

Memory Management, The Solution to Managing Large Scale Embedded Projects

A lot of problems in embedded system development can be directly addressed by taking advantage of the underutilized (and often entirely ignored) Memory Management Unit (MMU), a component of all modern CPUs. Designing an embedded framework around MMU architecture is known to provide an efficient solution to large-team development problems as well as runtime reliability. In fact, the solutions described in this article are drawn from real-life experiences with customers who have adapted their development efforts to take advantage of MMU architecture.

INTRODUCTION

Embedded systems are employed in many different markets, including medical (imaging and component processing), industrial automation (nuclear, chemical processing, vehicle production lines), telecoms (PBX, voicemail, switches, routers), gambling systems (transaction processing, slot machines), consumer electronics (set-top boxes, entertainment equipment), handheld computers, and a host of others.

Given the breadth of these markets, it should come as no surprise that there is explosive growth in the development and deployment of embedded systems. This presents an attractive opportunity for desktop development houses to take on embedded projects, which in turn increases the demand for embedded expertise.

There are many lessons that can be learned from companies already in the embedded arena - lessons for controlling problems that occur not only during development, but those affecting runtime reliability as well - such as moving the core technology forward while retaining code reliability in new product versions as more developers join the development team. Another difficult issue for most embedded systems is the lack of memory protection and the problems this causes large-system, large-team development over long periods of time.

The majority of these problems can be directly addressed by taking advantage of the underutilized (and often entirely ignored) Memory Management Unit (MMU), a component of all modern CPUs. Designing an embedded framework around MMU architecture is known to provide an efficient solution to large-team development problems as well as runtime reliability. In fact, the solutions described in this article are drawn from real-life experiences with customers who have adapted their development efforts to take advantage of MMU architecture.

LARGE TEAM CHALLENGES

At one time the term "embedded system" usually meant a simple black box with a small software component designed by a small development team. Modern embedded systems fulfill multiple roles, necessitating more complex functionality than ever before. In some cases, the software development effort often dwarfs the hardware effort, and what is known as an "embedded system" today can refer to any system that is not a desktop or commercial/back-end server system.

Though a company may have started out building small embedded systems with a basic function, demands for new revisions with new features and enhancements continuously drive up complexity. It makes sense to leverage code from previous implementations in developing newer designs.

But the danger here is to also adopt the mistakes of the past through every new generation of product. Those mistakes can directly affect the productivity of development staff and make the configurations of various teams of developers more difficult to manage.

Let's identify some basic problems facing large development teams. Multiple product versions can be developed by multiple teams, involving multiple code versions and multiple source control trees. Version control is always a significant problem. Integration and quality assurance testing consumes a significant proportion of development time. Because of the enormity of the source code, no single programmer can understand the entire code base. Staff turnover is also significant, causing knowledge erosion.

Then you have the morale issue of maintenance developers versus new product developers. Too many times you hear programmers wish that they were developing something "new", and perhaps to do so, leave to join another company.

In many embedded development environments, it's the hardware engineers that actually design software - much to the horror of software engineers that have to enhance the code later on (this also may have con-

tributed to the proliferation of non MMU-based designs).

Anyone in charge of large teams of developers recognizes all of these problems. Many people would be surprised to learn that basic architectural decisions on system architecture can contribute to easing all of these problems.

THE LARGE FLAT MEMORY ADDRESS SPACE

Many large companies that develop embedded products started small and developed into a wealth of highly functional products, offering all the latest bells and whistles, through a code base that may now number in the many millions of lines of source code. However, some of these companies started out by implementing their products based on a flat memory architecture - whole systems were architected from scratch (including the OS or underlying executive). Perhaps this architecture was the only choice depending on when the company started in the embedded market - perhaps the CPUs had no Memory Management Unit (MMU), and perhaps no suitable OS was available off the shelf that could use the MMU

A flat memory address space can be quite manageable for simple embedded systems, but when the code base gets larger and more developers start working on that code base, the problems creep in.

With such an architecture, all modules are folded into a single address space. Programming errors like corrupt C pointers and invalid array indices can cause a program to access memory areas out of its own address space. Without memory protection, these errors can cause programs to overwrite each other - even overwrite the kernel. If you had only a few modules, then errors like this could be identified fairly easily during integration testing. But imagine if you had 50 to 100 modules, written by 50 to 100 programmers? Or a thousand modules and a million lines of code? You might detect such an error occurred during testing - an area of memory being overwritten - but how do you identify which module caused the corruption? You may have had hundreds of threads of execution active - which one caused the corruption? You can spend days, weeks or even months trying to find the culprit. If you run the test again, the condition may not occur again - test teams can become frustrated to the point where they might throw up their arms and let the code go to product anyway. Such decisions, born out of frustration, can be hazardous. Then there is the case of the memory corruption that occurred but was never detected during integration test. Where do these problems raise their head? At the customer site, and at the time when it will be the most inconvenient [Murphy, Law No. 1 - see also Sod, Law No. 1].

So how do you find these problems? One solution is to have lots of system programmers who know the system inside out - they have so much experience with the system that they can identify the corruption by the corrupted data itself. Or they recognize the address of the corrupt data as an ASCII string that is used by a particular module. But this may only buy you limited

advantages - even with experienced systems programmers it could take days, weeks or months to find the problem.

You have a scenario whereby trivial applications (like configuration applications) can cause total system failure. A misunderstanding on the part of a not-so-experienced programmer can have the same result. You could just use experienced programmers, but they are expensive and hard to find, let alone hire.

With all programs in the one address space, creating new versions requires a re-link of the entire runtime image - each relink resulting in a different binary image, with different offset addresses, changes that ripple throughout the entire binary image. A program that has been overwriting an unused data area in a previous release (like memory used for structure or module alignment filler) may have instilled false confidence in a previously stable runtime system. Changing a single line of code in one of the modules might be just enough to make it unstable. Runtime confidence is lost and extensive re-testing is required.

Ultimately, the efficiency of the development effort suffers and the rate of new feature development slows and system reliability suffers. Product releases become further and further apart, and loss of competitive edge can result. All of these problems stem from a monolithic, non-memory protected runtime environment.

PROTECTED MEMORY DESIGN WITH FULL MEMORY MANAGEMENT

An OS that takes full advantage of the Memory Management Unit (MMU), and implements full memory management, provides benefits that cannot be understated.

As far as an application is concerned, its address space is linear, starting from 0. Page tables maintained by the OS and MMU translate virtual address space of the application into physical addresses in memory (described later). The key is that processes cannot overwrite each other's memory space. If a process attempts to access data outside of its address space, the MMU detects an invalid page table lookup, a condition that is in turn trapped by the OS. The OS can then take remedial action on the process at fault. This remedial action must provide as much information as possible to allow the developer to find the cause of the fault:

- Identify the process at fault
- Identify the location of the fault
- Record a process dump file that can be viewed through source-level debugging tools. This dump file would include all information that would allow the debugger to identify the exact line of code that caused the fault, and allow the developer to view all process resources (like contents of data items and a trace of function calls)
- Identify the fault to a system overseer or software watchdog process (discussed later)

A modular system design allows for ready identification of components that cause memory access violations.

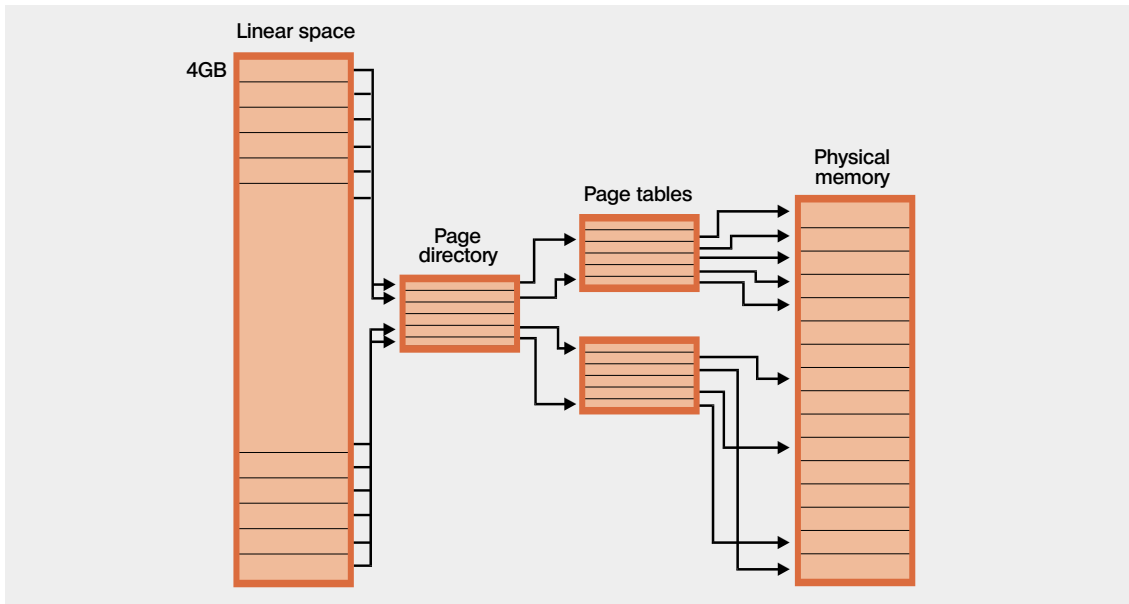


Figure 1. The MMU translates the virtual addresses used by the application into physical addresses applied to memory.

With the implementation of full memory management, each process has a linear address space starting from 0 - there is no need to relink unchanged applications from one product revision to the next. The exact binary image of processes can be re-used in a variety of designs. The result is confidence in utilizing field-tested binaries in new designs, knowing that they have a proven track record.

A protected memory design lets you identify which of the thousand processes tried to perform an invalid memory access, at the exact instruction! The error may even be identified and fixed by the programmer before the process is submitted to the integration test team.

Memory protection is just as important at runtime as it is at development time. No matter how well you test your system before deployment, you can't test everything. If you assume that errors such as memory violations can occur at runtime, memory protection can provide mechanisms for rapidly recovering from software failure - through a software watchdog.

HOW THE MMU WORKS

The MMU divides the physical memory into a number of pages, typically 4Kb each. A set of page tables, stored in system memory and managed by the operating system (OS), then map the virtual memory addresses used within application programs to the physical addresses emitted by the CPU. Note that the application uses memory addresses much as it would in a system without an MMU. The difference is that the OS controls how these addresses are mapped onto physical memory. The diagram in Figure 1 shows the mapping of process memory pages to physical memory pages.

For a large address space with many processes, the number of page-table entries needed to describe these mappings can be significant-more than can be stored within the processor. To maintain performance, the processor caches frequently used portions of the

external page tables within a translation look-aside buffer (TLB). The servicing of "misses" on the TLB cache contributes to the overhead imposed by enabling the MMU. To minimize this overhead, OS designers apply significant design effort to their use of page tables.

The page table entries have associated bits that define the attributes of each page. A page can be marked by the OS as read-only, read-write, and so on. Typically, the memory of an executing process is described with read-only pages for code and read-write pages for data.

When the OS performs a context switch, suspending the execution of one process and resuming another, it manipulates the MMU to use a potentially different set of page tables for the newly resumed process. (If the OS is just switching between threads within a process, no MMU manipulations are necessary.) When the new process resumes execution, any addresses generated are mapped to physical memory through the page tables that the OS assigned. If the process attempts to use an address not mapped to it or to use an address in a way that violates the defined attributes (such as attempting to write to a read-only page), the CPU will receive a "fault," typically implemented as a special type of hardware interrupt. By examining the instruction pointer pushed on the stack by the interrupt, the OS can determine the address of the instruction that caused the memory-access fault within the thread, and act accordingly.

MEMORY PROTECTION PROS AND CONS

By adding memory protection to your mission-critical systems you gain a key advantage: improved robustness. As a result, the OS prevents coding errors in the process from "damaging" other processes or even the OS itself.

Does memory protection have a downside? At one

DEBUGGING

time, it did. With older processors, the MMU was an external chip added to the system. This addition drove up costs and imposed propagation delays in the system address bus. Not surprisingly, most embedded systems didn't use memory protection, and as a result, development tools, and early operating systems for embedded platforms largely ignored MMU hardware.

With on-chip MMU hardware, the overhead from enabling memory protection has decreased significantly. MMUs integrated into the processor perform address translation either in parallel with instruction decode or as one of several overlapped stages within an execution pipeline. Clever restructuring of the MMU page tables by the OS designer can further reduce MMU overhead.

Typically, if enabling memory protection causes a real-time system to miss deadlines, then that system is so close to the edge that it is also at risk from other "occasional" latencies (cache misses, asynchronous interrupts, and so on). Simply put, memory protection fits within the performance budget of most embedded applications.

HARDWARE AND SOFTWARE WATCHDOGS

The software watchdog was identified earlier as a mechanism that aids software recovery. To understand the importance of a software watchdog, let's look at what many existing systems use to recover from software faults: a hardware watchdog timer attached to the processor reset line. Typically, a component of the system software checks for system integrity and then strobes the timer hardware to indicate that the system is "sane." If the hardware timer isn't strobed regularly, it expires and forces a processor reset. The good news is that the system recovers from the software or hardware lockup. The bad news is that the system must also completely restart, perhaps causing unacceptable down time.

Compare this behavior to a software watchdog, which can intelligently choose from several, less drastic, recovery methods. Instead of always forcing a full reset,

the software watchdog could:

- simply restart the process that failed without shutting down the rest of the system, or
- abort any related processes, initialize the hardware to a "safe" state, and restart the related processes in a coordinated manner, or
- if the failure is critical, perform a coordinated shutdown of the entire system and sound an audible alarm to notify the maintenance staff

The software watchdog lets you retain programmed control of the system, even though several processes within the control software may have failed. A hardware watchdog timer can still help you recover from hardware "latch-ups," but for software failures, a software watchdog provides much better control. Furthermore, by employing the "partial restart" approach, your system can survive intermittent software failures without experiencing any down time.

In addition, the ability to log process dump files allows for off-line analysis of runtime problems, giving you the opportunity to engineer a fix before other systems experience the same problems.

INTER-PROCESS COMMUNICATION

Of course, a modular process-based design must include mechanisms for communication between processes. IPC is a fundamental means for "legal" interaction between processes. However, this can have an effect on runtime performance compared to a monolithic flat execution environment.

The OS architecture must be engineered from scratch to provide low overhead IPC, rather than a CPU-expensive add-on layer. The advantages of memory protection must be delivered without any significant performance penalty (again, this is important at development time and runtime).

MICROKERNEL VS MONOLITHIC OS ARCHITECTURE

The memory protection mechanisms discussed so far apply to memory protection between separate

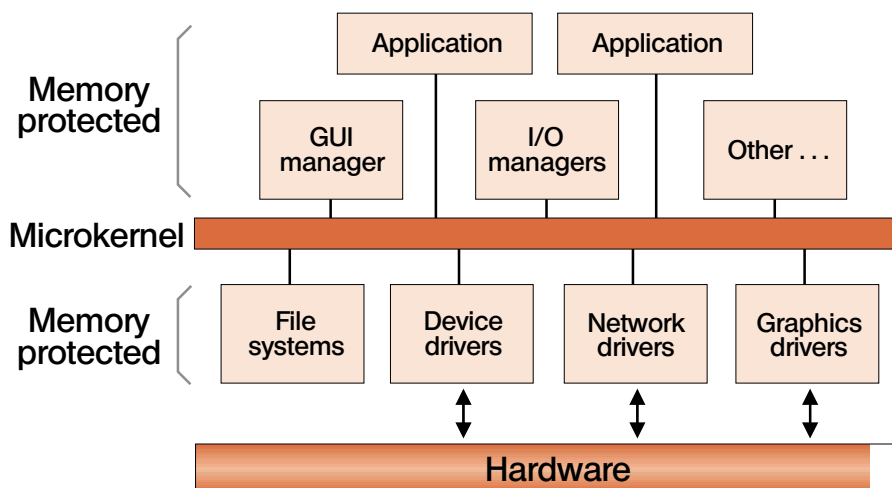


Figure 2. Monolithic kernel architecture.

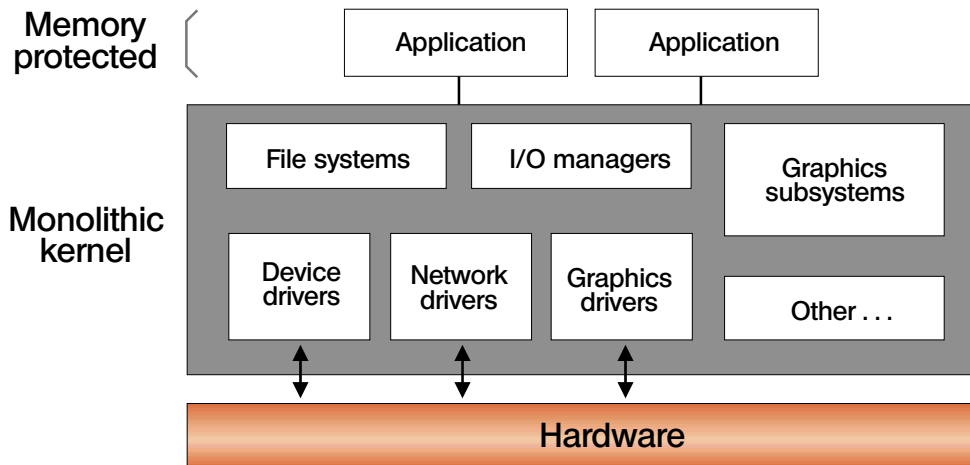


Figure 3. Microkernel architecture (QNX).

processes. Each process has its own unique address space, containing its code and data. Processes may not share any of their address space with any other process (unless explicitly done so - perhaps through shared memory).

The advantages of memory protection are of enormous significance to large team development. However, some OS architectures still incorporate the disadvantages of a single flat monolithic architecture -

in the implementation of monolithic OS kernels.

The embedded market is so diverse, many developers are required to write drivers to support new hardware, or perhaps extend the services of the OS itself. In a monolithic kernel, these components reside as part of the kernel, running in kernel mode (Figure 2). Applications run in user mode, with full memory protection. The kernel runs without memory protection because it must manage the application space and

AD MICROGOLD

provide IPC services across memory boundaries. All the modules running in the kernel are linked together (sometimes dynamically) as a single unit, a unit that has carte blanche access to all system memory - the kernel components can overwrite any part of the system. A stray pointer in a device driver can cause applications to behave incorrectly (or crash) and can also cause kernel faults, from which the only remedy is a hardware reboot.

A microkernel on the other hand provides a base level of services (e.g. process scheduling, IPC, first-level interrupt handling), with additional processes providing the rest of the OS functionality. In contrast to a monolithic kernel, components like drivers, filesystems, I/O managers, and graphics subsystems run in user mode, with all the advantages of memory protection (Figure 3). If all of these components are dynamically restartable, then they can all come under the control of the software watchdog. The possibility of kernel faults from add-on device drivers is almost eliminated, with only a small amount of kernel code and Interrupt Service Routines (ISRs) running in kernel mode.

DOES THIS GIVE US SOLUTIONS TO THE PROBLEMS?

The combination of MMU-based memory protection, full memory management, a microkernel architecture, and software watchdog recovery is a strong basis for developing mission-critical, crash-proof systems - a quality valued enormously in so many embedded applications.

Lets take a look at the large team problems identified earlier and see how this architecture contributes to solving them:

- Memory violations are identified at the exact instruction, before it is executed. The rogue process is immediately identified and a post-mortem dump can be generated. Unit, system, and integration testing becomes much more straightforward, having a marked effect on the duration of those phases. No more chasing for a rogue module amongst a thousand modules - the OS tells you immediately.
- More confidence in runtime systems. Processes cannot accidentally write to "filler" memory areas outside their scope. Exact binaries of processes can be used from one product to the next. This eases the QA burden, with systems built using components that are already proven in the field.
- Programmers don't have to know in intricate detail how each module works. All processes have a defined, MMU-hardware enforced communications interface through IPC. If you send a process the wrong message, it rejects it. Processes can be deployed by several programmers on a single target system simultaneously, without the fear of overwriting, or being overwritten by, other processes. Even a moderately experienced programmer could start writing device drivers, again simultaneously with others.
- Because all of the modules are decoupled from one another, this has a tremendous impact on unit,

system, and integration testing. You are guaranteed that the individual processes can ONLY talk to each other through certain, well-defined, interfaces, thus minimizing the actual amount that MUST BE tested

- The degree of maintenance is drastically reduced. More resources can be spent on enhancing products and producing new features.
- Teams can be structured around each process (or process subsystem) that makes up the embedded system - resulting in more flexibility in team management and makeup.
- Hardware engineers are forced into a development environment that software engineers find natural.

SUMMARY

Memory protection and microkernel architecture - two very simple concepts, provides such an enormous advantage for large teams in overcoming the big problems. Implementing both in your embedded designs also provide the basis for highly reliable, mission-critical systems.

The architectures and features described herein are all implemented within the QNX RTOS (QNX Software Systems Ltd., Ontario, Canada). The analogies were derived from real-life experiences with some of our largest customers involved in large-scale applications, with large numbers of developers. The applications implemented by these customers vary from set-top boxes, to nuclear monitoring and control, to television broadcast automation systems - mission-critical applications that require the utmost in reliability ■

By Paul Leroux, Senior Technology Analyst at QNX Software Systems.