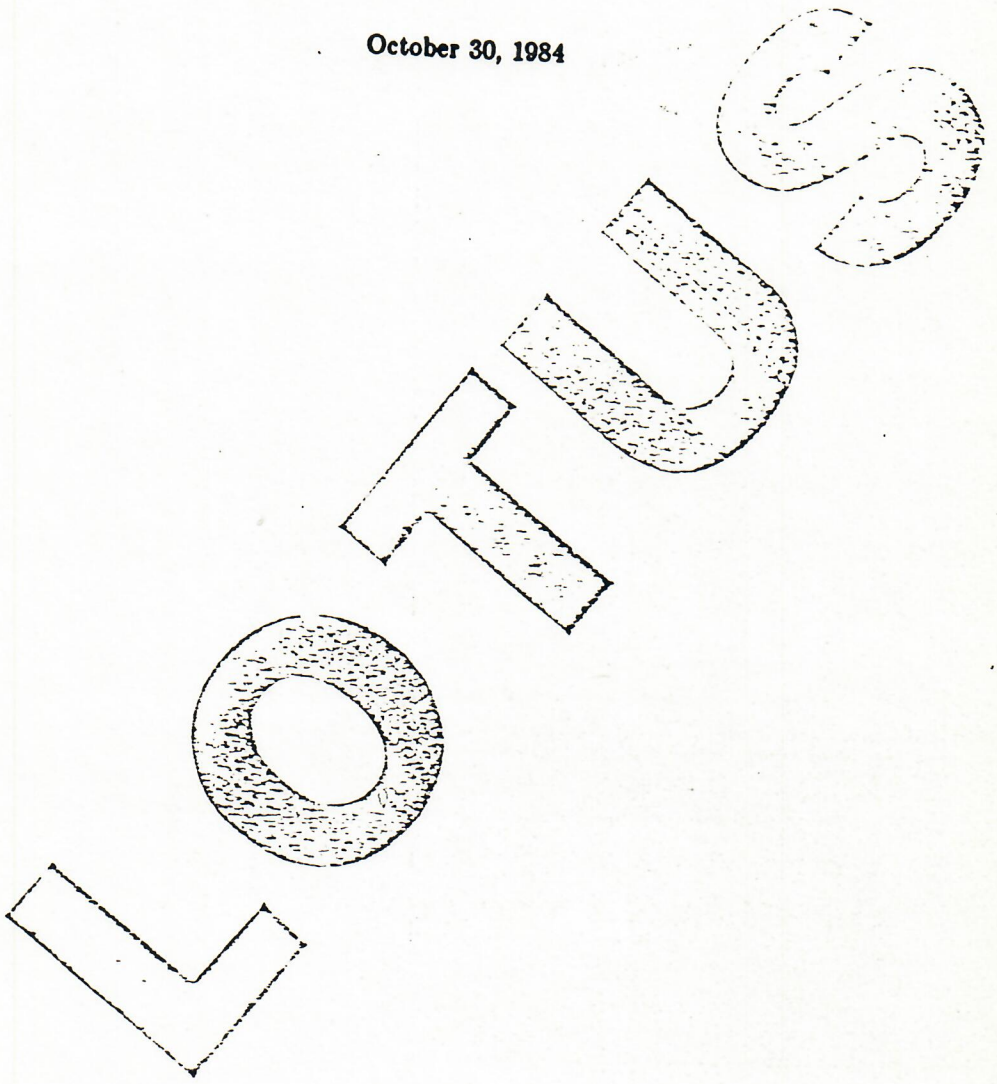
286 AND 8086 COMPATIBILITY

October 30, 1984



1. Overview

This document is one of a series of related documents. They are:

- *Microsoft Multitasking MS-DOS Product Specification OVERVIEW*
- *Microsoft Multitasking MS-DOS Product Specification DEVICE DRIVERS*
- *Microsoft Multitasking MS-DOS Product Specification SYSTEM CALLS*
- *286 and 8086 Compatibility*
- *Microsoft Multitasking MS-DOS Product Specification INTRODUCTION*
- *Microsoft Multitasking MS-DOS Product Specification MEMORY MANAGEMENT*
- *Microsoft Multitasking MS-DOS Product Specification DYNAMIC LINKING*
- *Microsoft Multitasking MS-DOS Product Specification SESSION MANAGER*

2. Introduction

The environment in which software products must run will shortly undergo some dramatic changes brought about by two new products: the 286 CPU and the MS-DOS 4.0 product.

The microcomputer industry is moving away from using machines in a one-program-at-a-time manner, as if they were sophisticated desk calculators. The time has come for the introduction of mainframe technology: multitasking, protection, and networks, as well as brand-new technology: windowing, and dynamic linking.

It is beyond the scope of this document to discuss Microsoft's plans in each of these areas; instead, this paper will concentrate specifically on the 286 CPU: what it is, what makes it different from the 8086, and how programming techniques must change to deal with it. In the interest of simplification, this paper contains some factual errors. Every programmer doing work in assembly language should obtain and study the *iAPX 286 Programmer's Reference Manual.*

The following figure illustrates the multiplicity of future Microsoft product environments:

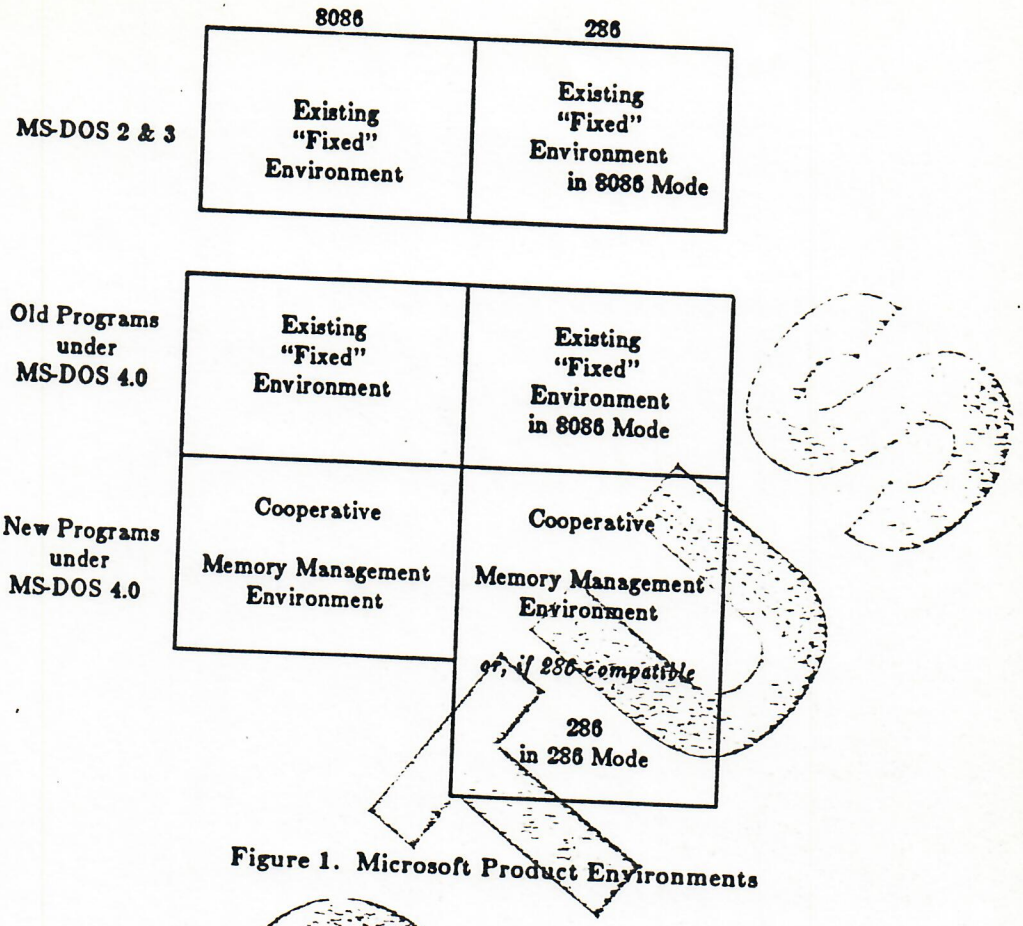


Figure 1. Microsoft Product Environments

Most readers that have not been exposed to the architectures of the 286 or MS-DOS 4.0 will undoubtedly find Figure 1 incomprehensible. It is presented now to make the point that there will be a multiplicity of environments and that each one has sufficient virtues to justify its existence. This document concentrates on the right half of the table: the 286 and its two modes, "8086" and "286."

The material presented here is intended to point out those aspects of the 286 processor which will affect the design and compatibility of microcomputer software.

The key issue throughout this discussion, and throughout all work at Microsoft, is compatibility. An ISV cannot afford to have separate versions of its products for each of the boxes drawn here. The effort required to stock and ship many different versions of a product is staggering. Further, retailers would rebel due to the ensuing customer confusion and the demands on their limited stockroom and shelf space.

Also, a classic "catch-22" situation arises. If a new version of a program is required to run under MS-DOS 4.0 that will only run under MS-DOS 4.0, then ISVs will not want to convert their products until many copies of MS-DOS 4.0

have been sold. Similarly, customers will not want to purchase MS-DOS 4.0 until many programs are available for it.

Microsoft resolves this situation by providing both upward and downward compatibility. The new environment is designed to allow old programs to run unchanged (upwardly compatible), and to allow most programs written for the new environment to run under the old environment (downwardly compatible).

The ultimate purpose of this paper is to allow for the creation of programs that are fully compatible across this spectrum of environments and that also take maximum advantage of the extra features of each environment.

3. 286 Compatibility

The Intel 80286 ("286") processor is the next-generation follow-on for the now familiar 8086/8088 processor. The 286 differs from the 8086 in the following ways:

1) It is much faster. It is approximately three times faster than the 8088 in the IBM PC/XT.
2) It includes extra instructions, such as "shift by count" and "push immediate."
3) The 286 supports two operating modes that Intel calls "286" (also known as virtual or protected mode) and "real." In this document, these modes will be called "286" and "8086", as the "real" mode is intended to be 8086-compatible.

   Note: The "8086" mode does not *exactly* emulate an 8086/8088; there are some small but meaningful differences.

The 286 will be a popular processor; Microsoft, for example, is buying a 286-based machine for each programmer. However, it will be years before the 286 replaces the 8086 as the dominant microprocessor. For this reason, it is of critical importance that all software be written so that a binary copy runs on either a 8086 or a 286 (so-called "binary compatibility"). The good news is that the machines are similar enough so that this is feasible. The bad news is that the two processors are not exactly identical, so care must be taken when designing software.

When stating that 8086 programs must be binary-compatible with the 286, the question "compatible with the 286 in 8086 mode or in 286 mode?" is raised. Ideally, the stronger condition, "compatibility with 286 mode," should be met. XENIX, for example, runs on the 286 only in 286 mode. MS-DOS 3.0 runs only in "8086" mode; future versions of MS-DOS will eventually run in both modes. If the program is 286-mode-compatible, it is maximally flexible and, when running in 286 mode, it can take advantage of some powerful 286 features.

This document discusses the requirements for compatibility first with the 286 in 8086 mode, and then in 286 mode.

3.1. Compatibility in 8086 mode

The 286, running in "8086-compatible" mode, closely emulates the 8086/8088 processor. Programmers should note the following exceptions and rules:

1) The 8086 generates physical memory addresses by adding the offset value to (16 * segment-value), yielding a 20-bit result. Any overflow is ignored, so it is possible to wrap-around to low memory by using sufficiently large segment and offset values. For example, on a true 8086, the address FFF0:20 references the same location as the address 0000:10. The 286, however, will generate a physical address of 10000:10, or "one megabyte plus 16." This is possible because the 286 has 24 address lines whereas the 8086 has only 20.

   Existing versions of MS-DOS allow programs to call the operating system by calling to location 5 in the PDB area. This mechanism makes implicit use of "wrap-around" addressing and therefore should not be used. MS-DOS supports this mechanism to provide compatibility with ancient CP/M programs. All new programs should use the "INT 21" interface, instead.

   Some machines (such as the IBM AT) contain special hardware to fix this problem by ignoring that 21st bit when in 8086 mode, but software should not rely upon this.

   Rule 1: Don't use "wrap-around" addressing.

   Use "INT 21" instead of "CALL 5".

2) When executing the instruction

```
PUSH    SP
```

   the 286 pushes the SP value *before* the push instruction, the 8086 pushes the value which SP will be *after* the push. This is an unusual sequence that doesn't appear in most code. If it should occur, the sequence:

```
MOV     AX,SP
PUSH    AX
```

   should be used.

   Rule 2: Don't use the instruction PUSH SP.

3) Shifts and rotate counts contained in the CL register are masked to 5 bits on the 286; they remain 8 bits wide on the 8086. Since the masked bits are all multiples of 16, most shifts and rotates with large counts produce the same result with a masked-off shift count. The exceptions are the rotate-with-carry instructions which involve 17 bits and can produce different results with shift counts greater than 31. Since larger shift counts waste CPU time, Microsoft software should never generate them.

Rule 3: Don't use shift counts greater than 31.

4) The 286 can generate "the most negative number" as a quotient for the IDIV instruction. The 8086 generates a divide error exception. (This occurs for quotients of 8000h (word) and 080h (byte).

Rule 4: Don't use IDIV operands that produce the most negative number.

5) After a divide error trap, the 286 will point at the divide instruction, including prefixes. The registers will be unchanged. The 8086 will point *after* the divide instruction and may change DX:AX or AH:AL, as appropriate.

Rule 5: Divide trap handlers should either not resume execution in the original code stream or they will have to detect and understand 286/8086 differences. The latter can easily be done by intentionally generating a divide trap during initialization, and examining the value on the stack.

6) Some 80186 processors have a mask bug such that the IDIV instruction operates incorrectly if the divisor is negative, and it is located in memory. In this one case, the result in the AX(AL) register is the two's complement of the correct answer. If the divisor is in a register, the instruction works correctly.

Rule 6: Don't use the memory-operand form of the IDIV instruction.

7) The 286 has some additional instructions that are not present on the 8086. These are:

push	<immediate>
PUSHA	
POPA	
IMUL	<immediate>
shift	<reg>,<count>
INS	
OUTS	
ENTER	
LEAVE	
BOUND	

Although it is tempting to use these since they fill in some obvious gaps in the 8086 instruction set, that temptation should be avoided unless the product is "286-only."

Rule 7: Don't use the 286 new instructions.

8) The 286 has an instruction length limit of 10 bytes. This is longer than any possible instruction unless the prefix bytes are repeated; don't repeat prefix bytes.

Rule 8: Don't use repeated (redundant) prefix bytes.

9) Don't use undefined opcodes in any processor at any time.

Rule 9: Never use undefined opcodes.

10) Don't rely upon the speed of the CPU for timing. Various machines use CPUs of different speeds, and naturally, the 286-as-an-8086 is much faster than a true 8086.

Rule 10: Don't depend upon CPU speeds for any purpose.

11) Don't examine the flag register too closely. PUSHF followed by POPF may change the contents of the flag register as there are more flag bits defined in the 286 flag word. Programs should set and test flags using the standard instructions, not set or examine specific bit patterns via LAHF or SAHF.

Rule 11: Never examine or set explicit flag register values; set and test the values only via flag-specific instructions.

12) Don't single-step an INT instruction. The 8086 and the 286 behave differently in this respect. This should be an extremely rare situation, limited to debuggers.

Rule 12: Don't single-step an INT instruction.

13) All of the existing 286 chips have a mask bug which messes up the POPF instruction. Specifically, if interrupts are off and a POPF is done to restore a flag word that also has interrupts off, an interrupt *may* be granted anyway. The POPF instruction should be replaced by a macro POPFF, defined as:

```
POPFF   MACRO
        JMP     $+3
        IRET
        PUSH    CS
        CALL    $-2
        ENDM
```

LOTUS

This macro uses "PUSH CS" and a short call to setup the stack so that an IRET can be used to restore the flags. (Unfortunately, the LAHF and SAHF instructions share the POPF bug.)

Rule 13: Don't use POPF.

14) Programs with self-modifying code may work differently on the 286 due to its extended prefetcher.

Rule 14: Don't write self-modifying code.

****

The previous exceptions are all related to the design of the 286 itself. The following exception applies specifically to the IBM AT machine which contains a 286. Since the AT machine will be popular, the following restriction should be obeyed by all programs.

****

AT) Don't do back-to-back IN and OUT instructions. Do a short-jump between otherwise sequential I/O. This is needed because of bus and device timing limitations. For example:

```
        Good                    Bad

        IN      al,30           IN      al,30
        JMP     SHORT L1        OUT     40,al
L1:     OUT     40,al
```

Rule AT: Don't do back-to-back I/O.

To summarize, compatibility between the 8086/88 and the 286 in "8086 compatible" mode is relatively easy to achieve, but does require some explicit effort:

Rule 1: Don't use "wrap-around" addressing.

Rule 2: Don't use the instruction PUSH SP.

Rule 3: Don't use shift counts greater than 31.

Rule 4: Don't use IDIV operands that produce the most negative number.

Rule 5: Divide trap handlers should either not resume execution in the original code stream or they will have to detect and understand 286/8086 differences.

Rule 6: Don't use the memory-operand form of the IDIV instruction.

Rule 7: Don't use the new 286 instructions.

Rule 8: Don't use repeated (redundant) prefix bytes.

Rule 9: Never use undefined opcodes.

Rule 10: Don't depend upon CPU speeds for any purpose.

Rule 11: Never examine or set explicit flag register values; set and test the values only via flag-specific instructions.

Rule 12: Don't single-step an INT instruction.

Rule 13: Don't use POPF.

Rule 14: Don't write self-modifying code.

Rule AT: Don't do back-to-back I/O.

3.2. Compatibility in 286 Protected Mode

Binary compatibility between the 8086/88 and the 286 in 286 mode involves more effort. Fortunately, proper programming practice makes it easy to avoid these incompatibilities because most instructions are binary compatible. The 286 in 286 mode has considerably different memory management than the 8086. Understanding these differences and their origins can best be approached by examining the structure of the 286 architecture, specifically, how it handles memory management and protection.

4. What is Memory Management?

The earliest computers had no memory management hardware. Memory started at location 0 and ran to location *N*. Only one program was run at one time; when a program was loaded into memory the operating system would relocate it to wherever free memory started. This relocation was possible because the load image (.EXE file) contained a list of all the spots in the program that contained addresses. The linker would put the program together as if it were to run starting at location 0; the operating system would add an offset to each listed location so that they would indicate the program's actual load address. This process is called ***relocation.***

Early computers had no memory management hardware because a relocating loader is sufficient for a single task. Exactly the same is true of the earliest microcomputers, the 8080 and the 8086. They don't have relocation hardware, but the DOS's relocating loader allows programs to be loaded into memory at whatever location the DOS prefers. The difficulty comes when we want to run more than one program at one time, or "multitask." Where should the second program be loaded?

An obvious idea is to relocate the second program into memory right after the first one. This works for a brief period, but soon the first program wants to extend its memory area and the second program is in the way. Although the second program can be loaded anywhere (because all the locations in the .EXE file containing addresses can be listed), once the program has started running it can't be moved because a running program may have return addresses on the stack and data pointers scattered throughout memory. Finding all the addresses that need updating in an arbitrary program is impossible.

Another problem arises when the first program finishes execution before the second program does. If a third program is to be run, and if it is larger than both the first program's prior space and the space remaining after the second program, it will have to wait until the second program finishes execution. There is enough free memory to run the third program, but the immovable second program has fragmented memory and made it unusable for the needs of the third program.

The problem of moving a running program in memory without its explicit cooperation was solved with the invention of memory relocation hardware. This hardware automatically takes every memory address generated by a program and adds the contents of a relocation or displacement register to it. The resultant sum is then used as the true memory address. With the aid of such hardware, the operating system doesn't need a relocating loader; it just puts the load address of the program in the relocation register and the program "thinks" it is truly running at location 0. If the user wants to run a second program at the same time, it can be loaded wherever there is free memory; before control is transferred to it, the contents of the relocation register can be changed to point to the second program. The second program then "thinks" that it is also running at location 0.

In this manner the OS moves programs around in memory whenever it is necessary. Each time, however, the OS checks that the relocation register is loaded with the proper value before the program is run.

4.1. 8086 Memory Model Reviewed

Intel has finally released a microprocessor with relocation hardware: the 80286. Unfortunately, the 286 implements relocation hardware in a way that is not fully compatible with the 8086.

The 8086 performs memory addressing by using a pair of 16-bit values: a segment value and an offset value. For each memory reference the CPU multiplies the segment value by 16 and adds it to the offset value, producing a 20-bit true address.

The name "segment register" is a misnomer; this is actually a form of a relocation register. When Intel designed the 8086, they wanted to address more than 65K bytes of memory, but to retain a 16-bit architecture. They did this by addressing only 65K bytes at a time, but they supplied a displacement register allowing the program to address any particular 65K piece of the 1 megabyte of memory. Also, the program can change the value in the displacement register (officially, "segment register") at any time.

Note: This description is simplified. There are actually four displacement registers instead of one, but the point remains valid.

It may seem contradictory to claim that the 8086 has no hardware relocation while describing the 8086 "segment registers" as actually being relocation registers. However, in a "proper" memory relocation unit, the actual relocation is invisible to the program; only the operating system loads in the relocation value. The problem with the 8086 architecture is that the *program* is responsible for placing the relocation values in the segment registers; this just pushes the problem up a level. To move a running 8086 program in physical memory, it is necessary to find and edit all the relocation values that the program might eventually load into a segment register. Although there are fewer relocation values than there are memory addresses, they are impossible to find, and 8086 programs that make use of the segment registers are just as immovable once they have started running.

One small loophole exists here: if a program is written in such a way that it *never* makes any references to the segment registers, it *never* makes long calls, *never* pushes, pops, or loads the segment registers, then the DOS can indeed move that program around in memory by using the segment registers as relocation registers. Although many programs make scant use of the segment registers, those that make *absolutely no* use of these registers are almost nonexistent. Unfortunately, once a program does make use of segment registers, it then "knows" about absolute addresses, and cannot be dynamically relocated.

See Figure 2 for a pictorial representation of 8086 memory addressing. Note that each memory location can be referenced by 4096 different "segment:offset" pairs. Location "0002:0000" is the same as "0000:0020" which is the same as "0001:0010", etc. This figure illustrates the reason to avoid calling these segments "segments" and the registers "segment registers." They are not segments at all, in the strict sense of the word, but a series of overlapping 65K "windows" mapped onto a 1 megabyte address space. As long as a byte's 20-bit physical address is known, a program can generate any one of 4096 "segment:offset" pairs to reference the byte.

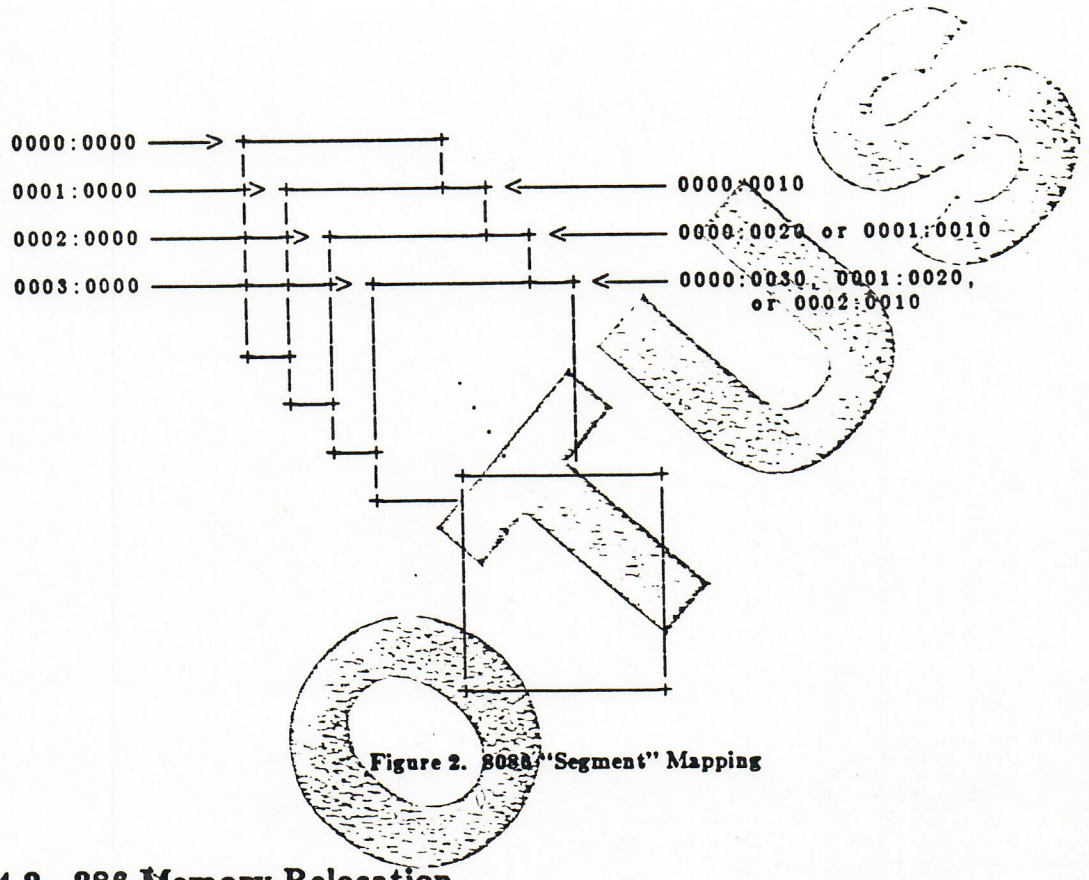


Figure 2. 8088 "Segment" Mapping

4.2. 286 Memory Relocation

The 286, running in 286 mode, changes this process considerably. Note that this process is being described strictly from a memory management viewpoint; for now, some important elements of the 286 design related to protection will be omitted.

The designers of the 286 wanted to add a true form of memory management to the system, and they wanted to increase the limit of addressable memory from 1 megabyte to 16 megabytes. To accomplish this, they added a second layer of relocation, one that is invisible to the running program. They also did not want their new mechanism to require any new or different program instructions. They thought this would assure upward compatibility with 8086 programs. This was untrue, however, as will be discussed later.

When a program loads a "segment number" into a "segment" register on an 8086, it is actually loading a displacement value. When a program loads a "segment number" into a segment register on the 286, the 286 uses the number to locate an entry in a master segment table, and then loads the displacement value for the segment from that table. This is the extra level of indirection mentioned. On the 8086, a segment register contains the displacement; on the 286, it contains an index into a table that contains the true displacement.

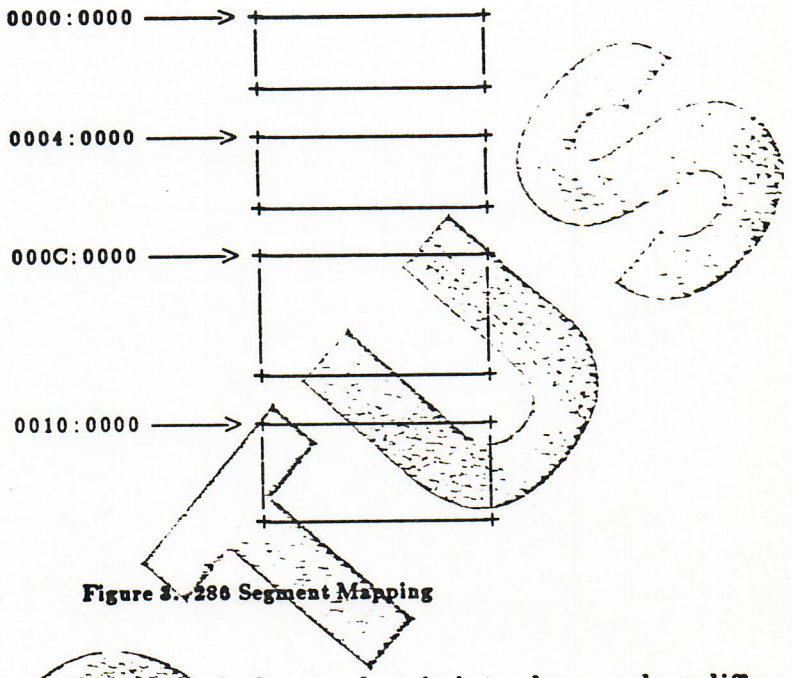


Figure 3. 286 Segment Mapping

This seems like a straightforward change, but it introduces a key difference in the semantics of the values placed in segment registers. Examine Figure 3 and recall that on the 8086 the segment value represents the high order bits of the ultimate physical address. All but 16 memory locations in segment *N* are also in segment *N*+1 (or *N*-1). On the 286, however, segment *N* bears no particular relationship to segment *N*+1; they may be far apart in physical memory. Usually *no* memory location in segment *N* is also in segment *N*+1.

4.3. What is a "Segment"?

A "segment" is a chunk of memory of variable size. A segmented machine has a two-dimensional address space: one must specify which segment, and then specify which location within that segment. Although the physical memory assigned to different segments may overlap partially or fully so that they share some memory locations, this is atypical. As a general rule, each memory location has only one valid address, and it takes the form segment:offset.

Thus, 8086 so-called "segments" are not segments at all according to the above definition. On the 8086, if the 20-bit physical address of a byte is known, the 4096 different segment:offset pairs to address the byte can be generated.

Similarly, if the segment:offset for a byte is known, the 20-bit physical address and/or other segment:offset pairs that address the same location can be calculated. Thus, the 8086 segment registers do not truly specify segments; they are merely relocation registers.

The 286, on the other hand, has true segments. Each memory location accessible to a program usually has only one segment:offset pair to address it. When the segment:offset pair for a memory location is known, a program cannot compute some other segment:offset pair to address the same location. This is the critical incompatibility between the 8086 and the 286 in 286-mode. Nearly all programs contain code that assumes a relationship between physical addresses and "segment numbers," a relationship that does not apply on the 286.

Some programs do this to an extreme. They reduce the 32-bit segment:offset representation of an address into a 20-bit physical address. When the programs reference the locations they use, they crack the 20-bit value into one of the 4096 valid segment:offset pairs to make the reference. Other programs assume the relationship in a more subtle way; for example, a program might want to address memory just beyond the 65K available via the DS register, and do it by the sequence:

```
MOV   AX,DS
ADD   AX,4096
MOV   ES,AX
```

This sequence also contains a built-in assumption that the physical address is (segment*16)+offset, a relationship that does not apply on the 286.

4.4. Making Programs Compatible

At first glance, the answer to the problem seems obvious: simply enter the proper offset values into the 286 master segment table so that the 8086 relationship is maintained; segment N+1 addressing the 65K which starts 16 bytes above the 65K for segment N. This would, indeed, solve the problem, except for some of the 286 details omitted until now: the 286 only supports 8192 segments, not 65536. The two low-order bits of the "segment number" are used as part of the 286 protection mechanism and are not actually part of the segment number. The 286 examples should have read N and N+4, since N+1 is the same segment as N, at a different protection level.

Since the 286 true-segments can't be made to behave like the 8086 pseudo-segments, compatibility must be obtained by writing 8086 programs so that they behave in a "true segment" manner and don't make any assumptions about how segment numbers are related to physical memory and each other. The term *segment games* is used for code sequences that understand the relationship between 8086 segment values and physical memory; a compatible program must not contain any segment games.

4.4.1. Avoiding Segment Games

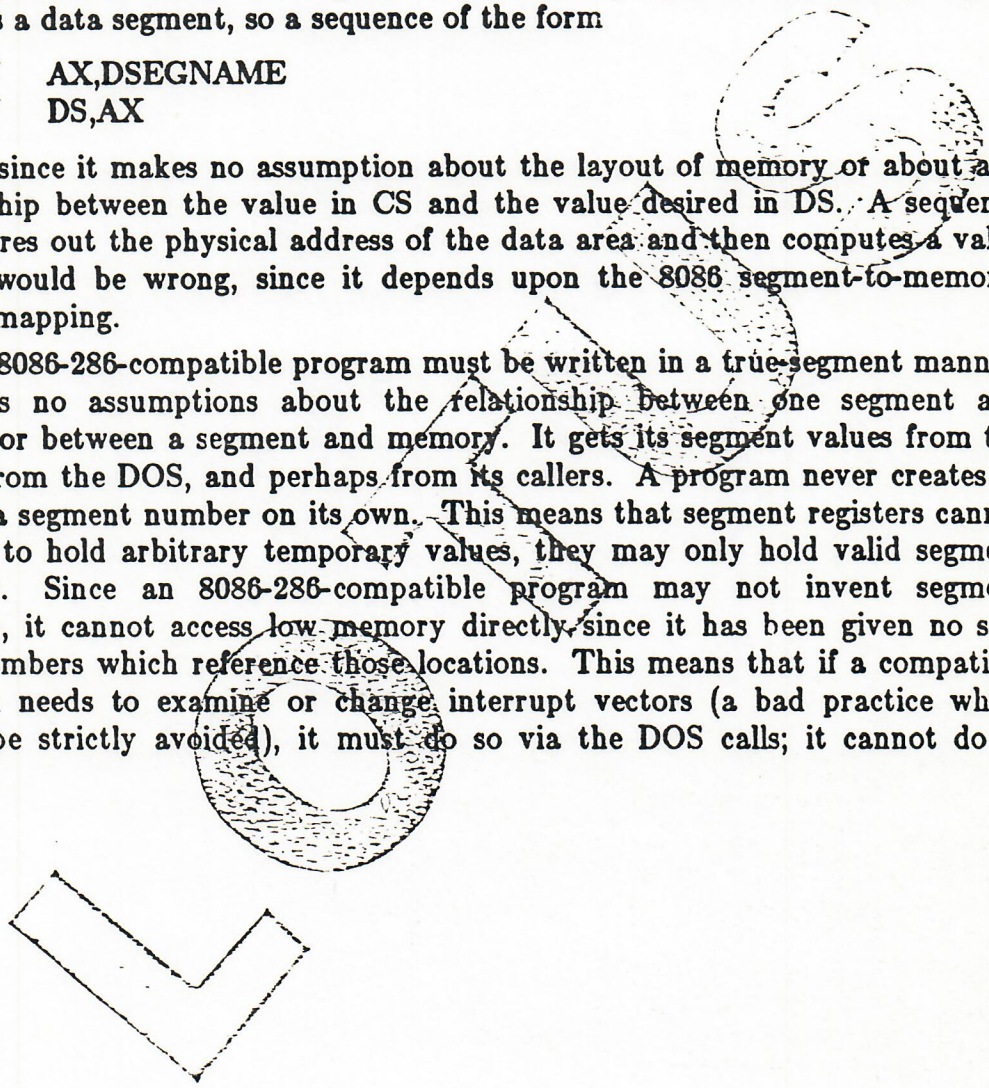
When the DOS starts an .EXE program it sets the initial CS to the proper value. If a program contains long-calls, the segment value in those calls is adjusted by the DOS to its proper value, so programs rarely play segment games with CS.

The DS and ES registers are different. Currently the DOS sets DS and ES to point to the Program Data Block (PDB, also known as the Program Prefix Header), 100h bytes before the first code segment. It is up to the program's initializer routine to find a good value for DS, and this is a critical matter. It should be thought about in terms of segments, not memory layout. Presumably a program has a data segment, so a sequence of the form

```
MOV   AX,DSEGNAME
MOV   DS,AX
```

is good, since it makes no assumption about the layout of memory or about any relationship between the value in CS and the value desired in DS. A sequence that figures out the physical address of the data area and then computes a value for DS would be wrong, since it depends upon the 8086 segment-to-memory-address mapping.

An 8086-286-compatible program must be written in a true-segment manner. It makes no assumptions about the relationship between one segment and another or between a segment and memory. It gets its segment values from the linker, from the DOS, and perhaps from its callers. A program never creates or invents a segment number on its own. This means that segment registers cannot be used to hold arbitrary temporary values, they may only hold valid segment numbers. Since an 8086-286-compatible program may not invent segment numbers, it cannot access low memory directly since it has been given no segment numbers which reference those locations. This means that if a compatible program needs to examine or change interrupt vectors (a bad practice which should be strictly avoided), it must do so via the DOS calls; it cannot do so directly.



5. 286 Protection Features

The 286 master segment table contains more than just the segment displacement value. It also contains an assortment of bits and fields relating to protection. When running in 8086-compatible mode, these fields and bits are ignored, but they carry important meaning in 286 mode.

There are three major protection-related pitfalls to avoid when writing programs that are 8086/286-compatible. They are:

- segment sizes,
- impure code segments, and
- privileged instructions.

These will be discussed in the next sections.

5.1. Segment Sizes

Recall the earlier emphasis that 8086 "segments" are not segments at all, but overlapping "windows" into physical memory. 286 segments are true segments and as such each has its own specific size. Given a program with 32K of code, in 8086 mode the address

CS:8000

is valid, although it is unclear what may be stored at that location. In 286 mode the "master segment table entry" (henceforth called the "segment descriptor"), contains a limit field. Any offset larger than (in this case) 32K used to reference that segment will generate a machine trap and the program will be aborted. Furthermore, a segment offset in 286 mode is limited to 65K and adjusting the segment number merely points to a new segment instead of pointing further into the existing segment.

This should be the easiest of the protection restrictions, since even in 8086 mode no program should be addressing beyond its allocated memory. Such a badly written program will probably run under MS-DOS 2.0 because the memory it illegally accesses is most likely unused. It will sometimes run a multitasking environment in 8086 mode because that memory probably belongs to some other program. It will always crash in 286 mode because the 286 protection hardware will protect the system and other programs from the illegal reference.

Note that a characteristic of a true segment is that it is limited to 65K, because it is impossible to generate a larger offset. Playing with the segment number does not address further into a segment but instead selects a different segment. This means that a program that is to be 286-mode-compatible cannot use the DOS ALLOC call to allocate more than 65K at a time because the memory above 65K would be unreachable. If the program needs 120K it will have to request it as two or more segments.

5.2. Impure Code Segments

An unfortunate restriction built into the 286 does not allow writable code segments. One of the bits in the 286 segment descriptor indicates a CODE or DATA segment. Although the DS and ES registers may contain the selector of a CODE segment the CS register may only contain selectors of CODE segments.

A DATA segment's descriptor has a bit to indicate read/write or read-only access, but a CODE segment's descriptor does not. As a consequence:

- Only valid CODE segments may be placed in CS.
- A program may not write into valid CODE segments, even if the segment selector is copied from CS to DS or ES.

It has been a common practice for programmers to place a program's data in the code segment, often intermingled with the code itself. Then, by loading DS with the CS value one can minimize the amount of arguing with the assembler and linker about segments and groups. Unfortunately this convenient practice is no longer acceptable since it prevents programs from being 286-mode-compatible.

5.3. Privileged Instructions

The final consequence of 286 mode is the existence of privileged instructions. A privileged instruction is one that might affect the system as a whole, and not just the environment of the issuing program. For this reason, such instructions are *not* obeyed when issued by a program in 286 mode. Instead, they cause a trap that will abort the program. Privileged instructions fall into three classes:

1) Instructions that deal with the 286 protection and memory management hardware. None of these instructions exist on the 8086, so all existing programs conform to this restriction.
2) I/O instructions: IN and OUT. A 286-compatible program cannot issue any I/O instructions.
3) CLI. A 286-compatible program cannot issue the CLI instruction; it causes a protection trap. An IRET instruction restores the previous value of the interrupt flag in 8086 mode but has no effect on the IFLG in 286 mode. A software INT instruction will not disable interrupts in 286 mode, although it will in 8086 mode.

6. What's Next

286 MS-DOS machines (running in 8086-compatibility mode) are already in the field. Currently, it is necessary to review all existing software for conflicts with the 286's compatible-mode limitations.

Once the 286-as-an-8086 issues have been resolved, it is necessary to consider the restrictions required to run programs in 286 mode as well as 8086 mode. As has been discussed, these restrictions are significant but not insurmountable. Although many of these restrictions are caused by the 286 itself and are thus inflexible, some of the restrictions can be eased by special support code in the DOS and/or by the technique of dynamic linking, described in the MS-DOS document, *Microsoft Multitasking MS-DOS Product Specification DYNAMIC LINKING.*

As long as a program is written in a high level language, the compiler and linker will take care of these 286 compatibility concerns. Programs which use special compiler features to directly reference the machines underlying architecture need to ensure 286 compatibility themselves.