REMOVE THE RISC FROM YOUR EMBEDDED DESIGN

WHITE PAPER

Abstract

Embedded software development is expensive

Silicon gates are cheap

These are two fundamental facts; they are especially true in embedded industrial systems, which have markedly different computing requirements than those of other markets, particularly consumer applications. Industrial applications are much more difficult - they are tightly time-constrained, need perfect predictability and are deeply embedded. Sometimes they are safety-critical. Any industrial microcontroller needs to communicate over a varied range of protocols, and must support product life cycles that are an order of magnitude longer than the PC space.

Unfortunately, most current industrial computing solutions (including those built on ARM®) were originally architected for the PC world, requiring additional efforts to solve industrial problems. Long software development and debug times, unpredictable performance and complex Operating Systems are a few of the headaches caused by force-fitting these architectures into industrial systems.

Innovasic Semiconductor's new family of microcontrollers, **fido™** (Flexible Input Deterministic Output), has been designed from the ground up for embedded industrial computing requirements. Combining the best of tried-and-true solutions with innovative new features, it provides a welcome relief to system designers and programmers. In contrast to ARM-based and other current solutions, it has predictable performance, reliable time-critical response and debug features, which make field deployment easy. These features lower processing requirements and reduce the power consumption, clock rate and design complexity.

The **fido** architecture includes the ability to program the peripherals, so that any of a broad selection of protocols can be supported. There is no need to buy multiple part numbers to handle the varied communication requirements – **fido** supports many interface configurations with a single part number.

Industrial systems are sometimes in production for decades. While other manufacturers regularly discontinue semiconductor products, leaving the industrial customer with only expensive options, Innovasic Semiconductor has built its reputation as a company that solves obsolescence problems.

Indeed, one reason that Innovasic was able to create the **fido** architecture is because we've designed replacements for obsolete integrated circuits for many years for industrial customers - including the 186/188, H8, Z88, 6805 and the 8051. We've seen the complexities, mistakes and limitations of various

architectures, allowing us to more clearly see the right way to architect an industrial microcontroller.

THE fido SOLUTION

Innovasic Semiconductor's **fido1100** (the first product in the **fido** family) was designed from the ground up for industrial systems. This fact alone distinguishes it from the pack of competitive solutions that originally targeted the personal computer space or consumer markets. In keeping with "software development is expensive - silicon gates are cheap," a few of the areas improved upon for industrial systems include time-critical execution, predictable performance, embedded debug and interface flexibility.

Each industrial system is sold to customers who impose their own set of requirements on it. Some want extensive Ethernet support. Some use CAN, others Profibus and some even have their own proprietary protocols. Providing different hardware configurations for each leads to increased inventory and logistics costs, yet all required protocols must be available.

The **fido** architecture includes a Universal Interface Controller (UIC) that supports many standard interfaces. Each interface supported by the architecture is provided as a standard library element that can be compiled right along with the rest of the application code. Proprietary protocols can also be supported.

TIME-CRITICAL EXECUTION

A manufacturing line requires precision. The design of programmable logic controllers (PLCs), for instance, requires that a set of inputs be read. The outputs are then set based on the inputs as well as other factors. This must happen on a predictable basis. If it does not, the entire factory operation is at risk. This requires that a microcontroller within an industrial automation system be aware of time. Unfortunately, most microcontrollers are not designed to handle time constraints – leaving it up to the software[i].

Typical microcontrollers spend a large percentage of time simply handling the overhead associated with switching and managing tasks. The **fido** architecture has several features that are specifically designed to reduce this overhead. Figure 1 shows the differences between a typical microcontroller and the **fido** architecture.

In a typical microcontroller, there is no inherent difference between real-time, safety-critical and user interface or user application code. The difference is determined by additional software – the Real-Time Operating System (RTOS). The RTOS manages when code is run, tasks are switched, what the relative priorities are and the entire associated overhead. The problem is that the system designer has little control over how the RTOS manages these tasks.

In the **fido** architecture, many of the real-time tasks are handled within the chip itself. The architecture allocates the priorities to one of five space- and time-partitioned hardware contexts. In addition, there is less software overhead, because most of the overhead associated with real-time operation has been moved into the chip. User interface and low-priority code cannot interfere with real-time or safety-critical code.

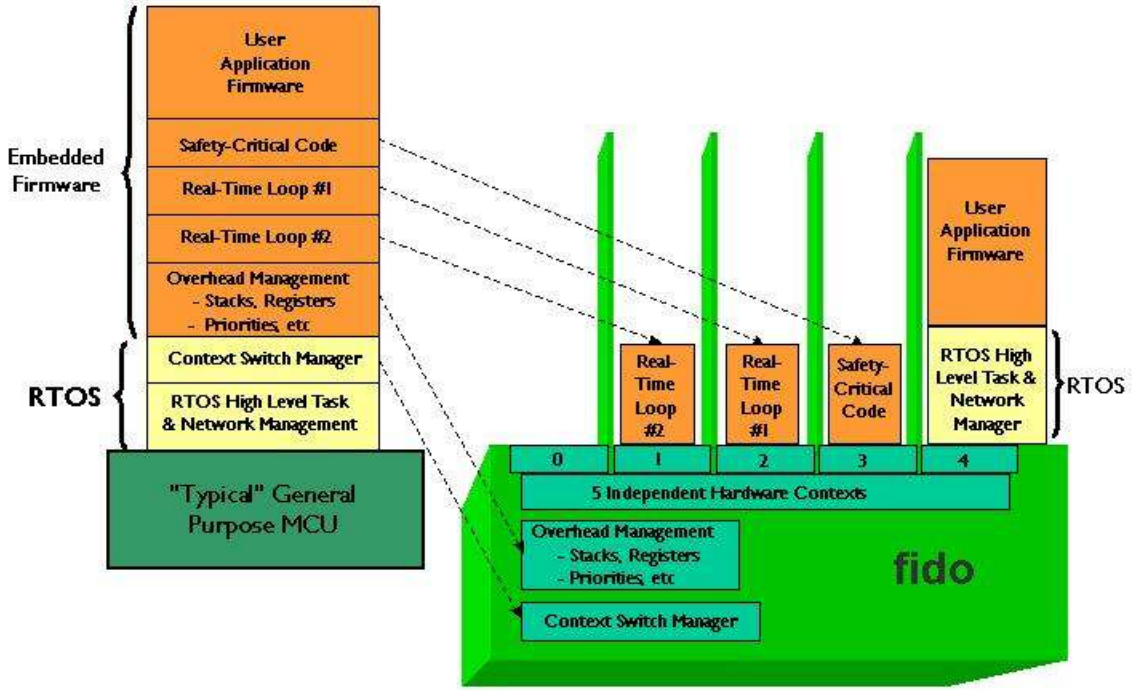

Figure 1. fido Family's Unique Task Partitioning and Management

PREDICTABLE PERFORMANCE

In a PC system, it doesn't matter if a task begins in one-tenth of a second one time, then two-tenths the next. There will be no noticeable difference to the user. Such a difference in an industrial application can be catastrophic. Tasks such as measuring important control parameters, controlling outputs and motion control must be done predictably, beginning at exactingly regular intervals and taking the same amount of time to execute. Latency, the time it takes to begin a task once it has been requested, must be within a tight tolerance. Most microcontrollers do not have a predictable latency. So a task may have widely differing execution times between instances. This problem, called jitter, is a particular problem in control systems. The **fido** family addresses these shortcomings with a novel context- and time-aware architecture that can switch contexts in one clock cycle.

CONTEXT-AWARE

The **fido** family is designed so that all five contexts are fully ready to be run when needed. All registers are fully replicated.

No pushing or popping of registers onto a stack is needed, but that's the easy part. The chip keeps track of the context that is being processed. Typically, when an interrupt occurs and a context is switched, a large amount of overhead processing is needed to stop the peripheral processes that the previous context had started. Then it must set up the peripherals and buses to handle the task that is about to be executed.

In the **fido** architecture that overhead is eliminated, thereby enabling switching of contexts within one clock cycle. One interesting benefit falls out of this. The hardware also knows when nothing needs to be executed. The processor then goes instantly into auto-sleep mode, again with no code intervention. It wakes up instantly upon receipt of an interrupt.

The context-aware Memory Protection Unit (MPU) manages the memory requirements of the different contexts without software overhead. The MPU, typically at boot time, is programmed so that certain blocks of memory are associated with specific contexts. A block of memory may be read-write to one context, read-only to another and unavailable to a third, ensuring that blocks of code cannot interfere with each other. This is important in safety-critical applications. The designer needs to know that the code controlling a moving saw blade or running a product line won't be corrupted by the user interface.

TIME-AWARE

In a time-critical system, it only makes sense that the microcontroller is aware of time. Yet current architectures support this weakly at best. In the **fido** family, time is built in. Time partitioning of tasks is done easily, with only minimal overhead. Each context can be allocated a maximum amount of time it can run. A context timer is then reset upon entrance to a new context. If a context overruns its allotted time, a fault is generated and the context will automatically be switched. In this way scheduling can easily be generated and protection against uncontrolled tasks is achieved.

Having context-aware and time-aware hardware handle much of the overhead means that less software needs to be written and debugged, saving time and money. Moreover, it frees software designers from writing mundane utility code and lets them concentrate on software that adds value to the system.

HANDS OFF EMBEDDED DEBUG

Industrial systems are deeply embedded in customer applications. On a factory floor there is little time or room for engineers and technicians to hook up monitoring equipment, debug code or measure performance. Physical access to the equipment is limited. Yet the designers of PLCs or other industrial systems must be able to do all of this without interfering with the customer's operation.

This issue is exacerbated by the code instrumentation problem. When you write code to find out how a bug occurs, the nature of the bug changes.

It may even disappear, leading to frustrated engineers and customers. In the debug of any system that includes hardware or software, there are two key features that must be in place to easily find, implement and verify the fix - controllability and observability. That is, the system must be easily and predictably controlled to steer the system toward the bug. Data also must be available so that the programmer can clearly identify the problem.

CONTROLLABILITY

The **fido** microcontroller is more accurately controlled than any other current competitor. Indeed, it has the capability of true single step mode through the Joint Test Action Group (JTAG) port. Other solutions are not capable of true single step hardware. They run software to emulate single step. True single step in a board design provides definitive control of the system - you can control and see what happens one instruction at a time. If desired, you can change the state of a register prior to clocking the system the next time, or change the state of a peripheral, one bit at a time, with easy-to-use tools.

Further, the JTAG emulation is context-aware. That is, there are register sets that are switched in and out depending on the context. Imagine being able to switch, without changing the state of the external hardware or running lots of extraneous software, to a piece of code that is not working as intended. No setting up unrealistic scenarios to make it switch, no changing the state of other parts of the system to make it switch, simply doing it within a single clock cycle.

OBSERVABILITY

The single step capability not only provides controllability, but observability. At each instruction, you can see the state of all the registers in the microcontroller, the state of the outputs and inputs, even the state of the peripherals.

Aside from the true JTAG emulation, the **fido** architecture has the capability of non-intrusive event logging. When debugging code running in real-time, the engineer wants to see how the code behaved as it went through its paces. The **fido** family has the capability to generate a trace buffer, with all the branch occurrences and data needed to trace the software's path. In addition, the trace buffer can be allocated anywhere in the address space – even sent over an Ethernet port. Thus, if the system is designed properly, an engineer can set watch points and break points, set the system to run, and then view the results, all remotely from the convenience of their office. Much of the debug can occur immediately, no waiting to jump on a plane and get to the customer's site.

Break points and watch points are common in microcontroller architectures, but because the hardware of the chip has no information about which context the code is running in the break points cannot be context-aware.

In the **fido** family, this comes for free because the hardware has built in context support. Thus, the break and watch points are context-aware, providing additional controllability and observability.

In typical systems it is difficult to determine just how your software is performing. The features of the **fido** family make it much easier to profile your performance. How much time is each critical context taking? Have the priorities been set properly? Again, the hardware of the **fido** family supports this with minimal code instrumentation. These features support embedded system debug without the need for emulators or logic analyzers.

MEMORY AND SPEED CONUNDRUM

The ARM processor is very successful, and rightly so. It was a fine, well thought-out architecture. However, RISC's fundamental assumption is that memory bandwidth is equal to the processor bandwidth[ii]. This was true years ago – but no longer. Today's processors clearly outpace their memory counterparts. Meeting strict industrial timing requirements pushes today's RISC architecture to its limits. To understand the power of fido, we should first look at some history behind the memory and speed relationships for RISC solutions.

MEMORY SPEED AND SIZE

Going back to the early RISC days, processor speeds quickly began to outstrip memory speeds. Since a RISC architecture executes a single instruction per clock cycle that meant that the processor was waiting on the memory, taking away from the advantages of RISC. Today, the problem has been further exacerbated by the adoption of FLASH memories. FLASH access is slower, but the benefits of programmability and non-volatility far outweigh that downside. Unfortunately for a RISC processor, FLASH is now widely used. Another disadvantage of a RISC architecture is larger code sizes – it takes a larger number of instructions to perform a given task[iii],[iv]. Larger code size translates to more data that needs to be transferred into the microcontroller, and since the internal processor clock speed is faster than memory access times it causes a bottleneck.

Two events in RISC history illustrate the inadequacies of RISC instruction sets. First, the Coldfire® processor was originally designed to be RISC-like by reducing the 68K instruction set. Yet the Coldfire instruction set has since grown substantially from its original size as the device has been put to practical use. The other was ARM's modification of its architecture to make the code denser - a second instruction set, called the THUMB, was introduced.

The THUMB instruction set, while an improvement over ARM for embedded applications, runs slower and is still based on the ARM RISC processor, loading multiple ARM instructions into a single THUMB instruction.

To attempt to address the fundamental memory, speed and code size issues, the concepts of cache memory and faster clocks were introduced.

CACHE MEMORIES

One way to work around external speed issues is to store portions of the code in an internal RAM that is as fast as the processor. Thus cache memory was born. A complicated algorithm predicts which program code will be needed, then loads it into cache. For many applications this works well. However, there are problems with cache in industrial systems. The foremost of which is that cache memory is inherently unpredictable. If the code you want happens to be in memory, it works well. But there is no possible way to guarantee that – if it's not there the processor must wait until the code is loaded into cache from slow external memory. If it happens in a time-critical piece of code, nothing will happen until the code is loaded into cache. If the code is already loaded into cache, the code starts execution quickly. The result is jitter –the irregularity and unpredictability of the time it takes to start a task. One way to solve the problem is simply to turn the cache off – the jitter goes away, it becomes predictable and then it is simply a matter of latency – how long does it take to execute a task. Because this involves executing code directly from external memory, it means fetching instructions at the slow memory clock rate, precisely the problem caches were designed to solve.

FASTER CLOCK RATES

Alternatively, it may work to simply speed up the system enough so that the jitter becomes acceptable. This requires turning up the clock rate of the processor. There are clear disadvantages of a faster clock rate – the most obvious is higher power consumption, but it also increases electromagnetic interference, and creates a more difficult board design. Also, the inherent unpredictability of a system means that it is difficult to tune the clock rate to an optimal analyzable point. Consequently, the clock rate must be much faster than would be needed to meet performance if the system were predictable. The end result is that we have industrial systems that are drastically over-designed, and thus over-powered and unnecessarily complex, simply to get over the memory bottleneck problem.

fido's DETERMINISTIC CACHE

As is obvious from above, the use of cache results in unpredictable system behavior. Many real-time designers simply disable it or create trade-offs in speed and power dissipation to compensate for it. The **fido** family solves this problem by supplying a Deterministic Cache. Under software control, typically at boot time, the critical pieces of code are stored in the Deterministic Cache and remain there. When that code needs to be executed, it is there each and every time. No cache misses and no unnecessary clock speed increases. As with many other features of the **fido** architecture, the deterministic cache (Figure 2) is context-aware.

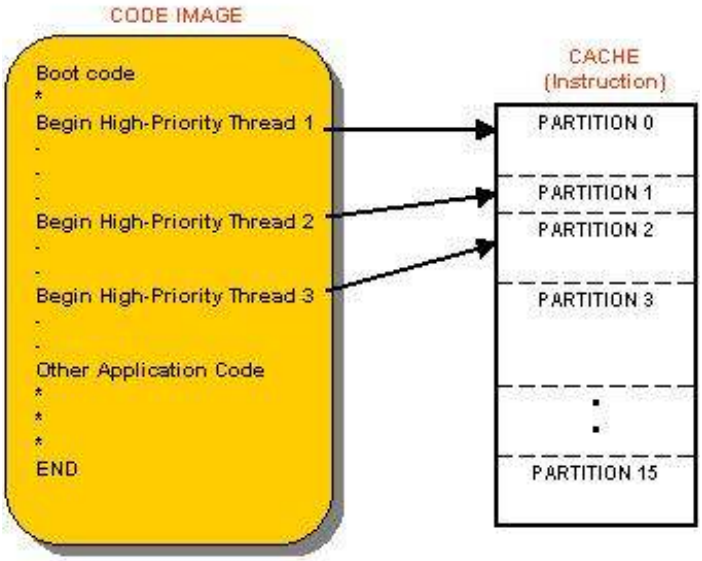

Figure 2. Deterministic Cache Concept

A NEW ERA OF PERFORMANCE

In a tight control system, there are likely very strict time constraints on maximum latency of an interrupt. To meet strict timing requirements the software developer might spend a few weeks organizing the code so that this interrupt is handled acceptably, from both the hardware and an operating system perspective. So the choice is: spend hours fiddling with the code or increase the clock rate and thus the power consumption and design complexity.

So what does all this mean? Let's look at the latency and jitter performance in an ARM-based microcontroller. In Figure 3 the interrupt response time was measured in an industrial system with an ARM7TDMI processor (widely found in today's embedded platforms) running an RTOS. The first (darkest) bar in each group illustrates the interrupt response time with the cache enabled.

The second shows the cache disabled, and the third (white) bar in each group represents the performance of **fido**.

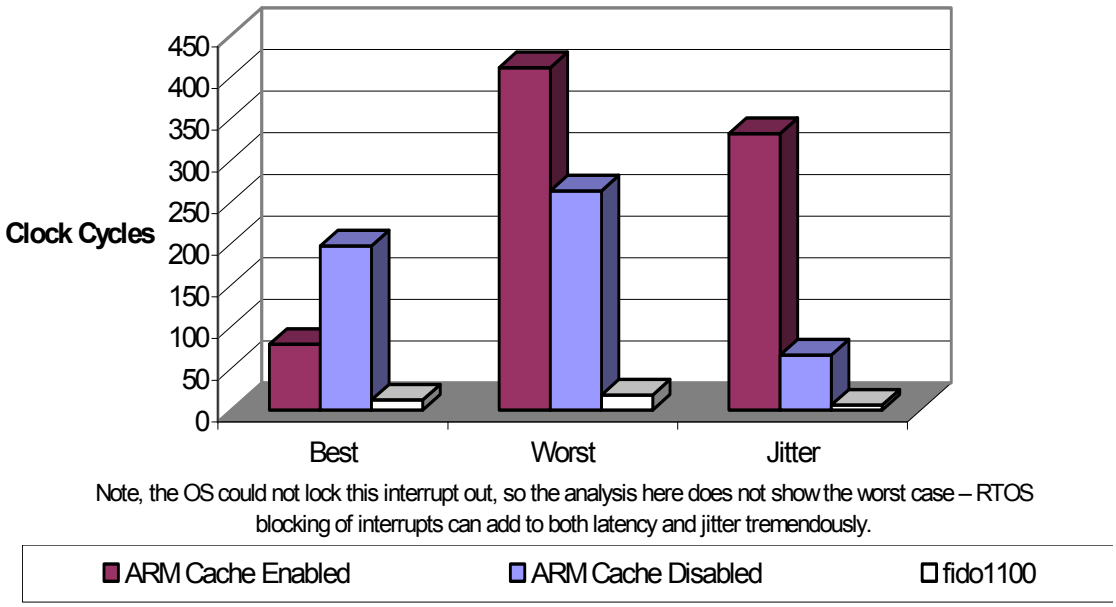

Figure 3. fido1100 Performance Comparison

With cache enabled, the best-case latency can be acceptable for some applications, providing the clock speed is high enough. The worst-case latency, however, is 5.2 times the best case. This is jitter in action – a jitter of 420 percent. In many cases, this type of jitter is unacceptable – in these cases the user will simply turn the cache off. What happens then? As we see in the cache disabled (light colored bar), the minimum latency increases dramatically, as we would expect; there is a tradeoff in the architecture between latency and jitter. One might expect the jitter to be nearly zero – but it remains a very high 33 percent. Why would this be? The main reason is jitter in the instruction set itself. Depending on which instruction the ARM is executing, the interrupt time will still vary widely. So even with the cache turned off, the latency and jitter are high – the device is still unpredictable.

The **fido** family's instruction set is a unique combination of tried-and-true technology and a novel approach. The basis for the instruction set is the CPU-32 (68K heritage), allowing well-proven existing tools to be used in firmware development. However, some important improvements have been made. The efficiency of the instruction set has been dramatically improved, reducing the number of clock cycles needed to execute most instructions. This, combined with the Deterministic Cache and single clock cycle context switching, provides the absolute minimum in jitter and latency.

The context and time-aware aspects of the **fido1100**, as well as the efficiencies of the instruction set, redefine the performance for embedded solutions.

CONCLUSION

Designing tools specifically for a use results in more efficiency. A mechanic has a set of tools that are each designed to make a specific job easy. Would a mechanic use one wrench to loosen all bolts? Of course not.

The **fido** microcontroller is the industrial system designer's special-purpose wrench. Unlike all other microcontrollers, it is designed with industrial applications specifically in mind. The ground up architecture addresses critical needs of industrial systems including time-critical execution, predictable performance, embedded debug and interface flexibility. The architecture also takes into account the practical, providing software programmable interfaces to eliminate the "sea of part numbers" mentality of other semiconductor suppliers.

The **fido** family outperforms conventional microcontrollers in the industrial market – unprecedented real-time performance, enviable debug access and a long-life supply guarantee. The results are lower software development costs, faster time-to-market, and increased profitability for the system manufacturer – all achieved by shifting the emphasis from software to silicon.

---

[i] Lee, Edward A. "Absolutely, positively on Time: What Would it Take?" *Embedded Computing*. July 2005.

[ii] Wilson, Ron. "The RISC That Did Not Pay Off." *Embedded.com*, July 11, 2005. <http://www.embedded.com//showArticle.jhtml?articleID=165701313>.

[iii] Turley, Jim. "RISCy Business."*Embedded.com*, February 5, 2003. <http://www.embedded.com/showArticle.jhtml?articleID=9901018 >.

[iv] Phelan, Ron. "Improving ARM Code Density and Performance." June, 2003 <www.arm.com>.