White Paper

The New Debugging

Robert Redfield

Green Hills Software

Document Number: NWDBGWP
Rev 0
4/2008

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

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 printfs. 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

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

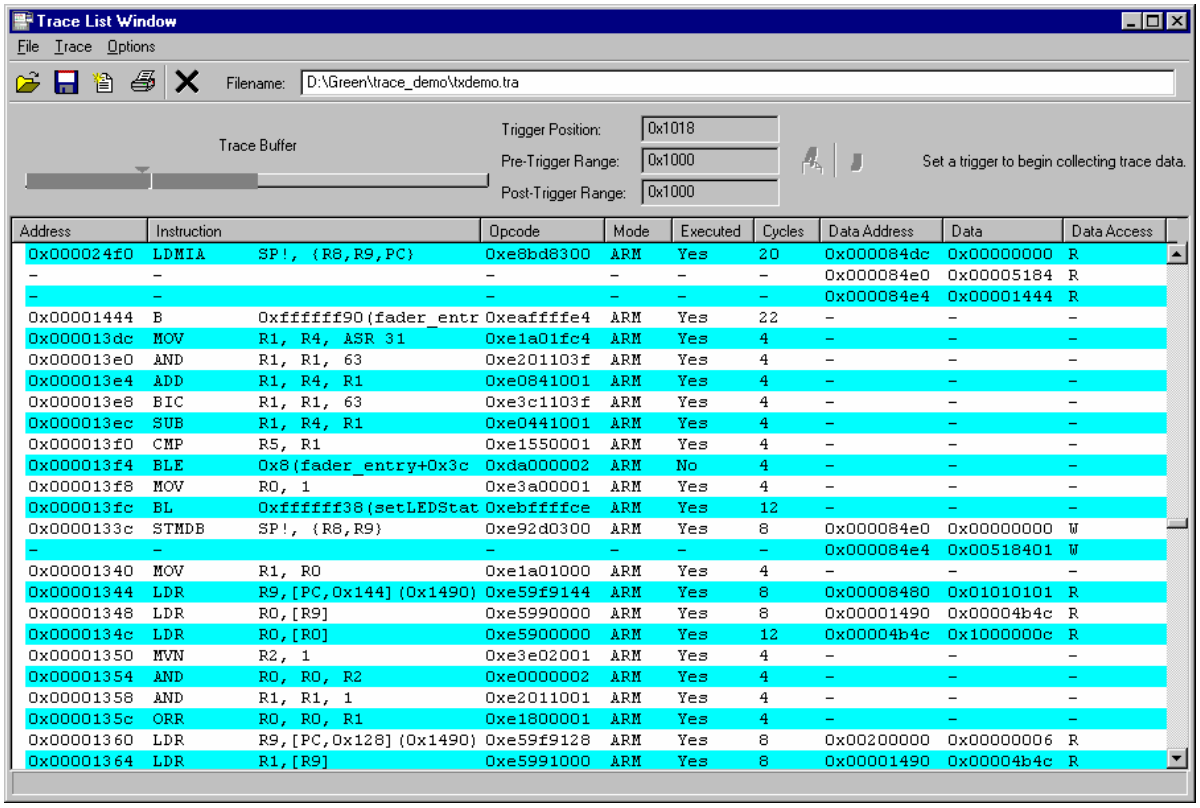
2.2 I've Mastered My Debugger and Profiler, Don't Make Me Learn Something New

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.



Filename: D:\Green\trace_demo\txdemo.tra

Trigger Position: 0x1018

Pre-Trigger Range: 0x1000

Post-Trigger Range: 0x1000

Set a trigger to begin collecting trace data.

Address	Instruction		Opcode	Mode	Executed	Cycles	Data Address	Data	Data Access
0x000024f0	LDMIA	SP!, {R8,R9,PC}	0xe8bd8300	ARM	Yes	20	0x000084dc	0x00000000	R
-	-		-	-	-	-	0x000084e0	0x00005184	R
-	-		-	-	-	-	0x000084e4	0x00001444	R
0x00001444	B	0xffffff90(fader_entr	0xeaffffe4	ARM	Yes	22	-	-	-
0x000013dc	MOV	R1, R4, ASR 31	0xe1a01fc4	ARM	Yes	4	-	-	-
0x000013e0	AND	R1, R1, 63	0xe201103f	ARM	Yes	4	-	-	-
0x000013e4	ADD	R1, R4, R1	0xe0841001	ARM	Yes	4	-	-	-
0x000013e8	BIC	R1, R1, 63	0xe3c1103f	ARM	Yes	4	-	-	-
0x000013ec	SUB	R1, R4, R1	0xe0441001	ARM	Yes	4	-	-	-
0x000013f0	CMP	R5, R1	0xe1550001	ARM	Yes	4	-	-	-
0x000013f4	BLE	0x8(fader_entry+0x3c	0xda000002	ARM	No	4	-	-	-
0x000013f8	MOV	R0, 1	0xe3a00001	ARM	Yes	4	-	-	-
0x000013fc	BL	0xffffff38(setLEDStat	0xebffffce	ARM	Yes	12	-	-	-
0x0000133c	STMDB	SP!, {R8,R9}	0xe92d0300	ARM	Yes	8	0x000084e0	0x00000000	W
-	-		-	-	-	-	0x000084e4	0x00518401	W
0x00001340	MOV	R1, R0	0xe1a01000	ARM	Yes	4	-	-	-
0x00001344	LDR	R9,[PC,0x144](0x1490)	0xe59f9144	ARM	Yes	8	0x00008480	0x01010101	R
0x00001348	LDR	R0,[R9]	0xe5990000	ARM	Yes	8	0x00001490	0x00004b4c	R
0x0000134c	LDR	R0,[R0]	0xe5900000	ARM	Yes	12	0x00004b4c	0x1000000c	R
0x00001350	MVN	R2, 1	0xe3e02001	ARM	Yes	4	-	-	-
0x00001354	AND	R0, R0, R2	0xe0000002	ARM	Yes	4	-	-	-
0x00001358	AND	R1, R1, 1	0xe2011001	ARM	Yes	4	-	-	-
0x0000135c	ORR	R0, R0, R1	0xe1800001	ARM	Yes	4	-	-	-
0x00001360	LDR	R9,[PC,0x128](0x1490)	0xe59f9128	ARM	Yes	8	0x00200000	0x00000006	R
0x00001364	LDR	R1,[R9]	0xe5991000	ARM	Yes	8	0x00001490	0x00004b4c	R

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

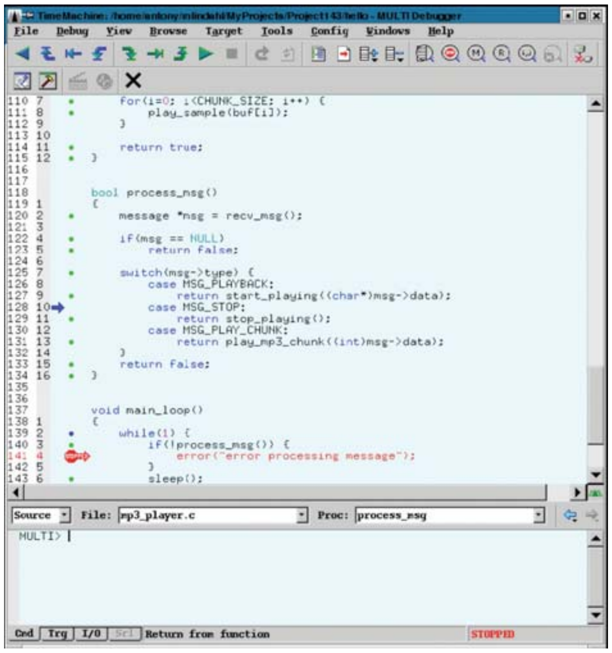
Here's an example showing how statistical sampling veils the view of actual execution from you. If a function executes 100 times, taking 100 cycles for 99 times it's executed, and then taking 10,100 cycles the final time, the traditional profiler will say that the function takes an average of 20,000/100 = 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.

Figure 2. A debugger with reverse execution. Note the blue arrow icons on the top of the debugger that are the reverse execution equivalent of the traditional (forward) green arrow icons. Other than that, it's the same debugger with the same features that you're already comfortable with.



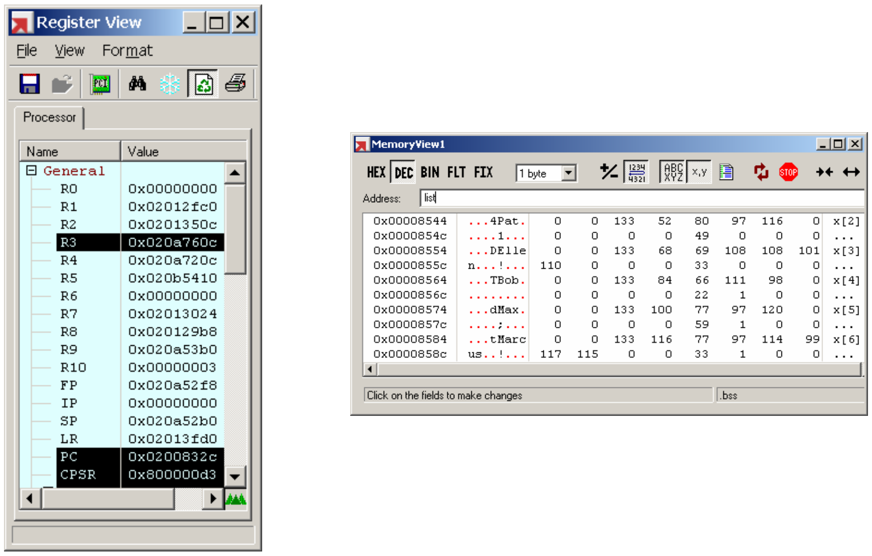
TimeMachine: /home/antony/nlindahl/MyProjects/Project143/hello - MULTI Debugger
File
Debug
View
Browse
Target
Tools
Config
Windows
Help

```
        for(i=0; i<CHUNK_SIZE; i++) {
            play_sample(buf[i]);
        }

        return true;
    }


    bool process_msg()
    {
        message *msg = recv_msg();

        if(msg == NULL)
            return false;

        switch(msg->type) {
            case MSG_PLAYBACK:
                return start_playing((char*)msg->data);
            case MSG_STOP:
                return stop_playing();
            case MSG_PLAY_CHUNK:
                return play_mp3_chunk((int)msg->data);
        }
        return false;
    }


    void main_loop()
    {
        while(1) {
            if(!process_msg()) {
                error("error processing message");
            }
            sleep();
```

```
MULTI>
```

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.

Figure 3. Trace-assisted debugging lets you see registers, variables and call stacks at any point in execution.

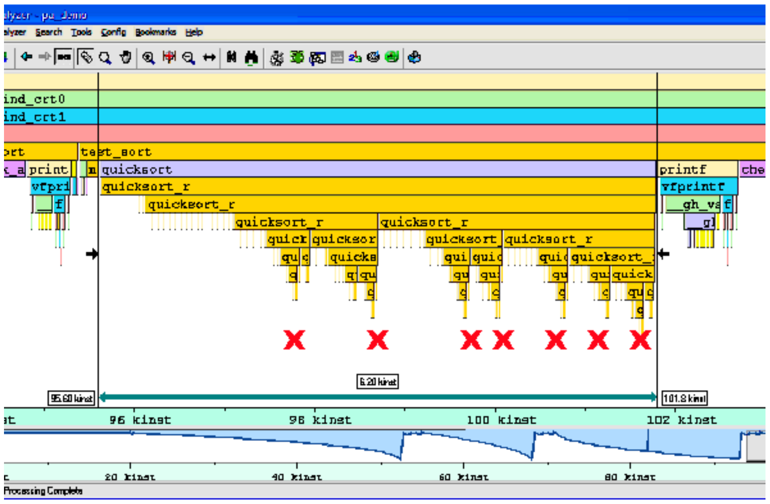


```
Name        Value
⊟ General
  R0        0x00000000
  R1        0x02012fc0
  R2        0x0201350c
  R3        0x020a760c
  R4        0x020a720c
  R5        0x020b5410
  R6        0x00000000
  R7        0x02013024
  R8        0x020129b8
  R9        0x020a53b0
  R10       0x00000003
  FP        0x020a52f8
  IP        0x00000000
  SP        0x020a52b0
  LR        0x02013fd0
  PC        0x0200832c
  CPSR      0x800000d3
```

```
Address: list

0x00008544   ...4Pat.     0    0  133   52   80   97  116    0   x[2]
0x0000854c   ....1...     0    0    0    0   49    0    0    0   ...
0x00008554   ...DElle     0    0  133   68   69  108  108  101   x[3]
0x0000855c   n...!...   110    0    0    0   33    0    0    0   ...
0x00008564   ...TBob.     0    0  133   84   66  111   98    0   x[4]
0x0000856c   ........     0    0    0    0   22    1    0    0   ...
0x00008574   ...dMax.     0    0  133  100   77   97  120    0   x[5]
0x0000857c   ....;...     0    0    0    0   59    1    0    0   ...
0x00008584   ...tMarc     0    0  133  116   77   97  114   99   x[6]
0x0000858c   us..!...   117  115    0    0   33    1    0    0   ...

Click on the fields to make changes        .bss
```

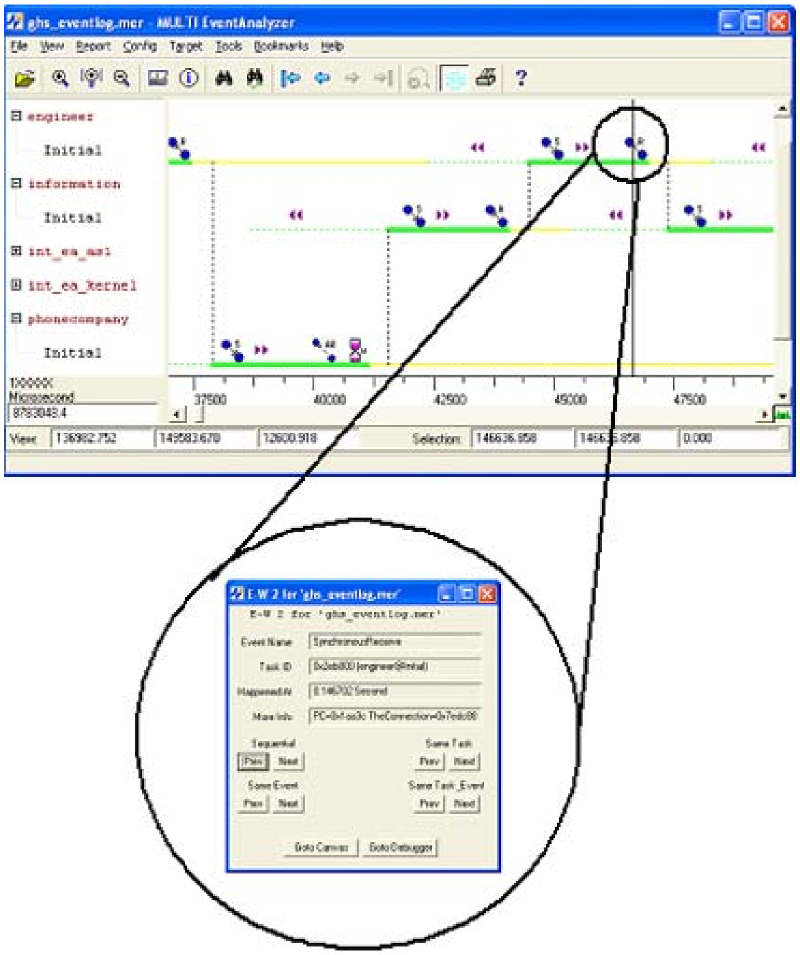
As with the trace-assisted debugger and profiler, there is no processor intrusion, recompilation or instrumentation necessary.

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-assited 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.

Figure 5. The EventAnalyzer charts RTOS-level events (y-axis) versus time. Shown here is an RTOS semaphore.



8783848.4

View: 136982.752 | 149583.670 | 12600.918

Selection: 146636.858 | 146636.858 | 0.000

E-W 2 for 'ghs_eventlog.mer'

Event Name: SynchronousReceive

Task ID: 0x2eb800 (engineer@Initial)

HappenedAt: 0.146702 Second

More Info: PC=0x1ca3c TheConnection=0x7edc88

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.

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: NWDBGWP
REV 0