



The magazine of record for the embedded computing industry

TimeMachines: The Future of Debuggers

There are a wide variety of problems in software development that most debuggers don't allow you to easily fix. Real-time trace and debuggers that run backward as well as forward can help ease this pain and lead to a much more efficient debugging environment.

by Michael Lindahl
Green Hills Software

Getting a product to market earlier can mean the difference between wild success and complete failure for a device software project. Yet many projects are delayed for software quality reasons, and even if they are not delayed, the schedules are often padded to allow for unexpected bumps and bugs in the road to shipping the software.

What is the number one factor that delays software projects and prevents them from shipping earlier? Once the software is written and integration and testing have begun, only one thing prevents the product from shipping—bugs. At this point, the vast majority of a software development team's time is spent debugging software problems, and shipping delays are almost always due to unfixed bugs or glitches. Sometimes these problems are bugs that reproduce easily and can be fixed quickly. However, the bugs that usually take up the

majority of this time are difficult to reproduce or only occur under certain circumstances. Sometimes the bugs only occur on the tester's machine; sometimes the bugs can't be reproduced in the lab—but these bugs must be fixed before a reliable product can be shipped.

Whereas simple bugs can be fixed by running them in the debugger, stepping through the code and understanding the flow, these hard to reproduce bugs take the majority of debugging time and are very difficult to fix because the bug cannot be easily reproduced with the debugger attached. This leads to a very inefficient, trial-and-error process where engineers speculate about the cause of bugs, study the code and generally guess at where the problem may lie. They may add print statements or other manual instrumentation in an attempt to corner the bug, but this process is often a matter of guesswork more than anything else. And as device software continues to grow in complexity, the amount of time that developers have to spend debugging their software will only continue to grow.

Today, innovations in debugging technology are working in the software engineer's favor to help bring this problem of shipping reliable software in a reasonable amount of time under control by making these bugs easier to trap and eliminate. Debuggers that allow you to stop when a bug occurs and then step and run backward to the source of the bug, are becoming a reality and can help eliminate these nasty bugs. This allows you to capture a bug, and once it is captured, to replay the sequence of events that led up to the bug, essentially allowing you to turn a very difficult-to-reproduce bug into a simple, easy-to-reproduce bug. The technology that is enabling these debuggers is real-time trace, and it is enabling debugging that can make software projects more predictable and manageable.

Real-Time Trace

Real-time trace refers to special debug ports built into many embedded microprocessors, which provide detailed in-



Get Connected
with companies mentioned in this article.
www.rtcmagazine.com/getconnected



Figure 1 A trace probe connected to an ARM 9 target.

formation about what instructions the processor executes as well as the sequence of memory loads and stores caused by these instructions. Processors equipped with this special debug port output this information as the processor runs at full speed, enabling an external device—called a trace probe—to collect this information non-intrusively as the system runs (Figure 1). Some examples of microprocessors that are equipped with a real-time trace port include many ARM7, ARM9 and ARM11 processors, which include the Embedded Trace Macrocell (ETM) from ARM, as well as the AMCC PowerPC 4xx family, the PowerPC 54xx family and the ColdFire family of processors from Freescale.

The trace probe collects this information using a circular buffer that enables you to collect detailed information about the execution of your software over relatively long periods of time. The circular buffer means that the trace probe is always storing the most recently executed instructions, so when your software encounters a bug, you can stop the trace collection and store the sequence of instructions and memory accesses that led up to that bug (Figure 2). The trace probe's collection of trace data can either be stopped manually, when the user presses a button—either on the probe or on a host machine—or by using flexible triggers that allow you to configure the location of a bug, so that trace can be retrieved without altering the system behavior in any way. For instance, you can trigger when a specific line of code is hit or when a variable is written to with a specific value.

Once you have captured the sequence of instructions that precede a software glitch, you have essentially captured the bug and you can analyze the data to find the root cause of the problem. Trace analysis software can be used to reconstruct the state of the microprocessor for every point in time that you have cap-

tured trace data using a standard instruction-set simulator. Thus you can now know the sequence of instructions and the value of most register and memory values that led up to the bug.

Debugging Forward and Backward

Once the trace analysis tools know the sequence of instructions and the memory values, this information can be used as the back-end of a standard software debugging session. Essentially, the debugger simply replays the instructions from the trace data rather than talking to a live target. The debugger translates the raw instructions into source code lines and the raw memory values into variable values. This enables you to easily browse the state of the system at any point back in time using a familiar interface. However, you can also use the debugger to fully replay the system along the same path that led to the bug you have captured. The debugger lets you step forward from one instruction to the next, and all of the state variables will update to display the

new values. In addition, the debugger allows you to run forward to a breakpoint by searching forward through the instructions for any address where you have set a breakpoint. You can even set watchpoints and data breakpoints, and again, the debugger will search forward for the next access to a memory location when you run the program.

The real power of such a tool, however, rests in its ability to run *backward* as well as forward. This is accomplished using the same techniques described above, except instead of stepping forward, the tool simply searches backward for instances in the trace data where breakpoints or watch points are hit.

Stepping and running backward enables you to quickly step or run backward to the source of the bug after encountering an error condition. You can investigate various possible solutions and find exactly what conditions led up to the error. Because you have stored the trace data, you can easily replay your software both forward and backward in time to understand its behavior and track down any type of bug (Figure 3).

This enables you to quickly track down the types of bugs that are otherwise extraordinarily difficult to reproduce using traditional debugging techniques, because traditional techniques require you to reproduce the error under the debugger. Some bugs may take several minutes to reproduce, simply because you have to restart the system and go through a complicated procedure to reach the buggy code. Other bugs don't even reproduce reliably and you have to try a certain sequence many times before the bug reproduces. And then, if you step past the bug in the debugger, you have to start all over again. But with stored trace data, you can eliminate all of these problems, because you know exactly what code paths were executed and what caused the glitch.

If you can capture a bug in the trace data once, then you can spend as much time as necessary tracking down the problem and fixing it. You can even attach a trace probe to testing and verification units in the quality department or to the first units shipped to help capture detailed information about glitches that occur rather than having to rely on bug reports and reproducing the problem in the lab. Capturing difficult-to-reproduce bugs during testing can eliminate weeks or months from the schedule of device software projects because so much time is often spent tracking down these types of bugs, especially as the software gets closer and closer to being released.

Aiming at Multicore

As embedded systems continue to grow in complexity, more and more systems will employ multicore designs. This may involve many identical cores running at the same clock frequency or different processor architectures running at vastly different speeds. Regardless of the specific multicore design, the problem of synchronization and communication between various processing elements remains daunting for software designers. A small timing delay at any point can potentially cause drastically different results if the software has a minor glitch. Tracking down these types of problems with traditional tools is very difficult, because reliably reproducing these bugs may be nearly impossible.

However, if you could collect synchronized real-time trace information from each of these processing elements, you would be able to replay the sequence of instructions on each processor to determine the exact interactions between the various cores and their software. Thus, if a timing error causes a failure in the system, you could find the bug in the software that did not properly handle the specific timing condition. Alternatively, if a shared resource, such as shared memory or a shared device, is not properly locked, you could find the offending code quickly and easily rather than having to rely on trial-and-error debugging techniques to find the problem.

In addition, system peripheral and coprocessor registers often heavily influence device software implementations, so adding trace support for these devices to enable software-level access and understanding of these devices over time would greatly simplify the process of tracking down various device-level problems caused by errors in the software.

Technology for solving both of the above sets of problems is beginning to emerge. ARM is moving toward multicore and multi-device trace with their new CoreSight technology. This technology includes modules to multiplex data from various on-chip sources into a single, synchronized trace stream that can be captured and replayed. In addition, other processor companies are embracing real-time trace technology to make the difficult job of releasing high-quality software in a predictable amount of time easier.

While today's software is much larger and more complicated than five or ten years ago, most engineers still rely primarily on debugging technology that does not solve some of the hardest problems facing software engineers today. Traditional debuggers do not provide enough visibility into the inner workings of your software to maximize your productivity when tracking down

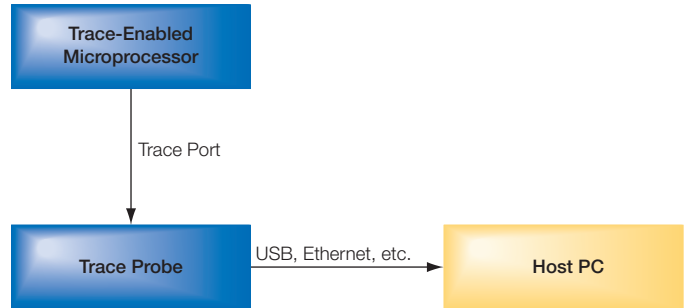


Figure 2 A block diagram of a trace probe collecting trace data from an ARM 9 system. The trace data is transferred from the trace port on the microprocessor and stored in the circular buffer in the trace probe. It is then transferred to the PC where the debugger can examine it in both backward and forward execution.

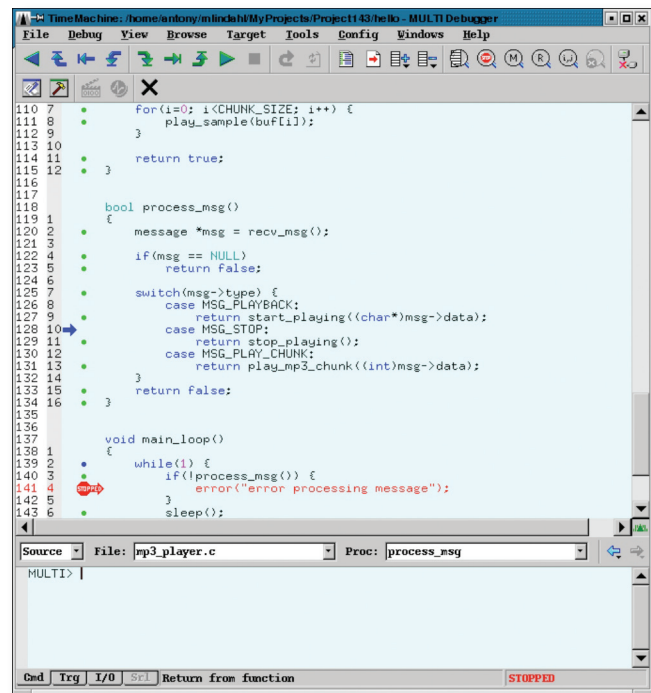


Figure 3 A screen shot of the TimeMachine debugger with both forward and backward run-control buttons enabled. You can also see the state of variables, registers and other processor information at any point in time through the TimeMachine debugger.

and debugging software errors. However, leveraging real-time trace technology and the most advanced software debuggers can greatly increase programmer productivity, helping eliminate some of the biggest schedule delays in software development. ■

Green Hills Software
 Santa Barbara, CA.
 (805) 965-6044.
www.ghs.com.

