QNX SOFTWARE SYSTEMS

Memory Errors in Embedded Systems: Analysis, Prevention, and Risk Reduction

Elena Laskavaia and Paul Leroux
QNX Software Systems
elaskavaia@qnx.com, paull@qnx.com

Abstract

Memory errors are particularly harmful in embedded systems. These systems have limited memory resources, and are often deployed in environments where there are no second chances: a system brought down by a memory error that appears weeks or months after deployment may be unrecoverable and have costly or even disastrous consequences.

To address the challenges presented by memory errors, developers can take advantage of tools for memory analysis and debugging, and OS architectures that minimize the impact of memory errors on the system. This paper discusses memory analysis techniques for solving memory problems such as heap corruption and memory leaks; and memory profiling for optimization of memory use in embedded systems.

Introduction

Have you ever had a customer say, "It was working fine for days, then suddenly it just crashed"? If you are a developer, it is unlikely that you *haven't* heard this, just as it is likely that the *it* in question was *your* program, and that your program contained a memory error — somewhere. The problem was, and remains: *where*?

In fact, most developers find memory errors and leaks hard to detect and isolate, and therefore difficult to correct. The problem they face is that, by the time a memory problem appears (often by crashing the program), the corruption has usually become widespread, making the source of the problem difficult — though not impossible — to trace[1]. The inherent difficulty of pinpointing the source of a memory error is compounded in a multi-threaded environment, where threads share the same memory address space.

Memory errors in embedded systems

If eliminating memory errors and optimizing memory allocation is important in all software systems, it is doubly so in an embedded system.

First, memory is a precious commodity in embedded systems; it must be managed efficiently as well as reliably. Less than optimal memory allocation can waste precious RAM and hinder performance. Projects with inefficient memory allocation may be forced to remove useful software features, add more RAM, or upgrade to a faster processor, all solutions that reduce the value of the project or increase its cost. Inversely, efficient memory allocation can help maximize software functionality while minimizing hardware costs.

Second, embedded systems are often deployed in environments where recovery strategies may be

[1] We are not discussing Heisenbugs, which by definition are not reproducible, but, strictly, memory errors, which are difficult to trace. For a discussion of Heisenbugs, see Chris Hobbs, "Protecting Applications Against Heisenbugs". QNX Software Systems, 2010. www.qnx.com.

difficult or impossible to implement. With their propensity for showing up long after a system appears to be running reliably, memory errors are particularly insidious in these environments. One only needs to consider the failure of the Mars Global Orbiter in January 2007, which John McNamee, NASA deputy program manager for Mars Exploration at the Jet Propulsion Laboratory, attributed to a memory error: "two memory addresses were overwritten"[2]; or the New Horizons mission to Pluto and the Kuiper Belt, which was saved only by "an amazing stroke of luck" when it encountered an "uncorrectable memory error"[3].

Addressing the challenge

To address the challenges presented by memory errors in embedded systems, developers can use a suite of tools for memory analysis and debugging, and an RTOS architecture that minimizes the impact of memory errors on the system.

Tools

Memory analysis tools enable developers to quickly detect and pinpoint the source of memory errors such as leaks, buffer overruns, invalid deallocations and double frees. Just as importantly, these tools can expose subtle, long-term allocation problems that waste RAM and, in many cases, cause the system to fail weeks or even months after being deployed. Ideally, these tools work in an extensible environment such as Eclipse, which allows a memory analysis tool to share information with source code editors, debuggers, and other diagnostic tools, providing smoother workflow and faster error isolation of errors.

[2] Clinton Parks, "Faulty Software May Have Doomed Mars Orbiter", *Space News* (10 January 2007), www.space.com.

[3] Alan Stern, "NASA New Horizons Mission: The PI's Perspective: Trip Report", *PlutoToday.com* (26 March 2007), www.plutotoday.com

Memory errors

Memory errors are many and varied; they range from buffer under- and overruns, to multiple frees of the same memory block, to slow leaks. They can be classed, however, into two broad categories: heap corruption and memory leaks. Careful and thorough memory analysis is the most effective strategy for detecting and resolving both categories of error, as well as for optimizing memory use.

Heap corruption

To dynamically request memory buffers or blocks in a POSIX-based runtime environment, developers typically use the *malloc()*, *realloc()*, or *calloc()* function calls. To release these resources once they are no longer required, developers use *free()*. The system's memory allocator satisfies these requests by managing an area of program memory called the *heap*.

A program can erroneously or maliciously damage the memory allocator's view of the heap, resulting in *heap corruption*. For example, this corruption can occur if a program tries to free the same memory twice, or if it uses a stale or invalid pointer.

These silent errors can cause surprising, apparently random application crashes. The source of the error often proves extremely difficult to find, since the incorrect operation may have been executed in a different section of code long before the crash actually occurred.

Causes of heap corruption

Heap corruption has multiple causes. For example, it can occur when a program:

- passes an incorrect argument to a memory allocation function
- writes before the start of the allocated block (underrun error)

- writes past the end of the allocated block (overrun error)
- passes invalid information, such as a stale or uninitialized pointer, to a *free()* call

The outcome of these errors can depend on several factors, making diagnosis difficult with conventional debug tools. Consider memory overruns and underruns, which are among the most elusive and fatal forms of heap corruption. In an overrun error, the program writes past the end of the allocated block. Frequently, this overrun causes corruption in the next contiguous block in the heap. When this corruption occurs, the behavior observed depends on whether that block is allocated or free, and whether it is associated with a part of the program related to the error.

For instance, when an unallocated block becomes corrupted, a fatal error will usually occur during a subsequent allocation request. While the error might occur during the next allocation request, the actual observed outcome depends on a complex set of conditions that could result in a fault at a much later point in time, in a completely unrelated section of the program.

Detecting sources of heap corruption

Conventional debugging techniques rarely locate the cause of a memory error, because these errors can occur in one area of the code base but manifest themselves in another. In a multithreaded application, for example, a thread that corrupts the heap can cause a different thread to fault.

This phenomenon occurs because threads interleave requests to allocate or release memory. Conventional debugging typically applies breakpoints — such as stopping the program from executing — to narrow down the search for the offending section of code. While this approach may work for single-threaded programs, it is often ineffective for multi-threaded execution, because the fault may occur at a difficult-to-predict point.

There are multiple scenarios in which an error can occur in one area, while manifesting itself in another. For instance, the problem can happen when:

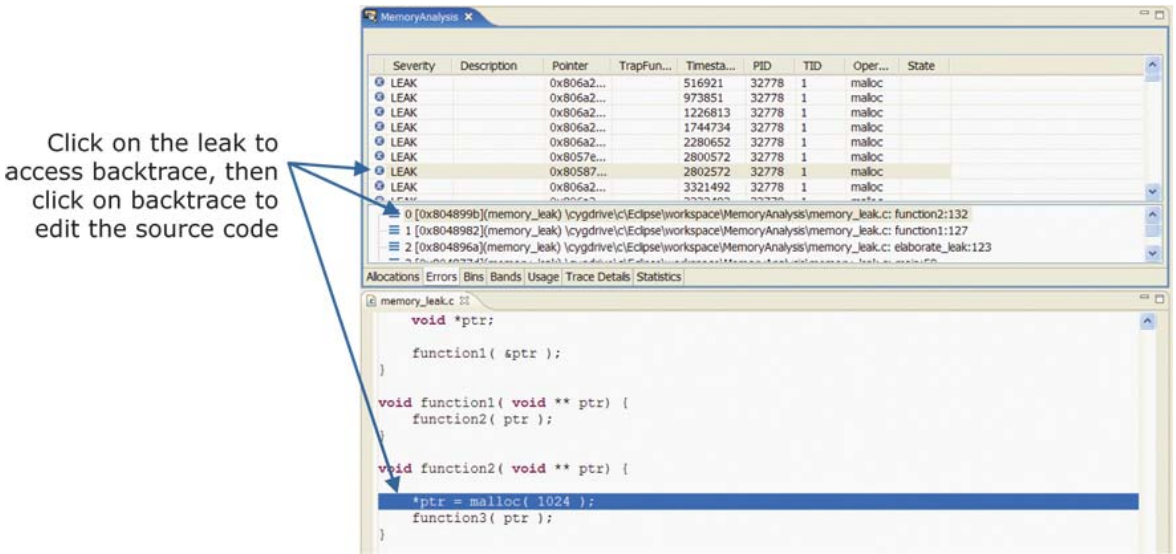

Figure 1: Using an error report to locate memory leaks.

- a program attempts to free memory
- a program attempts to allocate memory after it has been freed
- the heap is corrupted long before the release of a block of memory
- the fault occurs on a subsequent block of memory
- contiguous memory blocks are used

These problems all point to the importance of using effective memory analysis tools.

Memory leaks

Memory leaks occur when a program allocates memory, then forgets to free it later. In its mildest form, a memory leak allows the program to consume more memory than it actually needs. While wasteful, the leak may pose little danger if the program terminates occasionally; most modern OSs recover the memory (including lost memory)from terminated processes. However, if the program has a severe leak, or if it leaks slowly but never terminates—as may be required in an embedded system—the leak can ultimately consume all memory and cause system failure.

Programs can also use memory inefficiently. For instance, a program may allocate memory for a large data structure or continually grow a dynamic data structure, then fail to use the structure for a long period of time. Though, strictly speaking, this behavior does not constitute a memory leak, it can waste a significant amount of memory nonetheless, and can severely impact system performance.

Detecting memory leaks

A good memory analysis tool can report a memory leak in the same way that it reports other memory errors. Figure 1 above shows how the analysis tool from the QNX® Momentics® Tool Suite's Intergrated Development Environment (IDE) displays several leaks. As with other types of memory errors, the developer can click on any reported leak to get a backtrace to the associated source code.

Memory analysis

Memory analysis consists of capturing memory-related events on the embedded target, importing that information into the development environment, then using visualization tools to pinpoint errors and to identify areas that need correction or optimization.

Memory analysis workflow

Memory analysis not only lets developers find errors, but it also helps them fine-tune allocations to minimize RAM usage and ensure long-term system stability. Figure 2 illustrates the process of memory analysis, starting with observation and concluding with optimization. A well-designed memory analysis tool will provide robust support for each step of the memory analysis process:

Observe — First, the tool catches runtime errors, detects memory leaks, and displays all memory allocations and deallocations.

Correct — Next, the tool allows the developer to trace each error back to the offending source line.

Profile — Having eliminated obvious memory errors and leaks, the developer can now analyze memory usage over time, including average usage, peak usage,

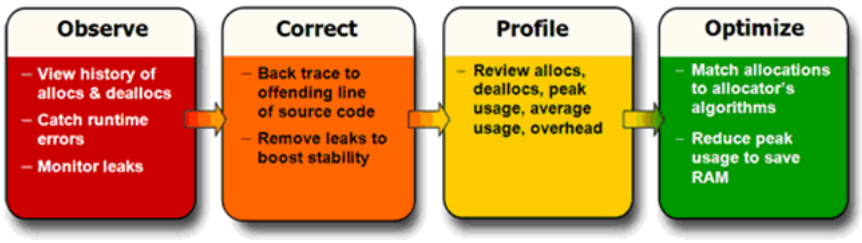

Figure 2: A typical memory analysis workflow.

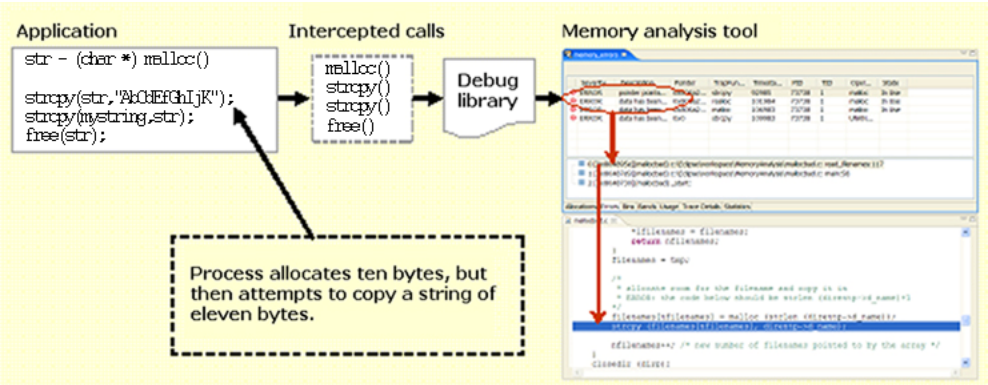

Figure 3: Memory analysis works by trapping memory-related API calls.

and overhead. Ideally, this tool will provide a visual presentation of longer-term memory usage, allowing immediate identification of spikes in memory allocation and other anomalies.

Optimize — Finally, using the tool's profiling information, the developer can fine-tune each program's memory usage to take best advantage of the system's memory allocation algorithm. Among other things, this optimization can minimize peak memory usage and overhead, lower RAM costs, and prevent the system from slowly running out of memory.

Observe and correct: the debug library

A memory analysis tool typically works in concert with a debug version of the memory allocation library to catch memory errors. This library, which is dynamically loaded at runtime, allows the tool to report overruns, underruns, double frees, and other errors, without requiring any modifications to the application source code.

To catch errors, the debug library records all memory allocations and deallocations — *malloc()*, *calloc()*, *free()*, *new()*, *delete()*, etc. — and performs a sanity check on their pointer values. It also intercepts string- and memory-related functions — *strcmp()*, *memcpy()*, *memmove()*, etc. — and verifies their parameters before using them. If the library detects an invalid pointer or incorrect parameter, it records the error and reports it to the IDE, making the information available to the memory analysis tool.

For example, if a program allocates 16 bytes of memory but forgets the terminating NUL character, then uses *strcpy()* to copy a 16-byte string into the block, the library will report the error to the IDE. The error message can indicate the point at which the error was detected, the program location that made the request, and information about the heap buffer that contained the problem.

As well as recording memory allocations and deallocations, a memory analysis tool should offer developers the option of recording a memory-analysis session, which they can play back at any time for analysis.

Using a debug library

Using a debug library should be straightforward. For instance, the QNX Momentics Tool Suite supports the creation of a launch configuration that automatically uses the debug library when it starts a specified application.

To illustrate how the debug library works, let's say a process allocates a string of ten bytes, then attempts to copy eleven bytes of string data into that memory space. In response, the debug memory allocation library would intercept the *malloc()* call, the two *strcpy()* calls, and the *free()* call; see Figure 3.

In this case, the library sees a mismatch between the parameter for `str`, which is allocated ten bytes, and

the other parameter, which is a string of eleven bytes. It also detects that in the case of the *free()*, the buffer referenced by `str` is now being corrupted, and that the *free()* call will therefore probably fail as well.

Figure 4 provides a closer look at what the IDE would show. At the top is a list of errors; below it is a back-trace view that shows a call stack of the steps in the code that led to the error. A click on an error provides a backtrace, and a click on the backtrace shows the associated source code, displayed in the source-code editor. From there, it becomes a relatively simple matter to correct the problem, do a rebuild and, using the memory analysis tool, confirm that the memory error no longer occurs.

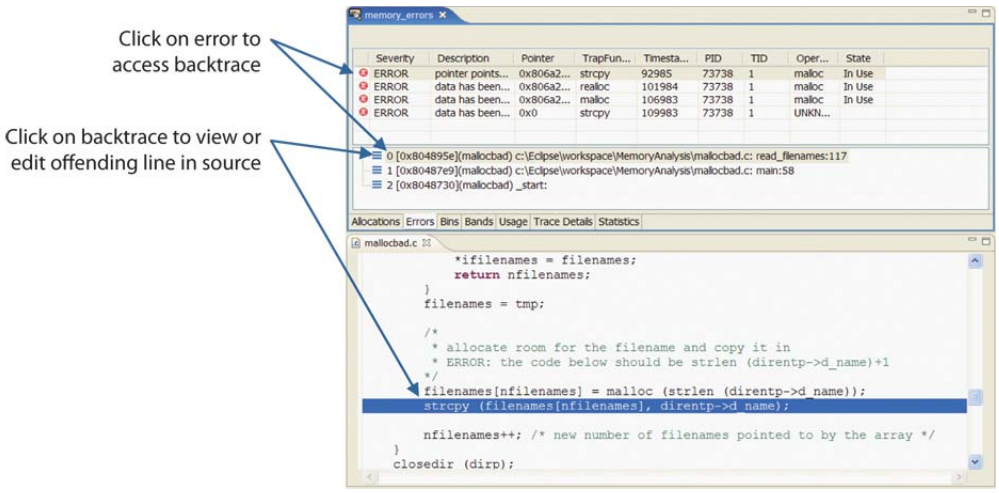

Figure 4: Using the error report to locate the offending source line.

IDE support for error tracking

A well-designed IDE will provide several options for dealing with errors. For instance, the QNX Momentics Tool Suite intelligently tracks each memory error as a task and automatically annotates the program source code with a warning. When the IDE detects a memory error, it lets the developer:

- let the program continue uninterrupted

  OR

- stop the program and immediately switch control to the debugger view, where the developer can use the debugger features to pinpoint the problem

  OR

- terminate the program and generate a process core dump file for postmortem analysis.

Pointer checking — Mudflap

Pointer errors are a common source of heap corruption, and they can be difficult and time-consuming to correct. Mudflap in the development environment can significantly reduce the effort required to chase down pointer errors.

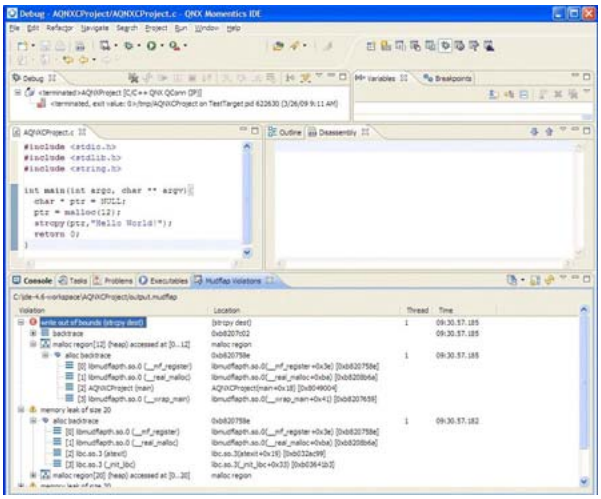

Figure 5: A screenshot from the QNX Momentics Tool Suite's Mudflap module.

Mudflap provides runtime pointer checking capability to the GNU C/C++ compiler. Since Mudflap is included with the compiler, it does not require additional tools in

the tool chain, and it can be easily added to a build by specifying the necessary GCC options.

Mudflap instruments risky pointer and array de-referencing operations, some standard library string/heap functions, and some other associated constructs with range and validity tests. Instrumented modules detect buffer overflows, invalid heap use, and some other classes of C/C++ programming errors.

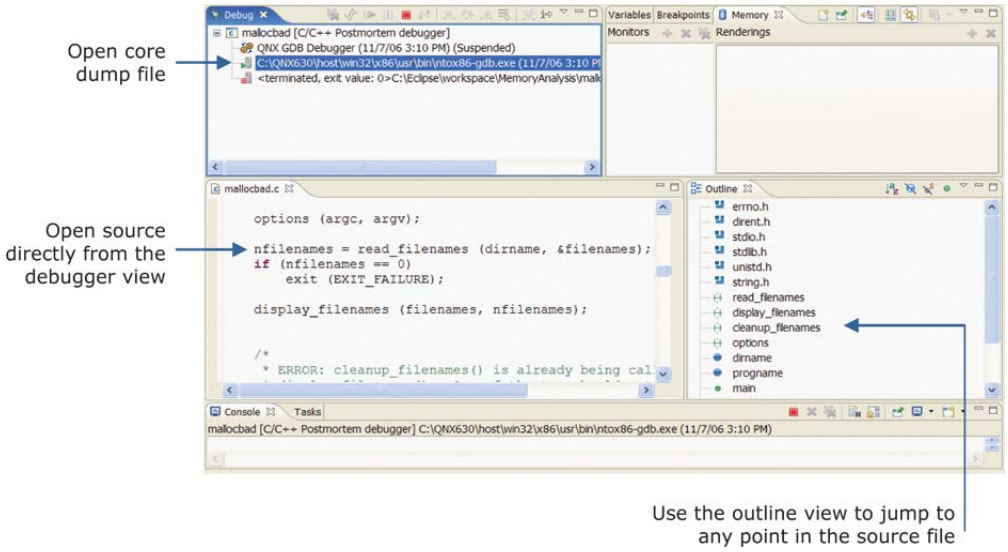

Figure 6: A post mortem debug session.

In the QNX Momentics Tool Suite, the instrumentation relies on a separate runtime library, which is linked into a program when the compile and linker options are provided for the build.

Postmortem debugging

If, for any reason, the above methods fail to trap the error and the program terminates abnormally, a background "dumper" utility can write a core dump of the program's state to the host file system. This dump file, viewable with source debugging tools, provides the information needed to identify the source line that caused the problem, along with a history of function calls, contents of data items, and other diagnostic information. The developer can then debug the dump file just as he would debug an application on the target system, stepping through call stacks to determine what events led to the crash.

Memory profiling

After eliminating obvious memory errors with the help of a memory analysis tool, developers should have a stable system which they can begin optimizing through memory profiling. Profiling is often done during a project's integration phase, when it becomes important to gauge how the system consumes memory over time.

Important measurements to make include peak memory usage; distribution of allocation sizes (16 bytes, 32 bytes, etc.); long-term trends in memory usage (for instance, does allocated memory slowly grow over time?); and overhead associated with how a program allocates memory.

Memory use by process

When optimizing memory usage, it is important to know how much memory individual processes use. First, if a process uses only a small portion of available memory — five percent, for instance — optimization of that process is unlikely to produce noticeable

improvements. Second, techniques for optimizing different types of memory can differ significantly.

Memory wastage

The manners by which a program can waste memory are manifold:

Linked lists of large data items — Linked lists or variable data structures are convenient to use, but can be very unpredictable.

An application may add or remove objects from a linked list of large data items, for example, and the length of that list may vary because of application load or throughput.

Data packet copy and forward — Programs "copy and forward" for a variety of purposes. For instance, a packet processing system may copy and forward every time it handles a packet, and it may issue a *malloc()* call (or, in C++, a *new* operator) for every copy.

Many class constructions — Objects and object-oriented programming are very convenient, but they can potentially waste memory, especially when classes contain large data structures. Every time a program creates one of these structures, it allocates a certain amount of memory.

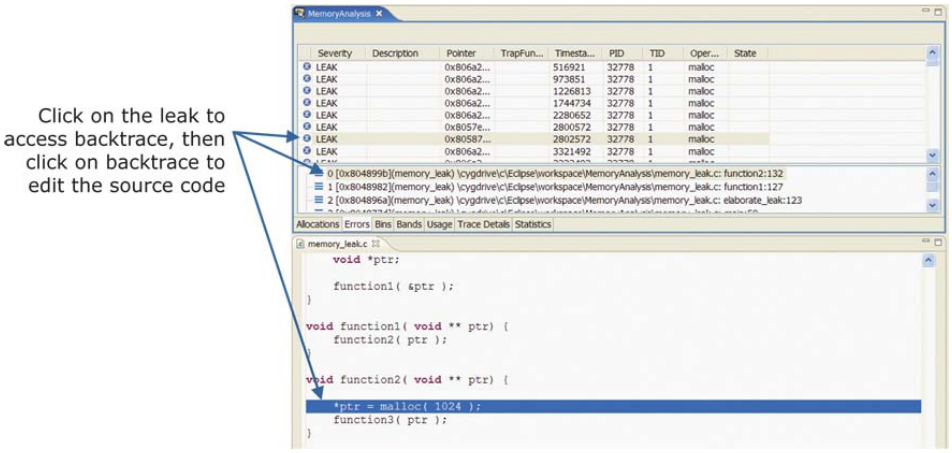

Figure 7: Using an error report to locate memory leaks.

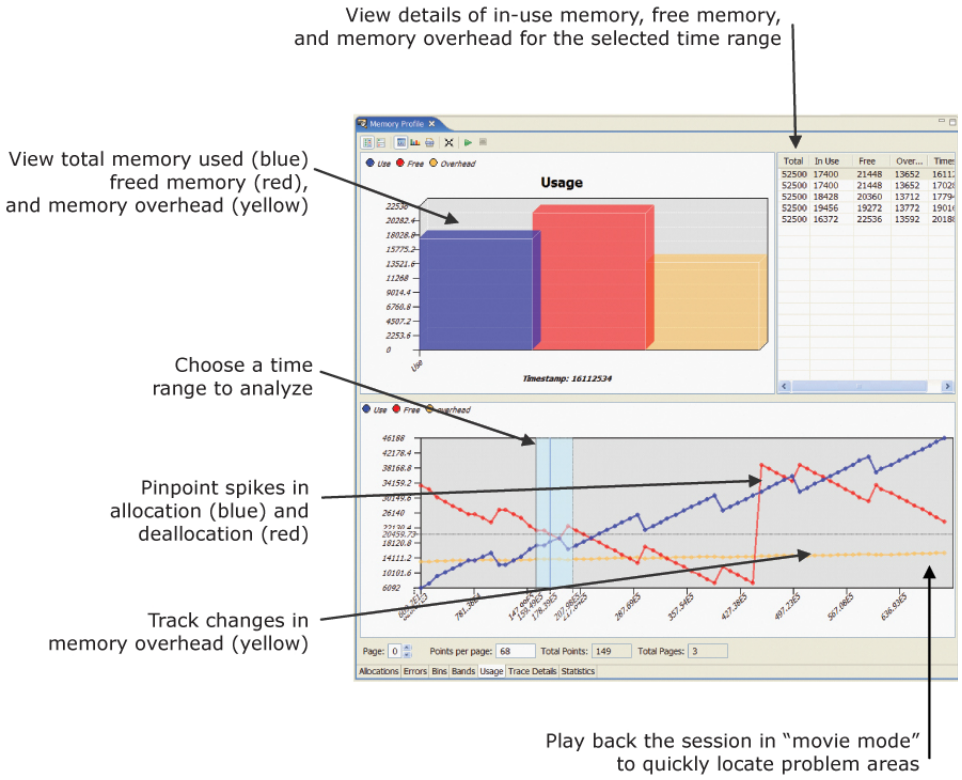

Figure 8: A profile session that shows growing memory usage over time (blue line).

In some cases, the program frees this memory inefficiently. In other cases, the program may continue allocating memory for a long time, then free a large

amount of memory all at once, rather than freeing allocated blocks as soon as they are no longer required. In still other cases, this memory may remain allocated indefinitely, even though the application is not using the data structure.

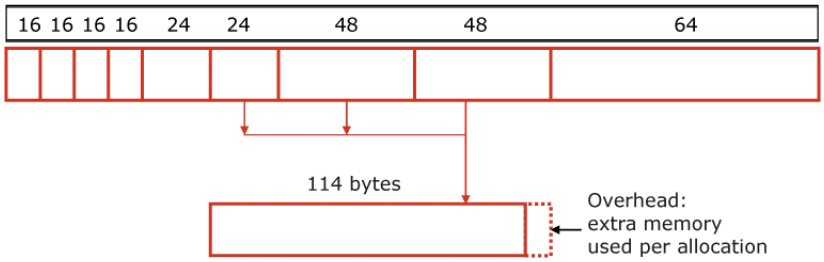

Figure 9: Overhead incurred by a mismatch between a malloc() call and the memory allocation scheme.

Diagnosing memory wastage

To help diagnose conditions of memory wastage, a memory profiling tool must monitor all the allocations and deallocations performed by the system. It should also keep a log of all allocations and match these with all the deallocations. This information permits the developer to go back and trace where memory is being used, which components are allocating it, and which components are freeing it.

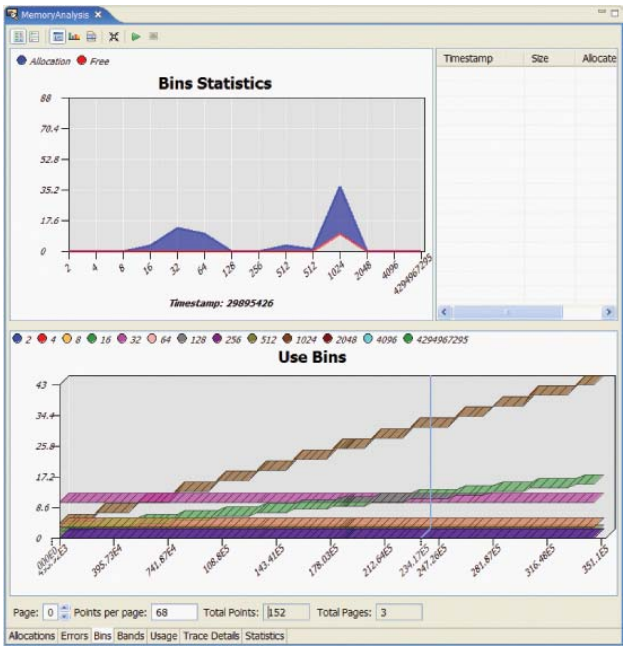

Figure 10: Distribution of memory allocations over time.

For example, consider the memory profile in Figure 8, generated by the QNX Momentics memory analysis tool. Using this profile, it is a simple matter to identify peak memory usage, average usage, and, importantly, any anomalies in how the application uses memory. Note how the timeline graph at the bottom of the screen shows that memory allocation (ascending blue line) is growing over time, indicating potential memory mismanagement or a possible memory leak.

Memory overhead

To avoid memory fragmentation and ensure deterministic response, most RTOSs use a memory-allocation scheme that divides the available heap space into smaller, fixed-size blocks. The memory allocator then distributes every allocation request among these blocks. To use memory efficiently, a developer must ensure that most allocation requests conform to these predetermined block sizes; otherwise, excessive memory overhead can result.

For example, in Figure 9, the memory allocator has divided the heap into various fixed-sized blocks: four 16-byte blocks, two 24-byte blocks, two 48-byte blocks, and so on. If an application does a *malloc()* of 114 bytes, it has to grab 24 bytes, 48 bytes, and another 48 bytes, for a total of 120 bytes. The difference between the total block size and the memory requested (120-114=6) yields an overhead of roughly five percent. In other words, five percent of the memory allocated goes to waste. This number is not huge, but the more such calls the program makes, the more that the memory overhead will grow.

To help avoid this problem, a memory analysis tool must let the developer compare allocation requests with the memory allocator's predetermined block sizes. With this information, he can quickly tune and optimize memory allocation on a per-application basis.

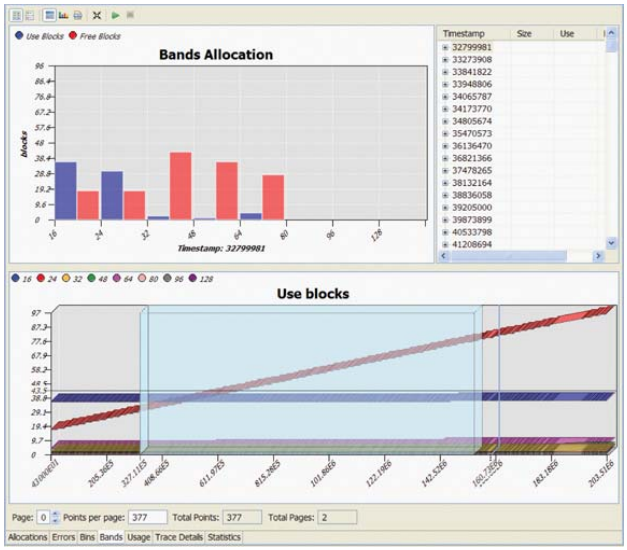

Figure 11: Comparison of the pattern of allocation requests with block sizes used by the memory allocation scheme.

To begin with, the memory analysis tool can display a distribution of memory allocations. For example, in Figure 10, the memory analysis tool displays two views:

Bin Statistics — The distribution of allocation requests. The peak indicates that the program frequently makes allocation requests of 1024 bytes.

Use Bins — How often the program has requested particular bin sizes over time. For example, the brown steps indicate multiple allocations of 1024 bytes and the green steps indicate multiple allocations of 16 bytes.

In Figure 11, the memory analysis tool displays bands, or block sizes. The block sizes in Figure 9 are 16, 24, 48, and 64, and that is close to what we see here. The Bands Allocation display at the top of the screen overlays what the program asked for with the bands (block sizes) used by the memory allocation algorithm. Using this graph, it is easy to determine whether a mismatch exists between the most commonly requested memory size and the block sizes used by the memory allocation scheme.

RTOS architectures

A discussion of RTOS architecture may seem out of place in a discussion of memory analysis tools. But as it turns out, a well-designed RTOS can make many memory problems much easier to isolate and resolve. To illustrate, let's look at the three most common RTOS architectures: *realtime executive*, *monolithic*, and *microkernel*.

Realtime executive architecture

The realtime executive model is now 50 years old, yet still forms the basis of many RTOSs. In this model, all software components — OS kernel, networking stacks, file systems, drivers, applications — run together in a single memory address space.

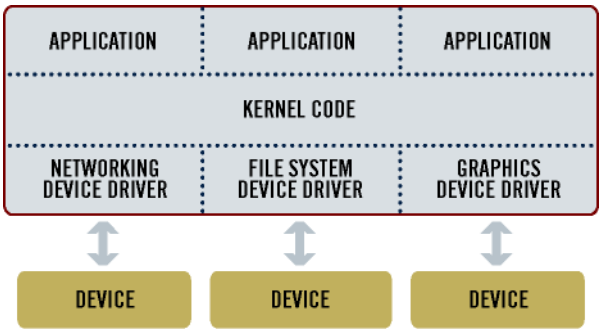

Figure 12: In a realtime executive, any software module can cause system-wide failure.

While efficient, this architecture has two immediate drawbacks. First, 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. Second, the system can crash without leaving

diagnostic information that could help pinpoint the location of the bug.

Monolithic architecture

Some RTOSs, as well as Linux, attempt to address the problem of a memory error provoking a system-wide corruption by using a monolithic architecture, in which user applications run as memory-protected processes.

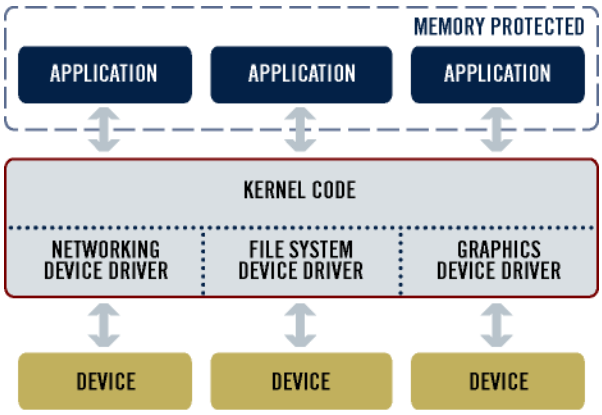

Figure 13: In a monolithic OS, the kernel is protected from errant user code, but can still be corrupted by faults in any driver, file system, or networking stack.

This architecture, shown in Figure 13, does protect the kernel from errant user code. However, 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 cause the entire system to crash. As with a realtime executive, where do you assign the blame? Where do you look? Is the problem a memory error or some other type of error? Unfortunately, there is often no easy way to find the answer.

Microkernel architecture

In a microkernel RTOS, applications, device drivers, file systems, and networking stacks all reside outside of the kernel in separate address spaces, and are thus isolated from both the kernel and each other. This approach offers superior fault containment: a fault in one component will not bring down the entire system.

Moreover, it is a simple matter to isolate a memory or logic error down to the component that caused it. For instance, if a device driver attempts to access memory outside its process container, the OS can identify the process responsible, indicate the location of the fault, and create a process dump file that is viewable with source-level debugging tools.

Meanwhile, the rest of the system can continue to run, allowing developers to isolate the problem and direct their efforts towards resolving it.

Compared to conventional OS kernels, a microkernel also provides a dramatically faster Mean Time to Repair (MTTR). Consider what happens if a device driver faults: the OS can terminate the driver, reclaim the resources the driver was using, and then restart the driver, often within a few milliseconds. With conventional operating systems, recovery would require a device reboot — a process that can take seconds to minutes.

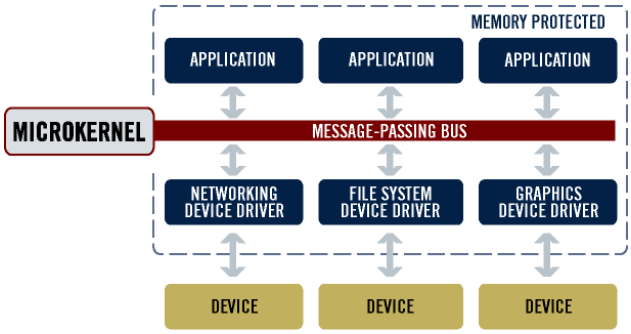

Figure 14: In a microkernel OS, memory faults in drivers, protocol stacks, and other services cannot corrupt other processes or the kernel. Moreover, the OS can automatically restart any failed component, without need for a system reboot.

Conclusion

The software in embedded systems is becoming very complex, making it harder than ever to pinpoint and stamp out memory errors. Memory analysis allows developers to visually pinpoint memory errors that conventional source debuggers are unable to detect. They can also help optimize long-term memory usage, thereby reducing RAM requirements and ensuring that the system does not run out of memory days, weeks, or months after deployment.

Developers and project implementers should not wait for a memory error to manifest itself before they decide to use a memory analysis tool. Even if a system appears to perform acceptably, a memory analysis tool can not only reveal latent memory errors early, when they are easier to correct, but they can also uncover hidden inefficiencies that, when corrected, allow for substantial improvements in performance and memory usage.

References

"Heap Analysis: Making Memory Errors a Thing of the Past", QNX Neutrino RTOS *Programmer's Guide*.

Hobbs, Chris. "Protecting Applications Against Heisenbugs". QNX Software Systems, 2010. www.qnx.com.

Laskavaia, Elena. "Memory Profiling Using the QNX IDE 4". QNX Software Systems, 2007. www.qnx.com.

Parks, Clinton. "Faulty Software May Have Doomed Mars Orbiter". *Space News* (10 January 2007). www.space.com.

Stern, Alan. "NASA New Horizons Mission: The PI's Perspective: Trip Report". *PlutoToday.com* (26 March 2007) www.plutotoday.com.

About QNX Software Systems

QNX Software Systems is the leading global provider of innovative embedded technologies, including middleware, development tools, and operating systems. The component-based architectures of the QNX® Neutrino® RTOS, QNX Momentics® Tool Suite, and QNX Aviage® middleware family together provide the industry's most reliable and scalable framework for building high-performance embedded systems. Global leaders such as Cisco, Daimler, General Electric, Lockheed Martin, and Siemens depend on QNX technology for vehicle telematics and infotainment systems, industrial robotics, network routers, medical instruments, security and defense systems, and other mission- or life-critical applications. The company is headquartered in Ottawa, Canada, and distributes products in over 100 countries worldwide.

www.qnx.com