# The ten secrets of embedded debugging

## Stan Schneider and Lori Fraleigh

## 9/15/2004 4:30 PM EDT

**Debugging your system may be the most important step in the development process. Here are ten hard-won lessons from the embedded trenches.**

Once upon a time, a developer seeking a higher plane of embedded proficiency climbed the Mountain of Experience to consult The Oracle at the summit. The Oracle knew that embedded systems programmers faced perils unknown to mere mortals, such as devices that must run for months without crashing, strict memory and processing limits, and the mysteries of real-world interactions.

New programmers still flush with classroom memories of template libraries and single-step debuggers could not absorb The Oracle's wisdom. It took many frustrating years of experience with embedded systems to appreciate the value of The Oracle's enlightenment. Each day, programmers would come seeking answers to questions like, "How can I write faster code?" and "Which compiler is best?" The Oracle rejected these immature supplicants with a terse "Ask not questions of little import. Only those who are worthy may understand my answers."

Our developer asked The Oracle, "O Oracle, how can I see and understand what my system is doing?"

The Oracle responded, "Your question shows unusual maturity, young pilgrim, for understanding is the key that opens the door to improvement. It is better to light a single candle than to curse the darkness."

And The Oracle shared these Ten Secrets, the results of years atop the Mountain of Experience:

1. Know your tools
2. Find memory problems early
3. Optimize through understanding
4. Don't put needles in your haystack
5. Reproduce and isolate the problem
6. Know where you've been
7. Make sure your tests are complete
8. Pursue quality to save time
9. See, understand, then make it work
10. Harness the beginner's mind
11. Know your tools

*"When all you have is a hammer, every problem looks like a nail."*

Good mechanics have many tools; you can't fix a car with just a hammer. Like good mechanics, good programmers need to be proficient with a variety of tools. Each has a place; each has a Heisenberg effect; each has power.

Tools promise great insight into system operation. Tools designed for embedded systems let you see, live while it happens, what your program is doing, what resources it's using, and how it interacts with the external world. The insight they provide is truly powerful; you can often quickly

spot problems, issues, or inefficiencies that would take days to discover by other means.

To deliver productivity, tools have to provide more than powerful visualization: they must also be usable on the system. Real-time systems are much more sensitive to intrusion than desktop systems. Will that profiler slow your system? Not a problem for your GUI. But if it makes the network interface drop packets, you may not be able to profile your embedded application at all.

Tools must also be used. Inserting instrumentation or code changes requires time and effort. Even simple recompilation with new flags can be a barrier, especially for larger team-built applications. As a result, most programmers don't immediately turn to the tool. They'll try changing the code and recompiling many times first. While each step seems insignificant ("I'll just try a **printf**"), this pot-shot technique is a huge drain on productivity. Fear and inertia are also factors; many programmers don't even try the tools they already own.

A **source-level debugger** lets you step through your code, stop it, and then examine memory contents and program variables. It's the fundamental first tool to bear on a problem.

A **simple printf statement** (or your language's equivalent) is perhaps the most flexible and primitive tool. Printing out variable values allows you to discover how your program is operating. Unfortunately, **printf** is both clumsy to use (requiring code changes and recompiling) and quite intrusive because it greatly slows execution. It also can produce reams of data that obscure the real problem.

**In-circuit emulators (ICE) and JTAG debuggers** allow you to carefully control the running chip. During the early stages of developing a custom board, an ICE is indispensable. There's just no substitute for full control of the processor when you don't trust the hardware. You can start debugging the board as soon as it comes out of reset, allowing you to see everything going on in those first crucial microseconds. You can debug code in ROM. You can usually ignore the minimal intrusion. However, once the operating system is up, hardware tools sometimes fall short. Hardware emulators aren't the same as the production CPU. They don't do well debugging multiple processes. They aren't nearly as flexible as a full set of software tools.

**Data monitors** show you what your variables are doing without stopping the system. Data monitors can collect data from many variables during execution, save the histories, and display them in a live graphical format.

**Operating system monitors** display events, such as task switches, semaphore activity, and interrupts. These monitors let you visualize the relationships and timing between operating system events. They easily reveal issues like semaphore priority inversion, deadlocks, and interrupt latency.

**Profilers** measure where the CPU is spending its cycles. A profiler can tell you where your bottlenecks are, how busy the processor is, and give you hints on where to optimize. Because they show where the CPU has been, profilers also give you insight as to what's really running—a valuable peek into the system's operation.

**Memory testers** search for problems in the use of memory. They can find leaks, fragmentation, and corruption. Memory testers are the first line of attack for unpredictable or intermittent problems.

**Execution tracers** show you which routines were executed, who called them, what the parameters were, and when they were called. They are indispensable for following a program's logical flow. They excel at finding rare events in a huge event stream.

**Coverage testers** show you what code is being executed. They help ensure that your testing routines exercise all the various branches through the code, greatly increasing quality. They can also aid in eliminating dead code that's no longer used.

## Find memory problems early

Memory problems are insidious. They fall into three main types: leaks, fragmentation, and corruption. The best way to combat them is to find them early.

Memory leaks are the best-known "hidden" programming problem. A memory leak arises when a program allocates more and more memory over time. Eventually, the system runs out of memory and fails. The problem is often hidden; when the system runs out of memory, the failing code frequently has nothing to do with the leak.

Even the most diligent of programmers sometimes cause leaks. The most obvious cause is programming error; code allocates memory and fails to free it. Anyone who has traced a leak in a complex piece of code knows it can be nearly impossible to find. Much more insidious (and common) leaks occur when a library or system call allocates memory that doesn't get freed. This is sometimes a bug in the library, but more often it's a mistake in reading the application programming interface documentation.

For example, programmers at Nortel were porting a large UNIX networking application to VxWorks. They found a leak during their final burn-in testing. The leak slowly ate 18 bytes every few seconds, eventually causing a crash. There was no indication where the leak was coming from or why it was occurring. Poring over the code for weeks didn't provide any clues.

Ten minutes with a leak-detection tool solved the mystery. It turned out that **inet_ntoa**, a simple network call, allocated memory in VxWorks but not in UNIX. The call was well documented; there was simply a difference between the two implementations. A one-line change fixed the problem.

The Nortel programmers were lucky. Unit and system testing rarely reveal leaks. A slowly leaking system may run for days or months before any problem surfaces. Even extended testing may not find a leak that only occurs during one high-traffic portion of the code during real use.

In fact, most leaks are never detected, even in fielded systems. They simply result in "random" failures, blamed on hardware, loads, power surges, operator error, or whatever happened to be going on at the time. In the best case, reliability isn't critical and users learn to reboot periodically. In the worst case, leaks can destroy the customer's confidence, make the product worthless or dangerous, and cause the product or project to fail.

Since leaks are so damaging, many methods have been developed to combat them. There are tools to search for unreferenced blocks or growing usage, languages that take control away and rely on garbage collection, libraries that track allocations, even specifications that require programmers to forgo runtime-allocated memory altogether. Each technique has pros and cons, but all are much more effective than ignorance. It's a pity so many systems suffer from leaks when effective countermeasures are available. Responsible programmers test for leaks.

Fragmentation presents an even sneakier memory challenge. As memory is allocated and freed, most allocators carve large blocks of memory into smaller variable-sized blocks. Allocated blocks tend to be distributed in memory, resulting in a set of smaller free blocks to carve out new pieces. This process is called fragmentation. A severely fragmented system may fail to find a single free 64k block, even with megabytes of free memory. Even paged systems, which don't suffer so badly, can become slow or wasteful of memory over time due to inefficient use of blocks.

Some fragmentation is a fact of life in most dynamically allocated memory. It's not a problem if the

system settles into a pattern that keeps sufficient memory free; the fragmentation will not threaten long-term program operation. However, if fragmentation increases over time, the system will eventually fail.

How often does your system allocate and free memory? Is fragmentation increasing? How can you code to reduce fragmentation? In most systems, the only solution is to get a tool that shows you the memory map of your running system. Understand what is causing the fragmentation and then redesign (often a simple change) to limit its impact.

Any code written in a language that supports pointers can corrupt memory. There are many ways corruption can occur: writing off the end of an array, writing to freed memory, bugs with pointer arithmetic, dangling pointers, writing off the end of a task stack, and other mistakes. In practice, we find that most corruption is caused by some saved state that isn't cleaned up when the original memory is freed. The classic example is a block of memory that's allocated, provided to a library or the operating system as a buffer, and then freed and forgotten. Corruption is then waiting to happen at some "random" time in the future. The corrupted system will then fail (or not) at some other "random" time. Corrupted systems are completely unpredictable. The errors can be hard to find, to say the least.

By the way, don't delude yourself that a "protected memory" system prevents corruption. Memory protection only stops other processes from cross-corrupting your process. Protected processes are perfectly capable of self-corrupting their own memory. In fact, most corruption is self-corruption. Most corruption errors result from writing to a location "near" other memory (off the end of an array, through an old pointer to memory that is now free, and so on). It's highly probable that the bad memory location is still in the writable address space of the process, and corruption will result. The only way to ensure no corruption occurs is by language support or testing.

## Optimize through understanding

Real time is more about reliability than speed. That said, efficient code is critical for many embedded systems. Knowing how to make your code zing is a fundamental skill that every embedded programmer must master.

The first rule is to trim the fattest hog first. So, given an application with a 20,000-line module and a 20-line function, where do you start optimizing?

Ha! That's a trick question. The 20-line function could be called thousands of times or spin forever on one line. The point: making the code run fast is the easy part. The hard part is knowing which code to make run fast.

An example may illustrate this. Loral was building a robotic controller to investigate possible space operations. It was a complex system combining many devices, complex equations, network interfaces, and operator interfaces.

One day, the system stopped working. The CPU was 100% busy. The control loop that usually ran 300 times/second barely made 40. Everyone claimed that "nothing has changed."

With the help of a simple profiler the problem was tracked to a matrix-transpose subroutine that was eating 80% of the CPU's time. A seemingly trivial (and forgotten) change in a controller equation caused the transpose code to run during each loop. The transpose routine allocated and freed temporary memory; since memory allocation is slow, performance suffered.

Optimizing every line of code, replacing all the hardware, switching compilers, and staring at the system forever would never have found this problem. This is typical; when it comes to performance optimization, the majority of the time a small change in the right place makes all the difference. All

the coding tricks on the planet don't matter if you don't know where to look.

Performance problems can even masquerade as other problems. If the CPU doesn't respond to external events, or queues overflow, or packets are dropped, or hardware isn't serviced, your application may fail. And you may never suspect a performance problem.

Fortunately, performance profiling is simple and powerful. It will also reveal things you never expected, giving you better overall understanding. How many times is a data structure copied? How many times are we accessing the disk? Does that call involve a network transaction? Have we correctly assigned task priorities? Did we remember to turn off the debugging code?

Profiling real-time systems presents a unique challenge. You need a profiler when you're running out of CPU. But most profilers use a lot of the CPU themselves. You can't slow down the system and still get an accurate picture of what's going on. The moral is, be sure you understand the Heisenberg effect of your profiler: every measurement changes the system.

We'll close this section with another story. ArrayComm was building a wireless-packet-data base station and user terminal. The product was nearly ready to ship, but it couldn't handle the high-load stress testing. Optimizing didn't help. Timing critical code sections didn't help. Intense management oversight didn't help, either. Running out of options, the developers were considering a redesign with a faster processor—a decision that would costs months of time and millions of dollars.

In desperation, they profiled the system. In a matter of minutes they found that more than one-third of the cycles were going to a function buried in a third-party protocol stack. Turning off an optional attribute-checking feature that had been turned on for debugging nearly doubled system throughput. The product shipped two weeks later.

Know how your CPU is executing your code. It's the only path to efficiency.

## Don't put needles in your haystack

Finding a needle in the haystack is a good metaphor for much of debugging. So how do you find needles? Start by not dropping them in the haystack. All developers know that nagging feeling that they're cutting a corner and creating a bug for the future. Stop and listen to your inner voice! Follow your good coding and design guidelines, check your assumptions, rethink your algorithms. If nothing else, put an easily found tag in a comment that this code is suspect.

There are many good sources of advice on how to design embedded programs. Dave Stewart's paper, "Twenty-Five Most Common Mistakes with Real-Time Software Development" found at www.embedded-zone.com/esc/mistake.pdf is a great start.

The next step is to seek needles before they poke you. For instance, tracing is great when you have an obscure problem. But its real power may be as a preventative. Regular scans for operating system calls that return errors can uncover many problems early in the development and test cycle. Think of it as a regular search for buggy needles in the haystack of your system.

How to do this? Use your trace tool to scan for error returns from all operating-system function calls and common application programming interfaces. Investigate any surprises. Do any calls to **malloc** fail? Are calls timing out unexpectedly? Check parameter values too. Are any pointer parameters into key functions null? Are you calling anything much more, or much less frequently than you expected? Tracing will give you that critical "feel" for the code. Doing it regularly, before needles become problems, will keep your hay safe.

## Reproduce and isolate the problem

If you do end up with a needle, divide the haystack. The first step in debugging a complex team-developed application is often a process of isolating the problem. The trick is to do this effectively.

If the problem is an intermittent glitch, then the critical first step is to reliably duplicate the problem: recreate it and defeat it. Get a sequence, any sequence, that reliably shows the problem and you're halfway there. While you're testing, write everything down. Keep logs of anything that changes. An organized, meticulous approach is critical to isolating the problem.

Once you have a way to duplicate the problem reliably, you can start dividing the haystack. Turn features off, **#ifdef** out whole blocks, divide the system into the smallest piece that still exhibits the problem.

With tough problems, break out the toolbox. If you suspect a damaged variable, trigger your ICE to stop the system when it changes. Track it with a data monitor. Use a tracing tool to get a detailed history of function calls, including the passed parameters. Check for memory corruption. Check for stack overflow.

Let's illustrate with an example. During development, Candera's developers noticed a strange problem that made the system clock jump ahead by a huge number. With an ICE, they traced the problem to a bad parameter passed to an operating system call. But this wasn't enough information to fix the bug. The operating-system call wasn't to blame; someone was passing it a bad parameter. Of course, that raises a needle-in-the-haystack problem: there were a lot of lines of code, and the particular operating-system call was heavily used. In this case, a trace tool was used to record all the invocations of the OS call, quickly identifying the offending code.

## Know where you've been

Hansel and Gretel were smart to drop a trail of bread crumbs. A backwards-traceable record is a great way to make sure you understand future problems. Hansel and Gretel's only real mistake (besides trusting the witch) was not using a more permanent means of recording their progress. Don't make their same mistake.

When you get your application or module working in any significant capacity, checkpoint it. Later, when it stops working, even though "nothing has changed," you will have a baseline to check your assumptions.

How do you do this? Use a source control system; when the code works, check in a tagged version so you can **diff** with future states. Run benchmarks to document the level of performance you should expect from various modules. Take traces of operating system behavior, key variables, and memory usage. Check those in with the working code. These maps of the past will greatly guide you when you're lost in the future. This habit alone will save you half your debugging time. The next time your application suddenly takes up 50% more CPU performance or memory, it will be much easier to isolate and identify the problem by looking at the characteristics of the previous version.

## Make sure your tests are complete

How thorough are your tests and how do you know that? Coverage testing can answer both questions; it shows you exactly what code has been executed. Coverage tools verify the completeness of your test suite.

Experienced coverage testers will tell you that most test suites exercise only 20 to 40% of the total code. The remainder of the code can include bugs waiting to happen.

Coverage testing should be part of every quality assurance process. How many revisions and rewrites has your code gone through over the years and releases? Has your test suite grown with

the changes? Or do the tests only exercise the features that existed in version 1.0? Coverage testing may keep your test group busy, but a solid coverage report will make your product manager sleep well.

The many different types of coverage testing range from just seeing if every function has been entered to checking that every possible way to get through every possible path has been tested. Most tools support various levels.

The level of testing required depends on the application. Are you writing code that controls an airplane's in-seat video or its wing flaps? For the video system, verifying that most statements in the program have been executed may suffice. For the wing flaps, you want to know that when a decision evaluates to false, the error-handling code is not executing for the very first time.

Coverage testing will also point out dead code. Most programs contain 20% or more code that is never executed. Have some of the features been dropped in the latest release? Is overall code size a concern for you? If so, take a look at your dead code.

Even if you're confident that your tests are complete and that you have no dead code, coverage testing can reveal latent bugs. Consider the following code snippet:

**if (i >= 0 && (almostAlwaysZero == 0 || (last = i)))**

This code has a bug; if **almostAlwaysZero** is ever nonzero, it will set "last" to whatever *i* is, probably not what you want. A condition coverage test will show that this last part of the decision never executed. By the way, function, statement, and decision coverage won't find the bug; they just ensure that both branches of the **if** statement executed.

## Pursue quality to save time

Studies show that 74% of statistics are made up. More seriously, our experience indicates that over 80% of development time is spent:

- Debugging your own code
- Debugging the integration of your code with other code
- Debugging the overall system

Worse, it costs 10 to 200 times more to fix a bug at the end of the cycle than at the beginning. The cost of a small bug that makes it to the field can be astronomical. Even if the bug doesn't have a significant impact on performance, it can significantly affect perceived quality. Just ask the developers of the original Pentium's floating-point microcode. Several years ago, Intel shipped millions of chips with an insignificant flaw. Even though the flaw affected almost no applications, it was a public-relations disaster for Intel, caused its first chip recall, and forced an eventual charge against company earnings of $475 million.

You can't achieve quality by saving testing until the end of development. A quality process, such as testing as you go, reduces the critical "what's going on" time and greatly speeds development. Get into the habit of testing and scanning for problems every day. You'll ship better code faster.

## See, understand, make it work

By definition, real-time systems interact with a dynamic world. Unfortunately, traditional debuggers can't reveal dynamic behavior; debuggers deal only with static values of stopped programs. Questions like:

- How noisy is my sensor?

- How fast is the queue growing?
- When did the valve close?

simply cannot be answered by any tool that stops the execution. These questions deal directly with the dynamic behavior of the real-world system. They can only be answered by a dynamic tool that analyzes your program while it's running.

Think of your embedded program as a car. A source-code debugger is like a mechanic's garage. You can bring your car in for service; the mechanic will stop the car, put it up on a jack, and see what each part looks like. It's a valuable service and fixes many problems. However, this static test cannot deal with dynamic, real-world issues like, "Why does the steering wheel shake?" While the garage may be able to take apart the entire steering system and discover a worn part, the garage will never find many causes.

If the answer is, "Because you are driving off-road," the garage won't find that. Other questions, such as, "How fast am I going?" and "How close did I come to hitting that truck?" are not answerable in the mechanic's shop. Stopping the car changes the nature of the system.

Some questions can only be asked when the car is moving and only be answered by being able to see how the car behaves on the road.

The same is true of most embedded systems. Many critical real-time systems cannot be stopped at all. For example, stopping a network system to discover how long a work queue is getting may cause it to immediately overflow. Placing a breakpoint in the control software for a moving robotic arm could be dangerous. Other operations take place over time. For instance, a wafer stepper executes many steps of a process in order. You may need to watch the behavior of the system during the entire sequence to evaluate its operation.

Real-time monitors answer the dynamic performance questions. They don't stop the system. They collect live data and display them in real time. There are several types for different types of problems; the most common are operating-system monitors, data monitors, and profilers.

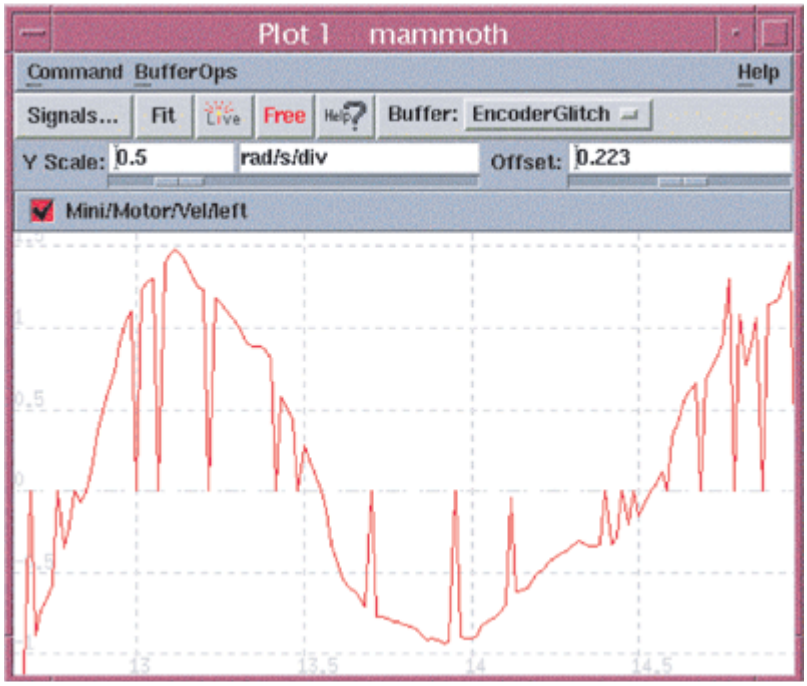


**Figure 1: Data from a motor sendor shows random zero readings**

An example may clarify this. Figure 1 shows data taken from a motor sensor. The motor was part of a system that was vibrating badly; it wasn't clear if the vibration was due to a poor control design, a bad mechanical component, or a problem with any of many sensors and motors in the system.

The data monitor immediately narrowed the problem to a velocity sensor on one motor. Further, it showed that the error was a sporadic false reading of zero.

The sensor was an optical encoder, which didn't have a direct velocity output. Instead, its position-output signal was used to derive the velocity of the motor from a high-frequency counter that counted pulses from the encoder. Software read this counter and computed the velocity signal in Figure 1.

The monitor immediately showed that the position signal from the optical encoder was glitch-free. Thus, the encoder itself was unlikely to be at fault. By monitoring all of the variables involved in the intermediate calculations, from the hardware registers to the final velocity signal, it was easy to determine that the driver software was also not to blame. However, the hardware registers showed occasional bad readings. Therefore the problem had to be in the encoder card electronics.

This is exactly the type of problem that's nearly impossible to find with a traditional debugger. It would take many breakpoints in just the right places (and a fair bit of luck) to detect that occasional zero reading. However, using a data monitor, it took only a few minutes to identify the symptom and pinpoint the problem. Without an online monitor, it could have (and probably would have) taken several days to identify the bad electronics.

This is an example of tracking down an unknown problem. Often, the real value is in just understanding what's going on at a deeper level—you'll be a better driver if you look out the window and make sure you're still on the road before the wheel begins to shake. Monitor your system on a regular basis, just to learn it. The bottom line: don't be satisfied that your application works; understand exactly how it works.

## Harness the beginner's mind

*"In the beginner's mind there are many possibilities; in the expert's mind there are few."--Shunryu Suzuki*

Most debugging is the process of learning more and more about your application. Marathon debugging sessions that examine every detail are sometimes the only way to gather sufficient information and enough focus to identify and tackle a bug.

But there is a tradeoff. During the focused marathon, your mind closes to possibilities. Assumptions become unquestioned facts. You begin to know things that just aren't true. You don't consider possibilities that you rejected for unsound reasons.

The "beginner's mind" is a Zen concept of emptying your mind so it can be open to new solutions. The power of the beginner's mind should not be underestimated. A fresh pair of eyes can work wonders. Even a truly empty mind, such as a manager's, can help.

So when you're really stuck, when you've looked at the problem in every way you can, when every tool in your toolbox has failed you, take time off to think. Go ride a bike. Take the afternoon off and come back in the evening to give yourself a new perspective. Explain the problem to a friend, a manager, a dog, or the wall. The external challenge to your assumptions is often the catalyst that breaks the barrier down.

## Achieve mastery

Debugging is certainly an art. But like most art, success is a combination of talent, experience, and mastery of the tools of the trade.

The secrets of debugging are not mysterious. Know your tools. Avoid the common, dangerous mistakes if you can, then check to make sure you've succeeded. Be proactive. Save data to help you compare the past with the present, even though "nothing has changed." Keep your eyes and your mind open. And accept the responsibility of excellence. You should ship no system without:

- Testing for memory leaks and corruption
- Scanning for performance bottlenecks
- Collecting a variable trace of key sequences
- Ensuring sufficient CPU-bandwidth margin
- Evaluating test coverage effectiveness
- Tracing operating-system resource usage
- Recording execution sequences (semaphores, process interactions) of key activities
- Checking for error returns from common application programming interfaces

If you learn only one thing from The Oracle, learn to integrate strategies to better understand your system into your development routine. It's hard to find a developer who thinks she has wasted time using a tool, testing for quality, or taking the time to understand what the system is doing. The opposite happens every day. Understand that, and The Oracle will keep you out of trouble. esp

**Lori Fraleigh** is director of Real-Time Innovations' ScopeTools product group where she's responsible for visualization and analysis tools for VxWorks and Linux. Lori has extensive experience designing and developing embedded debugging tools that focus on improving system performance and reliability with little overhead. She holds an MSEE from Stanford and BSCEE from Purdue and can be reached at lori@rti.com.

**Stan Schneider** is CEO of Real-Time Innovations, which he founded in 1991. He specializes in real-time software systems and architectures and has written many tools, including StethoScope data monitor, the ProfileScope execution profiler, and the original memory leak and corruption detection technology underlying MemScope. He has managed Stanford's Aerospace Robotics Laboratory and has extensive experience as a technical and management consultant in real-time digital signal processing and computer systems in many industries. Stan has a PhD in electrical engineering and computer science from Stanford University and a BS in applied mathematics. You may reach him at stan@rti.com.