A Program of the IEEE
Industry Standards and
Technology Organization

Nexus Standard Brings Order to Microprocessor Debugging

A white paper from the Nexus 5001Ô Forum

Contents

Nexus Standard Removes Some of the Pain from Debugging

The old joke goes, "if debugging is the act of removing bugs, then programming is the act of adding them." That's a sad reality of engineering today. Like death and taxes, we accept that unhappy fate as the immutable nature of things. But there is hope on the debugging front.

Statistics show that fully one-half of development time is spent debugging software and hardware. While it's unreasonable to expect that debugging could ever be eliminated, it's not too much to hope that it might be reduced. Debugging takes a long time because neither the skill nor the tools can be leveraged. That is, the expensive tools (and the skills to use them) are unique; every debugging project is a one-off. Unlike most professional endeavors, engineers never get significantly better at debugging because they're always starting over with new tools, new problems, and new learning curves.

Fortunately, the same innovations and advancements that drive the semiconductor industry feed back on themselves. Computers and semiconductors are increasingly used to improve computers and semiconductors. Physician, heal thyself.

"You can't solve problems with the same minds that created them."
–Albert Einstein

The problem boils down to one of familiarity and potentially, standards. Like anyone, engineers get significantly better with practice. Racing drivers practice a particular course over and over, and they know their cars well. A new driver on a new course in a new car isn't expected to be competitive. In fact, they're a hazard both to themselves and to others. In a similar vein, even a practiced team of engineers can't be productive when they're constantly presented with new debugging tools, new software interfaces, and new methods of applying those tools. Like standardizing the placement of the brake, accelerator and clutch pedals, applying some rational commonality to debug tools allows practitioners to concentrate on their skills, not on coaxing the equipment to work.

This wasn't always a problem. Microprocessors and embedded systems used to be relatively simple, so designing and debugging them was simpler. The first microprocessor chip had just 2,300 transistors in 1971; a Pentium 4 processor has more than 120 million transistors. This 50,000-fold improvement over 30 years (commonly called Moore's Law) means that microprocessor chips are vastly more complicated than yesteryear's. That much is obvious.

What's less obvious is that the inner workings of these chips – what goes on under the hood – is no longer observable. In a modern car, lifting the hood to look at the engine is no longer useful; it’s just a tradition. Modern automobile engines are buried under piles of inscrutable plumbing, hoses, wiring, heat shields, and cosmetic covers. The days of the shade tree mechanic rolling up his sleeves and fixing an engine with hand tools are gone. Modern mechanics rely on computer-diagnostic equipment and electronic “tools” to diagnose and repair faults. Every car sold for the past several years has come with a standard computer connector (called OBD II, or on-board diagnostic) for that very purpose. There’s no way to repair a modern car, or even know what’s wrong with it, without that standard plug under the hood.

Cadillac’s 1971 DeVille was the first car to have an electronic microprocessor in it. The chip, a simple 8-bit processor, controlled the in-dash “trip computer.” Today, even a basic economy car includes at least a dozen microprocessor chips, controlling everything from ignition timing to the automatic transmission. Luxury models can easily have several dozen processors. An S-class Mercedes-Benz includes about 65 microprocessors; a BMW 7-series can have nearly a hundred. Networks – not just tangles of wiring – connect all these chips, exactly the way that computers are networked in an office building.

None of this amazing complexity could have come about if the automobile manufacturers weren’t completely confident that they could maintain these “rolling data centers.” Automakers are notoriously risk-averse regarding new technology. Moreover, dealers and service centers aren’t generally noted for their computing and networking skills. Yet in spite of all this, the simple 8-bit processors in cars rapidly gave way to clusters of 32-bit processors stuffed with megabytes of code, data, and performance parameters. And it will only get more complex with time.

The key is the standard OBD II diagnostic plug, and the equipment that goes with it. No repairman could hope to divine the inner workings of such a complex system without computer aids. And no diagnostic machine could hope to work on more than a few cars unless OBD II was an industry standard. The OBD II standard has allowed automakers to go crazy with new features while protecting the investment of innumerable repair shops, large and small. Car models may come and go, and diagnostic machines may get upgraded or replaced over time, but the OBD II standard keeps them all working together.

In a similar vein, microprocessor chips are now sprouting standard diagnostic ports so that engineers and programmers can identify and fix problems in software or hardware. Like the global OBD II standard, this new microprocessor diagnostic plug is standardized so that all technicians, programmers, vendors, customers, and engineers can rely on it and use it. And like OBD II, it means that multiple tools can work on multiple chips, with no regard to who made either one. The diagnostic port provides insight into the inner workings of the microprocessor chip: a view “under the hood.” To the international standards committee publishing it, the diagnostic port is known as IEEE-ISTO 5001. To the engineering community developing and using it, it’s simply called Nexus.

Under the Hood of Your New Nexus

The goal of the Nexus standard is deceptively simple: to provide a standard way for programmers and engineers to see what's going on inside their systems. To do that, Nexus defines a standard way for diagnostic equipment to communicate with microprocessor chips. Nexus plugs allow emulators, debuggers, logic analyzers, and workstations from one maker to control and debug processors from any other maker.

"Modern microprocessor chips now have a standard diagnostic plug to help engineers and programmers identify and fix problems with their software or hardware"

The first benefit is obvious: if engineers and programmers can examine what's going on inside their systems, they can better find and fix bugs. That cuts debugging time, which also reduces cost and frustration. As a bonus, that same visibility gives programmers the opportunity to find performance-enhancing tweaks.

The second benefit is standardization. Having the same diagnostic port on all microprocessor systems means that diagnostic equipment, and engineers' diagnostic experience, are transferable to other projects. Like a car's OBD II plug, a standard Nexus port allows nearly any engineer to debug nearly any system without buying and learning to use specialized tools.

These two simple goals leverage numerous other benefits. If test equipment has a Nexus standard interface, it can be used on multiple projects, amortizing the considerable investment across different project or products. (Note that the test equipment itself is never standardized, only its interface with microprocessors. Equipment makers will innovate and compete just as before.) If microprocessor debugging and diagnostic interfaces are standardized, it alleviates the burden of creating and supporting in-house debug interfaces which those microprocessor makers now bear. Vital debug features can become simple "check box" items that don't require years of custom development effort.

This has already happened with a number of 32-bit processors from Motorola, STMicroelectronics, and other chip makers. These chips, although very different inside and out (and made by competing vendors), all sport a Nexus standard interface. Motorola's MPC55xx chips, for example, are PowerPC compatible parts, while its MAC71xx chips are ARM-based. ST's ST10 family uses still another microprocessor architecture. These utterly different devices all benefit from the same Nexus interface, and customers can use the same debug tools across all these families, and more.

"TCP/IP standards gave rise to the Internet. Nexus also lays the groundwork for common communication; what happens next is up to developers."

Under the Nexus standard, microprocessor makers are free to innovate and develop their own debug and diagnostic features. Nexus doesn't in any way impede differentiation, originality, or improvement. It merely rationalizes the connection between two moving targets: microprocessors and their development tools. When the TCP/IP standard simplified computer networks, it gave rise to the Internet. No one could watch the meteoric rise of the Internet and argue that TCP/IP somehow limited its usefulness or appeal. Likewise, Nexus lays the groundwork for common communication; what happens next is up to developers.

The Evolution of Debug Problems

Debuggers and debugging techniques have been around for as long as there have been computers. Legend has it that the first computer bug was literally that: a moth caught in the electromechanical relays of an early computing machine (programmed, it should be added, by Ada Byron, Countess of Lovelace and daughter of poet Lord Byron, and after whom the Ada programming language is named). So computer bugs are clearly not a new problem. Why is a new method of exterminating them required?

Without keyboards, disk drives, or video monitors, most embedded systems have no way to display or record what's happening. Unlike a computer, it's hard to observe an antilock brake system (for instance) and tell what it's doing. The same is true of an MP3 player, video recorder, network router, industrial robot, or cellular handset. Today 98% of all microprocessor chips are used in embedded systems, not computers, so nearly all engineers face this problem.

In past years, you hooked up a logic analyzer (or an oscilloscope – or even a voltmeter!) to observe and record what the chip was doing. Bugs were annoying, but at least you could *see* them. Microprocessors with caches and microcontrollers with on-chip ROM or flash memory started producing bizarre effects. Watching the outside of the chip told you nothing about what was happening inside. Thousands of lines of software could run and produce no outward effects at all – the chip appeared dead. Only by deduction could you hope to guess if the right thing had happened.

Bugs get more complex because computers get more complex. But it's subtler than that. Modern microprocessor chips have made bugs unobservable – invisible by normal means. It's like early medicine: before the advent of X-rays physicians had to rely on only what they could observe outside the patient. Broken bones might be guessed at, but any other internal ailments were utterly mysterious. A *post mortem* examination might reveal the cause, but then it's obviously too late to help the patient. The knowledge gained might or might not help the next patient.

Programmers and engineers also rely on post mortem debugging: wait until the system fails and then see what evidence is left behind. If there's enough evidence, the programmers might glean the cause of the failure. Fix the bug, reboot the machine, and wait for the next problem to arrive. When the system no longer fails (or doesn't fail for a sufficient period of time) it's deemed to be "fixed" and shipped to customers.

Still Banging Stones Together

A slightly more proactive form of debugging uses the popular debug monitor. "Monitor" is a generic term for any small piece of software that runs on a microprocessor (or other computer system) for the purpose of finding and isolating bugs. It's essentially a servant working "on the inside" of the system, carrying out orders on behalf of the programmer. When the programmer asks the monitor to examine a memory location or stop the processor chip from running, the monitor makes it happen. Debug monitors are very useful and cost very little to implement; every first-year engineering student is familiar with them. However, they're also very limited and suffer from a few major drawbacks.

First, debug monitors basically ask the processor to debug itself. That is, the monitor is simply a program running on the same microprocessor that's being scrutinized. If there are problems with the software (or the hardware) those problems could easily affect the monitor as well. Also, microprocessors can only do one thing at a time, so any time spent running the debug monitor is time taken away from running the actual system software. By their nature, monitors alter the performance and behavior of the system they're meant to observe. Like a policeman standing in a bank lobby or a health inspector in the middle of a restaurant kitchen, the very presence of a debug monitor may prevent the problems it's meant to catch.

Debug monitors depend on a healthy system. That is, the system can't be too badly out of whack or the debug monitor itself won't work. Monitors don't work at all if the system won't boot, or if it's being started for the first time. Finally, monitors aren't able to debug programs stored in ROM (read-only memory) or flash memory, which excludes most embedded systems. They also require system resources, such as RAM and an I/O port, that are normally reserved for the "real" application software. Still, because of their low cost and widespread familiarity, debug monitors hold a special place in most programmers' hearts.

Bug Hunting Moves Forward

The next step up the debugging food chain is the in-circuit emulator. An emulator, or ICE, replaces the microprocessor chip with a box that emulates its functions. The emulator box is connected through a cable to a standard computer that reports what's happening inside the emulator. The computer can also issue commands to the emulator, forcing it to stop, start, or modify the program that it's running.

Emulators were the *sine qua non* of debugging technology in the early 1990s. Every serious engineering team had one because they were (and still are) such an immense improvement over simple debug monitors, oscilloscopes, or logic analyzers. Emulators can capture information from inside the ersatz chip as it runs, buffer it, and send it to the computer for later analysis. They also don't require the run-time resources of a monitor; emulators leave no "footprint" on the system they're scrutinizing.

That's not to say emulators make no demands on the system; they do. First of all, they're bulky. An entire box full of electronics has to stand in for one small chip, so the mechanical considerations are bound to be tricky. For simple microprocessors and microcontrollers (i.e., low-end 8-bit and 16-bit parts), the emulator usually plugs into the same chip socket as the processor itself. Sometimes the emulator box has a short, flat cable ending in a mechanical stand-in for the processor. Either way, these only work up to a point.

"Beyond 100 MHz or so in-circuit emulators start to exert unwanted influences on the system and we're back to the problem of disrupting the very system we're meant to observe"

Modern microprocessors invariably come in surface-mount packages that can't be socketed. Emulators for these chips have to clip precariously on top of the actual processor chip. Instead of replacing the processor, they tri-state (deactivate) its pins and drive the I/O signals in its place. Even so, this only works for chip frequencies up to about 100 MHz or so. Beyond that speed the socket and the cables start to exert unwanted influences on the system – and we're back to the problem of disrupting the very system we're meant to observe.

Finally, in-circuit emulators are expensive and generally useable for one specific type of microprocessor chip only. Should the engineers later decide to use a different chip, they need to buy a different emulator. Given the rapid pace of microprocessor development over the past three decades it's no wonder that development labs everywhere are littered with obsolete in-circuit emulators.

Turning That Intelligence to Good Use

The current state of the debug art came about because microprocessors are just too darn fast for an in-circuit emulator. Emulators could keep up with microprocessors running at double-digit clock speeds but current chips run at well over 3 GHz (3,000 MHz). Even relatively low-cost embedded microprocessors routinely exceed 1 GHz. There is simply no way an in-circuit emulator can keep up with that kind of speed. And no one's suggesting that microprocessors are about to get slower.

It gets worse. A 1990s-era microprocessor might have contained 1 million transistors or so, and included perhaps 4 KB of on-chip cache memory. The whole thing ran at about 100 MHz and sat in a 128-pin socket with the pins spaced in neat rows 0.10 inch apart. Today, a 3-GHz microprocessor can easily contain 100 million transistors, 2 MB of cache (in multiple levels), and come in a 400-contact ball-grid array. The new processor is also likely to have multiple execution pipelines, non-deterministic branch prediction, superscalar execution, and one or more coprocessors. In short, there's no way to adequately emulate the behavior of such an immensely complex chip. It's not just a matter of daunting complexity; it's literally impossible. Today's complex processor chips can't work except in their micro-miniaturized form. Their very operation depends on being smaller than a postage stamp.

"At these speeds, being one second late means executing a billion instructions wrong"

Yet these tiny chips process millions – sometimes billions – of operations per second. Virtually none of this processing happens outside the chip. It's all internal, and therefore invisible to programmers and engineers yearning for some clue about what's happening inside their system. At these speeds, being even one second late means executing a billion instructions *wrong*.

The silver lining in all this is that now it's possible for the chip to help debug itself. With tens of millions of transistors devoted to the intricate complexities of the microprocessor, a few thousand more devoted to debug is silicon well spent. Thus did the next generation in debug technology come about: on-chip debug. One by one, microprocessor vendors developed special debug features and included them in their own processor chips. The result is somewhere between software debug monitors and hardware in-circuit emulators. Using the chip's own hardware, these processors could send simple progress reports about their behavior and follow limited commands to start, stop, or modify execution. Best of all, these features resided inside the chip, so no extra equipment was required and the chip could (theoretically, at least) report on conditions and events hidden inside the recesses of the microprocessor itself.

The only down side was that different chip vendors developed vastly different capabilities (and names) for these features. Motorola's name for "background debug mode," or BDM, has become the generic term for all forms of on-chip debugging including those from Texas Instruments, IBM, Analog Devices, and other vendors. All of them are specific to their vendor, their microprocessor architecture, and often to a particular chip.

Enter the Nexus

Nexus combines all of the above features in a way that nearly any chip or tool can use. Because it's a standard, engineers' actual tools are transferable across projects, products, and platforms; so is their experience. Because it's flexible and open, Nexus also offers leeway and freedom to innovate, improve, and differentiate products from one another. Nexus provides a common set of features the way NTSC or PAL television standards ensure all TVs work together. Like those broadcast standards, Nexus doesn't restrict programming or content; it simply lifts the technical burden from the toolmakers, microprocessor vendors, and programmers working in the industry.

By the way, Nexus isn't restricted only to debugging. Nexus-compatible tools can be – and are – used for performance monitoring, hardware-in-the-loop simulation, calibration, and measurement. Anything that helps the developer look inside his chips or systems is germane to what the Nexus standard does.

"Nexus isn't limited to debugging. It's just as useful for performance monitoring, calibration, and hardware-in-the-loop simulation,."

The Nexus standard groups products into four classes. Class 1 chips and tools support a few minimum recommended features, while Class 4 products provide a wealth of advanced debugging features. Naturally, Class 2 and Class 3 products fall somewhere in between. Classes 1 and 2 simply generalize and standardize features already found in existing chips and/or tools. In Class 3, and especially Class 4, Nexus brings new features to both side of the equation. The Nexus Forum (the independent worldwide industry group that created and maintains the Nexus standard) provides a checklist of features recommended for each class of debug support.

Each of the four classes includes some software features and some hardware features. Hardware and software features both get more advanced for the higher classes, and each class is a superset of the next-lower class. This ensures that any tool is compatible with any chip, and vice versa. For example, using an "overqualified" Class 4 tool to debug a simple Class 1 microcontroller chip works perfectly well. More important, the same tool will work on Class 2, Class 3, and Class 4 chips as well.

Nexus Uses Existing JTAG Port for Debug

The first order of business in communicating between a debug tool and a chip is finding some way to communicate. How do the two connect to each other? Here, Nexus took the prudent approach and used what's already there. Rather than define a new "Nexus debug port" that would need to be added to every new chip, Nexus generally uses standard existing JTAG ports. The JTAG standard has been around for many years and is internationally accepted as a low-cost and unobtrusive way to add a testing "keyhole" to processors, memories, and other chips.

JTAG ports (also called a TAP, or test-access port) need only a few pins, so they're almost *de rigueur* on any chip that's big enough to see. The JTAG pins frequently do double duty as general-purpose I/O pins or data bus pins, so in effect they don't take up any space at all. Nexus piggybacks on these pins for its communications path. The JTAG standard (technically called IEEE 1149.1) was originally developed as a way to test several chips on a single board through a single test port. Fortunately, its creators foresaw the need for other standards to build upon JTAG, and that's precisely what Nexus has done. Nexus is an ideal and expected application of JTAG.

Although the standard JTAG port is adequate and widely available, vendors can choose to add pins to make the JTAG/Nexus debug port faster. In fact, an entire 32-bit data bus with control signals can be added for high-speed communication between the debug tool(s) and the chip(s) being debugged.

The Nexus Basket of Features

Nexus provides a smorgasbord of features, functions, and options that vendors can choose from. Some of the class designations (Class 1, Class 2, etc.) specify certain features that *must* be included, but by and large most features are optional. Vendors – and customers – are free to choose the features that are most desirable.

Program Trace

The ability to follow, or trace, the flow of a program as it runs is fundamental to all debuggers. This gets harder all the time, however, since modern processors can execute more than a billion instructions per second. There's no way (and no need) to keep up with that speed. It's more important to know when the flow changes – that is, when the microprocessor chip changes its mind and heads down another software path. These changes are often the keys to locating bugs. They're also great opportunities to tune the software for better performance.

Nexus provides a "program trace" function that does exactly this. It reports changes in the program flow back to the debugging tool. Interrupts, exceptions, and taken branches can all change the program flow. Nexus-compatible chips can report this trace data back in either of two ways. The "traditional" method reports the number of machine instructions executed since the last report. A debug tool can use this information to pinpoint the exact place the flow changed. The "historical" method includes additional information such as the address of the branch target (the place the microprocessor branched to) and any branch-condition predicate bits.

It's important to cut down on the number of reports a chip produces to avoid overwhelming the chip's JTAG/Nexus port. That's why program traces report only branches, not every instruction executed. On the other hand, it's important for the debug tool to stay synchronized with the chip. To that end, the chip sends a "here I am" trace report after every 256th trace report. The chip can also send reports more frequently if the debug tool asks for them.

Data Trace

Sometimes tracing the flow of a program isn't enough. When data gets corrupted or an I/O register changes unexpectedly, it's sometimes hard to tell when or why it happened. Trapping an accidental write is notoriously difficult, and is the one feature most programmers wish they had.

Nexus Class 3 and Class 4 devices add a level of sophistication and detail that rarely appeared on previous debug methods: a data-trace feature that reports when writes occur within a specified address range. This means that the chip can monitor at least two different address ranges and capture all the writes within those spaces. It then reports back to the debug tool specifically what data was written to what address. This is usually enough to pinpoint the problem.

As with program traces, Nexus-compatible chips avoid reporting more data than they have to, so data trace reports are abbreviated, including only enough of the address to be unambiguous. Every 256th trace report includes a full address to guarantee that the debug tool and the chip stay synchronized.

Memory Access

Another fundamental feature that most programmers rely on is the ability to "peek" and "poke" memory locations. Usually when the processor is stopped, programmers will examine the contents of memory, looking for clues or assuring themselves that everything is well. To test infrequent behavior or force failure modes they can also modify memory and restart the program. Having full read/write access to memory is key not only to debugging but to testing and quality assurance.

Nexus codifies a method for debug tools to access memory through the processor. This "through the processor" method is important, for shared memory can sometimes behave differently depending on how it's accessed. For example, a microprocessor with a data cache (a category that includes most chips today) might have an "exclusive" copy of newly calculated data, while a DMA controller in the same system sees "stale" data that hasn't been flushed from the cache. Cache behavior like this has confounded more than one programmer. Without "through the processor" access to memory, these subtle problems escalate into frustrating debug sessions.

Memory Substitution

In-circuit emulators work by swapping out the actual microprocessor chip with a simulated or emulated version. They can also swap out the system's actual memory with their own. Simple debug monitors can't swap out memory; they run using the real memory in the system.

Nexus debuggers can work either way. This Class 4 feature lets the programmer command the debug tool to swap out an entire region of system memory and replace it with alternate memory (probably located in the debug tool or in the programmer's own computer). This feature is very useful for developing and debugging ROM code, which normally can't be modified.

Breakpoints and Watchpoints

At a minimum, all Nexus-compliant chips and debuggers must have at least two hardware breakpoints. These simple, but vital, features are key to most types of debugging. Breakpoints allow the debugger to stop the processor in its tracks as soon as it reaches a certain point in its software. It's like directing a student driver, "stop when you reach the intersection of 17th and Lighthouse and wait for further instructions."

Hardware breakpoints are more powerful than traditional breakpoint instructions (software breakpoints) because they don't modify the software in any way. Software breakpoints don't work at all in ROM code (which can't be modified) and they often upset the flow of other programs as well. Hardware breakpoints solve both these problems and are now a common feature in most processors. Nexus standardizes how they're handled across different debug tools.

"Watchpoints" are a variation on breakpoints. They don't stop the processor like breakpoints do; they just report back to the debugger that the stop would have occurred. Watchpoints are invaluable for debugging real-time code that can't be stopped.

Ownership Trace

"Ownership trace" is an advanced feature that allows the debugger to tell what task ID (in a multitasking operating system) or process ID (in a multithreaded operating system) is currently in control. This is important so that, for example, the debug tool can tell what symbol table (list of variable names and mnemonics) is valid for the currently running task. It also lets the programmer see what operating system task is running.

Ownership, processes, and task IDs are something early debuggers never worried about. Now, even video recorders run multitasking operating systems and development tools have to keep up with them.

Port Replacement

Examining registers and memory is all very well, but I/O is important too. Most embedded systems includes lots of external I/O, whether it's a network interface, serial channels, disk drives, switches, LEDs, toggles, relays, motors, or something else. Moreover, most embedded microcontrollers have an array of I/O features on them, and customers generally choose one chip over another precisely *because* of those features. The point is, the debugger mustn't get in the way of the chip's I/O ports; it has to work around them as unobtrusively as possible.

"Port replacement" is a Class 2 feature that requires the debugger to emulate up to 16 general-purpose I/O pins. This is meant to make up for the pins the debugger itself might use as part of a widened JTAG/Nexus interface.

Time Stamps

The microprocessor chip or debug tools can optionally time-stamp their reports, appending the actual time to normal program-trace reports or other messages. This might be hugely valuable to a programmer who's trying to locate a bug in a real-time system.

This is more than mere nicety. Real-time systems, by their nature, can't be interrupted or paused for interrogation. Their timing is just as important as their logical operation. Time-stamping and recording events is the only nonintrusive way to verify or debug such a system.

In a related vein, the chip can also toggle an external pin on certain events. A later trace report can then explain what the event was, with a time stamp. This is important for real-time reporting or data logging because it indicates the event when it happens. A logic analyzer or other hardware tool could be triggered in this way.

Table of Features by Class

The following table summarizes the basic Nexus features and characteristics by product class.

FEATURE	CLASS 1	CLASS 2	CLASS 3	CLASS 4
Read & write registers while in debug mode	⊕	⊕	⊕	⊕
Read & write memory while in debug mode	⊕	⊕	⊕	⊕
Enter debug mode from reset	⊕	⊕	⊕	⊕
Enter debug mode from user mode	⊕	⊕	⊕	⊕
Exit debug mode to user mode	⊕	⊕	⊕	⊕
Single-step instruction; reenter debug mode	⊕	⊕	⊕	⊕
Stop on breakpoint; enter debug mode	⊕	⊕	⊕	⊕
Set breakpoint or watchpoint	⊕	⊕	⊕	⊕
Device identification	⊕	⊕	⊕	⊕
Notify of watchpoint match	⊕	⊕	⊕	⊕

Monitor process ownership in real time (ownership trace)		⊕	⊕	⊕
Monitor program flow in real time (program trace)		⊕	⊕	⊕
Monitor data writes in real time (data trace)			⊕	⊕
Monitor data reads in real time			Optional	Optional
Read & write memory in real time			⊕	⊕
Execute program through Nexus port (memory substitution)				⊕
Begin trace on watchpoint				⊕
Begin memory substitution on watchpoint				Optional
Low-speed I/O port replacement		Optional	Optional	Optional
High-speed I/O port sharing		Optional	Optional	Optional
Transmit data-acquisition			Optional	Optional

Still More to Come

Nexus is designed to grow. It's extensible and it anticipates the rapid-fire pace of change in the microprocessor industry. Several aspects of the standard leave the door open to future improvement, enhancement, or unforeseen changes in the industry.

For example, Nexus has a provision for multiprocessor chips. Up to 32 processor "cores" can reside in a single microprocessor chip and still be debugged through the same Nexus debug channel. Trace reports and other features all have provisions for identifying not just process ID or thread ID, but the *core ID* of the program being debugged.

Multiple processors, and multiple cores per processor, are already the rule, not the exception. Research shows that the average custom ASIC or SoC (system on a chip) has 3.2 processor cores, and the average is rising. Even simple consumer devices like cell phones or MP3 players have multiple processors that must work together. Automobiles, as we've seen, have dozens of processors. Multi-processor debugging is a necessity for future expandability.

Nexus also has headroom for new features, commands, messages, report types, and other improvements. Cache-coherence features, for instance, can be folded into the standard without upsetting its current structure. Just as important, any vendor could choose to add those features, whether they were defined by the Nexus committee or not. The standard is not restrictive; it invites and encourages additions.

Nexus Private Messages allow for extensions to the standard that are secret or proprietary. For instance, a chip vendor and a tool vendor can collaborate to add company-specific or chip-specific features to both of their products, without divulging the specifics of those features to competitors. Copyrighted, licensed, or trade-secret information can be passed between the chip and the tools without compromising security or violating license agreements. Only those in the know would understand what data, commands or responses are being passed. This provides an elegant mechanism for silicon vendors and tool vendors to add unique, proprietary debug technology, while basing the nuts-and-bolts of their debug solution on an open standard.

Nexus uses message-passing rather than binary instructions to transfer commands and reports between the debug tool(s) and the chip(s) being debugged. Message passing is used in computer networks because it allows different vendors' machines to communicate with one another. Likewise, Nexus's message-passing protocol leaves the door open to all comers. It's easily interpreted by PCs, workstations, programmable logic analyzers, handheld PDAs, and future development tools yet to be created. In fact, Nexus works equally well over a local-area network (LAN), wireless WiFi connection, or even a satellite link!

And Nexus scales up or down as performance requires. From as few as four pins to as many as 64, Nexus can handle low-speed, low-pin-count interfaces on inexpensive microcontrollers all the way up to 64-bit RISC processors with multiple execution units, coprocessors, and pipelines. From dribbling out data through a single pin to pouring it out over dual 32-bit buses, the same Nexus protocols work across the spectrum of microprocessors and microcontrollers.

Who Benefits from Nexus?

Financial accounting has been called "the dismal science," but debugging embedded systems has to be one of the least-rewarding jobs of all. Engineers and programmers are highly skilled, highly motivated, and highly compensated employees. Yet when they spend one-half of their professional time debugging systems, everyone's unhappy.

Surely anything that could reduce that 50% debug time would pay for itself. Not just in reduced salary expenses either, but in better morale. Engineers who are finished debugging sooner are ready for their next project sooner. Products that are finished sooner reach their market sooner. Debugging, *per se*, puts money in no one's pocket. It's wasted time from any point of view.

Benefit to Users

In any development and debug project there are three parties involved: the microprocessor vendor; the development-tool vendor; and their mutual customer, the product-engineering team. Nexus-compliant tools and chips help all three parties. The latter group, the engineering team, benefits both immediately and over time. The immediate benefit is in increased choice of chips and the tools that support them.

Currently, microprocessor chips that support on-chip debug features at all do so in a vendor-specific, proprietary manner. They're often supported only by tools from the chip vendor itself (a necessary support overhead that raises the price of the chips). This lack of choice means that chip selection often dictates tool selection, and vice versa. Rather than using the microprocessor chip they want and the debug tools they want, engineers are forced to accept a package deal.

This situation gets worse with every new project or product. Once again, engineers are forced to choose a chips-and-tools package, but now there's the added dimension of experience with the previous package. The engineers naturally won't want to abandon all the hard-won experience they gained with the last set of tools, so there'll be a strong incentive to remain with the same vendor, regardless of how appropriate or applicable that vendor's products might be. Simple human inertia plays a big role in designing follow-on products.

If those same engineers could retain their experience with their tools, they'd be much happier – and more productive. Divorcing the choice of tools from the choice of processors allows each design team to make better decisions: decisions they'll stand behind and willingly support. Conversely, if the engineers decide to stick with the processor but switch tools, they again can make that choice and stand behind it.

Finally, there's the added dimension of performance analysis, calibration and rapid-prototyping via Nexus, showing that the interface standard is used for more than just debugging. Use of Nexus as a calibration and measurement interface to microprocessor-based engine control units is a key reason that why automotive suppliers are participating in the Nexus forum. Automotive designers are already using Nexus-based tools for calibration (tuning) of engine control units, adjusting fuel delivery or ignition-advance while controlling an engine in real-time. Nexus tools pay dividends as performance-measurement tools, monitoring tools, or hardware-in-the-loop rapid-prototyping tools in addition to their valuable role in bringing up new systems.

Benefit to Tool Vendors

Tool vendors, far from being rendered generic and interchangeable, now have large new markets opened up to them. Currently, makers of in-circuit emulators, logic analyzers, and other debug paraphernalia are forced to design their tools for specific chips. Each microprocessor has its own proprietary features, quirks, and protocols. No two are alike, and every new product is essentially custom.

"For the first time, all chips will speak the same debug language."

Under the Nexus standard, much of that low-volume custom engineering would be rendered pointless. Nexus-compliant chips all use the same basic interface to/from the debug tools, and they all understand the same basic commands. For the first time, all chips will speak the same debug language. Now tool vendors can concentrate on adding (or cost-reducing) features that add some value, not on getting Vendor X's latest new debug trick to work. Time and effort spent on new products will add tangible, visible features instead of just re-implementing the same old features in a new and incompatible way.

Benefit to Microprocessor Vendors

Nexus benefits microprocessor vendors as much as the other groups. Each microprocessor vendor naturally wants to make its chips attractive and easy to use. In that quest, many have developed their own proprietary on-chip debug features, desperately filling a need in the industry. Those vendors continue to pour manpower and money into extending their debug features for future chips. Those efforts are laudable and the debug features are valuable and useful, but there's no need to keep reinventing that wheel – particularly if it's incompatible with everyone else's wheel. The same time and money that currently goes into developing proprietary features can still be directed at new features, but those new features will be better supported. Microprocessor vendors, like the debug-tool makers, will gain access to a broader market of supporting development tools. The more tools that are available, the stronger the chips will appear in the market. Microprocessor vendors can also, if they choose, abandon the costly development of their own proprietary development tools. Such tools are a necessary evil, not a profit center, and don't usually suit the vendor's business model or strategic goals. Yielding that tool development to third-party tool developers will simultaneously free up resources while improving tool support. Becoming Nexus-compliant doesn't mean giving up differentiation or branded features. Microprocessor vendors still have plenty of room to invent, differentiate, and innovate. Nexus doesn't enforce any particular features (apart from the very basic ones) and it certainly doesn't put an upper limit on what's possible or permissible. It only stipulates that the features that *are* there are implemented in a standardized way.

Moving Forward

The Nexus standard is evolving and growing, and its members are seeking new input and recommendations. The current draft standard (revised December 2003) is proven to work, as compatible chips and tools from multiple vendors are available now. Even the tough automotive market, as we have seen, has adopted Nexus wholeheartedly. At the other extreme, disk-drive manufacturers also rely on Nexus to speed development of their high-volume, low-cost, real-time systems.

Membership in the Nexus Forum is open to any interested party; it's not limited to particular vendors or particular interests. Those interested in participating in the Nexus specifications are encouraged to visit the Nexus Forum online at http://www.nexus5001.org/membership.html. Current members represent a cross-section of chip vendors, tool vendors, software companies, designers, customers, and users. Although the Nexus forum is not part of the IEEE, that body's industry standards and technology organization (ISTO) has been appointed by the Nexus Forum as the "publisher" for the Standard and as secretariat for the Forum's work.

Troubleshooting and debugging will always be with us, for as long as there are electronic systems and engineers to create and repair them. The unprecedented pace of change and improvement has delivered some wondrous things to industry, society, and the world economy. From clever entrepreneurs in their garages to large multinational corporations, innovation and improvement come from every corner, every day. But these inveterate creators are only as good as their tools. In most professions, the mark of a true craftsman is unparalleled skill with first-rate tools and the best materials. Software developers and semiconductor vendors create the best materials. Now it's time for these valuable developers to have the best tools.