References


INTRODUCTION

In the few years since Intel released the 386 processor, it has gone from a tremendously overpriced compute engine to the minimum processor for anyone considering purchasing a PC. Proliferation versions (like the 386SX and AMD's variants) drive the chip cost down while maintaining software compatibility with the rest of the line.

It seems those of us in the embedded world could ignore this technology, since so many designs revolve around low performance controllers. Now, however, more and more embedded systems use the 386 series of components. Examples include high speed data communications devices (though in cheap modems the Z80 still reigns supreme), graphics equipment, and ultra-high-speed data acquisition gear. Even the cockpit displays of some modern jetliners use 386's as controllers.

Why? What's so great about the 386 that compels a designer to include a $325 processor in his embedded system? The 386 offers two important features: raw compute horsepower, and the potential for a huge address space.

386 BENEFITS

Most of us computing with a 386-based PC run the processor in its slowest and least functional mode. Yet, even then we get staggering performance improvements over that for which we lustred a decade ago. Most PC applications run in 'real mode', using 8088-like 20 bit addresses and 16 bit registers. The 386 can and does often act just like a very fast 8088. Its most obvious virtue is its raw speed. With no wait states machine cycles take only two clocks. At 33 MHz, this is a blazing 61 nsec per cycle. Short instructions (e.g., a register to register move) complete in two cycles, or about 122 nsec. This baby is no slouch at moving data!

There is a sort of hidden price to running so fast, though. How many memory systems can present data so quickly? Inject a single wait state, and the machine's performance declines by a third. Any high performance embedded system will likely need costly cache to properly match memory speeds to the processor's bandwidth.

The 386 has a richer instruction set than it's 80x86 cousins. 32 bit multiply/divides, barrel shifters that shift up to 32 bits in 7 cycles, and bit manipulations are all included. All registers are 32 bits, so handling decent sized data is a breeze.

Embedded people might be disappointed with its lack of peripherals, like 81C181, 8051, 80196, and other embedded parts include timers, serial ports, and the like, all designed to reduce the cost and size of a system. Not so the 386, which is targeted only at high performance, high cost applications. I hope Intel or AMD does eventually come up with versions specifically for embedded markets, including serial and parallel ports. It would seem a sensible use of the vendors' ability to cram more functionality onto a piece of silicon. After all, even the RISC folks are now targeting processors specifically towards the embedded marketplace.

PROTECTED VS. REAL MODES

If you've worked with the 80x86 family, you are intimately familiar with what 386 documentation calls 'Real Mode'. Real Mode addresses are limited to 20 bits, and are generated by adding a 16 bit segment register, shifted left four bits, to a 16 bit offset. This much maligned segmentation causes no end of grief for programmers trying to access large data structures. Since an offset cannot exceed 16 bits, you just can't increment beyond 64k; you'll have to watch for a 64k boundary and then play games with the segment register.

The 386's Protected Mode changes everything you ever learned about 80x86 segmentation. Protected mode offers direct access to 32 bit addresses. Though segment registers still play a part in every address calculation, their role is no longer one of directly specifying an address. In protected mode segment registers are pointers to data structures that define segmentation limits and addresses. More on this later.

On a 386 operating in real mode you have access to practically every feature the 386 has to offer - with the exception of 32 bit addressing. Just about all of the new instructions are available. All operands can be 8, 16, or even 32 bits. That's right - real mode programs can easily handle double word long data, using 32 bit registers. On the 386, in real or protected modes, you access operands as follows:

\[
\begin{align*}
\text{mov} & \quad \text{al},[1000] \quad \text{load 8 bits} \\
\text{mov} & \quad \text{ax},[1000] \quad \text{load a word} \\
\text{mov} & \quad \text{eax},[1000] \quad \text{load a double word}
\end{align*}
\]

Manipulate data the same way:

\[
\begin{align*}
\text{add} & \quad \text{al},\text{cl} \quad \text{add two bytes} \\
\text{add} & \quad \text{eax},\text{ecx} \quad \text{add two 32 bit numbers}
\end{align*}
\]
Building Protected Mode Embedded Systems

Jack Ganssle
President
Softaid, Inc.

INTRODUCTION

In the few years since Intel released the 386 processor, it has gone from a tremendously overpriced compute engine to the minimum processor for anyone considering purchasing a PC. Proliferation versions (like the 386SX and AMD’s variants) drive the chip cost down while maintaining software compatibility with the rest of the line.

It seems those of us in the embedded world could ignore this technology, since so many designs revolve around low performance controllers. Now, however, more and more embedded systems use the 386 series of components. Examples include high speed data communications devices (though in cheap modems the Z80 still reigns supreme), graphics equipment, and ultra-high-speed data acquisition gear. Even the cockpit displays of some modern jetliners use 386’s as controllers.

Why? What’s so great about the 386 that compels a designer to include a $325 processor in his embedded system? The 386 offers two important features: raw compute horsepower, and the potential for a huge address space.

386 BENEFITS

Most of us computing with a 386-based PC run the processor in its slowest and least functional mode. Yet, even then we get staggering performance improvements over that for which we lusted a decade ago. Most PC applications run in ‘real mode’, using 8088-like 20 bit addresses and 16 bit registers.

The 386 can and does often act just like a very fast 8088. It’s most obvious virtue is its raw speed. With no wait states machine cycles take only two clocks. At 33 MHz, this is a blazing 61 nsec per cycle. Short instructions (e.g., a register to register move) complete in two cycles, or about 122 nsec. This baby is no slouch at moving data!

There is a sort of hidden price to running so fast, though. How many memory systems can present data so quickly? Inject a single wait state, and the machine’s performance declines by a third. Any high performance embedded system will likely need costly cache to properly match memory speeds to the processor’s bandwidth.

The 386 has a richer instruction set than it’s 80x86 cousins. 32 bit multiply/divides, barrel shifters that shift up to 32 bits in 7 cycles, and bit manipulations are all included. All registers are 32 bits, so handling decent sized data is a breeze.

Embedded people might be disappointed with its lack of peripherals, n4186/8186, 8051, 80196, and other embedded parts include timers, serial ports, and the like, all designed to reduce the cost and size of a system. Not so the 386, which is targeted only at high performance, high cost applications. I hope Intel or AMD does eventually come up with versions specifically for embedded markets, including serial and parallel ports. It would seem a sensible use of the vendors’ ability to cram ever more functionality onto a piece of silicon. After all, even the RISC folks are now targeting processors specifically towards the embedded marketplace.

PROTECTED VS. REAL MODES

If you’ve worked with the 80x86 family, you are intimately familiar with what 386 documentation calls ‘Real Mode’. Real Mode addresses are limited to 20 bits, and are generated by adding a 16 bit segment register, shifted left four bits, to a 16 bit offset. This much maligned segmentation causes no end of grief for programmers trying to access large data structures. Since an offset cannot exceed 16 bits, you just can’t increment beyond 64k; you’ll have to watch for a 64k boundary and then play games with the segment register.

The 386’s Protected Mode changes everything you ever learned about 80x86 segmentation. Protected mode offers direct access to 32 bit addresses. Though segment registers still play a part in every address calculation, their role is no longer one of directly specifying an address. In protected mode segment registers are pointers to data structures that define segmentation limits and addresses. More on this later.

On a 386 operating in real mode you have access to practically every feature the 386 has to offer - with the exception of 32 bit addressing. Just about all of the new instructions are available. All operands can be 8, 16, or even 32 bits. That’s right - real mode programs can easily handle double word long data, using 32 bit registers. On the 386, in real or protected modes, you access operands as follows:

```
mov al, [1000] ; load 8 bits
mov ax, [1000] ; load a word
mov eax, [1000] ; load a double word
```

Manipulate data the same way:

```
add al, cl ; add two bytes
add eax, ecx ; add two 32 bit numbers
```
You can use the 32 bit registers to address memory, but in real mode the effective address may not exceed 20 bits. The 386 will generate an exception if the address is too large.

Take advantage of the 386's extended instructions (even in real mode), to greatly speed processing:

```assembly
mul eax,edx ; 32 x 32 multiply
; 64 bit result goes to edx:eax
```

The processor includes extra segment registers. Where an 80x86 CPU only provides ES, DS, SS, and CS, the 386 adds FS and GS, which you can use in real or protected mode.

**PROTECTED MODE ADDRESSING**

Segment registers are called "selectors" when operating in protected mode, to distinguish their operation from that of real mode. For these registers do indeed perform a selection process. In protected mode, segment register simply point to a data structures that contain the information needed to access a location.

Every protected mode program must include a table of "descriptors", which are 8 byte data structures that define the start and end of a segment. Depending on the type of segment, a descriptor may have other information such as access rights and the like. A typical descriptor contains the following information, packed into an 8 byte record:

- **Segment start**: absolute 32 bit address
- **Segment limit**: Maximum address this segment can reference
- **Segment status**: privilege level, segment present, segment available, segment type, etc.

Thus, the descriptor tells the 386 everything it needs to know about accessing data or code in a segment. Accesses to memory are qualified by the descriptor selected by the current segment register. This selector is a 12 bit number indicating which entry to use in the descriptor table; if the selector is 0, the first descriptor is taken, a selector of 1 takes the second, etc. The 386 multiplies the selector by 8 (8 bytes per entry), and adds this to the base address of the table of descriptors (contained in an internal 386 register loaded by the programmer before switching to protected mode.)

For example, a code fetch always uses the current CS. A protected mode fetch starts by multiplying CS by 8 and then adding the descriptor base register. The 386 then reads an entire 8 byte record from the descriptor table. The entry describes the start of the segment; the processor adds the current instruction pointer to this start to get an effective address.

A data access behaves the same way. A load from location DS:1000 makes the processor read a descriptor by shifting DS left 3 bits (i.e., times 8), adding the table's base address (stored in the 386's on-board descriptor table register), and reading the 8 byte descriptor at this address. The descriptor contains the segment's start address, which is added to the offset in the instruction (in this case 1000). Offsets, and segment start addresses, are 32 bit numbers - it's really easy to reference any location in memory.

Every memory access works through these 8 byte descriptors. If they were stored only in user RAM the 386's throughput would be pathetic, since each memory reference needs the information. Can you imagine waiting for an 8 byte read before every memory access? Actually, the processor caches a descriptor for each selector (one for CS, one for DS, etc) on-chip, so the segment translation requires no overhead. However, every load of a selector (like MOV DS,AX or POP ES) will make the 386 stop and read all 8 bytes to it's internal cache, slowing things down just a bit.

It's all a little mind boggling. The CPU manipulates these 8 byte data structures automatically, reading, parsing, caching, and working with them as needed, with no programmer intervention (once they are set up).

Not only does the CPU translate addresses as described. In parallel it checks every memory reference to insure it behaves properly. Remember the "limit" field in the descriptor? If the effective address (base plus offset) is greater than this limit, the 386 aborts the instruction and generates a protection violation exception. It won't let you do something stupid. You can even specify that a segment is read-only; a write will create the same exception.

But wait a minute! Everyone seems to think that segments aren't used in protected mode! In fact, segmentation is practically essential, and is far more useful than you might think.

On a 80x86 processor you'll frequently write programs divided into more than one named code segment. The linker combines like-named segments together, and then groups the segments into one hunk. In the embedded world, using a Locator (like ones sold by Systems and Software and Paradigm), you can separate named segments into specific RAM or ROM addresses to match the nuances of your particular hardware environment. The 386 takes this one step further.

A 386 linker groups like-named segment together. Then, if you wish, you can assign any group to any descriptor. Though the selector uses only 12 bits to pick a descriptor, another bit selects which of two descriptor tables to read from (the Local or Global tables), giving up to 8192 separate segments.

This is a lot of power; most DOS users ignore it. It is ideal for embedded applications. Suppose you have memory mapped I/O: group it into a named segment and assign read/write attributes to it. Even better, separate read and write ports into different segments to insure your code never accidently accesses one incorrectly. Make your code fetch-only, so illegal accesses create protection violation errors debugging will be a lot easier with this enabled.

Some embedded systems include a ROMed version of DOS. DOS runs in real mode only, so use the 386's segmentation to define real and protected segments. The real ones will (sigh) not have the great protection mechanisms. Restrict them to low addresses (under 20 bits), and put the protected mode code up high. The real mode
You can use the 32 bit registers to address memory, but in real mode the effective address may not exceed 20 bits. The 386 will generate an exception if the address is too large.

Take advantage of the 386's extended instructions (even in real mode), to greatly speed processing:

```
mul eax,edx ; 32 x 32 multiply
; 64 bit result goes to edx:eax
```

The processor includes extra segment registers. Where an 80x86 CPU only provides ES, DS, SS, and CS, the 386 adds FS and GS, which you can use in real or protected mode.

**PROTECTED MODE ADDRESSING**

Segment registers are called "selectors" when operating in protected mode, to distinguish their operation from that of real mode. For these registers do indeed perform a selection process. In protected mode, segment register simply point to a data structures that contain the information needed to access a location.

Every protected mode program must include a table of "descriptors", which are 8 byte data structures that define the start and end of a segment. Depending on the type of segment, a descriptor may have other information such as access rights and the like. A typical descriptor contains the following information, packed into an 8 byte record:

- **Segment start:** absolute 32 bit address
- **Segment limit:** Maximum address this segment can reference
- **Segment status:** privilege level, segment present, segment available, segment type, etc.

Thus, the descriptor tells the 386 everything it needs to know about accessing data or code in a segment. Accesses to memory are qualified by the descriptor selected by the current segment register. This selector is a 12 bit number indicating which entry to use in the descriptor table; if the selector is 0, the first descriptor is taken, a selector of 1 takes the second, etc. The 386 multiplies the selector by 8 (8 bytes per entry), and adds this to the base address of the table of descriptors (contained in an internal 386 register loaded by the programmer before switching to protected mode.)

For example, a code fetch always uses the current CS. A protected mode fetch starts by multiplying CS by 8 and then adding the descriptor base register. The 386 then reads an entire 8 byte record from the descriptor table. The entry describes the start of the segment; the processor adds the current instruction pointer to this start to get an effective address.

A data access behaves the same way. A load from location DS:1000 makes the processor read a descriptor by shifting DS left 3 bits (i.e., times 8), adding the table's base address (stored in the 386's on-board descriptor table register), and reading the 8 byte descriptor at this address. The descriptor contains the segment's start address, which is added to the offset in the instruction (in this case 1000). Offsets, and segment start addresses, are 32 bit numbers - it's really easy to reference any location in memory.

Every memory access works through these 8 byte descriptors. If they were stored only in user RAM the 386's throughput would be pathetic, since each memory reference needs the information. Can you imagine waiting for an 8 byte read before every memory access? Actually, the processor caches a descriptor for each selector (one for CS, one for DS, etc) on-chip, so the segment translation requires no overhead. However, every load of a selector (like MOV DS,AX or POP ES) will make the 386 stop and read all 8 bytes to it's internal cache, slowing things down just a bit.

It's all a little mind boggling. The CPU manipulates these 8 byte data structures automatically, reading, parsing, caching, and working with them as needed, with no programmer intervention (once they are set up).

Not only does the CPU translate addresses as described. In parallel it checks every memory reference to insure it behaves properly. Remember the "limit" field in the descriptor? If the effective address (base plus offset) is greater than this limit, the 386 aborts the instruction and generates a protection violation exception. It won't let you do something stupid. You can even specify that a segment is read-only; a write will create the same exception.

But wait a minute! Everyone seems to think that segments aren't used in protected mode! In fact, segmentation is practically essential, and is far more useful than you might think.

On a 80x86 processor you'll frequently write programs divided into more than one named code segment. The linker combines like-named segments together, and then groups the segments into one hunk. In the embedded world, using a Locator (like ones sold by Systems and Software and Paradigm), you can separate named segments into specific RAM or ROM addresses to match the nuances of your particular hardware environment. The 386 takes this one step further.

A 386 linker groups like-named segment together. Then, if you wish, you can assign any group to any descriptor. Though the selector uses only 12 bits to pick a descriptor, another bit selects which of two descriptor tables to read from (the Local or Global tables), giving up to 8192 separate segments.

This is a lot of power; most DOS users ignore it. It is ideal for embedded applications. Suppose you have memory mapped I/O: group it into a named segment and assign read/write attributes to it. Even better, separate read and write ports into different segments to insure your code never accidently accesses one incorrectly. Make your code fetch-only, so illegal accesses create protection violation errors - debugging will be a lot easier with this enabled.

Some embedded systems include a ROMed version of DOS. DOS runs in real mode only, so use the 386's segmentation to define real and protected segments. The real ones will (sigh) not have the great protection mechanisms. Restrict them to low addresses (under 20 bits), and put the protected mode code up high. The real mode
code will not physically be able to generate a high address that might affect the protected mode code.

**LINKERS**

If we had to define the selectors and descriptors ourselves, protected mode would be just too hard to use. The descriptors are arranged in a nasty, hard to assemble format. Fortunately, Intel and others supply linkers that do all of the hard work for you.

It is a little tedious to actually switch from real to protected mode, but Intel application notes do a pretty good job of describing the procedure. There seems to be surprisingly little written about actually building an application. It turns out that the linker does most of the work of building descriptors.

I've been using System & Software's (Irvine, CA) Link & Locate 386 lately, and find that writing protected mode code with it is a breeze. Writing protected mode code is really no different than for real mode. Break your code into named segments, separating data and code, and segment them further if you wish to restrict access in some fashion. Assemble the code with any decent assembler: Microsoft's MASM and Borland's TASM do just fine. Then, use a linker with a carefully scripted command file to assign descriptors as wished.

This program consists of just 4 segments. Real_code is real mode code executed occasionally by the program. Cgroup is the bulk of the program. Dgroup is a data area. Flat_seg is a special segment defined so the program can perform a linear address anywhere in memory.

The segments, in many cases, have absolute addresses assigned, defining their start. The Linker puts in ending limits automatically.

**FLAT_SEG** is a special case; we've set it to start at 0 and end at the end of memory. This more or less bypasses protection checking, as the segment's definition precludes getting an addressing error. Sometimes, in embedded systems we need to access any area to get to specific hardware.

A program operating with this structure will have its code all in segment cgroup, and all data in dgroup. The program will start with code that looks something like:

```assembly
    dgroup segment use32; data segment
data1 dd?
data2 dd?
dgroup ends
cgroup segment assumetcx: cgroupl dsgroup
mov ax, 0000h
mov ds, ax
mov ds, 0000h
mov eax, data1; using DS, reference data1

This looks just like 8086 code. Now, suppose we want an absolute reference anywhere in memory (say, we have some weird hardware device to read from). Do this:

```assembly
    mov ax, flat_seg
    mov es, ax ; set selector ES to flat_seg
    mov esi, address
    mov al, es:[esi] ; read from an absolute address
```

Since selector ES points to a descriptor that is a flat, 32 bit address space, any number in ESI is a 32 bit offset added to flat_seg's start address of 0.

Avoid writing code that runs in one 32 bit flat segment. Sure, it is the easiest way to generate a big program. You'll lose the benefits of the 386's protection checking. This is especially deadly with ROMed code - how will you know that the code is not sometimes accidentally writing over the ROM? A ROM write is not in itself a problem, but usually indicates some software flaw that may go undetected.

The code set up selectors just like real mode 8086 code sets segment registers. There really is no difference. The linker replaces segment references with pointers to the descriptor table. In the linker command file, we've defined 'gdt' (the Global Descriptor Table), and specific entries for each segment. GDT entries 1 to 8 are reserved in this case, but 9 corresponds to dgroup, to cgroup, etc. The linker will build GDT and insert it into the program.

```
segment
*segments (dpl = 0),
real_code( dpl = 0, base = 08000h, usereal ),
dgroup( dpl = 0 ),
cgroup( dpl = 0, base = 200000h ),
flat_seg( dpl = 0, base = 0, limit = 0xfffffffh ),
table
gdt(location = gdt_start, reserve = (1-8),
    entry =
    (9: dgroup,
     10: cgroup,
     11: flat_seg);
end;
```

**PROTECTION SYSTEMS**

So far I've glossed over the details of the format of selectors and descriptors. In fact, each contains information used to keep ill-behaved programs in check. The whole issue of capturing address violation errors is perhaps a bit new to the embedded world, but with the proliferation of even more complex systems will certainly become important in the next few years. As one who has suffered through watching programs crash and write over themselves, I find it breathtaking to watch buggy 386 code recover from practically any insult I toss at it; the protection
code will not physically be able to generate a high address that might affect the protected mode code.

LINKERS

If we had to define the selectors and descriptors ourselves, protected mode would be just too hard to use. The descriptors are arranged in a nasty, hard to assemble format. Fortunately, Intel and others supply linkers that do all of the hard work for you.

It is a little tedious to actually switch from real to protected mode, but Intel application notes do a pretty good job of describing the procedure. There seems to be surprisingly little written about actually building an application. It turns out that the linker does most of the work of building descriptors.

I've been using System & Software's (Irvine, CA) Link & Locate 386 lately, and find that writing protected mode code with it is a breeze. Writing protected mode code is really no different than for real mode. Break your code into named segments, separating data and code, and segment them further if you wish to restrict access in some fashion. Assemble the code with any decent assembler: Microsoft's MASM and Borland's TASM do just fine. Then, use a linker with a carefully scripted command file to assign descriptors as wished.

This program consists of just 4 segments. Real_code is real mode code executed occasionally by the program. Cgroup is the bulk of the program. Dgroup is a data area. Flat_seg is a special segment defined so the program can perform a linear address anywhere in memory.

The segments, in many cases, have absolute addresses assigned, defining their start. The Liner puts in ending limits automatically.

Flat_seg is a special case; we've set it to start at 0 and end at the end of memory. This more or less bypasses protection checking, as the segment's definition precludes getting an addressing error. Sometimes, in embedded systems we need to access any area to get to specific hardware.

A program operating with this structure will have its code all in segment cgroup, and all data in dgroup. The program will start with code that looks something like:

```asm
; segments (dpl = 0),
real_code( dpl = 0, base = 08000h, use real ),
dgroup(dpl = 0),
cgroup(dpl = 0, base = 200000h ),
flat_seg( dpl = 0, base = 0, limit = Offffffffh ),
table
gdt (location = gdt_start, reserve = (1..8),
    entry =
        (9: dgroup,
         10: cgroup,
         11: flat_seg));
end;
```

PROTECTION SYSTEMS

So far I've glossed over the details of the format of selectors and descriptors. In fact, each contains information used to keep ill-behaved programs in check. The whole issue of capturing address violation errors is perhaps a bit new to the embedded world, but with the proliferation of ever more complex systems will certainly become important in the next few years. As one who has suffered through watching programs crash and write over themselves, I find it breathtaking to watch buggy 386 code recover from practically any insult I toss at it; the protection...
mechanisms insure that the code never gets overwritten, and that the operating system, if any, remains intact and functional.

The 386 supports 3 privilege levels, numbered 0 to 3. The highest, most privileged level is 0 - a program running at this level can gain access to any 386 resource. Programs running with lower privilege levels are restricted in their ability to use memory, I/O, and some instructions.

Privilege levels are intimately tied to descriptors. As I mentioned, the descriptor contains the base address of a segment, the segment size, and access rights bits. Two of these bits specify the Descriptor's Privilege Level (DPL). Privileges are thus associated with segments, a somewhat novel concept when you consider that most CPUs simply have a global privilege setting that effects all of memory equally.

Before describing how a segment's DPL effects memory access rights, it makes sense to answer the obvious question: what defines the processor's privilege level? Clearly enough, this is handled entirely within the context of segment privilege levels. The CPU runs at the privilege level defined within the DPL of the current code segment - the Current Privilege Level (CPL). Privileges are somewhat removed from the code, then. A transfer to a segment with a DPL of 0 (say, the operating system), will always run with the greatest access rights. Vector off to a code segment with DPL = 3 and you'll be very limited in your ability to run amok.

Every time any section of code accesses another segment, the 386 hardware compares the CPL to the referenced segment's DPL (i.e., it compares the privilege level the CPU is running at to the privilege defined for the segment). If the CPL is the same or higher (smaller number) than the DPL, then it can proceed with the access. A segment's DPL is only as high as the CPU's own CPL, so this type of checking is cheap.

Thus, code running in a segment with a DPL of 0 pumps the CPU up to a CPL of 0, and gives the CPU access to every other segment.

Novice 8086 assembly programmers always moan about the complexity of segments and segment groups. Sometimes the ASSUMES, GROUPS, and other pseudo-ops seem to be an awful lot of trouble. When you switch to the 386 suddenly these constructs make perfect sense: group like segments together, simultaneously grouping privilege levels. Perhaps the operating system will be grouped into one segment with a DPL of 0 so it can access any resource. Maybe device drivers can fit into a less important group, giving them just as much power as needed but no more, preventing them from trashing code. Finally, run the application program at a very low privilege (i.e., high number, like 3), so it cannot effect system data structures or I/O.

We're now talking about two independent levels of protection. The first is defined by segment sizes: no task can access outside of whatever segment it is attempting to use, since an address that exceeds the segment-size field in the descriptor will generate an exception. Obviously, array subscripting errors just cannot cause major crashes if the segments are defined cleverly. The second level of protection is DPL checking, which prevents accesses to higher privileged segments.

In addition, the processor provides hardware protection of certain dangerous
mechanisms insure that the code never gets overwritten, and that the operating system, if any, remains intact and functional.

The 386 supports 3 privilege levels, numbered 0 to 3. The highest, most privileged level is 0 - a program running at this level can gain access to any 386 resource. Programs running with lower privilege levels are restricted in their ability to use memory, I/O, and some instructions.

Privilege levels are intimately tied to descriptors. As I mentioned, the descriptor contains the base address of a segment, the segment size, and access rights bits. Two of these bits specify the Descriptor's Privilege Level (DPL). Privileges are thus associated with segments, a somewhat novel concept when you consider that most CPUs simply have a global privilege setting that affects all of memory equally.

Before describing how a segment's DPL effects memory access rights, it makes sense to answer the obvious question: what defines the processor's privilege level? Cleverly enough, this is handled entirely within the context of segment privileges. The CPU runs at the privilege level defined within the DPL of the current code segment - the Current Privilege Level (CPL). Privileges are somewhat removed from the code, then. A transfer to a segment with a DPL of 0 (say, the operating system), will always run with the greatest access rights. Vector off to a code segment with DPL = 3 and you'll be very limited in your ability to run amok.

Every time any section of code accesses another segment, the 386 hardware compares the CPL to the referenced segment's DPL (i.e., it compares the privilege level the CPU is running at to the privilege defined for the segment). If the CPL is the same or higher (smaller number) than the DPL, then it can proceed with the access. A small attempt to access a segment more privileged than the CPU's CPL will result in an exception, letting us know something is wrong.

Thus, code running in a segment with a DPL of 0 pumps the CPU up to a CPL of 0, and gives the CPU access to every other segment.

Novice 8086 assembly programmers always moan about the complexity of segments and segment groups. Sometimes the ASSUMEs, GROUPs, and other pseudo-ops seem to be an awful lot of trouble. When you switch to the 386 suddenly these constructs make perfect sense: group like segments together, simultaneously protecting privilege levels. Perhaps the operating system will be grouped into one segment with a DPL of 0 so it can access any resource. Maybe device drivers can fit themselves inside of the descriptor table. That is, the call gate contains the complete protection status for each and every port.

Privilege levels are intimately tied to descriptors. As I mentioned, the descriptor contains the base address of a segment, the segment size, and access rights bits. Two of these bits specify the Descriptor's Privilege Level (DPL). Privileges are thus associated with segments, a somewhat novel concept when you consider that most CPUs simply have a global privilege setting that affects all of memory equally.

Before describing how a segment's DPL effects memory access rights, it makes sense to answer the obvious question: what defines the processor's privilege level? Cleverly enough, this is handled entirely within the context of segment privileges. The CPU runs at the privilege level defined within the DPL of the current code segment - the Current Privilege Level (CPL). Privileges are somewhat removed from the code, then. A transfer to a segment with a DPL of 0 (say, the operating system), will always run with the greatest access rights. Vector off to a code segment with DPL = 3 and you'll be very limited in your ability to run amok.

Every time any section of code accesses another segment, the 386 hardware compares the CPL to the referenced segment's DPL (i.e., it compares the privilege level the CPU is running at to the privilege defined for the segment). If the CPL is the same or higher (smaller number) than the DPL, then it can proceed with the access. A small attempt to access a segment more privileged than the CPU's CPL will result in an exception, letting us know something is wrong.

Thus, code running in a segment with a DPL of 0 pumps the CPU up to a CPL of 0, and gives the CPU access to every other segment.

Novice 8086 assembly programmers always moan about the complexity of segments and segment groups. Sometimes the ASSUMEs, GROUPs, and other pseudo-ops seem to be an awful lot of trouble. When you switch to the 386 suddenly these constructs make perfect sense: group like segments together, simultaneously protecting privilege levels. Perhaps the operating system will be grouped into one segment with a DPL of 0 so it can access any resource. Maybe device drivers can fit themselves inside of the descriptor table. That is, the call gate contains the complete protection status for each and every port.

Privilege levels are intimately tied to descriptors. As I mentioned, the descriptor contains the base address of a segment, the segment size, and access rights bits. Two of these bits specify the Descriptor's Privilege Level (DPL). Privileges are thus associated with segments, a somewhat novel concept when you consider that most CPUs simply have a global privilege setting that affects all of memory equally.

Before describing how a segment's DPL effects memory access rights, it makes sense to answer the obvious question: what defines the processor's privilege level? Cleverly enough, this is handled entirely within the context of segment privileges. The CPU runs at the privilege level defined within the DPL of the current code segment - the Current Privilege Level (CPL). Privileges are somewhat removed from the code, then. A transfer to a segment with a DPL of 0 (say, the operating system), will always run with the greatest access rights. Vector off to a code segment with DPL = 3 and you'll be very limited in your ability to run amok.

Every time any section of code accesses another segment, the 386 hardware compares the CPL to the referenced segment's DPL (i.e., it compares the privilege level the CPU is running at to the privilege defined for the segment). If the CPL is the same or higher (smaller number) than the DPL, then it can proceed with the access. A small attempt to access a segment more privileged than the CPU's CPL will result in an exception, letting us know something is wrong.

Thus, code running in a segment with a DPL of 0 pumps the CPU up to a CPL of 0, and gives the CPU access to every other segment.

Novice 8086 assembly programmers always moan about the complexity of segments and segment groups. Sometimes the ASSUMEs, GROUPs, and other pseudo-ops seem to be an awful lot of trouble. When you switch to the 386 suddenly these constructs make perfect sense: group like segments together, simultaneously protecting privilege levels. Perhaps the operating system will be grouped into one segment with a DPL of 0 so it can access any resource. Maybe device drivers can fit themselves inside of the descriptor table. That is, the call gate contains the complete protection status for each and every port.

Privilege levels are intimately tied to descriptors. As I mentioned, the descriptor contains the base address of a segment, the segment size, and access rights bits. Two of these bits specify the Descriptor's Privilege Level (DPL). Privileges are thus associated with segments, a somewhat novel concept when you consider that most CPUs simply have a global privilege setting that affects all of memory equally.

Before describing how a segment's DPL effects memory access rights, it makes sense to answer the obvious question: what defines the processor's privilege level? Cleverly enough, this is handled entirely within the context of segment privileges. The CPU runs at the privilege level defined within the DPL of the current code segment - the Current Privilege Level (CPL). Privileges are somewhat removed from the code, then. A transfer to a segment with a DPL of 0 (say, the operating system), will always run with the greatest access rights. Vector off to a code segment with DPL = 3 and you'll be very limited in your ability to run amok.

Every time any section of code accesses another segment, the 386 hardware compares the CPL to the referenced segment's DPL (i.e., it compares the privilege level the CPU is running at to the privilege defined for the segment). If the CPL is the same or higher (smaller number) than the DPL, then it can proceed with the access. A small attempt to access a segment more privileged than the CPU's CPL will result in an exception, letting us know something is wrong.

Thus, code running in a segment with a DPL of 0 pumps the CPU up to a CPL of 0, and gives the CPU access to every other segment.

Novice 8086 assembly programmers always moan about the complexity of segments and segment groups. Sometimes the ASSUMEs, GROUPs, and other pseudo-ops seem to be an awful lot of trouble. When you switch to the 386 suddenly these constructs make perfect sense: group like segments together, simultaneously protecting privilege levels. Perhaps the operating system will be grouped into one segment with a DPL of 0 so it can access any resource. Maybe device drivers can fit themselves inside of the descriptor table. That is, the call gate contains the complete protection status for each and every port.

Privilege levels are intimately tied to descriptors. As I mentioned, the descriptor contains the base address of a segment, the segment size, and access rights bits. Two of these bits specify the Descriptor's Privilege Level (DPL). Privileges are thus associated with segments, a somewhat novel concept when you consider that most CPUs simply have a global privilege setting that affects all of memory equally.

Before describing how a segment's DPL effects memory access rights, it makes sense to answer the obvious question: what defines the processor's privilege level? Cleverly enough, this is handled entirely within the context of segment privileges. The CPU runs at the privilege level defined within the DPL of the current code segment - the Current Privilege Level (CPL). Privileges are somewhat removed from the code, then. A transfer to a segment with a DPL of 0 (say, the operating system), will always run with the greatest access rights. Vector off to a code segment with DPL = 3 and you'll be very limited in your ability to run amok.

Every time any section of code accesses another segment, the 386 hardware compares the CPL to the referenced segment's DPL (i.e., it compares the privilege level the CPU is running at to the privilege defined for the segment). If the CPL is the same or higher (smaller number) than the DPL, then it can proceed with the access. A small attempt to access a segment more privileged than the CPU's CPL will result in an exception, letting us know something is wrong.

Thus, code running in a segment with a DPL of 0 pumps the CPU up to a CPL of 0, and gives the CPU access to every other segment.

Novice 8086 assembly programmers always moan about the complexity of segments and segment groups. Sometimes the ASSUMEs, GROUPs, and other pseudo-ops seem to be an awful lot of trouble. When you switch to the 386 suddenly these constructs make perfect sense: group like segments together, simultaneously protecting privilege levels. Perhaps the operating system will be grouped into one segment with a DPL of 0 so it can access any resource. Maybe device drivers can fit themselves inside of the descriptor table. That is, the call gate contains the complete protection status for each and every port.

Privilege levels are intimately tied to descriptors. As I mentioned, the descriptor contains the base address of a segment, the segment size, and access rights bits. Two of these bits specify the Descriptor's Privilege Level (DPL). Privileges are thus associated with segments, a somewhat novel concept when you consider that most CPUs simply have a global privilege setting that affects all of memory equally.
application routine calls for operating system service with a call gate. The transfer through the gate will raise the privilege level to that of the OS.

Call gates add yet another level of complexity to a program's structure, but most of the details can be left to the linker. One of the nice advantages of the gate is that every call to it uses the same selector. If the gate is defined at some sacred location that never changes from version to version, then the gate is sort of like a jump table. I've always been a big fan of using jump tables in embedded systems, so if you can figure out where routines are, even in the field with limited tools, even after 50 versions of the ROM.

Call gates are designed mostly for use when privilege level transitions are needed. Since they are stored in a descriptor table, you are limited in the number of gates the system will support. Remember that the GDT and each LDT is limited to 8k entries, which is far from infinity. Generally, gates are used to funnel requests for operating system service through a single OS dispatcher.

OTHER GOODIES

The 386 is just chock full of features for managing complex operating systems and code. This list is far too extensive to cover here in any detail. However, I'll briefly mention several other features that can help in developing any kind of system, embedded or otherwise.

The processor does support virtual memory. One of the attribute bits in every segment descriptor indicates if the segment is present. A reference to a not-present segment creates an exception, allowing system software to load the required segment from disk. Frankly, I'm not sure what this would be useful for in an embedded system, but it does seem like a neat feature. I'd welcome ideas.

The processor's memory management has yet another level beyond the segmentation I've described. Optionally, you can divide the 4 Gb address space into smaller chunks and then remap the physical address of each chunk through page tables. You define the page tables to translate practically any address into any other. Thus, two tasks could be compiled at identical addresses, yet run at different physical addresses by using different paging. Again, is this useful for an embedded system? Does someone out there have some devilishly clever technique you'd care to share with us?

The 386 does include a number of debug registers that let you set hardware breakpoints on up to 4 addresses simultaneously. These breakpoints work rather like those produced by an emulator: they are non-intrusive, and work in ROM or RAM. You can set them on code or data accesses. If you'd care to write a monitor to embed in the product (always a good idea for long term product maintenance), then by all means use these resources.

CONCLUSION

Why use protected mode in embedded applications? The biggest attraction is the large, 32 bit address space that becomes immediately available. Of course, most any other 32 bit CPU will give easier access to lots of memory.
application routine calls for operating system service with a call gate. The transfer through the gate will raise the privilege level to that of the OS.

Call gates add yet another level of complexity to a program's structure, but most of the details can be left to the linker. One of the nice advantages of the gate is that every call to it uses the same selector. If the gate is defined at some sacred location that never changes from version to version, then the gate is sort of like a jump table. I've always been a big fan of using jump tables in embedded systems, so you can figure out where routines are, even in the field with limited tools, even after 50 versions of the ROM.

Call gates are designed mostly for use when privilege level transitions are needed. Since they are stored in a descriptor table, you are limited in the number of gates the system will support. Remember that the GDT and each LDT is limited to 8k entries, which is far from infinity. Generally, gates are used to funnel requests for operating system service through a single OS dispatcher.

OTHER GOODIES

The 386 is just chock full of features for managing complex operating systems and code. This list is far too extensive to cover here in any detail. However, I'll briefly mention several other features that can help in developing any kind of system, embedded or otherwise.

The processor does support virtual memory. One of the attribute bits in every segment descriptor indicates if the segment is present. A reference to a not-present segment creates an exception, allowing system software to load the required segment from disk. Frankly, I'm not sure what this would be useful for in an embedded system, but it does seem like a neat feature. I'd welcome ideas...

The processor's memory management has yet another level beyond the segmentation I've described. Optionally, you can divide the 4 Gb address space into smaller chunks and then remap the physical address of each chunk through page tables. You define the page tables to translate practically any address into any other. Thus, two tasks could be compiled at identical addresses, yet run at different physical addresses by using different paging. Again, is this useful for an embedded system? Does someone out there have some devilishly clever technique you'd care to share with us?

The 386 does include a number of debug registers that let you set hardware breakpoints on up to 4 addresses simultaneously. These breakpoints work rather like those produced by an emulator: they are non-intrusive, and work in ROM or RAM. You can set them on code or data accesses. If you'd care to write a monitor to embed in the product (always a good idea for long term product maintenance), then by all means use these resources.

CONCLUSION

Why use protected mode in embedded applications? The biggest attraction is the large, 32 bit address space that becomes immediately available. Of course, most any other 32 bit CPU will give easier access to lots of memory.

Certainly the DOS based tools that so many non-embedded people use are a compelling incentive to stick with the 8086 architecture. How many millions use all of the great DOS Cs and assemblers? You can use any of these on the 386, and as they become more 32 bit aware they'll take even greater advantage of the 386's features. Quick development cycles demand proven tools, and it's awfully hard to argue against those from the DOS world. You can even do a lot of the development on a DOS machine, and port to the harder embedded world after removing most of the bugs.

Finally, protected mode really does protect your code. With the right segmentation, you'll never, and I mean never, see a rogue program overwrite the code. This could be important in medical and other life-critical applications.

For those wishing to explore the mysteries of this processor in more detail, be sure to get the complete set of Intel reference manuals.

Intel's "Microprocessors" manual (mine is dated 1990) contains a pretty complete hardware and software description of the part, but is definitely not for the faint hearted. It is complete but succinct.

Their "386 DX Microprocessor Programmer's Reference Manual" is far more readable, but neglects all hardware issues. It gives a pretty readable account of the operation of all of the processor's major modes. This is a must read for serious 386 users.

Intel's "80386 System Software Writer's Guide", though thin, does include lots of sample code, including routines to enter and exit protected mode. It is a good adjunct to the Programmer's Reference Manual.

Finally, the "80386 Microprocessor Hardware Reference Manual" helps explain how to design hardware that will really work with the 386. This is not a trivial problem, as the CPU can get out of sync with it's bus cycles - you have to build a sort of state machine to determine what it is doing when. Even adding wait states is a bit challenging.