Using System Tracing Tools to Optimize Software Quality and Behavior

Thomas Fletcher, Director, Automotive Solutions
QNX Software Systems Ltd.
thomasf@qnx.com

Gaining Insight

At one time, embedded devices had relatively modest software requirements — typically, a few thousand source lines of code. Today, however, a device may contain hundreds of thousands, even millions, of source lines, and use large numbers of software components that share resources and interact in complex ways.

Despite this increase in complexity, embedded software must still address strict timing requirements, even when running on low-cost hardware. It must also perform flawlessly; software faults are, with few exceptions, unacceptable in embedded devices. Unfortunately, this same complexity makes it difficult to isolate the causes of slow, unpredictable, or incorrect behavior. Developers can always attempt to understand such problems by using source debuggers and other conventional debug tools. The problem is, these tools aren't designed to analyze the issues that arise when many software components interact with one another. In fact, because of their invasive nature, conventional debug tools can change the very behavior that the developer wishes to diagnose.

The ultimate goal in addressing these challenges is twofold: 1) maximize the use of hardware resources throughout the development process, and 2) pinpoint performance bottlenecks and errors as early as possible. The earlier errors are identified, the faster they can be corrected.

Gaining insight into component interactions is key to achieving this goal. In this paper, we explore how system tracing tools can provide insight into such interactions, allowing developers to root out logic flaws, reduce resource contention, resolve timing conflicts, track down memory errors, streamline components, and optimize performance. We also look at complex debugging scenarios and examine how a development environment based on Eclipse can allow tracing tools to share information with other diagnostic tools, providing smother workflow and faster isolation of hard-to-detect errors. And, finally, we consider the significant role of RTOS architecture in pinpointing and resolving system problems.

Narrowing the Scope

While traditional design and debug tools do a good job of detecting logic-type errors, especially in sequential programs, they fail to address many of the issues that arise in complex systems. For instance, how do you diagnose synchronization problems in a system that has multiple processes or multiple threads running concurrently on multiple CPUs? How, in such a system, do you pinpoint a memory problem or some other type of contention? And how do you maintain — and monitor — consistency and coherency throughout the system?

To diagnose problems like these, you must typically work through the following stages:

1. *Sporadic domain* — At this stage, your system is failing, but you have little information as to exactly when and why the problem is occurring.

2. *Temporal domain* — At this stage, you've narrowed down the scope of the problem and have some idea as to when the error is occurring. You may also have information on the sequence of events leading up to the problem, but haven't yet pinpointed the actual cause.

3. *Logical domain* — At this stage, you've identified the exact sequence of events or conditions that trigger the error and can reproduce the error on demand.

In essence, your goal is to move as quickly as possible into the logical domain, where conventional tools, such as debuggers and application profilers, can come into play. Unfortunately, bridging the gap from the sporadic domain to the logical domain can be very time-consuming and requires, among other things, tools that can deal with timing issues and memory errors in all three domains.

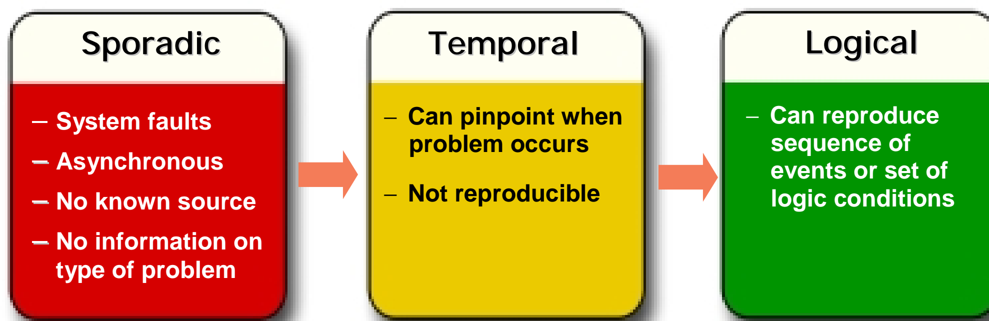

Figure 1 — Narrowing the scope so that problems can be addressed by conventional development tools.

Bridging Problem Domains

To bridge the domain "gap," you need tools that can help:

- *Isolate issues* — Whether you're looking for memory errors or logic problems, your tools must provide a snapshot for further analysis. And they must do so in a noninvasive manner.

- *Visualize root error conditions* — Your tools should offer a wide variety of techniques (graphs, charts, tables, and so on) to help you visualize data for further analysis.

- *Resolve problems* — To help achieve this goal, your tools should not only show where the problem is, but also share data seamlessly with other tools (e.g. debuggers) in the environment. Such tool-to-tool integration makes the task of bridging problem domains simpler, and faster.

- *Maintain system integrity* — Finally, your tools must help maintain system integrity during the problem-solving process. Consequently, they can't make changes that might adversely or unpredictably affect the behavior of any system component. And once you've made changes to any component, the tools should let you safely deploy and monitor that component in the field, again without affecting system behavior.

Gaining Visibility Through System Tracing

One way to bridge the domain gap — and to thereby gain visibility into your system — is through system tracing. By system tracing, we refer to a range of tools and techniques, including:

- *printf*() calls that annotate a program's progress

- system information tools (for instance, the Unix **top** command) that monitor task creation and track resource usage

- compiler-driven instrumentation techniques that enable application profiling and code coverage

- memory tracing tools that analyze a program's history of memory usage and that diagnose problems such as memory leaks and excessive memory fragmentation

- kernel-level instrumentation techniques that reveal events at an operating-system level, providing accurate timing traces and displaying complex interactions between multiple processes and threads

In most cases, tracing can present system activity as a linear sequence of events, letting you quickly determine which events caused what outcome to occur. Depending on the style of tracing used, additional about timing or task interrupt information can be extracted and incorporated into the analysis of the system's performance or as part of the debugging process.

Isolating Performance Problems

We've identified the challenges of analyzing complex systems; now let's explore how tracing tools can help pinpoint hard-to-detect problems, while maintaining system integrity. We'll start with system profiling.

When a complex system performs poorly, the sheer number of system interactions can make pinpointing the cause a frustrating, if not monumental, task. For instance, if dozens or hundreds of threads are running and interacting in a multiprocessor system, and one blocks unexpectedly, which event or interaction led to the problem? Without tools that provide a system-wide view, the cause may appear to be located in one part of the system when, in fact, it is located somewhere else.

To complicate matters, conventional tools are typically invasive, changing the behavior of the system being diagnosed. For instance, by halting only the program being debugged and not the whole system, a source debugger can change the order in which the system's operations occur. This phenomenon — often called the probe effect — can temporarily mask race conditions and introduce "errors" that occur only when debugging is performed.

Of course, high-quality tools for source debugging and application profiling are still important in today's complex, multiprocessor, multilanguage systems. But they're only useful once you've determined which component, or set of components, to fix. To do that, you must first understand how the system behaves as a whole. For instance, in a multiprocessor system, you must be able to determine which processors are exchanging messages, and in what order. You must also identify which processes or threads are involved in each inter-processor transaction and trace the execution path from one node to another — even if the processors are based on different architectures and the processes are written in different programming languages, such as C, C++, and Java.

To gain such insight, you need tools that can consolidate system-wide activity into a single context; for instance, the system profiler for the QNX® Neutrino® RTOS. Like a debugger that lets you trace the flow of control from one thread to another within a single program, this tool lets you "see" how the various components in a system interact, whether they all run on a single processor or across many heterogeneous processors. It is, in effect, a logic analyzer for your software system: if something goes wrong, the tool can help pinpoint when the event occurred, which software components were involved, what those components were doing, and, importantly, how to interpret the event.

The following table lists some common problems and describes how you can use a system profiler to diagnose them.

Problem	Technique
IPC bottleneck	Watch the flow of messages from one thread to another.
Resource contention	Watch threads as they change states.
Slow overall performance	View CPU usage to identify the processes or threads that consume the most CPU cycles.

Problem	Technique
Large interrupt latency	Search for user events to identify which thread is causing the delay, then insert custom events into that thread to pinpoint the problem.
Excessive thread migration in an SMP system	Watch threads as they migrate from one CPU to another.

Insight without impact

A good system profiler is noninvasive; it can provide insight without requiring code modifications and has minimal impact on system behavior. Properly implemented, it will let you diagnose a live system without interrupting or unduly degrading the services provided by that system — a real boon for high-end routers, 9-1-1 dispatch systems, and other applications that must remain continuously available.

To ensure this "noninvasivess," a system profiler uses fast, selective logging of system events, including messages, kernel calls, thread-state changes, and interrupts. User-written code doesn't have to be modified, since this event logging can be performed by an instrumented kernel.

Take, for example, the instrumented kernel for the QNX Neutrino RTOS, which is simply the standard QNX Neutrino microkernel with the addition of a small event-gathering module. When triggered, this module intercepts information about what the kernel is doing, generating time- and CPU-stamped events that are copied to a set of buffers grouped into a circular linked list. Once the number of events inside a buffer reaches a high-water mark, a logging utility either writes the data to a storage location (for instance, battery-backed SRAM) on the target or streams the data directly to the development host. — the latter approach eliminates the need for extra storage on the target.

Properly designed, an instrumented kernel can run at virtually same speed as a standard, non-instrumented microkernel. Performance is affected only when events are being collected. But, even then, the kernel can provide a variety of mechanisms to ensure minimal intrusion. For instance, the kernel could let you trigger event logging only when certain conditions occur. It could also provide user-definable filters so that the logging process only collects events of interest — developers can log as many or as few events as they need.

Of course, it's always possible that the overhead of event logging, no matter how small, will have a miniscule effect on system timing. To help you determine whether that is occurring, the instrumented kernel should be able to log all event types, including those generated by any event-logging operation. (It is important, by the way, that an instrumented kernel be fully preemptible; that way, any time-critical task can preempt an event-logging operation in order to meet its deadlines.)

Even when runtime event-filtering is applied, the events log from an instrumented kernel may contain data for many thousands of individual events. Thus, the system profiler should let you apply additional filters during analysis. Filters can significantly reduce the volume of data, making it easier to "zoom in" on events of interest.

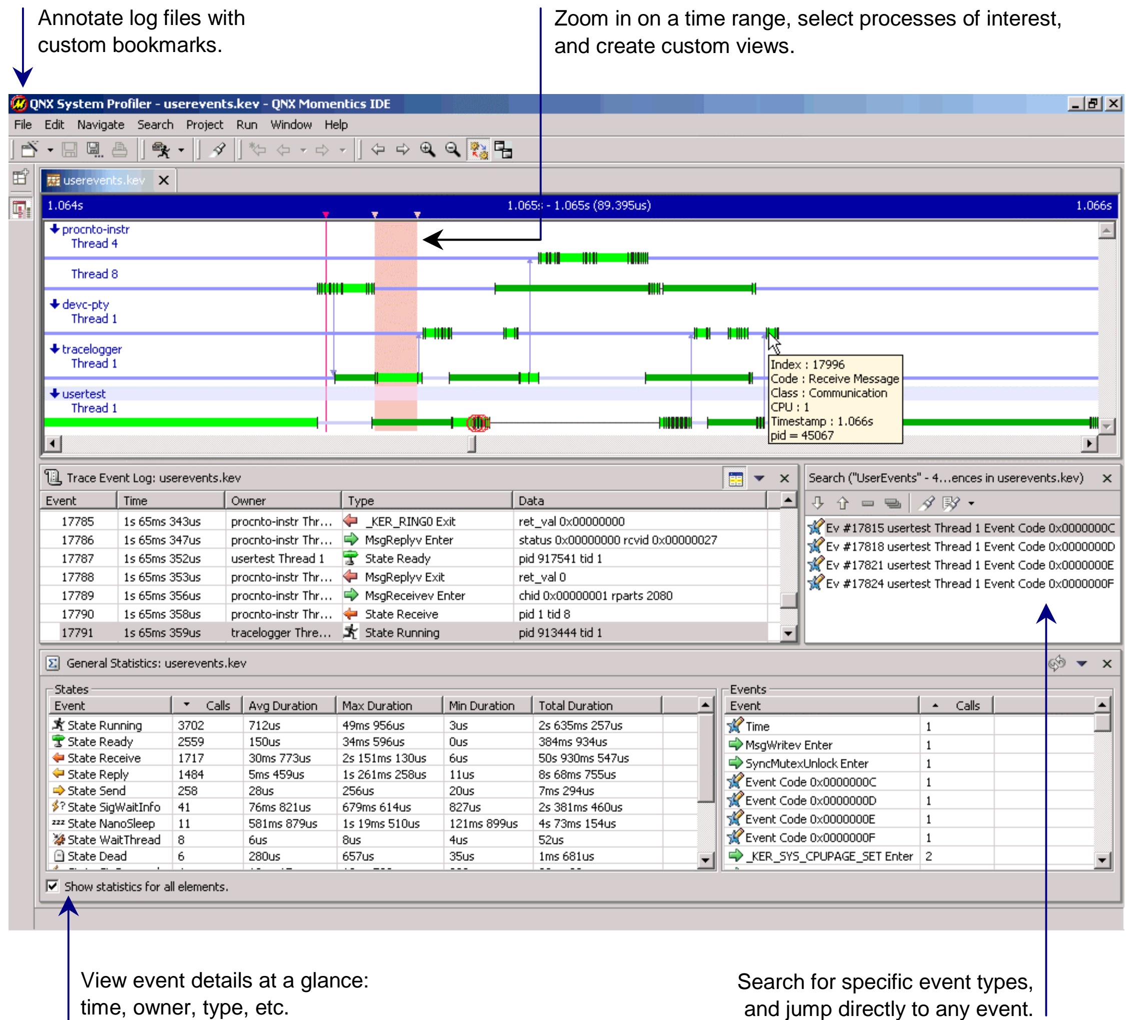

Figure 2 — A system profiler should be capable of displaying an enormous variety of events, including kernel calls, hardware interrupts, thread-state changes, and various forms of interprocess communication, such as signals and messages. As this screen capture illustrates, the profiler should also provide a rich set of event filters to help developers "zoom in" on events of interest and view complex interactions at a glance.

Injecting user-defined events

Sometimes, the data provided by an instrumented kernel can be so detailed that it becomes difficult to understand what, exactly, your code is doing. For such occasions, the kernel should let you insert your own user-defined trace events into the system. By placing such events throughout your application code or interrupt handler, you can construct an event sequence that shows which actions your program is reacting to, and which sections of your code were involved. Once you've injected the events, you can find them quickly, using the system profiler's search utility.

Isolating Memory Problems

Memory errors are the bane of many a programmer's existence. A well-designed memory tracing tool will help isolate and visualize such errors by providing a noninvasive profile of your embedded system. For instance, the tool should offer:

- **Continuous tracing** — Lets you monitor all the memory allocations and "frees" that your system is performing. The tool should keep a log of each allocation and match up all of the frees; that way, you can go back and trace where memory is being used, which components are allocating it, and which components are freeing it.
- **Snapshot tracing** — Lets you track and visualize where a memory error has occurred, when and where memory was initially allocated, and when and where memory was freed.

In effect, you can obtain a full backtrace of where the error occurred and, as a result, better understand the events leading up to the error.

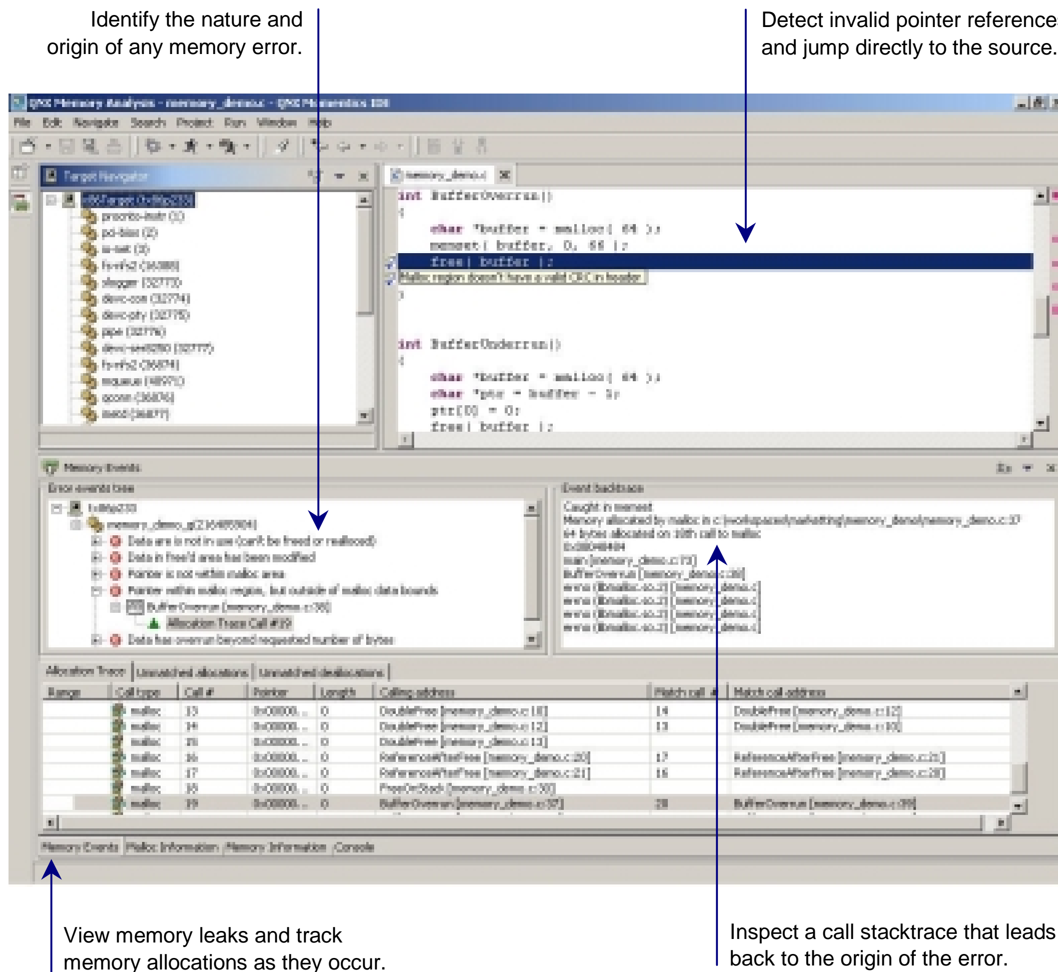

Figure 4 — Using a memory tracing tool, you can quickly detect overruns, underruns, double frees, invalid pointers, and other memory errors, without changing the behavior of the system you are monitoring.

Maintaining integrity

A good memory tracing tool is noninvasive; like a well-designed system profiler, it provides insight with minimal effect on system behavior. Consequently, it can let you diagnose a live system without affecting services provided by that system. If the system is based on a microkernel OS, you can also dynamically stop or restart the tool (or virtually any other software component, for that matter) without rebooting. There's no need to run the tool when it isn't needed.

The need for tool-to-tool integration

Once it detects an error, a memory tracing tool should be able to work closely with other tools, such as source debuggers, to help resolve the problem quickly. But, to achieve this level of integration, developers need a common framework that allows traditionally isolated or unextensible tools to share data with each other, without need for manual intervention. The open-source Eclipse development platform (www.eclipse.org) offers a good example of such a framework.

Eclipse gains its flexibility from *extension points*. These are well-known interfaces, defined in XML, that serve as the coupling points for development tools, or *plug-ins*, that add specific features (source debugging, memory tracing, etc.) to the Eclipse platform. An extension point consists of string identifiers that define a common interface for a particular set of functionality. In many cases, the extension point also defines the name of a Java class that is dynamically invoked in response to a user action. Any Eclipse plug-in can define extension points that other plug-ins can use and, conversely, fulfill extension points defined by other plug-ins.

To appreciate how this architecture works, consider the memory analysis plug-in that QNX Software Systems developed for its Eclipse-based QNX Momentics® development environment. This plug-in hooks into an extension point that allows monitoring tools to run when any target process is launched from the Eclipse framework. If a memory error occurs while processes are running on the target, the plug-in can record the error, then search through available debugging extension points to establish a remote debugging session with the process that generated the error. The plug-in can do all this without having any specific knowledge of the process and without actually knowing how to connect the debugger to the target.

Once the debugger is launched, you can step through the process and see how the memory area changes as the memory error is committed. Dropping directly into the debugger is just one option, however. An environment like this can, in fact, provide a variety of mechanisms for resolving the issue. For instance, you could:

- Double-click on the error to open a code editor that automatically annotates the offending line.
- Scan the entire program for "dangling" memory; that is, check to see if references are left dangling when memory is freed. You can thus detect whether any components are referencing memory they shouldn't be, or whether any memory has, in effect, escaped without being freed.
- Terminate the program and generate a process core dump file that you could then analyze offline with a debugger.

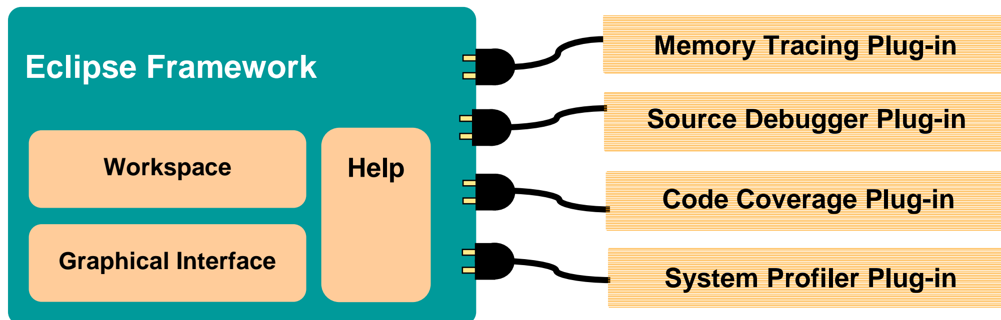

Figure 3 — Eclipse extension points, defined in XML, allow any diagnostic tool to act as a trigger for virtually any other tool. For instance, if a process commits a memory error, a memory tracing tool could record the error, then search through available debugging extension points to establish a remote debugging session with that process.

Tracking memory errors as tasks

An integrated environment based on Eclipse has other benefits as well. For instance, the IDE can intelligently track each memory error as a programming task and automatically annotate the program source code with a warning. Other tools in the IDE framework can then work with those annotations to help you resolve the issue.

The Role of RTOS Architecture

A discussion of RTOS architecture may seem out of place in a paper on diagnostic tools. But, as it turns out, a well-designed RTOS can make it much easier to isolate, visualize, and resolve a variety of system problems. To illustrate, let's look at three architectures used by RTOSs today: *realtime executive*, *monolithic*, and *microkernel*.

Realtime executive architecture

The realtime executive model is now 50 years old, yet is still being used by many RTOSs. In this model, all software components — OS kernel, networking stacks, file systems, drivers, applications — run together in a single memory address space. While efficient, this architecture has two immediate drawbacks: 1) a single pointer error in any module, no matter how trivial, can corrupt memory used by the OS kernel or any other module, leading to unpredictable behavior or system-wide failure; and 2) the system can crash without leaving diagnostic information that could help pinpoint the bug.

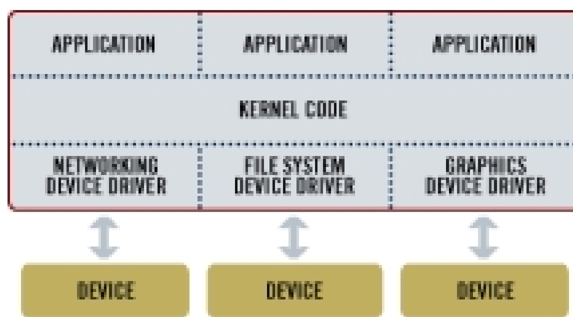

Figure 5 — In a realtime executive, an error in any software module can cause system-wide failure.

Monolithic architecture

Some RTOSs, as well as Linux, attempt to address this problem by using a monolithic architecture, in which user applications run as memory-protected processes (see Figure 6). The kernel is, as a result, protected from errant user code. Nevertheless, kernel components still share the same address space as file systems, protocol stacks, and drivers. Consequently, a single programming error in any of those services can still cause the entire system to crash. As with a realtime executive, where do you assign the blame? Where do you begin to look? Is the problem a memory error or some other type of error? There's often no easy way to find the answer.

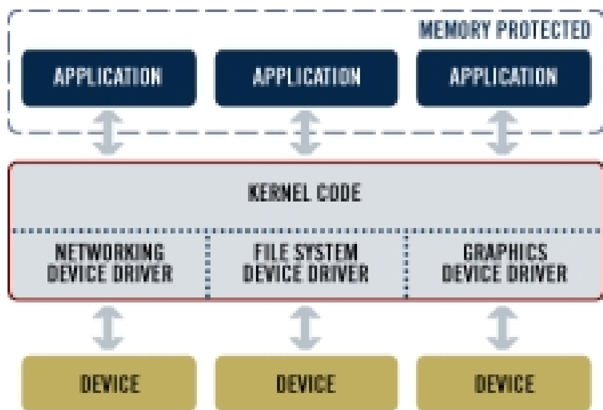

Figure 6 — In a monolithic OS, the system is protected from errant user code, but can still be corrupted by faults in any driver, file system, or protocol stack.

Microkernel architecture

In a microkernel RTOS, applications, drivers, file systems, and networking stacks all reside in separate address spaces, and are thus isolated from both the kernel and each other. With this approach, failures are easily contained: a fault in one component won't bring down the entire system. Moreover, it's easy to isolate a memory or logic error down to the component that caused it. For instance, if a memory fault occurs in a driver, the OS can identify the process responsible, at the exact instruction. Meanwhile, the rest of the system can continue to run, allowing you to isolate the problem and direct your efforts towards resolving it.

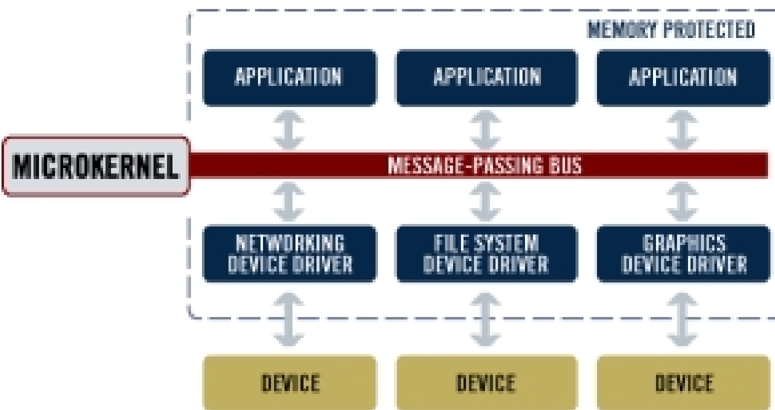

Figure 4 — In a microkernel OS, faults are easily isolated. Moreover, any component can fail and be restarted, without damaging the kernel or rebooting the entire system.

Uncovering Hidden Potential

Applications don't run in isolation; they affect each other significantly, sometimes simply by consuming CPU time. So, even if a system is performing acceptably, it can still be a candidate for system profiling, memory analysis, and other forms of system tracing. By graphically presenting complex behavior as a linear sequence of events, such tools often uncover hidden inefficiencies that, when corrected, allow for substantial increases in performance. In fact, it's worth your while to employ tracing tools throughout the entire development process. Not only will you identify problems early (when the problems are much easier to correct), but you'll also ensure that your system delivers all the features and performance that it is truly capable of supporting.

About QNX Software Systems

With millions of installations worldwide, QNX Software Systems Ltd. is the global leader in realtime, microkernel operating system technology. Companies like Cisco, DaimlerChrysler, Lockheed Martin, Panasonic, Siemens, and General Electric rely on QNX technology to build ultra-reliable systems for the networking, automotive, medical, military, and industrial automation markets. Founded in 1980, QNX Software Systems maintains offices throughout North America, Europe, and Asia.

www.qnx.com

© 2005 QNX Software Systems Ltd. All rights reserved. QNX, Momentics, and Neutrino are registered trademarks of QNX Software Systems Ltd. in certain jurisdictions. All other trademarks and trade names belong to their respective owners.