Reliability Starts at the Core

by Jim Turley

Abstract: *Real-time systems are much more common than PCs but harder to develop. Because real-time products have to be reliable, predictable, and deterministic they place unusual demands on computers, microprocessor chips, and software. The toughest part is predictability – what will the system do in every conceivable circumstance? The new* ***fido*** *processor from Innovasic Semiconductor is designed from the beginning for reliability and predictability, even replacing traditional software components with innovative hardware right in the chip.*

Nobody said designing computers was easy. Especially when "computer" encompasses more than just the familiar PC, engineering workstation, or occasional Macintosh. In fact, only about 2% of all the computers made each year would even be recognized as computers by the average person. The other 98% are called "embedded systems," meaning the computer is embedded into some other product such as an automobile or an airplane. Even such mundane and everyday items as garage-door openers, dishwashers, antilock brakes, and video games all contain embedded computers. And these are even harder to design than the average PC.

Part of the reason embedded systems are so complicated is because they absolutely, positively, have to be on time. With the average PC, a little delay here or a little interruption there is no big deal. Who can tell if their e-mail arrives five seconds later than normal, or if a spreadsheet takes fractionally longer to load? PCs only guarantee "same day service" and they're not very punctual.

Embedded systems, in contrast, are often "real-time systems." They deal with real time and take punctuality and dependability very seriously. Whereas a PC can occasionally garble an e-mail message and nobody's surprised, an antilock brake system or high-rise elevator controller can't fail – ever. They can't even be slightly late. They can't miss deadlines, can't garble messages, can't misunderstand commands, can't succumb to viruses, and can't take time out to defragment the hard drive. An embedded system has to be so completely reliable that it becomes invisible.

Achieving that total reliability is what makes embedded computer engineering so tough. Those programmers can't get away with "good enough" or systems that work "most of the time." That's no way to run a railroad – or an engine controller, an elevator, a flight-control system, or any of a thousand other everyday embedded applications. For these engineers, predictability and consistency are the keys to making products that work – and that sell profitably.

Really Real-Time

Just to make things difficult, real-time embedded systems also have to juggle multiple tasks at once. Whether it's a simple dishwasher or a complex airport security

system, the microprocessor chip at the heart of the system must divide its attention between gathering data, processing the input, dealing with interruptions, and responding to the user, consumer, or operator. Typical embedded systems juggle dozens or hundreds of different tasks.

To keep everything straight, most programmers employ a real-time operating system (RTOS). In contrast to Windows, MacOS, or Linux, an RTOS pays close attention to time: to exactly when things happen, how long events endure, and in what order actions occur. Unlike the PC market, there is no dominant RTOS supplier. Instead, the market is extremely fragmented, with each vendor capturing perhaps 1% or 2% of the overall market. The differences among the RTOS products tend to be minor.

Entire magazines are devoted to the topic. Task scheduling, priority handling, time-slicing, interrupt latency, threading, deadlocks, scheduling algorithms, and other topics endlessly fill the pages of such magazines year after year. Conferences and seminars are dedicated to the fine art of separating tasks into threads and managing those threads in software. Clearly, this is no trivial undertaking. It's an arcane science that baffles even experienced programmers, to say nothing of those who are new to the art.

If real-time task management is so important yet so complex, why does everyone do it alone, and do it differently? Why isn't this a feature of the microprocessor chip itself instead of add-on software?

There are a few reasons. For one, most microprocessor chips weren't designed for real-time embedded systems, they were designed for computers. Every chip maker wants (or wanted) a piece of Intel's PC business and so almost all high-end processor chips today are failed PC processors. Second, most microprocessor designers are trained in computer science, not embedded systems. Very few academic courses are available in real-time programming; most professors (and their graduates) studied conventional computers. Finally, when task-management features have been added to microprocessor chips, they've been ignored. Back in 1986, Intel's 386 chip handled simple task management in hardware – but Microsoft's operating systems did not (and still do not) use those features, so they were never exploited.

Shifting Responsibilities

Enter **fido**. The **fido** (flexible input, deterministic output) processor chip from Innovasic Semiconductor takes Intel's 386 task-management hardware and adds 20 years of technological advancement, tailored for embedded systems. With **fido**, hardware replaces part of the RTOS. The chip itself juggles multiple tasks without the need for so much additional software. The **fido** processor builds in hardware what many RTOS companies supply in software: task management, context switching, priority handling, state changes, interrupt management, and timeout features. That means **fido** complements existing RTOS products, minimizing duplicated effort.

Task performance becomes absolutely deterministic with **fido** because there's no ambiguity about how it handles tasks or switches among them. It stores all context data

internally instead of using external memory stacks. Besides being quicker, **fido's** hardware task-management completely eliminates the sources of unpredictability from:

- Delays due to bus arbitration
- Uninterruptible sections of code
- DMA transfers or transfers by other bus masters
- Cache misses
- Stack push/pop operations
- Memory page faults
- Memory-management translation overhead
- MMU page table misses
- …and more

As an example, a typical microprocessor might have to *wait* to acknowledge an interrupt, then *wait* to push its register contents onto the stack, then *wait* while its data cache handles the cache miss, then *wait* while its MMU processes a page miss, then *wait* for a DMA transfer to complete, then *wait* through bus arbitration, then *wait* while the stack push operation completes, then *wait* while the new state is popped off the stack, then *wait* while the entry vector is loaded, then *wait* while the instruction cache processes the cache miss, then *wait* while the new code loads, and *then* begin actual processing of the new task. Without exaggeration, this process can take hundreds of bus cycles and thousands of processor cycles.

The **fido** hardware completes the entire process in one cycle.

With **fido**, there's no pushing or popping (also called filling and spilling) of register contexts on every task switch. Instead, **fido** keeps duplicate sets of registers internally and simply switches between one set and another. The whole process is just part of the single-cycle task switch and because it requires no external memory it's completely predictable and reliable and not subject to the vagaries of external RAM, DMA, wait states, or arbitration. Task switches couldn't be simpler or more predictable.

Task switches can occur at predefined times, or when triggered by interrupts, or when one task has stalled and another is ready to run. Tasks can be prioritized and they can be given maximum time limits. Either way, **fido** handles all the interrelationships among tasks, switching back and forth as required. If there are no tasks to run, **fido** even shuts down, putting the whole processor in a low-power state – all automatically and with no software overhead.

Not Just Another Bug Hunt

Surveys show that when embedded programmers choose a new microprocessor the debug features are more important than performance, price, or power consumption. In fact, debug features rank as the single most important part of the chip. There's good reason for this: programmers spend more time debugging than they do writing code. Debugging is the least interesting, yet most time consuming, part of any development

project. It's no wonder that debugging has become the #1 concern among working engineers – and their managers.

The **fido** processor already has a leg up in debugging because of its built-in task switching, which eliminates one of the most complex sources of subtle and hard-to-fine bugs. But there's much more.

Of course, **fido** starts with the basics. Programmers can single-step the chip as it tiptoes through their code. In between steps the programmer can examine and/or alter the state of any internal register, including the on-chip peripherals. They can even switch between tasks even though the situation may not have called for a task switch. All this is done through the standard JTAG interface using standardized and low-cost instruments or a PC.

Some bugs only occur at full speed, so **fido** also includes a trace buffer that logs program branches and related information for post mortem evaluation. The trace buffer can be any size the programmer likes and located any place in the system's memory – it can even be transmitted over a network to a remote site.

Breakpoints and watchpoints are also standard equipment, but even these are better than usual. Both the breakpoints and the watchpoints are context-aware, so they only trigger when the desired task is running, not just any time the processor trips over a certain address or a certain instruction. This feature alone can save days of frustrating debug time that would be wasted trying to get the breakpoints to trigger at the right time.

Better still, **fido's** breakpoints can be trigger-chained so that one breakpoint or watchpoint enables the next one. This allows the programmer to create elaborate trigger conditions rather than simply monitoring addresses or instructions. This feature is normally only found in expensive logic analyzers; now it's part of every **fido** processor.

The best debug tool is to remove the source of bugs. Another way in which **fido** does this is through its deterministic code "stash," a cache-like storage area for time-critical code. Caches are common in computer systems and computer processors because they improve performance – most of the time. Caches are inherently unpredictable and nondeterministic, and so many embedded designers disable them (or choose another chip with no cache). With **fido**, programmers enjoy the advantages of a cache without the disadvantages. They know exactly what's in the code stash and what's not, so they can predict performance, every time.

Because **fido** has so many debug features built in there's less reliance on software tools. For example, there's no need for a resident debug "monitor" program running on the chip – a program that can crash just with it's needed most. With **fido**, debug features always work, even if everything else crashes. After all, that's exactly when debuggers are most important.

Conclusion

There's an axiom in the microprocessor business: silicon is cheap, it's the software that's expensive. Paradoxical as it sounds, semiconductor transistors are nearly free but the manpower required to create software is very costly. Moore's Law has given the industry an embarrassment of riches; saving transistors is counterproductive when you can already manufacture bigger chips than you can design. And processor chips are dead reliable. Computers rarely fail because of a silicon fault; they fail because of software bugs or crashes.

In contrast, software is time-consuming, labor-intensive, and drastically unreliable. Less software is almost always a good thing. Engineers and developers already spend more time debugging code than they do creating it, so there's clearly a need for better and more reliable development methods. The engineers themselves recognize the need: they rank debug tools as the #1 criterion when selecting a microprocessor chip. Performance and other hardware considerations rank much lower. Obviously, they want what their bosses want: to produce good products faster.

The **fido** microprocessor gives engineers and programmers what they want. It eliminates a third-party operating system, including its cost, its learning curve, its memory requirements, and its unreliability. It manages tasks and contexts with built-in hardware that delivers determinism and predictability, all in a smooth and invisible manner that doesn't eat up processor cycles or interfere with application code. And it gives unprecedented access into the inner workings of the processor for quick and painless debug sessions. They say every dog has his day and it's looking like the time has come for **fido**.

30

Jim Turley is an acknowledged authority on microprocessors, embedded systems, and semiconductor intellectual property. He is the author of seven books, owner and principal analyst of *Silicon Insider*, the former editor of *Embedded Systems Design* and *Microprocessor Report*, and is a regular speaker at industry events. He is frequently quoted in *The Wall Street Journal*, *New York Times*, *San Jose Mercury News*, and appears regularly on television, radio, and Internet broadcasts.

Jim lives and works in the Monterey Bay area, records books on tape for blind engineering students, and is a competitive race car driver. He has a talented, accomplished and stunningly attractive wife, two overachieving children, and an apparently brain-damaged dog. He can be reached (Jim, not the dog) at www.JimTurley.com or at 831.375.8086.