White Paper

The New Debugging

Robert Redfield
Green Hills Software
Overview

Are you growing weary of claims of “next generation”, “revolutionary” and “innovative” from embedded software tools providers? Are you discovering these to be nothing but marketing fluff and re-wrapped features? Need a dose of true innovation that solves real software development challenges you face while creating software for your i.MX-based products?

Freescale Semiconductor’s i.MX27 and i.MX31 applications processors enable a new era of software development tools. It’s a powerful marriage of silicon and software technology that produces hardware trace-assisted debugging and profiling. It’s The New Debugging.

Contents

1 The Old Debugging 1
2 Introducing “The New Debugging” 1
   2.1 You Can’t Fix (Or Improve) What You Can’t See 2
   2.2 I've Mastered My Debugger and Profiler, Don't Make Me Learn Something New 2
   2.3 Do No Harm 6
   2.4 A Thousand Deaths No More 7
3 Components of New Debugging 7
1 The Old Debugging

Before we cover New Debugging, take a look at old debugging. It’s what we do today.

- Day 1: Your colleagues in the test group claim that your work is “acting strangely” and “they didn’t change anything.” You stop working on some optimization improvements (your manager still wants ten percent speed improvement) to investigate their claim. You spend time watching the test behavior, check that the test folks have your latest work and double-check that they are using the same hardware prototype that you used. It’s different, but a couple calls around the company confirm that it shouldn’t be a problem. Reluctantly, you spend a day in the test room to determine if the problem is your code or code from other newly integrated components. You spend the last couple hours of the day researching what these other new components do, at a high level at least, by looking at the code and calling the authors. The other code, written by your global counterparts, is not available in source form today. You must wait until tomorrow for their help and hopefully, source code.

- Day 2: The system works right the first time. Then it fails. Then it just hangs. So, your trusted debugger comes out. You recompile the application with the debug switch turned on. In the debugger you step through the source code while the program runs on the target. You perform this repeatedly focusing where you see something suspicious, but then you realize the abnormality is because your debugger intrudes and changes the task interaction in the RTOS.

- Day 3: Not only are you not able to locate an obvious problem, but other application functionality changes because the system now runs more slowly. You recompile again (with the debugging switch turned off) and carefully apply some printf's. Fortunately, your JTAG connection will support this, but this technique really interferes with the timing.

- Day 4: Next, you instrument the code to write very small (and arcane) data to a memory location that you can read with an external device, a logic analyzer for example. Program behavior is fairly consistent in that it now fails as it does for the test group. It’s difficult to interpret the many trace listings and you can’t add bigger data dumps. A colleague who is good with low-level debugging and logic analyzers might be able to assist, but you would need to get his manager’s approval.

- Day 5: You’ve been involved with these kinds of issues before and you know that there will be false hopes, more people involved and nervous management breathing down your neck. The problem could be bad code in an interrupt service routine, a time or heat-sensitive hardware error, or memory corruption between software tasks. If only you could simply stay in your debugger and profiler without slowing down the processor, without setting up an elaborate test structure, without recompilation and without changing a line of code.

2 Introducing “The New Debugging”

Hardware trace is the first step. It brings non-intrusive visibility into the actual execution of your program through a raw history of what the microprocessor does as it runs. It may include a list of the instructions that were executed, the memory that was read or written as well as other information about the state of the processor. On Freescale’s i.MX31 and i.MX27 applications processors, the trace data is collected on-chip in the embedded trace macrocell (ETM) without slowing down or intruding on the ARM® core. But hardware trace data is only the first portion of the New Debugging. The second necessary ingredient is a new software technical innovation that automatically transforms hardware trace to existing tools already known and used by software developers.
2.1 You Can't Fix (Or Improve) What You Can't See

Studies show that 50 percent of product development time is spent in debugging. We debug code to either make it 1) work correctly, or 2) work better (optimize). In either scenario, the more visibility you have into exactly what the code is doing on the processor in real-time, the better.

Debugging code has evolved, of course. Our predecessors patched cables and watched blinking lights. Then came program punch cards, CRTs for looking at the mountains of assembly, single-line source level debugging, full program source level debugging with fancy windows, and finally multi-threaded multi-core debugging. However, today's debugging and analysis is essentially a repeated trial-and-error experiment observing veiled symptoms and side effects of your code's operation. In 2003, the next debugging step arrived: hardware trace-assisted development tools.

Table 1. Debugging and profiling methods available today

<table>
<thead>
<tr>
<th>Debugging/profiling technique</th>
<th>Visibility into code and processor operation</th>
<th>Changes program behavior?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run your application</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>Profiler</td>
<td>Fair to good but performance is averaged across iterations which hides spikes and valleys, often evidence of a bug</td>
<td>Some</td>
</tr>
<tr>
<td>Debugger</td>
<td>Excellent</td>
<td>Yes: halts processor or requires intrusive run-time monitor</td>
</tr>
<tr>
<td>Printfs</td>
<td>Limited amount of data; requires source code</td>
<td>Yes, and requires recompilation</td>
</tr>
<tr>
<td>Logic analyzer and software trace listings</td>
<td>Lots of raw data but not easy to understand with respect to your code</td>
<td>No</td>
</tr>
<tr>
<td>Hardware trace-assisted</td>
<td>Excellent</td>
<td>None</td>
</tr>
</tbody>
</table>

2.2 I've Mastered My Debugger and Profiler, Don't Make Me Learn Something New

Hardware trace has been available on some embedded microprocessors for nearly ten years and logic analyzers have been widely available even longer. Why haven't they been widely adopted by embedded software teams?

- Raw trace listings are difficult to use and understand with respect to the application
- Requires training and the mastering of another expertise
- Expensive hardware

Early trace tools did not translate the raw trace data output by the processor into the high-level software concepts that software engineers use to construct their programs. Rather, most tools simply display a list of the instructions that the processor executed and force the software engineer to manually sift through thousands or millions of instructions. Thus, most software engineers only take advantage of trace tools when they have exhausted all other debugging techniques.
Figure 1. A traditional trace listing shows thousands or millions of instructions that are difficult to use in the normal debugging cycle. Sophisticated triggers to filter the data are helpful, but require tedious trial-and-error.

On Freescale’s i.MX31 and i.MX27 applications processors, hardware trace data can be captured, transformed, and fed to several tools already familiar to software developers.

In the trace-assisted debugger, called TimeMachine, you simply step forward and backward through your code, set breakpoints (forward or backward in program execution), look at variables, registers, call stacks and memory locations in the same way you always do with a traditional debugger. This is one component of the New Debugging.

A traditional profiler profits greatly when trace-assisted because its biggest drawback, intrusion, is eliminated. Traditional profilers give you statistical information about how often each part of your program ran. This data is collected by either periodically sampling the program counter or by instrumenting the object code. An array of the program counters is incremented for each program location as it is encountered. This raw information is then correlated to the time spent in each task, function, and source code line in your program.

However, this information comes with costs that often disqualify it as an everyday debugging tool:

- A traditional profiler halts your system periodically to sample the program counter or modifies your executable, thereby changing the behavior of your program.
- Storing the profiling information on embedded systems is often a challenge because either additional memory or a way to transmit this information as the system runs is required.
- Your code size may be increased, potentially making it too large to fit into the limited memory available on many embedded systems.
- Samples are averaged, veiling the actual code performance.
Here’s an example showing how statistical sampling veils the view of actual execution from you. If a function executes 100 times, taking 100 cycles for 99 times it’s executed, and then taking 10,100 cycles the final time, the traditional profiler will say that the function takes an average of 20,000/100 = 200 cycles per call. The one unusual run that took 10,100 cycles is veiled and chances are good that your bug is associated with that one unusual run.

In contrast, trace-assisted profiling removes these drawbacks because the data has these characteristics:

- Collected in real-time
- Does not slow down the processor
- Does not require instrumentation
- Samples are not averaged

As we’ve seen, you gain unprecedented visibility into your code through a trace-assisted debugger and profiler. However, the i.MX trace data now enables two more powerful views of execution.

PathAnalyzer shows a graphical view of your program’s call stack over time and over number of instructions. Because the data is graphically represented and time stamped, anomalies often arise in unexpected areas of your code. For example, a developer uses iterative calls to perform a routine quicksort_r and observes that the application suspiciously runs much faster than expected. Why? She scans the PathAnalyzer graph, shown in Figure 4. It shows her that the sorting took 6,200 instructions. She then sees something odd in the quicksort region. Each inverted orange peak (red X) marks a sorted item in the list, and she sees far fewer of these than expected. That’s her clue. She clicks one of the peaks, which puts her in the debugger at that exact line of code and execution point. She views variables and memory, steps backward, views them again and solves the problem.
As with the trace-assisted debugger and profiler, there is no processor intrusion, recompilation or instrumentation necessary.

**Figure 4.** The PathAnalyzer shows the function stack (y-axis) versus time. The graphical representation often reveals software flaws not visible in a debugger or profiler.

Another new trace-assisted view into your application shows RTOS tasks, kernel calls, context switches, semaphores and more, versus time. It’s called EventAnalyzer. As with PathAnalyzer there is no processor intrusion, recompilation or instrumentation and you can go forward and backward in program execution. EventAnalyzer is especially powerful for multicore applications running multiple tasks.
2.3 Do No Harm

Software trace and code instrumentation that produce software trace logs have been around a long while. This type of instrumentation is effective for certain applications but hardware trace has several advantages and no downside, especially for embedded systems.

First, software trace requires you to instrument your code. If you have an existing code base that does not have trace instrumentation, it will take days of trial-and-error to find the right locations, the right values and right volume of reporting. If you neglect to instrument some functions, you may spend a good deal of time investigating the wrong cause of your bug. And what if you don’t even have source code available?

A second advantage of hardware trace is that it often has the ability to provide data values along with each instruction that is executed. This allows you to see the values that caused the program to take the incorrect execution path, which gives you more insight into what caused the problem and how to fix it. While you can certainly log values with software trace, it is impractical to log more than a few. If a bug occurs and you did not instrument one necessary value, then you may not be able to find the cause of your bug.

Finally, the most important advantage that hardware trace data has over instrumentation, especially for embedded systems, is that it does not require changing any aspect of your program to collect the data. This is vital for applications that may involve timing errors, interrupt latency issues or other bugs that only appear in certain combination of events.
Additionally, it is often impractical to instrument code in interrupt handlers or other timing-sensitive code. In nearly all high-reliability applications, you must certify and test the final code without any instrumentation, and software trace almost certainly cannot be enabled in production code, making it impossible to use software trace to track down bugs that appear in the production version of the code.

2.4 A Thousand Deaths No More

One of the insidious traits of debugging without trace is the need to repeatedly run the program in order to reproduce the problem within the debugger or custom test harness.

Some bugs take several minutes, hours, or even days to reproduce. Other bugs don't even repeat reliably and you must try certain sequences many times in trial-and-error fashion. If you step past the bug in the debugger, you have to start all over again. In contrast, New Debugging captures the bug (and your program state) at the time of execution and stores it away for you. You then use the trace-assisted tools to zero in by running backward and forward until you see the issue and then, the bug itself.

3 Components of New Debugging

The main components of trace-assisted debugging on the Freescale i.MX31 and i.MX27 applications processors are:

Hardware
- i.MX27 or i.MX31 ADS board
- SuperTrace Probe

Software
- MULTI™ IDE
- TimeMachine™ Debugger, Profiler and PathAnalyzer
- EventAnalyzer

For more information, visit the Green Hills website at www.ghs.com.
Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright license granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters which may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals” must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.