Real Mode

From OSDev Wiki

Real Mode is a simplistic 16-bit mode that is present on all x86 processors. Real Mode was the first x86 mode design and was used by many early operating systems before the birth of Protected Mode. For compatibility purposes, all x86 processors begin execution in Real Mode.

Contents

Information

All modern operating systems (Windows, Linux, ...) run in Protected Mode, due to the many limitations and problems that Real Mode presents (see below). Older operating systems (such as DOS) and programs ran in Real Mode because it was the only mode available at the time. For information on how to switch from Real Mode to Protected Mode, see the corresponding article.

Note: There is a mode called Virtual 8086 Mode which allows operating systems running in Protected mode to emulate the Real Mode segmented model for individual applications. This can be used to allow a Protected Mode operating system to still have access to e.g. BIOS functions, whenever needed.

Below you'll find a list of cons and pros. These are mostly 'in comparison to Protected Mode'.

Cons

- Less than 1 MB of RAM is available for use.
- There is no hardware-based memory protection (GDT), nor virtual memory.
- There is no built in security mechanisms to protect against buggy or malicious applications.
- The default CPU operand length is only 16 bits.
- The memory addressing modes provided are more restrictive than other CPU modes.
- Accessing more than 64k requires the use of segment register that are difficult to work with.

Pros

- The BIOS installs device drivers to control devices and handle interrupt.
- BIOS functions provide operating systems with a advanced collection of low level API functions.
- Memory access is faster due to the lack of descriptor tables to check and smaller registers.

Common Misconception

Programmers often think that since Real Mode defaults to 16 bits, that the 32 bit registers are not accessible. This is not true.

All of the 32-bit registers (EAX, ...) are still usable, by simply adding the "Operand Size Override Prefix" (0x66) to the beginning of any instruction. Your assembler is likely to do this for you, if you simply try to use a 32-bit register.

Memory Addressing

In Real Mode, there is a little over 1 MB of "addressable" memory (including the High Memory Area). See Detecting Memory (x86) and Memory Map (x86) to determine how much is actually **usable**. The usable amount will be much less than 1 MB. Memory access is done using Segmentation via a segment:offset system.

There are six 16-bit segment registers: CS, DS, ES, FS, GS, and SS. When using segment registers, addresses are given with the following notation (where 'Segment' is a value in a segment register and 'Offset' is a value in an address register):

```
12F3  :  4B27
  ^       ^
Segment   Offset
```

Segments and Offsets are related to physical addresses by the equation:

```
PhysicalAddress = Segment * 16 + Offset
```

Thus, 12F3:4B27 corresponds to the physical address 0x17A57. Any physical address can be represented in multiple ways, with different segments and offsets. For example, physical address 0x210 can be 0020:0010, 0000:0210, or 0021:0000.

The Stack

SS and SP are 16-bit segment:offset registers that specify a 20-bit physical address (described above), which is the current "top" of the stack. The stack stores 16-bit words, grows downwards, and must be aligned on a word (16-bit) boundary. It is used every time a program does a PUSH, POP, CALL, INT, or RET opcode and also when the BIOS handles any hardware interrupt.

High Memory Area

If you set DS (or any segment register) to a value of 0xFFFF, it points to an address that is 16 bytes below 1 MB. If you then use that segment register as a base, with an offset of 0x10 to 0xFFFF, you can access physical memory addresses from 0x100000 to 0x10FFEF. This (almost 64 kB) area above 1 MB

is called the "High Memory Area" in Real Mode. Note that you have to have the A20 address line activated for this to work.

Addressing Modes

Real Mode uses 16-bit addressing mode by default. Assembly programmers are typically familiar with the more common 32-bit addressing modes, and may want to make adjustments -- because the registers that are available in 16-bit addressing mode for use as "pointers" are much more limited. The typical programs that run in Real Mode are often limited in the number of bytes available, and it takes one extra byte in each opcode to use 32-bit addressing instead.

Note that you can still use 32-bit addressing modes in Real Mode, simply by adding the "Address Size Override Prefix" (0x67) to the beginning of any instruction. Your assembler is likely to do this for you, if you simply try to use a 32-bit addressing mode. But you are still constrained by the current "limit" for the segment that you use in each memory access (always 64K in Real Mode -- Unreal Mode can be bigger).

- [BX + val]
- [SI + val]
- [DI + val]
- [BP + val]
- [BX + SI + val]
- [BX + DI + val]
- [BP + SI + val]
- [BP + DI + val]
- [address]

Switching from Protected Mode to Real Mode

As noted above, it is possible for a Protected mode operating system to use Virtual 8086 Mode mode to access BIOS functions. However, VM86 mode has its own complications and difficulties. Some OS designers think that it is simpler and cleaner to temporarily return to Real Mode on those occasions when it is necessary to access a BIOS function. This requires creating a special Ring 0 program, and placing it in a physical memory address that can be accessed in Real Mode.

The OS usually needs to pass an information packet about which BIOS function to execute.

The program needs to go through the following steps:

1. Disable the interrupts:
    - Turn off maskable interrupts using CLI.
    - Disable NMI (optional).
2. Turn off paging:
    - Transfer control to a 1:1 page.
    - Ensure that the GDT and IDT are in a 1:1 page.
    - Clear the PG-flag in the zeroth control register.
    - Set the third control register to 0.
3. Use GDT with 16-bit tables (skip this step if one is already available):
    - Create a new GDT with a 16-bit data and code segment:
        - Limit: 0xFFFFF
        - Base: 0x0
        - 16-bit

        - Privilege level: 0
        - Granularity: 0
        - Read and Write: 1
    - Load new GDT ensuring that the currently used selectors will remain the same (index in cs/ds/ss will be copy of original segment in new GDT)
4. Far jump to 16-bit protected mode:
    - Far jump to 16-bit protected mode with a 16-bit segment index.
5. Load data segment selectors with 16-bit indexes:
    - Load ds, es, fs, gs, ss with a 16-bit data segment.
6. Load real mode IDT:
    - Limit: 0x3FF
    - Base 0x0
    - Use lidt
7. Disable protected mode:
    - Set PE bit in CR0 to false.
8. Far jump to real mode:
    - Far jump to real mode with real mode segment selector (usually 0).
9. Reload data segment registers with real mode values:
    - Load ds, es, fs, gs, ss with appropriate real mode values (usually 0).
10. Set stack pointer to appropriate value:
    - Set sp to stack value that will not interfere with real mode program.
11. Enable interrupts:
    - Enable maskable interrupts with STI.
12. Continue on in real mode with all bios interrupts.

x86 Assembly Example

```
[bits 16]

idt_real:
        dw 0x3ff                ; 256 entries, 4b each = 1K
        dd 0                    ; Real Mode IVT @ 0x0000

savcr0:
        dd 0                    ; Storage location for pmode CR0.

Entry16:
        ; We are already in 16-bit mode here!

        cli                     ; Disable interrupts.

        ; Need 16-bit Protected Mode GDT entries!
        mov eax, DATASEL16      ; 16-bit Protected Mode data selector.
        mov ds, eax
        mov es, eax
        mov fs, eax
        mov gs, eax
        mov ss, eax


        ; Disable paging (we need everything to be 1:1 mapped).
```

```
        mov eax, cr0
        mov [savcr0], eax       ; save pmode CR0
        and eax, 0x7FFFFFFe     ; Disable paging bit & enable 16-bit pmoc
        mov cr0, eax

        jmp 0:GoRMode           ; Perform Far jump to set CS.

GoRMode:
        mov sp, 0x8000          ; pick a stack pointer.
        mov ax, 0               ; Reset segment registers to 0.
        mov ds, ax
        mov es, ax
        mov fs, ax
        mov gs, ax
        mov ss, ax
        lidt [idt_real]
        sti                     ; Restore interrupts -- be careful, unhar
```

See Also

Articles

- BIOS
- System Initialization (x86)
- Protected Mode

External Links

- The Workings of: x86-16/32 RealMode Addressing (http://www.osdever.net/tutorials/rm_addressing.php?the_id=50) (2003)
- The workings of IA32 real mode addressing and the a20 line (http://therx.sourceforge.net/osdev/files/ia32_rm_addr.pdf) (2004)

References

- [http://www.intel.com/products/processor/manuals/ Intel® 64 and IA-32 Architectures Software Developer's Manual

Volume 3A: System Programming Guide, Part 1,Chapter 17:8086 EMULATION,which is a detailed reference about real mode using 32-bit addressing mode] (2011)

Categories: X86 CPU | Operating Modes
