

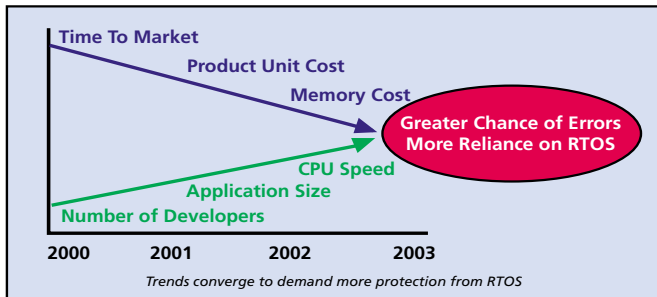
# INTEGRITY™ RTOS Uses Intel® XScale™ Microarchitecture to Deliver High Reliability

By Green Hills Software • [www.ghs.com](http://www.ghs.com)

## The Problem

My cell phone can play games, surf the web, remember phone numbers, ring in a variety of tunes, and, oh yes, it also can find a nearby cell site and enable me to talk to virtually anyone on the face of the earth. It does this in spite of atmospheric interference, competition for airwaves from millions of other phones, and reliance on its little self-contained battery. Of course, it uses a 32-bit microprocessor with gobs of memory and a real-time operating system to get all of this done. But, it still only costs about \$100. Amazing. And that's just today; what will phones be like next year? And beyond? If history is any indication, phones, along with dozens of other consumer and industrial devices, will get smarter, faster, and even more affordable. This is because the factors that influence their intelligence, speed and price, namely processor and memory cost, are declining. But, as cell phones get more and more powerful, will they remain reliable?

The demand for more sophisticated applications is strong, fueled by consumer interest and competitive pressure. Sophisticated applications require larger programs, hence more memory and faster processors. As memory gets less expensive, and processors get more powerful, application programs can become more feature-rich in order to achieve competitive advantage. The phone that doesn't evolve will not sell.



Today's application programs are very large, sometimes consisting of millions of lines of source code; in most cases, multiple programming teams combine to produce a final application, with each team tackling a separate function. For example, one team might program the address book, while another team programs compression/decompression of voice and data. In the end, all of the separately developed applications are linked together to form the finished product. But before the individual portions of the application can be joined together, they must be individually debugged.

Debugging 100% is impractical, if not impossible. Thus, bugs will exist in fielded programs. It has been long held that no 'non-trivial' program is ever bug-free. Testing can make sure that 'most' bugs are corrected, and that the product functions properly 'most of the time'. But the most insidious bugs are the ones that defy

testing and only occur under the most specialized circumstances. These bugs are only found when real-world events line up in such a way as to produce a certain combination of events. Race conditions, for example, will only appear in worst-case scenarios, difficult to construct in the testing lab. Instead, they emerge while you're talking to your customer who is just about to give you his home phone number. The larger the programs, the more bugs that will remain.

Without some form of protection, a single bug could cause total system failure. A bug in the address book program could easily wipe out memory locations elsewhere in the call processing program causing a connection to be dropped. Can anything be done to protect against this seemingly inevitable consequence? Yes; in fact the answer lies inside every Intel® XScale™ microarchitecture, unfortunately dormant in most cases.

## The Solution

The solution is the Intel XScale microarchitecture Memory Management Unit (MMU). The MMU associates certain pages of physical memory with a virtual 'address space' that a program, or Task, is able to access. A Task can only access physical memory that has been 'mapped' to its virtual addresses; it's not possible for it to see any other memory. Using the MMU, the system can prevent the address book program from overwriting the memory of the call processing program, or of any other program for that matter. That way, each program just has to watch out for itself, and not worry about being 'sabotaged' by errors elsewhere in the system. What's more, the more critical programs can be debugged more extensively than less critical programs. This way, a given amount of time and money can be allocated most effectively, with more time devoted to debugging the most critical portions, while less time and money is spent debugging the less critical portions of the system.

Automatic detection of the most serious bugs by the MMU provides protection against their effects and identification that can enable correction with minimal difficulty.

The Intel XScale microarchitecture MMU provides a hardware mechanism that can detect attempts to read/write/execute outside of one's assigned address space. This is the most serious type of bug, because if not blocked by the MMU, it can easily cause widespread havoc throughout the system, eventually causing catastrophic failure. But before complete failure occurs, it's anyone's guess what damage such a bug can cause along the way. For instance, such a bug could cause a call to be switched to a different person who all of a sudden is talking to Pizza Hut instead of his boss. Or, vice versa! Automatic prevention of such errors is not only possible, but virtually free of overhead as well. So why don't all programs take advantage of this? Surely it's a desirable feature, and if it's free, why not use it?

## The RTOS is Key

The answer lies within the Real-Time Operating System (RTOS) that controls the hardware. It is the RTOS that must set up and turn on the MMU. Sadly, the most popular operating systems have been around for so long, they were invented before MMUs became widely available. Thus, they simply use a 'flat memory' model that ignores the MMU and allows any program to access any area of memory in the whole system, setting the stage for disaster. By contrast, the INTEGRITY™ RTOS from Green Hills Software fully supports the Intel XScale microarchitecture MMU, and uses it to prevent programs from accessing any memory outside of the specific region (Address Space) to which they are assigned. This not only prevents one program from causing another to malfunction, but also automatically identifies attempts to do so, making correction much easier.

### Watertight Compartments Keep Programs Safe

Ever since the sinking of the Titanic, the concept of 'watertight compartments' has been an essential technique in shipbuilding. The Titanic sank because a 'small' gash in its hull from a collision with an iceberg allowed water to completely fill the ship. Ever since, watertight compartments have been used to seal off the rest of the ship in the event of a breach of one part of the hull. One compartment will flood, but the ship will continue to float, enabling repair and saving lives and lots of money. The same approach to programming is available, simply by using an RTOS that fully supports the system's MMU. It's difficult to understand why anyone would design a system without such protection, especially since it comes for free.

## The Benefits

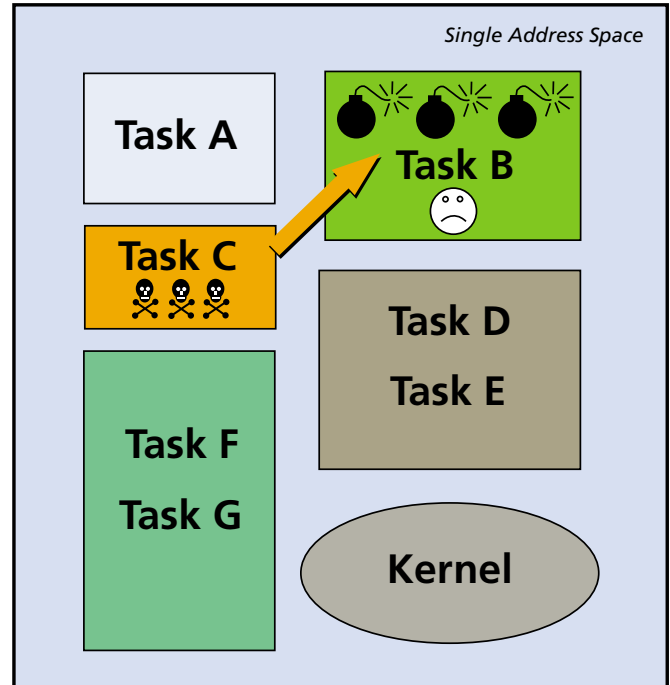
Of course, there will still be bugs in fielded systems. But, if critical programs are fully debugged, then at least those programs can be spared the effects of bugs elsewhere in the system. And, if debugging investment is focused on the most important programs in a system, then system reliability will be maximized even as other, less important, programs are fielded with lesser degrees of debugging.

This approach provides an additional benefit—faster time to market. Consider that a product can only reach market once it has been debugged to the point that it will not fail. Using the right RTOS, only failure-critical components need be 'perfect'. Lesser degrees of debugging can be tolerated in other areas, since no failure there can cause total system failure. Without the right RTOS, every program must be tested to 'perfection' before the system can be considered 'failsafe'. Thus, the right RTOS enables a product to get to market faster, and to perform reliably once there.

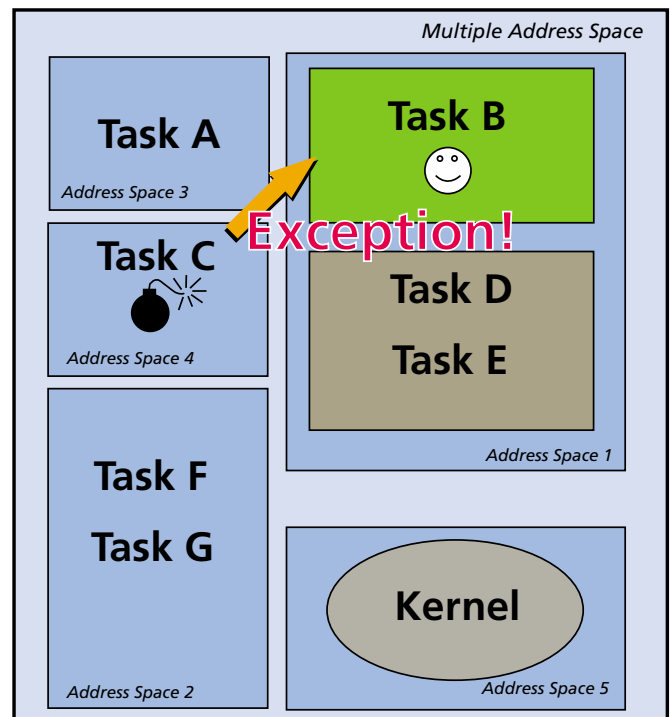
## Beyond Simple Protection

There's more. The right RTOS can provide even more protection, and enable greater degrees of reliability in areas other than simple memory protection through the MMU. What else could go wrong? What about 'denial of service'? Denial of service is the result of one part of the system 'hogging' system resources to such an extent that other parts of the system can't operate properly. For example, suppose one program 'hangs' in an endless loop? Other programs might never get CPU time. Or perhaps one program allocates too much memory, starving other programs' needs for their own memory, preventing them from being able to run as planned. Denial of service can be caused by a programming error (bug) or intentionally by a virus or hacker through external access to the system.

What about non-deterministic performance? Most real-time sys-



In a Single ("flat") Address Space model, a bug in any task can not only cause that task to fail, it also can cause another task to fail by writing across task boundaries. There is no limit to the damage such a bug – indeed, any bug – can do, even including causing total system failure.



In a Multiple Address Space model, each task is only able to access its own address space. The MMU enforces such access, and any attempt to access anything outside of its own space will cause an EXCEPTION, and the access will be denied. This prevents "innocent" tasks (Task B, in Address Space 1) from being "attacked" by errant tasks (Task C) in other address spaces (4).

tems have stringent response requirements (or else they shouldn't be called 'real-time'). The RTOS can present a problem to the system designer who must guarantee response within a bounded maximum. What if the RTOS is not 'fully deterministic'? This means that it cannot guarantee a specific maximum time to perform a certain function, such as responding to an interrupt. It does no good to have great 'average latency', if every once in a while the response is just not acceptable. What's needed is a guaranteed maximum latency that comes with full determinism.

```
int limit = CALL_LIMIT;
int calls = 0;
...
void spawn_servers(void)
{
    while (1) {
        fork();           /* for illustration: spawn a server process */
        calls++;
        if (calls > limit)
            break;
    }
}
```

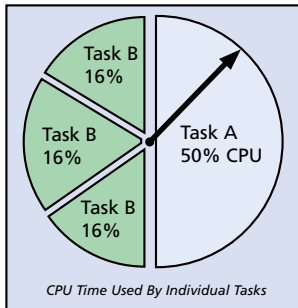
In this example, a bug could cause "limit" to be extremely high, effectively making the loop execute so many forks as to kill the system.

### Guaranteed Resource Availability

A program needs resources to operate. It needs memory for its instructions and data, and it needs CPU cycles to execute those instructions. What if it doesn't get the memory it requires? Well, it can't operate correctly. Perhaps it will not be able to store the next incoming message, or save the next phone number or name, all of which require memory. If there isn't any available, because some other program used it all up, then the requesting program cannot perform its intended function.

Likewise, what happens if a program doesn't get the CPU cycles it requires? Suppose one task starts spawning 'subtasks', but has a bug and just keeps spawning task after task after task. In most systems, each subtask will get CPU time in a 'round-robin' manner, sharing available CPU cycles with all other tasks at that priority level. An 'innocent' task, at the same priority level, thus can be starved for CPU time. Perhaps it can't decompress incoming data; or perhaps it can't find a free frequency for transmission. All because the CPU is tied up doing something unintended by the system designer.

How can this be prevented? By allocating CPU time for spawned tasks from the budget of the spawning task, rather than from the system's resources (see diagram below). This provides every program with guaranteed resources, both time (CPU cycles) and space (memory). The right RTOS can provide these guarantees, but not very many RTOSes actually do this. INTEGRITY does; it guarantees that a designer-specified percentage of CPU time will always be available to specified programs. If another program attempts to squander CPU time beyond its assigned limit, the RTOS detects this condition and forces the CPU to deliver its cycles to the intended program. The faulty program certainly will fail. But, not the 'innocent' program as a result of the faulty program's errors.



Subtasks spawned by Task B are allocated CPU time from Task B's budget of 50%. Task A still gets its budgeted 50%, no matter how many subtasks are spawned by Task B.

Likewise, memory is guaranteed to programs by requiring that each program contribute some of its own assigned memory when it requests action from the RTOS. It simply cannot exhaust the memory of any other program, even accidentally. That guarantees the 'innocent' program access to the system, using its own memory that is protected against 'theft' by other programs (or viruses or hackers).

### Virus Attack

The RTOS, if it's designed to do so, can prevent a virus from eating up a system's memory, or from causing the CPU to spend all of its time executing code it was not intended to run. This provides a 'final line of defense' against the incursion of a virus. The virus might be able to infect one program, but it will not be able to spread throughout the system and infect 'innocent' programs.

### Hacker Attack

Hackers threaten all network-connected devices. The RTOS can prevent a hacker from getting past the front door. While network security is designed to prevent the hacker from getting in the front door, if the hacker should manage to get through, he/she's now free to roam throughout the system and subvert it to perform whatever function the hacker desires. The RTOS can prevent this, through the same mechanisms that are used to prevent virus damage from spreading. By isolating the hacker to one program, the 'front door' through which he/she entered the system, the RTOS can Guarantee that 'innocent' programs elsewhere in the system aren't affected. This presents a 'final line of defense' against the hacker, should he/she make it past network security measures.

### Conclusion

Threats abound, from bugs to virus attack to hackers. The RTOS can prevent even the most serious bugs from spreading throughout the system, and the most potent virus from infecting innocent programs. The hacker, as well, will be stopped in his tracks by a well-prepared RTOS.

The Intel XScale microarchitecture provides all the hardware needed to enable the RTOS to prevent unwanted memory access and use. But the RTOS must do more than just support the MMU if it's to provide protection against external and internal threats. It must assure that memory and CPU cycles will always be available as designed, and never erroneously wasted by bugs, viruses or hackers. It must provide guaranteed resource availability to securely protect today's networked devices. ○