses adaptive partitioning. A system designer can launch POSIX-based applications in partitions, and the RTOS scheduler will ensure that each partition receives its allocated budget. Within each partition, each task is scheduled according to the rules of priority-based preemptive scheduling.

Adaptive partitioning technology guarantees minimum budgets to defined groups of threads without wasting unused processing time

Priority-based preemptive scheduling

Priority-based preemptive scheduling allows high-priority threads to meet their deadlines consistently, even when there is a lot of competition for resources. With priority-based preemptive scheduling, a high-priority thread can, within a small and bounded time interval, take over the CPU from any lower-priority thread. The high-priority thread can run uninterrupted until it has finished – unless it is preempted by an even higher-priority thread.

System determinism

Deterministic real-time scheduling ensures that the most urgent software runs when it needs to. Preemptive priority-based multitasking is deterministic. The scheduler uses priorities to determine which task should run next. Unexpected systems loads, including third-party code, will not adversely affect safe operation.

A deterministic RTOS ensures that priority threads get the time they need, when they need it, by preempting a lower-priority task. For example, in a car crash, the airbags must deploy immediately, not wait for another task to finish.

A deterministic RTOS ensures that priority threads get the time they need, when they need it, by preempting a lower-priority task.

Responsiveness

Embedded systems with hard real-time constraints require responsiveness. Real-time applications depend on the OS to handle multiple events and to ensure that the system reacts within an expected timeframe to those events. In other words, the system's response time must be predictable.

The QNX RTOS is ideal for mission-critical systems that require responsiveness and absolute reliability.

03/

How to choose a commercial RTOS

A commercial RTOS can save engineering time and effort and improve the reliability and performance of your embedded systems. While most real-time operating systems deliver high performance, other aspects of a commercial RTOS and related tools and services can affect your product quality and engineering effort. In this section, you'll learn about what else you may need to deliver the features, security and safety your customers want, and your development teams need.

When selecting a commercial RTOS, consider the following:

Safety:

Will you need to certify your embedded system to IEC 61508 (industrial), IEC 61511, EN 50128 (rail), IEC 62304 (medical) or ISO 26262 (automotive) or another industry standard? The choice of a pre-certified RTOS could help improve the system reliability and reduce your safety certification effort.

Security:

A microkernel architecture, adaptive partitioning and a hypervisor can all help protect safety-critical processes from attack. In addition, some commercial real-time operating systems, like the one from BlackBerry QNX, include a security policy. This enables system architects and integrators to control access to specific system resources and determine the type of access that is permitted (e.g. no root access). Security is achieved with a layered approach that includes mechanisms such as secure boot, integrity measurement, sandboxing, access controls and rootless execution.

Development environment:

A POSIX-compliant RTOS will simplify migration from a Linux-based prototype to a more reliable, secure and safe production system. Developers ramp up quickly on the **QNX® Software Development Platform (SDP)**, because it looks and feels like Linux and uses the same tools, such as the GNU Compiler Collection (gcc).

A pre-certified RTOS can simplify your safety-certification effort.

When selecting a commercial RTOS, consider the following:

Graphics and human machine interface (HMI):

Does your embedded system drive one or more displays with HMIs? Choosing an RTOS with a graphics subsystem that provides all the functionality necessary to develop interactive user interfaces is key. A graphics framework that supports industry standards such as OpenGL ES means developers can more readily build graphics user interfaces and benefit from the hardware acceleration provided by graphics processing units (GPUs).

Maintenance and updates:

Keeping your product up-to-date over its lifetime may require the ability to apply patches or easily add functionality. In a microkernel-based OS, a new service can be added to the user address space, without any kernel changes, whereas a monolithic OS requires the entire kernel to be modified.

Hardware support:

An RTOS must be customized for each processor or board, so look for an RTOS that offers board support packages (BSPs) for your preferred hardware to jumpstart your development. In addition, an RTOS with an extensive list of BSPs indicates it is widely used in multiple embedded markets. For example, BlackBerry QNX provides **board support packages** for a long list of hardware, for x86 and ARM processors.

An RTOS that supports your preferred boards with board support packages can save a lot of development time.

When selecting a commercial RTOS, consider the following:

Licensing:

Will you pay before, during or after your product is developed? An open source OS comes with hidden costs – there can be considerable engineering effort required to keep up with OS maintenance, patches and modifications to the kernel. Commercial RTOS vendors offer a variety of licensing options.

Vendor reputation and quality of support:

Look for a software vendor with a positive reputation that provides easy access to quality documentation and excellent customer support. In addition, you may value a services team that helps develop and secure your product, navigate safety certification, and help ensure you hit your start of production dates.

Total cost of ownership:

A commercial RTOS can provide lower total cost of ownership than an open source OS, such as Linux. BlackBerry QNX provides ongoing maintenance and support, allowing our customers to free up engineers for product innovation and differentiation, instead of kernel code changes.

How can BlackBerry QNX help you?

04/

BlackBerry QNX is trusted across multiple industries to provide the software foundation for safe, secure and reliable systems that get to market faster. In this section, you'll learn about our other tools and services – including a hypervisor, middleware, engineering services and supplementary solutions – plus our heritage and deep expertise in embedded system software.

More than an RTOS

BlackBerry QNX offers a time-tested and field-proven RTOS and so much more. With a hypervisor, middleware, engineering services and supplementary solutions, we have everything you need to build a safe, reliable and secure embedded system.

Hypervisor:

Will you want guest OS support to provide design flexibility and software reuse in a complex system? As system-on-chip (SoC) vendors add more computing cores with more processing power, developers gain long-term advantages with a hypervisor for virtualization, such as QNX® Hypervisor.

Additional frameworks and platforms:

Will your project require advanced driver assistance systems (ADAS), connectivity, or over-the-air programming (OTA)? Starting with a framework or platform can be a big help to deliver features such as sensor data management, Wi-Fi connectivity, streaming media, speech recognition, infotainment systems, and ADAS.

Engineering services:

Will you need help? BlackBerry QNX engineering services, safety services and security services extend your team to help you shorten development timelines with expertise in embedded software development.

More than an RTOS

Language support and development tools:

What tools are your developers already familiar with? The QNX Software Development Platform supports C/C++, HTML5, Qt, Python and more. QNX Neutrino RTOS looks and feels like UNIX. QNX is POSIX-compliant, so developers can port code easily from Linux and other operating systems to QNX OS.

Training:

Will your team need training? BlackBerry QNX offers hands-on, instructor-led **training courses** using real-world examples to give your development team a foundation in QNX best practices – and the features, services, and architecture of the QNX Neutrino RTOS.

Board support packages (BSPs):

A BSP, or board support package, is software responsible for the hardware-specific operations required to get an RTOS up and running. A **QNX board support package** typically includes initial program load (IPL), startup, default buildfile, networking support and board-specific device drivers, system managers and utilities.

QNX training opportunities

Want to learn more about QNX training?

LEARN MORE

Contact us

Our team of experts are here to answer your questions.

CONTACT US

Our reputation

BlackBerry® has decades of experience in powering mission-critical embedded systems in automotive and other industries. We are proud to share our expertise with you.

Contact us

Our team of experts are here to answer your questions.

CONTACT US

On time, every time:

Having worked on hundreds of automotive programs, BlackBerry QNX has remarkably never missed a start of production deadline. That means there have been no software delays, no issues delivering new products, and no task too complex to have affected the start of production.

In more than 150 million automobiles on the road:

BlackBerry QNX software is embedded in more than 150M cars on the road today and growing.

Our products and expertise in safety certifications:

Having certified our own products to the highest level of safety: ISO 26262 (ASIL D) and IEC 61508 (SIL 3), we have helped our customers achieve ISO 26262 and IEC 61508 certification and IEC 62304 compliance with a 100% success rate.

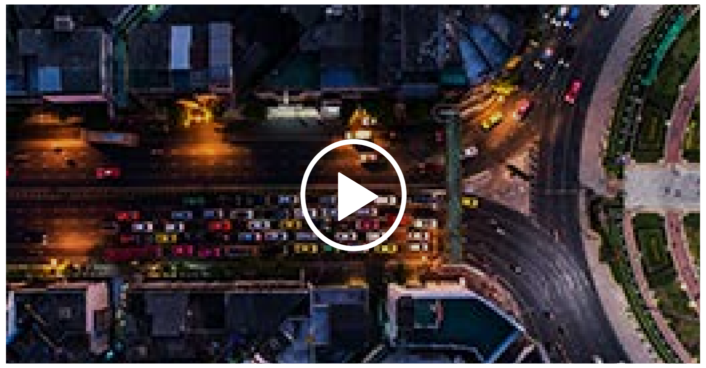
Our heritage in security:

BlackBerry offers the world's most trusted mobile security – hardened, tested, trusted and certified – backed by decades of experience. Similarly, BlackBerry QNX RTOS offers the gold standard in RTOS security with the expertise and safety certifications to help our customers build more secure products.

Resource Center

Videos

Connected and Autonomous Vehicles



Webinar 5 Ways Virtualization Keeps Your Embedded Tech Competitive

QNX Products

QNX Neutrino RTOS

If it's mission-critical, it runs on the QNX OS

LEARN MORE

QNX Hypervisor

Real-time type 1 microkernel hypervisor

LEARN MORE

QNX OS for Safety

A reliable foundation for building safety-critical systems

LEARN MORE

Contact us

Our team of experts are here to answer your questions.

CONTACT US

About BlackBerry® QNX®

BlackBerry QNX is a leading supplier of safe, secure, and trusted operating systems, middleware, development tools, and engineering services for mission-critical embedded systems. BlackBerry QNX helps customers develop and deliver complex and connected next generation systems on time. Their technology is trusted in over 150 million vehicles and more than a hundred million embedded systems in medical, industrial automation, energy, and defense and aerospace markets. Founded in 1980, BlackBerry QNX is headquartered in Ottawa, Canada and was acquired by BlackBerry in 2010.