



**QNX SOFTWARE SYSTEMS**

## **Lowering the Development Costs of Industrial Control Systems through Software Partitioning**

**By providing CPU guarantees for each software subsystem, partitioning can dramatically reduce software integration efforts.**

*Kerry Johnson, Senior Product Manager, QNX Software Systems*

## Growing Complexity

Not long ago, most industrial control systems had modest software requirements — typically, a few thousand source lines of code. Today, however, an embedded control system may contain hundreds of thousands of source lines and employ dozens of software tasks, all of them contending for a limited amount of memory and CPU time.

To speed development of these complex systems, companies often divide the work among multiple development teams, each responsible for developing a separate software subsystem. Given the parallel development paths, performance issues invariably arise at the integration phase, when, for the first time, the various subsystems begin competing with one other for system resources. Subsystems that worked well in isolation now respond slowly, if at all. Unfortunately, many of these issues emerge only during integration and verification testing, when the cost of software redesign and recoding is at its highest.

Diagnosing and solving such problems is intrinsically difficult. Designers must juggle task priorities, possibly change thread behavior across the system, and then retest and refine their modifications. The entire process can easily take several calendar weeks, resulting in increased costs and delayed product.

## Partitioning as a Solution

Recently, the concept of partitioning has gained mindshare as a way to manage these complex integration issues. Briefly stated, this approach allows design teams to compartmentalize software into separate partitions and allocate a guaranteed portion of system resources to each partition. As a result, each partition provides a stable, known runtime environment that development teams can build and verify individually. If the software processes within a partition perform well during unit testing, they will, with a high degree of confidence, exhibit the same performance at integration time. Unforeseen resource contention among subsystems is largely eliminated.

While dividing a static resource such as memory into partitions is straightforward, the partitioning of CPU processing time is more difficult. Since cost and power consumption are common constraints for embedded control systems, the processors selected for such systems typically have little spare processing capacity. Given this environment, CPU time is a scarce resource that must be managed carefully, especially during times of heavy CPU loading. CPU time partitioning provides a way to achieve this goal.

### The RTOS scheduler

To understand the need for CPU time partitioning, we must first consider the role of scheduling in a realtime embedded system.

To ensure code portability, most modern operating systems support the POSIX application programming interface (API). This open standard defines a thread model in which a single process can contain one or more executable threads. To achieve the concurrency and realtime behavior required for embedded products, POSIX-based realtime operating systems (RTOSs) employ a preemptive, priority-based thread scheduler. This form of scheduler always gives CPU time to the highest-priority thread that has

work to do. If a high-priority thread becomes ready (normally triggered by an external event), it will preempt any lower-priority thread currently running.

A priority-based scheduler offers several benefits, including:

- Predictable latency — By allocating time-critical functions to high-priority threads, developers can precisely control how long the system takes to respond to external events.
- Concurrency and flexibility — By using priority scheduling, embedded systems can handle a mix of tasks, including regularly occurring tasks with hard deadlines, high-priority event-driven tasks, and background processing tasks.
- Proven and familiar — Priority-based scheduling is widely used in industrial applications and is well-understood by embedded developers.

When a thread becomes ready to run, it enters a queue of ready threads with the same priority. The scheduling policy, set per thread by the developer, determines which of these queued ready threads will run next. POSIX defines three scheduling policies:

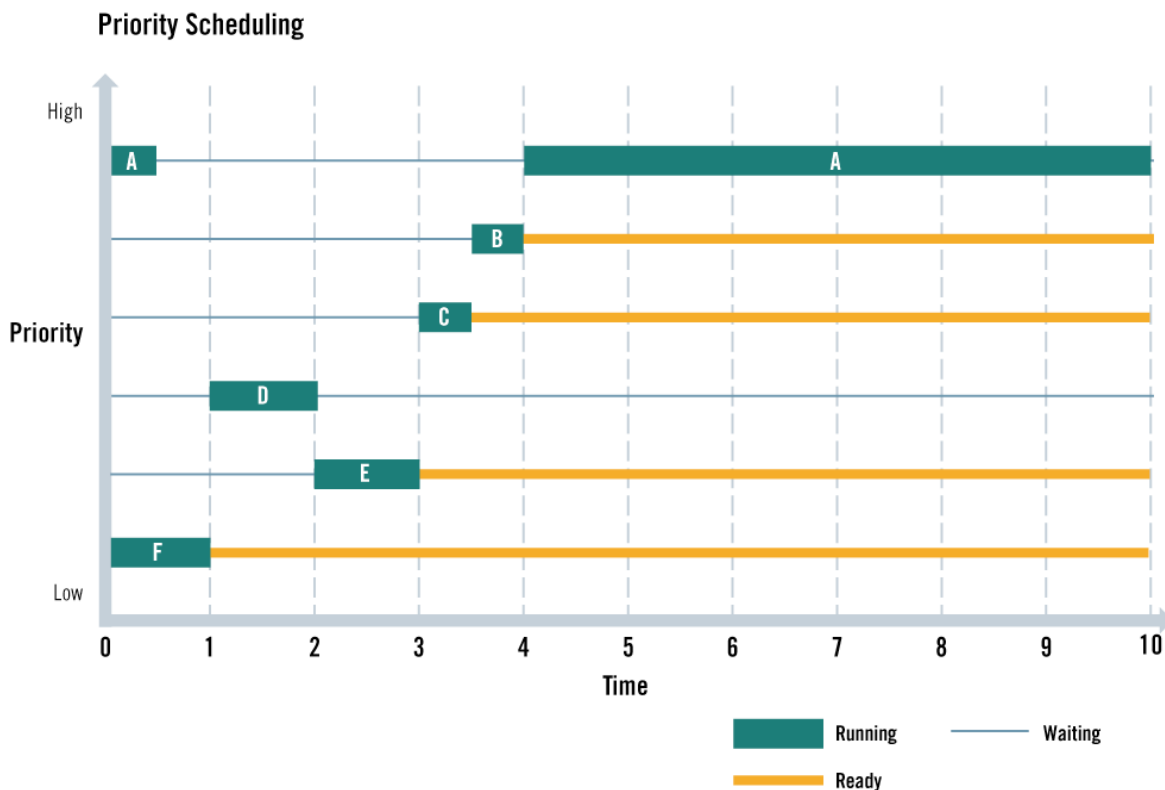
- First in, first out (FIFO) — Guarantees that a thread selected to run will continue executing until the thread voluntarily blocks or is preempted by a higher priority thread.
- Round robin — Guarantees that a thread will execute until it is preempted, voluntarily blocks, or consumes its scheduling time slice.
- Sporadic — Provides a limit on how long a thread will execute within a given period of time.

In sporadic scheduling, a thread has two assigned priorities, normal and low, and runs at its normal priority for a specified period of time. Once that time interval is consumed, the thread runs at its low priority. The normal priority processing time is refreshed regularly. This approach helps prevent a single thread from running at a high priority for too long. Nonetheless, designers often limit the use of sporadic scheduling since it is difficult to use in a system with many threads.

## Managing Priorities

A priority-based scheduler isn't a "fair share" scheduler. The highest-priority task with work to do can monopolize the CPU and starve all other tasks of CPU time. Consequently, developers must carefully assign and test task priorities across the entire system.

As systems become more complex and design teams grow, assigning and maintaining priorities for a large number of threads becomes increasingly difficult — and a significant point of contention among developers. Realizing that unconstrained priority assignment yields to chaos (or a non-operational system), system designers often choose to limit the number of priorities in use to a handful. However,



**Figure 1** — With priority scheduling, a high-priority task can inadvertently or maliciously consume CPU cycles needed by other tasks. For instance, Task A prevents all other tasks from accessing the CPU after T4.

this solution has an undesirable side effect: increased latency. Since many threads have the same priority, the ready queue of threads at a given priority can become very long. A ready thread must consequently wait until it reaches the head of a long queue before it can run.

As a further problem, a priority-based scheduler can allow downloaded malware or a denial of service (DoS) attack to monopolize the CPU, thereby making the system unavailable to users. Security organizations such as the SANS (SysAdmin, Audit, Network, Security) Institute see SCADA systems as particularly vulnerable to such network-based exploits.

To address these problems, systems designers and software engineers can choose from two general approaches:

- A supervisor process, or watchdog, that monitors how threads consume the CPU
- OS-controlled time partitions that allocate CPU time to specific threads

## Watchdog Processes

Sometimes, a group of related threads will starve other threads of CPU time. In other cases, starvation results from a single “runaway” thread that runs in a high-priority loop. To prevent either scenario, a watchdog process tracks CPU utilization and invokes corrective actions whenever it determines that a thread has exceeded its predetermined CPU “budget.”

Nonetheless, there is no simple way to throttle back the CPU usage of a runaway or hog thread. Consequently, the watchdog must employ heavy-handed methods, such as restarting the offending process (and all of its child threads) or lowering the priorities of the offending threads or processes.

Moreover, the watchdog approach doesn’t work well with all processes and threads. For instance, some processes will likely have CPU consumption peaks at certain times. The watchdog must therefore consult a list of exceptions to make decisions about the health of the system.

A watchdog can also consume a significant amount of processor time. For instance, it must query the OS at regular intervals to determine the usage of all threads and then compare that data with information from the previous polling interval. It must then account for all of the exception cases mentioned above. Since the watchdog runs at a higher priority than the threads it monitors, it itself can cause task starvation!

The watchdog approach has other drawbacks, including slow response times and a cap on CPU usage when legitimate processing is required. This cap prevents the system from handling peak demands.

## OS-Controlled Time Partitioning

Some RTOSs support CPU time partitioning. This concept provides a container for processes and threads, called a partition, that can be allocated a guaranteed share of CPU time. For example, a system designer can place a set of threads that have a common purpose (such as handling user interface input and output) into a partition and allocate the partition a processing budget of 5% of the entire CPU capacity. The partitioning scheduler will then guarantee that the partition receives its allocated CPU budget.

Using a time-partitioning approach, the high-level design can specify a CPU budget for each major software subsystem. The system designer can thus guarantee that the load on one subsystem won’t affect the availability of other subsystems. This approach prevents any single thread from consuming the entire CPU, even if the thread is running at the highest priority level.

Within a partition, threads are scheduled according to the traditional rules of a preemptive, priority-based scheduler. Scheduling policies such as FIFO, round robin, and sporadic all operate within a partition. In effect, each partition becomes a “mini” execution environment.

Partitioning schedulers vary. Some implementations strictly enforce budgets at all times, so that each partition will consume its full budget even when it has no work to do. Other implementations can dynamically allocate these unused CPU cycles to other partitions, thereby maximizing overall CPU utilization and allowing the system to handle peak demands.

## **Simplifying Parallel Development and Integration Testing**

Time partitioning facilitates parallel development by allowing the system designer to define a CPU budget for each subsystem. This approach eliminates the need to implement and enforce global priority schemes, and allows developers to define subsystem-level priority schemes according each subsystem's need. As a result, the design teams can proceed in parallel with their design and implementation.

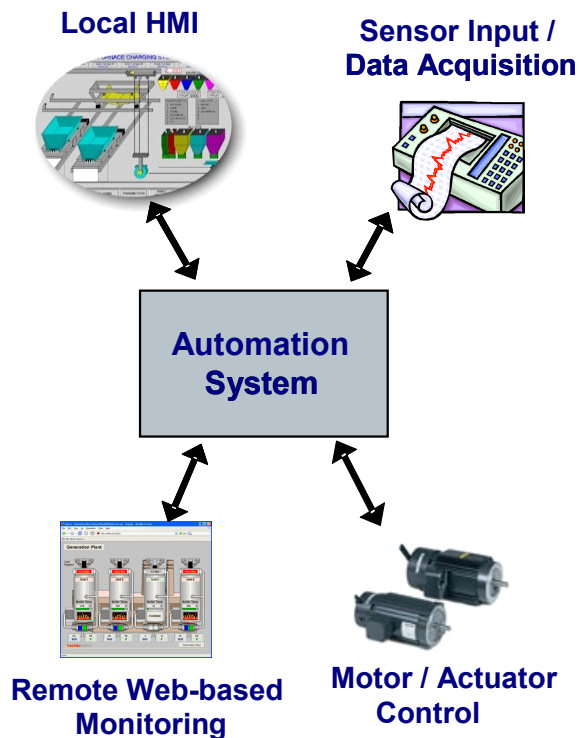
When subsequently testing processes within a partition, developers can simply create a CPU load outside of that partition to simulate a heavily loaded CPU. Developers can then test the operation and performance of their code under a simulated worse-case load condition. As a result, developers can resolve a number of performance and CPU consumption issues prior to system integration.

To appreciate these benefits, consider a relatively simple system designed without the use of time partitioning. The system, illustrated in Figure 2, contains the following processes:

- A medium-priority process that handles the local human machine interface (HMI)
- A medium-priority process that performs periodic sensor scanning
- A high-priority process for motor control
- A low-priority remote monitoring agent process that sends updates to a central, web-based monitoring system

During the integration phase, when the entire system is assembled, the web monitoring system works fine until someone uses the local HMI. At that point, the monitoring system often appears to freeze and ceases to display any updates. Troubleshooting reveals that when the HMI issues commands that result in a high level of motor control, the remote monitoring agent doesn't receive any CPU time. An assessment of the priorities illustrates why this occurs. Since the remote monitoring agent is the lowest priority, it becomes starved when the system operates at full CPU load.

To solve the problem, the system designer assigns the local HMI a lower priority than the remote monitoring agent. However, this approach leads to an unacceptable level of performance for the HMI. Setting the remote monitoring agent, sensor scanner, and HMI to medium priority doesn't work either, as it compromises the performance of all three processes. Because priority re-assignment doesn't resolve the issue, the development team must take the next step and attempt to change thread behavior — a costly solution at the integration stage.

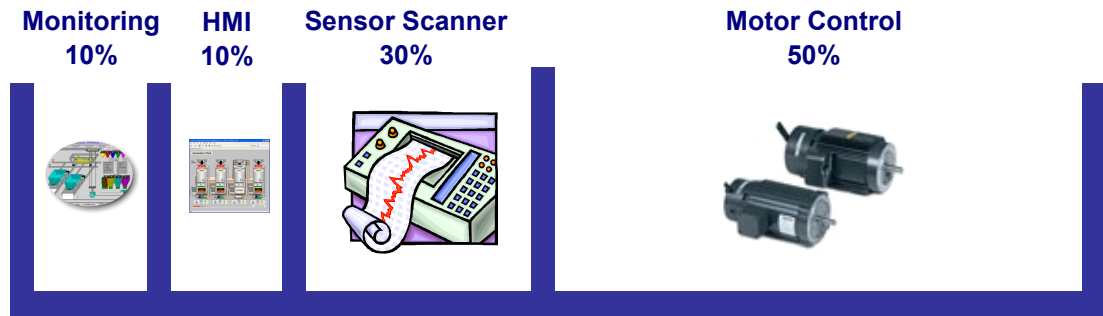


**Figure 2** — A simple automation system.

Partitioning provides a way to avoid these integration headaches. For instance, the system designer could set a CPU budget for each of the four partitions: 10% for the local HMI partition, 10% for the remote monitoring agent partition, 30% for the sensor scanner partition, and 50% for the motor control partition. See Figure 3.

With this approach, individual partitions can be verified according to their CPU budget. When the system is brought together at integration time, all processes will receive their share of CPU time as determined by their budget. As a result, the remote monitoring agent will no longer starve when the system becomes fully loaded. Furthermore, by simply controlling the partition budgets, the developers can trade off local HMI response time with remote update time to tune the system to the desired performance level. Properly implemented, a partitioning scheduler will allow developers to perform this tuning at runtime, without forcing them to rebuild their applications or the system image.

Although one could argue that appropriate system design and careful priority assignment can fix the problems in this relatively simple system, the many subtle interactions in a more complex system can result in task starvation issues that are much more difficult to troubleshoot and correct. It is in these types of systems where partitioning shows the largest benefits.



**Figure 3** — Assigning a CPU budget to each subsystem in a time-partitioned environment. Time partitioning prevents higher-priority subsystems, such as motor control, from starving lower-priority components of CPU time.

## Quantifying Development Savings

Using the build, test, find, and fix method to deal with task starvation is an expensive proposition. Often, starvation results in intermittent, unexplained system behavior rather than in hard failures. Consequently, it is difficult to collect the appropriate data for follow-up troubleshooting. Typically, the troubleshooting activities demand both a breadth and depth of system knowledge and therefore require a team to find and repair the problem. These activities can include the following:

1. A tester creates a problem report describing unexpected behavior that occurs during verification. For example, the HMI screen misses updates or is periodically unresponsive. Because the problem isn't easily reproducible, the tester can't easily collect the right information to assist with problem resolution.
2. The developer makes several trial-and-error attempts to reproduce the problem. Eventually, the developer determines that the cause of the problem isn't the HMI; rather, some other process is consuming too much CPU time and starving the HMI task.
3. At this point, troubleshooting and resolution of the problem broadens to include a number of people. The resolution can consist of thread-priority adjustments or of changing a process's behavior to correct the problems.
4. Each affected person makes the necessary modifications and tests, then integrates their changes into the system software build.
5. The tester retests for the problem and closes the problem report. This assumes that no additional problems were introduced as part of the changes, which could be extensive.

Based on these assumptions, Table 1 lists the conservative cost per incident.

Activity	Time Required
Verification and Problem Reporting (1 person tests and creates problem report)	1 day
Initial Troubleshooting (One person assigned to fix problem)	2 days
Joint Troubleshooting (3 people participate)	3 days
Joint Problem Resolution (3 people participate, 3 days each)	9 days
Re-verify (1 person retests)	1 day
<b>Total Effort</b>	<b>16 person days</b>

**Table 1** — Conservative cost of resolving a task-starvation incident. A moderately sized control system may experience several such incidents at integration time.

From this example, it is easy to see how task starvation can easily increase development costs and cause development delays; in this case, two to three calendar weeks. Even then, this example considers only four threads — many industrial control systems have several dozen threads that interact in hundreds of ways as they compete for CPU time. As a result, it is common for several cases of task starvation to occur in even a moderately sized system.

Since the system integration phase consumes a significant portion of any software project schedule, approaches that optimize this phase reduce development costs and result in faster time to market. By preventing one partition from robbing another partition's CPU time, partitioning provides a foolproof way of dealing with these task starvation issues.

## Minimum Effort

As complexity and code size grows, the probability that task starvation and other software defects will make their way into final product also grows. The cost of resolving such problems after a system has been deployed increases dramatically — not to mention the damage to the vendor's reputation and possible financial damages. Developers and vendors who develop code for industrial automation systems must therefore employ every tool and methodology at their disposal to ensure their code is correct and properly tested. The real challenge, however, is finding or implementing techniques that consume a minimum of development effort and computing resources. Properly implemented, CPU time partitioning provides such a solution. Moreover, it offers increased security and greater system availability by preventing malware or denial of service attacks from monopolizing the CPU. As a result, it allows embedded developers to create systems that are both well-integrated *and* well-protected.



### **About QNX Software Systems**

QNX Software Systems, a Harman International company, is the industry leader in realtime embedded OS technology. The component-based architectures of the QNX<sup>®</sup> Neutrino<sup>®</sup> RTOS and QNX Momentics<sup>®</sup> development suite together provide the industry's most reliable and scalable framework for building innovative, high-performance embedded systems. Global leaders such as Cisco, DaimlerChrysler, General Electric, Lockheed Martin, and Siemens depend on QNX technology for network routers, medical instruments, vehicle telematics units, security and defense systems, industrial robotics, and other mission- or life-critical applications. Founded in 1980, QNX Software Systems is headquartered in Ottawa, Canada, and distributes products in over 100 countries worldwide.

**[www.qnx.com](http://www.qnx.com)**