

Tracing Your Way to Better Embedded Software

Techniques to isolate and debug critical software issues.

by Jay Gould
Product Manager
Xilinx, Inc.
jay.gould@xilinx.com

Steve Veneman
Product Manager
Wind River Systems
steven.veneman@windriver.com

Have you ever worked on an embedded processing project where your team invested an exorbitant amount of test time only to find out at the end of the project that your product had a series of intermittent and hard-to-find bugs? You used your software debugger throughout the development process and spent man-weeks stepping through the code in your lab, yet when your software passed integration testing you were stunned to find that your QA team (or worse yet, a customer) discovered serious system flaws.

Some embedded bugs are harder to find than others and require advanced detection methods. Testing small units of code in

your office or lab does not fully exercise your embedded product the same way as when it is fully assembled and deployed in real-time conditions. Add other modules from other engineers and your simple unit tests may still pass for hours at a time. Often bugs will not reveal themselves until you run the code for much longer periods of time or with more complex interactions with other code modules.

So how do you know if you are really testing all your code? Is it possible that a colleague's software may be overwriting a variable or memory address utilized in your module?

The more complex your embedded products become, the more sophisticated your development and debugging tools must scale. Often "test time" is equated with "good testing" practices: the longer you run tests, the better you must be exercising the code. But this is often misleading for a number of reasons. Stubbing code and testing a module in a stand-alone method will miss many of the interaction mistakes of the rest of the final

system. Running dozens or hundreds of "use cases" may seem powerful, but may create a false sense of security if you don't actually track metrics like code "coverage," where you pay more attention to exercising all of the code instead of repeatedly testing a smaller subset.

Standard software debuggers are a useful and much-needed tool in the embedded development process, especially in the early stages of starting, stepping, and stopping your way through brand-new code units. Once you have significant code applications developed, testing the system as a whole becomes much more representative of an end-user product, with all of the interactions of a real-time embedded system. However, even a good software debugger may not improve your chances of finding an intermittent, seldom-occurring bug. Serious code interaction – made more complex with asynchronous interrupts of real-world embedded devices – mandates the use of proper tools to observe the entire system, creating a more robust validation environment in which to find the sneakier bugs.

Trace Your Code Execution

Once your code goes awry, locks up your system, core dumps, or does something seriously unexpected, it is time to improve the visibility of exactly what your software was doing before the flaw. Often the symptom of the failure is not clearly tied to the cause at all. Some debuggers or target tools running on the embedded device will die with a fatal system flaw, so they cannot be used to track that flaw. In these cases, you can use a separate and external tool with its own connections and hardware memory to provide an excellent view of system execution all the way up to a complete failure. “Trace” tools, often associated with JTAG probes or ICEs (in-circuit emulators), can display an abundance of useful information for looking at software instructions executed, lines of code executed, and even performance profiling.

Capturing software execution with a trace probe allows more debugging capability than just examining a section of code. An intelligent trace tool provides a wide variety of trace and event filters, as well as trigger and capture conditions. One of the most useful situations for trace techniques is to identify defects that cause target crashes on a random, unpredictable basis.

Connecting a probe (Figure 1) like Wind River ICE with the integrated Wind River Trace tool introduces a method by which you can configure trace to run until the target crashes. By triggering on “no trigger,” the trace will continue to capture in a circular buffer of memory in the probe until the system crashes. You may have unsuccessfully tried to manually find a randomly occurring defect with your software debugger, but with a smart trace tool, you can now run the full battery of system tests and walk away to let the trace run. When that intermittent event finally occurs again, crashing the target, the tool will capture and upload the data for your examination.

The execution trace will show a deep history of software instructions executed on your system leading up to the fatal flaw, providing much more insight into what the system was doing and allowing you to find and fix the bug faster. Often these prob-

lems are not easily found by post-mortem debugging through register and memory accesses. Embedded systems commonly crash when a function is overwriting a variable that should not be accessed/written, or when memory is corrupted by executable code being overwritten by another routine. It is difficult to diagnose these types of problems when no information exists on what actions led up to an event, and the crash of the system may not actually take place until much later when the corrupted

information is next accessed.

With advanced tools like Wind River Trace, you can capture the actual instruction and data information that was executed on the target before the event occurred. Using this information, you can back up through the executed code and identify what events happened before the crash, identifying the root cause of the problem (see Figure 2). In addition to a chronologically accurate instruction sequence, when used on a PowerPC™ embedded platform like Xilinx® Virtex™-II Pro or Virtex-4 devices, the Wind River Trace execution information also includes specific time-stamp metrics. Coincidentally, the Wind River ICE actually implements Xilinx Virtex-II Pro FPGA devices in the probe hardware itself.

Coverage Analysis and Performance Profiling

Because the trace tool and probe are able to provide a history of the software execution, they can tell you exactly where you have been – and what code you have missed. You miss 100% of the bugs that you don’t look for, so collecting coverage analysis



Figure 1 – The Wind River ICE JTAG probe implements Virtex FPGAs.

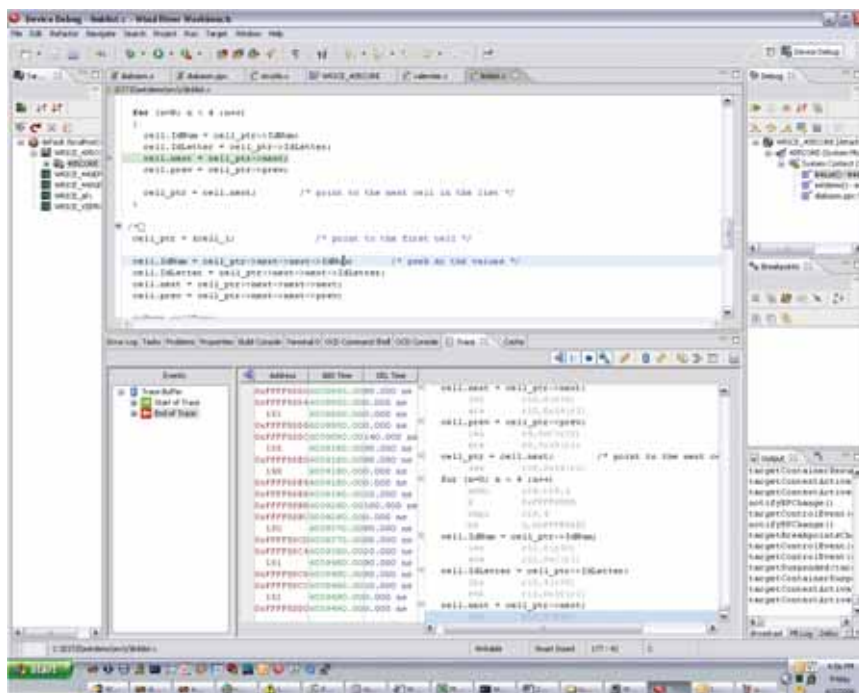


Figure 2 – Wind River Trace software execution view

metrics is important to your testing strategy. As code execution is almost invisible without an accurate trace tool, it is common for entire blocks or modules of code to go unexecuted during test routines or redundant user-case suites. Testing the same function 10 times over may make sense in many cases, but missing a different

space, higher than normal software testing standards may be mandated. Certainly the software running the ABS brakes in your car, a medical device, or an aircraft control system should be tested to a higher degree than a software game or other non-life-threatening application. In these life-critical applications code must be executed and

built into probes and trace tools allow you to look at the data in a performance view in addition to the go/no-go code coverage view. By identifying bottlenecks in software execution, you can identify routines that fail to meet critical timing specifications or optimize routines for better performance. With the time-stamp information in Virtex-II Pro and Virtex-4 devices, you can also use trace tools to determine how long a software function or a particular section of code took to run.

Whether you are trying to find routines that could use some optimization recoding (perhaps in-lining code blocks rather than accruing overhead by repeatedly calling a separate function) or identifying a timing flaw in a time-critical routine, performance-metric tools are required to collect multiple measurements. Settling for a couple passes of a routine with a manual debugger is not sufficient for validating timing specs on important code blocks. Additionally, if a real-time system is running an embedded operating system, verifying the capability of the interrupt service routines (ISRs) is essential to determine how quickly the system responds to interrupts and how much time it spends handling them.

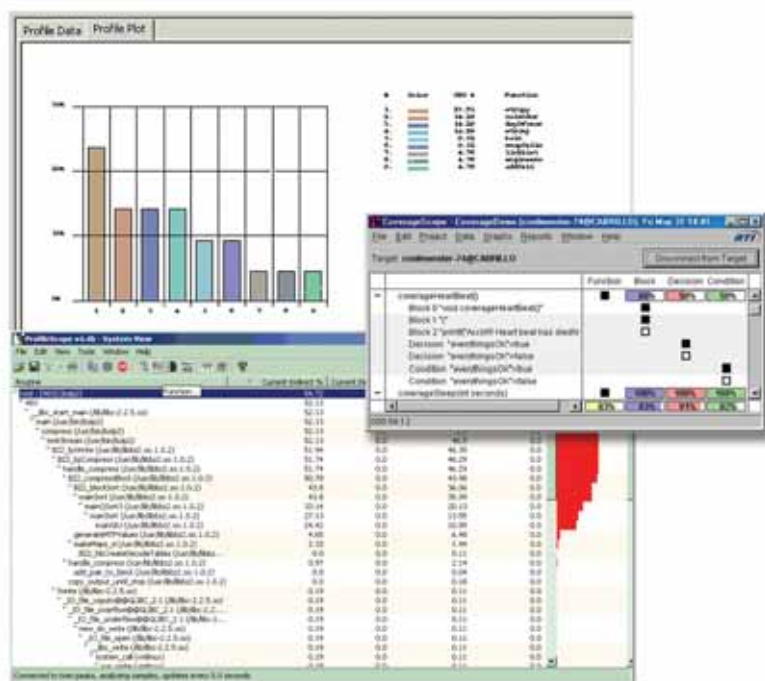


Figure 3 – Wind River profiling and coverage views

function altogether means that you will not find those bugs in your lab – your customer will find them in the field.

Coverage metrics showing which functions are not executed are useful for writing new, additional tests or identifying unused “dead” code, perhaps leftover from a previous legacy project. Because many designs are re-ports or updates of previous products, many old software routines become unnecessary in a new design, but the legacy code may be overlooked and not removed. In applications where code size is critical, removing dead code reduces both waste as well as risk; nobody wants to wander into an obsolete software routine that may not even interact properly with a newer version of hardware.

In safety-critical applications like avionics, medical, automotive, or defense/aero-

tested (covered) down into every function of software instructions, and that execution may need to be documented to a formal, higher level governing body.

In many cases code coverage can also be used to analyze errant behavior and the unexpected execution of specific branches or paths. Erratic performance, failure to complete a task, or not executing a routine in a repeatable fashion may be just as inappropriate in some applications as a fatal crash or other defect. By tracing function entries and exits, comprehensive data is collected on software operations and you can throw away the highlighter and program listing (Figure 3).

Because performance is always a concern in modern embedded applications, having an accurate way to measure execution times is paramount. The capabilities

Conclusion

If critical performance measurements or code coverage are required before your embedded product can ship, then you need to use specialized tools to capture and log appropriate metrics. Accurate system trace metrics are a must if you have ever been challenged by a seemingly random, hard-to-reproduce bug that your customers seem to experience regularly but that you cannot duplicate in your office.

Utilizing an accurate hardware probe like Wind River Trace with a powerful trace/coverage/performance tool suite will provide you with the data you require. Stop guessing about what is really going on and gain new visibility into your embedded applications.

For more information about Wind River Trace and other products, visit www.windriver.com/products/development_suite/wind_river_trace/.