

# Choosing and Taking Full Advantage of a Target with On-Chip Debugging Features

Selecting a CPU with on-chip trace and debug support along with the proper tools can significantly shorten development time contributing to overall project cost savings and shorter time-to-market.

by Darcy Wronkiewicz  
Green Hills Software

Choosing a CPU for an embedded product is tricky. The team set with the task of choosing a target architecture must ask a number of important questions. How fast does it need to be? How much power will it consume? How much memory does the application require? Most teams will consider these issues in the process of choosing their target architecture.

None of these questions, however, addresses the efficiency of the most expensive phase of building your product—the software development. Software problems can cost days, weeks or months on your schedule. Engineering time is expensive. Combining a target architecture with advanced on-chip debugging features, a hardware debug device capable of moving and analyzing data quickly and accurately, and a feature-rich debugger will create a debug environment that will shorten your time-to-market.

## Bare-Bones Application Debugging

A CPU that provides nothing in the way of on-chip debugging features paired

with a basic hardware interface (usually a serial cable) does very little to aid your host-side debugger. The shortcomings of this arrangement allow only primitive methods or force the developer to seek creative tricks. In the simplest case, you can instrument your code to light an LED on your hardware when a specific condition arises. If you have a serial port with a working driver, it can be used to transmit data to the development host.

A debug monitor may be employed to provide run control capability and access to memory, but because this is a program that runs in ROM it provides access to only a limited number of registers and utilizes valuable CPU time. Such a monitor may require the use of CPU capabilities needed by your own software. Because a debug monitor must be running at all times, it does not provide you with the option to completely halt your target to examine a problem. The monitor will also, of course, require use of your cache and other buffers, so debugging them will be impossible in this scenario.

It should be noted that some processors do allow for critical interrupt handlers

to run concurrently with a debug monitor. Specifically, newer ARM, IBM PowerPC and ColdFire cores allow you to run critical interrupts in the background while still giving you access to all of their hardware debug features in the foreground. If your system relies on handlers that must run to prevent damage to your system, even while you are debugging, this feature will be important to you.

Some debugging environments use a pairing of code instrumentation with a data collection agent, implemented either in software or via a hardware device, typically a logic analyzer. In this instance, the chip must have an external bus for data collection and your target must have a port capable of transferring the data to the debugger, making primitive code profiling and coverage analysis possible. You can also use this technique to monitor the performance of your memory allocation system, enabling you to detect memory leaks or other allocation bugs.

All of these solutions require you to resort to the use of code instrumentation, which severely limits your ability to accurately assess the real-world performance of

your final product. Additionally, when the instrumentation is removed, your system may be fundamentally changed. Subtle timing and memory bugs that appear at this stage are very difficult to diagnose.

### Upgrading with On-Chip Debugging

With a less austere setup in which on-chip features provide you with run control over your chip, you gain the ability to collect information during the program's execution. The program can be halted and information can be transported up to the development host for debugging purposes. This method does not require any code instrumentation, and because the data does not need to be stored on the target, a large amount of data can be collected.

It does, however, require the target to be halted for the information to be collected, so that the system you are debugging is still not a fully accurate representation of your deployed system. Depending on your project requirements, this may or may not be acceptable. Code instrumentation will still be necessary if you want to collect profiling data. The data you gather will be more accurate than that available with a debug monitor, but it remains an instrumented system, with the associated disadvantages.

Adding hardware breakpoints ushers in less intrusive methods for stepping, halting and monitoring execution. A hardware breakpoint is an on-chip debugging tool that watches a specific address and generates an interrupt if that address is executed or accessed for reading or writing. One register must be devoted to each hardware breakpoint. On some chips, a single hardware breakpoint can be configured to halt your program on the read, write or execution of a given address, as instructed. On other chips, each hardware breakpoint is designated for read/write or execution exclusively.

In either case, hardware breakpoints are superior to their high-level counterparts as implemented in a host-based debugger. Because hardware breakpoints require no software intervention or instrumentation, their existence does

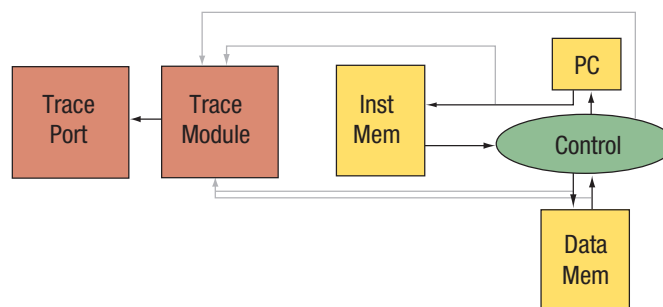


Figure 1

Each time an instruction is executed, the trace logic on the CPU collects the CPU status, and any other data necessary to describe the instruction executed, and presents it to the trace port where it can be recorded by a high-speed debug probe.

not degrade performance in any way. Additionally, hardware breakpoints allow you to debug code in ROM. With one hardware breakpoint you can single-step code in ROM. The debugger sets the hardware breakpoint at the next instruction to be executed and starts the target running. When the target is stopped the process repeats.

Where source-level debugging is preferable, the hardware breakpoint is set on the address correlating to the next line of source code to be executed to achieve source-stepping. The availability of additional hardware breakpoint registers brings with it the ability to set multiple hardware breakpoints in your code, or on specific pieces of data. Advanced debuggers may be able to stretch a single hardware breakpoint to simultaneously provide both single-stepping capability and a user hardware breakpoint by watching for the address specified by the user's hardware breakpoint at each target halt.

### System Analysis with Trace

The conditions outlined in each of the previous sections have given us increasingly effective debugging capabilities, but none has offered a complete view of our uninstrumented system's execution. On-chip tracing capability steps into this breach. Tracing generates an exhaustive record of a program's execution. What instructions were executed? What memory was read? What memory was written? What conditions arose that caused the system to behave as it did?

Trace-enabled CPUs have on-chip data collection logic built into the hardware. Each time an instruction is

executed, information is encoded and compressed into a well-defined structure before being transferred off the CPU. It is then either exported via a high-speed trace port to the debug hardware or stored in an on-chip buffer for later retrieval (Figure 1).

A CPU with execution trace capabilities will export its current status (whether the CPU is branching, handling an interrupt, etc.) and, in the case of an indirect branch, the program counter value on each instruction. The right debugger can use this information

and a copy of your program image to reconstruct the order in which your code was executed. High-quality debuggers can correlate this information with your source code, enabling you to virtually walk through the traced portion of your program run. Code coverage analysis and timing statistics are very accurate when execution trace is provided. Since the code does not need to be instrumented, performance analysis tools can utilize the data to provide accurate profiling information.

Adding data trace to the mix opens up a whole new level of debugging. At this point, the state of memory and registers can be tracked throughout the program's execution. With the value of the PC at every clock cycle and the value of every memory and register value in hand, you have everything you need to completely reconstruct the traced portion of the program run.

This survey results in a tremendous amount of data that can only be reliably collected and analyzed by a configuration of the most advanced solutions. In less ideal situations, the chip in use may not have the capability to export the relevant data. If you are storing information on an on-chip buffer, you may be limited to several kilobytes of information. Alternately, the debug hardware may not be up to the job of storing information in the volume and at the speed required. Activating data and execution trace generates hundreds of megabytes of data per second. The user may also lack the advanced software needed to present the data in a coherent fashion.

In practice, most people using some kind of trace solution trace only a very

small subset of this data in hopes of targeting the specific information they believe they are most likely to need. These users utilize triggers to determine when data should be collected. For example, if it is determined that a failure only occurs after the application's main data buffer is populated, a trigger can be set to wait until that point to begin data collection.

In the real world, however, the most devastating bugs are memory problems whose origins are difficult to track. In this case, it is preferable to be able to trace program execution from the beginning. This is only possible using a high-speed debug device that can connect to a trace port on your chip to collect data quickly and accurately. In such a setup, when you realize that something has gone wrong with your memory, you can simply gather your trace data and start exploring your program run.

Advanced debuggers will allow you to retroactively "debug" the trace data as if it were a live system. You will be able to set breakpoints, run and single-step your code. Because the run is a reconstruction, you can move forward and backward through it. You can set virtual hardware breakpoints on your corrupted memory and run your program reconstruction to see exactly when, how and why the memory was corrupted (Figure 2).

### Cost Considerations

How much is a debuggable chip worth? How much should you spend on a good development environment for your programmers? The answer is another series of questions. How important is meeting your schedule? Can you really afford for a key developer to be tied up in diagnosing a rarely occurring memory corruption problem for days or weeks? How often will you be upgrading your software? How tight will the schedule for these upgrades be? How efficient does your code need to be? Can you afford to have code that wasn't profiled by the most advanced tools available? Software development usually comes at the end of the development cycle for a product. Taking the risk out of software development significantly increases your chances of getting to market on time.

The most advanced tools on the market today paired with speedy debug

The screenshot shows the TimeMachine debugger window titled "TimeMachine: C:\blue\master\_engine - MULTI Debugger". The main window displays source code for a file named "trace\_demo.c". The code includes several functions and a main function. A red arrow points to line 59, which is highlighted in red and labeled "STOPPED". Below the source code, there is a window showing trace data for the "do\_tick" process. The trace data includes memory addresses, values for "message\_buf", "sin\_value", and a "parse error" message. The debugger interface includes a menu bar (File, Debug, View, Browse, Target, Tools, Config, Windows, Help), a toolbar with various icons, and a status bar at the bottom with buttons for "Cmd", "Trg", "I/O", and "Str", and a "STOPPED" indicator.

```

49 9      }
50 10      •   if(tick % 20 == 0) {
51 11      •       perform_maintenance(tick);
52 12      •       ++num_maintenance;
53 13      •   }
54 14
55 15      •   if (tick&0x1) {
56 16      •       export_data(sin_value);
57 17      •       angle=tick/(2*3.1413);
58 18      •       sin_value=sin(angle);
59 19      •       export_data(sin_value);
60 20      •   }
61 21
62 22      •   last_tick = tick; /* update the last_tick value */
63 23      •   ++num_ticks;
64 24      •   sprintf(message_buf, "tick %d\n", tick);
65 25      •   }
66
67      int main(int argc, char **argv)
68 1      {

```

```

message_buf
&message_buf = "tick 18^J"
sin_value
sin_value = 0.422051050120763
sin_value
sin_value = 0.422051050120763
&
parse error
MULTI>

```

**Figure 2** Trace data recorded by the debugger probe is correlated with source code on the host to reconstruct the exact program sequence. This can include the occurrence of an unexpected bug, which can then be examined without having to rerun the program in the hope that the bug will reoccur.

hardware and an advanced debugger can trace every step of your program so thoroughly that it allows you to replay the program backward and forward. There is enough memory available to leave the trace mechanism on, allowing you to debug those hard-to-reproduce problems the first time they occur. What could be next?

Watch for CPU vendors to get smarter about providing debugging capabilities on-chip. As memory gets cheaper, you can expect debug probes to have more of it and be able to collect more and more trace data. What this means for diagnosis is that your company will be able to ship out products with built-in trace devices. Users can then

ship malfunctioning products back to you complete with a detailed history of what went wrong. The implications of this prospect for testing and long-term product reliability would clearly be hard to overstate. ■

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



**Green Hills®**

• S O F T W A R E , I N C . •

[www.ghs.com](http://www.ghs.com) ▲ 805-965-6044