

Ruslan Ablyazov

# **Programming in Assembly on the x86-64 Platform**

Moscow, 2011

## Publication Information

**Ruslan Ablyazov.** *Programming in Assembly on the x86-64 Platform.* Moscow: DMK Press, 2011. 304 pages, illustrated.

**ISBN 978-5-94074-676-8**

This book discusses processor operation in its two principal modes: protected mode and 64-bit mode, also called long mode. In addition to presenting the principles and mechanisms of processor operation in protected and 64-bit modes, it discusses assembly programming under the Windows family of operating systems, in both 32-bit and 64-bit versions.

The book covers not only the development of ordinary Windows applications, but also the development of drivers in assembly language. Particular attention is paid to the practical side: the material is presented directly and is limited to what any system and low-level programmer needs to know. The final section is devoted to the principles of multiprocessor systems and to working with the Advanced Programmable Interrupt Controller (APIC).

The disc supplied with the book contains the complete source code for the examples, along with additional programs and materials.

The publication is intended for system and low-level programmers, as well as students and instructors in technical specialties at higher-education and secondary vocational institutions.

All rights reserved. No part of the original book may be reproduced in any form or by any means without written permission from the copyright holders.

The material in the book was checked repeatedly. However, because technical errors may still exist, the publisher cannot guarantee absolute accuracy or correctness and accepts no responsibility for possible errors connected with use of the book.

© Ruslan Ablyazov, 2011

© Design and publication, DMK Press, 2011

English revised edition note: This corrected English translation uses a revised digital layout of 153 PDF pages. The 304-page figure above describes the original 2011 Russian print edition.

# Contents

Page numbers and links refer to this revised English PDF edition.

|                                            |    |
|--------------------------------------------|----|
| <b>Compiler Used</b>                       | 7  |
| <b>Chapter 1. Fundamentals</b>             | 7  |
| <b>1.1. Basic Concepts</b>                 | 7  |
| 1.1.1. What Is a Processor?                | 7  |
| 1.1.2. A Brief History                     | 9  |
| 1.1.3. x86-64 Processors                   | 10 |
| 1.1.4. x86-64 Processor Registers          | 11 |
| 1.1.5. Memory                              | 11 |
| 1.1.6. Working with External Devices       | 12 |
| 1.1.7. Summary                             | 12 |
| <b>1.2. Assembly Fundamentals</b>          | 13 |
| 1.2.1. Some Notes on Assembly Language     | 13 |
| 1.2.2. Flags Register                      | 13 |
| 1.2.3. MOV Instruction                     | 14 |
| 1.2.4. Data Storage Format in Memory       | 15 |
| 1.2.5. SUB and ADD Instructions            | 15 |
| 1.2.6. Logical Operations                  | 15 |
| 1.2.7. Shifts                              | 16 |
| 1.2.8. Working with Processor Flags        | 16 |
| 1.2.9. Working with the Stack              | 17 |
| 1.2.10. Summary                            | 17 |
| <b>1.3. Labels, Data, and Jumps</b>        | 18 |
| 1.3.1. Data                                | 18 |
| 1.3.2. Labels                              | 18 |
| 1.3.3. Jumps                               | 20 |
| 1.3.4. Anonymous Labels                    | 21 |
| 1.3.5. Working with Bits                   | 22 |
| 1.3.6. Summary                             | 22 |
| <b>1.4. A Closer Look at Assembly</b>      | 23 |
| 1.4.1. Working with Memory and the Stack   | 23 |
| 1.4.2. Working with Numbers in Assembly    | 24 |
| 1.4.3. Multiplication and Division         | 25 |
| 1.4.4. I/O Ports                           | 25 |
| 1.4.5. Loops                               | 26 |
| 1.4.6. Processing Blocks of Data           | 26 |
| 1.4.7. Macros                              | 27 |
| 1.4.8. Structures                          | 28 |
| 1.4.9. Working with MSR Registers          | 29 |
| 1.4.10. The CPUID Instruction              | 29 |
| 1.4.11. The UD2 Instruction                | 30 |
| 1.4.12. Including Files                    | 30 |
| 1.4.13. Summary                            | 30 |
| <b>Chapter 2. Protected Mode</b>           | 31 |
| <b>2.1. Introduction to Protected Mode</b> | 31 |
| 2.1.1. Privilege Levels                    | 31 |
| 2.1.2. Segments in Protected Mode          | 31 |
| 2.1.3. Global Descriptor Table             | 33 |
| 2.1.4. Practice                            | 34 |
| 2.1.5. Summary                             | 38 |
| <b>2.2. Interrupts in Protected Mode</b>   | 39 |
| 2.2.1. What Is an Interrupt?               | 39 |
| 2.2.2. Gate Descriptors                    | 39 |
| 2.2.3. Exceptions                          | 41 |
| 2.2.4. Error Codes                         | 42 |
| 2.2.5. Software Interrupts                 | 42 |
| 2.2.6. Hardware Interrupts                 | 42 |
| 2.2.7. Interrupt Handler                   | 43 |
| 2.2.8. Practice                            | 44 |
| 2.2.9. Summary                             | 46 |
| <b>2.3. Address Translation</b>            | 47 |
| 2.3.1. What Is It?                         | 47 |
| 2.3.2. Ordinary Address Translation        | 47 |
| 2.3.2.1. 4 KB Pages                        | 47 |
| 2.3.2.2. 4 MB Pages                        | 49 |
| 2.3.2.3. CR3 Register                      | 49 |
| 2.3.3. Physical Address Extension Mode     | 50 |
| 2.3.3.1. 4 KB Pages                        | 50 |
| 2.3.3.2. 2 MB Pages                        | 51 |
| 2.3.3.3. CR3 Register                      | 52 |
| 2.3.3.4. NX Bit                            | 52 |

|                                                                           |           |
|---------------------------------------------------------------------------|-----------|
| 2.3.4. Page-Fault Handler .....                                           | 52        |
| 2.3.5. WP Flag in CR0 .....                                               | 53        |
| 2.3.6. Practice .....                                                     | 53        |
| 2.3.6.1. Ordinary Mode .....                                              | 53        |
| 2.3.6.2. PAE Mode .....                                                   | 55        |
| 2.3.6.3. Using Pages of Different Sizes .....                             | 56        |
| 2.3.7. Summary .....                                                      | 56        |
| <b>2.4. Multitasking .....</b>                                            | <b>57</b> |
| 2.4.1. General Information .....                                          | 57        |
| 2.4.2. Task State Segment (TSS) .....                                     | 57        |
| 2.4.3. TSS Descriptor .....                                               | 59        |
| 2.4.4. Local Descriptor Table .....                                       | 59        |
| 2.4.5. Task Register (TR) .....                                           | 60        |
| 2.4.6. Task Management .....                                              | 60        |
| 2.4.7. Task Gate .....                                                    | 61        |
| 2.4.8. I/O Privilege Level .....                                          | 61        |
| 2.4.9. I/O Permission Bitmap .....                                        | 61        |
| 2.4.10. Enabling Multitasking .....                                       | 62        |
| 2.4.11. Practical Implementation .....                                    | 62        |
| 2.4.12. Summary .....                                                     | 65        |
| <b>2.5. Protection Mechanisms .....</b>                                   | <b>66</b> |
| 2.5.1. Fields and Flags Used for Segment- and Page-Level Protection ..... | 66        |
| 2.5.2. Segment-Limit Checks .....                                         | 66        |
| 2.5.3. Type Checks .....                                                  | 66        |
| 2.5.4. Privilege Levels .....                                             | 67        |
| 2.5.5. Privilege Checks When Accessing Data Segments .....                | 68        |
| 2.5.6. Privilege Checks During Intersegment Control Transfer .....        | 68        |
| 2.5.6.1. Transfer to a Nonconforming Code Segment .....                   | 68        |
| 2.5.6.2. Transfer to a Conforming Code Segment .....                      | 68        |
| 2.5.7. Call Gates .....                                                   | 68        |
| 2.5.8. Stack Switching .....                                              | 69        |
| 2.5.9. Using SYSENTER and SYSEXIT .....                                   | 70        |
| 2.5.10. Practice .....                                                    | 71        |
| 2.5.11. Summary .....                                                     | 72        |
| <b>Chapter 3. Programming in Win32 .....</b>                              | <b>73</b> |
| <b>3.1. Introduction to Win32 .....</b>                                   | <b>73</b> |
| 3.1.1. General Information .....                                          | 73        |
| 3.1.2. Memory in Win32 .....                                              | 73        |
| 3.1.3. Windows Executable Components .....                                | 73        |
| 3.1.4. System Libraries and Subsystems .....                              | 74        |
| 3.1.5. Function Calling Model in Win32 .....                              | 74        |
| 3.1.6. Program Execution in Win32: General Picture .....                  | 74        |
| 3.1.7. Practice .....                                                     | 75        |
| 3.1.7.1. First Win32 Program .....                                        | 75        |
| 3.1.7.2. STDCALL Function .....                                           | 75        |
| 3.1.8. Summary .....                                                      | 78        |
| <b>3.2. Programming in Ring 3 .....</b>                                   | <b>79</b> |
| 3.2.1. General Overview .....                                             | 79        |
| 3.2.2. Working with Objects .....                                         | 79        |
| 3.2.3. Working with Files .....                                           | 79        |
| 3.2.4. Handling API Function Errors .....                                 | 80        |
| 3.2.5. Console Programs .....                                             | 80        |
| 3.2.6. GUI Programs .....                                                 | 81        |
| 3.2.7. Dynamic-Link Libraries .....                                       | 82        |
| 3.2.8. Exception Handling in a Program .....                              | 83        |
| 3.2.9. Practice .....                                                     | 84        |
| 3.2.9.1. Creating a Console Application .....                             | 84        |
| 3.2.9.2. Creating a Windowed Application .....                            | 85        |
| 3.2.9.3. Creating a DLL .....                                             | 86        |
| 3.2.9.4. Using SEH .....                                                  | 86        |
| 3.2.9.5. Debugging Applications .....                                     | 87        |
| 3.2.10. Summary .....                                                     | 87        |
| <b>3.3. Programming in Ring 0 .....</b>                                   | <b>88</b> |
| 3.3.1. Services .....                                                     | 88        |
| 3.3.2. General Overview .....                                             | 88        |
| 3.3.3. Driver Development Kit (DDK) .....                                 | 88        |
| 3.3.4. Thread Context and Interrupt Request Levels .....                  | 89        |
| 3.3.5. Example of a Simple Driver .....                                   | 89        |
| 3.3.6. Strings in the Windows Kernel .....                                | 90        |
| 3.3.7. I/O Subsystem .....                                                | 90        |
| 3.3.8. Practice .....                                                     | 93        |
| 3.3.8.1. Driver Control Program .....                                     | 93        |

|                                                      |            |
|------------------------------------------------------|------------|
| 3.3.8.2. GDTDump Driver .....                        | 94         |
| 3.3.8.3. Debugging Drivers .....                     | 97         |
| 3.3.9. Summary .....                                 | 98         |
| <b>Chapter 4. Long Mode .....</b>                    | <b>99</b>  |
| <b>4.1. Introduction to Long Mode .....</b>          | <b>99</b>  |
| 4.1.1. General Overview .....                        | 99         |
| 4.1.2. Segmentation in Long Mode .....               | 100        |
| 4.1.3. Page-Translation Mechanism .....              | 100        |
| 4.1.4. Entering Long Mode .....                      | 100        |
| 4.1.5. Practice .....                                | 101        |
| 4.1.6. Summary .....                                 | 102        |
| <b>4.2. Working with Memory in Long Mode .....</b>   | <b>103</b> |
| 4.2.1. General Overview .....                        | 103        |
| 4.2.2. 4 KB Pages .....                              | 103        |
| 4.2.3. 2 MB Pages .....                              | 104        |
| 4.2.4. 1 GB Pages .....                              | 105        |
| 4.2.5. CR3 Register .....                            | 106        |
| 4.2.6. Protection Checks .....                       | 106        |
| 4.2.7. Practice .....                                | 107        |
| 4.2.8. Summary .....                                 | 110        |
| <b>4.3. Interrupts in Long Mode .....</b>            | <b>111</b> |
| 4.3.1. Gate Descriptors .....                        | 111        |
| 4.3.2. IDT, 64-bit TSS, and the IST Mechanism .....  | 111        |
| 4.3.3. Invoking an Interrupt Handler .....           | 113        |
| 4.3.4. Practice .....                                | 113        |
| 4.3.5. Summary .....                                 | 116        |
| <b>4.4. Protection and Multitasking .....</b>        | <b>117</b> |
| 4.4.1. Segments .....                                | 117        |
| 4.4.2. Call Gates .....                              | 117        |
| 4.4.3. SYSCALL and SYSRET Instructions .....         | 117        |
| 4.4.4. Multitasking .....                            | 118        |
| 4.4.5. Practice .....                                | 119        |
| 4.4.6. Summary .....                                 | 120        |
| <b>Chapter 5. Programming in Win64 .....</b>         | <b>121</b> |
| <b>5.1. Introduction to Win64 .....</b>              | <b>121</b> |
| 5.1.1. Advantages and Disadvantages .....            | 121        |
| 5.1.2. Memory in Win64 .....                         | 121        |
| 5.1.3. Calling Convention .....                      | 121        |
| 5.1.4. Compatibility Mode .....                      | 122        |
| 5.1.5. Win64 API and System Libraries .....          | 122        |
| 5.1.6. Practice .....                                | 122        |
| 5.1.7. Summary .....                                 | 123        |
| <b>5.2. Programming in Win64 .....</b>               | <b>124</b> |
| 5.2.1. Changes in Data Types .....                   | 124        |
| 5.2.2. Stack Alignment .....                         | 124        |
| 5.2.3. GUI Applications .....                        | 124        |
| 5.2.4. Driver Programming .....                      | 126        |
| 5.2.5. Debugging Applications in Win64 .....         | 128        |
| 5.2.6. Summary .....                                 | 128        |
| <b>Chapter 6. Multiprocessor Systems .....</b>       | <b>129</b> |
| <b>6.1. Working with APIC .....</b>                  | <b>129</b> |
| 6.1.1. General Overview .....                        | 129        |
| 6.1.2. Enabling APIC .....                           | 129        |
| 6.1.3. Local APIC ID .....                           | 130        |
| 6.1.4. Local Vector Table .....                      | 130        |
| 6.1.5. Local APIC Timer .....                        | 131        |
| 6.1.6. Interrupt Handling .....                      | 131        |
| 6.1.7. Working with the I/O APIC .....               | 132        |
| 6.1.8. Practice .....                                | 134        |
| 6.1.9. Summary .....                                 | 135        |
| <b>6.2. Interprocessor Communication .....</b>       | <b>136</b> |
| 6.2.1. General Overview .....                        | 136        |
| 6.2.2. Interprocessor Interrupts .....               | 136        |
| 6.2.3. Synchronizing Access to Data .....            | 137        |
| 6.2.4. Initializing a Multiprocessor System .....    | 138        |
| 6.2.5. Practice .....                                | 138        |
| 6.2.6. Summary .....                                 | 140        |
| <b>APPENDICES .....</b>                              | <b>141</b> |
| <b>Appendix A. MSR Registers .....</b>               | <b>141</b> |
| <b>A.1. IA32.EFER Register .....</b>                 | <b>141</b> |
| <b>A.2. Registers Used by SYSENTER/SYSEXIT .....</b> | <b>141</b> |
| <b>A.3. Registers Used by SYSCALL/SYSRET .....</b>   | <b>141</b> |

|                                                         |     |
|---------------------------------------------------------|-----|
| A.4. APIC Registers .....                               | 142 |
| A.5. Registers for Managing Segments in Long Mode ..... | 142 |
| A.6. Auxiliary Registers .....                          | 142 |
| Appendix B. System Registers .....                      | 143 |
| B.1. CR0 Register .....                                 | 143 |
| B.2. CR2 and CR3 Registers .....                        | 143 |
| B.3. CR4 Register .....                                 | 144 |
| B.4. GDTR and IDTR Registers .....                      | 145 |
| B.5. LDTR and TR Registers .....                        | 145 |
| B.6. Flags Register .....                               | 145 |
| B.7. CR8 Register .....                                 | 146 |
| Appendix C. System Instructions .....                   | 147 |
| C.1. Working with System Registers .....                | 147 |
| C.2. System Instructions .....                          | 148 |
| C.3. Working with the Processor Cache .....             | 148 |
| C.4. Additional Instructions .....                      | 148 |
| Alphabetical Index .....                                | 150 |

## Compiler Used

FASM must be used to assemble all examples in this book. Why was this assembler chosen? There are many reasons.

- Maximum instruction-set coverage. FASM supports all, or almost all, x86-64 instructions.
- Linux and Windows versions are available, together with support for a wide range of output formats. Programs can be assembled for Windows in PE format and for Linux in ELF format.
- Flexible syntax and no completely unnecessary but mandatory directives such as `.386`, `.486`, and similar directives.
- Complete control over the placement of data in the executable file.
- A very powerful macro engine with which almost any macro can be created and the program text - even the language itself - can be changed beyond recognition.
- When assembling from the command line, there is no need to specify a large number of command-line options. To obtain an executable of the required type, the necessary parameters can be placed directly in the source file.
- The Windows version of FASM is supplied with an IDE, allowing a program to be assembled by pressing a single button or choosing a menu item.

FASM can be found on the compact disc supplied with the book, in the `fasm` folder. The latest version can also be downloaded free of charge from `flatassembler.net`.

## Chapter 1. Fundamentals

### 1.1. Basic Concepts

Because this book is devoted to programming on the x86-64 platform and all examples are written in assembly language, it is appropriate to begin with the fundamentals of assembly programming. Even a reader without prior preparation should be able to understand the later material after reading this first chapter.

The following pages describe the basic operating principles of the processor and the concepts required for assembly programming and for low-level programming in general. Here, the word *processor* refers to Intel processors and compatible implementations, above all AMD processors.

#### 1.1.1. What Is a Processor?

A processor is the most important part of a computer system; all, or almost all, computational activity takes place on it.

Every computer system consists of three parts: the central processor, system memory, and external devices. All three components interact through buses. The most important is the system bus, through which the processor communicates with system memory and the principal controllers. A controller may act as an intermediary between the processor and external devices - for example an I/O controller, interrupt controller, or USB controller - or it may perform a specialized operation, such as direct memory access.

The processor's main task is to execute instructions stored in memory. From power-on until shutdown, the processor executes instructions except when it enters a suspended state to save energy. Instructions are executed differently depending on the processor's current operating mode.

An instruction, or machine instruction, is a code - essentially an ordinary number stored in memory - that denotes an action. Instructions differ in type and size and can occupy from one byte to several bytes. They tell the processor what to do. To know where to fetch instructions, the processor uses a special instruction pointer. After each instruction is executed, that pointer is updated to identify the next instruction in memory. If an instruction cannot be executed, the processor generates an exception; if the exception cannot be handled, the processor resets.

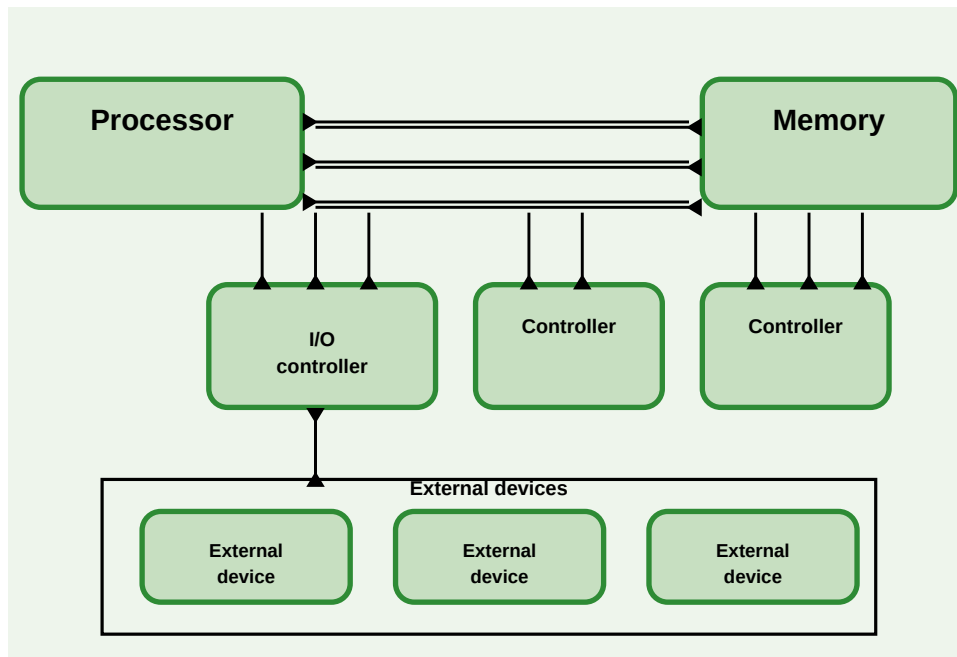


Figure 1.1. Simplified diagram of a computer system.

The lines connecting the blocks are buses: the address bus, data bus, and control bus. Together they form the system bus. Controllers are attached to the system bus to extend the capabilities of the system. Each controller executes commands sent to it through the system bus.

The most important component of a system is memory - system memory. Memory is of two kinds: physical and linear.

The processor also has two kinds of memory of its own: cache memory and register memory. Cache memory is, in effect, the same kind of storage as system memory, but located inside the processor die. It contains a copy of an area of main memory that the processor accesses frequently.

Register memory is represented by registers. A register is a device that stores information - that is, a value. The width of the value determines the width of the register. Some registers can store only particular information, while others can store arbitrary information. Registers that can store arbitrary information are called general-purpose registers. Other registers influence how the processor operates. Registers have different widths; most of the registers discussed in this book are 8-, 16-, 32-, or 64-bit.

To read a value at a particular memory address, the processor places the physical address on the address bus, after which the value appears on the data bus. It does not appear instantaneously; the delay depends on the speed of the system bus, memory, and other characteristics. If an address above the amount of physically installed memory is placed on the address bus, the value returned depends on the motherboard and memory and may simply be meaningless or zero. Writing to a memory location is performed in a similar way. The control bus carries a value indicating whether the processor intends to read from memory or write to it.

Accesses to particular memory addresses - addresses that may not correspond to installed RAM - can be intercepted by controllers, which redirect the data being written or read to a device. In this way, software can interact with external devices by reading and writing particular memory addresses. PCI devices are an example.

The address placed on the address bus is the physical address. Virtual memory is more complicated. A virtual address need not correspond directly to an existing physical location. When an operating-system application accesses such an address, the processor translates it to a physical address; if physical memory is insufficient, the operating system can move other occupied memory to external storage such as a hard disk or flash storage. Virtual memory is discussed in more detail in the chapters on protected mode.

All data on which the processor operates must be in memory. If data are located on an external device, they must first be loaded into memory. A program is also data: it consists of instruction codes. An instruction code is called an opcode. An opcode consists of one or more bytes encoded so that the processor can understand the requested action - for example, moving data between registers or memory, or performing subtraction or addition.

The processor uses EIP - IP or RIP in other modes - as its instruction pointer. This register contains the address of the next instruction. The processor fetches the instruction at that address, determines its size, updates the instruction pointer by the instruction length, and then executes the instruction. An instruction can take from one to several dozen processor cycles. During execution, the instruction may change EIP; in that case, the next instruction executed is the one at the new address, producing a jump. The cycle then repeats continuously until the processor is switched off or enters a special waiting state.

Another important concept is the stack. The stack is a special area of memory used to store intermediate data. When one subroutine calls another, the called subroutine must eventually return control to the caller. The address of the instruction to which control must return is kept at the top of the stack.

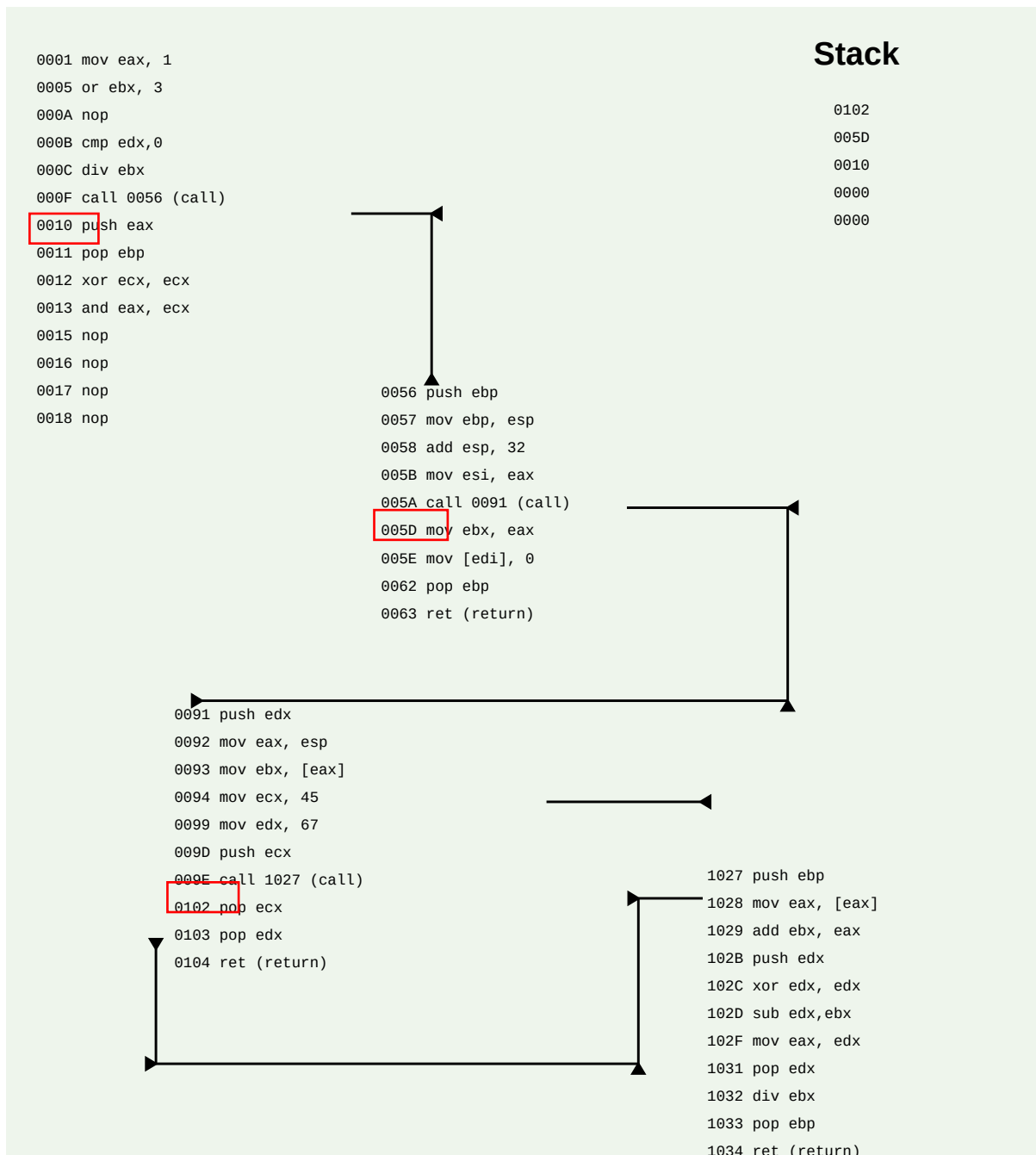


Figure 1.2. Example chain of subroutine calls and stack contents.

The stack behaves like a magazine: addresses are added one after another and then removed in reverse order. A subroutine can therefore call another subroutine, which can call another, and so on; the practical limit is the size of the stack. A dedicated register points to the top of the stack.

In Figure 1.2, the return addresses are outlined. A return address is the address of the instruction following the CALL instruction. When RET executes, the return address is removed from the top of the stack and control is transferred to it.

### 1.1.2. A Brief History

Historically, the processors considered here belong to the x86 family: 8086, 80186, 80286, 80386, 80486, 80586 (Pentium I), 80686 (Pentium II), and later processors. The 8086 through 80286 are 16-bit processors; the later pre-x86-64 processors are 32-bit.

Intel processors supporting EM64T and AMD processors supporting AMD64 are 64-bit processors. AMD64 and EM64T are, in practice, the same x86-64 extension. AMD developed the technology; Intel later implemented a compatible form.

Intel also developed a different 64-bit architecture called IA-64 and used it in Itanium processors. IA-64 was designed primarily for high-performance server systems and is not considered in this book.

Platforms based on AMD64 or EM64T are referred to here as x86-64 platforms. The name x64 is also common, but is used more narrowly here for the processor's 64-bit mode. x86-64 is compatible with x86 and is its logical continuation. Platforms based on

Itanium are called IA-64. x86-64 and IA-64 are completely different technologies.

### 1.1.3. x86-64 Processors

This book is concerned mainly with x86-64 processors. They operate primarily in three modes: real-address mode, protected mode, and 64-bit mode, or long mode.

- **Real-address mode.** This is the mode entered after power-on or reset. It is the standard 16-bit mode, with access to only 1 MB of physical memory and without page-based translation of virtual addresses to physical addresses. Addresses used by programs are therefore physical addresses and are placed on the address bus without page translation. The native data size is 2 bytes, or one WORD.
- **Protected mode.** This is the 32-bit mode. A 32-bit operating system can use the processor's protected-mode facilities and can address a 4 GB physical address space. With an additional address-translation mechanism, up to 64 GB of physical memory can be addressed. Protected mode can be entered only from real-address mode and provides mechanisms for protecting operating-system data from applications. The native data size is 4 bytes, or one DWORD; address operands are 32-bit.
- **Long mode.** This is the processor's 64-bit mode. Its operating principles are very similar to protected mode except for several architectural differences. It provides access to  $2^{52}$  bytes of physical memory and  $2^{48}$  bytes of virtual memory. Long mode can be entered only from protected mode. The native data size is described as a DWORD, although 8-byte data can be used; addresses are 8 bytes wide.

*Editorial note: the source prints the limits as powers  $2^{52}$  and  $2^{48}$ ; they are preserved here rather than converted to approximate capacities.*

The processor also supports System Management Mode (SMM), entered when an SMI is received. SMM is intended for operations that must be isolated from application software and even from the operating system. Entry is hardware-controlled. It can be used for power-management functions or for security and access-control functions.

Two additional submodes are supported:

- **Virtual-8086 mode.** A protected-mode submode for supporting old 16-bit applications. It can be enabled for an individual task in a multitasking protected-mode operating system.
- **Compatibility mode.** A long-mode submode that supports 32-bit and 16-bit code. Applications can access 4 GB of memory; the native data size is a doubleword and the address size is 32 bits. Eight-byte operands are not available in compatibility mode.

Processor manufacturers use somewhat different terminology. AMD documentation calls the 32-bit environment *legacy mode*, while Intel documentation uses *IA-32*; it includes real-address mode and protected mode. AMD calls the 64-bit environment *long mode*, while Intel calls it *IA-32e*; it contains 64-bit mode and compatibility mode.

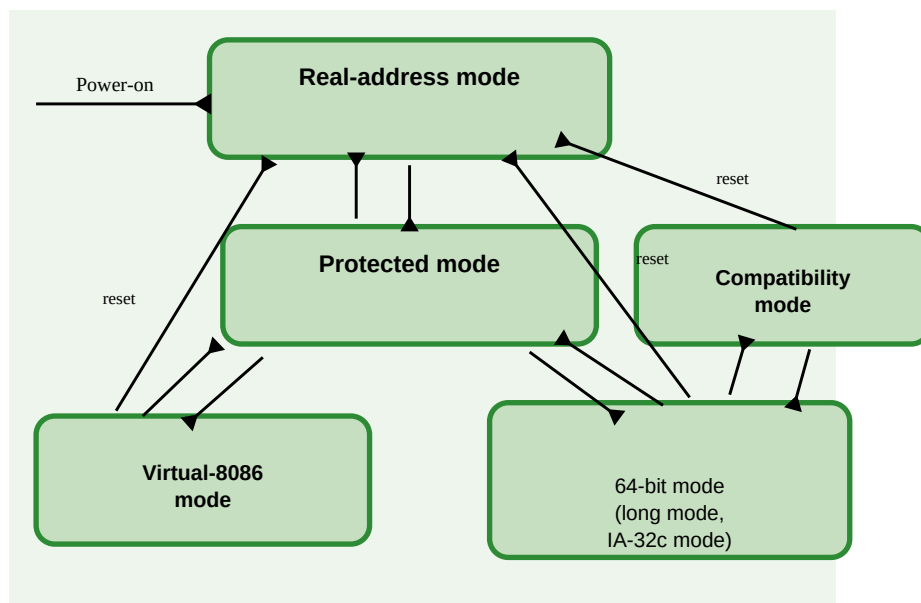


Figure 1.3. x86-64 processor-mode diagram.

*Editorial note: Figure 1.3 prints "IA-32c mode" inside the 64-bit-mode box, while the surrounding prose identifies Intel's 64-bit environment as IA-32e. The figure label is preserved as printed.*

The diagram shows that 64-bit mode can be entered only after protected mode has been enabled. After a processor reset, regardless of its previous mode, execution begins in real-address mode. A direct return to real-address mode is possible from protected mode; from the other modes, a processor reset is required.

Virtual-8086 mode and real-address mode are not developed further in this book. The remaining chapters concentrate on protected mode and the processor's 64-bit mode.

### 1.1.4. x86-64 Processor Registers

In protected mode, real-address mode, and compatibility mode, the following registers are available:

- general-purpose registers: 32-bit EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP; 16-bit AX, BX, CX, DX, SI, DI, SP, BP, which are the low parts of the 32-bit registers; and 8-bit AH, BH, CH, DH and AL, BL, CL, DL, the high and low parts of the corresponding 16-bit registers;
- 32-bit EIP - IP in real-address mode - as the instruction pointer;
- 16-bit segment registers CS, DS, SS, ES, FS, and GS;
- 32-bit flags register EFLAGS;
- 80-bit mathematical-coprocessor registers ST0-ST7 and related state;
- 64-bit MMX registers MM0-MM7;
- 128-bit XMM registers XMM0-XMM7 and the 32-bit MXCSR register;
- 32-bit control registers CR0-CR4; system-table pointer registers GDTR, LDTR, and IDTR; and the task register TR;
- 32-bit debug registers DR0-DR3, DR6, and DR7;
- MSR registers.

Not all of the registers listed above are usable in real-address mode, although the control registers remain available. Some registers wider than 16 bits cannot be used by ordinary real-mode code.

After switching to 64-bit mode, the following registers are available:

- general-purpose registers: 64-bit RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP, R8, R9, R15; 32-bit EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, R8D-R15D; 16-bit AX, BX, CX, DX, SI, DI, SP, BP, R8W-R15W; and 8-bit AH, BH, CH, DH and AL, BL, CL, DL, SIL, DIL, SPL, BPL, R8L-R15L;
- 64-bit RIP as the instruction pointer;
- 16-bit segment registers CS, DS, SS, ES, FS, GS;
- 64-bit flags register RFLAGS;
- 80-bit mathematical-coprocessor registers ST0-ST7;
- 64-bit MMX registers MM0-MM7;
- 128-bit XMM registers XMM0-XMM15 and 32-bit MXCSR;
- 64-bit control registers CR0-CR4 and CR8; system-table pointer registers GDTR, LDTR, IDTR; and task register TR;
- 64-bit debug registers DR0-DR3, DR6, DR7;
- MSR registers.

*Editorial note: the 64-bit GPR list jumps from R9 to R15 instead of explicitly listing R10-R14, and the 8-bit registers are printed as R8L-R15L. These source forms are preserved. Modern architectural naming uses R8B-R15B for those byte registers.*

Segment registers participate directly in address formation. CS identifies the code segment, DS the data segment, and SS the stack segment; ES, FS, and GS are additional segment registers and may be unused by a program. They cannot always be loaded freely. In protected and 64-bit modes, only particular values are valid, and register accessibility depends on the privilege level at which the program executes.

ESP, or RSP, normally points to the top of the stack. It can be used for another purpose, but doing so prevents normal stack operation. The general-purpose registers can otherwise be used freely, although some instructions conventionally use particular registers. EBP, or RBP, for example, is commonly used as the base of a stack frame containing a subroutine's local data.

EIP, or RIP, cannot be used directly like an ordinary general-purpose register; it is maintained by the processor as the instruction pointer.

STn, MMn, and XMMn are used for floating-point, multimedia, and SIMD operations. CRn, DRn, the system-table registers, and MSRs are system registers used by fundamental processor mechanisms. Their details are introduced later in the book.

### 1.1.5. Memory

In the 1970s, a computer with 64 KB of memory was considered exceptionally well equipped. Sixteen address bits were sufficient for such a memory. When 128 KB and 256 KB memory configurations appeared, 16-bit registers were no longer sufficient for direct addressing, so 16-bit segment registers were introduced. Memory was divided into 64 KB segments. A segment was selected by a segment register and an address within the segment was supplied by an ordinary register or by an immediate 16-bit value. Even with this arrangement, the classic real-mode address-formation scheme could address only 1 MB of physical memory.

The situation changed with the Intel 80386, the first full 32-bit processor in the line. It could operate in the old 16-bit mode and in protected mode, in which up to 4 GB of physical memory could be addressed. New 32-bit registers and 32-bit data instructions were added. Segment registers remained, but in protected mode their main purpose shifted toward describing and protecting memory

regions defined by the operating system. Access to a 4 GB address space was also technically possible by other means in real-address mode, but protected mode provided far more than a larger physical address space.

With AMD64, a 64-bit mode was added as a protected-mode-derived environment for 64-bit architecture. The importance of segmentation was reduced to a minimum, and protection of operating-system memory was placed primarily on the page-translation mechanism.

This brief history is useful because it explains the three kinds of address used in the book:

- physical address;
- linear, or virtual, address;
- logical address.

A physical address is an address in the computer's system memory - the address placed on the address bus. The logical address is more involved.

To access a value in memory, software identifies a segment and an offset within that segment. The segment is selected through a segment register or a direct segment value; the offset can come from a normal register or an immediate value and may be 16, 32, or 64 bits depending on the processor mode. A program can express the offset in different ways, and the segment may be omitted from the written instruction. When it is omitted, the appropriate code, data, or stack segment register supplies it. Conceptually, the address therefore always has the form `segment : offset`; this is the logical address.

Transforming the logical address according to the current processor mode produces an absolute 20-, 32-, or 64-bit address. This is the linear address. In real-address mode it is placed directly on the address bus. In protected and 64-bit modes, when page translation is active, the linear address can also be called a virtual address. If page translation is disabled, the linear address becomes the physical address and is placed directly on the address bus. If page translation is enabled, the linear - virtual - address is transformed into a physical address according to paging structures defined by the operating system.

### 1.1.6. Working with External Devices

One of the processor's main tasks is interaction with external devices. Figure 1.4 shows a processor system from this point of view.

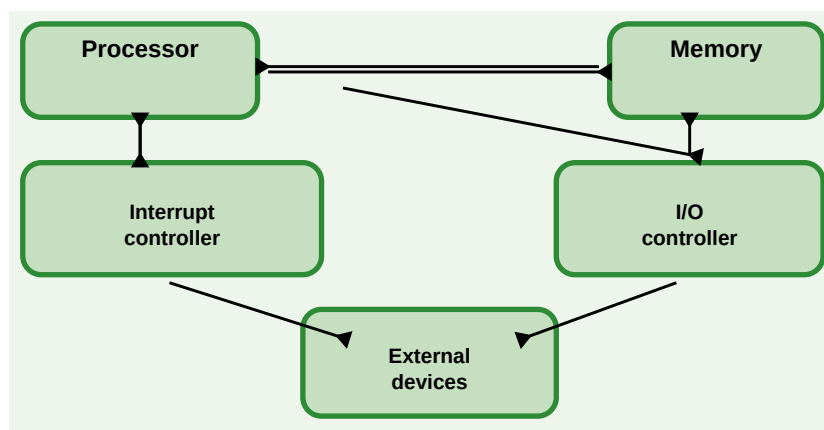


Figure 1.4. Processor interaction with external devices.

All arrows in the diagram represent buses. Some belong to the system bus; others connect devices and controllers.

The processor interacts with external devices through I/O ports - not to be confused with physical connectors such as LPT or PS/2 - and through device registers mapped into memory. The processor writes data to ports or to memory locations onto which device registers are mapped. In the general case shown in Figure 1.4, an I/O controller is the intermediary between the processor and the devices, although another type of controller can perform the same role.

How does the processor learn that a device has completed a request or received new data? When an external device needs to notify the processor, the device can send an interrupt signal through the interrupt controller.

An interrupt is a signal telling the processor to interrupt execution of the current program and transfer control to a special function called an interrupt handler. The program can then obtain the necessary data through the I/O controller and process the request.

### 1.1.7. Summary

This section introduced the processor, its principal operating modes, and the basic principles of its operation. Section 1.2 turns to the fundamentals of assembly language.

## 1.2. Assembly Fundamentals

When you write a program in assembly language, you are in effect writing commands for the processor. Processor commands are simply codes - operation codes, or opcodes. This is why assembly language is considered the lowest-level programming language: everything in assembly language is translated directly into hexadecimal codes. In other words, there is no compiler that first translates a high-level language into a lower-level language; the assembler program translates human-readable instructions directly into machine codes.

This section introduces several basic instructions related to calculations, bit operations, and similar tasks. Other instructions, including jumps and comparisons, are considered later.

### 1.2.1. Some Notes on Assembly Language

FASM is used throughout this book. All examples in the book are written for FASM, and the description of instruction syntax therefore takes FASM syntax into account.

In an assembly-language program, each instruction must appear on one line: one line, one instruction. Comments begin after a semicolon; every character after the semicolon on that line is treated as a comment.

Numbers may be written in binary, decimal, or hexadecimal notation. A suffix identifies the base. Binary numbers use `b`, for example `0000010b` or `001011010b`. Decimal numbers may be written without a suffix or with `d`, for example `4589` or `2356d`. Hexadecimal numbers use `h`; when the first hexadecimal digit would be A-F, the number is written with a leading zero, for example `0AC45h`. The source gives `F145Ch` and `C123h` as incorrect forms.

Hexadecimal values may also be written in forms familiar from high-level languages: with an initial `$`, as in Pascal, or with `0x`, as in C. Character values may be enclosed in either quotation marks or apostrophes.

Identifiers in a program may not contain the characters `+ - * / = < > ( ) [ ] { } : , | & ~ # ' .` Hexadecimal notation is used most often in this book. Bit numbering always begins with zero; therefore, what the book calls bit 5 is the sixth bit in ordinary counting.

### 1.2.2. Flags Register

If an instruction produces an error condition, subsequent operations may also produce incorrect results and undesirable consequences. To make the result of an operation available to the program, the processor records status in a special register: the flags register, EFLAGS (FLAGS, RFLAGS). The flags register is central to processor operation because it controls fundamental aspects of execution.

After an instruction executes, relevant status information is reflected in this register. Each bit represents a particular state or result. Some bits control processor behavior; others describe only the result of the preceding operation. For example, if an arithmetic or logical operation produces zero, the Zero Flag (ZF) is set to one; otherwise it is cleared.

The meanings of the flags-register bits are shown in Table 1.1.

Table 1.1. Flags register

| Bit       | Name | Description                          |
|-----------|------|--------------------------------------|
| 0         | CF   | Carry Flag                           |
| 1         | -    | Reserved (always set)                |
| 2         | PF   | Parity Flag                          |
| 3         | -    | Reserved (always clear)              |
| 4         | AF   | Auxiliary Carry Flag                 |
| 5         | -    | Reserved (always clear)              |
| 6         | ZF   | Zero Flag                            |
| 7         | SF   | Sign Flag                            |
| 8         | TF   | Trap Flag                            |
| 9         | IF   | Interrupt Enable Flag                |
| 10        | DF   | Direction Flag                       |
| 11        | OF   | Overflow Flag                        |
| 12        | IOPL | Current I/O privilege level          |
| 13        |      |                                      |
| 14        | NT   | Nested Task Flag                     |
| 15        | -    | Reserved (always clear)              |
| 16        | RF   | Resume Flag                          |
| 17        | VM   | Virtual-8086 Mode Flag               |
| 18        | AC   | Alignment Check Flag                 |
| 19        | VIF  | Virtual Interrupt Flag               |
| 20        | VIP  | Virtual Interrupt Pending Flag       |
| 21        | ID   | Processor identification-enable flag |
| 22-31(63) | -    | Reserved (always clear)              |

Some flag names are self-explanatory while others are not. Their detailed purposes are not discussed here; they are introduced as needed later.

### 1.2.3. MOV Instruction

The first instruction is MOV, the data-transfer instruction. It moves a byte, word, doubleword, or quadword from a source to a destination. MOV is the model for other two-operand instructions: instructions with two operands use the same general operand-order syntax. MOV is extremely common; the book states that in many programs as much as 40 percent of the instructions are MOV instructions.

MOV syntax: `mov <destination>, <source>`.

The instruction assigns a value to a register or memory location. The value may come from memory, a register, or an immediate constant. If the destination is a register, the source may be memory, another register, or an immediate value. If the destination is memory, the source may be only a register or an immediate value; a memory-to-memory MOV is not allowed. Source and destination must have the same size.

#### Listing 1.1. Working with registers using MOV

```
mov edx, ebx
mov rax, r10
mov eax, r12d
mov ah, dh
mov cx, r14w
mov ecx, 789056h
mov ax, 12h
mov al, 890h      ; incorrect
mov dx, rbx       ; incorrect
```

To work with memory, specify an address and enclose it in square brackets. The address must be compatible with the current processor mode; the book gives a 16-bit address for real-address mode and a 64-bit address for 64-bit mode as examples.

#### Listing 1.2. Working with memory using MOV

```
mov eax, dword [0000509Ah]
mov rbx, qword [00A056F1h]
mov dword [0F81562Dh], r9d
mov qword [0F8156F6h], 0DF899564FFh
```

A size operator before the address specifies the amount of data to transfer. The size operators used by the book are shown in Table 1.2.

Table 1.2. Size operators

| Size operator | Size in bytes |
|---------------|---------------|
| byte          | 1             |
| word          | 2             |
| dword         | 4             |
| fword         | 6             |
| pword         | 6             |
| qword         | 8             |
| tbyte         | 10            |
| tword         | 10            |
| dqword        | 16            |

The most commonly used size operators are byte, word, dword, and qword. The book states that an 8-byte operand cannot be used with MOV in protected mode and that a 4-byte operand cannot be used in real-address mode, although 4-byte registers may be used there. Other instruction classes can use operands of other sizes regardless of the current processor mode; for example, x87 instructions can use 10-byte tbyte/tword operands in protected and 64-bit modes.

*Editorial note: operand and address sizes on x86 can be affected by instruction encoding and size-override prefixes. The broad mode-based statement above is retained from the source.*

The book also states that the PTR directive may be written before an address instead of using square brackets.

#### Listing 1.3. Using the PTR directive

```
mov eax, dword ptr 0000509Ah
mov rbx, qword ptr 00A056F1h
mov qword ptr 0F81562Dh, r9d
mov qword ptr 0F8156F6h, 09564FFh
```

Memory-to-memory transfer is not allowed. Sometimes the size need not be written explicitly: when moving from memory into a register, the register size already determines the operand size. A register may also be used to address memory.

#### Listing 1.4. Using a register to address memory

```
mov eax, [eax]
mov rbx, ptr ebx
mov [rax], r9d
mov ptr edx, 09564FFh
```

*Editorial note: Listings 1.3 and 1.4 reproduce the PTR forms exactly as printed. Several of these forms do not match normal FASM memory-operand syntax; they are preserved rather than silently rewritten.*

General-purpose registers can be used freely. The other processor registers are normally accessed only through general-purpose registers; hence the name general-purpose. The book collectively calls the remaining non-x87 registers privileged registers because protected-mode access to them is available only from privileged code. It says MOV is, in practice, the instruction used to transfer values to and from such registers.

There is also a set of machine-specific registers, the MSRs. They are not accessed with MOV; RDMSR reads them and WRMSR writes them.

Register-to-register, register-to-memory, and other MOV forms are actually separate machine instructions with different opcodes. Because they perform the same conceptual operation, assembly language groups them under one mnemonic. Many assembly mnemonics similarly represent several related opcodes rather than a single opcode.

#### 1.2.4. Data Storage Format in Memory

Consider the following memory region:

##### Memory example

|         |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
|---------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| address | 275 | 276 | 277 | 278 | 279 | 27A | 27B | 27C | 27D | 27E | 27F | 280 | 281 | 282 |
| data    | 0A  | 50  | 132 | 44  | 157 | 25  | 7A  | 5E  | 72  | EF  | 7D  | FF  | AD  | C7  |

Now suppose the following instruction executes:

```
mov eax, dword [0000027Ah]
```

After the instruction, EAX contains 725E7A25h. In memory the bytes appear as 25 7A 5E 72. This is the little-endian storage format: the least-significant byte is stored at the lowest address, and successively more significant bytes occupy increasing addresses.

*Editorial note: the Russian source describes the lowest-address byte as the 'most significant' byte while simultaneously describing little-endian order. The numeric example itself is unambiguous; the English sentence uses the standard little-endian terminology.*

The book next divides integer values into signed and unsigned values. Unsigned values are represented by the ordinary binary representation; for an n-bit unsigned value it gives the range 0 to  $2^n - 1$ .

For signed values, the most-significant bit is the sign and that all remaining bits form the magnitude, while giving the range  $-2^{(n-1)}$  to  $2^{(n-1)} - 1$ . It then says addition and subtraction work according to the same principles for signed and unsigned values, while differences appear during multiplication and division. Signed and unsigned arithmetic is discussed in more detail in Section 1.4.

*Editorial note: ordinary x86 signed integers use two's-complement representation, not sign-magnitude representation. The source's signed-integer description is retained and explicitly flagged.*

Real numbers are still represented in binary as 32-, 64-, or 80-bit values. The book assigns arithmetic on such values to the mathematical coprocessor and does not cover floating-point programming in this edition.

#### 1.2.5. SUB and ADD Instructions

ADD adds its two operands and stores the result in the destination operand. Syntax: `add <destination>, <source>`. Its operand format is the same as MOV. Relevant flags are changed according to the result.

SUB performs subtraction: it subtracts the source operand from the destination operand and stores the result back in the destination. Its syntax is otherwise identical to ADD.

If the result of addition or subtraction does not fit in the destination, overflow occurs and OF is set. It also says that a carry or borrow out of the most-significant bit sets CF, and states that OF and CF usually accompany each other, while noting that subtraction may leave OF clear. The detailed use of OF and CF is deferred to Section 1.4.

*Editorial note: OF and CF report different conditions - signed overflow and unsigned carry/borrow - and they do not generally rise together. The source statement is preserved and flagged.*

INC and DEC increase or decrease one operand by one. Each takes a single register or memory operand. The book gives possible operand sizes of 1, 2, 4, or 8 bytes. The operands discussed above may represent either signed or unsigned values.

#### 1.2.6. Logical Operations

The next group consists of the basic logical operations: OR, AND, NOT, and XOR.

**OR** performs bitwise logical OR. Syntax: `or <destination>, <source>`. The result bit is one when either input bit is one, and the result is stored in the destination.

**AND** performs bitwise logical AND. Syntax: `and <destination>, <source>`. The result bit is one only when both input bits are one.

**NOT** performs bitwise logical negation. Syntax: `not <destination>`. Every one bit becomes zero and every zero bit becomes one.

**XOR** performs bitwise exclusive OR. Syntax: `xor <destination>, <source>`. The result is stored in the destination.

For these instructions, the book gives operand sizes of 1, 2, 4, or 8 bytes.

Table 1.3. XOR instruction logic

| Destination | Source | Result |
|-------------|--------|--------|
| 0           | 0      | 0      |
| 0           | 1      | 1      |
| 1           | 0      | 1      |
| 1           | 1      | 0      |

If the operands are equal, XOR produces zero; if they differ, it produces one. Consequently, XORing a register with itself clears that register, a technique the book notes as useful during optimization.

All four logical operations affect the appropriate flags according to the result.

### 1.2.7. Shifts

A shift moves the bits of an operand to the right or left. For example, shifting 11101001 left by 3 gives 01001000 when only eight result bits are retained. All shift instructions use the same basic syntax: <instruction> <destination>, <bit count>. The book divides shifts into logical, arithmetic, and cyclic forms.

**SHL** - logical shift left. Bits move left and zeroes enter at the right. CF receives the last bit shifted out of the left side. If the count is not one, OF is undefined. If the count is one, OF is zero when the two most-significant bits of the original destination agree; otherwise OF is one.

**SHR** - logical shift right. The source says the newly freed bits on the right are filled with zeroes. CF receives the last bit shifted out of the right side. It first states that OF is zero if the sign bit keeps its value and one otherwise, then states that when the count is not one OF is undefined, and when the count is one OF equals the original most-significant bit.

*Editorial note: for SHR, zeroes enter from the left, not the right. The source also gives overlapping descriptions of OF; both source statements are represented above rather than silently reconciled.*

**SAL** - arithmetic shift left. The book states that SAL is completely identical to SHL; they are the same instruction under different names.

**SAR** - arithmetic shift right. It is described as similar to SHR except that each new bit inserted on the left equals the original most-significant bit. The sign is therefore preserved. The source example shifts 10101001 right by 3 and obtains 11110101.

**ROL** - rotate left. Bits rotate left; a bit leaving the left side is inserted at the right. CF receives the last bit rotated out. For a count other than one, OF is undefined. For a count of one, OF is the XOR of the two most-significant bits of the original operand.

**ROR** - rotate right. The source says the operand rotates right, but then states that the bit leaving the left side is inserted on the left. CF receives the last bit shifted out of the right side. For a count other than one, OF is undefined; for a count of one, OF is the XOR of the two most-significant bits of the result.

*Editorial note: in a right rotate, the bit leaving the right side is reinserted at the left. The contradictory source sentence is retained and flagged.*

**RCL** - rotate left through flag CF, as printed in the source. The bit leaving the left side is moved into CF, and the old CF value enters the right side. For counts other than one, OF is undefined; for a count of one, OF is the XOR of the two most-significant bits of the result.

*Editorial note: CF is printed in the RCL description; the surrounding text and the instruction semantics clearly refer to CF.*

**RCR** - rotate right through CF. The bit leaving the right side is moved into CF, and the old CF value enters the newly opened bit on the left. For counts other than one, OF is undefined; for a count of one, OF is the XOR of the two most-significant bits of the result.

For all of the shift and rotate instructions above, the book gives destination operand sizes of 1, 2, 4, or 8 bytes.

### 1.2.8. Working with Processor Flags

Several commonly used flag bits can be changed directly with dedicated instructions:

1. CLC - clear Carry Flag (CF = 0).
2. CLD - clear Direction Flag (DF = 0).
3. CLI - clear Interrupt Enable Flag (IF = 0).
4. LAHF - copy the first byte of the flags register into AH (SF, ZF, AF, PF, CF).
5. SAHF - copy AH into the first byte of the flags register (SF, ZF, AF, PF, CF); reserved-value bits are ignored.
6. STC - set Carry Flag to one (the source prints CF-1).
7. STD - set Direction Flag to one (the source prints DF-1).
8. STI - enable interrupts (IF = 1).

All of these instructions except CLI and STI are described as unprivileged. CLI and STI cannot always be used freely in application code; their privilege rules are discussed later.

### 1.2.9. Working with the Stack

The top of the stack is addressed by ESP (SP, RSP). Two instructions operate directly on the stack: PUSH and POP. PUSH places an operand on the stack; the source says that if the operand is smaller than the stack element size it is padded with zeroes. Syntax: `push/pop <operand>`. Depending on the processor mode, the source gives operand sizes of 2, 4, or 8 bytes; a segment register may also be used. POP cannot use an immediate value as its destination.

The source prose says PUSH stores the operand at the memory location addressed by ESP (SP, RSP) and then decreases that register by 2, 4, or 8 depending on the mode. It says POP performs the inverse operation. To illustrate PUSH and POP, the book gives equivalent 32-bit sequences:

#### Equivalent of PUSH

```
sub esp, 4
mov [esp], <operand>
```

#### Equivalent of POP

```
mov <operand>, [esp]
add esp, 4
```

For 16-bit and 64-bit operands, the corresponding adjustment is 2 or 8 bytes. The book also reminds the reader that each processor mode has its own restrictions on the size of a value placed on the stack.

*Editorial note: the prose describes PUSH as storing first and decrementing afterward, while the source's own equivalent sequence decrements the stack pointer first and then stores. The code sequence is the one consistent with normal PUSH behavior.*

*Editorial note: the statement that a smaller PUSH operand is padded with zeroes is an oversimplification; immediate PUSH encodings can use sign extension. The source statement is preserved and flagged.*

### 1.2.10. Summary

This section introduced the basic and most important assembly-language instructions: MOV, addition and subtraction, bit shifts, logical operations, flag operations, and stack operations. The next section discusses data, labels, and control transfers.

## 1.3. Labels, Data, and Jumps

As the title suggests, this section discusses the use of variables in assembly-language programs, together with labels and jumps.

Conditional and unconditional jumps are an essential part of any program. They make it possible to write branching and cyclic algorithms. Without them, little more than the simplest "Hello World!" type of program could be written; any more complicated program requires conditional or unconditional control transfer.

### 1.3.1. Data

Data in assembly-language programs is declared, or reserved, with data directives. Table 1.4 lists the directives used to declare data variables.

**Table 1.4. Data-definition directives and their sizes**

| Size (bytes) | Data definition |
|--------------|-----------------|
| 1            | DB              |
| 2            | DW, DU          |
| 4            | DD              |
| 6            | DF, DP          |
| 8            | DQ              |
| 10           | DT              |

A data-definition directive is followed by one or more numeric values separated by commas. These expressions specify the values of the elementary data items, whose size depends on the directive. A character may be used in place of a number; it is then interpreted as the numeric code of the character or characters. Listing 1.5 shows several forms of data declaration.

#### Listing 1.5. Examples of data-definition directives

```
db 67h
db 5dh, 0f6h
db "z"           ; same as db 7ah
db "w", "k", "y"
dw 8a34h, 0c51h, 8bh
du 9e3ah, 07deh
dw "wE"          ; same as db 57h, 45h
dd 01F243D5Eh
dd "wEGa"        ; same as db 57h, 45h, 47h, 61h
dq 1122334455667788h
```

The DB and DU directives may also be followed by strings of arbitrary length. Each character is interpreted as its numeric value. DU differs from DB in that each character is represented by two bytes, with the high byte filled with zero. The following examples:

```
db "assembler"
; same as:
db 61h, 73h, 73h, 65h, 6Dh, 62h, 6Ch, 65h, 72h

du "assembler"
; same as:
db 0, 61h, 0, 73h, 0, 73h, 0, 65h, 0, 6Dh, 0, 62h, 0, 6Ch, 0, 65h, 0, 72h
```

*Editorial note: the DU example is reproduced exactly as printed. Its prose says the high byte is zero, but the printed byte sequence places the zero before each character byte; that ordering is inconsistent with the little-endian UTF-16-style layout normally expected on x86.*

For a long sequence of identical data, FASM provides the DUP directive. It follows the data-definition directive. The repetition count appears before DUP, and the value or sequence to repeat appears in parentheses.

#### Listing 1.6. Using the DUP directive

```
db 7 dup (1Ah)
db 6 dup (45h, 0A3h, 90h)
dd 13 dup (0A713E445h)
dd 9 dup (0A713E445h, 0F8D3E412h)
dw 5 dup (?)
```

If a question mark follows a data-definition directive, the initial value is undefined. Such data is called uninitialized data. Uninitialized data is not stored in the executable file and becomes available only after the program has been loaded into memory, provided the chosen executable format supports it. This can reduce the executable size because in many cases the initial value of a variable is unimportant.

### 1.3.2. Labels

A label is an identifier used in a program that evaluates to the memory address at which it is located. Labels are the primary way of referring to data in memory. When a label is used, the assembler substitutes the address to which the label refers.

There are several ways to define labels. The simplest is to place a colon after the label name. An instruction or directive may follow on the same line. The label's value is the offset - the address of the point at which it is defined, more precisely the offset of the following directive or instruction. This form is usually used to mark locations in code, although it can also be used for data.

Data is more commonly labeled by writing the label name immediately before the data-definition directive. The label then evaluates to the address of the data that follows it. The assembler remembers the size of the labeled data and reports incompatible operand sizes,

so explicit size operators such as WORD, DWORD, and QWORD are not normally required with that form.

The third and most flexible method uses the LABEL directive. LABEL is followed by the label name, optionally an operand size, and optionally the operator AT plus a numeric expression giving the address to which the label refers. Another label may be used as the address expression.

#### Listing 1.7. Examples of label declarations

```
metka1:    dd ...
data1     db ...
          dw ...
          ...
          mov ...
          ...
          add ...
metka2:
          and ...
          xor ...
label metka3 dword at metka2
          rcl ...
```

The memory examples in the previous section were not completely realistic. An expression such as `mov eax, dword [0000509Ah]` is rarely seen in an ordinary program because a programmer cannot reasonably calculate the address of every variable by hand. Most programs use labels when accessing memory.

#### Listing 1.8. Examples of using labels

```
metka1:    dd ...
data1     db ...
          dw ...
          ...
          mov al, [data1]
metka2:
          mov ebx, eax
          mov ebx, dword [metka1]
          mov [data1], 4567AADDh
          mov cx, word [metka2]
```

The last instruction illustrates that assembly language makes no fundamental distinction between code and data. It reads the bytes encoding `mov ebx, eax` as an ordinary numeric value and places them in CX.

A label whose name begins with a period is treated as local. Its name is attached to the most recent global label - a label whose name does not begin with a period - to form its full name. The short local name may be used until the next global label; elsewhere the full name must be used. Labels beginning with two periods are an exception: they have global-label properties but do not establish a new prefix for local labels.

#### Listing 1.9. Local and global labels

```
globmetka1:
    .locmetka1 dd ...
    .locmetka2:
    mov [.locmetka1], eax
    mov word [.locmetka2], cx
    mov word [.locmetka3], cx          ; error!
    mov word [globmetka2.locmetka2], cx
globmetka2:
    .locmetka3: ...
    mov [globmetka3.locmetka4], cx    ; error!
    mov [globmetka3.locmetka4], cx    ; error!
..globmetka3:
    .locmetka4: dw ...
    mov [.locmetka4], cx
```

*Editorial note: the space between the period and locmetka4 in .locmetka4: dw ... is printed in the original listing and is preserved.*

The label \$ denotes the current offset, or the offset of the current instruction. Thus `jmp $` creates an unconditional jump to itself and loops forever.

Labels may be near or far. Nearness is determined by the particular instruction in which the label is used, so the same label may be near in one place and far in another. The source defines a near label as one no more than 127 bytes forward or 128 bytes backward from the instruction that uses it; labels outside that range are called far. The distinction matters only for control-transfer instructions.

It is sometimes useful to declare several views of the same memory at one address. This is useful when the same data is interpreted differently in different contexts and separate variable names make the code clearer. FASM's VIRTUAL directive provides this facility.

#### Listing 1.10. Using the VIRTUAL directive

```
param dd ?
virtual at param
    LowPart dw ?
    HighPart dw ?
end virtual

virtual at param
    Byte1 db ?
    Byte2 db ?
    Byte3 db ?
    Byte4 db ?
end virtual
```

In Listing 1.10, LowPart refers to the low part of param. Byte3 refers to the third byte of param, which is also the first byte of HighPart. VIRTUAL may also describe data at an address held in a general-purpose register:

```
virtual at eax
  Char1 db ?
  Char2 db ?
  Char3 db ?
  Char4 db ?
end virtual
```

With that definition, `mov bl, [Char1]` is equivalent to `mov bl, [eax]`, and `mov bl, [Char3]` is equivalent to `mov bl, [eax+2]`. A clearer form is shown in Listing 1.11.

**Listing 1.11. Using VIRTUAL for data addressed by a register**

```
mov eax, string

virtual at eax
  eax.Char1 db ?
  eax.Char2 db ?
  eax.Char3 db ?
  eax.Char4 db ?
end virtual

mov bl, [eax.Char1]
mov bl, [eax.Char2]
mov bl, [eax.Char3]
mov bl, [eax.Char4]
```

The `eax.` prefix makes the code more intuitive and readable. These labels do not cause an assembly error because EAX is simply part of the label name and the period is treated as an ordinary character in this context. VIRTUAL becomes still more useful when registers address structured data; that use is discussed in Section 1.4.

### 1.3.3. Jumps

A jump is a transfer of control to another instruction. In a sense, control transfers after every instruction: EIP (IP, RIP) contains the address of the next instruction, and after the current instruction the processor executes the instruction at that address. The instruction pointer cannot be modified directly; the processor updates it. Jump instructions are used when execution must continue somewhere other than the ordinary next instruction. Jumps are unconditional or conditional.

The three basic unconditional control-transfer instructions are JMP, CALL, and RET. Conceptually, JMP changes EIP (IP, RIP) to the address supplied by its operand. The operand may be a value in memory, a register containing an address, or a direct address. The book illustrates the idea with the following non-executable equivalent:

```
mov eip, <address>
```

CALL transfers control while saving the address of the following instruction on the stack so the called function or procedure can return to the caller. The operand format is the same as JMP. The book gives this conceptual equivalent:

```
push eip
jmp <address>
```

RET takes the return address from the top of the stack and transfers control there. It has no operands. The conceptual equivalent printed by the book is `pop eip`.

RETN may take one operand specifying how many bytes are to be removed from the stack before returning. The source gives this equivalent for RETN `n`:

```
pop temp
sub esp, n
mov eip, temp
```

RETF performs a far return after an intersegment control transfer. It may also take an operand like RETN. For RETF `n`, the source gives:

```
pop temp_eip
pop temp_cs
sub esp, n
mov cs, temp_cs
mov eip, temp_eip
```

When the transfer or return changes privilege level, the stack is switched. Stack switching is discussed in Section 2.5. The pseudo-equivalents above are not executable because EIP cannot be written directly; they are presented only to explain the idea.

*Editorial note: the pseudo-code for RETN `n` and RETF `n` uses `sub esp, n`. Removing arguments from a downward-growing x86 stack would normally advance the stack pointer rather than subtract from it. The printed pseudo-code is preserved.*

JMP and CALL may use a memory operand or a direct address. A direct address is usually a label. The assembler analyzes the distance and can emit a short encoding when the target is no more than 127 bytes forward or 128 bytes backward; otherwise it emits a longer encoding. The short form can be generated only when the operand is not bracketed, not written with PTR, and is not a register.

Conditional jumps transfer control only when their condition is true. Each depends on one or more flags. For example, JZ jumps only when ZF is set. Conditional jumps take a near label and can transfer control only 127 bytes forward or 128 bytes backward.

*Editorial note: the range statement describes short conditional-jump encodings. x86 processors from the 80386 onward also provide near conditional jumps with larger relative displacements. The source statement is retained and qualified here.*

Conditional jumps are commonly used with comparison instructions. CMP compares two operands and modifies the flags register. Syntax: `cmp <operand1>, <operand2>`.

The source allows the first operand to be a register or memory value of 1, 2, 4, or 8 bytes, and the second to be a register, memory value, or immediate value. It states that an immediate value cannot be 64 bits, that both operands cannot simultaneously be memory, and that a 32-bit value compared with a 64-bit value is zero-extended. Table 1.5 lists the conditional jumps for a preceding `cmp x, y`.

**Table 1.5. Conditional-jump instructions**

| Instruction     | Condition   | Flags/condition                   |
|-----------------|-------------|-----------------------------------|
| JA              | $X > Y$     | $CF = 0 \ \& \ ZF = 0$            |
| JAEC            | $X \geq Y$  | $CF = 0$                          |
| JB              | $X < Y$     | $CF = 1$                          |
| JBEC            | $X < Y$     | $CF = 1 \ \text{or} \ ZF = 1$     |
| JC              |             | $CF = 1$                          |
| JCXZ            |             | $CX = 0$                          |
| JE (same as JZ) | $X = Y$     | $ZF = 1$                          |
| JG              | $X > Y$     | $ZF = 0 \ \& \ SF = OF$           |
| JGE             | $X \geq Y$  | $SF = OF$                         |
| JL              | $X < Y$     | $SF \neq OF$                      |
| JLE             | $X \leq Y$  | $ZF = 1 \ \text{or} \ SF \neq OF$ |
| JNA             | $(X \leq Y$ | $CF = 1 \ \text{or} \ ZF = 1$     |
| JNAEC           | $X < Y$     | $CF = 1$                          |
| JNB             | $X \geq Y$  | $CF = 0$                          |
| JNBEC           | $X > Y$     | $CF = 1 \ \& \ ZF = 0$            |
| JNC             |             | $CF = 0$                          |
| JNE             | $X \neq Y$  | $ZF = 0$                          |
| JNG             | $X \leq Y$  | $ZF = 1 \ \text{or} \ SF \neq OF$ |
| JNGEC           | $X < Y$     | $SF \neq OF$                      |
| JNL             | $X \geq Y$  | $SF = OF$                         |
| JNLE            | $X > Y$     | $ZF = 0 \ \& \ SF = OF$           |
| JNO             |             | $OF = 0$                          |
| JNP             |             | $PF = 0$                          |
| JNS             |             | $SF = 0$                          |
| JNZ             | $X \neq Y$  | $ZF = 0$                          |
| JO              |             | $OF = 1$                          |
| JP              |             | $PF = 1$                          |
| JPE             |             | $PF = 1$                          |
| JPO             |             | $PF = 0$                          |
| JS              |             | $SF = 1$                          |
| JZ              | $X = Y$     | $ZF = 1$                          |

*Editorial note: the JBE row prints  $X < Y$  although its flag condition corresponds to unsigned  $X \leq Y$ ; the JNBE row prints  $CF = 1 \ \& \ ZF = 0$ , although JNBE/JA normally requires  $CF = 0$  and  $ZF = 0$ . The opening parenthesis in the JNA condition is also present in the source.*

TEST is useful in some situations. Its format is almost the same as CMP except that the source does not allow a memory operand as the second operand. TEST performs a bitwise AND and, changes only SF, ZF, and PF. It is useful for testing a value against a bit mask.

*Editorial note: TEST also has defined effects on CF and OF in the architecture. The source lists only SF, ZF, and PF here.*

### 1.3.4. Anonymous Labels

FASM can create anonymous labels. An anonymous label is written `@@:`. To jump to the nearest anonymous label after the jump, use `@f`; to jump to the nearest anonymous label before it, use `@b`.

#### Listing 1.12. Using anonymous labels

```
...
@@:
    ; code
    Ja @f      ; jump forward
;code
    Jz @b      ; jump backward
;code
@@:
...
```

Anonymous labels can be useful in complex algorithms because they prevent the code from becoming crowded with many similar named labels.

### 1.3.5. Working with Bits

Sometimes a program must inspect a particular bit or change its state. Four instructions are provided for this purpose: BT, BTC, BTR, and BTS. All four use the general form `bt* <operand>, <bit number>`. The operand may be a register or a memory variable, and the bit number may be an immediate value or a general-purpose register. Bit numbering begins at zero.

BT copies the selected bit into CF. BTC copies the bit into CF and then complements that bit in the operand. BTR copies it into CF and clears the bit. BTS copies it into CF and sets the bit. Because every instruction reports the old bit in CF, JC and JNC can be used immediately afterward.

#### Listing 1.13. Using the BT instruction

```
mov ax, 4      ; ax = 100b
bt ax, 2       ; test the third bit in ax
jc metka       ; jump to the label
....
metka:
....
```

BT, BTC, BTR, and BTS can help optimize code and may replace TEST when only one specific bit must be checked.

### 1.3.6. Summary

This section examined data and labels in assembly-language programs, together with conditional and unconditional jumps. Both forms of control transfer are fundamental because they make branching programs possible. Section 1.4 examines assembly instructions in greater detail.

## 1.4. A Closer Look at Assembly

The previous two sections introduced the basic assembly-language instructions. This section examines more complex and equally important instructions in greater detail.

### 1.4.1. Working with Memory and the Stack

**XCHG** exchanges the values of two operands. Syntax: `xchg <operand1>, <operand2>`. Each operand may be a register or a memory value, but both operands cannot be memory at the same time. The operands may be of any size.

**ADC** adds with carry. Its syntax is identical to **ADD**. After adding the two operands, **ADC** also adds the value of **CF** to the result; if **CF** is set, the result is therefore incremented by one.

**LEA** loads an effective address. Syntax: `lea <register>, <variable>`. The source notes that LEA can also perform simple address arithmetic. For example `lea eax, [ebx + ebx * 4]` produces EBX multiplied by five, and `lea ecx, [edx + 4]` can replace a MOV/ADD pair.

**PUSHF** stores the flags register on the stack. The source says PUSHF stores the first two bytes; in protected mode the 32-bit form is PUSHFD and in 64-bit mode the 64-bit form is PUSHFQ. POPF, POPFD, and POPFQ restore flags from the stack.

Reserved flags cannot be changed. The source also notes that code at an unprivileged protection level cannot modify privileged flags.

**PUSHA** stores the general-purpose registers on the stack. The 32-bit protected-mode form is PUSHAD. The source gives the order EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, with the saved ESP being the value from before the instruction began. The source says PUSHA/PUSHAD are not supported in 64-bit mode.

**POPA** restores general-purpose registers. The stack position corresponding to ESP is ignored. The 32-bit form is POPAD. PUSHA/POPA are described as available in real-address and protected modes only; in 64-bit mode they generate #UD.

**NOP** takes no operands and performs no useful computation. The source describes it as occupying code space and execution time, and says it can reserve space in the code segment or be used for a software delay.

### 1.4.2. Working with Numbers in Assembly

The source returns to signed and unsigned integers. Floating-point values are represented by 32-, 64-, or 80-bit binary encodings and are normally processed by the mathematical coprocessor, although software floating-point arithmetic could be implemented manually.

For unsigned n-bit integers the source gives the range 0 to  $2^n - 1$ . For signed values it describes the high bit as a sign indication and discusses the remaining bits in terms of magnitude or inverted magnitude plus one, giving the range  $-2^{(n-1)}$  to  $2^{(n-1)} - 1$ .

At the machine-instruction level, the source says the bit pattern itself has no inherent signedness. Signedness is an interpretation used by the programmer and by particular instructions and flag conditions.

For example, the 16-bit bit pattern 1001101101011100 is 9B5Ch (39772) if interpreted as unsigned, but represents -25764 when interpreted as signed according to the source.

Addition and subtraction use the same binary operation regardless of signedness. CF indicates carry/borrow beyond the operand width; OF indicates that a signed result lies outside the representable signed range.

**Table 1.6. Examples of additions and their results**

| No. | Operands            | Result    | Flags         | Unsigned interpretation   | Signed interpretation         |
|-----|---------------------|-----------|---------------|---------------------------|-------------------------------|
| 1   | 10000000b 10000000b | 00000000b | CF = 1 OF = 1 | 128 + 128 = 0 incorrect   | (-127) + (-127) = 0 incorrect |
| 2   | 01000000b 01000000b | 10000000b | CF = 0 OF = 1 | 64 + 64 = 128 correct     | 64 + 64 = -128 incorrect      |
| 3   | 01111111b 00000001b | 10000000b | CF = 0 OF = 1 | 127 + 1 = 128 correct     | 127 + 1 = -128 incorrect      |
| 4   | 11111100b 00000101b | 00000001b | CF = 1 OF = 0 | 252 + 1 = 1 incorrect     | (-4) + 5 = 1 correct          |
| 5   | 01100000b 11100000b | 01000000b | CF = 1 OF = 0 | 96 + 224 = 64 incorrect   | 96 + (-32) = 64 correct       |
| 6   | 11000000b 11000000b | 10000000b | CF = 1 OF = 0 | 192 + 192 = 128 incorrect | (-64) + (-64) = -128 correct  |
| 7   | 11111111b 00000001b | 00000000b | CF = 1 OF = 0 | 255 + 1 = 0 incorrect     | (-1) + 1 = 0 correct          |

Table 1.6 is translated from the printed table without silently correcting its arithmetic text. In particular, row 1 prints -127 for the bit pattern 10000000b, and row 4 prints the unusual expression '252 += 1'.

The source explains three cases: both CF and OF set, OF set without CF, and CF set without OF. It concludes that for unsigned addition/subtraction, CF = 1 indicates overflow of the unsigned range, while for signed addition/subtraction, OF = 1 indicates a signed overflow or loss of the expected sign.

The same reasoning applies to subtraction, except that borrows replace carries.

When an integer is wider than the processor's directly supported operand size, arithmetic can be performed in parts. The source uses 64-bit addition in protected mode as the example: add the low 32-bit halves with ADD, then the high halves with ADC so that a carry from the low half is included.

#### Listing 1.14. Adding two unsigned 64-bit numbers

```
Val1_high dd 0750AC072h
Val1_low dd 0780D1E56h
Val2_high dd 0004B015Fh
Val2_low dd 0E057DAF1h
...
mov eax, [Val1_low]
mov edx, [Val1_high]
add eax, [Val2_low]
adc edx, [Val2_high]
jnc @f
; overflow occurred
```

```

@@:
; calculation succeeded

```

After this code, EDX:EAX contains the 64-bit sum. For signed addition, OF should be checked after the operation.

The prose following Listing 1.14 refers to it as 'Listing 1.4.1' in the source. The listing itself is correctly numbered 1.14.

#### Listing 1.15. Adding two unsigned 64-bit numbers stored as single variables

```

Val1 dq 05B493E89C0F44Ah
Val2 dq 06A4937796D80D1h
...
mov eax, dword [Val1]
mov edx, dword [Val1+4]
add eax, dword [Val2]
adc edx, dword [Val2+4]
jnc @f
; overflow occurred
@@:
; calculation succeeded

```

When a multi-byte number is stored as one variable, the source reminds the reader that little-endian order places the low bytes first.

### 1.4.3. Multiplication and Division

**MUL** performs unsigned multiplication. Syntax: `mul <operand>`. The operand is multiplied by AL, AX, EAX, or RAX depending on operand width, and the product is stored in AX, DX:AX, EDX:EAX, or RDX:RAX.

**Table 1.7. MUL result by operand size**

| Operand size | Result                  |
|--------------|-------------------------|
| 1 byte       | AX = AL * operand       |
| 2 bytes      | DX:AX = AX * operand    |
| 4 bytes      | EDX:EAX = EAX * operand |
| 8 bytes      | RDX:RAX = RAX * operand |

**DIV** performs unsigned division. Syntax: `div <operand>`. The dividend is AX, DX:AX, EDX:EAX, or RDX:RAX depending on operand width; quotient and remainder are returned in the corresponding low/high registers.

**Table 1.8. DIV operands and results by operand size**

| Operand size | Dividend | Quotient | Remainder |
|--------------|----------|----------|-----------|
| 1 byte       | AX       | AL       | AH        |
| 2 bytes      | DX:AX    | AX       | DX        |
| 4 bytes      | EDX:EAX  | EAX      | EDX       |
| 8 bytes      | RDX:RAX  | RAX      | RDX       |

**IMUL** performs signed multiplication and has three forms in the source.

1. One operand: operates like MUL but with signed multiplication.
2. Two operands: `imul <operand1>, <operand2>`. The first operand must be a general-purpose register. The second may be a register, memory value, or immediate. The result is stored in the first operand. The source says an immediate second operand cannot be 8 bytes.
3. Three operands: `imul <operand1>, <operand2>, <operand3>`. The first operand is a general-purpose register; the second is a register or memory value of the same size; the third is an immediate value, not 8 bytes. Operand2 is multiplied by operand3 and the result is stored in operand1.

**IDIV** performs signed division. The source states that its register result locations are the same as DIV and describes the sign of quotient and remainder in terms of the operands' sign bits.

DIV and IDIV do not define the flags according to the source. MUL clears OF and CF when the high half of the product is zero and sets them otherwise. For the one-operand IMUL form, OF and CF indicate whether significant bits are required in the high half; for the two- and three-operand forms, they indicate whether the full product exceeds the destination width.

#### 1.4.4. I/O Ports

The processor communicates with external hardware through I/O ports. Each port is described as one byte wide; two adjacent ports form a 2-byte port and four adjacent ports form a 4-byte port.

Writing to or reading from a port causes the connected controller or device logic to react. If more than four bytes must be transferred, the device can present successive portions through the port.

Processor I/O ports are not physical connectors such as PS/2, USB, or IEEE ports. From the processor's point of view, the directly addressed external entity is normally a controller or chipset component - for example, a USB bus controller.

The processor has 65,535 I/O ports. It gives the example that writing 1AF4h as a two-byte value starting at port 56h corresponds to 1Ah at port 56h and F4h at port 57h. It recommends alignment of 4-byte ports to multiples of four and 2-byte ports to multiples of two.

*Editorial note: a 16-bit I/O-port number provides 65,536 possible port numbers (0 through 65535), but the book prints '65,535 ports'. The translation preserves the source statement and notes the discrepancy here.*

Four instructions are introduced: IN, OUT, INS, and OUTS. INS and OUTS are discussed later.

**IN** reads a port. Syntax: `in EAX/AX/AL, <port>`. The port may be an immediate number or DX. An immediate port number must not exceed 255; larger port numbers must be placed in DX. Operand size selects the port width.

**OUT** writes a port. Syntax: `out <port>, EAX/AX/AL`. The source again says a port number greater than 255 must be supplied through DX.

### 1.4.5. Loops

A loop can be organized by placing a repetition count in a register, decrementing it each time, and branching while it remains nonzero.

#### Listing 1.16. Loop using DEC/JNZ

```
mov ecx, <number of repetitions>
metka1: ...
<body of loop>
...
dec ecx
jnz metka1
```

DEC decreases ECX, setting ZF when the result becomes zero; JNZ returns to the label while the count remains nonzero.

**LOOP** takes a near label, decrements ECX (CX or RCX), and transfers control if the register is not zero. The source says its main advantage is smaller code size.

**LOOPE/LOOPZ** also requires ZF to be set for the branch to occur. **LOOPNE/LOOPNZ** branches only while ZF is clear.

On modern processors a DEC/JNZ or DEC/JZ style loop is preferable for performance and may run about twice as fast as a LOOP instruction.

### 1.4.6. Processing Blocks of Data

The source introduces the string/block instructions before discussing their REP prefixes.

**LDSB/W/DI/Q** loads a byte, word, doubleword, or quadword from memory addressed by ESI (SI/RSI) into AL, AX, EAX, or RAX. ESI is then incremented or decremented by 1, 2, 4, or 8 depending on DF.

#### Source's illustrative equivalent for a 2-byte LODS

```
mov ax, word [esi]
add/dec esi, 2
```

*The source literally prints add/dec esi, 2. This is preserved as printed rather than being silently replaced by separate ADD and DEC alternatives.*

**STOSB/W/DI/Q** stores AL/AX/EAX/RAX to memory addressed by EDI (DI/RDI), then adjusts EDI according to DF.

**MOVS/W/DI/Q** copies data from memory addressed by ESI (SI/RSI) to memory addressed by EDI (DI/RDI), then adjusts the index registers.

*Editorial note: in the prose describing MOVS and CMPS, one of the source-index references is printed as EDI where ESI is evidently being discussed. The translation keeps the intended source/destination roles explicit without changing any listing code.*

**SCASB/W/DI/Q** compares AL/AX/EAX/RAX with memory addressed by EDI and updates flags like CMP, then adjusts EDI.

**CMPSB/W/DI/Q** compares memory addressed by ESI with memory addressed by EDI, updates flags like CMP, and adjusts the indexes.

**INSB/W/D** transfers a byte, word, or doubleword from the port in DX to memory addressed by EDI. **OUTSB/W/D** transfers a byte, word, or doubleword from memory addressed by ESI to the port in DX.

DS is used with ESI for the source logical address and ES is used with EDI for the destination logical address.

These instructions become especially useful with REP, REPE/REPZ, and REPNE/REPZ. REP repeats while ECX (CX/RCX) is nonzero, decreasing the count each iteration.

#### Listing 1.17. Using STOS

```
metka:
db 100 dup (0)
...
mov ecx, 100
mov edi, metka
mov al, 0dah
rep stosb
```

Listing 1.17 fills the 100 bytes at metka with 0DAh.

#### Listing 1.18. Using MOVS

```
metka1:
db 40 dup (11h)
metka2:
db 40 dup (0)
mov ecx, 10
mov esi, metka1
mov edi, metka2
rep movsd
```

REPE/REPZ repeats while the count remains nonzero and ZF is set. REPNE/REPNZ repeats while the count remains nonzero and ZF is clear.

#### Listing 1.19. Using SCAS

```
str1: db "9"
      db 0
str2: db "qwerty987654"
endstr:
...
start:
mov ecx, endstr-str2+1
std
mov al, [str1]
mov edi, endstr-1
repnz scasb
```

The source describes Listing 1.19 as searching for the character str1 in str2 from the end. The remaining ECX value is described as the position of the first matching character from the end; ECX = 0 means no match.

#### Listing 1.20. Using CMPS

```
str1: db "qwerty99",0
endstr1:
str2: db "qwerty98",0
endstr2:
...
start:
mov eax, endstr1-str1
mov ecx, endstr2-str2
cmp ecx, eax
jnz _not
go:
mov esi, str1
mov edi, str2
repz cmpsb
cmp ecx, 0
jz _yes
_not:
; strings are not equal
_yes:
; strings are equal
```

Listing 1.20 compares the two strings defined by their labels.

### 1.4.7. Macros

This material concerns FASM's facilities more than processor instructions, but assembly programming becomes much more difficult without knowledge of FASM's macro language. It groups constants, macros, and structures under these macro facilities.

The source describes two ways to define constants. The simplest uses the equals sign. Such a constant can be declared anywhere and its value is substituted wherever the constant appears.

The second method uses EQU, which the source describes as more flexible.

#### Listing 1.21. Using EQU and =

```
_C = 45;
_name = 0A1h
Value = "S"; same as Value = 53h
CONST1 equ 0123h
CONST2 equ 14d*15h
CONST3 equ "slovo"
```

```

CONST4 equ 56-45
CONST5 equ (metka1)
CONST6 equ (metka2+ metka3)
CONST7 equ (CONST2+10b)
CONST8 equ CONST7/2
CONST9 equ (metka1-metka5)
CONST10 equ (metka4+CONST4)
CONST11 equ add edx,edi

```

*The identifier CONST6 uses a zero after C in the printed source and is preserved.*

The examples show that a constant may represent a label address and even an instruction. When a constant represents an instruction, the source says it cannot be used as an operand; when placed by itself it is replaced by its value.

The source then demonstrates an important difference between = and equ.

#### Listing 1.22. Difference between EQU and =

```

C1 = 10
C2 equ 2
C3 equ C1+C2
C4 = C1+C2
mov eax, C3 ; mov eax, C1+C2
mov eax, C4 ; mov eax, 12

```

With =, the source says the constant's value is evaluated immediately and the evaluated value is substituted. With equ, the definition itself is substituted without being evaluated at declaration time, so the eventual replacement may depend on later definitions.

A macro is described as a more advanced form of EQU: a set of instructions substituted wherever the macro name is invoked.

#### General macro form

```

macro <macro name> <macro parameters>
{
<macro body>
}

```

For example, macro `tst {cmp eax, ebx}` defines TST so that it expands to `cmp eax, ebx`. The source notes that a macro name is recognized as an instruction mnemonic, and macro expansion occurs before symbolic constants are replaced.

#### Listing 1.23. Macro with three parameters

```

macro mov op1,op2,op3
{
if op3 eq
mov op1,op2
else
mov op1,op2
mov op2,op3
end if
}

```

The source introduces conditional assembly with IF, ELSE, ELSEIF, END IF, and EQ. IF assembles its block when the condition is true; ELSE assembles when previous conditions have failed; ELSEIF checks an additional condition after earlier failures; END IF terminates the chain. EQ tests expressions for identity.

In Listing 1.23, if a third operand is supplied, the source describes the expansion as assigning op1 from op2 and then op2 from op3:

#### Example expansion

```

mov eax, ebx, edx

; expands to:
mov eax, ebx
mov ebx, edx

```

If only two operands are supplied, the macro expands to the ordinary two-operand MOV. FASM's full macro engine is a large topic of its own and limits this section to the basics needed for later examples.

### 1.4.8. Structures

A structure is a set of variables. FASM defines a structure with the STRUC directive.

#### Listing 1.24. Declaring a structure

```

Struc sample
{
.x1 dd ?
.y1 dw ?
.y2 db ?
}

```

After the structure has been declared, a variable can be defined and its fields referenced:

#### Structure instance and field access

```

...
_Var sample
...
mov ax, [_Var.y1]

```

The field expression is replaced by the address of the structure variable plus the offset of that field. The source gives the equivalent:

#### Equivalent address expression

```
mov ax, word [_Var+4]
```

If field values must be supplied when the structure instance is declared, parameters can be used in the same way as macro parameters.

#### Listing 1.25. Declaring a parameterized structure

```

struc numbers x,y
{
.x1 dd x
.y1 dw ?
.y2 db y
}

```

All parameters must then be supplied when a variable is declared:

#### Parameterized structure instance

```

...
_Var numbers 34, 67
...
mov ax, [_Var.y1]

```

To use structures addressed through registers, the source again uses VIRTUAL.

#### Listing 1.26. Using VIRTUAL with a structure

```

...
_Var numbers
...
mov ebx, _Var
virtual at ebx
.ebx numbers
end virtual
mov dl, [.ebx.y2]

```

The .ebx label prefix is required because FASM does not allow the register name EBX itself to be used as a label name.

### 1.4.9. Working with MSR Registers

MSRs (Model Specific Registers) are registers whose definitions may differ between processor families. Each MSR has an index and all MSRs discussed here are 64-bit. Two privileged instructions are used: RDMSR and WRMSR.

**RDMSR** reads the MSR selected by ECX (RCX in the source's wording) into EDX:EAX (RDX:RAX). In 64-bit mode, the source says the high parts of RDX and RAX are cleared.

**WRMSR** writes EDX:EAX (RDX:RAX) to the MSR selected by ECX (RCX). In 64-bit mode the source says only the low parts of RDX and RAX are used.

These instructions were introduced with Pentium I and are privileged, usable at privilege level 0; it also says they may be executed in real-address mode.

### 1.4.10. The CPUID Instruction

CPUID identifies processor features. Depending on the information selector placed in EAX - and sometimes ECX - it returns information in EAX, EBX, ECX, and EDX.

For example, with EAX = 0, the source says EAX returns the maximum basic selector value that can be supplied to CPUID.

Listing 1.27 obtains the processor brand string. After execution, message contains the name string.

#### Listing 1.27. Using CPUID

```

message db 49 dup (0)
...
mov edi, message
mov eax, 80000002h
cpuid
mov [edi], eax
add edi, 4
mov [edi], ebx
add edi, 4
mov [edi], ecx
add edi, 4
mov [edi], edx
add edi, 4

```

```

mov eax, 80000003h
cpuid
mov [edi], eax
add edi, 4
mov [edi], ebx
add edi, 4
mov [edi], ecx
add edi, 4
mov [edi], edx
add edi, 4
mov eax, 80000004h
cpuid
mov [edi], eax
add edi, 4
mov [edi], ebx
add edi, 4
mov [edi], ecx
add edi, 4
mov [edi], edx

```

Selector 80000002b returns the first 16 bytes of the processor-name string in EAX, EBX, ECX, EDX; selector 80000003h returns the next 16 bytes; and selector 80000004b returns the final 16 bytes.

*Editorial note: the prose prints 80000002b and 80000004b, while Listing 1.27 uses the hexadecimal forms 80000002h and 80000004h. The discrepancy is preserved rather than normalized silently.*

More detailed processor-feature detection is deferred until later chapters, where particular mechanisms require checking processor support.

### 1.4.11. The UD2 Instruction

UD2 is described as deliberately encoding an invalid opcode. It exists specifically to guarantee generation of the invalid-opcode exception. Unsupported instructions may also cause #UD, but UD2 is designed to do so predictably.

### 1.4.12. Including Files

When the same structures and macros are needed by multiple programs, redefining them in each source file is inconvenient. FASM's `include` directive inserts the text of another file into the program.

The path to the file is written in quotation marks, for example `include 'file.ext'`. The source says the file may use any extension such as `.txt`, `.inc`, or `.asm` as long as it is a text file. The effect can be thought of as replacing the `INCLUDE` directive with the entire contents of the included file.

The file may be included at any point in a program. Identifiers declared in it - constants, labels, macros, and so on - can then be used in the main program without additional restrictions.

### 1.4.13. Summary

This brings Chapter 1 to an end. The chapter introduced the main assembly instructions and established a foundation for programming in the processor's different operating modes. The next chapter, beginning with Section 2.1, covers protected mode. From that point onward, the source assumes the reader has an intermediate level of knowledge and understands the instructions used in the examples.

# Chapter 2. Protected Mode

## 2.1. Introduction to Protected Mode

In the previous chapter we studied the fundamentals of assembly language. With those basic concepts in place, we can proceed to protected mode. This chapter examines the principles and mechanisms by which the processor operates in protected mode. This section gives a general description, and its practical part builds a program that switches the processor into protected mode.

Protected mode is so named because it allows operating-system data to be protected from applications and also allows application data to be protected from other applications. This protection is achieved by dividing software among different privilege levels.

After power-on or a hardware reset, the processor is in real-address mode - the 16-bit mode. A 20-bit addressing system in this mode, allowing only 1 MB of physical memory to be addressed, and gives MS-DOS is described as an operating system that runs there. From this point onward the chapter concentrates on protected mode, which the source presents as a way to use the computer's memory, performance, protection mechanisms, and other resources more fully.

### 2.1.1. Privilege Levels

Real-address mode and protected mode are fundamentally different. In protected mode the processor checks whether the running program obeys various conditions: for example, it may be forbidden from executing certain instructions or accessing particular memory regions. If a condition is violated, the processor generates a special kind of interrupt called an exception and supplies information describing where and how the violation occurred. An interrupt handler then processes the event and decides what to do with the offending program, for example terminating it.

The operating system is responsible for defining the conditions and restrictions under which programs run. A program that switches the processor into protected mode becomes, in effect, the owner of the machine - the operating system - and establishes rules and permissions for other programs.

To prevent another program from redefining those rules, the processor provides privilege levels. Real-address mode does not have the same privilege system because the source treats it as a mode in which only one program is executing. Protected mode, by contrast, can employ multitasking, so several tasks may share the processor by rapidly alternating.

Any one of those tasks could otherwise damage system or user data. The privilege system prevents an application from changing rules established by the operating system and thereby gaining access to protected resources. The source gives direct disk access through I/O ports as an example of something the operating system can reserve for itself.

Privilege levels are the foundation of protected mode. The source describes four levels numbered 0 through 3. Level 0 is the most privileged; software running there is allowed the broadest access. Level 1 is less privileged, level 2 less privileged again, and level 3 has the lowest privilege.

The source presents the following classical arrangement:

- **Level 0:** operating-system kernel.
- **Level 1:** operating-system drivers.
- **Level 2:** operating-system interface.
- **Level 3:** application programs.

This is not the only possible design. Drivers and user-interface components can be placed at the same level as the kernel. Doing so gives up some of the processor's built-in isolation between components, making the system more vulnerable to failures in kernel-level components. It cites Windows driver failures as an example of a failure that can bring down the whole system.

Privilege levels 1, 2, and 3 operate under rules established by level 0. Functionally, the source therefore groups the four levels into a system level (0) and user levels (1, 2, and 3). Although two levels might initially appear simpler, four levels can be useful; it also notes that popular UNIX and Windows systems commonly use only levels 0 and 3.

Programs and data are constrained by their privilege levels. The source gives the example that a program running at privilege level 2 cannot simply transfer control to code at another level or access data intended for another level. If a privilege check fails, the processor generates a general-protection exception and transfers control to the operating system so that it can respond.

A privilege level is not simply a single value that software 'sets in the processor'. Instead, privilege is a property attached to processor-managed objects such as code segments and is checked when those objects are used.

### 2.1.2. Segments in Protected Mode

Protected mode can use up to 4 GB of physical memory in the standard mode and 64 GB in a special mode discussed later. Before explaining protected-mode memory, it briefly recalls real-mode addressing.

In real-address mode, memory is divided into 64 KB segments. To access memory, a segment number and an offset within that segment are specified. The source gives the physical-address calculation as:

## Real-mode physical-address formula

$\text{physical\_address} = \text{segment} * 10h + \text{offset}$

The offset is a 16-bit value in a general-purpose register or an immediate value, while the segment is a 16-bit value held in a segment register.

A protected-mode memory address is 32 bits, allowing 4 GB to be addressed. Memory is still divided into segments, but a segment may range from 0 to 4 GB in size. Access to memory therefore begins with a logical address: a description of a segment plus an address within that segment.

In real mode the segment was identified by its segment number. In protected mode it is identified by a selector. A segment selector is a 16-bit identifier, the same width as a real-mode segment value but with an entirely different meaning.

A segment must be described by its starting address, the privilege level required for access, its limit (size), and other attributes. Protected mode uses an 8-byte structure called a descriptor for this purpose. More generally, processor-managed objects whose access is controlled by the processor are described by descriptors.

Descriptors are stored in descriptor tables - global and local. Different kinds of objects use different descriptor types. A memory segment is described by a segment descriptor.

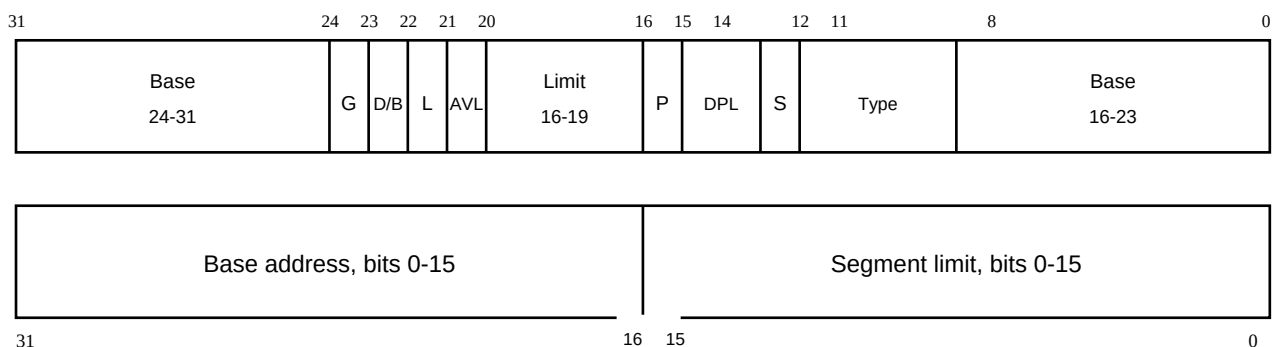


Figure 2.1. General descriptor format

The segment descriptor is conventionally divided into two doublewords. Figure 2.1 reproduces the source layout in English.

**Base address.** A 32-bit address specifying the start of the memory region occupied by the segment.

**Segment limit.** The maximum valid offset within the segment; it can also be viewed as the segment size minus one. A segment may use byte or page granularity. The source states that with byte granularity the maximum segment size is 1 MB, while page granularity allows a 4 GB segment.

**Segment type.** A 4-bit field whose individual bits have the following meanings in the source:

1. The highest type bit (bit 11 of the second doubleword): 0 denotes a data segment and 1 denotes a code segment. If the bit is clear, control cannot be transferred for execution to code stored in the described region.
2. Bit 10: for a code segment, it indicates whether the segment is conforming; for a data segment, it controls the direction in which the segment expands. A value of 0 means upward toward higher addresses; a value of 1 means downward toward lower addresses, as with a stack.
3. Bit 9: for a code segment, it indicates whether the segment is readable - 0 for execute-only, 1 for execute/read. For a data segment, it indicates whether the segment is writable - 0 for read-only, 1 for read/write.
4. Bit 8, A (Accessed): indicates whether the processor has accessed the segment for reading, writing, or execution. The processor sets A when the segment is used; software or the operating system is responsible for clearing it. A newly created descriptor is normally expected to begin with A = 0.

**S (System).** If set, the descriptor defines a code or data segment. If clear, it describes a system object. The source says S is always set in a segment descriptor; otherwise the preceding type bits are interpreted differently.

**DPL (Descriptor Privilege Level).** A 2-bit field defining the privilege level of the object described by the descriptor, with values 0 through 3. The source text prints the abbreviation as 'DPI.' in this paragraph, although the field itself is DPL.

*Editorial note: the prose prints 'DPI. (Descriptor Privilege Level)'. The diagram and architecture use DPL. This edition keeps DPL in the diagram and explicitly records the source typo here.*

**P (Present).** If set, the segment is present in memory; if clear, it is not. The source describes a hypothetical segment-swapping mechanism in which software could save a segment to disk, clear P, and reload it when a not-present exception occurs. It then notes that real virtual-memory paging is normally implemented through page translation and that P is often useful for marking a free descriptor-table entry.

**AVL.** Not used by the processor; software or the operating system may use it for its own purposes.

**L.** Relevant to 64-bit mode. In protected mode it is reserved and clear. The source says that in 64-bit mode, L = 0 selects compatibility mode and L = 1 selects 64-bit mode; if L is set, D must be clear. Chapter 4 returns to this bit.

**D (Default size).** If clear, the described object is treated as 16-bit; if set, as 32-bit. A 32-bit code program should reside in a 32-bit code segment. For SS, D also determines whether the stack uses a 32-bit or 16-bit element/stack address size in the source's description. Protected mode can use both 16-bit and 32-bit segments, though the source assumes new programs will normally use 32-bit segments.

**G (Granularity).** If G = 0, the segment uses byte granularity; if G = 1, it uses 4 KB page granularity. The source gives the example that a limit of 0FFh represents 255 bytes when G = 0 and 1020 KB (255 pages) when G = 1.

All descriptors are stored in the global or local descriptor table. The local descriptor table is introduced later.

### 2.1.3. Global Descriptor Table

The Global Descriptor Table (GDT) is stored in memory and may be placed at any memory address. The source recommends an 8-byte-aligned address because descriptor access is faster when the table is aligned to descriptor boundaries.

A GDT can hold up to 8192 descriptors. Descriptor 0 - the first descriptor in the table - is not used by the processor. An actual access through the null descriptor causes an exception, although loading a null selector into a segment register does not by itself generate one.

The table address and table limit are stored in GDTR. The limit prevents accesses beyond the end of the table. In protected mode, the source describes GDTR as a 6-byte register image. Figure 2.2 reproduces its format.



Figure 2.2. GDTR format

Because the limit field is 16 bits, the table can be at most 64 KB. Dividing 65536 by the 8-byte descriptor size gives 8192 table slots, of which only 8191 are usable because descriptor 0 is reserved.

LGDT loads GDTR. It is a privileged instruction and can be executed by code at privilege level 0. The operand is a 6-byte memory image in the protected-mode form described here. SGDT stores GDTR and, can be executed at any privilege level.

A selector is the index of a descriptor in a descriptor table. It is 2 bytes wide and is held in a segment register or supplied as an immediate value. Figure 2.3 shows the selector format.



Figure 2.3. Selector format

The TI bit selects which descriptor table is used; the book discusses it in more detail in Section 2.4. RPL is the Requested Privilege Level.

The source describes several ways in which a logical address can be specified:

- The segment is supplied by a segment register, while the address/offset is an immediate value or a register.
- The segment selector is supplied directly and the address is an immediate value or a register. The selector is loaded into the relevant segment register - CS for a control transfer, DS for an ordinary memory access in the source's example.
- Only an offset is written explicitly. The selector is then taken from the appropriate segment register - CS for control transfers and DS for ordinary data in the simplified description.

A selector is therefore involved in every protected-mode memory access. The logical-to-linear address conversion shown in Figure 2.4 is performed whenever memory is accessed.

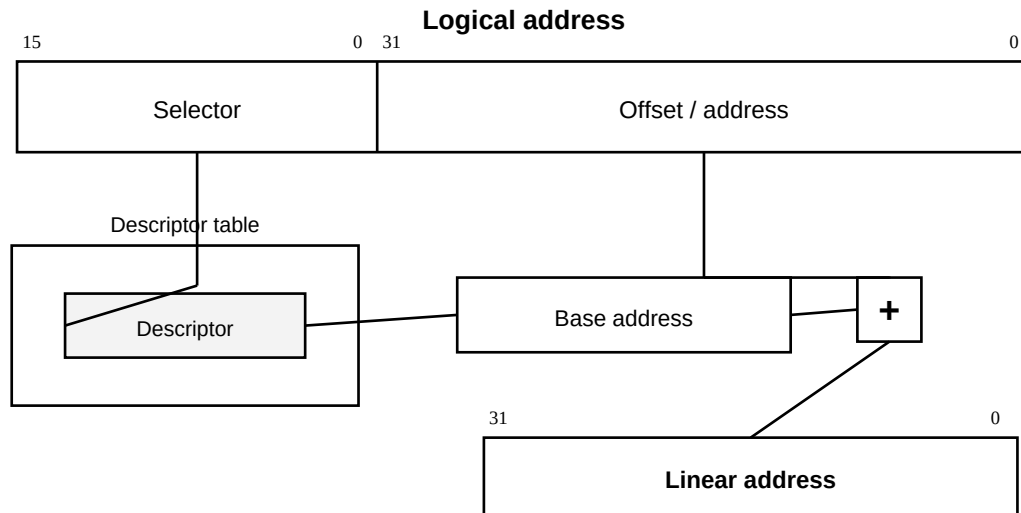


Figure 2.4. Address translation in protected mode

The selector's index field identifies a descriptor. The processor obtains the base-address field from that descriptor and adds the offset. The result is a linear address. That linear address may then be translated into a physical address.

The programs in this section do not yet use paging, so the source treats the linear address as equal to the physical address.

Protection checks are performed when a selector is loaded and used. If software attempts to use an invalid descriptor or one that fails the privilege rules, the processor generates the corresponding exception.

For performance, every segment register has a hidden descriptor-cache portion, inaccessible to ordinary software. When a selector is loaded, the processor performs the protection checks and caches the descriptor in this hidden portion. Subsequent memory accesses use the cached descriptor rather than rereading the table each time.

Consequently, changing a descriptor in memory does not immediately change the state already cached for a segment register; the selector must be loaded again before the new descriptor takes effect there.

#### 2.1.4. Practice

The source summarizes the protected-mode model developed so far: to address any byte of memory, software supplies, explicitly or implicitly, a segment selector and an offset. The segment is described by a descriptor. The descriptors currently known to us are kept in the GDT, whose base address is in GDTR. Descriptor 0 is unused.

After power-on the processor is in real-address mode. The practical task is therefore to write a program that switches from real-address mode to protected mode.

The actual mode-enable control is only one bit: bit 0 of CR0. The following minimal sequence:

##### Minimal CR0 sequence

```

mov eax, cr0
or al, 1
mov cr0, eax

```

But the processor must be prepared before that sequence is executed. First, a valid GDT must exist. Otherwise the value already in CS - a real-mode segment number - will suddenly be interpreted as a protected-mode selector.

Second, after CR0.PE is set, an invalid CS value and the absence of a valid descriptor environment can lead to a general-protection exception. With no interrupt-descriptor table established, the resulting failure can escalate to a double fault and then a reset when no double-fault handler is available. Interrupts and the interrupt table are covered in the next section.

EIP must also be updated so that execution continues at the intended offset in the new code segment. The first protected-mode control transfer must therefore load the code-segment selector into CS and establish the new instruction offset. The source uses a far jump:

##### Far jump form

```

jmp <selector> : <offset>

```

The source asks why CALL or RET should not be used instead. CALL needs a working stack to save its return address, but SS still contains an old real-mode segment value at this point. RET requires a valid return address already stored on a valid stack. A far JMP avoids both problems.

All interrupts must also be disabled before the transition. Otherwise even the next timer interrupt could arrive before a protected-mode interrupt table has been established.

#### Listing 2.1. Program that switches the processor to protected mode

```

ORG 100h
start:
; clear the screen:
mov ax,3
int 10h

; open the A20 line (for 32-bit addressing):
in al,92h
or al,2
out 92h,al

; calculate the linear address of the protected-mode entry point
xor eax,eax
mov ax, cs
shl eax,4
add eax, PROTECTED_MODE_ENTRY_POINT
mov [ENTRY_OFF],eax

; calculate the linear address of the GDT
xor eax,eax
mov ax,cs
shl eax,4
add ax, GDT

; place the linear address of the GDT into the prepared variable:
mov dword [GDTR+2],eax

; load GDTR:
lgdt fword [GDTR]

; disable all interrupts:
cli
in al,70h
or al,80h
out 70h,al

; switch to protected mode:
mov eax,cr0
or al,1
mov cr0,eax

; load the new selector into CS
db 66h ; operand-size override prefix
db 0EAh ; opcode of JMP FAR
ENTRY_OFF dd PROTECTED_MODE_ENTRY_POINT ; 32-bit offset
dw 00001000b ; selector of the first descriptor

; GLOBAL DESCRIPTOR TABLE
GDT:
NULL_descr db 8 dup(0)
CODE_descr db 0FFh,0FFh,00h,00h,00h,10011010b,11001111b,00h
DATA_descr db 0FFh,0FFh,00h,00h,00h,10010010b,11001111b,00h
VIDEO_descr db 0FFh,0FFh,00h,80h,0Bh,10010010b,01000000b,00h
GDT_size equ $-GDT
label GDTR fword
dw GDT_size-1 ; 16-bit GDT limit
dd ? ; 32-bit linear GDT address will be placed here

use32 ; 32-bit code follows
PROTECTED_MODE_ENTRY_POINT:

; load the required selectors into the segment registers
mov ax, 00010000b
mov bx, ds ; code-segment number from real-address mode
mov ds, ax
mov ax, 00011000b
mov es, ax

xor esi,esi
mov si,bx
shl esi,4
add esi, message
xor edi,edi
mov ecx,18

; output to the screen:
rep movsb
jmp $

message: db '1',35h,'2',35h,'3',35h,'4',35h,'5',35h,'6',35h,'7',35h,'8',35h,'9',35h

```

*Listing 2.1 preserves the source code and identifiers. Only Russian comments were translated. In particular, add ax, GDT is preserved exactly as printed rather than changed to EAX.*

The source next explains the program. ORG 100h tells FASM that the program is a DOS COM program. A COM executable is essentially a memory image in a file. DOS loads it into a free segment beginning at offset 100h (256 bytes); the first 256 bytes of the segment are reserved for DOS process information. A COM program therefore fits within one 64 KB segment.

The GDT contains descriptors for three segments: code, data, and video memory. The code and data segments cover the full linear address space. The video segment begins at 0B8000h and has limit 0FFFFh.

The first two instructions clear the screen in text mode using the firmware/video service invoked by INT 10h. The next three instructions enable the A20 address line by setting bit 1 of I/O port 92h. Without A20, classic real-mode address wrapping would cause an access at 100000h to wrap to address 0.

Two groups of instructions then calculate the linear addresses of the protected-mode entry point and the GDT. FASM has assembled labels according to DOS COM rules as offsets within the current real-mode segment, so their linear addresses must be calculated using:

#### Address calculation used by the source

```
physical_address = segment * 10h + offset
```

Multiplication by 10h (16, or  $2^4$ ) is equivalent to shifting left by four bits. The calculated addresses are stored in the prepared variables.

LGDT then loads GDTR. Interrupts are disabled and CR0.PE is set. The first protected-mode transfer is encoded manually using the operand-size prefix and the FAR JMP opcode; its target address was calculated earlier.

The data and string-operation segment registers are then loaded with protected-mode selectors. The program copies a string directly into text video memory. In this text mode, each character cell uses two bytes: the first byte is the character and the second contains foreground/background color information. Each color component is described as four bits; for example, attribute 0Fh gives white text on a black background.

The source address of the string is again converted from the DOS segment-relative label value. After the output operation, the program loops forever and requires a reset to leave that state.

Repeatedly calculating real-mode linear addresses is inconvenient, so the source introduces a revised program.

#### Listing 2.2. Revised program that switches the processor to protected mode

```
ORG 100h
STACK_BASE_ADDRESS equ 200000h
USER_PM_CODE_BASE_ADDRESS equ 400000h
USER_PM_CODE_SIZE equ USER_PM_CODE_END - USER_PM_CODE_BASE_ADDRESS
CODE_SELECTOR equ 8h
DATA_SELECTOR equ 10h
VIDEO_SELECTOR equ 18h

start:
mov ax, 3
int 10h
in al, 92h
or al, 2
out 92h, al

xor eax, eax
mov ax, cs
shl eax, 4
add eax, PROTECTED_MODE_ENTRY_POINT
mov [ENTRY_OFF], eax

xor eax, eax
mov ax, cs
shl eax, 4
add ax, GDT
mov dword [GDTR+2], eax
lgdt fword [GDTR]

cli
in al, 70h
or al, 80h
out 70h, al

mov eax, cr0
or al, 1
mov cr0, eax

db 66h
db 0EAh
ENTRY_OFF dd PROTECTED_MODE_ENTRY_POINT
dw CODE_SELECTOR

align 8
GDT:
NULL_descr db 8 dup(0)
CODE_descr db 0FFh, 0FFh, 00h, 00h, 00h, 10011010b, 11001111b, 00h
DATA_descr db 0FFh, 0FFh, 00h, 00h, 00h, 10010010b, 11001111b, 00h
VIDEO_descr db 0FFh, 0FFh, 00h, 80h, 0Bh, 10010010b, 01000000b, 00h
```

```

GDT_size equ $-GDT
label GDTR fword
dw GDT_size-1
dd ?

use32
PROTECTED_MODE_ENTRY_POINT:
mov ax, DATA_SELECTOR
mov ds, ax
mov es, ax
mov ss, ax
mov esp, STACK_BASE_ADDRESS

call delta
delta:
pop ebx
add ebx, USER_PM_CODE_START-delta
mov esi, ebx
mov edi, USER_PM_CODE_BASE_ADDRESS
mov ecx, USER_PM_CODE_SIZE
rep movsb

mov eax, USER_PM_CODE_BASE_ADDRESS
jmp eax

USER_PM_CODE_START:
ORG USER_PM_CODE_BASE_ADDRESS
include 'PM_CODE.ASM';
USER_PM_CODE_END:

```

The first immediately visible change is the `align 8` directive before `GDT`. `ALIGN` moves the following code/data to the specified alignment boundary so that the subsequent offset is a multiple of the requested value.

*Editorial note: the code says `align 8`, but the following prose says the `GDT` will be aligned on a 4-byte boundary. The code is preserved and the mismatch is noted here.*

Skipped alignment space is filled with `NOP` instructions. After entering protected mode, the revised program initializes a stack whose base is at `200000h` (2 MB). The text describes allocating 4 MB around that point - 2 MB downward and 2 MB upward.

The main protected-mode code is stored in `PM_CODE.ASM` and is intended to execute from address `400000h`. The bootstrap code copies it there. To locate the in-file copy of that code, the program uses the `CALL/POP 'delta'` technique:

#### Delta calculation from Listing 2.2

```

call delta
delta:
pop ebx
add ebx, USER_PM_CODE_START-delta

```

`CALL` transfers control to the next instruction while pushing its address, allowing `POP EBX` to recover the current code position. The program then adds the compile-time difference to `USER_PM_CODE_START`.

`ORG` changes the addresses `FASM` assigns to labels after the directive without moving those bytes to a different position in the file. This lets the code assembled from `PM_CODE.ASM` behave as though it were located at `400000h` after it is copied there.

#### Listing 2.3. Using `ORG`

```

ORG 100h
db 10h dup (0)
metka1: ; label has address 110h
ORG 500h
metka2: ; label has address 500h
db 10h dup (0)
metka3: ; label has address 510h

```

After the `PM_CODE.ASM` block is copied, control is transferred to it. From that point the code can use its labels directly without manually converting them according to real-mode segment arithmetic.

#### Listing 2.4. Code executed in protected mode

```

mov ESI, message
mov EDI, 0B8000h
mov ECX, 18
rep movsb
jmp $
message: db "152535455565758595"

```

Listing 2.4 displays the string `'123456789'`. The stored byte string alternates each digit with the character `'5'`, whose ASCII value is `35h`; the source uses those alternating bytes as text-mode display attributes.

This program becomes the framework for later examples: subsequent programs place their protected-mode code in `PM_CODE.ASM`.

The source then discusses how to run the program. Assembled as a `COM` file, it is intended for DOS. Although historical versions of Windows could run many DOS programs, executing this particular program under ordinary Windows user mode fails at `LGDT` because `LGDT` is privileged.

The program therefore needs to run in real-address mode, for example under MS-DOS booted from removable media. The source mentions a bootable DOS floppy or USB flash drive and recommends placing that device ahead of the normal system disk in the firmware boot order.

Testing protected-mode experiments on a physical computer is inconvenient because a mistake may hang or reset the machine, requiring repeated reboots. The source therefore recommends virtual-machine emulators.

It describes QEMU as a convenient free, open-source emulator available on many platforms with relatively high emulation speed, and Bochs as another free, open-source emulator with debugging facilities and detailed CPU emulation, but slower speed.

The source says VirtualBox, QEMU, and Bochs were supplied on the book's accompanying CD in the vm folder.

*Historical note: the emulator comparisons and Windows/DOS workflow above reflect the book's 2011 environment. They are translated as source material rather than updated to current software capabilities.*

### **2.1.5. Summary**

This section introduced the fundamentals of protected-mode operation, the formation of linear addresses, and the algorithm for entering protected mode. In the practical portion, a program was written that switches the processor into protected mode. The next section examines interrupts in protected mode.

## 2.2. Interrupts in Protected Mode

After the basic introduction to protected mode, the source turns to interrupt handling, which it describes as one of the most important mechanisms in any operating system. Without interrupts, an operating system could not interact with external devices or protect its own data from applications.

### 2.2.1. What Is an Interrupt?

To protect operating-system data and receive information from external devices, the processor generates interrupts when protection rules are violated and when the interrupt controller reports signals from external or system devices.

When an interrupt occurs, the current program is interrupted, the address of the next instruction and the flags register are saved, and control is transferred to a handler. After the handler finishes, control returns to the interrupted program.

The source divides interrupts into three groups:

- Hardware interrupts.
- Software interrupts.
- Exceptions.

Hardware interrupts are generated by the interrupt controller. Software interrupts are generated by the `INT n` instruction. Exceptions are generated by the processor when protection rules are violated or when execution errors occur.

In real-address mode, the source describes 256 possible interrupts. At address 0000:0000 is a table of 256 four-byte addresses of the form `<dw segment>:<dw offset>`. Each address points to an interrupt handler, and the addresses themselves are called vectors.

In protected mode there are still 256 interrupt-vector numbers, but the table can be located anywhere in memory and is called the IDT (Interrupt Descriptor Table). Each entry is no longer merely a four-byte address; it is a gate descriptor.

A gate is a protected system object that allows programs at different privilege levels to use an interrupt. The usual case is described as a handler at privilege level 0 and a caller at a less-privileged level; the processor passes through the gate to the handler and later returns to the caller's level.

Three gate types are introduced: task gates, interrupt gates, and trap gates.

IF in the flags register controls maskable external interrupts. If IF is clear, maskable interrupts are ignored; if IF is set, they are serviced. They are called maskable because software can selectively or globally mask them.

### 2.2.2. Gate Descriptors

A gate descriptor is structurally similar to a segment descriptor. Table 2.1 gives the source's general bit layout. The descriptor privilege level determines which callers may invoke the gate.

| Bits  | Doubleword:bits | Purpose                    |
|-------|-----------------|----------------------------|
| 0-15  | 1:0-15          | Depends on type            |
| 16-31 | 1:16-31         | Selector (depends on type) |
| 32-36 | 2:0-4           | Reserved                   |
| 37-39 | 2:5-7           | Depends on type            |
| 40-42 | 2:8-10          | Gate type                  |
| 43    | 2:11            | Depends on type            |
| 44    | 2:12            | Always 0                   |
| 45-46 | 2:13-14         | Descriptor privilege level |
| 47    | 2:15            | Present bit                |
| 48-63 | 2:16-31         | Depends on type            |

With a task gate, an interrupt causes an automatic task switch. Bits 0-15 hold the selector of the TSS to which the processor will switch. The gate-type field contains 101b. Apart from P, the remaining fields are zero or reserved in the source's description. Task gates are discussed again in Section 2.4.

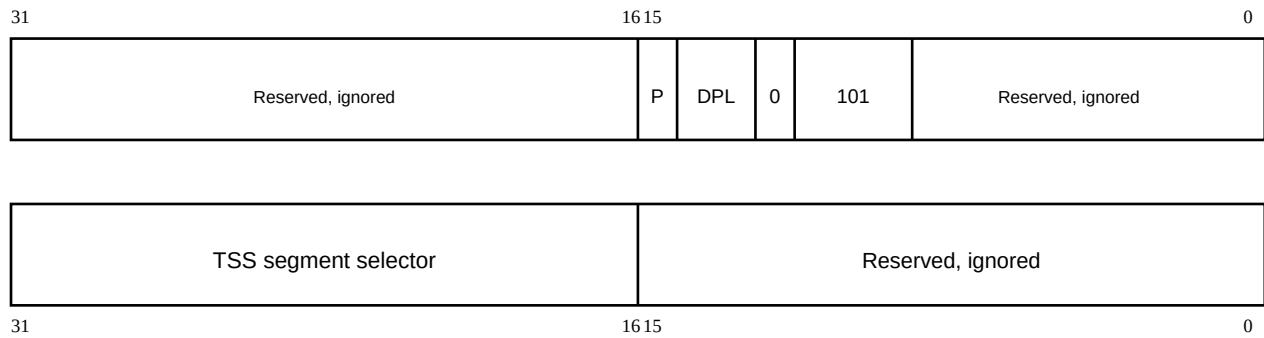


Figure 2.5. Task-gate descriptor format

Trap-gate and interrupt-gate descriptors are almost identical. Their gate-type fields differ: the source gives 111b for a trap gate and 110b for an interrupt gate. Bits 16-31 hold the code-segment selector containing the handler. The low and high 16 bits of the handler address occupy bits 0-15 and 48-63 respectively.

When an interrupt gate is entered, the processor automatically clears IF, preventing further maskable hardware interrupts during the handler. A trap gate leaves IF unchanged. Trap gates are used for debugging, where trap handling should remain transparent to external interrupts.

D is the gate-size bit: set for a 32-bit gate, clear for a 16-bit gate. The gate size determines the default stack width. Before invoking the handler, the processor saves CS, EIP, and EFLAGS, plus SS and ESP when privilege level changes, and for some exceptions a 16-bit error code. With a 32-bit gate, the source says 16-bit values are zero-extended to 32 bits.

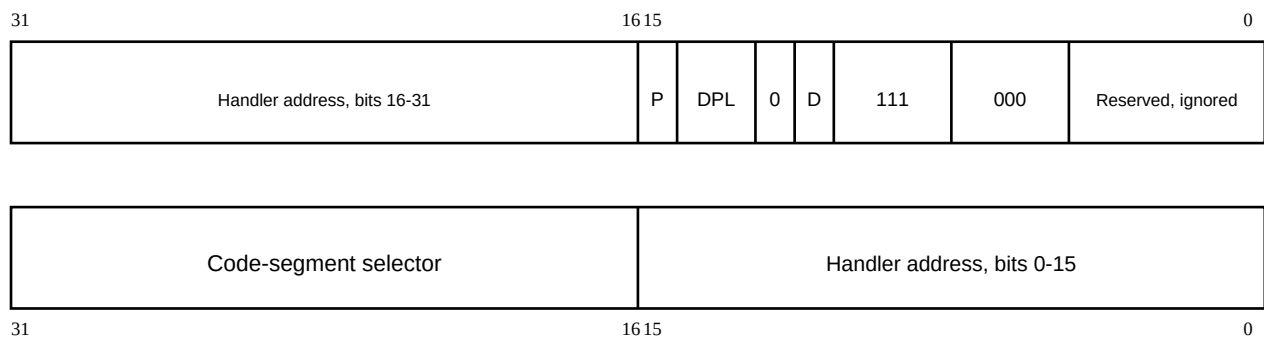


Figure 2.6. Trap-gate descriptor format

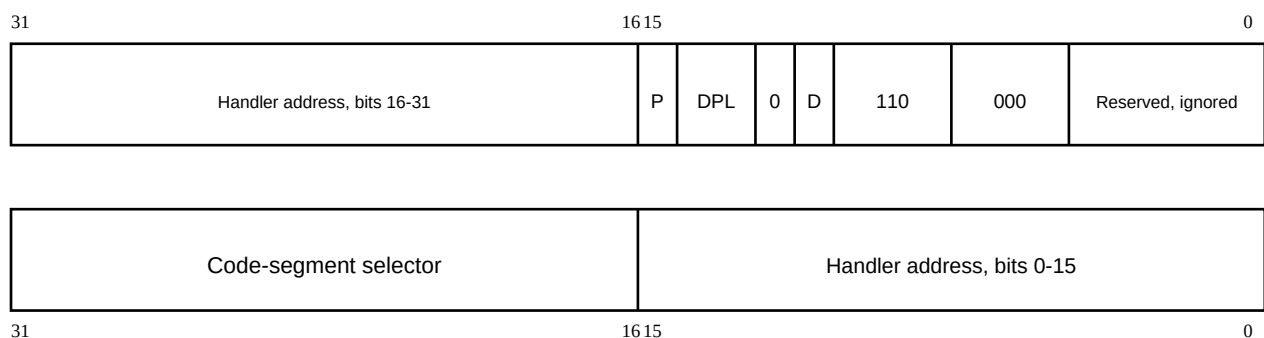


Figure 2.7. Interrupt-gate descriptor format

If P is clear, the source says the processor treats the handler as absent and generates a general-protection exception. It identifies vector 13 as the #GP handler. If a more specific exception occurs but its handler is unavailable, the source describes escalation through a not-present exception and ultimately a hang or reset if no usable handler remains.

DPL contains the maximum numerical privilege level from which software may explicitly invoke the gate. The source gives DPL = 1 as an example: programs at levels 0 and 1 may execute `int <vector>` for that gate, while levels 2 and 3 receive a general-protection exception.

All gate descriptors are stored in the IDT, which can contain from 0 to 256 descriptors. IDTR defines the base address and limit of the table, analogous to GDTR.

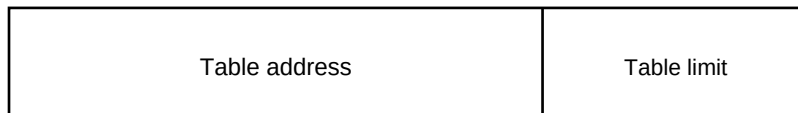


Figure 2.8. IDTR format

LIDT loads IDTR and is privileged; the source says it can be executed only at privilege level 0. SIDT stores IDTR and is not privileged in the source's description. In this protected-mode form, both instructions use a 6-byte memory operand.

### 2.2.3. Exceptions

Exceptions are generated by the processor when software violates protection rules or when execution errors occur. Applications cannot mask processor exceptions and that they are generated regardless of IF.

The source divides exceptions into three classes:

**1. Fault.** Generated by an erroneous operation that can potentially be corrected. The handler may repair the condition and restart the faulting instruction; therefore the saved return address points to the instruction that caused the exception. A not-present page is given as an example.

**2. Trap.** Generated immediately after the trapped instruction has executed. Execution can continue at the following instruction. The source associates traps with debugging.

**3. Abort.** Does not permit normal continuation of the interrupted program and indicates a serious integrity problem. The double-fault exception is given as an example.

For some exceptions, the processor pushes a two-byte error code so the handler can identify the source of the failure. The source lists 32 reserved exception vectors, 0 through 1Fh, and says they must not be reused for unrelated interrupts.

Table 2.2. Exception types in protected mode

| Vector | Name | Description                    | Class         | Error code  | Source                                      |
|--------|------|--------------------------------|---------------|-------------|---------------------------------------------|
| 0      | #DE  | Divide error                   | Fault         | No          | DIV or IDIV                                 |
| 1      | #DB  | Debug                          | Fault or trap | No          | Any instruction or INT 1                    |
| 2      | -    | NMI interrupt                  | Interrupt     |             | Non-maskable external interrupt             |
| 3      | #BP  | Breakpoint                     | Trap          | No          | INT3                                        |
| 4      | #OF  | Overflow                       | Trap          | No          | INTO                                        |
| 5      | #BR  | BOUND range exceeded           | Fault         | No          | BOUND                                       |
| 6      | #UD  | Invalid opcode                 | Fault         | No          | Unsupported instruction or UD2              |
| 7      | #NM  | Device not available           | Fault         | No          | Floating-point instruction or WAIT/FWAIT    |
| 8      | #DF  | Double fault                   | Abort         | Always zero | Any instruction                             |
| 9      | -    | Reserved by Intel - do not use |               |             |                                             |
| 10     | #TS  | Invalid TSS                    | Fault         | Yes         | Task switch or TSS access                   |
| 11     | #NP  | Segment not present            | Fault         | Yes         | Loading segment registers or segment access |
| 12     | #SS  | Stack-segment fault            | Fault         | Yes         | Stack operations and loading SS             |
| 13     | #GP  | General protection             | Fault         | Yes         | Memory access and other protection checks   |
| 14     | #PF  | Page fault                     | Fault         | Yes         | Memory access                               |
| 15     | -    | Reserved by Intel - do not use |               |             |                                             |
| 16     | #MF  | x87 FPU floating-point error   | Fault         | No          | x87 FPU instruction or WAIT/FWAIT           |
| 17     | #AC  | Alignment check                | Fault         | Always zero | Memory access                               |
| 18     | #MC  | Machine check                  | Abort         | No          | Codes/content depend on model               |
| 19     | #XF  | SIMD floating-point exception  | Fault         | No          | SSE instructions                            |
| 20-31  | -    | Reserved by Intel - do not use |               |             |                                             |

During early operating-system development, exception handlers do not necessarily have to implement full recovery. The source suggests simple stubs that display the exception number and stack parameters and then stop the processor.

IF does not affect exceptions. The source also describes vector 2 as the only non-maskable interrupt and, in this paragraph, says that bit 7 of port 0A2h masks NMI when set.

*Editorial note: later in the same section, and in the practical code, NMI masking is performed through bit 7 of port 70h. The earlier printed 0A2h value is preserved here as a source inconsistency rather than silently corrected.*

A double fault occurs when another exception is generated while attempting to invoke an exception handler. If another exception occurs while attempting to invoke the double-fault handler, the source says the processor shuts down or resets.

### 2.2.4. Error Codes

Error codes are pushed for #TS, #NP, #SS, #GP, and #PF, except where an exception's code is defined as always zero. Often the error code identifies the selector of the segment involved in the failure. In this context, the low RPL bits are repurposed.

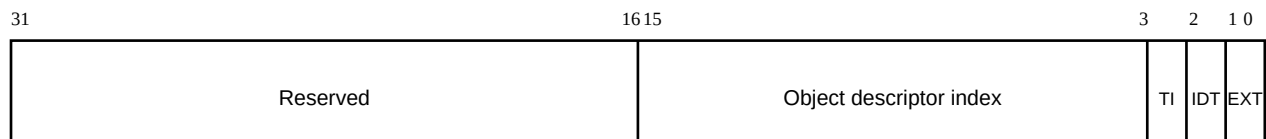


Figure 2.9. Selector format in an error code

**EXT.** If clear, the exception was caused by an internal source; if set, by an external source.

**IDT.** The source text says that when this bit is clear, the selector identifies a descriptor in the GDT or IDT; when set, it identifies a descriptor in the IDT.

*Editorial note: the immediately following TI description distinguishes GDT from LDT when IDT is clear, so the preceding printed 'GDT or IDT' wording appears inconsistent. This translation records both statements rather than silently reconciling them.*

**TI.** If clear, the selector refers to a descriptor in the GDT; if set, to a descriptor in the LDT. If IDT is set, TI has no meaning.

The page-fault error-code format is deferred to Section 2.3.

### 2.2.5. Software Interrupts

Interrupts can be generated by software. INT generates the interrupt whose operand is the IDT entry number, for example INT 25. INT3 invokes vector 3 using a short opcode. INTO invokes vector 4 when OF is set.

The source calls software interrupts the simplest way to invoke privileged code and states that systems before Windows XP used this method. It then says Pentium II and later processors provide a more protected method for entering privileged code.

*Editorial note: the source says this is a method of invoking privileged code for programs 'working at privilege level zero'. That statement is preserved in substance, although the surrounding gate discussion concerns less-privileged callers entering privileged code.*

### 2.2.6. Hardware Interrupts

An interrupt controller acts as an intermediary between devices and the processor, prioritizing requests and resolving simultaneous signals. The source introduces the legacy programmable interrupt controller (PIC) and notes that modern systems use APIC, which is covered in Chapter 6.

The legacy PC arrangement uses two cascaded i8259A controllers, with the slave controller connected to the master's second interrupt line. Together they provide 16 IRQ inputs. For compatibility, later systems can still expose legacy PIC behavior.

The source limits this section to basic legacy PIC programming.

When a device requests service, the PIC requests an interrupt from the processor. If IF is set, the processor responds, and the controller places the IRQ number plus a configurable base value on the data bus.

The source gives the historical real-mode/DOS bases as 8 for the master controller and 70h for the slave. In these protected-mode examples, the master is remapped to 20h and the slave to 28h, giving vectors 20h-2Fh.

When requests arrive simultaneously, the source says they are handled beginning with the lower IRQ number. Individual IRQ inputs can also be masked.

Two controller registers are introduced: iMr and iSr. Each of the two PICs has its own copy.

**iMr** is the interrupt-mask register. Each bit masks one interrupt line when set. The master iMr is connected to I/O port 21h and the slave iMr to 0A1h. Masking the master's second line also masks all interrupts arriving through the slave.

**ISr** indicates the interrupt currently in service. The source says bit n is set while interrupt n is being serviced and that an in-service bit blocks inputs with numbers greater than or equal to n. Software must notify the controller when handling is complete, normally by writing the nonspecific EQI value 20h.

The master iSr is at port 020h and the slave iSr at 0A0h. The following schematic initialization sequence:

### Source's generic PIC initialization form

```
out iSr 00010001b
out iMr, <number added before placing the vector on the data bus>
; must be a multiple of 8
out iMr, 00000100b ; depends on the controller
```

```
out iMr, 00000001b
```

The source then gives a simplified master-controller sequence:

#### Master PIC initialization shown by the source

```
out 20h, 00010001b
out 21h, 20h ; handler number for line 0
out 21h, 00000100b
out 21h, 00000001b
```

The third value is the bit mask indicating the master line to which the slave is connected.

The slave-controller sequence is:

#### Slave PIC initialization shown by the source

```
out A0h, 00010001b
out A1h, 28h ; handler number for line 0
out A1h, 00000010b
out A1h, 00000001b
```

Here the third value identifies the slave's cascade identity. After initialization, all 16 legacy IRQ lines map to vectors 20h-2Fh.

IF controls maskable external interrupts. If IF is clear, the processor does not respond to ordinary PIC requests.

Finally, the source describes NMI (vector 2) as an interrupt used to report hardware failure. On NMI, the processor does not ask the PIC for a vector number; it directly invokes vector 2. NMI can be masked by setting bit 7 of port 70h:

#### NMI masking sequence

```
in al, 70h
or al, 80h
out 70h, al
```

### 2.2.7. Interrupt Handler

When an interrupt or exception is generated, the source describes the following sequence:

1. The IDT base address is obtained from IDTR.
2. The interrupt number selects an interrupt-gate descriptor in the IDT. If its P bit is clear, a general-protection exception is generated.
3. If the current privilege level differs from the handler's privilege level, the processor switches stacks and saves the interrupted task's SS and ESP on the handler stack.
4. EFLAGS, CS, and EIP are pushed. Some exceptions also push an error code, which the exception handler must remove before returning.
5. TF is cleared in the handler's flags. For a software interrupt or exception, the source says VM, RF, and NT are also cleared.
6. Entry through an interrupt gate clears IF, blocking further maskable hardware interrupts. Entry through a trap gate does not.

After handling, the handler removes any error code and executes IRETD. IRETD restores the saved flags. If privilege level changes back, the saved SS and ESP are used to restore the interrupted task's stack. The source summarizes IRETD as performing the reverse of interrupt entry.

Hardware interrupt handlers also have to notify the interrupt controller that servicing is complete. The following i8259A skeleton:

#### Listing 2.5. Hardware-interrupt handler template

```
Handler:
... ..
... ..
push ax
mov al, 20h
out 020h, al ; reset controllers
out 0a0h, al
pop ax
iretd
```

Resetting/notifying the controllers is also called an EOI (End Of Interrupt) operation.

The source next discusses RF, the resume flag. It describes RF as controlling restart of the instruction that caused an exception. When set, the current instruction - described as the instruction preceding the one named by EIP - is executed again, and #DB is suppressed.

The processor clears RF after every instruction except, in the source's description, after an IRET that set RF and after JMP, CALL, or INT n when the destination was a task gate.

For fault-type exceptions, the saved flags image is described as having RF = 1 so that the faulting instruction can be retried. The source also says an interrupt saves RF = 1 when the interrupted instruction is a string instruction: MOVS, LODS, CMPS, SCAS, STOS, or the printed name OUDS.

*Editorial note: OUDS is printed in the source. It is likely intended to be OUTS, but it is preserved as a noted source typo.*

For trap-type exceptions, instruction restart is not implied, so RF = 0 is saved. To restart manually, software can set RF in the flags image on the handler stack. The source gives the stack order after an error code as EIP, CS, then EFLAGS.

## 2.2.8. Practice

The practical example extends the protected-mode framework from Section 2.1. The program responds to keyboard input and system-timer interrupts. Only PM\_CODE.ASM needs to be changed.

### Listing 2.6. Macro for declaring a gate descriptor

```
macro DEFINE_GATE _address, _code_selektor, _type
{
    dw _address and 0FFFFh, _code_selektor, _type, _address shr 16
}
```

The third macro parameter is the descriptor type. The source defines the interrupt-gate value as:

### Interrupt-gate constant

```
INT_GATE equ 1000111000000000b
```

### Listing 2.7. Declaring the IDT with DEFINE\_GATE

```
align 8
IDT:
dq 0 ; 0
DEFINE_GATE syscall_handler, CODE_SELECTOR, INT_GATE
dq 0 ; 2
dq 0 ; 3 ...
dq 0 ; 12
DEFINE_GATE exGP_handler, CODE_SELECTOR, INT_GATE ; 13 #GP
dq 0 ; 14
...
dq 0 ; 31
DEFINE_GATE irq0_handler, CODE_SELECTOR, INT_GATE ; 20h (IRQ 0)
DEFINE_GATE irq1_handler, CODE_SELECTOR, INT_GATE ; 21h (IRQ 1)ss
DEFINE_GATE int_EOI, CODE_SELECTOR, INT_GATE ; 22h (IRQ 2)
...
DEFINE_GATE int_EOI, CODE_SELECTOR, INT_GATE ; 2Fh (IRQ 15)
IDT_END:
```

*The trailing ss after the IRQ 1 comment is present in the printed source and is preserved as a source artifact.*

The IDT size is calculated as:

### IDT size

```
IDT_size equ IDT_END-IDT
```

IDTR is loaded with:

### Loading IDTR

```
lidt fword [IDTR]
```

The PICs are then remapped so that their IRQs use vectors 20h-2Fh. The source places the master and slave base values in BL and BH respectively.

### Listing 2.8. Remapping the interrupt-controller vectors

```
mov bx, 2820h
mov al, 00010001b
out 020h, al
out 0A0h, al
mov al, bl
out 021h, al
mov al, bh
out 0A1h, al
mov al, 00000100b
out 021h, al
mov al, 2
out 0A1h, al
mov al, 00000001b
out 021h, al
out 0A1h, al
```

All hardware interrupts are then enabled by setting IF and unmasking NMI at port 70h:

### Listing 2.9. Enabling hardware interrupts

```
in al, 70h
and al, 7Fh
out 70h, al
sti
```

A default empty handler is installed for all remapped hardware interrupts except the first two, timer and keyboard:

### Listing 2.10. Empty hardware-interrupt handler

```
int_EOI:
push ax
mov al, 20h
out 020h, al
```

```

out 0a0h, al
pop ax
iretd

```

This handler sends an end-of-interrupt indication to both controllers.

#### Listing 2.11. Timer interrupt handler

```

irq0_handler:
push eax
push edx
push ebx
xor edx, edx
inc dword [counter]
mov eax, dword [counter]
mov ebx, 18;
div ebx
cmp edx, 0
jnz .cont
inc byte [es:6]
cmp byte [es:6], ":"
jnz .cont
mov byte [es:6], "0"
inc byte [es:4]
cmp byte [es:4], ":"
jnz .cont
mov byte [es:4], "0"
inc byte [es:2]
cmp byte [es:2], ":"
jnz .cont
mov byte [es:2], "0"
inc byte [es:0]
cmp byte [es:0], ":"
jnz .cont
mov byte [es:0], "0"
mov byte [es:2], "0"
mov byte [es:4], "0"
mov byte [es:6], "0"
.cont:
pop ebx
pop edx
pop eax
jmp int_EOI

```

*The dangling semicolon in mov ebx, 18; is present in the source listing and is preserved.*

The default timer generates about 18 interrupts per second. A counter is incremented on every timer interrupt; whenever it is divisible by 18, the displayed seconds value is incremented. The display allows values up to 9999 seconds, approximately 2.7 hours.

Each decimal digit is advanced and compared with ':' because ':' immediately follows '9' in ASCII. On reaching ':', that digit is reset to '0' and the next higher digit is incremented. The handler then jumps to int\_EOI, which notifies the controllers.

The keyboard handler is omitted from the book because implementing a keyboard driver is outside this section's scope; the complete example was supplied separately.

#### Listing 2.12. Program code after interrupts are enabled

```

mov eax, 00F00h
mov ecx, 0820h
mov edi, 00
rep stosw
mov byte [es:0], "0"
mov byte [es:2], "0"
mov byte [es:4], "0"
mov byte [es:6], "0"
mov dword [cursor], 80
mov esi, message1
int 1
mov dword [cursor], 160
mov esi, message2
int 1
mov dword [cursor], 240
jmp $

```

The screen is cleared by filling video memory with 0F00h so that characters use the 0Fh attribute, which the source describes as white text on a black background. Four zero digits are placed in the upper-left corner. Two strings are then printed with the program's INT 1 service.

#### Listing 2.13. Handler code on interrupt vector 1

```

syscall_handler:
pushad
_puts:
lodsb
mov edi, dword [cursor]
mov [es:edi*2], al
inc dword [cursor]
test al, al

```

```
jnz _puts  
popad  
iretd
```

Listing 2.13 writes a zero-terminated string to the text video buffer selected by ES. It does not process carriage-return or newline characters. The variable `cursor` holds the current screen position.

The source acknowledges that using vector 1, which Intel reserves for debugging, is not a good design choice. In this example the debug interrupt is unused, and the author considers this placement more convenient than putting the text-output service after the hardware-interrupt vectors.

The complete source code was supplied in the `part2` folder on the book's accompanying CD.

### **2.2.9. Summary**

This section covered interrupt and exception handling in protected mode and reinforced the material with a program that works with interrupts. Section 2.3 turns to what the source calls the second most important operating-system mechanism: translation of virtual addresses into physical addresses.

## 2.3. Address Translation

One of the principal innovations of protected mode is translation of virtual addresses into physical addresses, also called page translation. The operating system can use paging both to protect its own data from user programs and to give each program the impression of having an independent address space and, in the source's simplified description, the full 4 GB address range even when the computer does not contain that much physical memory.

With paging enabled, a virtual address need not equal the physical address actually used. Address conversion is performed by the processor in hardware. The source also introduces Physical Address Extension (PAE), which in this historical 32-bit context permits addressing up to 64 GB of physical memory.

### 2.3.1. What Is It?

The source first returns to logical addressing. Every memory access conceptually uses a selector and an offset. The selector identifies a segment, usually through a segment register, and the offset is supplied by a register, constant, or variable expression. The resulting selector:offset pair is a logical address.

The processor converts the logical address into a 32-bit absolute linear address. We already encountered linear addresses when defining descriptor-table bases and the IDTR/GDTR base addresses.

The linear address is then converted into the physical address used for the actual memory transaction. In Sections 2.1 and 2.2 paging was disabled, so linear and physical addresses were identical. With address translation enabled, the source calls the linear address a virtual address because it is translated to a physical one.

The minimum allocation unit discussed here is the page: a contiguous fixed-size region of memory. Intel386 introduced 4 KB pages; Pentium/Pentium Pro later allowed 4 MB pages in ordinary paging. In PAE mode the page sizes considered are 4 KB and 2 MB.

A 4 KB page has an address aligned to 1000h bytes, so its lowest 12 address bits are zero. The entire 32-bit address space contains  $2^{20}$  such pages (1,048,576).

Each page has a virtual address and a physical address. Different virtual pages may map to the same physical page.

The source gives an example in which a program accesses address 8D138Ch and then states that the virtual page begins at 01000h, while a possible physical page begins at 7F4000h, producing physical address 7F438Ch.

*Editorial note: for the virtual address 8D138Ch, a 4 KB-aligned virtual page base would ordinarily retain the upper address bits rather than become 01000h. The printed 01000h value is preserved here as a source error.*

A page can be marked absent from physical memory. If physical memory is scarce, the operating system can write a page's contents to external storage and mark the page not present. Accessing it then generates #PF; the handler can bring the contents back into a free physical page, possibly evicting another page, and then restart the interrupted operation.

Paging is enabled by the PG bit, the most-significant bit of CR0. When PG = 0, linear addresses are physical addresses; when PG = 1, page translation is enabled. PAE is selected by bit 5 of CR4.

### 2.3.2. Ordinary Address Translation

#### 2.3.2.1. 4 KB Pages

Each 4 KB page is described by a 4-byte page-table entry (PTE). A page table is 4 KB and therefore contains 1024 entries. One page table can describe 1024 pages, or  $2^{22}$  bytes (4 MB). A 10-bit value selects an entry.

Describing the full 4 GB address space requires 1024 page tables. Each page table is itself described by a 4-byte page-directory entry (PDE), and the 1024 PDEs form the page directory. A 10-bit value selects a PDE.

Every 32-bit virtual address is divided into 10-10-12 bits. The first 10 bits index the page directory, the next 10 index the selected page table, and the low 12 bits are the offset within the physical page.

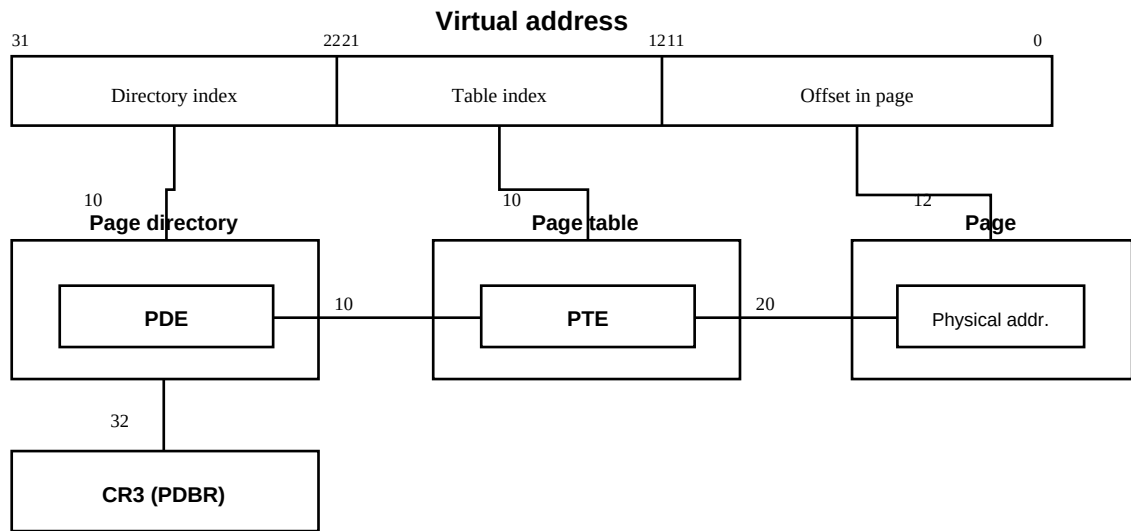


Figure 2.10. Translation of a virtual address to a physical address in ordinary 4 KB paging

The source next defines the formats of a PTE and a PDE.

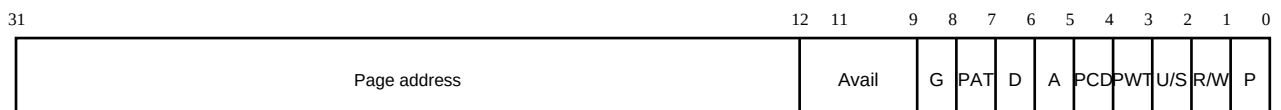


Figure 2.11. Page-table entry format

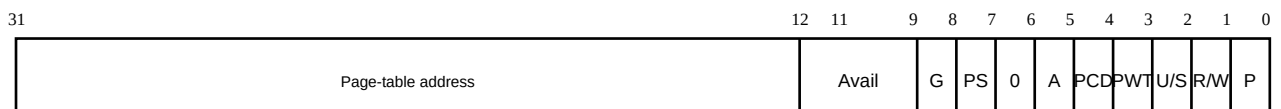


Figure 2.12. Page-directory entry format for 4 KB pages

Bits 12-31 of the PTE contain the physical base address of the page; because the page is 4 KB aligned, the low 12 address bits are zero. In the PDE, bits 12-31 contain the physical address of the page table.

**Bits 9-11 (Avail).** The processor does not use these bits; the operating system may use them for its own purposes.

**P (Present).** If clear, the page is not mapped to physical memory, either because it is undefined or because its contents have been moved to external storage. Access to a not-present page generates #PF. In a PDE, P = 0 means the referenced page table is not present; the processor ignores the remaining fields when P = 0.

**R/W.** When set, the page may be written; when clear, it is read-only. For a page to be writable, R/W must permit writing in both the relevant PDE and PTE.

**U/S.** When set, the page or group of pages may be used by code at all privilege levels. When clear, level-3 code may not access it. User access requires U/S to allow access in both the PDE and PTE.

**PWT and PCD.** These control page caching. PCD disables caching. PWT controls write-through behavior. The source does not explore caching further here.

**A (Accessed).** The processor sets A when the page has been accessed since software last cleared the bit. An operating system can use it to estimate whether a page has been recently used. In a PDE it is set when a page reachable through that page table is accessed.

**D (Dirty).** Set when a write occurs to the page.

**PS.** In a PDE, this selects page size. The source says PS = 1 indicates a 4 MB page and PS = 0 a page-table pointer for 4 KB pages.

**PAT and G.** PAT participates with PCD and PWT in memory-type selection on processors supporting the Page Attribute Table. G marks a page global so it remains valid in TLB caches across CR3 reloads or task switches. The source says G is ignored in a PDE in the case being discussed.

### 2.3.2.2. 4 MB Pages

With 4 MB pages, a separate page table is not required. The page directory itself maps large pages. A virtual address is divided 10-22: the upper 10 bits select a PDE and the low 22 bits are the offset within the 4 MB physical page.

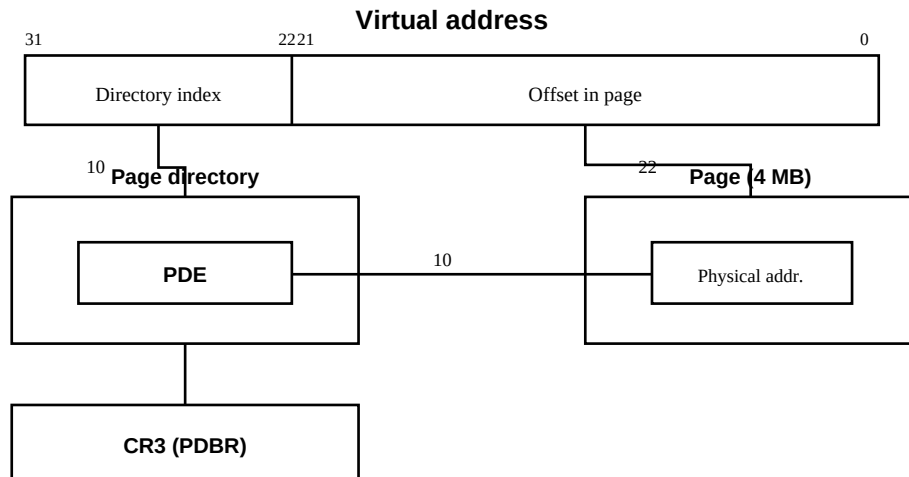


Figure 2.13. Linear-to-physical translation using 4 MB pages

The page directory effectively acts as the page table because each PDE points directly to a page.

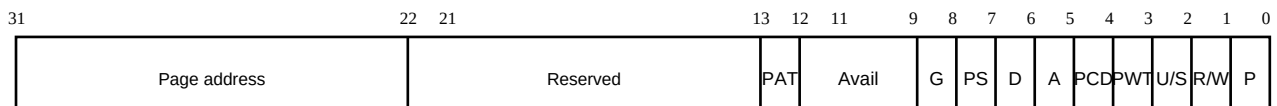


Figure 2.14. Page-directory entry format for a 4 MB page

The low 12 control bits otherwise have the same general form as those used with 4 KB paging. To permit 4 MB pages, PSE (bit 4 of CR4) must be set. During translation, PS in the PDE determines whether the entry is treated as a 4 MB page or as a pointer to a page table.

### 2.3.2.3. CR3 Register

CR3 is also called PDBR (Page Directory Base Register). Since the processor can obtain every lower-level paging structure from the page directory, it needs only the physical base address of that directory.

CR3 contains the page-directory physical address aligned to a 4 KB boundary, so the low 12 address bits do not form part of the base address.

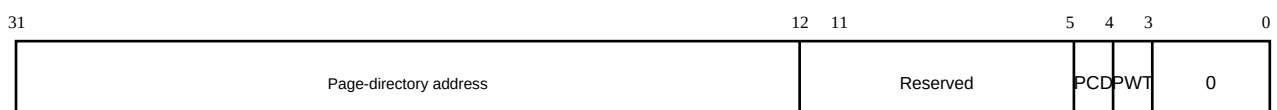


Figure 2.15. CR3 format in ordinary paging

PWT and PCD control write-through and caching. The source leaves them clear in the examples. CR3 is accessed with MOV and a general-purpose register:

#### Loading CR3

```

mov eax, PDE_BASE
mov cr3, eax

```

### Reading the current page-directory address

```

mov eax, cr3

```

The source reminds the reader that if CR3 control flags are required, they must be included when reloading CR3; it describes this as loading the page-directory address plus 8, 16, or 24 depending on the required low-bit flags.

### 2.3.3. Physical Address Extension Mode

Physical Address Extension (PAE) expands physical addresses to 36 bits in the source's 32-bit model, allowing up to 64 GB of physical memory while each individual 32-bit application still has at most a 4 GB virtual address space.

PAE is enabled by setting bit 5 of CR4:

#### Enabling the PAE control bit

```

mov eax, cr4
or eax, 0100000b
mov cr4, eax

```

In PAE mode the source considers only 4 KB and 2 MB pages. It says PSE in CR4 is ignored while PAE is enabled. PAE also permits an execute-disable protection mechanism on processors that support it.

#### 2.3.3.1. 4 KB Pages

PAE still uses page tables, but each entry is 8 bytes. A 4 KB page table therefore contains 512 entries and maps  $2^{21}$  bytes. Page directories also contain 512 eight-byte entries; one directory maps  $2^{30}$  bytes (1 GB). Four page directories are required to cover a 4 GB virtual address space.

The four page directories are themselves described by four 8-byte entries held in a page-directory-pointer table.

A 32-bit virtual address is divided 2-9-9-12 bits: a 2-bit page-directory number selects a PDPTE, the next 9 bits select a PDE, the next 9 bits select a PTE, and the low 12 bits form the page offset.

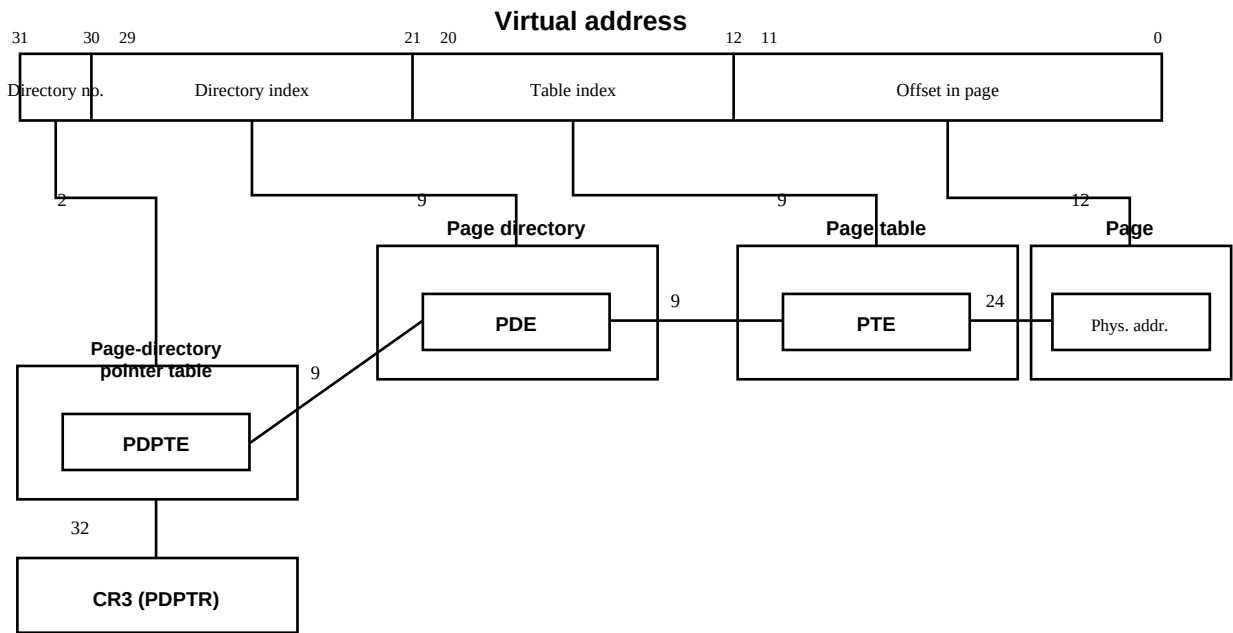


Figure 2.16. Address translation in PAE mode using 4 KB pages

The number of addressable virtual pages is unchanged from ordinary 32-bit paging, while the paging entries are widened to 8 bytes so that 36-bit physical addresses can be stored.

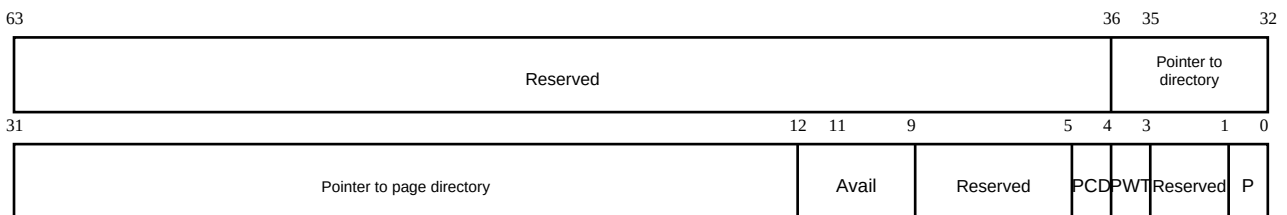


Figure 2.17. Page-directory-pointer-table entry format

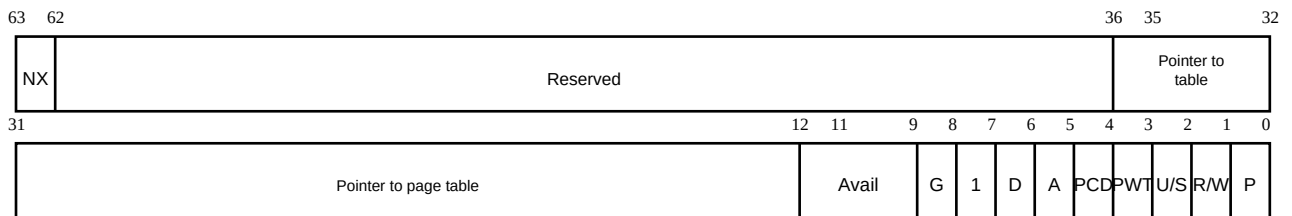


Figure 2.18. PAE page-directory entry format for 4 KB pages

Editorial note: Figure 2.18 prints bit 7 as '1', while the surrounding prose says bit 7 distinguishes a page-table pointer from a 2 MB page and later states that a set bit selects the 2 MB page form. The diagram is reproduced as printed and the conflict is noted here.

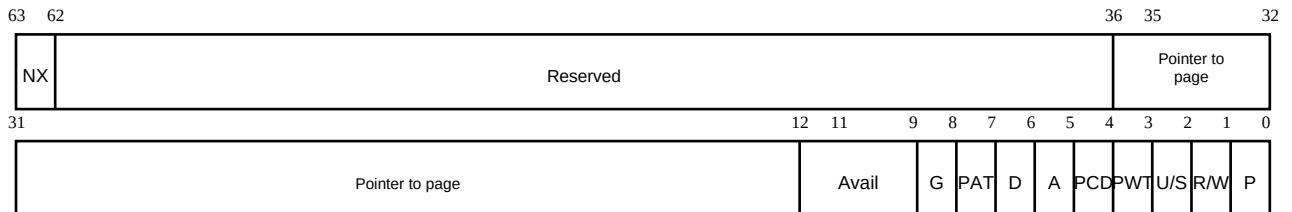


Figure 2.19. PAE page-table entry format

The meanings of corresponding fields are the same as in ordinary paging. The new field emphasized here is NX, discussed in Section 2.3.3.4.

### 2.3.3.2. 2 MB Pages

No additional processor mode is required beyond PAE to use 2 MB pages. The PDPTE format remains the same, but the PDE format changes.

When bit 7 is clear the PDE is interpreted as a page-table pointer, while when bit 7 is set the low 21 bits of the virtual address become the offset within a 2 MB page.

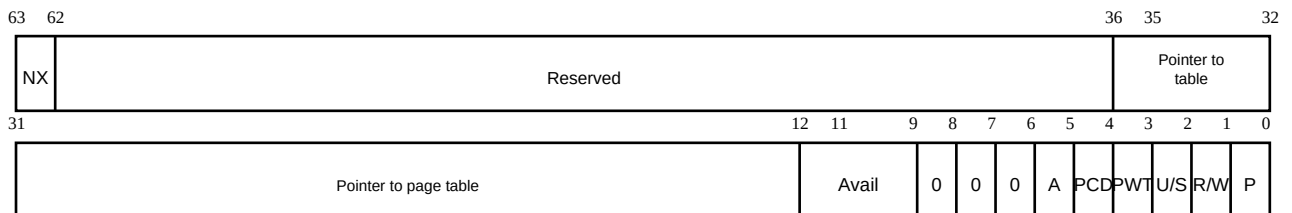


Figure 2.20. PAE page-directory entry format shown by the source for 2 MB pages

Editorial note: Figure 2.20 prints zeros in bits 8, 7, and 6 even though the immediately preceding prose says bit 7 must be set for a 2 MB page. The English redraw preserves the printed bit values.

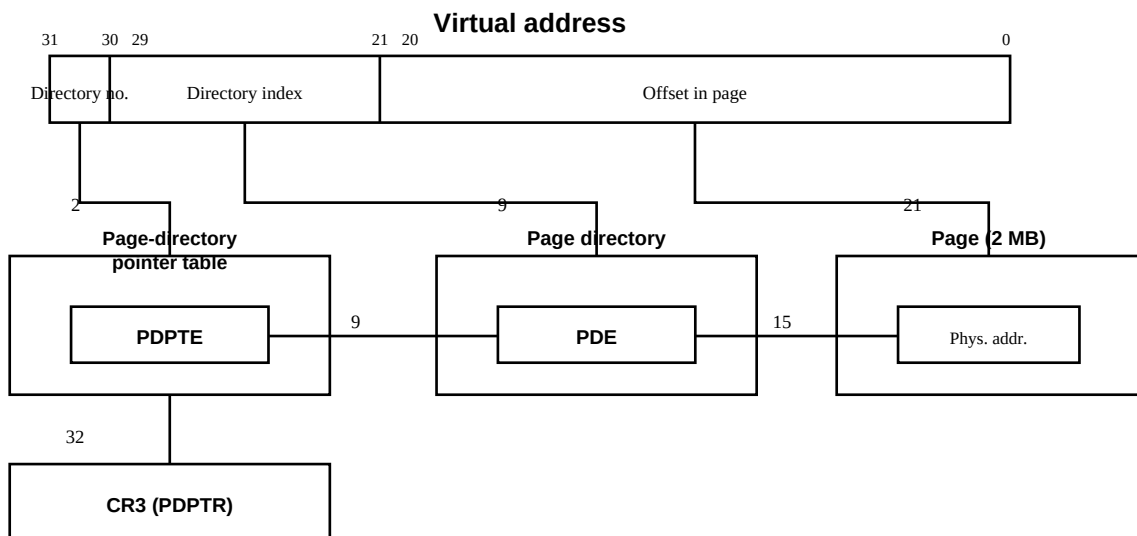


Figure 2.21. Address translation in PAE mode using 2 MB pages

### 2.3.3.3. CR3 Register

In PAE mode CR3 contains the 32-bit address of the page-directory-pointer table. The table is aligned on a 32-byte boundary, so the low 5 address bits do not contribute to the base address. Unlike the ordinary paging structures discussed earlier, the PDPT does not have to begin at a 4 KB boundary, although the source says it cannot be placed above 4 GB.

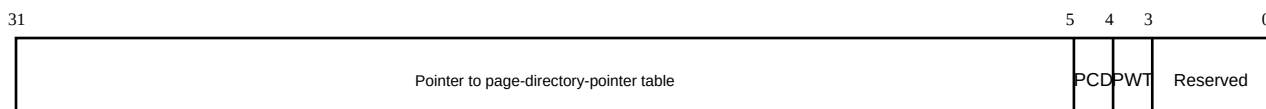


Figure 2.22. CR3 format in PAE mode

### 2.3.3.4. NX Bit

If NX is set for a page, code execution on that page is prohibited; control must not be transferred to code located there. NX must be clear in the relevant paging structures for code to execute.

NX support is checked with CPUID using EAX = 80000001h. If bit 20 of EDX is set, the source says the feature is supported.

Setting NX in page structures is not sufficient by itself. NXE, bit 11 of the IA32\_EFER MSR, must also be set. If the processor does not support NX, attempting to use paging structures with NX set causes a general-protection exception according to the source.

### 2.3.4. Page-Fault Handler

The page-fault handler is located at IDT vector 14. #PF is a fault-class exception, so after the condition is repaired the operation can normally be restarted. The processor pushes an error code and places the virtual address responsible for the page fault in CR2.

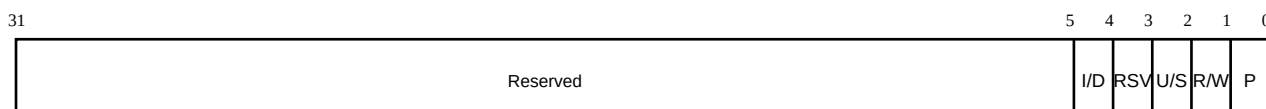


Figure 2.23. Page-fault error-code format

**P.** Clear means the fault was caused by a not-present page; set means a protection violation.

**R/W.** Set means the fault occurred during a write; clear means during a read.

**U/S.** Set means the fault was generated by user code; clear means code at privilege level 0, 1, or 2.

**RSV.** Set means the processor found a reserved bit set in a paging structure. The source says this form occurs only when PSE or PAE is enabled.

**IID.** Relevant only when execute-disable protection is enabled. Set means the fault occurred during an attempt to transfer control to a page where code execution is prohibited.

A page-fault handler commonly creates a required paging entry, finds an unused physical page, maps the virtual page to it, and possibly loads page contents from external storage. If the fault is a protection violation, the operating system may instead terminate the program.

After the handler finishes, instruction restart is expected, but in this paragraph it describes the saved return address as the instruction immediately following the faulting instruction.

*Editorial note: Section 2.2 correctly described fault-class exceptions as saving the address of the faulting instruction so it can be restarted. The 'following instruction' wording here conflicts with that earlier description and is preserved as a noted source error.*

A practical difficulty is determining which physical pages are free. The source proposes a bitmap in which each bit represents one physical page. For 1 MB of memory, 32 bytes are required; for 512 MB, 16 KB; for 4 GB, 128 KB; and for 64 GB, 2 MB.

A large bitmap takes time to scan, though algorithms can be designed to find a zero bit efficiently.

For performance, the source recommends making page-fault handlers as fast and compact as possible and suggests implementing them completely in assembly. It also recommends saving CR2 immediately on entry, because the handler itself might trigger another page fault and overwrite the original CR2 value.

### 2.3.5. WP Flag in CR0

By default in the source's description, privileged code at levels 0, 1, and 2 can write to pages marked read-only. Write protection for privileged code is enabled by setting CR0.WP, bit 16.

When WP is set, privileged code can modify data only on pages whose paging entries allow writes. R/W must permit writing in every paging entry that describes the page, regardless of U/S.

### 2.3.6. Practice

The practical examples implement 4 KB paging in both ordinary and PAE modes. To demonstrate virtual mapping, the source creates three virtual pages at different addresses and maps them all to the same physical address, 0B8000h, the text video buffer. Text is then written through each virtual address.

All page tables are placed together in one memory area. In ordinary paging, the address of the required page table is calculated as:

#### Ordinary-mode page-table formula

$$pta = pdi * 4096 + ptba$$

where pta is the page-table address, pdi the page-directory index, and ptba the base of the memory region containing all page tables.

In PAE mode, the page-directory address is calculated as:

#### PAE page-directory formula

$$pda = pdti * 4096 + pdba$$

where pda is the page-directory address, pdti the index in the page-directory-pointer table, and pdba the base of the memory region containing all page directories.

The required page-table address is then given by the source as:

#### PAE page-table formula

$$pta = pdti * pdi * 4096 + ptba$$

#### 2.3.6.1. Ordinary Mode

The ordinary-mode example needs one page directory plus 1024 page tables. The source allocates 4 MB for the page tables and one additional 4 KB page for the directory. The protected-mode startup framework already exists; only PM\_CODE.ASM needs to be changed.

The central function accepts a virtual address in EAX and a physical address in EBX.

#### Listing 2.14a. Creating a virtual page

```
create_PDEPTE:
; in
; EAX page address
; EBX phys page address
pushad
mov edi, ebx
mov edx, eax
shr eax, 22
shl eax, 2
mov esi, PAGE_DIR_BASE_ADDRESS
add esi, eax
shr eax, 2
```

The function first saves the registers and copies the physical and virtual addresses to EDI and EDX. It extracts the page-directory index by shifting the virtual address right 22 bits, multiplies that index by four, and adds it to PAGE\_DIR\_BASE\_ADDRESS so ESI points to the required PDE.

#### Listing 2.14b. Creating a virtual page (continued)

```
mov ebx, eax
shl ebx, 12
mov eax, PAGE_TABLES_BASE_ADDRESS
add eax, ebx
or eax, 011b
mov [esi], eax
and eax, 0FFFFFF00h
```

The directory index is multiplied by 4096 and added to PAGE\_TABLES\_BASE\_ADDRESS to obtain the page-table address. The Present and Write flags are set and the resulting value is stored in the PDE. The flags are then cleared from EAX so it again contains the page-table base.

#### Listing 2.14c. Creating a virtual page (continued)

```
.create_PTE: ;eax = table base address
mov esi, eax
mov eax, edx
shl eax, 10
shr eax, 22
shl eax, 2
add esi, eax
```

The saved virtual address is used to derive the page-table index, multiply it by four, and add it to the page-table base. ESI therefore points to the required PTE.

#### Listing 2.14d. Creating a virtual page (continued)

```
mov eax, edi
or eax, 011b
mov [esi], eax
.end:
popad
ret
```

The saved physical address is placed in EAX, Present and Write are set, and the value is stored in the PTE.

#### Listing 2.15. Using create\_PDEPTE

```
START_CODE:
mov eax, START_CODE
mov ebx, eax
call create_PDEPTE
mov eax, 0B8000h
mov ebx, eax
call create_PDEPTE
mov eax, 0FF000000h
mov ebx, 0B8000h
call create_PDEPTE
mov eax, 0EE000000h
mov ebx, 0B8000h
call create_PDEPTE
```

Code pages must be mapped as well as data pages because executable code also resides in memory.

Before paging is enabled, CR3 must contain the page-directory address. Paging is then enabled by setting CR0.PG:

#### Listing 2.16. Enabling page translation

```
mov eax, PAGE_DIR_BASE_ADDRESS
mov cr3, eax
mov eax, cr0
or eax, 80000000h
mov cr0, eax
```

After the paging structures are initialized, the program writes three messages through virtual addresses that ultimately map to the same physical video buffer.

#### Listing 2.17. Demonstrating page translation

```
mov esi, message1
mov edi, 0B8000h
mov ecx, message2-message1
rep movsb
mov esi, message2
mov ecx, message3-message2
rep movsb
mov esi, message3
mov ecx, end_messages-message3
rep movsb
jmp $
message1 db "152535455565758595 5 5"
message2 db "A5d5r5F5F50505050505 5"
message3 db "A5d5r5E5E50505050505 5"
end_messages:
```

If the screen shows the text 123456789 AdrFF000000 AdrEE000000, the source considers the example successful.

### 2.3.6.2. PAE Mode

The PAE example is more complex because more paging structures are involved. The implementation otherwise resembles the ordinary example, so it prints only the virtual-page creation function.

#### Listing 2.18a. Creating a PAE virtual page

```
create_VirtAddressPAE:
; in
; EAX page address
; EBX phys page address
pushad
and eax, 0FFFFFF00h
and ebx, 0FFFFFF00h
mov edi, eax
mov esi, ebx
```

The low 12 bits of both addresses are cleared and the aligned virtual and physical addresses are saved.

#### Listing 2.18b. Creating a PAE virtual page (continued)

```
shr eax, 30
mov ebx, eax
mov ecx, eax ;ebx, ecx, eax index in Page Directories Table
shl eax, 3 ; eax = eax*8
add eax, PAGE_DIRECTORIES_POINTERS_TABLE_BASE_ADDRESS
; eax point to entry of Page Directories Table
```

The function extracts the index in the page-directory-pointer table, saves it in EBX and ECX, multiplies it by eight, and adds the PDPT base so EAX points to the required PDPTE.

#### Listing 2.18c. Creating a PAE virtual page (continued)

```
shl ebx, 12
add ebx, PAGE_DIRECTORIES_TABLE_BASE_ADDRESS
; ebx point to Page Directory
push ebx ; save ebx !!!!!
or ebx, 1 ; set present flag !!!!!
mov [eax], ebx
pop ebx ; backup ebx !!!!!!!
```

The index is multiplied by 4096 and added to the base of all page directories. The resulting directory address is saved, marked Present, written into the PDPTE, and then restored.

#### Listing 2.18d. Creating a PAE virtual page (continued)

```
mov eax, edi
shl eax, 2
shr eax, 23
mov edx, eax ; edx, eax index in page dir
shl eax, 3 ; eax = eax*8
add eax, ebx ; eax point to page dir entry
```

The virtual address is used to obtain the page-directory index. That index is saved in EDX, multiplied by eight, and added to the page-directory address in EBX so EAX points to the required PDE.

#### Listing 2.18e. Creating a PAE virtual page (continued)

```
imul edx, ecx
shl edx, 12
add edx, PAGE_TABLES_BASE_ADDRESS ; edx point to page table
push edx ; save edx !!!!!
or edx, 3 ; set present, write, supervisor flags
mov [eax], edx
pop edx ; backup edx !!!!!!!

mov ebx, edi
shl ebx, 11
shr ebx, 23 ; ebx index in page table
shl ebx, 3 ; ebx = ebx*8
add ebx, edx ; ebx point to page table entry
or esi, 3
mov [ebx], esi
.end:
popad
ret
```

The source multiplies the page-directory index by the page-directory-pointer index, multiplies the result by 4096, and adds PAGE\_TABLES\_BASE\_ADDRESS to locate a page table. It then sets Present and Write in the PDE, restores the page-table address, derives the PTE index, and writes the physical address with low flag bits set.

To enable PAE paging, the source loads CR3 with the PDPT address, sets CR4.PAE, and then sets CR0.PG:

#### Listing 2.19. Enabling PAE mode

```
mov eax, PAGE_DIRECTORIES_POINTERS_TABLE_BASE_ADDRESS
mov cr3, eax
mov eax, cr4
or eax, 32
mov cr4, eax
mov eax, cr0
or eax, 80000000h
```

```
mov cr0, eax
```

The source says the complete source code for both paging examples was supplied in the `part2` directory on the accompanying CD.

### **2.3.6.3. Using Pages of Different Sizes**

An operating system can use 4 KB and large pages at the same time. The source gives kernel memory as a typical candidate for large pages.

Large pages can improve performance because the processor spends less time translating addresses. Small pages let applications use memory more efficiently by reducing allocation granularity.

For example, if an application requires 48 KB, using a 4 MB page leaves 4048 KB of that page unused for the application's requested data, while twelve 4 KB pages allocate only the amount required. The same general trade-off applies to ordinary paging and PAE.

### **2.3.7. Summary**

This section covered page translation in both ordinary paging mode and Physical Address Extension (PAE) mode. The next section turns to protected-mode multitasking.

## 2.4. Multitasking

Multitasking is described as one of the fundamental characteristics of modern operating systems. The basic unit in a multitasking system is a task, or thread - in this section the source uses the word task. A task is an independent sequence of instructions executing in its own environment.

The task environment is characterized by the state of the general-purpose registers, segment registers, the address space - represented by CR3 when the operating system isolates tasks from one another - and the state of the mathematical coprocessor.

Because the number of tasks is usually much greater than the number of processors, multitasking is implemented by rapidly switching between tasks, creating the impression that all are running simultaneously.

### 2.4.1. General Information

The source presents x86 multitasking as hardware-supported: the processor itself can save the state of the interrupted task into a dedicated memory area and load the saved state of the next task. This memory area is the Task State Segment (TSS).

Each task can have four stacks, one for each privilege level. Each task may also use its own virtual address space and therefore its own page directory, providing isolation from other tasks and from system memory.

A task can have an I/O permission bitmap controlling access to individual I/O ports. In addition to the GDT, a task may use its own Local Descriptor Table (LDT), which may contain certain descriptor types.

The source distinguishes 16-bit and 32-bit tasks. 16-bit tasks originated with the 80286, while the 80386 and later processors added 32-bit tasks. The section discusses only 32-bit tasks.

### 2.4.2. Task State Segment (TSS)

Each task is described by a memory area containing its parameters, at minimum its register state. This is the Task State Segment. Like other memory objects used here, a TSS is described by a descriptor in the GDT and therefore has a selector.

|                      |                      |     |
|----------------------|----------------------|-----|
| I/O Map Base Address | Reserved   T         | 100 |
| Reserved             | LDT segment selector | 96  |
| Reserved             | GS                   | 92  |
| Reserved             | FS                   | 88  |
| Reserved             | DS                   | 84  |
| Reserved             | SS                   | 80  |
| Reserved             | CS                   | 76  |
| Reserved             | ES                   | 72  |
| EDI                  |                      | 68  |
| ESI                  |                      | 64  |
| EBP                  |                      | 60  |
| ESP                  |                      | 56  |
| EBX                  |                      | 52  |
| EDX                  |                      | 48  |
| ECX                  |                      | 44  |
| EAX                  |                      | 40  |
| EFLAGS               |                      | 36  |
| EIP                  |                      | 32  |
| CR3 (PDBR)           |                      | 28  |
| Reserved             | SS2                  | 24  |
| ESP2                 |                      | 20  |
| Reserved             | SS1                  | 16  |
| ESP1                 |                      | 12  |
| Reserved             | SS0                  | 8   |
| ESP0                 |                      | 4   |
| Reserved             | Previous Task Link   | 0   |

Figure 2.25. 32-bit Task State Segment format

The source divides TSS fields into dynamic and static fields. Dynamic fields are updated by the processor during task switching.

The dynamic fields listed by the source are:

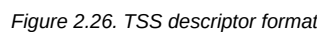
- General-purpose registers EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI.
- Segment registers ES, CS, SS, DS, FS, and GS.
- The flags register.
- EIP. The source calls this the most important field because without a saved instruction pointer a task would restart from its beginning after every switch.
- Previous Task Link, the first TSS field, containing the selector of the task that was executing before the current one and supporting nested tasks.

Static fields are not automatically updated by the processor; they are read from the TSS and should be initialized when the task is created:

- LDT selector.
- CR3 state.
- Stack segment and stack pointer for privilege levels 0, 1, and 2.

- Figure 2.25 shows the minimum permitted TSS. Additional data, such as the I/O permission bitmap, can extend the TSS. Software can also store other processor state in an enlarged TSS, although the processor will not automatically manage those extra fields.
- When paging is enabled, the TSS should lie completely within one page because the processor translates the TSS virtual address only once. If a TSS crosses two virtual pages that map to noncontiguous physical pages, the processor can obtain incorrect task-state data.

Every TSS must be described in the GDT by a special TSS descriptor; a TSS descriptor may be defined only in the GDT.



Because the TSS descriptor can exist only in the GDT, the source says the TI bit in a selector referencing a TSS must be clear.

**B (Busy).** The Busy bit indicates that a task is currently active. When set, it prevents switching to the same task recursively.

The Local Descriptor Table is analogous to the GDT but is intended for use in the context of one task. A task can therefore have its own set of directly accessible descriptors.

The LDT has a structure similar to the GDT and also does not use its zero descriptor. The main difference in the source is that system-object descriptors may not be defined in the LDT. A TSS is given as an example of a system object, so a task cannot define another task inside its own LDT.

An LDT may contain code- and data-segment descriptors and gate descriptors.

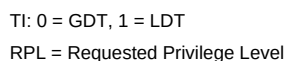


Figure 2.27. Selector format

TI determines which descriptor table is used: TI = 0 selects the GDT and TI = 1 selects the LDT.

The LDT itself must be described in the GDT by an LDT descriptor.

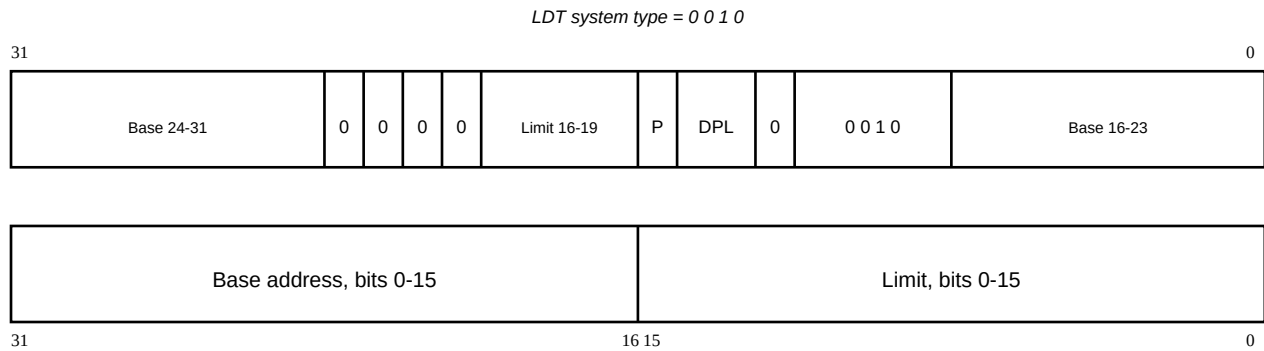


*Figure 2.28. LDT descriptor format*

LLDT loads the selector of an LDT descriptor into LDTR. Its operand is a 16-bit register or memory value containing the selector. The source groups LLDT with LGDT and LIDT as privileged instructions executable only at privilege level 0.

SLDT reads LDTR and is described as unprivileged. LDTR has an 8-byte hidden part that caches the descriptor so the processor does not have to reread the GDT on every LDT access.

If the LDT descriptor is modified, LDTR must be reloaded for the change to take effect in the hidden descriptor cache. LDTR is updated on every task switch.

### 2.4.5. Task Register (TR)

TR is a 16-bit register holding the selector of the TSS of the current task. Like LDTR, it has an 8-byte hidden part.

The source gives two purposes for TR:

1. Identifying the current task inside interrupt handlers that are not themselves tasks, which the source notes is useful during debugging.
2. Entering hardware multitasking mode.

LTR loads TR; STR reads it. Their syntax is analogous to LLDT/SLDT. LTR is privileged and STR is unprivileged according to the source.

Loading TR sets the Busy flag in the task's TSS descriptor. This is described as associating the current processor state with that task.

### 2.4.6. Task Management

The processor performs a hardware task switch automatically by using the TSS descriptor. A switch can be requested with three far-control-transfer instructions: JMP, CALL, and IRET.

For JMP or CALL, a TSS selector is supplied as the segment portion of the far target; the offset is ignored. With IRET, the target TSS selector is taken from Previous Task Link in the current TSS.

Switching to a task is possible only while the processor is already executing a task. Consequently, the first task must be established specially, and once hardware multitasking has begun, the source treats the processor as remaining in that regime until real-address mode or shutdown/reset.

Several flags participate in hardware task switching:

- **B (Busy)** in the TSS descriptor is set when switching to a task. Switching to an already busy task is disallowed. The source says JMP and IRET clear the old task's Busy bit, while CALL and interrupt/exception task switches leave it set.
- **T (Trace)** is bit 0 at TSS offset 64h. When set, a debug exception (vector 1) is generated after the new task state has been loaded. The source suggests this can be used for task debugging and for software extension of the saved context, for example loading FPU state.
- **NT (Nested Task)** is in EFLAGS. A switch caused by CALL, or by an interrupt/exception whose handler is a task, sets NT in the new task. IRET with NT = 1 switches back to the task whose selector is stored in Previous Task Link.
- **TS (Task Switched)** is in CR0. The processor sets it on every task switch. The source describes it as enabling lazy context handling for FPU, MMX, and XMM state through #NM.

The source lists four situations in which the processor performs a task switch:

1. A far JMP or far CALL whose selector directly names a TSS.
2. A far JMP or far CALL whose selector names a task gate in the GDT or LDT.
3. An interrupt or exception whose IDT entry is a task gate.
4. IRET when EFLAGS.NT = 1.

The source then gives the task-switch sequence:

1. Obtain the target TSS selector from the instruction, task gate, or Previous Task Link.
2. Perform privilege checks. For direct TSS switching, max(RPL, CPL) must not exceed the TSS descriptor DPL. For a task gate, max(RPL, CPL) must not exceed the gate DPL. These checks are not performed for cases 3 and 4.
3. Check the TSS limit.
4. Check task availability: in cases 1-3 the target must be not busy; for case 4, returning from a nested task, it must be marked busy.
5. Check presence of the old TSS, new TSS, and the segments used by the new task.
6. For JMP or IRET, clear the old task's Busy bit. For CALL or interrupt/exception switching, leave it set.
7. For IRET, clear NT in the old task's flags; otherwise leave the old task's NT value unchanged.
8. Save the old processor context into the old task's TSS.
9. For CALL or interrupt/exception switching, set EFLAGS.NT = 1 in the new task and store the old TSS selector in its Previous Task Link. For JMP or IRET, leave NT unchanged.
10. Mark the new task's TSS descriptor busy, or leave it busy if already required by the return case.
11. Load the new value into TR.
12. Load the processor context from the new TSS.
13. Begin executing the new task.

### 2.4.7. Task Gate

A task gate can be used as an interrupt handler. Using a task gate further isolates interrupt handlers from ordinary applications and can increase system reliability.

A task gate may appear in the IDT, GDT, or LDT.

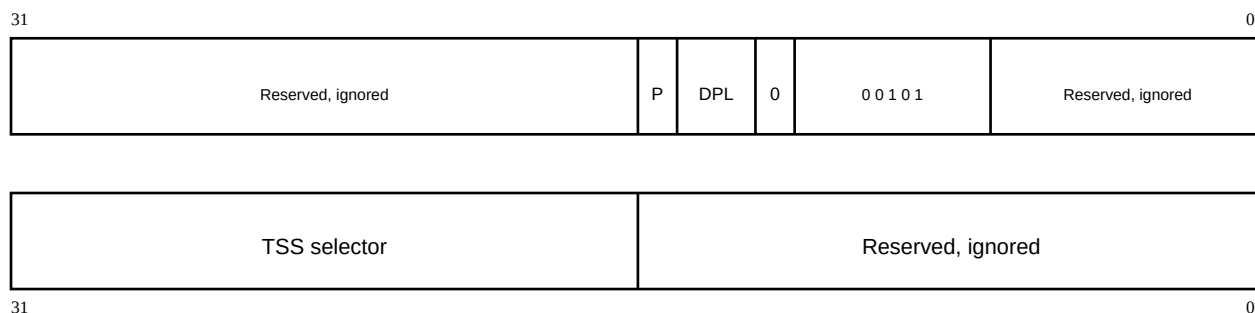


Figure 2.29. Task-gate descriptor format

The only object-specific value in the gate is the TSS selector, which determines the task to which the processor switches when the gate is entered.

### 2.4.8. I/O Privilege Level

The IOPL field in the flags register controls I/O operations performed by applications. It is a 2-bit current I/O privilege level.

IN, INS, OUT, OUTS, CLI, and STI may execute only when  $CPL \leq IOPL$ ; these instructions are therefore sensitive to I/O privilege.

An attempt by less-privileged code to use a prohibited I/O-sensitive instruction generates #GP.

A task or program can modify IOPL only with POPF or IRET, and such a change is privileged. Only code running at privilege level 0 can change IOPL. If less-privileged code attempts to change IOPL, no exception is generated; the field simply remains unchanged.

POPF can also alter IF, but in that use it is also sensitive to IOPL. Software may change IF with POPF only when  $CPL \leq IOPL$ ; otherwise the attempted IF change is ignored without an exception.

### 2.4.9. I/O Permission Bitmap

The I/O permission bitmap controls access to I/O ports by programs whose CPL is numerically greater than IOPL. A separate bitmap can be defined for each task and is stored within the TSS.

The first bitmap byte is located at the offset given by IOMapBaseAddress in the TSS. The map's address and size are flexible as long as the actual TSS size does not exceed the limit in the TSS descriptor.

Each bit corresponds to one 1-byte I/O port. A set bit denies access; a clear bit allows access. The source gives port 29h (41 decimal) as an example: its control bit is bit 1 of the sixth bitmap byte.

For wider port operations, the processor checks all bits corresponding to all constituent byte ports. A doubleword access therefore checks four adjacent bits. If any required bit is set, #GP is generated; if all are clear, the I/O operation is permitted.

The processor reads two bytes at a time when checking the bitmap. To avoid exceptions, the source says the bitmap should be followed by a byte with all bits set (0FFh).

The bitmap does not have to include every I/O port. Ports beyond the map are treated as prohibited. The bitmap must be at the end of the TSS because the TSS limit determines where the map ends.

The source gives an example: if the TSS limit extends 10 bytes beyond the bitmap base, the map is 11 bytes and covers the first 80 I/O ports; accesses above port 80 are prohibited.

If IOMapBaseAddress is greater than or equal to the TSS limit, the source treats the bitmap as absent and all I/O instructions generate exceptions whenever CPL > IOPL.

## 2.4.10. Enabling Multitasking

Because a hardware task switch can occur only from an already established task, the source says multitasking must be entered in three stages:

1. Define the TSS regions and their descriptors in the GDT.
2. Fill the TSS fields with the desired initial values. The source identifies EIP, CS, CR3 when paging is enabled, SS, and ESP as the most important fields.
3. Load TR with the selector of the TSS that describes the currently executing code.

After that, the processor is considered to be in the hardware multitasking environment and additional tasks can be defined and switched to.

The initial values in whichever TSS selector is first loaded into TR are not especially important because the current processor state will overwrite that TSS on the first task switch.

## 2.4.11. Practical Implementation

The example creates three tasks. The first does nothing. The second and third display messages to demonstrate that they execute. Interrupts remain enabled, and the source says a task gate is installed on vector 0 to act as the handler that alternates between the second and third tasks.

The task handler examines Previous Task Link in its own TSS and replaces the selector: if it contains the second task, it changes it to the third; otherwise it changes it to the second. In this way, the interrupt handler interrupts one task and returns into another.

The example first adds TSS descriptors to the GDT. Instead of modifying the existing GDT in place, it copies the GDT elsewhere, appends the new descriptors, and reloads GDTR. The source explicitly says this is not an optimal requirement; it is done to move the GDT above the first megabyte and to vary the example.

### GDTR32 structure used by the example

```
struct GDTR32
{
    .Limit dw ?
    .BaseAddress dd ?
}
GDTR_Image GDTR32
```

### Listing 2.20. Updating the GDT

```
mov esi, message1
mov al, 0
mov ah, 0
mov bl, "5"
call OutText
xor eax, eax
mov edi, NewGDT
mov ecx, NewGDTSize/4
rep stosd
sgdt [GDTR_Image]
mov esi, [GDTR_Image.BaseAddress]
mov edi, NewGDT
xor ecx, ecx
inc word [GDTR_Image.Limit]
mov cx, [GDTR_Image.Limit]
shr ecx, 2
rep movsd
mov edi, NewGDT
mov ax, [GDTR_Image.Limit]
add edi, eax
mov esi, TssDescriptors
mov ecx, TssDescriptorsSize/4
rep movsd
```

The program first reports that protected mode has been entered. The source does not print the OutText function here, but says ESI points to a zero-terminated string, AL is the column number, AH is the row number, and BL contains character attributes.

NewGDT is cleared, then SGDT obtains the existing GDT parameters and the table is copied. GDTR stores a limit rather than a byte count, so the source increments the limit before copying. The TSS descriptors are then appended.

#### Listing 2.21. Macro that creates a TSS descriptor

```
macro DEFINE_TSS_DESCRIPTOR _base_address, _limit
{
    dw _limit and 0FFFFh
    dw _base_address and 0FFFFh
    db (_base_address shr 16) and 0FFh
    db TSS_TYPE
    db (_limit shr 16) and 0Fh
    db _base_address shr 24
}
```

The macro takes the TSS base address and limit.

#### Listing 2.22. Declaring the required descriptors with DEFINE\_TSS\_DESCRIPTOR

```
TssDescriptors:
DEFINE_TSS_DESCRIPTOR FirstTaskTss, 100h
DEFINE_TSS_DESCRIPTOR TASK_NUMBER_1_Tss, 100h
DEFINE_TSS_DESCRIPTOR TASK_NUMBER_2_Tss, 100h
DEFINE_TSS_DESCRIPTOR Irq0_handler_Tss, 100h
TssDescriptorsEnd:
```

#### Listing 2.23. Updating GDTR

```
mov [GDTR_Image.BaseAddress], NewGDT
mov ax, [GDTR_Image.Limit]
add ax, TssDescriptorsSize
dec ax
mov [GDTR_Image.Limit], ax
lgdt [GDTR_Image]
mov esi, message2
mov al, 0
mov ah, 1
mov bl, "5"
call OutText
```

The example next defines a FASM structure to make TSS fields easier to address.

#### Listing 2.24. TSS32 structure

```
struc TSS32
{
    .PreviousTaskLink dw ?
    .Reserved0 dw ? ; --4
    .ESP0 dd ?
    .SS0 dw ?
    .Reserved1 dw ? ; --12
    .ESP1 dd ?
    .SS1 dw ?
    .Reserved2 dw ? ; 20
    .ESP2 dd ?
    .SS2 dw ?
    .Reserved3 dw ? ; 28
    .tsCR3 dd ?
    .tsEIP dd ? ; 36
    .tsEFLAGS dd ?
    .tsEAX dd ? ; 44
    .tsECX dd ?
    .tsEDX dd ? ; 52
    .tsEBX dd ?
    .tsESP dd ? ; 60
    .tsEBP dd ?
    .tsESI dd ?
    .tsEDI dd ? ; 72
    .tsES dw ?
    .Reserved4 dw ? ; 76
    .tsCS dw ?
    .Reserved5 dw ?
    .tsSS dw ?
    .Reserved6 dw ?
    .tsDS dw ?
    .Reserved7 dw ?
    .tsFS dw ?
    .Reserved8 dw ?
    .tsGS dw ?
    .Reserved9 dw ?
    .LDTSegmentSelector dw ?
    .Reserved10 dw ?
    .DebugByte db ?
    .Reserved11 db ?
    .IOMapBaseAddress dw ?
}
```

The initial state of a task is filled by Set\_TSS.

#### Listing 2.25. TSS initialization function

```

Set_TSS:
;IN
; EAX - CR3
; EBX - EIP
; ECX - ESP
; EDX - EFLAGS
; EDI - TSS base address
pushad
push eax
push edi
push ecx
xor eax, eax
mov ecx, 26
cld
rep stosd
pop ecx
pop edi
pop eax
virtual at edi
.edi TSS32
end virtual
mov [.edi.tsCR3], EAX
mov [.edi.tsEIP], EBX
mov [.edi.tsESP], ECX
mov [.edi.tsEFLAGS], EDX
mov [.edi.tsCS], CODE_SELEKTOR
mov [.edi.tsDS], DATA_SELEKTOR
mov [.edi.tsSS], DATA_SELEKTOR
mov [.edi.tsES], DATA_SELEKTOR
popad
ret

```

The function accepts five important parameters in general-purpose registers. It clears the first 104 bytes of the TSS so unspecified state begins as zero.

The FASM `virtual at` directive is used to access structure fields through the register pointing to the TSS. The source gives `mov [.edi.tsCR3], EAX` as an expression that becomes an address equivalent to `mov [edi+28], EAX`.

#### Listing 2.26. Initializing the tasks

```

mov eax, CR3 ; EAX = CR3
pushfd
pop edx ; EDX = EFLAGS
mov ecx, Main_task_stack
mov ebx, FirstTask
mov edi, FirstTaskTss
call Set_TSS
mov ecx, Task_number1_Stack
mov ebx, TASK_NUMBER_1
mov edi, TASK_NUMBER_1_Tss
call Set_TSS
mov ecx, Task_number2_Stack
mov ebx, TASK_NUMBER_2
mov edi, TASK_NUMBER_2_Tss
call Set_TSS
mov ecx, irq0_handler_Stack
mov ebx, irq0_handler
mov edi, Irq0_handler_Tss
call Set_TSS

```

The interrupt-controller and IDT initialization code is not reproduced in this section.

#### Listing 2.27. Enabling multitasking

```

mov ax, FirstTask_Selector
ltr ax
FirstTask:
mov esi, message3
mov al, 0
mov ah, 2
mov bl, "5"
call OutText
sti
jmp $

```

The `FirstTask` label is not fundamentally required and is present to make the code easier to read. The first task displays a message, enables interrupts, and enters an infinite loop.

#### Listing 2.28. Code of the first displayed task

```

message5 db "Task number 1 message "
counter1 dd 11111111h
db 0
TASK_NUMBER_1:
sti
mov esi, message5
mov al, 0
mov ah, 6
mov bl, "5"
@@:
call OutText
inc dword [counter1]
jmp @b

```

This task enables interrupts and repeatedly displays text. Interrupts have to be enabled again because the flags image used to initialize the task was captured before multitasking was enabled, when IF was clear.

Each iteration increments counter1; because the bytes following the message change, the source says four characters after the text should flicker. The anonymous label @@ works with @b for the nearest previous anonymous label and @f for the nearest following one.

#### Listing 2.29a. IRQ0 interrupt handler

```
irq0_handler:
mov edi, Irq0_handler_Tss
virtual at edi
.edi TSS32
end virtual
.repeat:
inc dword [counter]
mov esi, interrupt_mes
mov al, 0
mov ah, 4
mov bl, "5"
call OutText
```

The handler places a pointer to its TSS in EDI and, at the start of the loop, displays a message indicating that the handler was invoked.

#### Listing 2.29b. IRQ0 interrupt handler (continued)

```
xor eax, eax
mov ax, [.edi.PreviousTaskLink]
call ClearBusyFlag
cmp ax, TASK_NUMBER_2_Selector
jz @f
mov ax, TASK_NUMBER_2_Selector
call SetBusyFlag
mov [.edi.PreviousTaskLink], TASK_NUMBER_2_Selector
jmp .end_dispatch
@@:
mov ax, TASK_NUMBER_3_Selector
call SetBusyFlag
mov [.edi.PreviousTaskLink], TASK_NUMBER_3_Selector
```

The handler replaces the selector stored in Previous Task Link. It first clears the Busy flag in the descriptor of the previous task with ClearBusyFlag, which receives the selector in AX. The source does not print that function here; it says its code is available in the complete example. SetBusyFlag performs the analogous operation.

If Previous Task Link contains the second task's selector, the handler stores the third task's selector; otherwise it stores the second task's selector.

#### Listing 2.29c. IRQ0 interrupt handler (continued)

```
.end_dispatch:
mov al, 20h
out 020h, al
out 0a0h, al
iretd
jmp .repeat
```

At the end of the handler, an EOI operation is performed and IRETD returns control. On the next handler invocation execution resumes after the interrupted IRETD, so the following JMP returns execution to the beginning of the handler loop.

The full program source was supplied in the part2 folder on the book's accompanying CD.

## 2.4.12. Summary

This section examined the protected-mode mechanisms that provide hardware multitasking. The source then remarks that widely used operating systems such as Windows and UNIX do not use the x86 hardware task-switching mechanism for ordinary multitasking and instead implement task switching in software. Section 2.5 turns to protected-mode protection mechanisms.

## 2.5. Protection Mechanisms

Protected mode provides an operating system with a broad set of protection mechanisms. When protection is in use, every memory access must satisfy a series of protection conditions. These checks are completed before the memory access begins; violations generate exceptions, and the checks occur together with address translation without reducing performance.

Setting PE in CR0 enters protected mode and automatically enables protection. The part based on privilege checks can effectively be neutralized if a protected-mode program uses only descriptors and selectors at privilege level 0.

Page-level protection is automatically active when paging is enabled with CR0.PG. This protection can effectively be disabled by clearing CR0.WP and setting R/W and U/S in every relevant page-directory and page-table entry, making each page readable and writable from every privilege level.

### 2.5.1. Fields and Flags Used for Segment- and Page-Level Protection

**Descriptor-type flag S.** Bit 12 of the descriptor's second doubleword. If 0, the descriptor describes a system object; if 1, a code or data segment.

**Descriptor type field.** Bits 8-11 of the second doubleword. These identify the type of system object, code segment, or data segment.

**Descriptor limit.** Bits 0-15 and 48-51 of the descriptor. This defines segment size in bytes or 4 KB pages depending on G.

**Granularity G.** Bit 23 of the descriptor's second doubleword. G = 0 means the limit is measured in bytes; G = 1 means 4 KB pages.

**E flag.** Bit 10 in a data-segment descriptor. E = 0 means the segment expands upward; E = 1 means it expands downward, as with a stack.

**DPL.** Bits 13-14 of the second doubleword of a descriptor. Defines the privilege level of the described object.

**RPL.** Bits 0-1 of a selector. Contains the requested privilege level used when accessing a descriptor.

**CPL.** Bits 0-1 of CS. Contains the privilege level of the currently executing code.

**U/S.** Bit 2 of a page-directory or page-table entry. U/S = 0 denotes a system page and U/S = 1 a user page.

**R/W.** Bit 1 of a page-directory or page-table entry. R/W = 0 means read-only and R/W = 1 read/write.

The source remarks that Windows uses very little segment-level protection in practice: programs are given segments covering the available address space, while operating-system data protection is primarily enforced through page translation.

### 2.5.2. Segment-Limit Checks

The segment-descriptor limit prevents a program from accessing beyond the segment. The meaning of the limit depends on G and represents the maximum permissible offset for an ordinary upward-growing segment.

For a downward-expanding data segment, the source describes the limit as the final forbidden address within the segment. Valid addresses range from limit + 1 through FFFFh when D = 0, or through FFFFFFFFh when D = 1. Such a segment has its maximum size when the limit is zero.

The processor also checks the limits of the GDT, IDT, LDT, and current TSS so that software cannot access beyond those structures.

### 2.5.3. Type Checks

A segment descriptor carries type information in two places: the S flag and the type field. The processor uses them to reject operations that attempt to use a segment or gate in an invalid or inappropriate way.

Bit 8 of the type field is A, indicating whether the segment has been accessed since software last cleared the bit. Code segments may be conforming or nonconforming; C, bit 10 of the descriptor's second doubleword, selects that behavior.

The segment loaded into SS must be writable. A stack segment does not have to be downward-expanding; Windows and the book's earlier examples use an ordinary data segment for the stack.

**Table 2.3. Flags in non-system descriptors**

| Bit 11 | Bit 10 | Bit 9 | Type                           |
|--------|--------|-------|--------------------------------|
| 0      | 0      | 0     | Data, read-only                |
| 0      | 0      | 1     | Data, read/write               |
| 0      | 1      | 0     | Data, read-only, expands down  |
| 0      | 1      | 1     | Data, read/write, expands down |
| 1      | 0      | 0     | Code, execute-only             |
| 1      | 0      | 1     | Code, execute/read             |
| 1      | 1      | 0     | Code, execute-only, conforming |
| 1      | 1      | 1     | Code, execute/read, conforming |

**Table 2.4. System-descriptor types**

| Value | 11 | 10 | 9 | 8 | Description            |
|-------|----|----|---|---|------------------------|
| 0     | 0  | 0  | 0 | 0 | Reserved               |
| 1     | 0  | 0  | 0 | 1 | 16-bit TSS (available) |
| 2     | 0  | 0  | 1 | 0 | LDT                    |
| 3     | 0  | 0  | 1 | 1 | 16-bit TSS (busy)      |
| 4     | 0  | 1  | 0 | 0 | 16-bit call gate       |
| 5     | 0  | 1  | 0 | 1 | Task gate              |
| 6     | 0  | 1  | 1 | 0 | 16-bit interrupt gate  |
| 7     | 0  | 1  | 1 | 1 | 16-bit trap gate       |
| 8     | 1  | 0  | 0 | 0 | Reserved               |
| 9     | 1  | 0  | 0 | 1 | 32-bit TSS (available) |
| 10    | 1  | 0  | 1 | 0 | Reserved               |
| 11    | 1  | 0  | 1 | 1 | 32-bit TSS (busy)      |
| 12    | 1  | 1  | 0 | 0 | 32-bit call gate       |
| 13    | 1  | 1  | 0 | 1 | Reserved               |
| 14    | 1  | 1  | 1 | 0 | 32-bit interrupt gate  |
| 15    | 1  | 1  | 1 | 1 | 32-bit trap gate       |

The source lists common operations in which descriptor type is checked:

- Only a code-segment selector may be loaded into CS.
- A selector for execute-only code cannot be loaded into DS, ES, FS, or GS.
- SS may contain only the selector of a writable data segment.
- LDTR may contain only an LDT selector.
- TR may contain only a TSS selector.
- Software cannot write to a code segment.
- Software cannot write to a read-only data segment.
- Software cannot read from an execute-only code segment.
- LAR must refer to one of the descriptor classes permitted for that instruction: LDT, TSS, call gate, task gate, code segment, or data segment, in the source's list.
- LSL must refer to an LDT, TSS, code-segment, or data-segment descriptor.
- The source states that an IDT entry may be an interrupt, trap, or call gate.

*Editorial note: the IDT normally uses interrupt, trap, and task gates; the source prints 'call gate' in this list. This edition preserves that statement rather than silently replacing it.*

- For FAR CALL and FAR JMP, the processor checks the target descriptor type. A code-segment or call-gate target performs a control transfer; a TSS or task-gate target performs a task switch.

Loading the null selector into CS or SS generates #GP. The null selector may be loaded into DS, ES, FS, or GS, but a later memory access through that register generates #GP according to the source.

## 2.5.4. Privilege Levels

Protected mode uses four privilege levels, 0 through 3; smaller numerical values mean greater privilege. These are often pictured as protection rings.

**CPL** is the privilege level of the currently executing code and is stored in bits 0-1 of CS and SS. Normally it equals the DPL of the current code segment. CPL changes when control passes to a code segment at a different privilege level, except for conforming code segments, where CPL remains that of the caller.

**RPL** is assigned to a selector and stored in bits 0-1. The processor evaluates RPL together with CPL. Even if the caller's CPL would permit access, an insufficiently privileged RPL can deny the access. The source presents this as a way to stop privileged code from accessing a segment on behalf of an application unless the application itself has sufficient privilege.

**DPL** is the privilege level stored in the descriptor of a segment or gate. Its interpretation depends on descriptor type:

- For a data segment, DPL is the numerically greatest privilege level permitted to access it. If DPL = 1, code at CPL 0 or 1 may access it.
- For a nonconforming code segment without a call gate, DPL is the exact privilege level required for access.
- For a call gate, DPL is the numerically greatest privilege level from which the current program may access the gate.

- For conforming code, or nonconforming code entered through a call gate in the source's combined description, DPL defines the numerically smallest privilege level allowed to access the segment.
- For a TSS, DPL is the numerically greatest privilege level from which the program or task may access the TSS.

Privilege checks occur when a segment selector is loaded. Data access and intersegment control transfer use different rules, treated separately below.

The stack privilege level must equal the code privilege level: the stack selector's RPL must equal its DPL and the code CPL. Otherwise #GP is generated.

### 2.5.5. Privilege Checks When Accessing Data Segments

To access an operand in a data segment, software loads the selector into DS, ES, FS, or GS using instructions such as MOV, POP, LDS, LES, LFS, or LGS.

Before accepting the selector, the processor compares CPL, selector RPL, and descriptor DPL. The source gives the acceptance rule as DPL being numerically greater than or equal to both CPL and RPL. Otherwise the load fails and #GP is generated.

For SS, the source repeats the stricter rule: stack-selector RPL must equal the stack-segment DPL and the current code CPL.

### 2.5.6. Privilege Checks During Intersegment Control Transfer

To transfer control from one code segment to another, the target selector must be loaded into CS. The processor checks the target segment's limit, type, and privilege level; if the checks pass, control is transferred.

The source lists JMP, CALL, RET, SYSENTER, SYSEXIT, INT n, and IRET as instructions that can participate in control transfer.

A far JMP or CALL may reach another segment in four ways:

- The instruction operand directly contains the target segment selector.
- The operand selects a call gate containing the target segment selector.
- The operand selects a TSS containing the target code segment.
- The operand selects a task gate pointing to a TSS containing the target code segment.

The source limits the following discussion to the first two cases because task switching was covered in Section 2.4. SYSENTER and SYSEXIT are covered later in this section.

Near JMP, CALL, and RET stay within the current code segment and do not perform intersegment privilege checks. Far forms move between code segments and do.

For direct transfer to another code segment without a call gate, the source says the processor checks the caller CPL, target code-segment DPL, target-selector RPL, and target segment's conforming flag C.

#### 2.5.6.1. Transfer to a Nonconforming Code Segment

For direct access to a nonconforming code segment, the caller CPL must equal the target DPL or #GP is generated. The target selector RPL must be numerically no greater than the current CPL.

When the selector is loaded into CS, the source says CPL does not change even if the selector RPL has a different value.

#### 2.5.6.2. Transfer to a Conforming Code Segment

For a conforming code segment, the current CPL may be numerically equal to or greater than the target DPL. #GP occurs only if the current CPL is numerically less than the DPL. RPL is not checked.

The DPL of a conforming segment therefore represents the numerically smallest privilege level from which callers may enter it.

CPL does not change when control enters conforming code, even if target DPL is more privileged. Therefore no stack switch occurs.

The source gives mathematical libraries and exception-support modules as examples of conforming code: they can be shared by applications without granting those applications access to more privileged system resources.

Most code segments are normally nonconforming, so direct transfer to them requires the same privilege level except when a call gate is used.

### 2.5.7. Call Gates

For controlled access to code at other privilege levels, the processor provides four gate descriptor types:

1. Call gate.
2. Trap gate.
3. Interrupt gate.
4. Task gate.

Trap and interrupt gates are used in the IDT and were covered in Section 2.2. Task gates were covered in Section 2.4. This subsection focuses on call gates.

A call gate transfers control between privilege levels and can also bridge 16-bit and 32-bit code. Call-gate descriptors are stored in the GDT or LDT, not in the IDT.

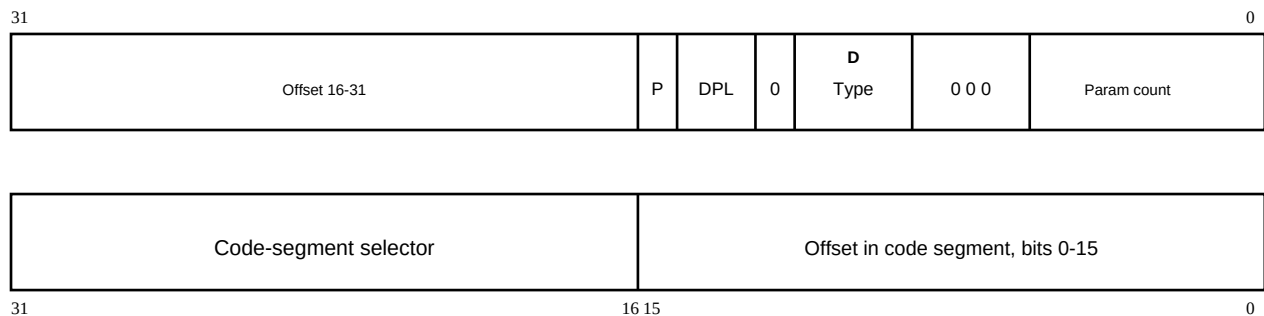


Figure 2.30. Call-gate descriptor format

A call gate performs six functions, but the printed list contains five:

1. Specifies the code segment to access.
2. Specifies the entry point in that segment.
3. Specifies the privilege level required of the calling procedure.
4. When the stack changes, specifies the number of optional parameters the processor copies from the caller's stack to the callee's stack.
5. Specifies the size of values placed on the destination stack: a 16-bit gate uses 16-bit values and a 32-bit gate uses 32-bit values.

*Editorial note: the prose explicitly says 'six functions' but lists only five. This edition preserves that count and notes the mismatch.*

The gate's selector field identifies the target code segment; the offset fields identify the entry point. Gate DPL is the privilege level required of code that wants to invoke the gate.

The parameter-count field specifies how many 16- or 32-bit parameters are copied from the caller's stack to the destination stack if a stack switch occurs.

P is normally set. If clear, attempting to use the gate generates #NP. An operating system could deliberately clear P to count gate traversals: the #NP handler increments a counter, sets P, and returns, after which the processor retries the gate.

To use a call gate, software performs a far CALL or JMP with the selector of the desired gate. The offset portion of the far pointer is ignored.

The processor compares four privilege values during gate transfer: CPL, selector RPL, gate DPL, and target code-segment DPL. It also checks the target's conforming flag.

These rules:

1. For both CALL and JMP, CPL and RPL must be numerically no greater than the call-gate DPL.
2. For both instructions, if the target code segment is conforming, its DPL must be numerically no greater than CPL.
3. For a nonconforming target, CALL permits target DPL ≤ CPL, while JMP requires target DPL = CPL.

Thus only CALL can transfer control to a more privileged nonconforming code segment. JMP can enter only a nonconforming segment at the same CPL. Both instructions can enter a more privileged conforming segment.

If CALL enters a more privileged nonconforming segment, CPL changes to the target DPL and the processor switches stacks. If control enters a more privileged conforming segment, CPL and the stack remain unchanged.

The source presents call gates as a mechanism by which less-privileged software can invoke operating-system services without gaining direct access to devices or other protected resources.

## 2.5.8. Stack Switching

Whenever a call gate transfers control to a more privileged nonconforming code segment, the processor automatically switches to the stack associated with the destination privilege level.

The source gives two reasons: privileged code should not fail merely because the user stack lacks room, and privileged code should be protected from interference through a less-privileged stack.

Each task may have up to four stacks: the current level-3 stack and stacks for privilege levels 0, 1, and 2. TSS contains SS:ESP pairs for levels 0-2. The processor does not update these static pointers while the task runs; it uses them when creating a stack for a

more-privileged call.

No separate TSS field is provided for the level-3 stack because the processor will not normally transfer control from privilege levels 0-2 to level 3 except during a return or privileged procedure, according to the source.

The operating system must create and size the stacks. Each should have room for:

1. Saved caller registers SS, ESP, CS, and EIP.
2. Parameters passed to the callee and temporary variables used by it.
3. EFLAGS and an error code when an interrupt or exception handler is involved.

Even an operating system that does not use hardware task switching must define at least one TSS if it uses privilege-level stack switching.

For a call-gate transfer that changes privilege, the source lists these actions:

1. Use target-code DPL as the new CPL and select the corresponding stack pointer from the TSS.
2. Check the new stack selector and offset; violations generate #TS.
3. Check the stack-segment descriptor's type and privilege; violations generate #TS.
4. Temporarily save the current SS and ESP.
5. Load the new SS and ESP.
6. Push the temporarily saved SS and ESP onto the new stack.
7. Copy the number of parameters specified by the gate descriptor from the caller's stack to the new stack. A zero parameter count means none are copied.
8. Push the return address - current CS and EIP - onto the new stack.
9. Load CS and EIP from the call gate and begin executing the called procedure.

The source then returns to RETF. RETF performs a far return by restoring CS and EIP and, when the destination privilege differs, switching stacks.

A RETF privilege change can return only to a less-privileged level, meaning a numerically larger DPL. The processor uses the RPL in the saved CS to decide whether the privilege level must be raised numerically. If saved CS.RPL is not numerically greater than current CPL in the source's description, #GP is generated; otherwise the processor performs the reverse of the call-gate transition.

### 2.5.9. Using SYSENTER and SYSEXIT

SYSENTER and SYSEXIT provide a fast path for invoking privileged system procedures. SYSENTER enters the system procedure and, may be invoked from any privilege level. SYSEXIT returns from the system procedure, may execute only at privilege level 0, and always returns to privilege level 3.

The pair is described as optimized for transfers from level 3 to level 0 and says using them for other privilege transitions is not useful.

SYSENTER/SYSEXIT are companion instructions, but unlike CALL/RET they do not preserve return state for each other. They have no explicit operands; their state is supplied through MSRs.

For SYSENTER:

1. Target CS comes from IA32\_SYSENTER\_CS.
2. Target EIP comes from IA32\_SYSENTER\_EIP.
3. Target SS is computed as IA32\_SYSENTER\_CS + 8.
4. Target ESP comes from IA32\_SYSENTER\_ESP.

For SYSEXIT:

1. Target CS is IA32\_SYSENTER\_CS + 16.
2. Target EIP comes from EDX.
3. Target SS is IA32\_SYSENTER\_CS + 24.
4. Target ESP comes from ECX.

Because very little memory state is touched, the source calls this the fastest mechanism for system-procedure entry among those discussed.

**Table 2.5. MSR indexes for SYSENTER/SYSEXIT**

| MSR               | Index |
|-------------------|-------|
| IA32_SYSENTER_CS  | 174h  |
| IA32_SYSENTER_ESP | 175h  |
| IA32_SYSENTER_EIP | 176h  |

### 2.5.10. Practice

The final protected-mode example demonstrates SYSENTER/SYSEXIT. The program is divided into ring-3 and ring-0 code. Each side mainly prints messages. Ring 3 loops and invokes ring 0 with SYSENTER; ring 0 prints a message and returns.

The main constants and structures are moved into `pmstructures.asm` for convenience.

#### Listing 2.30. Updating the GDT

```
include 'pmstructures.asm';
Ring0Stack equ 200000h ; ring-0 stack
Ring3Stack equ 180000h ; ring-3 stack

CS_r0 equ 8h ; selector for ring-0 code
DS_r0 equ 10h ; selector for ring-3 stack

START_CODE:
mov esi, message1
mov al, 0
mov ah, 0
mov bl, "5"
call OutText ; call procedure that outputs a string to the screen
sgdt [GDTR_Image]
mov eax, NewGDT
mov [GDTR_Image.BaseAddress], eax
mov ax, NewGDTEnd - NewGDT - 1
mov [GDTR_Image.Limit], ax
lgdt [GDTR_Image]
jmp CS_r0:@f
@@:
mov ax, DS_r0
mov ds, ax
mov ss, ax
```

*Editorial note: the printed comment for DS\_r0 says 'selector for the stack of the third ring' even though the symbol name and surrounding code are used in ring 0 as well. The comment is translated faithfully.*

At startup the example prints a protected-mode message. It replaces the GDT so two ring-3 descriptors can be added and all descriptors placed in the required order. GDTR is loaded with NewGDT and the segment registers are reloaded afterward.

#### Listing 2.31. Initializing SYSENTER/SYSEXIT

```
mov esi, message2
mov al, 0
mov ah, 1
mov bl, "5"
call OutText
xor edx, edx
mov ecx, IA32_SYSENTER_CS
mov eax, CS_r0
wrmsr
mov ecx, IA32_SYSENTER_CS
mov eax, CS_r0
wrmsr
mov ecx, IA32_SYSENTER_EIP
mov eax, Ring0Code
wrmsr
mov ecx, IA32_SYSENTER_ESP
mov eax, Ring0Stack
wrmsr
mov esi, message3
mov al, 0
mov ah, 2
mov bl, "5"
call OutText
```

*Editorial note: IA32\_SYSENTER\_CS is written twice with the same value in Listing 2.31. This duplication is preserved exactly rather than silently removed.*

The program reports that the GDT was updated, initializes the required MSRs, and then reports that initialization.

#### Listing 2.32. Ring-3 and ring-0 code using SYSENTER/SYSEXIT

```
mov edx, Ring3Code
mov ecx, Ring3Stack
sysexit
Ring3Code:
mov ax, DS_r3
mov ds, ax
mov esi, message4
mov al, 0
mov ah, 3
mov bl, "5"
call OutText
```

```

@@:
inc [counter]
mov edx, esp
sysenter
sysenter_ret:
jmp @b
Ring0Code:
mov ax, DS_r0
mov ds, ax
mov esi, message5
mov al, 0
mov ah, 4
mov bl, "5"
call OutText
mov ecx, edx
mov edx, sysenter_ret
sysexit

```

After reporting that the structures for SYSENTER/SYSEXIT have been initialized, the program executes SYSEXIT to enter ring 3. Ring-3 code prints its own message and repeatedly enters ring 0 with SYSENTER while incrementing a counter.

Because the counter is not converted to text, the source says changing non-text characters appear after the displayed message.

Ring-3 code in this example is otherwise barely restricted and can print directly. The ring-0 print is included only to demonstrate SYSENTER/SYSEXIT.

For a real operating system, ES, FS, and GS would also have to be reloaded as appropriate because ring-3 and ring-0 data segments may differ.

The remainder of the example is omitted from the book; the complete source was supplied in the par t2 folder on the accompanying CD.

### 2.5.11. Summary

This section completes the book's description of protected mode. It covered the fundamental mechanisms used to protect operating-system data from ordinary programs. The following chapters move on to programming under 32-bit and 64-bit Windows and to processor operation in 64-bit mode.

# Chapter 3. Programming in Win32

## 3.1. Introduction to Win32

The previous chapter discussed protected mode, more precisely the foundations from which an operating system can be built: protection mechanisms, multitasking, memory management, and interrupt handling. The source points out that creating programs that operate without an operating system is itself a difficult engineering task and in practice amounts to creating a small operating system.

This chapter turns to assembly programming under the 32-bit version of Windows.

The source defines Win32 as a family of protected-mode multitasking operating systems and then gives a historical overview. It says that the first 32-bit Windows operating system appeared in 1995 as Windows 95, version 4.0; Windows 98 followed in 1998 and Windows Millennium Edition in 2000. These products are grouped as the Windows 9x family.

*Historical note: later on the same page, the book describes Windows NT 3.1 as a fully 32-bit Windows system released in 1993. The statement calling Windows 95 the first 32-bit Windows operating system is therefore internally inconsistent. It is translated rather than silently corrected.*

The source characterizes the principal goal of Windows 9x as moving users to 32-bit programs while preserving compatibility with MS-DOS software. Because MS-DOS gave programs extensive access to hardware, operating-system data, drivers, and other programs, the source presents Windows 9x as vulnerable to accidental or deliberate memory corruption and system instability.

In parallel, Microsoft developed the Windows NT line, which the source describes as a fully 32-bit family not dependent on MS-DOS. It lists Windows NT 3.1 (1993), NT 3.5 (1994), NT 3.51 (1995), NT 4.0 (1996), Windows 2000, Windows XP, Windows Server 2003, 64-bit Windows XP, Windows Vista, and later systems.

Because Windows 9x was already obsolete when the book was written and is described by the author as an incomplete 32-bit design, the remainder of the chapter concentrates on the NT family.

*Historical note: this chapter reflects Windows architecture and development practices as presented in the 2011 source. Statements about obsolete systems, subsystem behavior, fixed library addresses, and system-call mechanisms are retained as historical source material.*

### 3.1.1. General Information

The 32-bit Windows NT systems are protected-mode operating systems. Their security model is based on separating user code from system code. In the source, 'user code' means any application code executing at privilege level 3, not code literally associated with a particular human user.

User-mode code executes in ring 3 and is subject to many restrictions. System code executes in ring 0, or kernel mode, and is subject to far fewer restrictions.

The fundamental unit of execution in Win32 is the thread, which corresponds roughly to a task in the terminology used in the protected-mode chapter. Threads are grouped into processes. In the general case, one running program corresponds to one process, and a process may contain any number of threads.

Each process is isolated from the others through its own virtual address space. If a program needs to access another process's memory, it can do so only through system services.

Thread scheduling is priority-based. Thread priorities are not static and may change according to what a thread is doing. At any instant, the processor executes a thread chosen according to the scheduler's priority rules; this subject is too complex to summarize fully here.

Most executable components in Windows run in ring 3. The source also emphasizes that Windows multitasking is implemented by software rather than the processor's hardware task-switching mechanism.

### 3.1.2. Memory in Win32

Each process has its own virtual address space. The normal 32-bit split is described as two equal halves: the lower 2 GB for the process and the upper 2 GB for the system.

System memory is shared conceptually across processes but is inaccessible from user mode even for reading. Attempts to access it produce a memory-access error and terminate the application in the source's description.

The source also notes a 3 GB / 1 GB split for applications requiring more user virtual address space. It remarks that after the arrival of 64-bit systems this arrangement became less important.

Addresses from 0 through FFFFh are described as unavailable so that null-pointer errors can be detected. Any pointer whose value is below 10000h is treated as null. It therefore gives the normally usable per-process Win32 virtual address space as 2 GB minus 64 KB.

### 3.1.3. Windows Executable Components

Windows uses several executable-file types, all based on the PE (Portable Executable) format in the source's discussion. The most common are EXE applications, DLL dynamic-link libraries, and SYS kernel-mode drivers.

A PE file contains a header and sections. The header contains the principal characteristics of the file and the section table. The source does not attempt a complete PE-format description here, but introduces the concepts needed for later examples.

Two of the most important header values are the entry point and image base. The image base indicates the address at which the module is intended to be loaded. The entry point identifies the address from which execution begins.

The linker resolves labels into concrete addresses under the assumption that the module will be loaded at its preferred base. If a DLL or SYS file cannot be loaded at that base because the address range is occupied or unavailable, the loader may place it elsewhere and use relocation information to correct memory references.

A module may contain import and export tables. Through the import table, a module can use functions exported by other modules loaded into the same address space. To make its own functions available to other modules, a module records them in its export table.

The section table describes each PE section: where its data begin, its size, where it should be mapped, and its characteristics. Common sections contain data, code, imports, exports, or relocations.

EXE files are ordinary programs. DLL files contain reusable functions for programs and other DLLs. SYS files contain kernel-mode driver code. For DLL and SYS modules, the source says the entry point identifies an initialization function.

### 3.1.4. System Libraries and Subsystems

User-mode Win32 code is prohibited from directly accessing I/O ports, invoking arbitrary interrupts, using privileged instructions, or directly using memory-mapped device registers. Applications would therefore be unable to interact with the outside world without operating-system services.

Windows exposes such services as API functions located in system DLLs. The source names KERNEL32.DLL for interaction with the system, USER32.DLL for the user interface, and GDI32.DLL for graphics.

It describes many KERNEL32.DLL functions as wrappers around functions in NTDLL.DLL. In the historical systems discussed by the source, NTDLL routines prepare parameters for entry into kernel services through the INT 2Eh mechanism on Windows 2000 or SYSENTER on Windows XP and later.

Windows NT was originally designed to support multiple executable subsystems, including Win32, POSIX, OS/2, and MS-DOS. It describes DOS programs as running inside NTVDM and calls NTDLL.DLL fundamental to the user-side implementation of the subsystems.

It further states that NTDLL.DLL is loaded into every process early and at the same address, and that KERNEL32.DLL is similarly loaded into every Win32 process after NTDLL.

*Historical note: the fixed-address statements reflect the source's pre-ASLR-era framing and should not be read as a current Windows guarantee.*

### 3.1.5. Function Calling Model in Win32

Win32 system functions use the stdcall calling convention. Under stdcall, arguments are pushed onto the stack in reverse order and the called function is responsible for removing them.

For a function with three parameters, the source gives:

#### stdcall call with three parameters

```
push param3
push param2
push param1
call FunctionAddr
```

The function result is returned in EAX.

The stdcall convention requires EBX, ESI, EDI, and EBP to be preserved across the call. The source also emphasizes that callback functions must preserve these registers because system code does not expect them to be changed.

A complete example of writing a stdcall-compatible function appears in Section 3.1.7.2.

### 3.1.6. Program Execution in Win32: General Picture

The source summarizes Win32 program startup as follows. When the Windows loader loads an executable file, it creates a 4 GB virtual address space for the process, with the lower 2 GB accessible to the application under the standard split.

The loader maps NTDLL.DLL and KERNEL32.DLL, then loads any additional DLLs named by the executable's import table. Once the required modules are available, Windows creates the process's primary thread and begins execution at the program's entry point.

During execution, user-mode threads cannot directly access I/O ports, invoke privileged interrupts, manipulate privileged registers, or execute privileged instructions. Instead they call API functions supplied by Windows.

API functions provide controlled access to devices, system resources, files, processes, and other protected objects. Functions that need protected resources eventually invoke the corresponding kernel service in the source's description.

The source concludes that Windows places the program in an isolated address space and gives it access to the external world only through services the operating system itself provides. This relieves the assembly programmer from having to manage system registers, hardware devices, file-system implementation, and similar operating-system responsibilities directly.

### 3.1.7. Practice

#### 3.1.7.1. First Win32 Program

The first practical example is a FASM program that displays the message 'Hello World!'.

##### Listing 3.1. Source code of a simple Win32 program

```
format PE GUI 4.0
include 'win32a.inc'
entry start

section '.data' data readable writeable
caption db 'First Win32 program',0
Message db 'Hello World!',0

section '.code' code readable executable
start:
push 0
push caption
push Message
push 0
call [MessageBox]
push 0
call [ExitProcess]

section '.relocs' fixups readable writeable
section '.idata' import data readable writeable
library kernel,'KERNEL32.DLL',\
        user,'USER32.DLL'
import kernel,\
        ExitProcess,'ExitProcess'
import user,\
        MessageBox,'MessageBoxA'
```

format PE GUI 4.0 tells FASM to produce a PE GUI executable. The source says the GUI 4.0 designation marks the file as a windowed application.

win32a.inc provides macros and constants for the ANSI versions of Win32 functions. The source explains the Windows naming convention: functions ending in A use ANSI strings, while versions ending in W use Unicode strings.

The source also discusses configuring FASMW.INI so that the INCLUDE directory can be referenced without writing a long relative or absolute path every time:

##### FASMW.INI example from the source

```
[Environment]
include=E:\FASMW166\include
```

The entry directive sets the program entry point. section declares a PE section. The source lists section name, section type, and access attributes as the important section properties, and recommends separating code, data, and imports rather than placing everything into one section.

The data section contains two zero-terminated strings: the message-box caption and message text. Win32 string functions generally use a terminating zero; with Unicode strings, termination is represented by two zero bytes.

MessageBox receives four parameters: owner-window handle, message-text pointer, caption pointer, and flags. Passing zero as the owner gives the message box no owner. Passing zero as the flags gives a simple OK button.

Because stdcall arguments are pushed in reverse order, the source highlights this call:

##### MessageBox call from Listing 3.1

```
push 0
push caption
push Message
push 0
call [MessageBox]
```

The return value identifies which button the user pressed, but this example does not use it.

The program terminates with ExitProcess, passing a process exit code of zero:

##### ExitProcess call

```
push 0
call [ExitProcess]
```

The example creates a relocation section even though the source says ordinary EXE files almost never need one in the situations being discussed.

The import section tells the loader which modules and exported functions the program needs. The source uses FASM's library and import macros to simplify creation of the import table.

The imported name MessageBoxA selects the ANSI version of MessageBox from USER32.DLL.

#### 3.1.7.2. STDCALL Function

The second example examines how to write a function that follows stdcall.

Every function or subroutine can own a region of the stack containing its parameters, local variables, and return address. The source calls this region the function's stack frame.

EBP is conventionally used as the stack-frame pointer. The source associates this convention partly with historical practice and partly with the ENTER and LEAVE instructions, which use EBP.

ENTER takes two operands. The first gives the size of local data and the second gives the nesting level of the stack frame. With a zero nesting value, ENTER saves EBP, copies ESP to EBP, and subtracts the local-data size from ESP. The source does not cover nested stack frames because it considers them to have little practical use.

LEAVE takes no operands. It copies EBP to ESP and then restores the caller's EBP from the stack. Because EBP preserves the original frame position, temporary changes to ESP can be undone by LEAVE.

Function parameters lie at positive offsets from EBP and local variables at negative offsets.

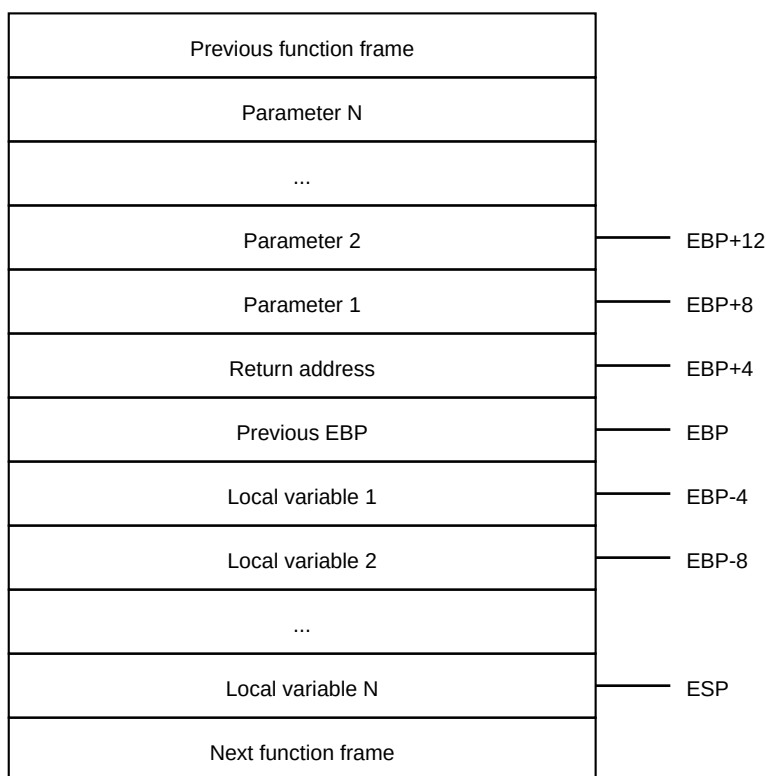


Figure 3.1. Function stack frame

The source then gives a function with four parameters and two 4-byte local variables.

### Listing 3.2. Example stdcall function

```
Func:
enter 8, 0
mov eax, [ebp+8] ; eax = first function parameter
mov [ebp-4], eax ; first local variable = eax
mov ebx, [ebp+16] ; ebx = third function parameter
mov [ebp-8], ebx ; second local variable = ebx
. . .
. . .
leave
retn 16
```

The source says ENTER is rarely used in practice. A normal prologue instead uses push ebp and mov ebp, esp, optionally followed by sub esp, N for local variables.

It claims that ENTER always occupies four opcode bytes and that PUSH/MOV together occupy three bytes, and further claims PUSH/MOV/SUB ESP executes approximately five times faster than ENTER when locals are used.

*Historical note: the ENTER performance comparison is presented as a source claim tied to the processors considered by the 2011 book; it is not generalized here to modern processors.*

The source says LEAVE is more commonly retained than an explicit mov esp, ebp / pop ebp pair.

It then presents the same function without ENTER.

### Listing 3.3. Example stdcall function without ENTER

```
Func :
push ebp
```

```

mov ebp, esp
sub esp, 8
mov eax, [ebp+8] ; eax = first function parameter
mov [ebp-4], eax ; first local variable = eax
mov ebx, [ebp+16] ; ebx = third function parameter
mov [ebp-8], ebx ; second local variable = ebx
. . .
. . .
leave
retn 16

```

The next program displays a message periodically. The timer is implemented with USER32.DLL's SetTimer function, which receives the address of a stdcall callback.

#### Listing 3.4. Program using stdcall functions - code and data sections only

```

entry start

section '.data' data readable writeable
caption db 'First Win32 program',0
Message db 'Hello World!',0
counter dd 10
timerID dd ?
msg MSG

section '.code' code readable executable

ConcatStrings:
push ebp
mov ebp, esp
push esi
push edi
; param1 - [ebp+8]
; param2 - [ebp+12]
; param3 - [ebp+16]
invoke strlen, [ebp+8]
mov ecx, eax
mov esi, [ebp+8]
mov edi, [ebp+16]
rep movsb
invoke strlen, [ebp+12]
mov ecx, eax
mov esi, [ebp+12]
rep movsb
pop edi
pop esi
leave
retn 12

StdcallProcSample:
push ebp
mov ebp, esp
sub esp, 8
push ebx
; previous ebp - [ebp]
; return address [ebp+4]
; param1 - [ebp+8]
; param2 - [ebp+12]
; local var 1 - [ebp-4]
; local var 2 - [ebp-8]
invoke strlen, [ebp+8]
mov [ebp-4], eax
invoke strlen, [ebp+12]
add [ebp-4], eax
inc dword [ebp-4]
invoke GetProcessHeap
mov [ebp-8], eax
invoke HeapAlloc, eax, HEAP_ZERO_MEMORY, [ebp-4]
mov [ebp-4], eax
stdcall ConcatStrings, [ebp+8], [ebp+12], [ebp-4]
invoke MessageBox, 0, [ebp-4], [ebp+12], 0
invoke HeapFree, [ebp-8], 0, [ebp-4]
pop ebx
leave
retn 8

start:
stdcall StdcallProcSample, Message, caption
invoke ExitProcess, 0

```

The program contains two stdcall functions. ConcatStrings concatenates two strings into a destination buffer. StdcallProcSample uses it to combine the two strings supplied as parameters and then displays the result.

StdcallProcSample obtains the two string lengths with strlen. It allocates a buffer large enough for the combined string using HeapAlloc.

HeapAlloc receives a heap handle, allocation flags, and buffer size. The process's primary heap handle is obtained with GetProcessHeap. The source uses HEAP\_ZERO\_MEMORY to request a zero-filled buffer and says HeapAlloc returns the allocated buffer address on success.

The first instructions that construct a stack frame are called the function prologue; the final instructions that dismantle it are the epilogue. Because writing these sequences is repetitive, FASM's Win32 include files provide the `proc` and `endp` macros.

The `proc` macro creates the function prologue and lets the programmer assign names to parameters and local variables instead of writing explicit `[ebp+/-N]` offsets. `endp` generates the epilogue and ends the scope of those names.

### Listing 3.5. Using `proc` and `endp`

```
proc MyFunc param1, param2
local lvar1:DWORD
local lvar2:DWORD
mov eax, [param1] ; mov eax, [ebp+8]
mov ebx, [param2] ; mov ebx, [ebp+12]
mov [lvar1], eax ; mov [ebp-4], eax
mov [lvar2], ebx ; mov [ebp-8], ebx
ret
endp
```

The final `RET` is required for `endp` to generate the epilogue. Without it, no assembly error occurs, but the expected epilogue and return are not generated. `endp` finds the final `RET` and replaces it with `LEAVE/RETN`.

The same program is then rewritten with `proc/endp`.

### Listing 3.6. Program using `stdcall` functions with `proc/endp`

```
section '.code' code readable executable

proc ConcatStrings str1, str2, str3
push esi
push edi
invoke strlen, [str1]
mov ecx, eax
mov esi, [str1]
mov edi, [str3]
rep movsb
invoke strlen, [str2]
mov ecx, eax
mov esi, [str2]
rep movsb
pop edi
pop esi
ret
endp

proc StdcallProcSample str1, str2
local buff:DWORD
local hProcHeap:DWORD
invoke strlen, [str1]
mov [buff], eax
invoke strlen, [str2]
add [buff], eax
inc dword [buff]
invoke GetProcessHeap
mov [hProcHeap], eax
invoke HeapAlloc, eax, HEAP_ZERO_MEMORY, [buff]
mov [buff], eax
stdcall ConcatStrings, [str1], [str2], [buff]
invoke MessageBox, 0, [buff], [str2], 0
invoke HeapFree, [hProcHeap], 0, [buff]
ret
endp

start:
stdcall StdcallProcSample, Message, caption
invoke ExitProcess, 0
```

The source concludes that `proc/endp` make the functions easier to read.

It says the same stack-frame principles also apply to `CDECL` and `PASCAL`. In `CDECL`, parameters are passed in reverse order and the caller removes them from the stack. In `PASCAL`, parameters are passed in forward order and the callee removes them.

A programmer does not have to use `EBP` as the frame pointer and may choose another register or no dedicated frame pointer at all. However, the source repeats that `EBX`, `ESI`, `EDI`, and `EBP` must remain unchanged across a `stdcall` function, so any of those registers used by the function must be saved and restored.

## 3.1.8. Summary

This section introduced the fundamentals of ring-3 Win32 programming. The practical portion built a simple Win32 program and then examined the `stdcall` calling convention in detail. The next section moves to a more detailed study of user-mode Win32 programming. The source says the complete example source code is on the accompanying CD in the `part3` folder.

## 3.2. Programming in Ring 3

The previous section introduced the fundamentals of assembly programming under 32-bit Windows and noted that most executable components run at protection level 3. The source points out that the ordinary programs users see on the screen execute at this third privilege level.

Section 3.2 presents the basic principles for creating several kinds of programs that execute at privilege level 3.

### 3.2.1. General Overview

When programming for Windows, both in ring 3 and ring 0, API functions supplied by the operating system are called frequently. Almost all ring-3 API functions use the `stdcall` convention: parameters are passed on the stack in reverse order and the called function removes them.

Because it is easy to confuse argument order during frequent API calls, FASM is supplied with include files containing macros that simplify these calls.

FASM provides two such macros: `stdcall` and `invoke`. Both accept a function and its parameters. The source says the difference is how the function address is supplied: `stdcall` expects the function address itself, while `invoke` expects an address containing a pointer to the function.

#### Calling MessageBox with stdcall

```
stdcall [MessageBox],0, Message, caption,0
```

#### Calling MessageBox with invoke

```
invoke MessageBox,0, Message, caption,0
```

The distinction can be convenient when a function address, or a pointer to one, is held in a register.

FASM also supplies include files containing values for most constants used with the Win32 API. Including `win32a.inc` provides ANSI definitions; `win32w.inc` provides Unicode definitions.

Within the native Win32 subsystem, the source divides ordinary ring-3 programs into two types:

1. Console programs.
2. GUI programs.

Their main distinction is the way they interact with the user. Console programs use a console; GUI programs use a graphical interface. The difference is largely formal: a console program can create windows, and a GUI program can allocate a console.

### 3.2.2. Working with Objects

To work with an object in Windows, a program normally obtains an object descriptor, or handle. A handle is described as an ordinary numeric value that identifies an entry in a table describing objects.

It divides Windows objects into two broad groups: kernel (system) objects and user objects. Kernel objects include files, processes, threads, synchronization objects, and other objects directly or indirectly involved in normal system operation. User objects are mostly graphical-subsystem objects such as windows and controls.

Each process has its own handle table describing the system objects to which that process has obtained some level of access. An entry contains information such as the object and access level. A handle can therefore be described as, for example, a handle with full access to a file.

Because each process has its own handle table, the same numeric handle value can identify a different object in another process or be invalid there.

The source presents user-object handles as simpler: user objects are available system-wide and their handles are common across processes.

### 3.2.3. Working with Files

File I/O is one of the key aspects of Windows programming. Many Windows devices use a file-like interface: files, directories, COM and LPT ports, storage devices, some peripheral devices, and certain system objects. Even where a device is not literally file-structured, Windows often abstracts requests as file operations.

The principal API functions named by the source are `CreateFile`, `ReadFile`, and `WriteFile`, which it says reside in `KERNERL32.DLL`.

*Editorial note: KERNERL32.DLL is printed in the source. The usual library name is KERNEL32.DLL, but the source spelling is retained here as a noted typo.*

`CreateFile` can create files and also open named file-like objects. The source lists files, pipes, mailslots, disk devices, consoles, directories, and any named device that supports file-style access.

The source lists the seven `CreateFile` parameters as:

1. File/object name (LPCTSTR).

2. Requested access (DWORD).
3. Sharing mode (DWORD).
4. Security attributes (LPSECURITY\_ATTRIBUTES).
5. Requested action (DWORD).
6. Additional flags and attributes (DWORD).
7. Template-file handle (HANDLE).

The first parameter names the object. The string may be ANSI or Unicode. The second parameter specifies access such as `GENERIC_READ`, `GENERIC_WRITE`, or a combination. The third specifies what access other programs may obtain while the object is open; the source gives `FILE_SHARE_READ` and `FILE_SHARE_WRITE` as examples. The fourth specifies security attributes; this section uses zero to request the system defaults.

For the fifth parameter the source introduces these action codes:

- **CREATE\_NEW.** Create a new file; fail if it already exists.
- **CREATE\_ALWAYS.** Create a new file, replacing an existing file if necessary.
- **OPEN\_EXISTING.** Open an existing file; fail if it does not exist.
- **OPEN\_ALWAYS.** Open the file if it exists, otherwise create it.

For devices and other objects that cannot be created or deleted, the source says `OPEN_EXISTING` must be used. The sixth parameter controls additional open attributes. The seventh may provide a template-file handle for `CREATE_NEW`, `CREATE_ALWAYS`, or `OPEN_ALWAYS`; those parameters are not explored further here.

On success, `CreateFile` returns a handle. On failure it returns `INVALID_HANDLE_VALUE`, which the source identifies as -1 or `FFFFFFFFh`.

Because the NT family is network-oriented, the source says a program can open file objects on other systems by using a name such as `\\remote-computer\object`. It calls this a full object/file name. Local paths may also use ordinary names such as `C:\folder\file.ext`.

The source says the local computer can be written explicitly or as a dot in a device path. It gives `\\.\\D:\File.dat`, `\\.\\.\PHYSICALDRIVE0`, and `\\.\\D:` as examples, including direct access to a physical or logical disk as a large file-like object.

`OpenFile` is described as a simpler wrapper used only for opening files or devices, ultimately reducing to `CreateFile`.

`ReadFile` takes five parameters in the source: a file handle, buffer pointer, requested byte count, pointer to the number of bytes actually read, and an `OVERLAPPED` pointer for asynchronous operations. `WriteFile` takes the analogous five parameters for writing.

*Editorial note: in the prose explaining `ReadFile`, the source calls the third parameter a pointer to a `DWORD` receiving the actual byte count, even though the immediately preceding parameter list correctly identifies parameter 3 as the number of bytes requested and parameter 4 as the pointer receiving the actual count.*

After the operation is finished, the handle should be closed with `CloseHandle` so other programs can use the object. `CloseHandle` accepts one handle and can close handles to system objects generally, not only files.

The source also mentions `DeviceIoControl` as a more universal and flexible function for communicating with files and devices.

### 3.2.4. Handling API Function Errors

API functions can fail, and a program often needs the reason. `GetLastError` takes no parameters and returns the most recent error code in `EAX`. `SetLastError` accepts one numeric code and sets the thread's last-error value in the source's description.

### 3.2.5. Console Programs

Win32 console programs must not be confused with old DOS programs. Console programs are full 32-bit Windows programs and can use normal Windows services. DOS programs are described as 16-bit real-mode programs running inside the NTVDM virtual DOS machine.

A console is attached automatically when a console program starts. A GUI program that wants a console must create one with `AllocConsole`. The source presents this as the principal formal distinction between the two program types.

To work with the console, a program needs handles. The source gives two methods: `GetStdHandle` and `CreateFile`.

Historically, two handles are used: one for input and one for output. `GetStdHandle` returns an input handle for `STD_INPUT_HANDLE` and an output handle for `STD_OUTPUT_HANDLE`. With `CreateFile`, the source says the object name is `CON`, requested access is `GENERIC_READ` or `GENERIC_WRITE`, and the action must be `OPEN_EXISTING`.

Console I/O may be performed with `ReadFile/WriteFile` or `ReadConsole/WriteConsole`. The important distinction is that `ReadConsole/WriteConsole` also have Unicode twins, `ReadConsoleW/WriteConsoleW`.

It describes the parameter sets as handle, buffer pointer, requested count, pointer receiving the actual count, and a final reserved parameter for `ReadConsole/WriteConsole` that must be zero.

WriteConsole/WriteFile output immediately. ReadConsole/ReadFile wait for Enter and place the entered line into the supplied buffer, including carriage-return and line-feed characters.

### 3.2.6. GUI Programs

GUI programs form the basis of the Windows user interface. Almost everything visible on the Windows screen is produced by GUI programs.

The central concept is the window. Application windows, buttons, edit controls, and multiline edit controls are all windows in this model. Windows are not kernel objects and attributes most of the GUI implementation to USER32.DLL in user mode.

Each window has two key properties: a class name and a window procedure. The class defines basic characteristics; the window procedure processes messages sent to the window. Messages are queued, and each thread has its own message queue.

Before creating a window, the program creates/registers a class with RegisterClass. The function receives a pointer to WNDCLASS. The source gives this structure:

#### WNDCLASS structure as shown by the source

```
Struct WNDCLASS
style dd ?
lpfnWndProc dd ?
cbClsExtra dd ?
cbWndExtra dd ?
hInstance dd ?
hIcon dd ?
hCursor dd ?
hbrBackground dd ?
lpszMenuName dd ?
lpszClassName dd ?
ends
```

The source highlights three style values: CS\_HREDRAW redraws when the width changes, CS\_VREDRAW redraws when the height changes, and CS\_GLOBALCLASS creates a globally usable class.

lpfnWndProc points to the window procedure. cbClsExtra and cbWndExtra reserve extra class/window data and are zero in the examples. hInstance names the module containing the window procedure and must be set for a global class in the source's description. hIcon and hCursor provide the icon and cursor. hbrBackground supplies the background brush/color, lpszMenuName the menu name, and lpszClassName the class name.

The window procedure is a callback invoked by the system. It follows stdcall and receives four parameters: window handle, message code, wParam, and lParam.

DefWindowProc is the default/general window procedure. A program can process only the messages it cares about and pass all others to DefWindowProc to obtain standard window behavior.

Once the class exists, CreateWindowEx creates the window. The source lists twelve parameters but numbers them 5 through 16: extended style, class name, window title, window style, X, Y, width, height, parent handle, menu handle, module handle, and a pointer passed with WM\_CREATE.

*Editorial note: the CreateWindowEx parameter list begins at 5 and ends at 16 in the printed source, even though it is presented as the function's complete parameter list. The numbering is retained as a source artifact rather than silently renumbered.*

After creating the window, a program enters a message loop. A message conceptually is described as a structure carrying a message code, wParam, and lParam. The code identifies an action such as resize, movement, repaint, or key press; wParam and lParam carry message-specific data.

Messages can be read with GetMessage or PeekMessage. GetMessage receives a MSG pointer, optional window handle, minimum message code, and maximum message code. The source gives MSG as:

#### MSG structure as shown by the source

```
struct MSG
hwnd dd ?
message dd ?
wParam dd ?
lParam dd ?
time dd ?
pt POINT
ends
```

hwnd is the destination window; message is the code; wParam and lParam carry associated data; time records when the message entered the queue; pt records the cursor coordinates at that moment. The source treats the first four fields as most important.

If GetMessage receives a window handle, it selects messages for that window; zero selects the first matching message in the thread queue. The third and fourth parameters filter by message-code range. GetMessage removes the selected message from the queue.

PeekMessage has the same first four parameters plus a fifth controlling whether the message remains in the queue: PM\_NOREMOVE leaves it and PM\_REMOVE removes it.

The major difference is blocking behavior. GetMessage waits when the queue is empty; PeekMessage returns immediately. PeekMessage returning zero means no suitable message was available. GetMessage returning zero means WM\_QUIT was received and the message loop should end.

### 3.2.7. Dynamic-Link Libraries

Dynamic-link libraries are another important ring-3 component. A DLL is described as broadly similar to an EXE, with the major additional feature being an export table containing names and addresses of exported functions.

A DLL has an entry point. Code there usually prepares state required by exported functions. Control reaches the entry point when the library is loaded into a process and under other DLL lifecycle events described below.

DLLs save physical memory because pages containing the library can be shared among processes. If a 64 KB DLL is mapped into three processes, it can consume 64 KB of physical memory while occupying 64 KB of virtual address space in each process.

Because writable shared data could let one process affect another, the source describes DLL pages as copy-on-write. On the first write to a shared page, a private copy is created for the writing process, so its modifications no longer affect other processes.

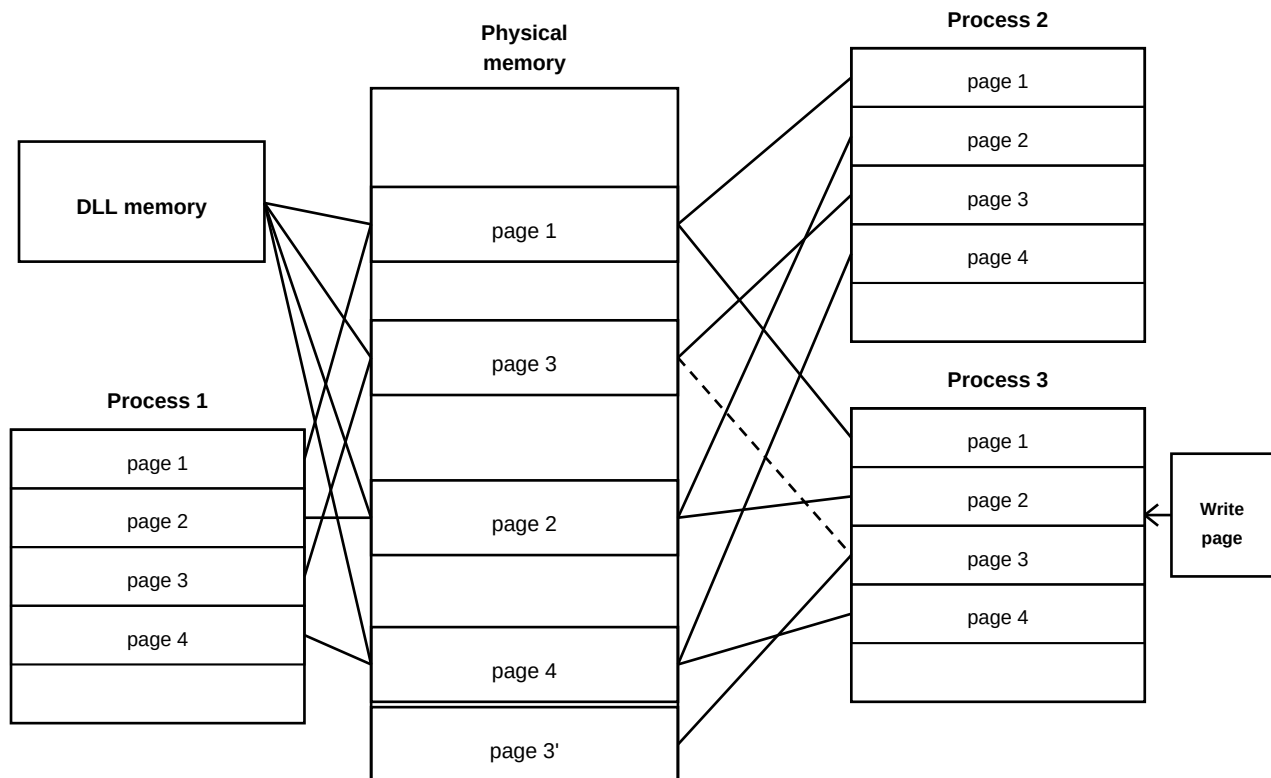


Figure 3.2. Example of a DLL mapped into physical and virtual memory

In the source figure, one DLL is mapped into three processes. Processes 1 and 2 continue sharing all four physical pages. Process 3 writes to page 3, so a private copy, shown as page 3', is created and mapped only into process 3.

Relocations are especially important for DLLs. An ordinary EXE loaded early in process startup will almost always find its preferred image base free, whereas a DLL is more likely to find that preferred range occupied. If the DLL must be loaded elsewhere, the loader uses relocation information to adjust memory references.

The source distinguishes static and dynamic importing. Static import is what the earlier MessageBox and ExitProcess example used: the DLL and function name (or less commonly an ordinal) are recorded in the import table, and the loader performs the work. FASM's library and import macros build this table.

Dynamic import uses LoadLibrary and GetProcAddress. It requires more work by the programmer, but allows a DLL to be loaded only when needed and unloaded with FreeLibrary when no longer required.

Functions may be exported by name or ordinal. LoadLibrary takes the DLL filename and returns a module handle. A module handle is effectively the module's load base address. GetProcAddress takes the module handle and a function name or ordinal and returns the exported function address. FreeLibrary takes the DLL handle and unloads the module from the process.

Unlike an ordinary program, a DLL entry point is also called during unload and thread/process attach/detach events. The entry-point function uses stdcall and receives three parameters: DLL handle, event code, and event-specific information.

The event codes listed are:

1. DLL\_PROCESS\_ATTACH - the library has been loaded into the process.

2. DLL\_THREAD\_ATTACH - a new thread has been created in the process.
3. DLL\_THREAD\_DETACH - a thread has been destroyed in the process.
4. DLL\_PROCESS\_DETACH - the library is about to be unloaded from the process.

The meaning of the third parameter depends on the event. For DLL\_PROCESS\_ATTACH, the source says zero indicates dynamic loading and nonzero static loading. For DLL\_PROCESS\_DETACH, zero indicates unloading through FreeLibrary and nonzero unloading because the process is terminating.

### 3.2.8. Exception Handling in a Program

Windows provides Structured Exception Handling (SEH) for handling exceptions in ring 3. A program can define its own handler for errors that occur during execution.

When an exception can be handled in user mode, control is transferred to the current SEH handler. The source gives divide error (#DE), general protection (#GP), and page fault (#PF) as examples.

SEH supports multiple handlers arranged as a chain. If one cannot handle the exception, it passes the search to the next. If the final handler does not handle it, the source says control reaches UnhandledExceptionFilter, the standard handler that displays an error and terminates the application depending on the error.

The source suggests SEH can be useful when using untrusted code, accessing memory that may become invalid, processing file/data formats that may be corrupted or undefined, and handling unforeseen errors.

The SEH chain is represented as a singly linked list of \_EXCEPTION\_REGISTRATION structures:

#### **\_EXCEPTION\_REGISTRATION**

```
struct _EXCEPTION_REGISTRATION
prev dd ?
handler dd ?
ends
```

prev points to the next registration record in the chain and handler points to the handler function. To install a handler, software adds a new registration record at the head of the chain.

Each thread has a TEB (Thread Environment Block). The first DWORD of the TEB points to the first \_EXCEPTION\_REGISTRATION record. FS references the TEB, so the head pointer is obtained from [FS:0].

#### **Installing an SEH registration record**

```
push exception_handler
push dword [FS:0]
mov [FS:0], esp
```

The two pushes create the registration record on the stack: one DWORD contains the previous chain head and the other the new handler address. ESP then points to the new record, and [FS:0] is updated to that value.

The source explicitly asks why the registration structure is created on the stack instead of as a global variable and answers that Windows validates the location of the registration record; if it is not on the stack, the source says a fatal exception is generated.

*Historical note: this is the classic 32-bit FS:[0] SEH model presented by the source. The rebuild preserves that model rather than replacing it with later Windows exception-handling mechanisms.*

The handler function uses the cdecl calling convention. Cdecl differs from stdcall in that the caller, not the callee, removes the parameters.

The handler receives four parameters:

1. Pointer to EXCEPTION\_RECORD.
2. Pointer to the registration-chain node from which the handler was invoked.
3. Pointer to CONTEXT, containing the thread register state at the time of the exception.
4. Pointer associated with nested-exception dispatch; the source labels this as EXCEPTION\_RECORD.

The first and third are the important ones for the example. CONTEXT contains the register values. EXCEPTION\_RECORD describes the exception:

#### **EXCEPTION\_RECORD structure as shown by the source**

```
struct EXCEPTION_RECORD
ExceptionCode dd ?
ExceptionFlags dd ?
ExceptionRecord dd ?
ExceptionAddress dd ?
NumberParameters dd ?
ExceptionInformation dd EXCEPTION_MAXIMUM_PARAMETERS dup (?)
ends
```

**Table 3.1. Common exceptions**

| Exception code             | Description                                               |
|----------------------------|-----------------------------------------------------------|
| EXCEPTION_ACCESS_VIOLATION | Attempt to read/write memory without the required access. |

| Exception code                     | Description                                                   |
|------------------------------------|---------------------------------------------------------------|
| EXCEPTION_BREAKPOINT               | A breakpoint was reached.                                     |
| EXCEPTION_ILLEGAL_INSTRUCTION      | Attempt to execute an unsupported instruction.                |
| EXCEPTION_IN_PAGE_ERROR            | Attempt to access a page that is absent and cannot be loaded. |
| EXCEPTION_INT_DIVIDE_BY_ZERO       | Attempt to perform integer division by zero.                  |
| EXCEPTION_NONCONTINUABLE_EXCEPTION | Attempt to resume a thread after a noncontinuable exception.  |
| EXCEPTION_PRIV_INSTRUCTION         | Attempt to execute a privileged instruction.                  |
| EXCEPTION_SINGLE_STEP              | One instruction was executed during tracing.                  |
| EXCEPTION_STACK_OVERFLOW           | Stack overflow.                                               |

ExceptionCode contains the exception code. ExceptionFlags indicates whether the exception is continuable; the source says EXCEPTION\_NONCONTINUABLE is set when it is not. Attempting to resume from such an exception produces EXCEPTION\_NONCONTINUABLE\_EXCEPTION.

ExceptionRecord links EXCEPTION\_RECORD structures for nested exceptions. ExceptionAddress identifies where the exception occurred. NumberParameters gives the number of extra values stored in ExceptionInformation.

For EXCEPTION\_ACCESS\_VIOLATION, the source says the first element of ExceptionInformation identifies the operation type (read/write) and the second contains the virtual address that was accessed.

A handler examines the type and cause of the exception and, if possible, removes that cause. After successful handling it should return ExceptionContinueExecution (0). If the response requires leaving nested protected blocks, the source refers to an unwind operation but does not cover it here.

If the handler cannot handle the exception, it returns ExceptionContinueSearch (1), after which Windows calls the next handler in the chain.

### 3.2.9. Practice

The source states that theory should be reinforced with practical examples. Only the most important fragments are printed; the complete programs were supplied in the part3 folder on the accompanying CD.

#### 3.2.9.1. Creating a Console Application

The console example evaluates the formula  $(a + 3) * 6 + a * (100 - a)$ , where a is entered from the console.

First the program obtains the standard input and output handles, prints a prompt, and reads text from the console.

##### Listing 3.7. Obtaining handles and displaying a console prompt

```
invoke GetStdHandle, STD_INPUT_HANDLE
mov [InputHandle], eax
invoke GetStdHandle, STD_OUTPUT_HANDLE
mov [OutputHandle], eax
invoke WriteConsole, [OutputHandle], inputa, 8,Written,0
invoke ReadConsole, [InputHandle], a_str, MAX_VALUE_LENGTH, Readed,0
```

ReadConsole waits for text input and Enter, then copies the entered string into the buffer addressed by a\_str.

The line ending must then be removed and the text converted to an integer.

##### Listing 3.8. Reading/converting the number

```
mov edi, a_str
call GetZSLength
sub eax, 2
mov ebx, eax
add eax, a_str
mov word [eax], 0
mov esi, a_str
call STR_to_DWORD_EX
```

GetZSLength obtains the string length. Two final characters are cleared because ReadConsole stores carriage-return and line-feed characters in the buffer. STR\_to\_DWORD\_EX receives the string pointer in ESI and returns the converted number in EAX.

##### Listing 3.9. Calculating the formula

```
push eax
add eax, 3
imul eax, 6
mov ebx, eax ; ebx = (a+3)*6
mov edx, 100 ;
pop eax
sub edx, eax
imul eax, edx ; eax = a*(100-a)
add eax, ebx ; eax = res
```

The original value is saved temporarily on the stack, the first part of the expression is calculated, then the second part, and finally the two are added.

##### Listing 3.10. Converting the result to text and writing it to the console

```
mov esi, a_str
mov ebx, 10
```

```

call dword_to_STR
mov edi, message
call GetZSLength
invoke WriteConsole, [OutputHandle], message, eax,Written,0
invoke ReadConsole, [InputHandle], a_str, 1,0,0

```

dword\_to\_STR receives the number in EAX and a destination buffer pointer in ESI. After output, the final ReadConsole waits for Enter so the console window does not close immediately.

### 3.2.9.2. Creating a Windowed Application

The GUI example creates a main window with two buttons. Clicking the first displays a message; clicking the second terminates the program.

The first step is registering the window class.

#### Listing 3.11. Registering a new window class

```

invoke GetModuleHandle,0
mov [hInst], eax
mov [wc.style], CS_HREDRAW + CS_VREDRAW + CS_GLOBALCLASS
mov [wc.lpfnWndProc], WndProc
mov [wc.cbClsExtra], 0
mov [wc.cbWndExtra], 0
mov [wc.hInstance], eax
invoke LoadIcon,0,IDI_APPLICATION
mov [wc.hIcon], eax
invoke LoadCursor,0, IDC_ARROW
mov [wc.hCursor], eax
mov [wc.hbrBackground], COLOR_BACKGROUND+1
mov dword [wc.lpszMenuName], 0
mov dword [wc.lpszClassName], szClassName
invoke RegisterClass, wc

```

The program obtains the current module handle, fills WNDCLASS, obtains the icon and cursor handles, and passes the structure to RegisterClass.

#### Listing 3.12. Creating the main window and buttons

```

invoke CreateWindowEx,0,szClassName,szTitleName, WS_OVERLAPPEDWINDOW, 50,50, 300, 250, 0,0,[hInst],0
mov [main_hwnd], eax
invoke CreateWindowEx, 0, button_class, AboutTitle, WS_CHILD, 50, 50, 200, 50, [main_hwnd],0,[hInst],0
mov [AboutBtnHandle], eax
invoke CreateWindowEx, 0, button_class, ExitTitle, WS_CHILD, 50, 150, 200, 50, [main_hwnd],0,[hInst],0
mov [ExitBtnHandle], eax

```

*Editorial note: Listing 3.12 stores the second button handle in ExitBtnHandle, while subsequent code refers to ExitBtnHandle. The printed identifiers are preserved rather than silently unified.*

WS\_OVERLAPPEDWINDOW creates a conventional top-level window with a title bar and standard controls. The two buttons use the predefined BUTTON class, WS\_CHILD, and the main-window handle as their parent.

#### Listing 3.13. Message-retrieval loop

```

invoke ShowWindow, [main_hwnd],SW_SHOWNORMAL
invoke UpdateWindow,[main_hwnd]
invoke ShowWindow, [AboutBtnHandle],SW_SHOWNORMAL
invoke ShowWindow, [ExitBtnHandle],SW_SHOWNORMAL
msg_loop:
invoke GetMessage, msg,0,0,0
cmp eax, 0
je end_loop
invoke TranslateMessage, msg
invoke DispatchMessage, msg
jmp msg_loop
end_loop:

```

ShowWindow displays the windows and UpdateWindow refreshes the main window. GetMessage retrieves messages. TranslateMessage converts key messages into character-input messages, and DispatchMessage calls the window procedure belonging to the destination window. This matters because different windows and controls may have different system-defined procedures.

The window procedure is declared as stdcall through FASM's proc macro.

#### Listing 3.14. Window message procedure

```

proc WndProc hwnd, wmsg, wparam, lparam
pushad
cmp [wmsg], WM_DESTROY
je .wmdestroy
cmp [wmsg], WM_COMMAND
jne .default
mov eax, [wparam]
shr eax, 16
cmp eax, BN_CLICKED
jne .default
mov eax, [lparam]
cmp eax, [AboutBtnHandle]
je .about
cmp eax, [ExitBtnHandle]
je .wmdestroy
.default:
invoke DefWindowProc, [hwnd],[wmsg],[wparam], [lparam]
jmp .finish

```

```
.about:
invoke MessageBox, 0, AboutText, szTitleName,0
jmp .finish
.wmdestroy:
invoke ExitProcess,0
.finish:
mov [esp+28], eax
popad
ret
endp
```

The central logic handles WM\_COMMAND. Child windows send WM\_COMMAND to their parent for relevant events. When a BUTTON control is clicked, the source says BN\_CLICKED appears in the high part of wParam and the button handle appears in lParam. The About button displays a message; the Exit button terminates the process.

### 3.2.9.3. Creating a DLL

The DLL example moves the earlier formula calculation into an exported function and modifies the console program to load and call that function dynamically.

#### Listing 3.15. DLL code section

```
section '.code' code readable executable
proc DllEntryPoint hinstDLL,fdwReason,lpvReserved
    mov eax, TRUE
    ret
endp
proc CalcValue Value
    push ebx
    push edx
    push eax
    add eax, 3
    imul eax, 6
    mov ebx, eax ; ebx = (a+3)*6
    mov edx, 100 ;
    pop eax
    sub edx, eax
    imul eax, edx ; eax = a*(100-a)
    add eax, ebx ; eax = res
    pop edx
    pop ebx
    ret
endp
section '.edata' export data readable
    export 'DLL_sample.DLL', CalcValue,'CalcValue'
section '.reloc' fixups data discardable
```

*Editorial note: CalcValue declares a parameter named Value, but the printed body operates directly on EAX and never loads the declared stack parameter. The code is preserved exactly rather than silently corrected.*

The source identifies the export table as the key DLL-specific element and uses FASM's export macro to build it.

#### Listing 3.16. Code using a function loaded from the DLL

```
invoke WriteConsole, [OutputHandle], inputa, 8,Writed,0
invoke ReadConsole, [InputHandle], a_str, MAX_VALUE_LENGTH, Readed,0
mov edi, a_str
call GetZSLength
sub eax, 2
mov ebx, eax
add eax, a_str
mov word [eax], 0
mov esi, a_str
call STR_to_DWORD_EX
push eax
invoke LoadLibrary, DLL_name ; eax = DLL handle
invoke GetProcAddress, eax, FUNC_name ; eax = func address
mov ebx, eax ; ebx = eax
pop eax ; eax = value
stdcall ebx, eax ; call func
mov esi, a_str
mov ebx, 10
call dword_to_STR
mov edi, message
call GetZSLength
invoke WriteConsole, [OutputHandle], message, eax,Writed,0
invoke ReadConsole, [InputHandle], a_str, 1,0,0
```

The program loads the DLL with LoadLibrary, obtains CalcValue with GetProcAddress, stores the returned function address in EBX, and invokes it through the stdcall macro.

### 3.2.9.4. Using SEH

The SEH demonstration installs a custom exception handler and deliberately generates two errors: integer division by zero and invalid memory access.

#### Listing 3.17. Exception handler

```
proc ExceptionHandler c ExceptionRecord, EstablisherFrame, ContextRecord,
    DispatcherContext
    push ebx
    push esi
    mov ebx, [ExceptionRecord]
    virtual at ebx
```

```

    .ebx EXCEPTION_RECORD
end virtual
mov esi, [ContextRecord]
virtual at esi
.esi CONTEXT
end virtual
cmp [.ebx.ExceptionCode], EXCEPTION_ACCESS_VIOLATION
je .AccessViolation
cmp [.ebx.ExceptionCode], EXCEPTION_INT_DIVIDE_BY_ZERO
je .DivideError
.ContinueSearch:
mov eax, ExceptionContinueSearch
jmp .exit
.AccessViolation:
invoke MessageBox, 0, szAccessViolationError, szTitle, 0
add [.esi.regEip], 2
mov eax, ExceptionContinueExecution
jmp .exit
.DivideError:
invoke MessageBox, 0, szDivideByZeroError, szTitle, 0
add [.esi.regEip], 2
mov eax, ExceptionContinueExecution
.exit:
pop esi
pop ebx
ret
endp

```

The `c` after the proc name tells the FASM proc macro to generate a cdecl-compatible function. The source says an exception callback should preserve EBX, ESI, and EDI and preferably avoid changing any registers unnecessarily.

The handler puts the `EXCEPTION_RECORD` and `CONTEXT` pointers into EBX and ESI, identifies the exception type, and advances the saved EIP by two bytes so execution resumes after the faulting instruction in these two specific test cases.

The source says the exception codes used here and those in Table 3.1 are defined in `DDK\ntdefs.inc`.

#### Listing 3.18. Registering and testing the exception handler

```

start:
push ExceptionHandler
push dword [fs:0000h]
mov dword [fs:0000h], esp
xor eax, eax
div eax
xor eax, eax
mov eax, [eax]
pop eax
mov [fs:0000h], eax
pop eax
invoke MessageBox, 0, szMessage, szTitle, 0
invoke ExitProcess, 0

```

The handler is installed at the head of the SEH chain. The program then generates divide-by-zero and null-address access exceptions. Finally it restores the previous chain head and continues to the final message and `ExitProcess`.

### 3.2.9.5. Debugging Applications

Assembly development is extremely difficult without a debugger. Bare-metal programs offered few convenient debugging options beyond basic facilities in emulators such as QEMU and Bochs, while Windows user-mode development offered a much wider choice.

It names OllyDebug, IDA Pro, and WinDBG as common ring-3 debuggers. It describes OllyDebug and WinDBG as free and IDA Pro as commercial, while noting IDA's broad functionality. WinDBG could be obtained from Microsoft, and the accompanying CD also contained WinDBG and OllyDebug in a debuggers folder.

*Historical note: product names, licensing, distribution, and availability are translated as they appeared in the 2011 source and are not updated here.*

The `INT3` instruction is highlighted as useful during debugging. Unlike the general `INT n` encoding, `INT3` has a one-byte opcode. `INT3` reliably invokes the breakpoint exception when a debugger has installed the appropriate handler, while the longer `INT 3` form may be affected by security restrictions in some contexts.

For a normal executable, the debugger typically stops at the entry point, allowing breakpoints to be placed before execution continues. For a DLL, catching the exact moment of loading can be less convenient, so the source recommends placing `INT3` in the DLL entry point; when a host application loads the DLL under the debugger, execution stops as soon as control reaches that entry point.

### 3.2.10. Summary

This section introduced key concepts needed for ring-3 Win32 programming. This is only a small part of the subject. Topics not covered include more extensive work with system objects and the graphics subsystem, multithreaded applications, COM and OLE objects, the registry, virtual memory, heaps, and other facilities. The next section turns to ring-0 programming in Win32.

## 3.3. Programming in Ring 0

Section 3.2 covered the fundamentals of user-mode programming on 32-bit Windows. This section examines the basic principles of creating programs that execute at privilege level 0.

Windows kernel code executes at privilege level 0; execution at that level is also called kernel-mode execution. Drivers is described as the principal executable components of the Windows kernel and therefore as a large part of the code that runs in ring 0.

### 3.3.1. Services

Before discussing kernel-mode driver code, the source first explains how drivers are registered with the operating system and started. The first concept introduced is therefore the Windows service mechanism.

A service is a user-mode process that can start and operate without an interactive user logging on. For that reason most services do not have a user interface. The source gives servers as a typical use and notes that services may start during operating-system boot or afterward.

Microsoft terminology also applies the word service to device drivers, even though ordinary services execute in user mode and drivers execute in kernel mode.

The source lists the components used to start and manage a driver:

- **Service Control Manager (SCM).** The system component through which drivers can be installed and started.
- **Service Control Program (SCP).** An ordinary ring-3 program that calls SCM functions to install, start, control, stop, and remove a driver.
- **The driver itself.**

A driver is first registered with `CreateService` and then started with `StartService`. Other SCM functions mentioned by the source are `ControlService`, `DeleteService`, and `CloseServiceHandle`. Access to the SCM is obtained through `OpenSCManager`. These functions are exported by `ADVAPI32.DLL`.

The book does not write a driver-control program at this point. Instead it recommends the then-available open-source `KmdManager` utility, written in MASM and supplied on the accompanying CD.

Near the end of system startup, the source says the SCM process examines `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\`, builds its internal service database, and loads services and device drivers marked for automatic startup.

*Historical note: the service-management workflow is translated from the 2011 source. Specific tool availability and implementation details are historical rather than current guidance.*

### 3.3.2. General Overview

A device driver is a program intended to control some device, which may be physical or virtual. The examples in this section use virtual devices, but the same basic knowledge is useful when writing drivers for physical hardware.

The source lists several categories of kernel-mode driver:

- File-system drivers, which implement I/O to local and network disks.
- Legacy drivers, written for earlier Windows NT versions.
- Video drivers, which implement graphics operations.
- Streaming drivers for audio and video I/O.
- WDM drivers, supporting Plug and Play and power management and historically intended to provide source compatibility across Windows 98/ME and Windows NT.

The book points out that its own examples do not fit neatly into these categories. They are essentially kernel-mode programs that expose a virtual device and provide a controlled path into ring 0.

A driver is structurally a PE-format file, like an EXE or DLL, but it is loaded and executed under different rules. The source suggests thinking of a driver as a kernel-mode DLL for operations unavailable to ring-3 code.

The key difference is that applications do not directly obtain addresses of driver functions and call them. Instead, interaction takes place through the I/O Manager. A driver also cannot directly use user-subsystem objects such as windows, consoles, or GUI controls; it works with system objects such as files, registry objects, processes, and threads.

### 3.3.3. Driver Development Kit (DDK)

The source introduces the Driver Development Kit (DDK) as the collection of tools, documentation, declarations, and examples needed for Windows driver development. It especially values the documentation because it describes internal structures and kernel functions used by drivers.

Although the DDK is aimed at C development, the assembly programmer still needs its documentation, constants, data structures, and examples. C header declarations must be translated into assembler syntax.

The examples in this chapter use FASM include files placed in a DDK directory. The book says these files were supplied on the accompanying CD and intentionally contain only the declarations needed directly or indirectly by the chapter example.

The source also mentions the Windows Driver Kit (WDK) as a broader package for developing drivers for multiple Windows versions.

*Historical note: the source statement that Microsoft no longer distributed the DDK free of charge reflects the book's period and is not treated here as a current product statement.*

### 3.3.4. Thread Context and Interrupt Request Levels

A system normally has many more threads than processors. The illusion of simultaneous execution is created by rapidly switching the processor among threads. During a context switch, the old thread state is saved and the new thread state is restored. If the new thread belongs to another process, the processor must also begin using that process's address translation structures; the source expresses this as loading the appropriate CR3 value.

Drivers often avoid creating their own threads when it is unnecessary because context switching has a cost. Driver code generally executes in one of three contexts:

- The context of the user thread that initiated an I/O request.
- The context of a kernel system thread belonging to the System process.
- As the result of an interrupt, and therefore in the context of whichever thread was current when the interrupt occurred.

If a driver is executing in the context of the requesting user thread, it can directly refer to that process's user virtual address space, subject to the normal kernel rules. In a system-thread context, the source says there is no corresponding user process address space to address directly; system space remains available.

Windows NT assigns interrupt handling priorities called Interrupt Request Levels (IRQLs). The source describes 32 levels from 0 (passive) to 31 (high). It distinguishes IRQL from both thread scheduling priority and the priority mechanism implemented by an interrupt controller.

At PASSIVE\_LEVEL, the source says any kernel function whose documentation permits that level may be called, and pageable memory can be touched. At DPC/dispatch level and above, attempting to access a page that is not resident in physical memory can crash the system because a page fault cannot be serviced safely in that context.

*Editorial note: the IRQL classification in this introductory description is simplified. This translation preserves the source's framing rather than replacing it with a modern driver-model exposition.*

### 3.3.5. Example of a Simple Driver

The first driver example generates three tones through the system speaker immediately after loading and then causes initialization to fail so that Windows unloads the driver.

#### Listing 3.19. Example of the simplest driver

```
format PE Native 4.0
include 'win32w.inc'
include 'DDK\ntstatus.inc'
entry DriverEntry
TIMER_FREQUENCY equ 1193167
OCTAVE equ 2
D0 equ 523 ; C - 523.25 Hz
M1 equ 659 ; E - 659.25 Hz
SOL equ 784 ; G - 783.99 Hz
TONE1 equ TIMER_FREQUENCY/(D0*OCTAVE)
TONE2 equ TIMER_FREQUENCY/(M1*OCTAVE)
TONE3 equ TIMER_FREQUENCY/(SOL*OCTAVE)
section '.code' code readable executable
proc Sound dwPitch
cli
invoke WRITE_PORT_UCHAR, 43h, 10110110b
mov eax, [dwPitch]
out 42h, al
mov al, ah
out 42h, al
; enable speaker
invoke READ_PORT_UCHAR, 61h
or al, 11b
invoke WRITE_PORT_UCHAR, 61h, eax
sti
mov ecx, 5000000h
loop $
cli
; disable speaker
invoke READ_PORT_UCHAR, 61h
and al, 1111100b
invoke WRITE_PORT_UCHAR, 61h, eax
sti
ret
endp
proc DriverEntry DriverObject, RegistryPath
stdcall Sound, TONE1
stdcall Sound, TONE2
stdcall Sound, TONE3
mov eax, STATUS_DEVICE_CONFIGURATION_ERROR
ret
endp
```

```

section '.relocs' fixups readable writeable discardable
section '.idata' import readable writeable
library hal, 'hal.dll'
import hal, \
WRITE_PORT_UCHAR, 'WRITE_PORT_UCHAR', \
READ_PORT_UCHAR, 'READ_PORT_UCHAR'

```

format PE Native 4.0 tells FASM to generate a native PE image suitable for a SYS driver. The program includes the ordinary Win32 definitions plus `ntstatus.inc`, which contains kernel status codes.

The driver entry routine receives two parameters: `DriverObject`, a pointer to the `DRIVER_OBJECT` structure created for the newly loaded driver, and `RegistryPath`, a pointer to a `UNICODE_STRING` describing the registry location containing the driver's configuration parameters.

The Sound procedure demonstrates direct hardware access through I/O ports and through functions exported by `HAL.DLL`. The source does not attempt to teach the PC speaker hardware in this example; the point is that kernel code can perform operations unavailable to ordinary ring-3 applications.

`DriverEntry` returns an `NTSTATUS` value indicating whether initialization succeeded. Returning `STATUS_SUCCESS` leaves the driver loaded. Returning an error causes the system to unload it. This example deliberately returns `STATUS_DEVICE_CONFIGURATION_ERROR` because it performs its one demonstration and has not defined an unload routine.

The relocation section is important because the driver may be loaded at an address other than its preferred base. The source marks the relocation section discardable so it can be removed from memory when no longer required. The final section is the import table, here importing the port-access helpers from `HAL.DLL`.

### 3.3.6. Strings in the Windows Kernel

Kernel-mode string handling differs from the user-mode convention used earlier. Kernel interfaces use Unicode strings and represent them with a `UNICODE_STRING` structure instead of merely passing a pointer to a zero-terminated character array.

#### UNICODE\_STRING structure as printed in the source

```

struct UNICODE_STRING
{
    .Length dw ?
    .MaxiimmLength dw ?
    .Buffer dd ?
}

```

*Editorial note: the printed structure field is `MaxiimmLength`. The surrounding prose calls the field `MaximumLength`. The typo is preserved in the structure.*

**Length** is the current string length in bytes, excluding the terminating two zero bytes. **MaximumLength** is the size of the backing buffer, also in bytes. **Buffer** points to the Unicode data.

Because length information is carried explicitly, a kernel function receiving a `UNICODE_STRING` does not have to scan for the terminating null character and does not need a separate buffer-size parameter for many operations.

The source names several helper routines: `RtlInitUnicodeString`, `RtlUnicodeStringToAnsiString`, `RtlUppcaseUnicodeString`, `RtlAnsiStringToUnicodeString`, `RtlInitAnsiString`, and `RtlCompareUnicodeString`.

### 3.3.7. I/O Subsystem

The I/O subsystem is central to kernel-driver programming. A driver can be viewed loosely as a set of privileged functions, but a user program does not call those functions directly. The I/O Manager acts as an intermediary, building an I/O Request Packet (IRP) and sending it to the driver.

The source identifies four important objects or structures in a typical I/O operation: `DRIVER_OBJECT`, `DEVICE_OBJECT`, `FILE_OBJECT`, and `IRP`. When user mode communicates with kernel mode, a symbolic link is also required so that user code can name the device.

The source uses an analogy: the driver object is the addressee, the device object is the mailbox, the IRP is the letter containing the request, and the file object is the postage stamp that authorizes the request. A file object can represent different granted access rights, such as read-only or write-only access to the same device.

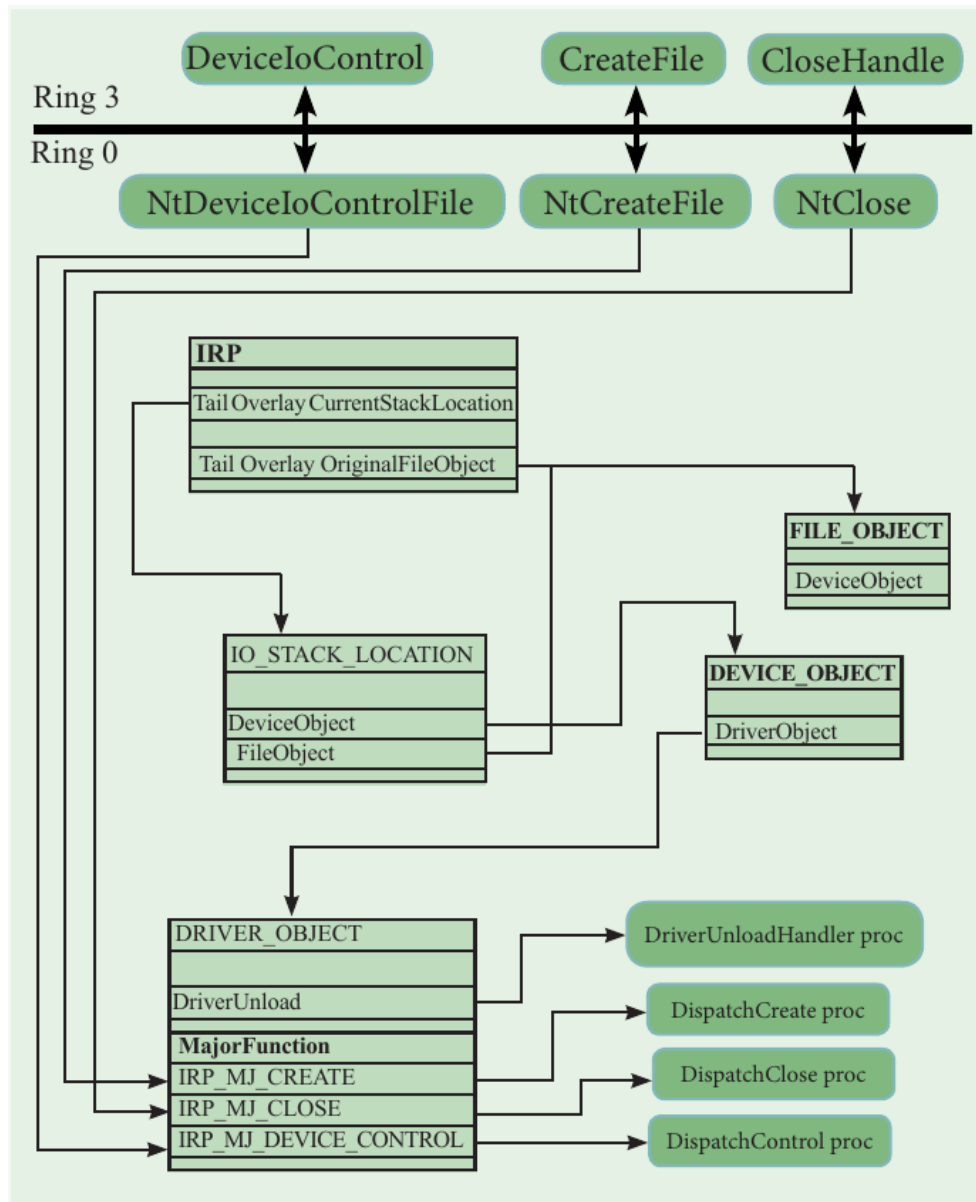


Figure 3.3. General scheme of the Windows NT I/O subsystem

For normal interaction with ring-3 code, the driver creates a device object and a symbolic-link object and defines dispatch routines for the I/O request types it supports.

IoCreateDevice, exported by NTOSKRNL.EXE, creates the device object. Device and driver objects live in the Object Manager namespace, a hierarchical namespace similar in appearance to a file-system directory tree.

The source says most Object Manager directories are not directly visible to user mode. A device object resides under \Device. To make it accessible from user mode, the driver creates a symbolic link in \?? that resolves to the internal device name. It also notes the historical \DosDevices alias.

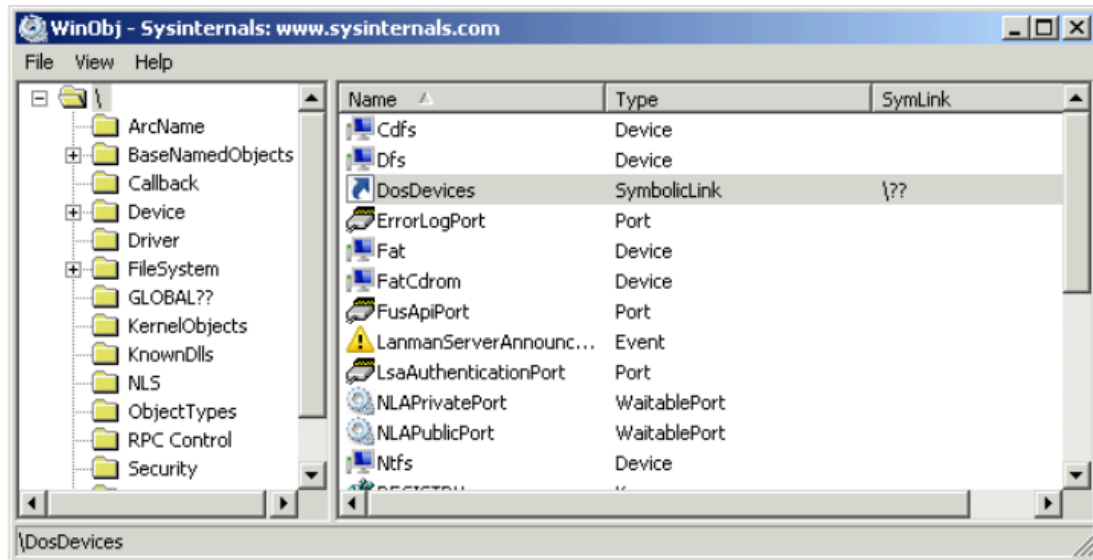


Figure 3.4. Object Manager namespace (original WinObj screenshot from the source; interface labels are already in English).

A driver that exposes this sort of device should at minimum define handlers for `IRP_MJ_CREATE`, `IRP_MJ_CLOSE`, and `IRP_MJ_DEVICE_CONTROL`. `CreateFile/NtCreateFile` causes `IRP_MJ_CREATE`; closing the handle causes `IRP_MJ_CLOSE`; `DeviceIoControl/NtDeviceIoControlFile` causes `IRP_MJ_DEVICE_CONTROL`.

*Editorial note: the prose prints `CloseHandle` once. The standard API name is `CloseHandle`; the typo is noted rather than propagated into code.*

`IRP_MJ_DEVICE_CONTROL` requests are differentiated by an I/O control code (IOCTL). The source divides the 32-bit code into `DeviceType`, `Access`, `Function`, and `Method` fields.

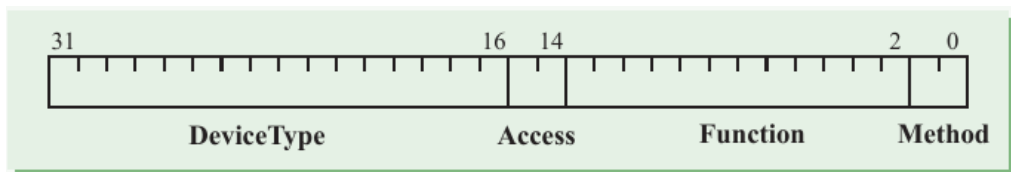


Figure 3.5. I/O control-code format

**DeviceType** is 16 bits. The source describes `0000h-7FFFh` as reserved for Microsoft-defined types and `8000h-FFFFh` as available for custom types. The examples use `FILE_DEVICE_UNKNOWN`.

**Access** is 2 bits: `FILE_ANY_ACCESS` (0), `FILE_READ_ACCESS` (1), `FILE_WRITE_ACCESS` (2), or their combination (3).

**Function** is 12 bits and identifies the operation. The source describes `000h-7FFh` as Microsoft-reserved and `800h-FFFh` as available for custom functions.

**Method** is 2 bits and selects buffer management: `METHOD_BUFFERED` (0), `METHOD_IN_DIRECT` (1), `METHOD_OUT_DIRECT` (2), or `METHOD_NEITHER` (3).

With buffered I/O, the I/O Manager allocates a nonpaged system buffer at least as large as the larger of the user input and output buffers. For an input operation it copies the user input data into that system buffer and places its address in `IRP.AssociatedIrp.SystemBuffer`. The input size is available in `IO_STACK_LOCATION.Parameters.DeviceIoControl.InputBufferLength`.

The driver processes the request and writes any output back into the same system buffer. When the IRP completes, the I/O Manager copies the requested number of bytes from the system buffer to the user output buffer. The driver communicates that byte count through `IRP.IoStatus.Information`.

Buffered I/O is described as the slowest of the three mechanisms because the I/O Manager copies data, but also the safest and simplest for small buffers because user-buffer validation and copying are handled for the driver.

The source then describes direct I/O. It says the user output buffer is locked into physical memory and represented by an MDL whose address is placed in `IRP.MdlAddress`. The driver can then access the described pages independently of the initiating process context. The chapter does not develop this method further.

*Editorial note: the book closely associates direct I/O with DMA in this introductory discussion. Modern driver design distinguishes the I/O Manager's direct-I/O buffering model from whether a particular device actually performs DMA. The translation retains the source's framing.*

With `METHOD_NEITHER`, the I/O Manager passes the user pointers essentially unmanaged: the input pointer appears in `IO_STACK_LOCATION.Parameters.DeviceIoControl.Type3InputBuffer` and the output pointer in `IRP.UserBuffer`. The driver is then responsible for validation, locking, mapping, synchronization, and safe access.

`METHOD_NEITHER` is dangerous: a user thread may pass an invalid pointer, free the backing memory from another thread, or otherwise make the address unsafe while the driver is processing it. An unhandled kernel-mode fault can crash the

system.

The source summarizes a DeviceIoControl request as follows. User mode calls DeviceIoControl, which eventually invokes NtDeviceIoControlFile in kernel mode. The I/O Manager prepares buffering according to the selected method, creates an IRP, and calls the driver's IRP\_MJ\_DEVICE\_CONTROL dispatch routine. After the driver completes the request, the I/O Manager performs any required copying and returns status and data to the caller.

Other ordinary file APIs can generate driver requests as well. For example, ReadFile can produce IRP\_MJ\_READ if the driver supplies the corresponding dispatch routine. Windows abstracts device I/O as operations on file-like objects so applications can use a unified interface.

### 3.3.8. Practice

The practical example creates a driver that returns the contents of the Global Descriptor Table to a ring-3 process. The complete control program and driver source are in the part3 directory on the accompanying CD.

#### 3.3.8.1. Driver Control Program

The control program loads the driver, sends requests to it, and unloads it. Listing 3.20 prints the important parts of the application.

**Listing 3.20. Driver control program**

```
format PE GUI 4.0
. . .
entry start
IOCTL_DEFINE_EX IOCTL_DUMP_GDT, FILE_ANY_ACCESS, 1, METHOD_BUFFERED
section '.data' data readable writable
DriverFileName db 'E:\FASM\PROJECTS\drivers\gdt Dump.sys',0
DriverServiceName db 'gdt Dump',0
DeviceName db '\\.\GDT Dump',0
DriverDisplayName db 'GDT dumper',0
. . .
inputData:
.Offset dd ?
.Size dd ?
FS_Descriptor dq ?
DescriptorInfo:
db 'Base address : '
.BaseAddr db ' '
db 13, 10
db 'Limit : '
.Limit db ' '
db 13, 10
db 'DPL : '
.DPL db ' ',0
align 10h
Buffer db 10000h dup (?)
BufferSize = $-Buffer
section '.code' code readable executable
. . .
LoadDriver:
invoke OpenSCManager, 0, 0, SC_MANAGER_ALL_ACCESS
cmp eax, 0
jz @f
mov [hSCManager], eax
invoke CreateService, eax, DriverServiceName, DriverDisplayName, \
SERVICE_START + SERVICE_STOP + DELETE, SERVICE_KERNEL_DRIVER, \
SERVICE_DEMAND_START, SERVICE_ERROR_IGNORE, DriverFileName, 0, 0, 0, 0
cmp eax, 0
jz @f
mov [hService], eax
invoke StartService, [hService], 0, 0
@@:
ret
UnloadDriver:
invoke ControlService, [hService], SERVICE_CONTROL_STOP, temp
invoke DeleteService, [hService]
invoke CloseServiceHandle, [hService]
invoke CloseServiceHandle, [hSCManager]
ret
start:
call LoadDriver
cmp eax, 0
jnz @f
invoke MessageBox, 0, MsgFail, DriverDisplayName, 0
jmp .exit
@@:
invoke MessageBox, 0, MsgSuccess, DriverDisplayName, 0
invoke CreateFile, DeviceName, GENERIC_READ + GENERIC_WRITE, FILE_SHARE_READ, 0, \
OPEN_EXISTING, 0, 0
mov [hDevice], eax
invoke DeviceIoControl, eax, IOCTL_DUMP_GDT, 0, 0, Buffer, BufferSize, Readed, 0
invoke CreateFile, DumpFileName, GENERIC_WRITE, FILE_SHARE_READ, 0, \
CREATE_ALWAYS, 0, 0
push eax
invoke WriteFile, eax, Buffer, [Readed], Readed, 0
pop eax
invoke CloseHandle, eax
xor eax, eax
mov ax, fs
and al, 11111000b
mov [inputData.Offset], eax
mov [inputData.Size], 8
```

```

invoke DeviceIoControl, [hDevice], IOCTL_DUMP_GDT, inputData, 8, \
    FS_Descriptor, 8, Readed, 0
mov esi, FS_Descriptor
call DecodeDescriptor
push eax
push ecx
mov eax, ebx
mov ebx, 16
mov esi, DescriptorInfo.BaseAddr
call dword_to_STR
pop eax
mov ebx, 16
mov esi, DescriptorInfo.Limit
call dword_to_STR
pop eax
mov ebx, 16
mov esi, DescriptorInfo.DPL
call dword_to_STR
invoke MessageBox, 0, DescriptorInfo, DriverDisplayName, 0
invoke CloseHandle, [hDevice]
.exit:
call UnLoadDriver
push 0
call [ExitProcess]

```

*Editorial note: line continuations in Listing 3.20 are laid out to fit the translated page width. Identifiers, constants, paths, numeric values, and operations are preserved from the source.*

LoadDriver obtains an SCM handle with OpenSCManager, creates a kernel-driver service with CreateService, and starts it with StartService. On failure the function returns EAX = 0; on successful load the example displays a message.

The program then opens the device with CreateFile("\\.\\GDTDump", . . .). The source explains the user-visible \\.\ prefix as being translated through the Object Manager namespace so the symbolic link resolves to the internal device object.

Before CreateFile returns, the I/O Manager sends the driver an IRP\_MJ\_CREATE request. The dispatch routine runs in the context of the calling thread at PASSIVE\_LEVEL. If the driver reports success, a file-object handle is returned to the application.

DeviceIoControl takes eight parameters: device handle, IOCTL, input-buffer pointer, input-buffer size, output-buffer pointer, output-buffer size, pointer receiving the number of bytes returned, and an OVERLAPPED pointer for asynchronous operation. The example uses synchronous I/O and therefore does not develop OVERLAPPED operation.

The first DeviceIoControl call requests the entire GDT and writes the returned bytes to a file. The second supplies an 8-byte input structure containing an offset and size, requesting the descriptor selected by FS. DecodeDescriptor then extracts the descriptor base, limit, and DPL into EBX, ECX, and EAX; the program converts them to text and displays them.

At the end, the application closes the device, stops and deletes the service, closes the SCM handles, and exits.

### 3.3.8.2. GDTDump Driver

The source next presents the driver itself. Although in real development the driver would normally be written first, the book reverses the order for teaching purposes.

#### Listing 3.21. GDTDump driver

```

format PE Native 4.0
include 'win32w.inc'
include 'DDK\ntstatus.inc'
include 'DDK\ntdefs.inc'
include 'DDK\ntddk.inc'
include 'DDK\macros.inc'
include 'DDK\advmacro.inc'
entry DriverEntry
section '.code' code readable writeable executable
IOCTL_DEFINE_EX IOCTL_DUMP_GDT, FILE_ANY_ACCESS, 1, METHOD_BUFFERED
UNICODE_STRING_define DeviceName, "\\Device\\devGDTDump"
UNICODE_STRING_define SymbolicLinkName, "\\??\\GDTDump"
DeviceObject dd 0
GDTR:
.Limit dw ?
.Address dd ?
proc DispatchControl pDeviceObject, pIrp
local status:DWORD
push edi
push esi
push ecx
push edx
mov [status], STATUS_DEVICE_CONFIGURATION_ERROR
mov edi, [pIrp]
virtual at edi
.edi IRP
end virtual
mov esi, [.edi.Tail.Overlay.CurrentStackLocation]
virtual at esi
.esi IO_STACK_LOCATION
end virtual
cmp [.esi.Parameters.DeviceIoControl.IoControlCode], IOCTL_DUMP_GDT
jz .IOCTL_DUMP
jmp .next
.IOCTL_DUMP:
;int3
sgdt fword [GDTR]
xor ecx, ecx

```

```

mov cx, [GDTR.Limit]
inc ecx ; ecx = GDT size
xor eax, eax
mov edx, [.esi.Parameters.DeviceIoControl.InputBufferLength]
cmp edx, 8
jnz .copydata
mov eax, [.edi.AssociatedIrp.SystemBuffer]
mov edx, [eax+4] ; edx = needed data size
mov eax, [eax] ; eax = start offset in GDT
add edx, eax
cmp edx, ecx
jna @f
mov edx, ecx
@@:
sub edx, eax ; edx = fact data size
cmp edx, 7FFFFFFFh
jna @f
mov edx, 0
@@:
mov ecx, edx
.copydata:
mov [status], STATUS_BUFFER_TOO_SMALL
cmp ecx, [.esi.Parameters.DeviceIoControl.OutputBufferLength]
ja .error
push ecx
mov esi, [GDTR.Address]
add esi, eax
mov edx, edi ; save EDI in EDX
mov edi, [.edi.AssociatedIrp.SystemBuffer]
rep movsb
mov edi, edx ; restore EDI
pop [.edi.IoStatus.Information]
mov [status], STATUS_SUCCESS
jmp .exit
.next:
jmp .exit
.error:
mov [.edi.IoStatus.Information], 0
jmp .exit
.exit:
push [status]
pop [.edi.IoStatus.Status]
fastcall IofCompleteRequest, edi, IO_NO_INCREMENT
mov eax, [status]
pop edx
pop ecx
pop esi
pop edi
ret
endp
proc DispatchCreate pDeviceObject, pIrp
mov eax, [pIrp]
virtual at eax
.eax IRP
end virtual
mov [.eax.IoStatus.Status], STATUS_SUCCESS
and [.eax.IoStatus.Information], 0
mov ecx, [pIrp]
mov edx, IO_NO_INCREMENT
call [IofCompleteRequest]
mov eax, STATUS_SUCCESS
ret
endp
proc DispatchClose pDeviceObject, pIrp
mov eax, [pIrp]
virtual at eax
.eax IRP
end virtual
mov [.eax.IoStatus.Status], STATUS_SUCCESS
and [.eax.IoStatus.Information], 0
mov ecx, [pIrp]
mov edx, IO_NO_INCREMENT
call [IofCompleteRequest]
mov eax, STATUS_SUCCESS
ret
endp
proc DriverUnloadHandler pDriverObject
invoke IoDeleteSymbolicLink, SymbolicLinkName
invoke IoDeleteDevice, [DeviceObject]
ret
endp
proc DriverEntry pDriverObject, pusRegistryPath
local status:DWORD
mov [status], STATUS_DEVICE_CONFIGURATION_ERROR
invoke IoCreateDevice, [pDriverObject], 0, DeviceName, FILE_DEVICE_UNKNOWN, 0, \
FALSE, DeviceObject
cmp eax, STATUS_SUCCESS
jnz .END
invoke IoCreateSymbolicLink, SymbolicLinkName, DeviceName
cmp eax, STATUS_SUCCESS
jnz .simlinkfail
mov eax, [pDriverObject]
virtual at eax
.eax DRIVER_OBJECT
end virtual
mov [.eax.MajorFunction+IRP_MJ_CREATE*4], DispatchCreate
mov [.eax.MajorFunction+IRP_MJ_CLOSE*4], DispatchClose

```

```

mov [.eax.MajorFunction+IRP_MJ_DEVICE_CONTROL*4], DispatchControl
mov [.eax.DriverUnload], DriverUnloadHandler
mov [status], STATUS_SUCCESS
jmp .END
.simlinkfail:
invoke IoDeleteDevice, [DeviceObject]
.END:
mov eax, [status]
ret
endp
section '.relocs' fixups readable writeable discardable
section '.idata' import readable writeable
library ntoskrnl, 'ntoskrnl.exe'
import ntoskrnl,\
    IoCreateDevice, 'IoCreateDevice',\
    IoCreateSymbolicLink, 'IoCreateSymbolicLink',\
    IoDeleteDevice, 'IoDeleteDevice',\
    IoDeleteSymbolicLink, 'IoDeleteSymbolicLink',\
    IoCompleteRequest, 'IoCompleteRequest'

```

DriverEntry begins by creating a `DEVICE_OBJECT` with `IoCreateDevice`. The source lists seven parameters: the `DRIVER_OBJECT` pointer; device-extension size; device name; device-type identifier; additional device characteristics; exclusive-access flag; and a pointer that receives the created `DEVICE_OBJECT` pointer.

The device-extension field is useful when one driver manages multiple device objects and needs per-device state. This example does not use it. `FILE_DEVICE_UNKNOWN` is used for the device type.

The driver then creates a symbolic link with `IoCreateSymbolicLink` so ring-3 code can resolve a user-visible name to the kernel device name. The source points out that `UNICODE_STRING_define` is a helper macro for declaring `UNICODE_STRING` objects.

### Listing 3.22. Filling the MajorFunction array

```

mov eax, [pDriverObject]
virtual at eax
.eax DRIVER_OBJECT
end virtual
mov [.eax.MajorFunction+IRP_MJ_CREATE*4], DispatchCreate
mov [.eax.MajorFunction+IRP_MJ_CLOSE*4], DispatchClose
mov [.eax.MajorFunction+IRP_MJ_DEVICE_CONTROL*4], DispatchControl
mov [.eax.DriverUnload], DriverUnloadHandler

```

The `DRIVER_OBJECT` structure contains a `MajorFunction` array of dispatch-routine pointers. Each IRP major code indexes one entry. A driver fills only the requests it supports; other entries are initialized by the I/O Manager to an internal routine that reports an unsupported request.

The constants listed by the source include `IRP_MJ_CREATE = 0`, `IRP_MJ_CLOSE = 2`, `IRP_MJ_READ = 3`, `IRP_MJ_WRITE = 4`, `IRP_MJ_DEVICE_CONTROL = 0Eh`, and `IRP_MJ_CLEANUP = 12h`. Multiplying the major code by four gives the pointer offset in the 32-bit `MajorFunction` array.

`DriverUnload` is filled with `DriverUnloadHandler` so the system has a routine to call when the driver is stopped dynamically. If initialization succeeds, `DriverEntry` returns `STATUS_SUCCESS`.

### Listing 3.23. IRP\_MJ\_CREATE request handler

```

proc DispatchCreate pDeviceObject, pIrp
mov eax, [pIrp]
virtual at eax
.eax IRP
end virtual
mov [.eax.IoStatus.Status], STATUS_SUCCESS
and [.eax.IoStatus.Information], 0
mov ecx, [pIrp]
mov edx, IO_NO_INCREMENT
call [IoCompleteRequest]
mov eax, STATUS_SUCCESS
ret
endp

```

`DispatchCreate` and `DispatchClose` are effectively identical in the example. They set the IRP status to success, set `Information` to zero, complete the I/O request, and return `STATUS_SUCCESS`.

The source discusses the `fastcall` form `IoCompleteRequest`. The first argument is placed in `ECX` and the second in `EDX`; any additional arguments would be pushed right-to-left. It contrasts this with a `stdcall`-form name in the prose.

*Editorial note: the explanatory text repeatedly prints `IoCoirpleteRequest` and `IoCoirpleteRequest`. The actual listing and import table use `IoCompleteRequest`. The listing spelling is preserved in code and the prose typo is recorded here.*

### Listing 3.24. Macro for calling fastcall functions

```

macro fastcall func, p1, p2, [px]
{
if ~ px eq
reverse push px
end if
if ~ p1 eq
mov ecx, p1
end if
if ~ p2 eq
mov edx, p2
end if
call [func]
}

```

With the opening and closing handlers defined, the remaining major operation is IRP\_MJ\_DEVICE\_CONTROL.

### Listing 3.25. IRP\_MJ\_DEVICE\_CONTROL handler

```
proc DispatchControl pDeviceObject, pIrp
. . .
    cmp [.esi.Parameters.DeviceIoControl.IoControlCode], IOCTL_DUMP_GDT
    jz .IOCTL_DUMP
    jmp .next
.IOCTL_DUMP:
    sgdt fword [GDTR]
    xor ecx, ecx
    mov cx, [GDTR.Limit]
    inc ecx ; ecx = GDT size
    xor eax, eax
    mov edx, [.esi.Parameters.DeviceIoControl.InputBufferLength]
    cmp edx, 8
    jnz .copydata
    mov eax, [.edi.AssociatedIrp.SystemBuffer]
    mov edx, [eax+4] ; edx = needed data size
    mov eax, [eax] ; eax = start offset in GDT
    add edx, eax
    cmp edx, ecx
    jna @f
    mov edx, ecx
@@:
    sub edx, eax ; edx = fact data size
    cmp edx, 7FFFFFFFh
    jna @f
    mov edx, 0
@@:
    mov ecx, edx
.copydata:
    mov [status], STATUS_BUFFER_TOO_SMALL
    cmp ecx, [.esi.Parameters.DeviceIoControl.OutputBufferLength]
    ja .error
    push ecx
    mov esi, [GDTR.Address]
    add esi, eax
    mov edx, edi ; save EDI in EDX
    mov edi, [.edi.AssociatedIrp.SystemBuffer]
    rep movsb
    mov edi, edx ; restore EDI
    pop [.edi.IoStatus.Information]
    mov [status], STATUS_SUCCESS
. . .
    push [status]
    pop [.edi.IoStatus.Status]
    fastcall IofCompleteRequest, edi, IO_NO_INCREMENT
    mov eax, [status]
. . .
    ret
endp
```

When the control code is IOCTL\_DUMP\_GDT, the routine obtains GDTR with SGDT and calculates the table size as limit + 1. If the input buffer length is not 8, the request is interpreted as a request for the entire GDT. If an 8-byte input structure is present, its first DWORD is the starting offset and its second DWORD is the requested number of bytes.

After bounds checking, EAX contains the starting offset and ECX the actual byte count. The routine checks the caller's output-buffer length, copies the requested bytes from the GDT into the buffered-I/O system buffer, stores the byte count in IRP.IoStatus.Information, stores the final NTSTATUS in IRP.IoStatus.Status, and completes the IRP.

*Editorial note: the explanatory prose later prints RP.IoStatus.Status and IofCompleteRequest. The structure and function spellings used by the code are retained above.*

DriverUnloadHandler removes the symbolic link and deletes the device object, freeing the system objects created during DriverEntry.

### 3.3.8.3. Debugging Drivers

Kernel-mode debugging is substantially less forgiving than user-mode debugging. A mistake in an ordinary application can often be handled by terminating and restarting the process; an invalid kernel operation can instead produce a system crash and a BSOD.

The book names WinDBG, SoftICE, and Syser Kernel Debugger as kernel debuggers available in its period. It describes WinDBG as powerful but requiring a separate debugger/target arrangement, while also noting that a virtual machine can substitute for a second physical computer.

SoftICE is described as historically popular and capable of local kernel debugging, but no longer maintained and incompatible with then-new Windows versions beginning with Windows XP SP3. Syser is described as a commercial alternative that was still maintained when the book was published.

*Historical note: the debugger availability and limitations above are source-era statements from 2011 and are not current recommendations.*

The source again recommends INT3 as a useful debugging aid. Because a driver may load at an unpredictable address, placing INT3 in DriverEntry or another point of interest allows a connected kernel debugger to stop as soon as execution reaches that location.

### **3.3.9. Summary**

This section introduced the fundamentals of Win32 driver programming. This is only an introduction: asynchronous I/O, direct-I/O buffer management, interaction with file systems, the registry, system memory, and many other kernel-programming topics remain outside the chapter. A complete treatment of Windows kernel programming would require far more space than this book can devote to it.

# Chapter 4. Long Mode

## 4.1. Introduction to Long Mode

The previous sections dealt with 32-bit programming: programming in protected mode and programming under the protected-mode Windows operating system. This section and the next turn to 64-bit programming: programming in the processor's 64-bit operating mode (long mode) and programming under 64-bit versions of Windows.

AMD64 is a 64-bit microprocessor architecture and corresponding instruction-set extension developed by AMD. AMD64 is described as a 64-bit extension of x86 with full backward compatibility. Intel licensed the architecture, with minor changes, under the name EM64T. Adopting one compatible architecture avoided the fragmentation that two incompatible x86-64 architectures would have caused for operating-system development.

*Historical note: the source uses Intel's older name EM64T. Intel later standardized the public name as Intel 64. This translation retains the book's terminology.*

According to the terminology quoted by the source, Intel documentation officially calls the processor's 64-bit operating environment IA-32e, while AMD documentation calls it long mode. In this book, 'long mode' means the whole 64-bit operating environment, including both of its submodes. The phrase '64-bit mode' means specifically the submode in which 64-bit code executes.

### 4.1.1. General Overview

Long mode contains two submodes: 64-bit mode itself and compatibility mode. In 64-bit mode the default address size is 8 bytes and the default operand size is 4 bytes. In compatibility mode, the source gives a 4-byte address size and a 4-byte default operand size, as in protected mode. Compatibility mode is selected through a bit in the code-segment descriptor.

In 64-bit processor mode, the source lists the following registers as available:

- General-purpose registers: 64-bit RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP, and R8-R15; 32-bit EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, with R8D-R15D as the low halves of the added 64-bit registers; 16-bit AX, BX, CX, DX, SI, DI, SP, BP, and R8W-R15W; and the 8-bit registers AH, BH, CH, DH, AL, BL, CL, DL, SIL, DIL, SPL, BPL, and the source's printed R8L-R15L names.

*Editorial note: the book prints R8L-R15L. Current AMD/Intel documentation normally names these low-byte registers R8B-R15B. The printed source form is retained here rather than silently changed.*

- 64-bit RIP, the instruction pointer.
- 16-bit segment registers CS, DS, SS, ES, FS, and GS.
- 64-bit RFLAGS.
- 80-bit x87 floating-point registers ST0-ST7.
- 64-bit MMX registers MM0-MM7.
- 128-bit XMM registers XMM0-XMM15 and the 32-bit MXCSR register.
- Control registers CR0-CR4 and CR8, plus the system-table registers GDTR, LDTR, IDTR, and task register TR.
- Debug registers DR0-DR3, DR6, and DR7.
- MSRs (Model Specific Registers).

The long-mode instruction set did not undergo a fundamental redesign and that instruction formats remain largely the same. Access to 64-bit operands and the added registers is provided through the REX prefix. It therefore recommends using 32-bit registers wherever possible as a code-size optimization.

*Editorial note: the statement that opcodes operating on 64-bit or added registers grow by at least one byte and therefore create a 'serious' code-size optimization problem is translated as the book presents it; the exact encoding cost depends on the instruction and register selection.*

PUSHAD and POPAD are no longer supported. PUSH and POP operate on 64-bit and 16-bit operands. A pushed 16-bit operand is not widened to 64 bits, so RSP becomes misaligned relative to an 8-byte boundary.

EFLAGS is extended to 64 bits and is called RFLAGS, but the source says no new flag fields are introduced. The 64-bit instruction set also makes many operations on 64-bit integers directly expressible with a single instruction.

Far control transfers are more restricted. The source states that an immediate instruction of the form `JMP selector16:offset64` does not exist. For an intersegment transfer - commonly to reload CS - the source uses a far pointer stored in memory.

#### Listing 4.1. Far jump in long mode

```
jump_value:
dq jump_offset
dw jump_selektor
...
jmp tbyte[jump_value]
```

This memory-indirect form was also available in protected mode, but describes it as the usable form for this purpose in 64-bit mode.

The book also states that an instruction of the form `mov [address], value64` cannot encode an arbitrary 64-bit immediate value directly into memory. If such a value must be stored, the source recommends loading it into a register first and then storing the register to memory.

Long mode introduces RIP-relative addressing. That, in this form, 32-bit relative displacements are used instead of embedding full 64-bit data or branch addresses and claims that this can reduce code size by nearly 40 percent. The book does not develop this optimization further.

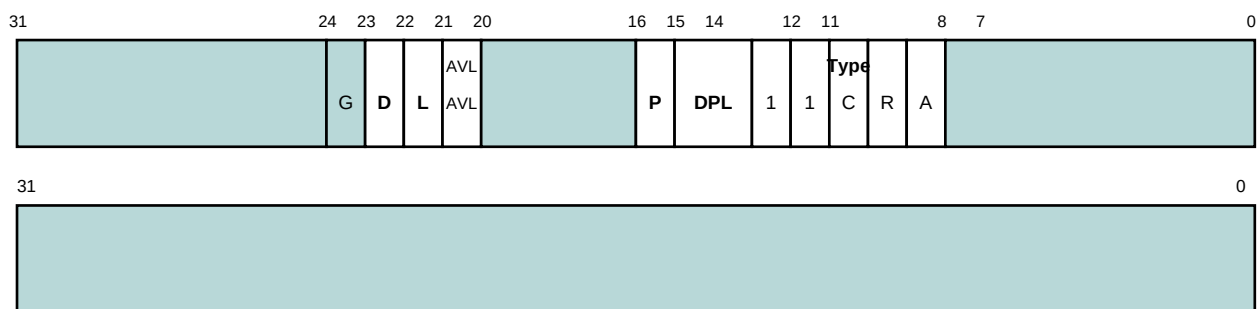
Segmentation changes substantially in long mode. In the source's overview, code and data segments begin at address zero and cover the full 64-bit address space, segment-limit checks are not performed, most data-descriptor fields and many code-descriptor fields are ignored, and hardware task switching is no longer supported; multitasking must be implemented in software.

Page translation changes as well because of the much larger address space. The source gives up to  $2^{48}$  bytes (256 TB) of virtual address space and up to  $2^{52}$  bytes of physical address space.

### 4.1.2. Segmentation in Long Mode

As stated above, ordinary segmentation is effectively disabled in 64-bit mode: code- and data-segment limit checks are not performed and their base address is treated as zero. For compatibility, the hidden base values associated with FS and GS can be accessed through the IA32\_FS\_BASE and IA32\_GS\_BASE MSRs. This allows software to assign nonzero bases to FS and GS, which the source presents as useful when porting 32-bit systems that used a segment register for per-thread information.

Figure 4.1 shows the long-mode code-segment descriptor. As in the original figure, shaded fields are those the processor ignores in 64-bit mode.



*Shaded fields are ignored by the processor in 64-bit mode.*

Figure 4.1. Code-segment descriptor in long mode

The limit and base-address fields are ignored. The remaining fields have the same general meanings as in protected mode. The source emphasizes the L and D fields. If L = 0, code in the segment executes in compatibility mode. If D = 0, the default operand size is 16 bits; if D = 1, it is 32 bits. If L = 1, code executes in 64-bit mode.

The combination D = 1 and L = 1 is reserved in the source's description.

The same general simplification applies to data-segment descriptors. Because segment limits are not checked in 64-bit mode, the fields associated with the segment base, limit, and G are ignored.

The source further states that AMD processors ignore even more descriptor fields: for data segments, all fields except Present; for code segments, R, A, and G are also ignored. It warns that differences between AMD and Intel behavior should be considered when designing 64-bit operating systems and low-level software.

*Editorial note: this paragraph is translated from the source as written. Exact long-mode descriptor checks are processor-architecture details and should be verified against the AMD64 and Intel 64 manuals for the target processor.*

### 4.1.3. Page-Translation Mechanism

Long mode uses the PAE-style paging mechanism and that PAE must be enabled before entering long mode. Long mode cannot be entered with PAE disabled.

The book again gives up to  $2^{48}$  bytes of virtual address space and up to  $2^{52}$  bytes of physical address space. It lists 4 KB, 2 MB, and 1 GB page sizes and says they may be mixed.

Four kinds of paging tables are used. In addition to page tables, page directories, and page-directory-pointer tables, long mode introduces the Page Map Level 4 (PML4). CR3 is widened for the address of the PML4. Section 4.2 develops this mechanism in detail.

### 4.1.4. Entering Long Mode

Entering 64-bit mode is more involved than entering protected mode. In addition to preparing a GDT, software must prepare the paging structures required for long-mode address translation.

The transition is controlled in part by the EFER MSR, called IA32\_EFER in Intel documentation, at MSR index 0C0000080h. The source highlights bit 8, LME, and bit 10, LMA. LME is read/write. LMA is read-only and indicates that long mode is active.

*Editorial note: one sentence says LMA must be 'set' to make the processor operate in 64-bit mode, but the same passage correctly states that LMA is read-only and later explains that hardware sets it after paging is enabled with LME already set.*

The source gives the transition sequence as:

1. Disable paging (CR0.PG = 0) and enable PAE (CR4 bit 5 = 1).
2. Load CR3 with the physical address of the PML4.
3. Set IA32\_EFER.LME. LMA is read-only.
4. Enable paging by setting CR0.PG. This causes the processor to set IA32\_EFER.LMA.
5. Execute JMP using a selector for a code descriptor whose L bit is set.

After paging is enabled, the processor is initially executing in compatibility mode because CS still refers to a 32-bit code-segment descriptor. The environment is already in long mode, but completing the transition into the 64-bit submode requires loading CS with a selector for a 64-bit code descriptor.

#### 4.1.5. Practice

The practical example switches the processor into long mode and writes a message to the screen. The earlier real-mode to protected-mode bootstrap is omitted because it is unchanged. As with the protected-mode examples, the source builds a reusable long-mode bootstrap and includes a separate file containing the code to run after the transition.

#### Listing 4.2. Switching the processor to 64-bit mode

```
.....
align 8
GDT:
    NULL_descr    db 8 dup(0)
    CODE32_descr  db 0FFh,0FFh,00h,00h,00h,10011010b,11001111b,00h
    DATA_descr   db 0FFh,0FFh,00h,00h,00h,10010010b,11001111b,00h
    CODE64_descr  db 00h,00h,00h,00h,00h,10011000b,00100000b,00h
    GDT_size equ $-GDT

label GDTR fword
    dw GDT_size-1
    dd ?

use32
PROTECTED_MODE_ENTRY_POINT:
    mov ax, DATA_SELEKTOR
    mov ds, ax
    mov es, ax
    mov ss, ax
    mov esp, STACK_BASE_ADDRESS

    call delta
delta:
    pop ebx
    add ebx, PM_CODE_START-delta

    mov esi, ebx
    mov edi, PM_CODE_BASE_ADDRESS
    mov ecx, PM_CODE_SIZE
    rep movsb

    mov eax, PM_CODE_BASE_ADDRESS
    jmp eax

PM_CODE_START:
ORG PM_CODE_BASE_ADDRESS

    mov eax, cr4
    bts eax, 5 ; PAE = 1
    mov cr4, eax

    mov dword [PDE_addr], 010000011b ; PS or Present or Write
    mov dword [PDE_addr+4], 0
    mov dword [PDPE_addr], PDE_addr or 3 ; Present or Write
    mov dword [PDPE_addr+4], 0
    mov dword [PML4_addr], PDPE_addr or 3 ; Present or Write
    mov dword [PML4_addr+4], 0

    mov eax, PML4_addr
    mov cr3, eax

    mov ecx, 0xC0000080 ; EFER
    rdmsr
    bts eax, 8 ; EFER.LME = 1
    wrmsr

    mov eax, cr0
    bts eax, 31 ; PG = 1
```

```

mov cr0, eax

jmp CODE64_SELECTOR:LONG_MODE_ENTRY_POINT

use64
LONG_MODE_ENTRY_POINT:
    mov ax, ds ; reload the segment registers
    mov ds, ax
    mov ss, ax
    mov es, ax

include 'LM_CODE.ASM'
PM_CODE_END:

```

*Listing 4.2 preserves the source identifiers, numeric values, spelling, and instruction forms. Only the Russian comment on reloading the segment registers has been translated. The initial row of dots is also retained because the book deliberately omits the unchanged protected-mode bootstrap at that point.*

After entering protected mode, the program copies its main protected-mode block to PM\_CODE\_BASE\_ADDRESS and jumps there. It first sets CR4.PAE and then prepares a minimal set of paging structures.

The section intentionally does not yet explain the page-table construction; the source asks the reader to treat those instructions as a prepared sequence until Section 4.2. The PML4 is the root table. Its address is loaded into CR3, IA32\_EFER.LME is set, and paging is enabled through CR0.PG.

The program then jumps to the 64-bit code segment. The subsequent reloads of DS, SS, and ES are precautionary rather than an essential part of entering long mode. Finally it includes the file containing the code that confirms the transition on screen.

### Listing 4.3. Code executing in 64-bit mode

```

LM_CODE_START:
    mov rsi, message2
    mov rdi, 0B8000h
    mov rcx, mess2end-message2
    rep movsb
    jmp $

message2 db "w5e5 5i5n5 5l5o5n5g5 5m5o5d5e5!5"
mess2end:

```

The complete LM.ASM and LM\_CODE.ASM source files were supplied in the source book's part4 directory. As with the bare-metal programs in Chapter 2, the book instructs the reader to run the mode-switching program under MS-DOS after booting from removable media.

*Historical note: the MS-DOS/floppy/USB workflow is reproduced from the book's 2011 environment. It is source material, not a recommendation for modern development.*

## 4.1.6. Summary

This section introduced the processor's 64-bit operating environment, described the main characteristics of long mode, and built a program that switches the processor into 64-bit mode. The next section examines memory management in long mode.

## 4.2. Working with Memory in Long Mode

Section 4.1 introduced the foundations of the processor's 64-bit mode. This section describes virtual-memory operation in long mode. Memory management is the second major aspect of long-mode operation because 64-bit mode requires page translation to be enabled and the source places most operating-system code and data protection at the page level.

Virtual memory is somewhat more complicated than in protected mode because more tables and structures are used to describe pages, but the basic principles of virtual-to-physical translation remain the same.

### 4.2.1. General Overview

The source gives an architectural maximum of  $2^{52}$  bytes of physical memory and  $2^{48}$  bytes of virtual memory. It notes that not every processor implements a 52-bit physical address and recommends CPUID to determine the physical-address width implemented by the current processor.

The 52-bit physical-address limit is presented as an architectural limit. The 48-bit virtual-address limit is described as less permanent: future processors might extend the effective virtual-address width by adding further translation structure or otherwise expanding the scheme.

Long mode supports 4 KB pages, for which the virtual address is divided into five functional parts, and 2 MB pages, for which it is divided into four. 1 GB pages are also available on processors that support them, producing a three-part virtual address. CR4.PSE is ignored in long mode.

All virtual addresses used by the 48-bit scheme must be in canonical form. The source expresses this as requiring the upper 16 bits to be either all zeros or all ones; otherwise #GP or, for stack-related use, #SS is generated. In effect, bits 63-48 reproduce bit 47.

The source presents canonical form as a compatibility measure for future processors. Most operating systems divide the virtual address space into a lower application region and an upper kernel region; sign extension of bit 47 leaves room for a future increase in effective virtual-address width without changing that broad split.

### 4.2.2. 4 KB Pages

The source calls 4 KB the most commonly used page size because it provides comparatively fine allocation granularity, while noting that the additional translation level makes address translation more expensive than with larger pages.

For a 4 KB page, the virtual address is divided as follows: bits 47-39 index the Page Map Level 4 (PML4); bits 38-30 index the page-directory-pointer table (PDPT); bits 29-21 index the page directory (PD); bits 20-12 index the page table (PT); and bits 11-0 are the offset within the page. Each table contains 512 entries of 8 bytes.

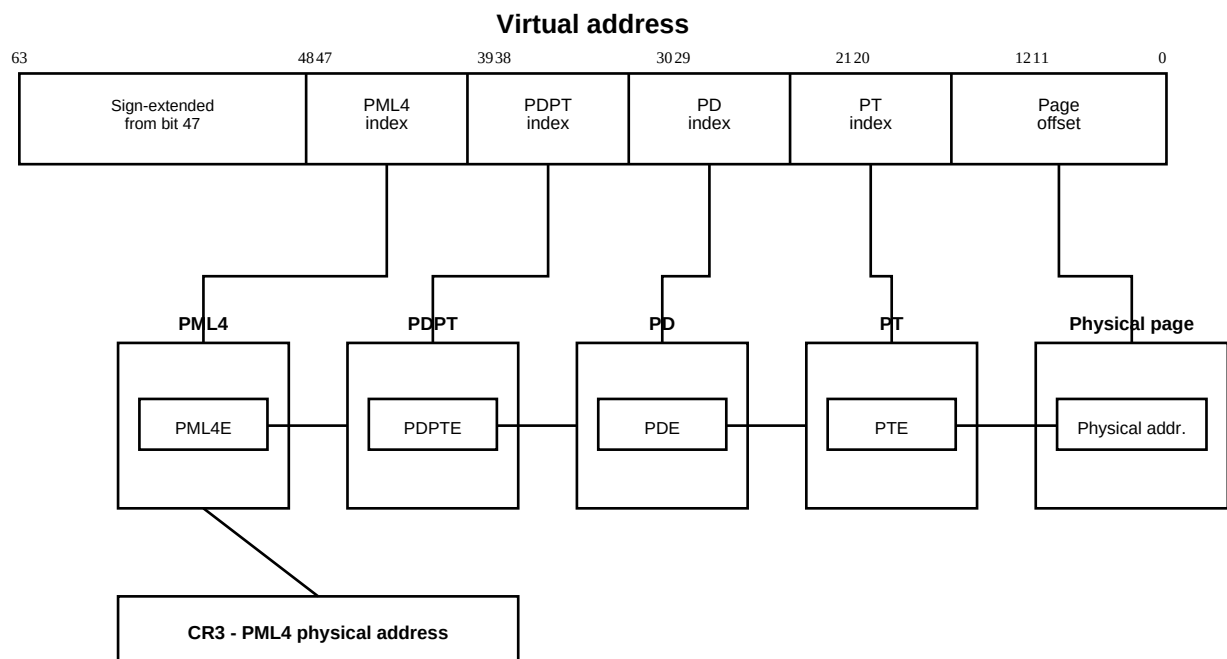


Figure 4.2. Address translation for 4 KB pages in long mode

## PML4E

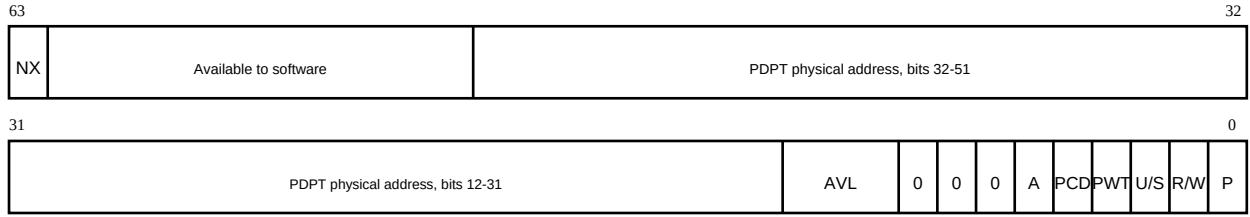


Figure 4.3. PML4 entry format (PML4E)

## PDPTE for a 4 KB page

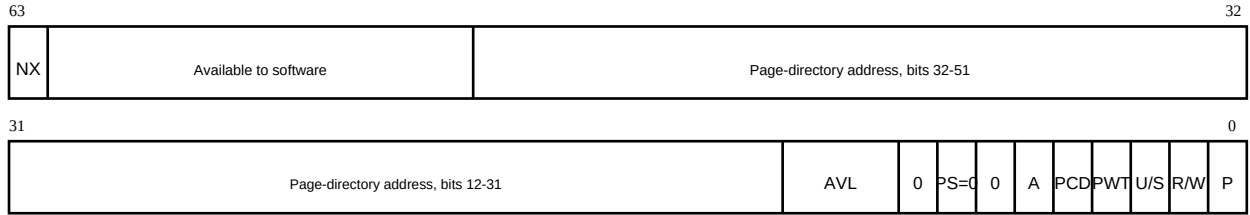


Figure 4.4. Page-directory-pointer-table entry (PDPTE) for 4 KB paging

## PDE for a 4 KB page

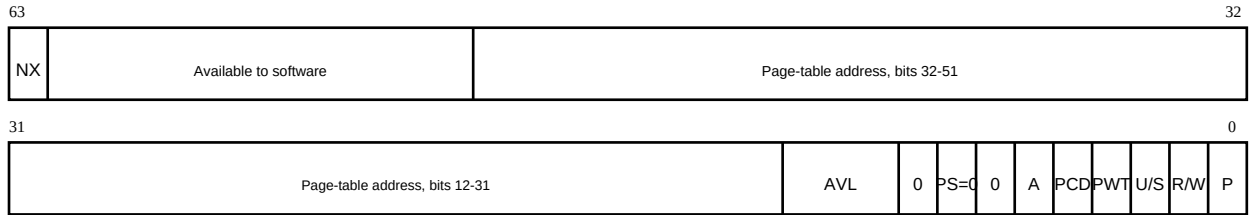


Figure 4.5. Page-directory entry (PDE) for 4 KB paging

## PTE



Figure 4.6. Page-table entry (PTE)

The named protection and caching bits have essentially the same meanings as their equivalents in protected-mode paging. It identifies bit 7 in the translation-table entries other than PML4 as the PS page-size bit. For the 4 KB path, PS is clear in the PDPTE and PDE so translation continues to the next table.

### 4.2.3. 2 MB Pages

With 2 MB pages the virtual address is divided into four parts rather than five, reducing the number of table walks and greatly reducing the memory consumed by translation tables. The trade-off is much coarser allocation granularity.

Bits 47-39 select the PML4 entry, bits 38-30 select the PDPTE, bits 29-21 select the PDE, and bits 20-0 are the offset within the 2 MB page. No page table is used.



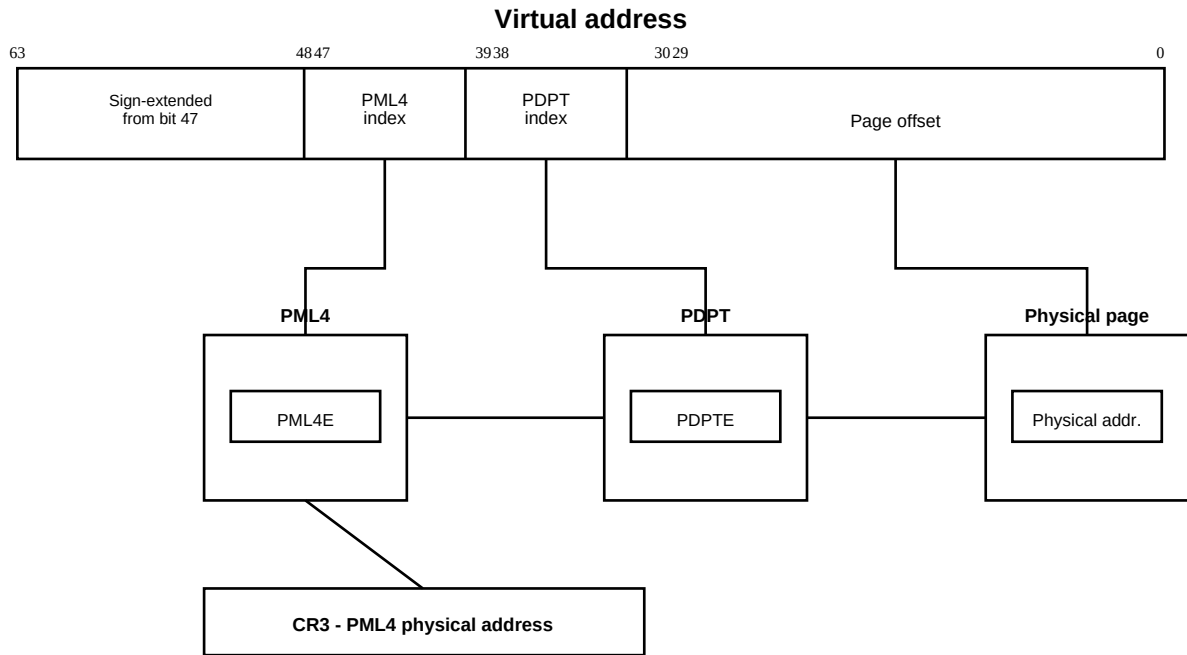


Figure 4.9. Address translation for 1 GB pages in long mode

**PDPTE for a 1 GB page**

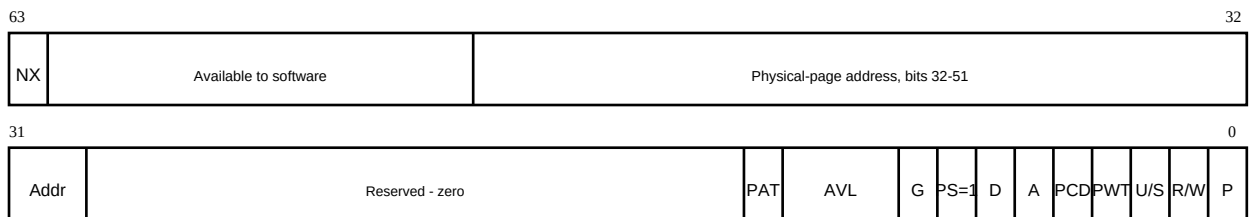


Figure 4.10. PDPTE format for a 1 GB page

The source recommends 1 GB pages as potentially useful for operating-system kernels, performance-critical code, and frequently used data because they reduce the number of translation tables and therefore the amount of work required for translation.

#### 4.2.5. CR3 Register

In long mode CR3 is treated as a 64-bit register. Bits 51-12 contain the physical address of the PML4. PCD and PWT control caching of the memory containing the PML4 in the source's description.

**CR3 in long mode**

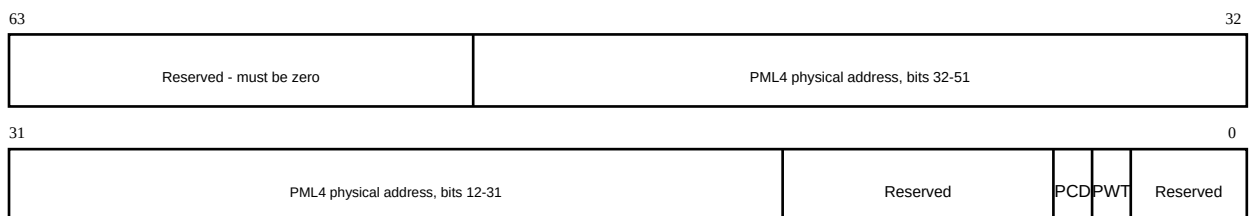


Figure 4.11. CR3 in long mode

#### 4.2.6. Protection Checks

Correct page-level protection is central to a 64-bit operating system because long mode places most code and data protection in the paging mechanism.

It identifies five relevant controls: R/W in every paging-structure entry; U/S in every paging-structure entry; NX in every paging-structure entry; WP in CR0; and NXE in IA32\_EFER.

User code can access a page only when U/S permits user access in every structure describing that page. Privileged code can access the page regardless of U/S in the source's simplified description.

User code can modify a page only when R/W permits writing at every translation level. With CR0.WP clear, privileged code can write even to pages marked read-only. When CR0.WP is set, privileged writes obey the same R/W checks.

Code, whether user or privileged, may execute from a page only when NX is clear at every relevant level, assuming execute-disable support has been enabled through IA32\_EFER.NXE. The source refers back to Section 2.3 for NX support detection and enabling.

#### 4.2.7. Practice

The practical example maps several virtual pages to one physical page and writes through those virtual addresses. To make the result visible, the mappings ultimately point at physical address 0B8000h, the text-mode video buffer.

All translation tables begin in one memory region. Its first page contains the PML4. Additional pages are allocated for lower-level tables only when required. A variable records the next free table page. If the Present bit in an entry is clear, the routine zeroes the next free page, installs its address with Present and Write set, advances the free-page index, and continues. If Present is already set, it follows the existing entry.

The book omits the complete program and prints the virtual-page creation routine in several fragments.

##### Listing 4.4a. Function for creating a virtual page

```
NextAvail4KPage dd 1 ; variable storing the ordinal number
                  ; of the next free page

AllocPage_4K:
;IN
; RAX physical address
; RBX virtual address
push rax
push rbx
push r8
push rdi
push rdx
push r9
push r10
push r11
push r12 ; ; save all used registers
mov r12, rax ; r12 = physical address
mov r8, rbx
shr r8, 39 ; r8 = PML4 index
shl r8, 3 ; r8 = offset in PML4
add r8, PLM4_BASE_ADDRESS ; r8 = PML4 entry addr
mov r9d, [r8+4]
shl r9, 32
mov r9d, [r8] ; r9 = PML4 entry
mov eax, r9d ; eax = r9d
and eax, 1 ; check present flag
jz @f
jmp .setPDPE
@@:
```

The routine saves all registers it uses and preserves the physical address in R12 because RAX is then reused. It computes the PML4 entry address from the virtual address in RBX and tests the Present bit.

##### Listing 4.4b. Function for creating a virtual page (continued)

```
xor rdi, rdi
mov edi, [NextAvail4KPage]
shl rdi, 12 ; rdi = address of next free page
add rdi, PLM4_BASE_ADDRESS
call ZeroPage_4K
mov rax, rdi ; rax = PDP table address
or eax, 3 ; rax = PDP table addr or Present or Write (R/W)
mov rdx, rax ; rdx = rax
mov [r8], eax
shr rax, 32
mov [r8+4], eax
inc dword [NextAvail4KPage]
mov r9, rdx ;
```

If the PML4 entry was not present, the routine zeroes the next free page, makes it the PDPT, stores its address with Present and R/W set, increments NextAvail4KPage, and leaves the new entry value in R9.

##### Listing 4.4c. Function for creating a virtual page (continued)

```
.setPDPE: ;r9 = contents of PML4 entry
;-----
and r9d, 0FFFFFF00h ; r9 = address of PDP table
mov r8, rbx
shr r8, 30
and r8d, 1FFh ; r8 = PDP index
shl r8, 3 ; r8 = offset in PDP
add r8, r9 ; r8 = address of entry in PDP table
mov r10d, [r8+4]
shl r10, 32
mov r10d, [r8] ; r10 = PDP entry
mov eax, r10d ; eax = r10d
and eax, 1 ; check P flag (present flag)
jz @f
jmp .setPDE
```

@@:

The same process is repeated for the PDPT: extract its address from the PML4E, calculate the PDPTE address, read the entry, and test Present.

#### Listing 4.4d. Function for creating a virtual page (continued)

```
xor rdi, rdi
mov edi, [NextAvail4KPage]
shl rdi, 12 ; rdi = address of next available page
add rdi, PLM4_BASE_ADDRESS
call ZeroPage_4K
mov rax, rdi
or eax, 3 ; rax = PD table addr or Present or Write
mov rdx, rax
mov [r8], eax
shr rax, 32
mov [r8+4], eax
inc dword [NextAvail4KPage]
mov r10, rdx ; r10 = PDP entry

.setPDE: ; r10 = contents of PDP-table entry
;-----
and r10d, 0FFFFFF00h ; r10 = address of page directory
mov rdx, rbx
shr rdx, 21
and edx, 1FFh ;
xor r8, r8
mov r8d, edx ; r8 = index in page directory
shl r8, 3 ; r8 = offset in page directory
add r8, r10 ; r8 = address of entry in page directory
mov r11d, [r8+4]
shl r11, 32
mov r11d, [r8] ;
mov eax, r11d ; eax = r10d
and eax, 1 ; check P flag (present flag)
jz @f
jmp .setPTE
@@:
xor rdi, rdi
mov edi, [NextAvail4KPage]
shl rdi, 12 ; rdi = address of next available page
add rdi, PLM4_BASE_ADDRESS
call ZeroPage_4K
mov rax, rdi ; rax = page-table address
or eax, 3 ; rax = PT addr or Present or Write
mov rdx, rax
mov [r8], eax
shr rax, 32
mov [r8+4], eax
inc dword [NextAvail4KPage]
mov r11, rdx ; r11 = PD entry
```

*Listing 4.4d preserves the source comment `eax = r10d` even though the immediately preceding value was loaded from R11; this appears to be a source-comment error. The Russian comments have been translated, but identifiers and instructions are unchanged.*

This block allocates a page directory when needed and then repeats the same process for the page table. On exit from the allocation path, R11 contains the PDE value that points at the PT.

#### Listing 4.4e. Function for creating a virtual page (continued)

```
.setPTE:
;-----
and r11d, 0FFFFFF00h ; r11 = PT addr
mov rdx, rbx
shr rdx, 12
and edx, 1FFh
xor r8, r8
mov r8d, edx ; r8 = index in page table
shl r8, 3 ; r8 = offset in page table
add r8, r11 ; r8 = address of page-table entry
; physical address is in r12
or r12d, 3 ; page addr or present or write
mov [r8], r12d
shr r12, 32
mov [r8+4], r12d
pop r12
pop r11
pop r10
pop r9
pop rdx
pop rdi
pop r8
pop rbx
pop rax
ret
```

The final fragment fills the PTE with the physical address saved in R12 plus Present and Write, then restores the registers. The virtual region containing the translation tables must itself be mapped and suggests arranging this during the protected-mode stage.

The 2 MB page-allocation routine follows the same general procedure, except that the PDE has PS set.

#### Listing 4.5. Filling structures to create a virtual page

```

mov dword [PDE_addr], 010000011b ; PS or Present or Write
mov dword [PDE_addr+4], 0
mov dword [PDPE_addr], PDE_addr or 3 ; Present or Write
mov dword [PDPE_addr+4], 0
mov dword [PML4_addr], PDPE_addr or 3 ; Present or Write
mov dword [PML4_addr+4], 0

```

These are the lines omitted from the explanation in Section 4.1. They build the minimal paging structures needed to represent the first 2 MB of virtual memory during the transition into long mode.

The example then maps several virtual addresses to the same physical video page to verify the page-creation routines.

#### Listing 4.6a. Demonstrating address translation in long mode

```

PLM4_BASE_ADDRESS equ 100000h
LM_CODE_START:
mov rdi, PLM4_BASE_ADDRESS
call ZeroPage_4K
xor rax, rax
mov eax, LM_CODE_START
and eax, 0FFFFFF000h
mov rbx, rax
call AllocPage_4K
mov eax, 0B8000h
mov rbx, rax
call AllocPage_4K
mov eax, 008000h
mov rbx, rax
call AllocPage_4K
mov eax, 07000h
mov rbx, rax
call AllocPage_4K

```

At the beginning, the code page, the text-video-buffer page, and the stack pages around 07000h and 08000h are mapped to their original physical addresses in the source's explanation.

*Editorial note: the printed fragment explicitly shows the call for 008000h and then the call for 07000h. The prose describes the stack pages as 07000h and 08000h. This edition preserves both the code and the source explanation rather than silently reconciling them.*

#### Listing 4.6b. Demonstrating address translation in long mode (continued)

```

mov eax, 0B8000h
mov ebx, 12345000h
call AllocPage_4K
xor eax, eax
mov ebx, 0ABC8000h
and ebx, 0FFE0000h
call AllocPage_2M
mov eax, 0B8000h
mov rbx, 48A98765000h
call AllocPage_4K
xor rax, rax
mov eax, PLM4_BASE_ADDRESS
mov cr3, rax
mov rsi, message
mov al, 0
mov ah, 0
mov bl, "5"
call OutText
mov rsi, testmsg1
mov al, 0
mov ah, 1
mov bl, "5"
mov rdx, 12345000h
call OutTextEx
mov rsi, testmsg2
mov al, 0
mov ah, 2
mov bl, "5"
mov rdx, 0ABC8000h
call OutTextEx
mov rsi, testmsg3
mov al, 0
mov ah, 3
mov bl, "5"
mov rdx, 48A98765000h
call OutTextEx
jmp $
message db "We in long mode!",0
testmsg1 db "Write to address 0x 12345000",0
testmsg2 db "Write to address 0x ABC8000",0
testmsg3 db "Write to address 0x 000048A 98765000",0

```

For 0ABC8000h the example deliberately uses a 2 MB mapping, demonstrating that 4 KB and 2 MB pages may be used together. OutText writes through the ordinary video address 0B8000h; OutTextEx writes through the virtual address supplied in RDX.

*Editorial note: the prose following the listing refers to it as "Listing 4.66". The actual heading is Listing 4.6b; the translation uses the heading number and records the source typo here.*

The complete program was supplied in the part4 directory on the accompanying CD.

#### 4.2.8. Summary

This section presented the basic principles of virtual-memory operation in 64-bit mode. The next section turns to interrupts in long mode.

*Editorial note: the printed summary says the next section discusses interrupts "in protected mode", even though Section 4.3 is explicitly titled "Interrupts in long mode". The translation follows the chapter structure and notes the mismatch rather than silently treating the printed wording as authoritative.*

## 4.3. Interrupts in Long Mode

Interrupts are one of the most important parts of any operating system: handling interrupts and exceptions allows the operating system to interact with external devices and protect its data. Because page translation is a prerequisite for entering long mode, virtual memory was covered before interrupts.

The interrupt mechanism in 64-bit mode changed very little compared with protected mode and in some respects became simpler. Task gates are no longer supported because hardware task switching is no longer supported in long mode.

Long mode also introduces the IST mechanism. It allows an interrupt handler to use one of as many as seven dedicated stack pointers. The assignment of the processor's exception vectors is unchanged.

### 4.3.1. Gate Descriptors

In 64-bit mode there are only two interrupt-related gate types: interrupt gates and trap gates. Both descriptors are widened to 128 bits so they can hold a 64-bit handler address.

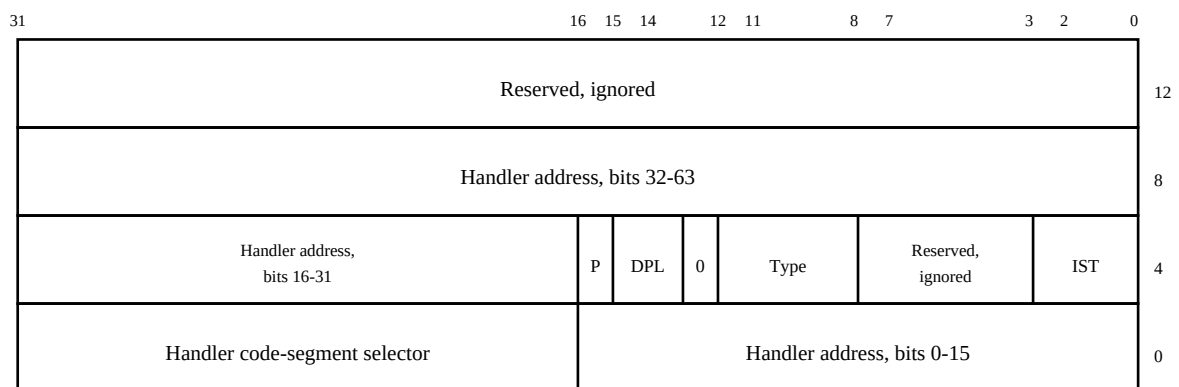


Figure 4.12. Interrupt- and trap-gate descriptor format in long mode

For an interrupt gate the type field contains 1110b; for a trap gate it contains 1111b. The behavioral difference is the same as in protected mode: entry through a trap gate leaves IF unchanged, while entry through an interrupt gate clears IF. A trap-gate handler is therefore transparent to maskable interrupts in the source's description.

The IST field contains the number of the stack pointer used by the interrupt handler. The IST mechanism is described below.

### 4.3.2. IDT, 64-bit TSS, and the IST Mechanism

As in protected mode, all gate descriptors are collected in the IDT. The IDT may be located at any address; its base and limit are stored in IDTR. The IDTR format changes only in that its base-address field is widened to 64 bits. LIDT and SIDT are used in the same way as before, except that the source specifies a tbyte operand.

Although hardware multitasking is not supported in long mode, at least one TSS is still required so the processor can switch stacks when changing privilege levels. Because general-purpose register state is no longer saved by hardware task switching, the 64-bit TSS is repurposed primarily for privilege stacks and the IST stack pointers.

Unlike the protected-mode TSS, the 64-bit TSS does not store segment-register values. It contains 64-bit RSP0-RSP2 values and seven 64-bit IST pointers.

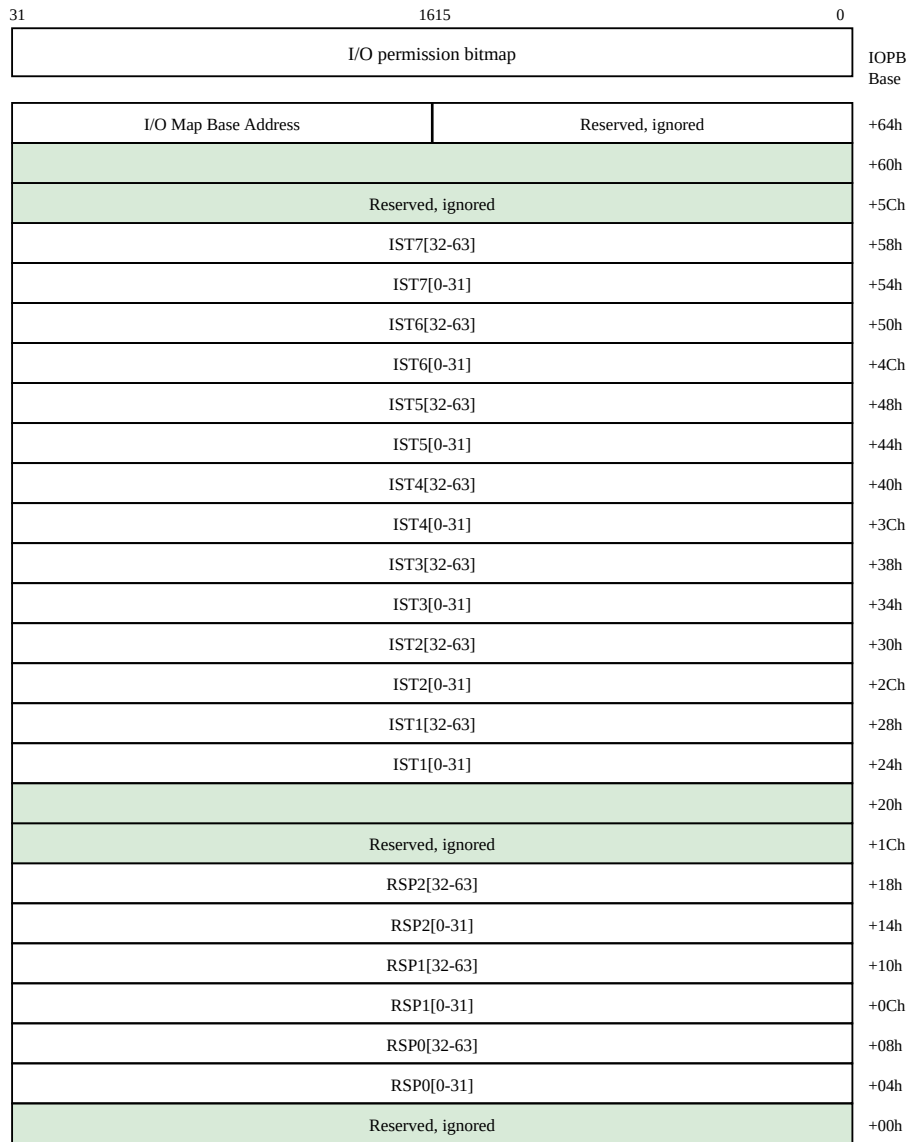


Figure 4.13. 64-bit TSS format

The TSS descriptor is stored in the GDT and is widened to 128 bits so it can hold a 64-bit base address and a 20-bit limit.

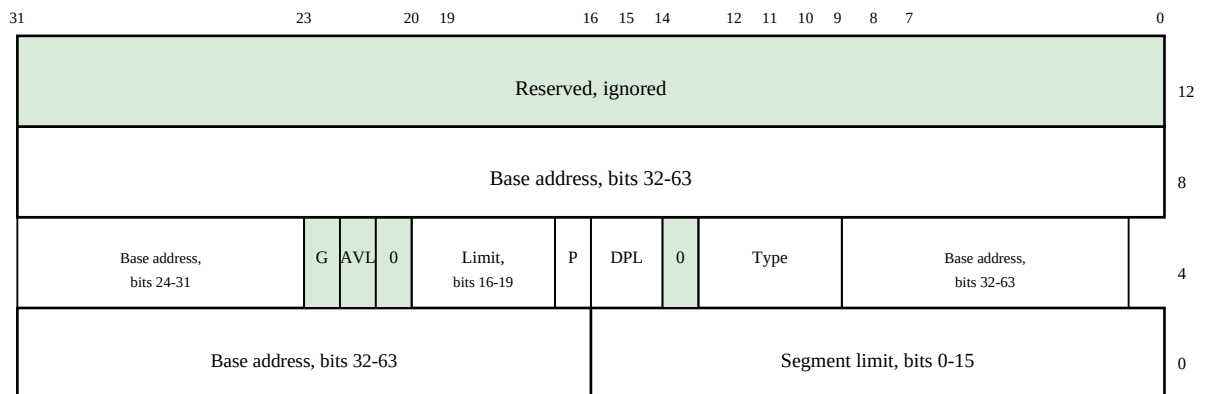


Figure 4.14. System-descriptor format in long mode

*Editorial note: Figure 4.14 prints several base-address ranges inconsistently, including "Base address, bits 32-63" in more than one field. Those printed labels are preserved in the redraw rather than silently corrected.*

The type field must contain 1001b for an available TSS or 1011b for a busy TSS. LTR loads the TSS selector into TR exactly as in protected mode. The descriptor must initially be marked available; the processor automatically marks it busy after LTR.

The Interrupt Stack Table mechanism allows an interrupt or trap gate to select one of seven dedicated stacks. If the gate's IST field is nonzero, the new stack pointer is obtained from IST1-IST7 rather than RSP0-RSP2. The source suggests using such a stack for important interrupts so that the handler has a reliable stack isolated from the normal one and resident in physical memory.

### 4.3.3. Invoking an Interrupt Handler

The principal difference from protected mode is that the previous SS:RSP pair is placed on the handler stack regardless of whether a privilege-level change occurred. Handler code must be 64-bit. When the privilege level changes, SS is loaded with zero because the 64-bit TSS does not contain stack-segment selectors.

The source gives the interrupt-entry sequence as follows:

1. Examine the gate descriptor's IST field. If it is nonzero, load the corresponding IST stack pointer.
2. If IST is zero and the privilege level changes, load the stack pointer for the corresponding privilege level.
3. Align the resulting stack pointer. It is aligned on an 8-byte boundary while also saying that the four low bits are cleared.

*Editorial note: clearing four low bits aligns a pointer on a 16-byte boundary, not an 8-byte boundary. Both statements are preserved here.*

4. If the privilege level changed, load zero into SS.
5. Push the previous SS and RSP onto the new stack. SS is widened to 8 bytes.
6. Push RFLAGS.
7. Clear TF, NT, and RF in RFLAGS.
8. If entry was through an interrupt gate, clear IF; if through a trap gate, leave IF unchanged.
9. Push CS and RIP. CS is widened to 8 bytes.
10. If the interrupt vector supplies an error code, push the error code widened to 8 bytes.
11. Transfer control to the handler address stored in the gate descriptor.

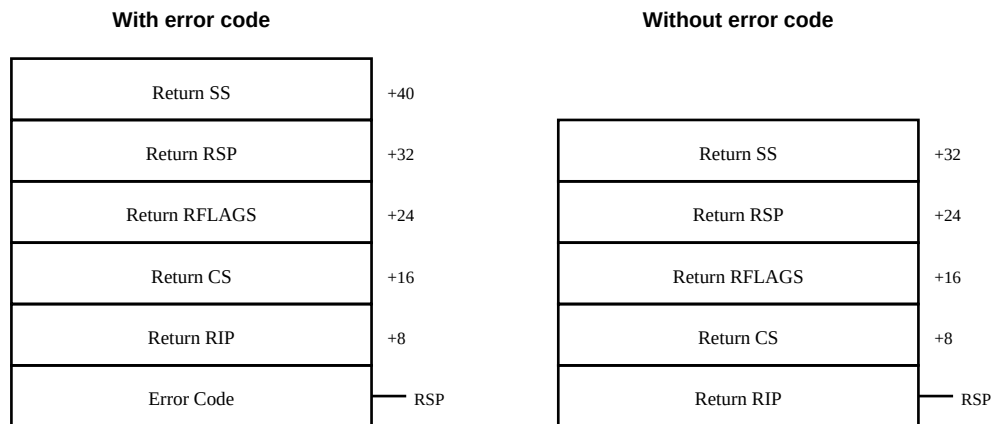


Figure 4.15. Interrupt-handler stack contents

The source emphasizes the zero selector loaded into SS during privilege changes. For nested handler invocations, IRETQ can later restore a zero SS without generating #GP when the destination is 64-bit code and the destination privilege level is numerically less than 3.

### 4.3.4. Practice

The example constructs macros for declaring 64-bit gates, then builds an IDT and TSS, initializes the interrupt controllers, and installs a timer handler.

#### Listing 4.7. Macro for declaring a gate descriptor

```
macro DEFINE_GATE64 selector, offset, IST, gate_type, DPL
{
    dd (offset and 0FFFFh) or (selector shl 16)
    dd (8000h or (DPL shl 13) or (gate_type shl 8) or IST) or ((offset
    shr 16) and 0FFFFh)
    dd (offset shr 32)
    dd 0
}
```

```
}
```

The macro parameters are the handler code-segment selector, entry-point offset, IST value, gate type, and DPL.

#### Listing 4.8. Macros for declaring interrupt- and trap-gate descriptors

```
macro DEFINE_INTGATE64 selector, offset, IST
{
    DEFINE_GATE64 selector, offset, IST, INTGATE64, 0
}
macro DEFINE_TRAPGATE64 selector, offset, IST
{
    DEFINE_GATE64 selector, offset, IST, TRAPGATE64, 0
}
```

INTGATE64 and TRAPGATE64 contain the corresponding type values 1110b and 1111b. Both wrapper macros create gates with DPL zero.

#### Listing 4.9. Macro for declaring a 64-bit TSS descriptor

```
macro DEFINE_TSS64Descr BaseAddress, Limit
{
    dd (Limit and 0FFFFh) or ((BaseAddress and 0FFFFh) shl 16)
    dd ((BaseAddress shr 16) and 0FFh) or (TSS64_type shl 8) or (8 shl
    12) or (Limit and 0F0000h) or (BaseAddress and 0FF000000h)
    dd (BaseAddress shr 32)
    dd 0
}
```

#### Listing 4.10. Declaring the IDT

```
IDT64:
dq 0,0 ; #DE Divide by zero error
. . . . . ; the first 32 vectors are reserved for exceptions
dq 0,0 ; Reserved
DEFINE_INTGATE64 CODE64_SELEKTOR,IRQ0_handler,0 ; 20 IRQ-0: Timer
DEFINE_INTGATE64 CODE64_SELEKTOR,int_EOI,0 ; 21 IRQ-1
. . . . .
DEFINE_INTGATE64 CODE64_SELEKTOR,int_EOI,0 ; 2F IRQ-F
IDT64_Size equ $-IDT64
label IDTR64
dw IDT64_Size-1
dq IDT64
```

*The source's printed comment Divide by zero error is preserved exactly. The Russian comment after the dotted omission has been translated.*

IDTR64 is the in-memory image used with LIDT. IRQ0\_handler and int\_EOI are the handler labels defined later.

#### Listing 4.11. 64-bit TSS structure

```
struct TSS64
{
    .TSSBase:
    dd ?
    .RSP0 dq ?
    .RSP1 dq ?
    .RSP2 dq ?
    dq ?
    .IST1 dq ?
    .IST2 dq ?
    .IST3 dq ?
    .IST4 dq ?
    .IST5 dq ?
    .IST6 dq ?
    .IST7 dq ?
    dq ?
    dw ?
    .IOMapBase dw $-.TSSBase
}
```

The TSS is then declared like an ordinary variable:

#### TSS declaration

```
TSS TSS64
TSS_Size equ $ - TSS
```

#### Listing 4.12. Global Descriptor Table

```
GDT64:
dq 0
db 0FFh,0FFh,00h,00h,00h,10011010b,11001111b,00h
db 0FFh,0FFh,00h,00h,00h,10010010b,11001111b,00h
CODE64Descr db 00h, 00h,00h,00h, 00h,10011000b,00100000b,00h
DEFINE_TSS64Descr TSS, TSS_Size-1
GDT64_size equ $-GDT64
label GDTR64
dw GDT64_size-1
dq GDT64
```

With the structures declared, the example begins initializing memory.

#### Listing 4.13. Memory initialization

```
LM_CODE_START:
mov rdi, PLM4_BASE_ADDRESS
call ZeroPage_4K
```

```

xor rax, rax
mov rbx, rax
call AllocPage_2M
mov eax, 0200000h
mov rbx, rax
call AllocPage_2M
mov eax, 0400000h
mov rbx, rax
call AllocPage_2M
xor rax, rax
mov eax, PLM4_BASE_ADDRESS
mov cr3, rax
mov rsi, message1
mov al, 0
mov ah, 0
mov bl, "5"
call OutText

```

This creates the required paging structures for the first 6 MB of memory. AllocPage\_2M is reused from the previous section. CR3 is then reloaded and the first status message is displayed.

#### Listing 4.14. Initializing the global tables

```

lidt tbyte [IDTR64]
lgdt tbyte [GDTR64]
mov [TSS.RSP0], RING0_Stack
mov ax, TSS_SELECTOR
ltr ax
mov rsi, message2
mov al, 0
mov ah, 1
mov bl, "5"
call OutText

```

The example loads IDTR and reloads GDTR, fills TSS.RSP0, and loads TR. The source remarks that reloading the segment registers after GDTR would also be reasonable, but does not do so.

#### Listing 4.15. Initializing the interrupt controller

```

mov bx, 2820h
mov al, 00010001b
out 020h, al
out 0A0h, al
mov al, bl
out 021h, al
mov al, bh
out 0A1h, al
mov al, 00000100b
out 021h, al
mov al, 2
out 0A1h, al
mov al, 00000001b
out 021h, al
out 0A1h, al
mov rsi, message3
mov al, 0
mov ah, 2
mov bl, "5"
call OutText
in al, 70h
and al, 7Fh
out 70h, al
sti
mov rsi, message4
mov al, 0
mov ah, 3
mov bl, "5"
call OutText
jmp $

```

The PICs are remapped so hardware interrupts use vectors 20h-2Fh, NMI is unmasked through port 70h, and maskable interrupts are enabled.

#### Listing 4.16. IRQ0 interrupt handler

```

IRQ0_handler:
push rax
push rdx
push rbx
inc [counter]
xor edx, edx
xor eax, eax
inc dword [counter]
mov eax, dword [counter]
mov ebx, 18;
div ebx
cmp edx, 0
jnz .cont
inc byte [sec_counter+3]
cmp byte [sec_counter+3], ":"
jnz .cont
mov byte [sec_counter+3], "0"
inc byte [sec_counter+2]

```

```

cmp byte [sec_counter+2], ":"
jnz .cont
mov byte [sec_counter+2], "0"
inc byte [sec_counter+1]
cmp byte [sec_counter+1], ":"
jnz .cont
mov byte [sec_counter+1], "0"
inc byte [sec_counter]
cmp byte [sec_counter], ":"
jnz .cont
mov byte [sec_counter], "0"
mov byte [sec_counter+1], "0"
mov byte [sec_counter+2], "0"
mov byte [sec_counter+3], "0"
.cont:
mov rsi, sec_counter
mov al, 0
mov ah, 4
mov bl, "5"
call OutText
pop rbx
pop rdx
pop rax
jmp int_EOI

int_EOI:
push rax
mov al, 20h
out 020h, al
out 0a0h, al
pop rax
iretq

```

*The dangling semicolon in `mov ebx, 18;` is present in the source and is preserved. The source also increments counter twice at the start of the handler; this is preserved rather than corrected.*

IRQ0\_handler displays the elapsed counter value using the same decimal-digit technique as the earlier protected-mode timer example. int\_EOI is shared by the hardware-interrupt handlers and notifies both interrupt controllers that servicing is complete.

The complete example source was supplied on the book's accompanying CD in the part4 folder.

### 4.3.5. Summary

This section studied interrupts in the processor's 64-bit mode, including gate-descriptor formats, the interrupt table, and the 64-bit TSS. The book then turns to protection mechanisms and other long-mode details not covered in Sections 4.1-4.3.

## 4.4. Protection and Multitasking

The earlier sections of Chapter 4 covered interrupts and virtual memory in the processor's 64-bit mode. This final long-mode section discusses protection, multitasking, and other mechanisms that were not covered previously.

Most long-mode protection mechanisms are unchanged from protected mode; the source therefore concentrates on the differences.

The source remarks that the word 'multitasking' in the section title is somewhat unnecessary because the section does not study multitasking in depth. It also collects several long-mode features not mentioned earlier.

### 4.4.1. Segments

As stated repeatedly earlier, 64-bit mode does not perform ordinary segment-limit checks. Code and data segments begin at linear address zero and cover the entire address space. Protection checks are nevertheless performed when a selector is loaded into a segment register; only after those checks succeed is the selector accepted.

In some situations, notably interrupt entry, the processor permits a null selector to be loaded into SS.

Protected mode allowed null selectors to be loaded into DS, ES, FS, and GS, but a later memory access through such a register generated #GP. In long mode the relevant checking is performed when the selector is loaded. The interrupt-entry case in which the processor itself places a null selector in SS is treated as an exception.

Although ordinary 64-bit code and data segments use base zero, FS and GS may have nonzero base addresses. Their hidden bases can be set through MSRs: IA32\_FS\_BASE at C0000100h and IA32\_GS\_BASE at C0000101h.

The LDT descriptor is widened to 16 bytes so a 64-bit LDT base address can be stored. Its format is the system-descriptor format shown previously in Figure 4.14. An LDT descriptor uses type 0010b. Because normal segmentation is largely unused in 64-bit mode, the source says the practical value of the LDT is correspondingly small.

### 4.4.2. Call Gates

Call gates undergo several changes in long mode. Their descriptor is widened to 16 bytes so that the handler's 64-bit address can be stored. The parameter-count field present in the protected-mode call gate is removed. The type field must contain 1100b.

As with an interrupt handler, call-gate handler code must be 64-bit; the code descriptor used by the handler therefore has L set.

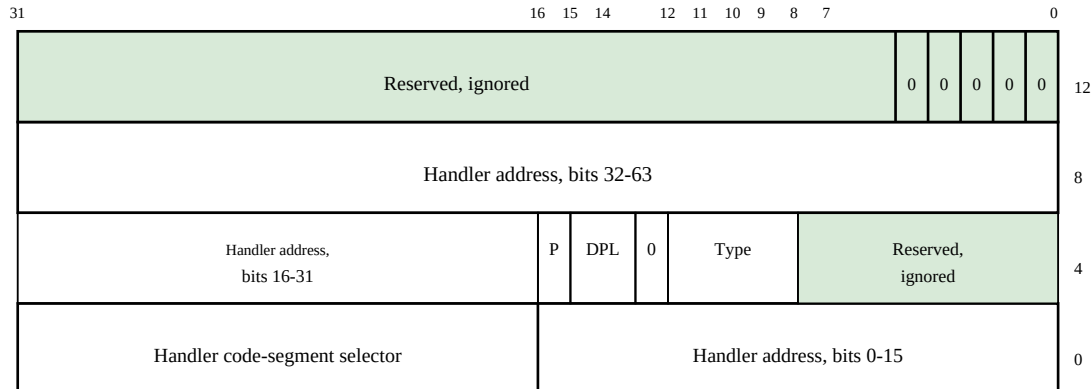


Figure 4.16. Call-gate format in long mode

The processor's actions are broadly similar to interrupt-handler entry. Unlike protected mode, long mode does not copy parameters from the caller's stack to the handler's stack. Parameters, or a pointer to a parameter block, can instead be passed in general-purpose registers.

The source presents call gates as the principal interface between 32-bit compatibility-mode programs and a 64-bit operating system. It argues that AMD processors do not support SYSENTER/SYSEXIT in long mode, while Intel processors do not support SYSCALL/SYSRET in compatibility mode, so call gates provide a common way for compatibility-mode code to invoke 64-bit operating-system services.

*Editorial note: this paragraph is translated according to the 2011 source's vendor comparison. Current processor behavior should be checked against the manuals for the exact AMD or Intel CPU being targeted.*

### 4.4.3. SYSCALL and SYSRET Instructions

Protected mode used SYSENTER and SYSEXIT for fast system-service entry and return. These instructions minimize memory references and therefore reduce the overhead of entering system code. Both vendors support SYSENTER/SYSEXIT in protected mode; in long mode it states that Intel supports them and AMD does not.

Long mode adds SYSCALL and SYSRET. SYSCALL invokes a system procedure and SYSRET returns from it. As with SYSENTER/SYSEXIT in the source's discussion, they are intended for transitions between privilege level 3 and privilege level 0.

AMD supports SYSCALL/SYSRET in protected mode, compatibility mode, and 64-bit mode, whereas Intel supports them only in long mode, including compatibility mode. The following explanation is explicitly based on Intel behavior.

SYSCALL saves the current RIP in RCX and loads RIP from IA32\_LSTAR. It saves the low 32 bits of the flags register in R11. The source then says the processor performs an AND operation on the flags using IA32\_FMASK so that a mask for the handler's flags can be predetermined.

*Editorial note: the source describes the IA32\_FMASK operation simply as AND with the MSR value. Architectural descriptions normally express SYSCALL flag masking differently; the source statement is preserved rather than silently rewritten.*

The target code-segment selector is obtained from IA32\_STAR[47:32]. The DPL in the hidden CS state is forced to zero, so the target runs at privilege level 0 regardless of the descriptor present in the GDT or LDT for that selector. It nevertheless advises the operating system to keep the cached segment state consistent with the descriptor table. SS is formed by adding 8 to the code-segment selector.

SYSRET restores RIP from RCX and the low 32 bits of the flags from R11. For a return to 64-bit mode, the source says CS receives IA32\_STAR[63:48] + 16 and SS receives IA32\_STAR[63:48] + 8. For a return to compatibility mode, CS receives IA32\_STAR[63:48] and SS receives IA32\_STAR[63:48] + 8.

The hidden CS and SS state is set so that the result executes at privilege level 3 regardless of the descriptors stored under those selectors. The operand size selects the return mode in the source's explanation: REX.W selects return to 64-bit mode; without that prefix, the return is to compatibility mode.

The source gives IA32\_STAR, IA32\_LSTAR, and IA32\_FMASK MSR indexes as C0000081h, C0000082h, and C0000084h respectively. Selectors obtained from IA32\_STAR[47:32] and IA32\_STAR[63:48] must be nonzero or #GP is generated.

To test processor support, the source instructs software to call CPUID with index 80000001b and examine bit 29 of EDX. It says this check is meaningful only before long mode is entered, because a processor capable of entering long mode must already support these instructions.

*Editorial note: the CPUID leaf is printed with a binary suffix as 80000001b. The source form is preserved. In normal architectural notation this extended CPUID leaf is written 80000001h.*

SYSCALL/SYSRET are enabled by setting bit 0 of IA32\_EFER.

#### 4.4.4. Multitasking

Long mode does not support hardware task switching, so a 64-bit operating system must implement task switching in software. Windows already used software task switching in protected mode and speculates that the limited use of x86 hardware task switching was one reason not to retain it in the 64-bit architecture.

As a consequence, control-transfer operations cannot use a TSS descriptor as the target of a hardware task switch. The source specifically names far JMP/CALL, IRETQ, and other operations whose destination or return segment would be a TSS selector. LTR remains the way to load TR because it does not itself switch from one task to another.

At least one TSS is still required for privilege-level stack switching and for restricting access to I/O ports. As in protected mode, its descriptor is defined in the GDT and its selector is loaded into TR.

Figure 4.17 repeats the 64-bit TSS layout, this time emphasizing the I/O permission bitmap. IOMapBaseAddress contains the bitmap's offset from the start of the TSS. The bitmap format and I/O permission checks are unchanged from protected mode.

|                       |                   |   |           |
|-----------------------|-------------------|---|-----------|
| 31                    | 1615              | 0 |           |
| I/O permission bitmap |                   |   | IOPB Base |
| IOMapBaseAddress      | Reserved, ignored |   | +64h      |
|                       |                   |   | +60h      |
| Reserved, ignored     |                   |   | +5Ch      |
| IST7[32-63]           |                   |   | +58h      |
| IST7[0-31]            |                   |   | +54h      |
| IST6[32-63]           |                   |   | +50h      |
| IST6[0-31]            |                   |   | +4Ch      |
| IST5[32-63]           |                   |   | +48h      |
| IST5[0-31]            |                   |   | +44h      |
| IST4[32-63]           |                   |   | +40h      |
| IST4[0-31]            |                   |   | +3Ch      |
| IST3[32-63]           |                   |   | +38h      |
| IST3[0-31]            |                   |   | +34h      |
| IST2[32-63]           |                   |   | +30h      |
| IST2[0-31]            |                   |   | +2Ch      |
| IST1[32-63]           |                   |   | +28h      |
| IST1[0-31]            |                   |   | +24h      |
|                       |                   |   | +20h      |
| Reserved, ignored     |                   |   | +1Ch      |
| RSP2[32-63]           |                   |   | +18h      |
| RSP2[0-31]            |                   |   | +14h      |
| RSP1[32-63]           |                   |   | +10h      |
| RSP1[0-31]            |                   |   | +0Ch      |
| RSP0[32-63]           |                   |   | +08h      |
| RSP0[0-31]            |                   |   | +04h      |
| Reserved, ignored     |                   |   | +00h      |

Figure 4.17. 64-bit TSS format with I/O permission bitmap

A software multitasking implementation must maintain a separate memory area for each task containing the general-purpose-register state, flags, and any other context that must survive a switch. Switching tasks therefore requires software to save the current task's state, load another task's state, and resume execution where that task was interrupted.

#### 4.4.5. Practice

The practical example demonstrates SYSCALL/SYSRET with code executing at privilege levels 0 and 3. Because ring-3 code must access the pages used by the example, the page-allocation routine from the previous section must be modified so U/S is set in every paging structure describing those pages. The modified AllocPage\_2M routine is not printed in the book.

After allocating the required memory, the program replaces the GDT. The GDT must be arranged specially because of the selector relationships used by SYSCALL/SYSRET.

#### Listing 4.17. Global Descriptor Table

```
GDT64:
dq 0
db 0FFh,0FFh,00h,00h,00h,10011010b,11001111b,00h ; 08
db 0FFh,0FFh,00h,00h,00h,10010010b,11001111b,00h ; 10
CODE64R0_Descr db 00h, 00h,00h,00h, 00h,10011000b,00100000b,00h ;18
DATA64R0_Descr db 0FFh,0FFh,00h,00h,00h,10010010b,11001111b,00h ;20
DATA64R3_Descr db 0FFh,0FFh,00h,00h,00h,11110010b,11001111b,00h ;28
CODE64R3_Descr db 00h, 00h,00h,00h, 00h,11111000b,00100000b,00h ;30
GDT64_size equ $-GDT64
label GDTR64
dw GDT64_size-1
dq GDT64
```

The first two descriptors are inherited from the protected-mode bootstrap. They are followed by two descriptors for ring 0 and two for ring 3.

The example then sets IA32\_EFER bit 0, programs the base selectors in IA32\_STAR, places the ring-0 handler address in IA32\_LSTAR, and writes the ring-0 flags mask to IA32\_FMASK.

#### Listing 4.18. Initializing the SYSCALL/SYSRET mechanism

```
mov ecx, IA32_EFER
rdmsr
or eax, 1
wrmsr ; enable syscall/sysret mechanism
pushf
xor r11, r11
pop r11 ; r11 - RFLAGS image
mov ecx, IA32_STAR
xor eax, eax
mov edx, 000230018h
wrmsr
mov ecx, IA32_LSTAR
mov eax, RING0_CODE
xor edx, edx
wrmsr
mov ecx, IA32_FMASK
mov eax, r11d
xor edx, edx
wrmsr
mov rsi, message3
mov al, 0
mov ah, 2
mov bl, "5"
call OutText
xor rcx, rcx
mov ecx, RING3_CODE
;sysretq
db 48h, 0Fh, 07h
```

*The source spelling mechanism in the WRMSR comment is preserved exactly. The final SYSRET is emitted manually as the byte sequence shown by the source.*

The high 16 bits used as the ring-3 base selector in IA32\_STAR must contain RPL = 3. At the end of Listing 4.18, RCX receives the ring-3 code address and SYSRET transfers execution to ring 3.

SYSRET is written as the bytes 48h, 0Fh, 07h. The source explains that 48h is the REX.W prefix and 0F07h is the SYSRET opcode; explicitly encoding the 64-bit form is intended to guarantee return to 64-bit mode rather than compatibility mode.

#### Listing 4.19. Ring-3 code and SYSCALL handler

```
RING3_CODE:
mov ax, 02Bh
mov ds, ax
@@:
inc dword [R3_c]
mov rsi, message4
mov al, 0
mov ah, 3
mov bl, "5"
call OutText
syscall
jmp @b
RING0_CODE:
mov ax, 20h
mov ds, ax
inc dword [R0_c]
mov rsi, message5
mov al, 0
mov ah, 4
mov bl, "5"
call OutText
;sysretq
db 48h, 0Fh, 07h
message1 db "We in long mode!", 0
message2 db "GDT and TR is updated", 0
message3 db "SYSCALL/SYSRET mechanism was initialized", 0
message4 db "RING3 code running "
R3_c dd 0
db 0
message5 db "RING0 code running "
R0_c dd 0
db 0
```

The ring-3 and ring-0 blocks simply display messages and increment their counters before each display. Changing non-text glyphs beside the two messages indicate that the program is running.

The program and complete source code were supplied on the accompanying CD in the part4 folder.

### 4.4.6. Summary

This final section on 64-bit processor operation covered long-mode call gates and the fast system-call mechanism based on SYSCALL/SYSRET. The next chapter turns to assembly-language programming under 64-bit Windows.

# Chapter 5. Programming in Win64

## 5.1. Introduction to Win64

The previous chapter discussed the processor's 64-bit operating mode. Chapter 3 discussed assembly-language programming under the 32-bit Windows family. Having examined the architecture of processor operation in 64-bit mode, the book now turns to programming under 64-bit Windows, or Win64.

The first 64-bit versions of Windows appeared with Windows XP and Windows 2003 Server, almost simultaneously, in 2005. Later Windows systems such as Windows 2008 Server and Vista were available in both 32-bit and 64-bit versions. Only the WinNT family had 64-bit versions; Win9x did not.

### 5.1.1. Advantages and Disadvantages

The principal advantage of 64-bit Windows is support for much larger amounts of virtual and physical memory. The source presents this as especially important for server systems running memory-intensive applications such as database management systems. A 32-bit application normally has access to 2 GB of virtual memory, or 3 GB in some configurations. AWE can provide access to more than 2 GB, but the author regards it as an awkward substitute for a large flat address space.

A 64-bit Windows application can access 8 TB of virtual memory, with the possibility that the limit could be increased in future versions. It regards this as more than sufficient for memory-intensive server applications of the period.

A second advantage given by the source is computational speed. It attributes this partly to an expanded floating-point instruction set and partly to faster work with double-precision and QWORD-sized values, describing operations that formerly required several steps as now completing in one.

*Historical note: this paragraph translates the source's broad performance claim as written. Actual performance depends on the instruction sequence, processor generation, compiler or assembler choices, and workload.*

The source also lists disadvantages. First, it says 64-bit instructions are at least one byte longer, increasing executable-code size. Second, the additional paging structures used to describe virtual memory create memory overhead; as more virtual memory is used, more space is consumed by PML4, PDPT, PD, and PT structures.

*Editorial note: not every instruction in 64-bit mode necessarily becomes one byte longer. The book's statement is preserved as a general source claim rather than silently corrected.*

Ordinary desktop workloads of the period rarely exceeded 1 GB of total memory use and says that describing 1 GB of memory in Win64 consumes at least twice as much physical memory as in Win32. On that basis, it concludes that 64-bit Windows offered little practical advantage for everyday use and was not yet widespread among ordinary users.

*Historical note: this reflects the source's 2011-era usage assumptions and adoption levels, not current Windows usage.*

The source then points out that the number of general-purpose registers has doubled. Well-written code can therefore keep more intermediate values in registers and reduce memory accesses. The trade-off is that a context switch must save and restore more register state.

### 5.1.2. Memory in Win64

Virtual memory in Win64 remains organized around a private virtual address space for each process. Memory is still divided between user and kernel space, but no longer into equal halves.

Ring-3 memory is given as the range 0h-80000000000h. The lowest and highest 64 KB of that range are unavailable, leaving 8 TB minus 128 KB. Kernel memory is given as 0000080000000000h-FFFFFFFFFFFFFFFFh.

At first this appears to be an enormous range. The source then notes that long-mode addresses must be canonical and that only 48 address bits are effectively used. It therefore treats the kernel range as equivalent to FFFF080000000000h-FFFFFFFFFFFFFFFFh. The high 16 address bits may be all zeroes or all ones; the source says kernel addresses are conventionally written with ones in the upper bits.

The resulting totals given by the book are 8 TB of virtual memory for ring-3 applications and 248 TB for the kernel.

*Historical note: these are the virtual-address ranges and limits described by the source for the Windows versions it covers. They should not be treated as universal limits for every modern 64-bit Windows release.*

### 5.1.3. Calling Convention

Unlike Win32, Win64 uses a standardized fastcall calling convention for operating-system functions. The source treats the calling convention as the most important low-level difference between Win32 and Win64. User-defined functions may of course use another private convention, but Win64 API calls follow the platform convention.

The first four parameters are passed in RCX, RDX, R8, and R9 respectively. Additional parameters are passed on the stack in reverse order. Space is always reserved on the stack for the first four parameter positions, even if a function has fewer than four parameters. This allows the register parameters to be spilled to the stack and accessed in the older stack-frame style while freeing the registers for other work.

A function result is returned in RAX, or in XMM0 when the result is floating-point.

The 32 bytes reserved for the first four register parameters are called the shadow area. They are reserved by subtracting 32 from RSP because the stack grows downward. The caller is responsible for cleaning up both stack arguments and the shadow area.

The stack pointer must be aligned to a 16-byte boundary. Misalignment often appears to work, but some functions can raise an exception and, more generally, misalignment can reduce application performance.

The source additionally says that floating-point values appearing among the first four parameters are duplicated through XMM0L-XMM3L. Parameters smaller than 64 bits are extended to 64 bits with zeroes or, for negative signed values, ones before being placed in a register or on the stack.

*Editorial note: the treatment of floating-point duplication is reproduced from the book. The exact Microsoft x64 rules distinguish ordinary prototyped calls from varargs/unprototyped calls, so the source's simplified description should not be read as a complete ABI specification.*

For a six-parameter function `function (int1, real1, int2, real2, int3, real3)`, the source gives the following sequence:

#### Source example of a Win64 call

```
1. push real3
2. push int3
3. R9=real2, XMM3L=real2
4. R8=int2
5. RDX= real1, XMM1L=real1
6. RCX=int1
7. RSP= RSP-32
8. ;Function call
9. ;Stack cleanup (if necessary)
```

In practice, the source says stack space is usually reserved at the beginning and additional arguments are written with MOV rather than PUSH. Since the caller performs cleanup, a function can reserve enough outgoing-argument space once for the largest call it will make, reducing repeated changes to RSP.

The book says PUSH is rarely encountered in Win64 call setup and that instructions of the form `mov [esp+n], val` are used instead. It further claims MOV is faster because it writes memory without also updating the stack pointer.

*Editorial note: the MOV-versus-PUSH speed statement is a broad source claim and is not generalized here to all modern microarchitectures. The printed source also uses ESP in the example form even though the surrounding discussion is about Win64; that wording is retained in the translation.*

The source concludes that careful Win64 code can minimize both RSP modifications and stack accesses, improving procedure-call performance.

FASM already supplies macros that hide most of these mechanics and allow functions to be called with one line while also simplifying the definition of Win64 fastcall-compatible procedures.

The volatile registers listed by the source are RAX, RCX, RDX, R8-R11, and XMM0-XMM5. A caller that needs their values after a call must save them. The nonvolatile registers listed are RBX, RSI, RDI, RSP, RBP, R12-R15, and XMM6-XMM15; a callee that uses them must restore them before returning. Callback functions must obey the same rules.

Function prologues and epilogues follow the same general principles as in Win32 and continues to describe RBP as the stack-frame pointer. Floating-point parameter passing through XMM registers is not developed further in this edition.

### 5.1.4. Compatibility Mode

For backward compatibility, older ring-3 Win32 applications can run under Win64 without major restrictions. Kernel-mode drivers, however, must be 64-bit. The source therefore notes that products such as antivirus and firewall software that install drivers must also provide 64-bit drivers.

All 32-bit applications run under the WOW64 subsystem. When such a program calls system services or functions, the source says the calls first pass through their 32-bit counterparts and are then redirected to 64-bit implementations.

Old 32-bit applications normally retain access to the lower 2 GB, while specially built applications may access all 4 GB. 32-bit and 64-bit code cannot be mixed within one process: a 32-bit program cannot load a 64-bit DLL, and a 64-bit program cannot load a 32-bit DLL.

Where a system DLL has the same filename in both architectures, the source says WOW64 automatically redirects the request to the appropriate 32-bit library.

### 5.1.5. Win64 API and System Libraries

The DLL-based API system changes very little from Win32. Most functions keep the same names and parameter meanings, but the sizes of platform-dependent parameters change; pointers, for example, become 64-bit.

The names and purposes of the principal system libraries also remain the same. The internal code changes, but names such as `kernel32.dll` are retained rather than being renamed to something like `kernel64.dll`.

### 5.1.6. Practice

The practical example creates a 64-bit Windows application that displays the familiar message 'Hello world!'.

#### Listing 5.1. Application source code

```

format PE64 GUI 5.0
entry start
include 'win64a.inc'

section '.data' data readable writeable
text db 'Hello world!',0

section '.code' code readable executable
start:
sub rsp, 32
mov rcx, 0
mov rdx, text
mov r8, text
mov r9, 0
call [MessageBox]

xor rcx, rcx
call [ExitProcess]

section '.idata' import data readable writeable
library kernel32, 'KERNEL32.DLL', \
user32, 'USER32.DLL'
import kernel32, \
ExitProcess, 'ExitProcess'
import user32, \
MessageBox, 'MessageBoxA'

```

The only instruction the source expects to be unclear at first glance is `sub rsp, 32`. The Win64 calling convention requires space to be reserved for four register-parameter slots even when the called function has fewer than four parameters.

There is no need for `add rsp, 32` before `ExitProcess` because `ExitProcess` does not return. In any subroutine or function that calls another fastcall function and then returns normally, the stack pointer must be restored; otherwise `RET` would no longer find the correct return address.

The source then shows the same application using FASM's `invoke` macro.

#### Listing 5.2. Application source code using the `invoke` macro

```

format PE64 GUI 5.0
entry start
include 'win64a.inc'

section '.data' data readable writeable
text db 'Hello world!',0

section '.code' code readable executable
start:
invoke MessageBox,0,text,text,0
invoke ExitProcess,0

section '.idata' import data readable writeable
library kernel32, 'KERNEL32.DLL', \
user32, 'USER32.DLL'
import kernel32, \
ExitProcess, 'ExitProcess'
import user32, \
MessageBox, 'MessageBoxA'

```

The `invoke` macro hides the details of the Win64 API calling convention and makes the program look almost unchanged from the Win32 version. The main visible difference is the executable format: PE64, also referred to by the source as PE32+.

### 5.1.7. Summary

This section introduced 64-bit Windows and the principal differences from Win32. The next section examines several aspects of assembly-language programming under Win64 in greater detail.

## 5.2. Programming in Win64

Programming in Win64 and Win32 does not differ fundamentally. The differences become noticeable mainly in low-level programming, particularly in assembly language. In high-level languages, programs for Win32 and Win64 differ very little; in many cases the programmer mainly recompiles the application for Win64. This section examines the low-level differences in greater detail, both for ring 3 and ring 0.

### 5.2.1. Changes in Data Types

Many Windows data types change size in Win64. The source gives only general information and recommends consulting the C/C++ header files supplied with the SDK and DDK/WDK for exact definitions. For assembly programming, it treats those headers as the primary source of information about Windows data types and their sizes.

All pointers are now 8 bytes. In Microsoft terminology, this means that types whose names begin with the letter P are generally 8-byte pointer types.

Handles are also 8 bytes. The source lists HANDLE, HBRUSH, HBITMAP, HCOLORSPACE, HCURSOR, HDC, HFONT, HICON, HINSTANCE, HKEY, HMENU, HMODULE, HPEN, HPALETTE, HWND, and similar handle types.

The window-message parameters WPARAM and LPARAM are likewise 8 bytes in Win64.

Types whose size is explicit in their names, or whose sizes are historically fixed, do not change. The source gives CHAR as 1 byte; UINT, UINT32, ULONG, and ULONG32 as 4-byte unsigned integers; UINT64, ULONG64, and ULONGLONG as 8-byte unsigned integers; and BOOL as 4 bytes.

The same pointer-size changes affect kernel programming. Because kernel interfaces use many pointers, most pointer-valued parameters become 8 bytes, except for explicitly fixed-size types such as those listed above.

### 5.2.2. Stack Alignment

The previous chapter briefly mentioned 16-byte stack alignment without explaining why it is required. An unaligned stack does not necessarily cause an immediate error or exception, but Microsoft nevertheless requires Win64 programs to keep RSP aligned on a 16-byte boundary.

The reason given is the use of 128-bit XMM registers. Because an XMM register is 16 bytes wide, storing it at a 16-byte-aligned memory address is preferable. System interrupt and system-call handlers save XMM registers on the interrupted task's stack, so keeping the stack aligned allows those registers to be saved and restored efficiently.

The simple example in Section 5.1 ignored alignment, but more complicated programs should not do so because performance can suffer.

*Editorial note: the book presents stack alignment largely as a performance issue and says misalignment causes no error or exception in general. The exact Microsoft x64 ABI requirements are stricter for calls and for instructions that require aligned operands; the source's framing is preserved here.*

### 5.2.3. GUI Applications

As a practical ring-3 Win64 example, the book constructs a windowed program analogous to the one used earlier in Section 3.2.

#### Listing 5.3. Creating the application window

```
start:
sub rsp, 8*5 ; align stack and alloc space for 4 parameters
xor rcx, rcx
call [GetModuleHandle]
mov [hInst], rax
mov [wc.style], CS_HREDRAW + CS_VREDRAW + CS_GLOBALCLASS
mov rbx, WndProc
mov [wc.lpfnWndProc], rbx
mov [wc.cbClsExtra], 0
mov [wc.cbWndExtra], 0
mov [wc.hInstance], rax
mov rdx, IDI_APPLICATION
xor rcx, rcx
call [LoadIcon]
mov [wc.hIcon], rax
mov rdx, IDC_ARROW
xor rcx, rcx
call [LoadCursor]
mov [wc.hCursor], rax
mov [wc.hbrBackground], COLOR_BACKGROUND+1
mov qword [wc.lpszMenuName], 0
mov rbx, szClassName
mov qword [wc.lpszClassName], rbx
mov rcx, wc
call [RegisterClass]

sub rsp, 8*8 ; alloc place in stack for 8 parameters
xor rcx, rcx
mov rdx, szClassName
mov r8, szTitleName
mov r9, WS_OVERLAPPEDWINDOW
mov qword [rsp+8*4], 50
mov qword [rsp+8*5], 50
```

```

mov qword [rsp+8*6], 300
mov qword [rsp+8*7], 250
mov qword [rsp+8*8], rcx
mov qword [rsp+8*9], rcx
mov rbx, [hInst]
mov [rsp+8*10], rbx
mov [rsp+8*11], rcx
call [CreateWindowEx]
mov [main_hwnd], rax

xor rcx, rcx
mov rdx, button_class
mov r8, AboutTitle
mov r9, WS_CHILD
mov qword [rsp+8*4], 50
mov qword [rsp+8*5], 50
mov qword [rsp+8*6], 200
mov qword [rsp+8*7], 50
mov rbx, [main_hwnd]
mov qword [rsp+8*8], rbx
mov qword [rsp+8*9], rcx
mov rbx, [hInst]
mov [rsp+8*10], rbx
mov [rsp+8*11], rcx
call [CreateWindowEx]
mov [AboutBtnHandle], rax

xor rcx, rcx
mov rdx, button_class
mov r8, ExitTitle
mov r9, WS_CHILD
mov qword [rsp+8*4], 50
mov qword [rsp+8*5], 150
mov qword [rsp+8*6], 200
mov qword [rsp+8*7], 50
mov rbx, [main_hwnd]
mov qword [rsp+8*8], rbx
mov qword [rsp+8*9], rcx
mov rbx, [hInst]
mov [rsp+8*10], rbx
mov [rsp+8*11], rcx
call [CreateWindowEx]
mov [ExitBtnHandle], rax
add rsp, 8*8 ; free place in stack

mov rdx, SW_SHOWNORMAL
mov rcx, [main_hwnd]
call [ShowWindow]
mov rcx, [main_hwnd]
call [UpdateWindow]

mov rdx, SW_SHOWNORMAL
mov rcx, [AboutBtnHandle]
call [ShowWindow]
mov rdx, SW_SHOWNORMAL
mov rcx, [ExitBtnHandle]
call [ShowWindow]

msg_loop:
xor r9, r9
xor r8, r8
xor rdx, rdx
mov rcx, msg
call [GetMessage]
cmp rax, 1
jb end_loop
jne msg_loop
mov rcx, msg
call [TranslateMessage]
mov rcx, msg
call [DispatchMessage]
jmp msg_loop

end_loop:
xor rcx, rcx
call [ExitProcess]

```

The source deliberately avoids the `invoke` macro in this example so that the Win64 calling convention is visible. It notes that `invoke` can generate repeated `sub rsp, n / add rsp, n` sequences, while outgoing argument space can instead be reserved once or only a few times. The author does not present the manual approach as inherently more correct; it is simply an alternative.

At program entry, `sub rsp, 8*5` reserves the 32-byte area for the first four parameters and an additional 8 bytes for alignment, for a total of 40 bytes. Before `CreateWindowEx`, another 64 bytes are reserved for its eight stack parameters; the function takes twelve parameters in total, and the first four positions are already covered by the previous reservation.

The main program is followed by the window procedure.

#### Listing 5.4. Window procedure

```

proc WndProc hwnd, wmsg, wparam, lparam
sub rsp, 8*4 ; alloc space for 4 parameters
cmp rdx, WM_DESTROY
je .wmdestroy
cmp rdx, WM_COMMAND

```

```

jne .default
mov rax, r8
shr rax, 16
cmp rax, BN_CLICKED
jne .default
cmp r9, [AboutBtnHandle]
je .about
cmp r9, [ExitBtnHandle]
je .wmdestroy
.default:
call [DefWindowProc]
jmp .finish
.about:
xor rcx, rcx
mov rdx, AboutText
mov r8, AboutTitle
xor r9, r9
call [MessageBox]
jmp .finish
.wmdestroy:
xor rcx, rcx
call [ExitProcess]
.finish:
add rsp, 8*4 ; restore stack
ret
endp

```

The window function reserves space for the four parameter positions required by DefWindowProc and MessageBoxA. It first checks the message code in RDX. If the message is neither WM\_DESTROY nor WM\_COMMAND, it calls DefWindowProc.

The source points out that the parameters do not need to be reloaded before DefWindowProc because they are already in the same register positions used by another window procedure.

For the window procedure itself, the source says no additional 8-byte alignment adjustment is needed. The system calls the procedure with an aligned stack, CALL pushes the return address, and the proc-generated prologue saves RBP, restoring 16-byte alignment before nested fastcall calls are made.

## 5.2.4. Driver Programming

Win64 driver programming adds restrictions connected with technology introduced to protect system code and data. Kernel-mode drivers in 64-bit Windows are prohibited from modifying system-function code and critical system data structures such as the GDT and IDT; violating the rule causes a blue screen.

*Historical note: the source is describing the 64-bit Windows kernel-protection mechanisms of its period. It does not name the mechanism explicitly in this passage.*

The practical ring-0 example is analogous to the driver from Section 3.3, except that it sends the interrupt descriptor table to a ring-3 application rather than the global descriptor table.

### Listing 5.5. Main IRP handler of the idtdump driver

```

proc DispatchControl pDeviceObject, pIrp
local status:DWORD
push rdi
push rsi
push rbx
sub rsp, 8 ; for align stack
mov [status], STATUS_DEVICE_CONFIGURATION_ERROR
virtual at rdx ; rdx = pIRP
.rdx IRP
end virtual
mov rsi, [.rdx.Tail.Overlay.CurrentStackLocation]
virtual at rsi
.rsi IO_STACK_LOCATION
end virtual
cmp [.rsi.Parameters.DeviceIoControl.IoControlCode], IOCTL_DUMP_IDT
jz .IOCTL_DUMP
jmp .next

.IOCTL_DUMP:
sidt tword [IDTR]
xor rcx, rcx
mov cx, [IDTR.Limit]
inc rcx ; ecx = IDT size
xor rax, rax
xor rbx, rbx
mov ebx, [.rsi.Parameters.DeviceIoControl.InputBufferLength]
cmp ebx, 8
jnz .copydata
mov rax, [.rdx.AssociatedIrp.SystemBuffer]
mov ebx, [rax+4] ; ebx = needed data size
mov eax, [rax] ; eax = start offset in GDT
add rbx, rax
cmp rbx, rcx
jna @f
mov rbx, rcx
@@:
sub rbx, rax ; ebx = fact data size
cmp rbx, 7FFFFFFFh
jna @f
mov rbx, 0
@@:

```

```

mov rcx, rbx

.copydata:
mov [status], STATUS_BUFFER_TOO_SMALL
cmp ecx, [.rsi.Parameters.DeviceIoControl.OutputBufferLength]
ja .error
mov rbx, rcx ; save rcx in rbx, rbx = data size
mov rsi, [IDTR.Address]
add rsi, rax
mov rdi, [.rdx.AssociatedIrp.SystemBuffer]
rep movsb
mov [.rdx.IoStatus.Information], rbx
mov [status], STATUS_SUCCESS
jmp .exit

.next:
jmp .exit

.error:
mov [.rdx.IoStatus.Information], 0

.exit:
mov eax, [status]
mov [.rdx.IoStatus.Status], eax
invoke IoCompleteRequest, rdx, IO_NO_INCREMENT
mov eax, [status]
add rsp, 8
pop rbx
pop rsi
pop rdi
ret
endp

```

*Editorial note: Listing 5.5 retains the comment `eax = start offset in GDT` even though this example is dumping the IDT. The source comment is not silently changed.*

The format, size, and purpose of the application's request parameters are unchanged from the 32-bit driver. The starting offset and requested copy size remain 4-byte values, so the source says the range-checking code needs little change.

The IRP\_MJ\_CREATE handler, which is identical in structure to the IRP\_MJ\_CLOSE handler and DriverEntry are shown next.

#### Listing 5.6. IRP\_MJ\_CREATE handler and entry point of the ittdump driver

```

proc DispatchCreate pDeviceObject, pIrp
virtual at rdx
.rdx IRP
end virtual
mov [.rdx.IoStatus.Status], STATUS_SUCCESS
and [.rdx.IoStatus.Information], 0
sub rsp, 32
mov rcx, rdx
mov rdx, IO_NO_INCREMENT
call [IoCompleteRequest]
add rsp, 32
mov eax, STATUS_SUCCESS
ret
endp

proc DriverEntry pDriverObject, pusRegistryPath
local status:DWORD
mov [pDriverObject], rcx
mov [status], STATUS_DEVICE_CONFIGURATION_ERROR
invoke IoCreateDevice, [pDriverObject], 0, DeviceName, FILE_DEVICE_
UNKNOWN, 0, FALSE, DeviceObject
cmp eax, STATUS_SUCCESS
jnz .END
invoke IoCreateSymbolicLink, SymbolicLinkName, DeviceName
cmp eax, STATUS_SUCCESS
jnz .simlinkfail
mov rax, [pDriverObject]
virtual at rax
.rax DRIVER_OBJECT
end virtual
mov rdx, DispatchCreate
mov [.rax.MajorFunction+IRP_MJ_CREATE*8], rdx
mov rdx, DispatchClose
mov [.rax.MajorFunction+IRP_MJ_CLOSE*8], rdx
mov rdx, DispatchControl
mov [.rax.MajorFunction+IRP_MJ_DEVICE_CONTROL*8], rdx
mov rdx, DriverUnloadHandler
mov [.rax.DriverUnload], rdx
mov [status], STATUS_SUCCESS
jmp .END

.simlinkfail:
invoke IoDeleteDevice, [DeviceObject]

.END:
mov eax, [status]
ret
endp

```

*Listing 5.6 preserves the source's line break in `FILE_DEVICE_ / UNKNOWN`, because it is printed that way in the book. In a real source file that token would normally appear as the single identifier `FILE_DEVICE_UNKNOWN`.*

The 64-bit driver is otherwise not very different from the 32-bit version. Stack alignment still applies in ring 0. DispatchControl saves RBX, RSI, and the source's printed 'RBI' in its prose description; after three pushes the stack is unaligned, so the routine subtracts another 8 bytes from RSP.

*Editorial note: the prose prints RBI when discussing the saved registers, although Listing 5.5 actually pushes RDI. The typo is retained here as a source note rather than silently normalized.*

The author argues that subtracting 8 from RSP is preferable to pushing an unnecessary value merely for alignment, because the extra push would perform a memory access. The source then makes the broader claim that long-mode address translation takes twice as long as 32-bit translation, and uses this to reinforce a general optimization rule: avoid memory accesses whenever possible.

*Editorial note: the claim that 64-bit address translation inherently takes twice as long as 32-bit translation is presented by the book as a general statement. It is preserved, but should not be treated as a universal property of modern CPUs.*

When local variables are used, the source says separate stack-alignment code is unnecessary because FASM's proc macro rounds the space allocated for locals to a multiple of 16.

The controlling user-mode application changes very little. It now obtains the contents of the interrupt-gate descriptor responsible for a page fault and identifies that as vector 13.

*Editorial note: x86 page faults use exception vector 14 (#PF), while vector 13 is #GP. The book prints vector 13 here; the translation preserves the source statement and flags the discrepancy.*

The complete source code for the examples in this section was supplied on the accompanying CD in the part5 folder.

### 5.2.5. Debugging Applications in Win64

That, compared with 32-bit Windows, 64-bit Windows had a much narrower selection of debugging programs for both ring 3 and ring 0. It recommends WinDbg as the most suitable debugger for 64-bit Windows in either environment.

For kernel debugging, the source says two computers are still required unless a virtual machine is used.

*Historical note: this describes the debugging-tool landscape at the time of the 2011 source.*

### 5.2.6. Summary

This section covered the fundamentals of programming under 64-bit Windows and was primarily practical. It produced a ring-3 windowed application and a ring-0 driver.

# Chapter 6. Multiprocessor Systems

## 6.1. Working with APIC

We have studied the two main processor modes and learned how to switch between them. Modern computers, however, commonly contain processors with several cores, while servers may contain several physical processors. This chapter therefore turns to interprocessor communication.

The foundation of interprocessor communication is the Advanced Programmable Interrupt Controller, or APIC. Before studying communication between processors, the book first examines APIC itself.

### 6.1.1. General Overview

APIC replaces the older interrupt-controller arrangement built from two i8259A chips. The name APIC is not entirely precise for the controller described here: earlier APIC versions were used with Pentium-class processors, while the newer version used by Pentium 4 and Xeon systems is xAPIC. In the remainder of the chapter, the term APIC means xAPIC.

By default, APIC operates in a compatibility mode with the older i8259A controller. APIC has two main parts: a Local APIC and an I/O APIC. Every processor or core has its own Local APIC. The I/O APIC is shared and receives interrupts from external devices, then routes them to Local APICs for further processing.

The source presents APIC as fundamental to multiprocessor interrupt handling and therefore to practical interprocessor communication.

APIC is controlled through memory-mapped registers. Reading or writing a particular memory address therefore reads or writes a device register rather than ordinary RAM. Such device-register ranges are normally placed in a reserved high-memory region. Local APIC registers are usually mapped beginning at 0FEE0000h, although the base can be changed through an MSR.

The Local APIC can:

1. Handle interrupts from internal sources such as the timer and thermal sensor.
2. Handle interrupts from external devices through the I/O APIC.
3. Send interprocessor interrupts, discussed in Section 6.2.
4. Handle internal APIC errors.

The practical example later in this section receives interrupts from the Local APIC timer and keyboard interrupts routed through the I/O APIC.

### 6.1.2. Enabling APIC

Local APIC support can be detected with CPUID. The source instructs software to call CPUID with EAX = 1 and test bit 9 of EDX; if it is set, a Local APIC is present. It remarks that APIC support was already common enough that this check was becoming largely formal.

APIC is enabled through the IA32\_APIC\_BASE MSR, index 1Bh. This MSR also exposes the physical base address of the APIC register region.

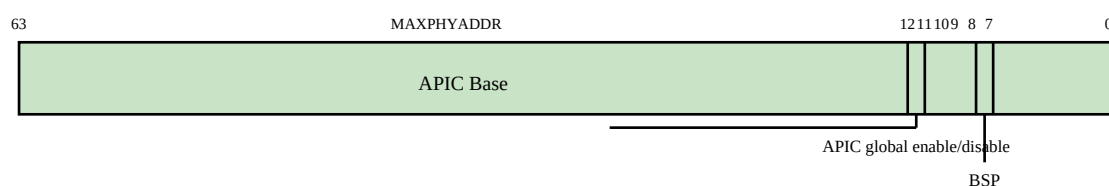


Figure 6.1. IA32\_APIC\_BASE register

The APIC Base field contains the base address of the APIC registers. Because the base is page-aligned, the low 12 address bits are not stored. The source gives the power-on/reset base as 0FEE0000h and says the maximum address width is model-dependent but cannot exceed 52 bits.

*Editorial note: elsewhere in the same section the Local APIC register base is written as 0FEE00000h. The shorter 0FEE0000h form above is the value printed in the source and is retained rather than silently normalized.*

The APIC global enable/disable bit controls global APIC operation. A one enables the APIC controllers; a zero disables them. The BSP field indicates whether the current processor is the bootstrap processor - the processor that starts first after power-on or reset. Section 6.2 returns to this flag.

APIC should be accessed only after it has been enabled and states that disabling APIC resets it to its initial state. If the base address is to be changed, the book recommends doing so as part of the enabling operation by writing the required value into the APIC Base field.

6.1.3. Local APIC ID

Every Local APIC has an identifier used to distinguish it from the others. The I/O APIC uses Local APIC IDs when routing external interrupts among processors.

The Local APIC ID register is at offset +20h from the Local APIC base, or 0FEE00020h at the default base.



Figure 6.2. Local APIC ID register

The source assigns eight bits to the APIC ID and therefore states that the motherboard can contain up to 256 processors or cores. It notes that changing the Local APIC ID is not supported by every processor.

6.1.4. Local Vector Table

Every Local APIC has its own Local Vector Table (LVT). The LVT specifies an interrupt vector and delivery type for each local interrupt source. The source describes six 32-bit LVT registers:

- 1. LVT Timer Register (+320h) - APIC timer interrupts.
- 2. LVT Thermal Monitor Register (+330h) - thermal-sensor interrupts.
- 3. LVT Performance Counter Register (+340h) - performance-monitor interrupts.
- 4. LVT LINT0 Register (+350h) and LVT LINT1 Register (+360h) - the LINT0 and LINT1 inputs.
- 5. LVT Error Register (+370h) - interrupts generated when the controller detects an internal error.

LINT0 and LINT1 may be connected to external interrupt sources. They can be configured independently for each processor.

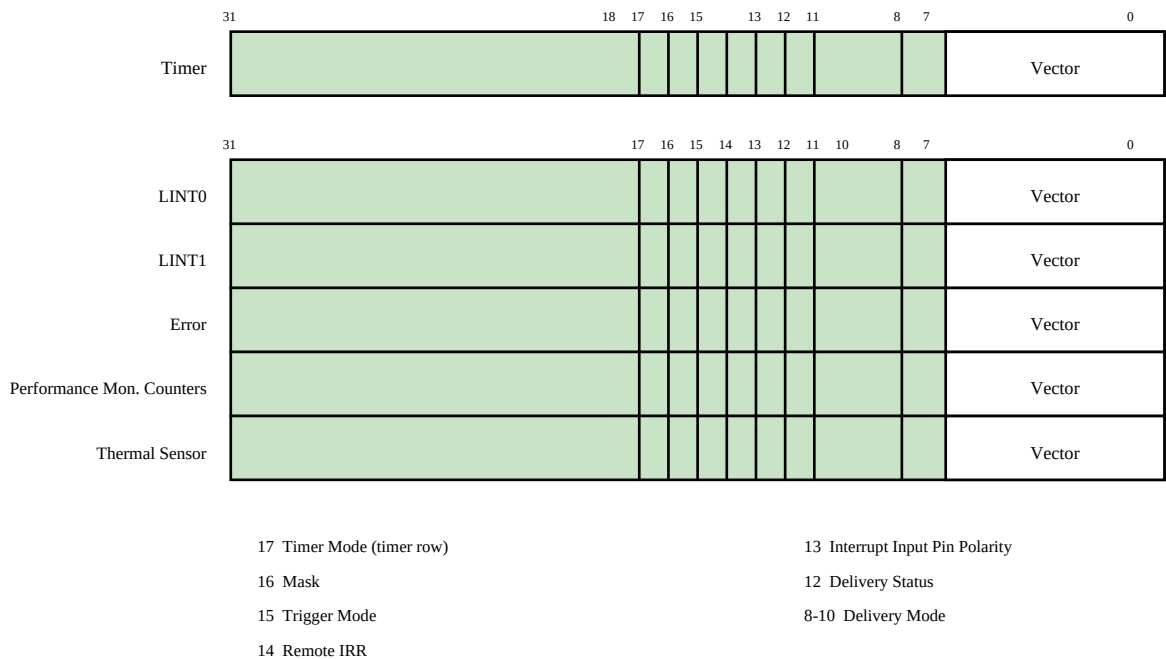


Figure 6.3. Local Vector Table

Vector, bits 0-7, gives the interrupt vector, meaning the IDT handler number invoked when that source fires. Vector values below 16 are invalid for Local APIC delivery, while vectors 16-31 - although reserved by Intel - are not treated as invalid by the Local APIC.

Delivery Mode, bits 8-10, selects how the interrupt is delivered:

- 1. Fixed (000) - invoke the handler in the specified vector.
- 2. SMI (010) - deliver an SMI and enter System Management Mode; the vector should be zero.

3. NMI (100) - deliver a non-maskable interrupt; in protected or 64-bit mode vector 2 is invoked and the vector field is ignored.
4. INIT (101) - send INIT to the processor; the source says the vector should be zero.
5. ExtINT (111) - normally used with LINT0/LINT1 to connect the interrupt to an external 8259A-compatible controller.

Delivery Status, bit 12, is read-only. Zero means that no delivery is pending or that the interrupt has already been accepted for processing. One means delivery has occurred but the processor has not yet accepted it for processing.

Trigger Mode, bit 15, applies to Fixed delivery on LINT0/LINT1. Zero selects edge-sensitive behavior and one selects level-sensitive behavior. Interrupt Input Pin Polarity, bit 13, determines whether the active level is noninverted or inverted.

Remote IRR, bit 14, is read-only and applies to level-sensitive interrupts. It is set when the processor accepts the interrupt and cleared when the handler performs EOI.

Mask, bit 16, masks the source when set. All six LVT sources are masked by default. Timer Mode, bit 17 in the LVT Timer Register, selects one-shot mode when zero and periodic mode when one.

The source emphasizes LINT0 and LINT1 because the operating system can configure them separately on each processor. For example, one input might receive SMI while another receives interrupts forwarded by an 8259A-compatible controller. This makes it possible to distribute categories of interrupt work among processors.

### 6.1.5. Local APIC Timer

Each Local APIC contains a timer interrupt source. Four registers configure it: LVT Timer Register (+320h), Divide Configuration Register (+3E0h), Initial Count Register (+380h), and Current Count Register (+390h).

LVT Timer Register selects the vector, timer mode, and mask state. Divide Configuration Register sets the timer-frequency divisor.



Figure 6.4. Divide Configuration Register (+3E0h)

The source gives the Divide Value encodings as:

- 000 - divide by 2; 001 - divide by 4; 010 - divide by 8; 011 - divide by 16;  
 100 - divide by 32; 101 - divide by 64; 110 - divide by 128; 111 - no division.

Initial Count Register supplies the initial counter value; Current Count Register exposes the current value. Writing Initial Count starts the timer.

In one-shot mode, Current Count is loaded from Initial Count and decremented according to the selected divisor until it reaches zero, at which point the configured interrupt is generated. In periodic mode, reaching zero generates the interrupt and reloads Current Count from Initial Count so the cycle repeats.

### 6.1.6. Interrupt Handling

The source gives the Local APIC interrupt-handling algorithm as follows:

1. NMI, SMI, INIT, ExtINT, and Start-up IPI signals are sent to the processor immediately.
2. For Fixed interrupts, the Local APIC decides whether to deliver the interrupt using the IRR, ISR, TPR, and PPR priority state.
3. Completion is signaled by EOI, written through the EOI register. After EOI, another queued interrupt may be delivered. NMI, SMI, INIT, ExtINT, and Start-up IPI do not require EOI.

Interrupt priority is obtained from the interrupt vector. It is described as dividing the vector by 16 and rounding to the nearest integer. For valid vectors 32-255 it therefore gives priority classes 2-15, with 16 vectors per class.

*Editorial note: APIC priority class is represented by the high four bits of the vector; describing this as division by 16 and 'rounding to the nearest integer' is imprecise. The source wording is preserved.*

TPR (Task Priority Register) is at Local APIC offset 80h. It allows software to block lower-priority interrupts temporarily.



Figure 6.5. TPR register (+80h)

Task Priority, bits 4-7, contains the threshold priority. An interrupt is accepted if its priority is greater than the value in this field. Bits 0-3 provide a priority subclass.

In long mode, CR8 provides an alternate interface to TPR. Writing CR8 performs  $\text{APIC.TPR}[7:4] = \text{CR8}[3:0]$  and clears  $\text{APIC.TPR}[3:0]$ . Reading CR8 returns  $\text{APIC.TPR}[7:4]$ .

PPR (Processor Priority Register) is read-only at offset 0A0h and has the same field layout as TPR.



Figure 6.6. PPR register (+0A0h)

The source computes PPR as:

#### PPR calculation from the source

```
IF TPR[7:4] >= ISRV[7:4]
THEN
    PPR[7:0] <- TPR[7:0]
ELSE
    PPR[7:4] <- ISRV[7:4]
    PPR[3:0] <- 0
```

ISRV is the highest interrupt vector currently being serviced, corresponding to the most significant set bit in ISR.

IRR (Interrupt Request Register) and ISR (In-Service Register) are both 256-bit read-only register sets made from eight 32-bit registers. IRR occupies offsets 200h, 210h, ... 270h; ISR occupies offsets 100h, 110h, ... 170h.

When the Local APIC accepts an interrupt, it sets the corresponding vector bit in IRR. When the processor is ready to service it and IF is set, the Local APIC clears the highest-priority pending bit in IRR and sets the same bit in ISR. ISR therefore records interrupt vectors currently in service.

EOI is performed by writing zero to the EOI register at offset 0B0h, normally address 0FEE000B0h. Writing any value causes EOI but recommends zero for future compatibility, and states that a nonzero value would generate #GP.

After EOI, the Local APIC clears the highest set ISR bit and promotes the highest pending IRR request into ISR. Higher-priority interrupts can preempt lower-priority ones while interrupts are enabled, so more than one ISR bit may be set.

### 6.1.7. Working with the I/O APIC

The Local APIC mediates between the processor, the system bus, and the I/O APIC. The I/O APIC mediates between external devices and processors. The source contrasts its 24 hardware-interrupt input lines with the 16 lines of the 8259A, while noting that newer controllers may support more than 24.

These commonly known input assignments:

1. INTIN1 - PS/2 keyboard.
2. INTIN12 - PS/2 mouse.
3. INTIN16-19 - PCI devices.
4. INTIN23 - SMI interrupt line.

It notes that some motherboard models may remap INTIN16-19.

I/O APIC registers are also memory-mapped, but software accesses them through two main registers. IOREGSEL, normally at 0FEC00000h, selects an internal I/O APIC register. IOWIN, normally at 0FEC00010h, is the data window used to read or write the selected register.

IOAPICID, index 0, contains the controller ID in bits 24-27. IOAPICVER, index 1, is read-only; bits 0-7 hold the version and bits 16-23 hold the maximum redirection-table entry number.

IOREDTBL0-23 form the interrupt redirection table. Each entry is a 64-bit register describing vector, delivery mode, destination, and related routing attributes for one interrupt input.

**Table 6.1. Redirection-table register indexes**

| Register   | Index of high part | Index of low part |
|------------|--------------------|-------------------|
| IOREDTBL0  | 10h                | 11h               |
| IOREDTBL1  | 12                 | 13h               |
| IOREDTBL2  | 14                 | 15h               |
| IOREDTBL3  | 16                 | 17h               |
| IOREDTBL4  | 18                 | 19h               |
| IOREDTBL5  | 1A                 | 1Bh               |
| IOREDTBL6  | 1C                 | 1Dh               |
| IOREDTBL7  | 1E                 | 1Fh               |
| IOREDTBL8  | 20                 | 21h               |
| IOREDTBL9  | 22                 | 23h               |
| IOREDTBL10 | 24                 | 25h               |
| IOREDTBL11 | 26                 | 27h               |
| IOREDTBL12 | 28                 | 29h               |
| IOREDTBL13 | 2A                 | 2Bh               |
| IOREDTBL14 | 2C                 | 2Dh               |
| IOREDTBL15 | 2E                 | 2Fh               |
| IOREDTBL16 | 30                 | 31h               |
| IOREDTBL17 | 32                 | 33h               |
| IOREDTBL18 | 34                 | 35h               |
| IOREDTBL19 | 36                 | 37h               |
| IOREDTBL20 | 38                 | 39h               |
| IOREDTBL21 | 3A                 | 3Bh               |
| IOREDTBL22 | 3C                 | 3Dh               |
| IOREDTBL23 | 3E                 | 3Fh               |

*Editorial note: Table 6.1 labels the first index column 'high part' and the second 'low part' while listing IOREDTBL0 as 10h / 11h. Standard I/O APIC documentation normally assigns 10h to the low dword and 11h to the high dword. The source table is reproduced as printed and the discrepancy is noted rather than silently corrected.*

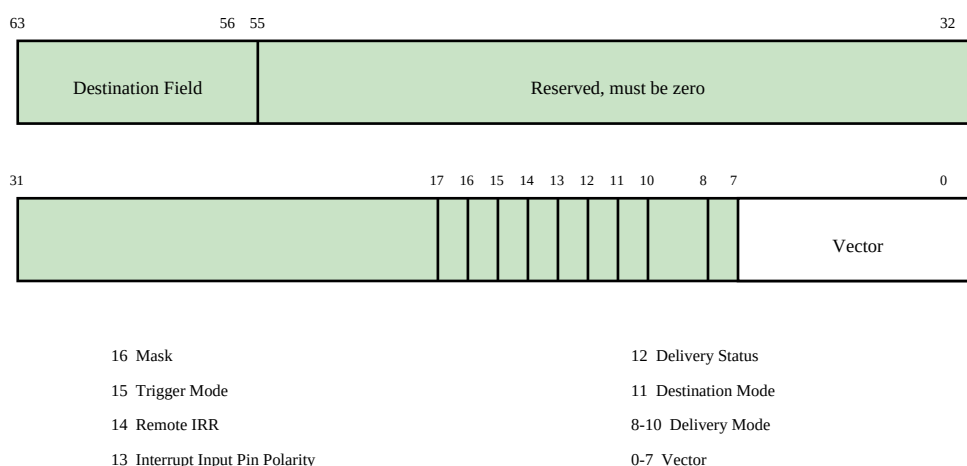


*Figure 6.7. I/O APIC redirection-table entry format*

Destination Field, bits 56-63, depends on Destination Mode. In physical mode it contains the target Local APIC ID; in logical mode it contains a processor-selection mask, discussed in the next section.

Mask, bit 16, masks the interrupt when set. Trigger Mode, bit 15, selects edge sensitivity when zero and level sensitivity when one. Remote IRR, bit 14, is read-only for level-sensitive interrupts and is cleared after EOI. Interrupt Input Pin Polarity, bit 13, selects inversion of the input line.

Delivery Status, bit 12, is read-only and reports that delivery has been initiated but is temporarily delayed because the APIC bus is busy or the destination cannot accept the interrupt.

Destination Mode, bit 11, selects physical or logical addressing. Delivery Mode, bits 8-10, selects Fixed (000), Lowest Priority (001), SMI (010), NMI (100), INIT (101), or ExtINT (111); values 011 and 110 are reserved. Vector, bits 0-7, contains the interrupt vector, with the source giving 10h-0FEh as valid.

### 6.1.8. Practice

The practical example switches to protected mode and initializes APIC so that it receives Local APIC timer interrupts and keyboard interrupts through the I/O APIC. It is based on the protected-mode framework developed in Section 2.1.

For readability, APIC register addresses and indexes are defined as constants in a separate file. The program first reports that it is in protected mode, loads IDTR, enables APIC by setting IA32\_APIC\_BASE bit 11, and then reports that APIC is enabled.

#### Listing 6.1. Initializing the Advanced Programmable Interrupt Controller

```
include 'pmstructures.asm';
include 'APIC_defs.asm'
IDT_size equ IDT_END-IDT
START_CODE:
mov esi, message1
mov al, 0
mov ah, 0
mov bl, "5"
call OutText
lidt fword [IDTR]
; init APIC
mov ecx, IA32_APIC_BASE
rdmsr
bts eax, 11
wrmsr
;APIC enabled
mov esi, message2
mov al, 0
mov ah, 1
mov bl, "5"
call OutText
```

The Local APIC timer is then configured through LVT Timer, Divide Configuration, and Initial Count.

#### Listing 6.2. Initializing the Local APIC timer

```
; enabling LAPIC timer
mov dword [APIC_LVT_Timer_REG_DEF], ((1 shl 17) or 20h)
mov dword [APIC_Timer_Divide_Configuration_REG_DEF], 1001b
mov dword [APIC_Timer_Initial_Count_REG_DEF], 2083333
mov esi, message3
mov al, 0
mov ah, 2
mov bl, "5"
call OutText
```

The timer uses vector 32 (20h) in periodic mode. The timer frequency is not documented and speculates that it depends on processor clock frequency. It suggests adjusting the divisor or Initial Count if the displayed counter advances too quickly. In the example, the divisor is described as 64 and Initial Count is 2083333.

The program next reads IOAPICVER to determine the redirection-table size.

#### Listing 6.3. Configuring the I/O APIC

```
mov dword [IOAPIC_IOREGSEL_REG_DEF], IOAPIC_IOAPICVER
mov eax, dword [IOAPIC_IOWIN_REG_DEF]
shr eax, 16
inc eax ;
mov ebx, 10
mov esi, Max_RTE_Str
call dword_to_STR
mov esi, message4
mov al, 0
mov ah, 3
mov bl, "5"
call OutText
```

*The dangling semicolon in inc eax ; is present in the source and is preserved.*

The high portion of IOAPICVER is used to obtain the number of redirection entries. dword\_to\_STR converts the value to a string for display.

#### Listing 6.4. Configuring the I/O APIC for keyboard interrupts

```
mov eax, 1
cpuid
and ebx, 0FF000000h ; high byte = Local APIC ID
mov dword [IOAPIC_IOREGSEL_REG_DEF], IOAPIC_IOAPICID
mov dword [IOAPIC_IOWIN_REG_DEF], 2000000h
mov dword [IOAPIC_IOREGSEL_REG_DEF], IOAPIC_IOREDTBL1_hi
mov dword [IOAPIC_IOWIN_REG_DEF], ebx
mov dword [IOAPIC_IOREGSEL_REG_DEF], IOAPIC_IOREDTBL1_low
mov dword [IOAPIC_IOWIN_REG_DEF], 21h
mov esi, message5
mov al, 0
mov ah, 4
mov bl, "5"
call OutText
sti
```

```
jmp $
```

The program obtains the current processor's Local APIC ID from CPUID leaf 1, where the source says the high byte of EBX contains that ID. It then sets the I/O APIC ID and programs the redirection-table entry for keyboard interrupts with the destination Local APIC ID and interrupt vector 21h. Other fields are left zero. Interrupts are enabled and the program enters an infinite loop.

#### Listing 6.5. Timer interrupt handler

```
counter dd 0
LVT_APIC_Timer_handler:
push esi
push eax
push ebx
inc dword [counter]
mov eax, [counter]
mov ebx, 10
mov esi, TimerCounterStr
call dword_to_STR
mov esi, TimerCounterStr
mov al, 0
mov ah, 5
mov bl, "5"
call OutText
mov dword [APIC_EOI_REG_DEF], 0 ;EOI
pop ebx
pop eax
pop esi
iretd
```

The timer handler increments a counter, converts it to a string, displays it, and ends by performing EOI through the Local APIC.

#### Listing 6.6. Keyboard interrupt handler

```
kbrd_counter dd 0
Keyboard_handler:
push esi
push eax
push ebx
inc dword [kbrd_counter]
mov eax, [kbrd_counter]
mov ebx, 10
mov esi, KBRDCounterStr
call dword_to_STR
mov esi, KBRDCounterStr
mov al, 0
mov ah, 6
mov bl, "5"
call OutText
in al, 060h
dec al
mov ah, al
and ah, 80h
jz @f
in al, 061h
or al, 80
out 061h, al
xor al, 80
out 061h, al
@@:
mov dword [APIC_EOI_REG_DEF], 0 ;EOI
pop ebx
pop eax
pop esi
iretd
```

The keyboard handler performs the same counter/display work as the timer handler. When the source detects a keyboard-key release through the high bit of port 60h, it acknowledges the event by briefly setting the high bit of port 61h.

The output routine, integer-to-string conversion routine, and interrupt-table declarations are omitted from the book because the author considers them trivial. The complete program was supplied on the accompanying disc in the part6 folder.

### 6.1.9. Summary

This section covered the basic operation of the Advanced Programmable Interrupt Controller, including both the Local APIC and the I/O APIC. With these foundations in place, the next section turns to interprocessor communication.

## 6.2. Interprocessor Communication

Section 6.1 examined the principles of APIC operation. Because APIC can send interprocessor interrupts, it forms the basis of interprocessor communication. This section describes how interprocessor interrupts are sent and how a multiprocessor system is initialized. The practical example initializes every available processor in a multiprocessor computer.

### 6.2.1. General Overview

Before discussing communication itself, the source distinguishes three main kinds of multiprocessor system:

1. A motherboard with several processor sockets.
2. A processor containing several physical cores.
3. A processor exposing several virtual cores through Hyper-Threading.

The first is the most straightforward multiprocessor arrangement: a motherboard contains several processor sockets. This design is frequently used in server systems such as Internet servers and database servers.

The second type is described as the most common in ordinary personal computers of the period. One processor package contains two or more execution cores. The source characterizes a multicore processor as several processors on one die and notes that this avoids the cost of a multisocket motherboard. It gives shared use of the system bus as the principal disadvantage, because the cores compete for access to memory and external devices.

The third type is a processor using Hyper-Threading to expose several, usually two, virtual cores. This is described as one physical processor core that gives software the illusion of several processors. It says this can improve performance for some workloads at little additional system cost.

*Historical note: the source's descriptions of shared buses, core organization, and Hyper-Threading reflect the processor designs it discusses and should not be generalized to every modern multiprocessor system.*

These arrangements may also be combined. The example given is a two-socket motherboard where one installed processor exposes two Hyper-Threading logical processors and the other processor contains four full execution cores, yielding six processors from software's point of view.

The source deliberately uses the word 'processors' rather than 'cores': a physical core or Hyper-Threading logical processor has its own architectural register state and its own Local APIC. System memory and motherboard devices/controllers are shared.

All three arrangements use the same initialization protocol. Therefore, software that initializes processors need not determine whether it is running on a Hyper-Threaded processor, a multicore package, or a multisocket system. The source concludes that from the initialization program's or operating system's point of view, the protocol is the same.

### 6.2.2. Interprocessor Interrupts

Intel provides the Interrupt Command Register (ICR) for generating interprocessor interrupts.

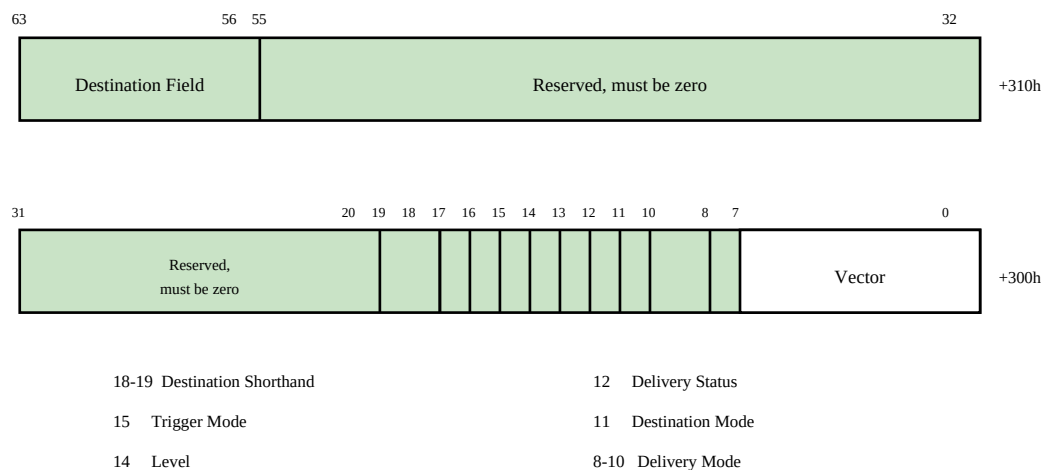


Figure 6.8. Interrupt Command Register (ICR)

ICR is a 64-bit register. Writing its low dword generates an interprocessor interrupt. The high dword is at offset 0310h from the Local APIC base; the low dword is at offset 0300h.

Vector, bits 0-7, contains the interrupt vector generated in the destination processors.

Delivery Mode, bits 8-10, selects the delivery type:

1. Fixed (000) - invoke the interrupt handler at the specified vector.
2. Lowest Priority (001) - send to processors operating at the lowest priority; the source says this is not supported by every processor.
3. SMI (010) - deliver an SMI. The vector should be zero for future compatibility.
4. NMI (100) - deliver a non-maskable interrupt. In protected or 64-bit mode, IDT vector 2 is invoked and the Vector field is ignored.
5. INIT (101) - send INIT to the destination processors; the vector should be zero.
6. Start-Up (110) - the Start-up IPI, or SIPI, sent to processors that have not yet started. The vector normally identifies the initialization code that brings the processor into the operating mode required by the operating system.

The source marks Delivery Mode values 011 and 111 as reserved.

*Editorial note: the delivery-mode list is translated exactly as presented. The supported ICR delivery modes depend on APIC mode and processor documentation; the source's reserved-value statement is not silently replaced.*

Destination Mode, bit 11, chooses physical or logical addressing. In physical mode, Destination Field contains the target Local APIC ID. In logical mode, the destination is selected through the Local APIC logical-addressing mechanism.

Delivery Status, bit 12, is read-only: zero means the interprocessor interrupt was successfully delivered; one means delivery is still in progress.

Level, bit 14, is described by the source as unused on Pentium 4 and Xeon processors. It says the bit must be zero for INIT and one for the other signal types. Trigger Mode, bit 15, is described as being used only for INIT, with zero for edge-sensitive and one for level-sensitive behavior.

Destination Shorthand, bits 18-19, provides four destination shortcuts:

- 00 - No Shorthand: use the destination or destinations specified in Destination Field.
- 01 - Self: send the interrupt to the current processor.
- 10 - All including self: send to every processor in the system, including the sender.
- 11 - All excluding self: send to every processor except the sender.

Destination Field, bits 56-63, is used only when Destination Shorthand is zero. With physical addressing it contains a Local APIC ID. With logical addressing its interpretation depends on the LDR and DFR registers.

Every Local APIC has LDR and DFR. LDR is at offset 0D0h, normally 0FEE000D0h; bits 24-31 contain the Local APIC's logical identifier. DFR is at offset 0E0h, normally 0FEE000E0h; only bits 28-31 are significant in the source's description.

DFR bits 28-31 may be 1111 or 0000. The first selects the flat logical addressing model and the second the cluster model. The book does not develop the cluster model further.

*Editorial note: the book states that the cluster model requires additional motherboard controllers. This is retained as source content rather than generalized beyond the APIC implementation being described.*

Under the flat logical model, a Local APIC receiving an interprocessor interrupt ANDs its logical identifier with the received Destination Field. If the result is nonzero, the Local APIC accepts the interrupt.

### 6.2.3. Synchronizing Access to Data

With several processors, access to shared data must be synchronized. The source uses a logging routine as a simple example. A shared variable points to the end of the log. One processor begins writing new data and later updates that pointer; meanwhile a second processor may read the old pointer and begin writing to the same location, overwriting the first processor's data.

The same problem occurs in multitasking systems with a single processor. This leads to the idea of a critical section: a region of code that may be executed by only one processor, or more generally only one thread, at a time.

The fundamental mechanism introduced here is a semaphore. The book compares it to railway signaling on a single-track section: a green signal means the track is free; a train entering the section changes the signal to red and restores green after leaving.

In software, a bit in memory can play the role of the semaphore. A variable alone is not sufficient, however; the processor needs a way to make the read-modify-write operation atomic with respect to other processors. The source introduces the LOCK prefix for this purpose.

The source lists the following instruction groups as usable with LOCK:

1. Bit operations: BTR, BTS, and BTC.
2. Exchange operations: XADD, CMPXCHG, and CMPXCHG8B.
3. Unary arithmetic/logical operations: INC, DEC, NOT, and NEG.
4. Binary arithmetic/logical operations: ADD, ADC, SUB, SBB, AND, OR, and XOR.
5. XCHG, for which the source says LOCK applies implicitly even when the prefix is not written.

*Editorial note: the implicit locking property of XCHG applies when XCHG has a memory operand. The source states the rule more generally; the translation preserves its framing.*

Using LOCK with other instructions, or with a listed instruction when no memory write occurs, is said to generate #GP or #UD. When LOCK is used, the memory location involved in the operation is protected from conflicting access by other processors.

The following entry sequence for a critical section, using bit 0 of SEMAPHORE:

#### Critical-section entry sequence

```
@@:
bt dword [SEMAPHORE], 0
jnc @b
lock btr dword [SEMAPHORE], 0
jnc @b
```

BT tests bit 0 and copies it to CF. The loop waits until that bit becomes one. BTR then clears the bit and reports its previous value through CF. If another processor cleared it first, the code returns to the waiting loop; otherwise the critical section has been acquired.

The exit operation printed by the source is:

#### Critical-section exit sequence

```
lock bts dword [SEMAPHORE], 0
```

This sets bit 0 back to one, making the semaphore available to another processor. The book calls this technique a spin lock and treats it as a fundamental synchronization mechanism.

*Editorial note: the Russian prose immediately before the BTS instruction says the bit should be 'cleared,' but BTS sets the bit. The code and the earlier semaphore convention make the intended release operation clear, so the translation preserves the instruction and explicitly notes the prose mismatch.*

When programming under modern operating systems such as Windows, Linux, or BSD, the source says application programmers generally do not need to implement these primitives directly because the operating system provides synchronization functions.

### 6.2.4. Initializing a Multiprocessor System

A multiprocessor system contains two processor roles: the BSP (bootstrap processor) and APs (application processors). At power-on, hardware dynamically selects one processor to start first; it becomes the BSP. BIOS executes on the BSP and in turn starts the operating system. The remaining processors wait for a Start-up IPI (SIPI).

On INIT, the source says the processor examines its BSP flag, bit 8 of IA32\_APIC\_BASE. If the flag is set, the processor continues with BIOS code; if it is clear, the processor waits for SIPI.

Fully waking an application processor requires the sequence INIT-SIPI-SIPI. After this sequence the processor is still initially in real-address mode, so each processor must then be switched into the protected or 64-bit mode required by the operating system.

The source gives this initialization algorithm:

1. The BSP enters the operating-system mode and globally enables APIC by setting IA32\_APIC\_BASE bit 11.
2. The BSP sends INIT to all processors except itself. After INIT, the APs wait for SIPI. To wait 10 ms after sending INIT.
3. The BSP sends Start-up IPI to all processors except itself. The vector contains the page number of the AP initialization code. The source prose says to wait 200 ms after the first SIPI.
4. The BSP sends the same Start-up IPI a second time to all processors except itself.

*Editorial note: the prose says to wait 200 milliseconds after the first SIPI, while Listing 6.11 calls delay with EAX = 200 and later defines the delay argument as microseconds. Both are preserved; the source therefore contradicts itself on the unit.*

The SIPI vector is shifted left by 12 bits, equivalent to multiplication by 1000h, and execution begins at that resulting address. The AP initialization code must therefore begin at an address divisible by 1000h (4096), meaning it must be page-aligned.

### 6.2.5. Practice

The practical program initializes a multiprocessor system from scratch rather than using the Chapter 2 framework. It remains a DOS COM program. Its purpose is to display the Local APIC ID of every processor and finally display the total number of processors.

All processors use the same protected-mode transition code. Because the AP initialization code must begin on a 4 KB boundary, the program first copies its initialization block to a suitably aligned address.

#### Listing 6.7. Moving initialization code to a 4 KB-aligned address

```
mov ax, 3
int 10h
mov si, CODE_START
xor ax, ax
mov es, ax
mov di, CODE_BASE_ADDRESS
mov cx, PROGRAM_END - PROGRAM_START
rep movsb
mov ds, ax
mov ss, ax
mov es, ax
jmp 0000:CODE_BASE_ADDRESS
```

The protected-mode transition is common to every processor. Unlike the earlier Chapter 2 program, the GDT linear address and protected-mode jump address do not need to be calculated dynamically because the program has already been copied to its intended location.

#### Listing 6.8. Switching the processor to protected mode

```
CODE_START:
ORG CODE_BASE_ADDRESS
PROGRAM_START:
in al,92h
or al,2
out 92h,al
lgdt fword [GDTR]
cli
in al,70h
or al,80h
out 70h,al
mov eax,cr0
or al,1
mov cr0,eax
jmp CODE_SELECTOR:PROTECTED_MODE_ENTRY_POINT
```

After entering protected mode, the segment registers are reloaded, the processor receives its own stack, and the shared processor count is updated. Because several processors can execute this code at the same time, the counter and stack assignment are protected by the semaphore.

#### Listing 6.9. Semaphore initialization and processor counting

```
use32
PROTECTED_MODE_ENTRY_POINT:
mov ax, DATA_SELECTOR
mov ds, ax
mov es, ax
mov ss, ax
@@:
bt dword [SEMAPHORE], 0
jnc @b
lock btr dword [SEMAPHORE], 0
jnc @b
;-----
inc dword [CPU_Count]
mov esp, [CPU_Count]
mov ebp, esp ; save current CPU Count in ebp
shl esp, 12
add esp, STACK_BASE_ADDRESS
;-----
lock bts dword [SEMAPHORE], 0
mov esi, message1
mov al, 0
mov ah, 0
mov bl, "5"
call OutText
```

If the code only incremented `CPU_Count`, a separate critical section would not be required because `INC` could itself be prefixed with `LOCK`. Here, however, the newly obtained processor count is also used to choose a nonoverlapping stack, so no other processor may modify `CPU_Count` between incrementing it and reading the value for the stack calculation.

`EBP` retains the processor number assigned at that moment. `ESP` is formed by shifting the count left 12 bits and adding `STACK_BASE_ADDRESS`, giving each processor its own 4 KB-spaced stack.

The processor then enables APIC, reads its Local APIC ID, converts it to text, and displays it.

#### Listing 6.10. Displaying the APIC ID

```
mov ecx, IA32_APIC_BASE
rdmsr
bts eax, 11
wrmsr
mov edx, eax
mov eax, [LOCAL_APIC_ID_REG_DEF]
shr eax, 24
mov esi, LAPICID
mov ebx, 16
call dword_to_STR
mov esi, message2
mov al, 0
mov ebx, ebp
mov ah, bl
mov bl, "5"
call OutText
```

The display row is derived from `EBP`, which contains the processor count captured while `CPU_Count` was protected. The source treats that value as the processor's sequence number so that the Local APIC ID strings do not overwrite one another on screen.

After displaying its ID, each processor checks whether it is the BSP. Only the BSP wakes the other processors with the INIT-SIPI-SIPI sequence.

### Listing 6.11. Initializing AP processors

```
mov ecx, IA32_APIC_BASE
rdmsr
bt eax, 8
jnc .END

;boorstrap code
mov eax, INIT_IPI_command
mov [APIC_ICR_low_DEF], eax
@@:
bt dword [APIC_ICR_low_DEF],12
jc @b
mov eax, 10000
call delay

mov eax, STARTUP_IPI_command
mov [APIC_ICR_low_DEF], eax
@@:
bt dword [APIC_ICR_low_DEF],12
jc @b
mov eax, 200
call delay

mov eax, STARTUP_IPI_command
mov [APIC_ICR_low_DEF], eax
@@:
bt dword [APIC_ICR_low_DEF],12
jc @b

mov eax, 50000
call delay

mov eax, [CPU_Count]
mov esi, CPU_Count_Str
mov ebx, 10
call dword_to_STR

mov esi, message3
mov ebx, eax
inc ebx
mov al, 0
mov ah, bl
mov bl, "5"
call OutText
```

*The source spelling ;boorstrap code is preserved exactly. The printed excerpt branches to .END but does not show the corresponding label in Listing 6.11; the translation does not invent omitted code.*

The delay function waits for the number of microseconds specified in EAX. After INIT-SIPI-SIPI, the other processors begin execution at CODE\_BASE\_ADDRESS. They perform the same initialization as the BSP except that, because their BSP flag is clear, they do not send another INIT-SIPI-SIPI sequence.

The complete program was supplied with the book. For multiprocessor-system emulation, the source recommends QEMU and gives the command-line form -smp N, where N is the processor count, for example: qemu.exe -fda A: -smp 8.

*Historical note: the DOS COM/floppy/QEMU invocation is reproduced from the source's development environment.*

### 6.2.6. Summary

This section examined interprocessor communication and the algorithm for initializing a multiprocessor system. The practical program applied those concepts by initializing the processors in a multiprocessor machine.

## Appendix A. MSR Registers

Intel and AMD processors have a number of additional flags in some registers. This appendix gives only those bits and flags that are common to processors from both manufacturers, or that serve the same purpose on both.

The IA32\_EFER register, also called EFER, has index 0C0000080h. It is one of the most important MSRs because it controls switching the processor into 64-bit mode and enables several of its principal features.



The LME bit controls the transition into long mode. If this bit is assigned the value one, the source says the processor switches to long mode.

The NXE bit enables use of the NX bit in page-table entries that describe virtual pages. If NXE is clear, using NX causes a general-protection exception (#GP).

IA32\_SYSENTER\_EIP is located at index 176h. It contains the address of the SYSENTER handler.

IA32 STAR is located at index 0C0000081h. The register contains base values for selectors used by SYSCALL/SYSRET.



141

stack during SYSRET.

*Editorial note: Figure A.2 labels bits 63-48 as SYSRET CS/SS and bits 47-32 as SYSCALL CS/SS, while the prose says 32-47 for both.*

The source next names IA32\_STAR again at index 0C0000082h and says that this register contains the 64-bit address of the SYSCALL handler.

*Editorial note: the register at 0C0000082h is normally IA32\_LSTAR. The source prints IA32\_STAR here, so that name is retained and noted.*

IA32\_FMASK is located at index 0C0000084h. Its low 32 bits contain a mask for the flags register; the high 32 bits are reserved. When SYSCALL executes, the flags value is calculated by ANDing the current flags value with the value in this register.

*Editorial note: the FMASK operation is described here exactly as in the book. Architectural manuals describe the precise flag-masking operation differently.*

## A.4. APIC Registers

IA32\_APIC\_BASE is located at index 1Bh. This register allows software to specify the base address of the Local APIC registers in memory.

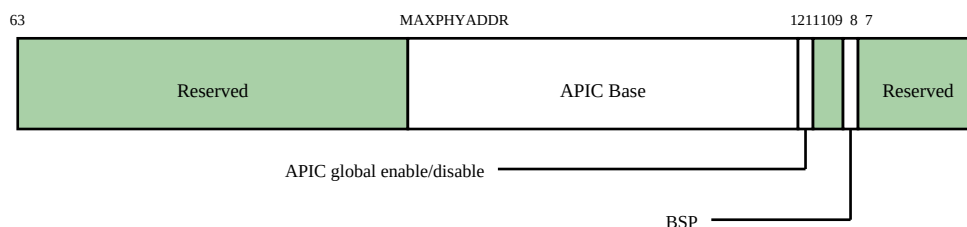


Figure A.3. IA32\_APIC\_BASE register format

The APIC Base field contains the base address of the APIC registers. The base is always page-aligned, so the low 12 address bits are not significant. After processor power-on or reset, the base address is 0FEE0000h. The maximum address width depends on the processor model but cannot exceed 52 bits.

The APIC global enable/disable bit controls global enabling and disabling of the APIC controllers, Local APIC and I/O APIC. A one enables the controller; a zero disables it.

The BSP bit indicates whether the current processor is the bootstrap processor. If this bit is set, the current processor was the first processor that began executing after power-on.

## A.5. Registers for Managing Segments in Long Mode

IA32\_FS\_BASE is located at index 0C0000100h. It allows software to set the base address of the segment described by FS.

IA32\_GS\_BASE is located at index 0C0000101h. It allows software to set the base address of the segment described by GS.

IA32\_KERNEL\_GSBASE is located at index 0C0000102h. It is used by SWAPGS to exchange its value with IA32\_GS\_BASE.

## A.6. Auxiliary Registers

IA32\_TSC\_AUX is located at index 0C0000103h. Its low 32 bits are used by RDTSCP; the high 32 bits are reserved. RDTSCP places the low 32-bit value of this register in ECX. The value may be arbitrary. It is normally used to pass information about the processor, or other auxiliary information, to programs running at nonzero privilege levels.

## Appendix B. System Registers

This appendix gives a complete description of the system registers and the flags register.

Intel and AMD processors contain a number of additional flags in some registers. Only bits and flags common to both manufacturers, or having the same purpose on both, are described below.

### B.1. CR0 Register

CR0 controls important processor operating parameters, including entry into protected mode and enabling or disabling page translation.

In protected mode the register is 32 bits wide; in 64-bit mode it is extended to 64 bits. Fields shown as reserved in Figure B.1 must always contain zero. Writing a nonzero value to a reserved field causes a general-protection exception (#GP).

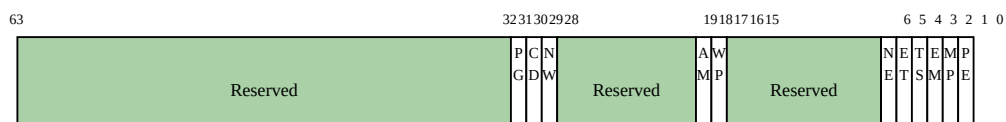


Figure B.1. CR0 register format

**PE (Protection Enable)** enables protected mode. If the bit is one, the processor operates in protected mode; if zero, it operates in real-address mode.

**MP (Monitor Coprocessor)** works with TS and controls WAIT/FWAIT. If MP and TS are both set, WAIT generates #NM. If MP is clear, WAIT does not generate that exception.

**EM (Emulation)** indicates whether an x87 mathematical coprocessor is present or connected. If EM is set, x87 instructions generate #NM; the source also states that MMX and SSE instructions then generate #UD, allowing x87 operation to be emulated in software.

**TS (Task Switched)** is used to support saving the context of x87, MMX, and SSE state. If TS is set, executing an x87, MMX, or SSE instruction generates #NM. If MP is also set, WAIT/FWAIT generates #NM as well. The source says TS is set by the processor on every task switch.

**ET (Extension Type)** is now reserved and always has the value one. It was used on i386 and i486 processors to detect Intel 387 DX support.

**NE (Numeric Error)** selects the x87 numeric-error mechanism. When set, internal x87 exception handling is used. When clear, x87 error signals are controlled through #IGNNE and #FERR, as described by the source.

**WP (Write Protect)** enables write protection for code executing at privilege level 0. When set, ring-0 code cannot write to pages marked read-only. When clear, ring-0 code may write to such pages.

**AM (Alignment Mask)** participates in alignment checking. Alignment checking is enabled when CR0.AM = 1, the AC flag is set, and CPL = 3. Under those conditions, an unaligned operand address generates #AC.

**NW (Not Writethrough)** is said by the source to be ignored on most modern processors.

**CD (Cache Disable)** globally controls processor caching. If clear, caching operates normally. If set, no new data is brought into the cache. The source says that existing cache hits may still be used for reads; on a write hit, memory is written and the cached data is invalidated. It also says PWT and PCD in paging structures are ignored while caching is disabled.

**PG (Paging)** enables or disables address translation. Paging cannot be enabled while PE is clear.

### B.2. CR2 and CR3 Registers

CR2 and CR3 are used after page translation is enabled. CR2 is relevant after a page-fault handler is invoked and contains the virtual address that caused the page fault. It is 32 bits wide in protected mode and expands to 64 bits in 64-bit mode.

CR3 contains the address of the root paging structure used for virtual-to-physical translation: a page directory, page-directory-pointer table, or PML4 depending on the paging mode.

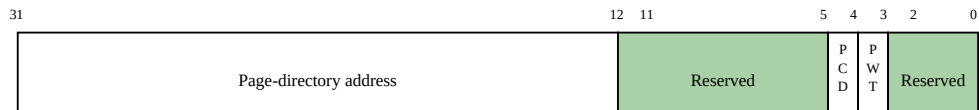


Figure B.2. CR3 in protected mode, ordinary address translation

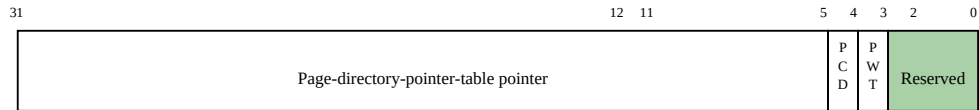


Figure B.3. CR3 in protected mode, PAE translation

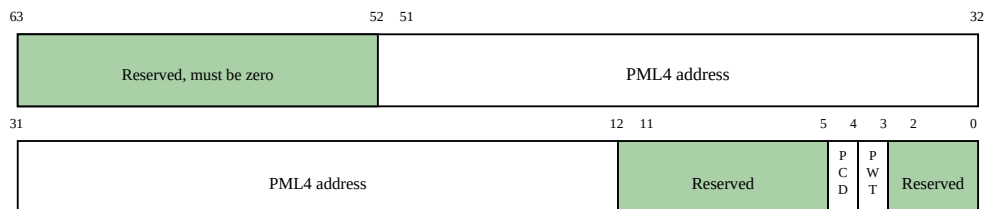


Figure B.4. CR3 in long mode

PCD and PWT control caching for the page containing the root paging structure. PCD disables caching for that page; PWT controls write-through behavior.

With PAE enabled in protected mode, the page-directory-pointer table need not be aligned to 4 KB; 32-byte alignment is sufficient. The source nevertheless says PCD and PWT control caching for the entire page that contains that table.

### B.3. CR4 Register

CR4 controls additional processor facilities not controlled by CR0. It is 32 bits wide in protected mode and extends to 64 bits in 64-bit mode. Reserved fields must contain zero or #GP is generated.

Intel processors define additional vendor-specific CR4 bits, but those are not covered here.

**VME (Virtual-8086 Mode Extensions)** controls hardware support for interrupt and exception handling for programs running in virtual-8086 mode.

**PVI (Protected-Mode Virtual Interrupts)** enables or disables support for the VIF and VIP virtual-interrupt flags.

**TSD (Time-Stamp Disable)** restricts RDTSC and RDTSCP at nonzero privilege levels. If set, those instructions may execute only at privilege level 0; if clear, they may execute at any privilege level.

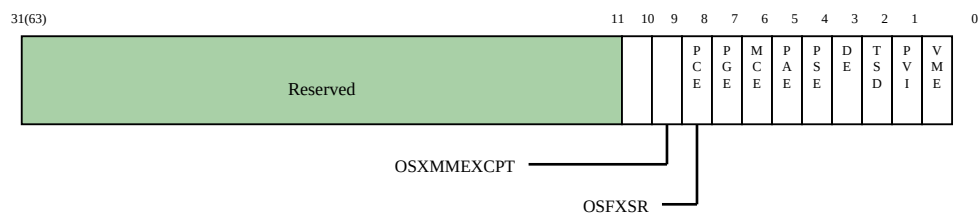


Figure B.5. CR4 register format

Editorial note: the original figure caption is printed as 'Figure V.5' using the Cyrillic letter Ve rather than the appendix letter Be. The corrected appendix numbering B.5 is used here.

**DE (Debugging Extensions)** blocks access to DR4 and DR5; when set, access to those registers generates #UD.

**PSE (Page Size Extensions)** enables 4 MB pages in the ordinary protected-mode paging mechanism.

**PAE (Physical Address Extension)** enables extended physical addressing. The source says this flag may be enabled only together with CR0.PG and notes that PAE is required for long mode.

*Editorial note: earlier chapters correctly show PAE being enabled before paging during entry to long mode. The sentence above is retained as printed in this appendix.*

**MCE (Machine-Check Enable)** enables the machine-check exception (#MC). The source states that if the bit is clear, the machine-check exception will not be generated.

**PGE (Page-Global Enable)** enables use of the G bit in page tables and directories. If PGE is zero, the source says G is ignored.

**PCE (Performance-Monitoring Counter Enable)** allows RDPMC at nonzero privilege levels. If clear, only privilege-level-0 code may execute RDPMC.

**OSFXSR** enables operating-system support for FXSAVE/FXRSTOR and SSE. The source states that when set, FXSAVE/FXRSTOR can save and restore x87, MMX, and SSE state and SSE instructions are enabled; when clear, it says only x87 and MMX state are handled and SSE instructions generate #UD.

**OSXMMEXCPT** enables unmasked SIMD floating-point exceptions. When set, SSE floating-point errors generate #XF; when clear, the source states that such errors generate #UD.

### B.4. GDTR and IDTR Registers

GDTR and IDTR identify the global descriptor table (GDT) and interrupt descriptor table (IDT).

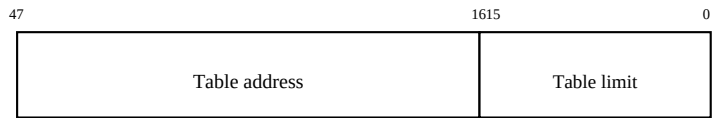


Figure B.6. GDTR and IDTR format

In protected mode the registers are 48 bits wide. In long mode they expand to 80 bits so they can hold 64-bit table addresses.

The table-limit field contains the maximum permitted offset inside the table, equal to the table size in bytes minus one. The table address is an absolute virtual address and does not depend on any segment register.

*Editorial note: the source's Figure B.6 depicts the 48-bit protected-mode form (32-bit table address plus 16-bit limit), while the accompanying text separately states that the registers widen to 80 bits in long mode.*

### B.5. LDTR and TR Registers

LDTR and TR are 16-bit registers that hold selectors for the local descriptor table (LDT) and the currently executing task. Both also have hidden parts containing the corresponding descriptors so repeated GDT accesses can be avoided.

In long mode, TR is used only to identify the stacks required for privilege-level transitions and to identify IST frames.

### B.6. Flags Register

The flags register is central to processor operation because it controls fundamental aspects of execution. It is 32 bits wide in protected mode and extends to 64 bits in long mode.

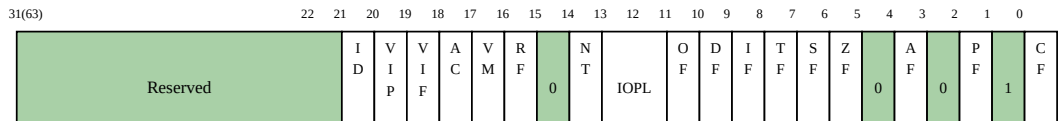


Figure B.7. Flags register

**CF (Carry Flag)** is set when a carry or borrow occurs from the most significant bit of a result. It is useful for arithmetic on values larger than one machine word.

**PF (Parity Flag)** is set when the low byte of the result contains an even number of one bits.

**AF (Adjust Flag)** records a carry or borrow between half-bytes and is used by decimal arithmetic instructions.

**ZF (Zero Flag)** is set when the previous result was zero and cleared otherwise.

**SF (Sign Flag)** equals the most significant bit of the previous result and therefore indicates a negative signed result when set.

**TF (Trap Flag)** causes a debug exception (#DB) after each instruction when set. If TF is set by POPF or IRET, the source says the debug exception occurs after the following instruction.

**IF (Interrupt Flag)** is described by the source as the 'interrupt disable flag' and says that when it is set, all maskable hardware interrupts are disabled. It does not affect exceptions or NMI.

*Editorial note: architectural x86 behavior is the opposite of that sentence - IF = 1 enables recognition of maskable external interrupts, while IF = 0 masks them. The source statement is preserved and flagged rather than silently corrected.*

**DF (Direction Flag)** controls MOVS, CMPS, SCAS, LODS, OUTS, and STOS. When set, string operations proceed from higher addresses toward lower addresses; when clear, they proceed in the normal increasing-address direction.

**OF (Overflow Flag)** is set when signed arithmetic overflows the destination size.

**IOPL (I/O Privilege Level)** controls port-I/O privilege. For a task to access a restricted I/O port, its privilege level must be numerically less than or equal to IOPL. The source says IOPL may be modified by POPF and IRET only at privilege level 0.

**NT (Nested Task)** is set when a task switch occurs through CALL or through an interrupt or exception whose handler is a task. If NT is set when IRET executes, the processor returns to the interrupted task.

**RF (Resume Flag)** controls restarting the instruction that caused an exception and suppresses #DB at the same time. For trap-type exceptions, the source says the handler receives RF = 0 because the instruction is not restarted. RF is cleared after each instruction except after IRET that sets it, and after JMP/CALL/INT n when the target is a task gate.

The source also says that during interrupt generation the saved flags contain RF = 1 when the interrupted instruction is a string instruction (MOVS, LODS, CMPS, SCAS, STOS, or OUDS).

*Editorial note: OUDS is printed in the original list and is preserved. It appears to be a typo for OUTS.*

**VM (Virtual-8086 Mode)** selects virtual-8086 mode when set. Clearing it returns to protected mode. The source says VM may be changed only by IRET.

**AC (Alignment Check)**, together with CR0.AM, causes #AC for unaligned operand addresses at privilege level 3.

**VIF (Virtual Interrupt Flag)** substitutes for IF when VME or PVI is enabled. The source describes how changes that would affect IF are reflected in VIF in virtual-8086 mode or at privilege level 3 when IOPL is less than 3.

**VIP (Virtual Interrupt Pending)** is set to indicate that a virtual interrupt is pending. The source says the processor never modifies this flag and checks it only when VME or PVI state is involved.

**ID (Identification Flag)**: if software can modify this bit, the processor implementation supports CPUID.

## B.7. CR8 Register

CR8 is one of the additions introduced with long mode and provides another way to access the Local APIC TPR.

It sets the priority of the currently executing task. CR8 is 64 bits wide, but only the low four bits are significant; all other bits are reserved and must be zero. The low four bits specify the current task-priority level.

The processor accepts only interrupts whose numeric priority is greater than the current task-priority level. A value of zero allows all interrupts; a value of 15 blocks all interrupts.

## Appendix C. System Instructions

This appendix describes all system instructions. System instructions include all privileged instructions, as well as instructions that operate on system registers and system data structures.

Intel and AMD processors also provide additional instruction sets, such as VT-x and AMD-v; they are not described here. The source also notes that some instructions may behave differently on processors from the two manufacturers. This appendix considers only instructions supported by processors from both manufacturers and, having the same properties.

### C.1. Working with System Registers

1. **LGDT** - loads GDTR, the pointer to the Global Descriptor Table. Syntax: `lgdt <source>`. The source may be a 48-bit variable in protected or real-address mode, or an 80-bit variable in 64-bit mode, containing an image of GDTR. The instruction may be executed only at privilege level 0 and in real-address mode.
2. **SGDT** - obtains the contents of GDTR. The source says the instruction is identical to LGDT except that it may be executed at any privilege level.
3. **LIDT** - loads IDTR, the pointer to the Interrupt Descriptor Table. Syntax: `lidt <source>`. The source may be a 48-bit variable in protected or real-address mode, or an 80-bit variable in 64-bit mode, containing an image of GDTR. The instruction may be executed only at privilege level 0 and in real-address mode.

*Editorial note: the LIDT description says the memory operand contains an image of GDTR. In context, LIDT operates on IDTR; the source wording is preserved and the discrepancy is noted.*

4. **SIDT** - obtains the contents of IDTR. The source says it is identical to LIDT except that it may be executed at any privilege level.
5. **LLDT** - loads LDTR. Syntax: `lldt <source>`. The source may be a 16-bit register or a 16-bit value in memory containing the selector of the LDT segment in the GDT. The instruction may be executed only at privilege level 0.
6. **SLDT** - obtains the contents of LDTR. The source says it is identical to LLDT except that it may be executed at any privilege level.
7. **LMSW** - loads the machine status word. Syntax: `lmsw <source>`. The source may be a 16-bit register or a value in memory. The instruction loads the low four bits of the source into the low four bits of CR0. It may be executed only at privilege level 0. The source says it is retained only for compatibility with old software.
8. **SMSW** - obtains the machine status word, the low 16 bits of CR0. Syntax: `smsw <destination>`. The destination may be a register of any size; the result is zero-extended to the required width. A 16-bit memory location may also be used.
9. **CLTS** - clears the TS flag in CR0. The instruction takes no operands and may be executed only at privilege level 0.
10. **LTR** - loads the task register. Syntax: `ltr <source>`. The source may be a 16-bit register or a value in memory containing a TSS selector in the GDT. The instruction may be executed only at privilege level 0.
11. **STR** - obtains the contents of the task register. The source says it is identical to LTR except that it may be executed at any privilege level.
12. **MOV (CRn)** - operates on control registers CRn. Syntax as printed by the source: `mov <source>, <destination>`. If the source is a control register, the destination may only be a general-purpose register. If the destination is a control register, the source may only be a general-purpose register. The instruction may be executed only at privilege level 0 and in real-address mode. In protected and real-address modes the registers are 32 bits wide; in 64-bit mode they are 64 bits wide.

*Editorial note: the printed operand order is source, destination. Intel/FASM assembly syntax normally writes MOV operands as destination, source. The source form is retained rather than silently rewritten.*

13. **MOV (DRn)** - operates on debug registers. Syntax as printed by the source: `mov <source>, <destination>`. If the source is a debug register, the destination may only be a general-purpose register. If the destination is a debug register, the source may only be a general-purpose register. The instruction may be executed only at privilege level 0 and in real-address mode. In protected and real-address modes the registers are 32 bits wide; in 64-bit mode they are 64 bits wide.

*The source uses the same source,destination operand-order wording here as in its MOV (CRn) entry.*

14. **RDMSR** - reads an MSR. The instruction takes no explicit operands. The register index is supplied in ECX, and the MSR contents are returned in EDX:EAX, with the high part in EDX and the low part in EAX. The instruction may be executed only at privilege level 0 and in real-address mode.

15. **WRMSR** - writes an MSR. The instruction takes no explicit operands. The source states that the operation is identical to RDMSR.

*Editorial note: WRMSR is described as 'identical' to RDMSR even though one writes and the other reads. The statement is retained as printed.*

16. **RDPMS** - reads performance-monitoring counter registers. The instruction takes no explicit operands. ECX supplies the register index, printed by the source as 0-3, and the counter value is returned in EDX:EAX. If CR4.PCE is set, the instruction may be executed at any privilege level; if clear, only at privilege level 0. The source describes RDPMS as equivalent to accessing performance-counter MSRs directly, while providing one unified instruction even though the corresponding MSR indexes vary by processor type.
17. **RDTS** - reads the time-stamp counter. The instruction takes no explicit operands. The counter value is returned in EDX:EAX. If CR4.TSD is clear, the instruction may be executed at any privilege level; if set, only at privilege level 0.
18. **RDTS** - reads the time-stamp counter and an auxiliary value. It operates like RDTS except that ECX receives the low 32 bits of IA32\_TSC\_AUX. The source says IA32\_TSC\_AUX may contain an arbitrary value and is normally used to provide processor

information to programs running at nonzero privilege levels.

19. **SWAPGS** - exchanges the hidden GS base with `IA32_KERNEL_GS_BASE`. The instruction takes no explicit operands. The new base for the segment described by GS is loaded into the hidden part of GS, and the old GS base is written to `IA32_KERNEL_GS_BASE`. The instruction may be executed only at privilege level 0 and only in 64-bit mode.

## C.2. System Instructions

1. **HLT** - places the processor in the HALT state. The instruction takes no operands. In HALT, the processor executes no instructions and consumes minimal energy. The source says the processor leaves HALT because of enabled hardware interrupts, NMI, SMI, RESET, or INIT. After leaving HALT, execution continues with the instruction following HLT. HLT may be executed only at privilege level 0.
2. **IRET (IRETD, IRETQ)** - returns from an interrupt. The instruction takes no operands. If the privilege level changes, the stack also changes: old SS and ESP are restored from the interrupt-handler stack; in long mode, the source says SS and RSP are always restored from the handler stack. If NT is set in the flags register, protected mode performs a task switch, whereas long mode generates #GP.
3. **MONITOR** - initializes monitoring and enters the monitoring state. The instruction takes no explicit operands. EAX, or RAX, must contain the address of a memory location to monitor; when that location changes, the processor leaves the monitoring state. EDX and ECX contain additional parameters, but the source says they are not yet defined on contemporary processors: EDX is completely ignored and ECX must be zero. The instruction may be executed only at privilege level 0.
4. **MWAIT** - enters a wait state. The instruction takes no explicit operands. EAX and ECX specify additional waiting parameters. The source says EAX is completely ignored. Bit 0 of ECX allows exit from the wait state on hardware interrupts even when IF is clear; all other ECX bits must be zero. The instruction may be executed only at privilege level 0 and in real-address mode.
5. **RSM** - leaves System Management Mode (SMM). The instruction takes no operands and may execute only in SMM; in any other mode it generates #UD.
6. **SYSENTER** - calls a system function. The instruction takes no operands. The source says it works only in protected mode and transfers execution to privilege level 0. CS is loaded from `IA32_SYSENTER_CS`, while SS is formed by adding 8 to CS. The hidden CS and SS state receives base addresses of zero and limits of 4 GB. ESP is loaded from `IA32_SYSENTER_ESP` and EIP from `IA32_SYSENTER_EIP`. The source says VM, IF, and RF are cleared in the flags register.

*Editorial note: Chapter 4 states that Intel processors support SYSENTER/SYSEXIT in long mode, while this appendix says SYSENTER works only in protected mode. Both source statements are preserved.*

7. **SYSEXIT** - returns from a system function. The instruction takes no operands and, according to the source, works only in protected mode. CS is calculated by adding 16 to `IA32_SYSENTER_CS`, while SS is calculated by adding 24 to `IA32_SYSENTER_CS`. The hidden CS and SS bases are zero and the limits are 4 GB. The instruction always transfers to privilege level 3. ESP is loaded from ECX and EIP from EDX. The instruction may be executed only at privilege level 0.
8. **SYSCALL** - calls a system function in 64-bit mode. The instruction takes no operands and, according to this appendix, works only in 64-bit mode. Execution moves to privilege level 0. CS is loaded from `IA32_STAR` bits 32-47 and SS is formed by adding 8 to CS. RIP is loaded from `IA32_LSTAR`, while the return address is saved in RCX. The source says the new flags value is obtained by performing AND with `IA32_FMASK`; the old flags value is saved in R11. `IA32_EFER.SCE` must be set to enable the instruction.

*Editorial note: the `IA32_FMASK` operation is described here exactly as in the book. Architectural manuals describe the precise flag-masking operation differently.*

9. **SYSRET** - returns from a system function in 64-bit mode. The instruction takes no operands, may execute only at privilege level 0, and always transfers to privilege level 3. It can return either to 64-bit mode or to compatibility mode. With `REX.W`, the return is to 64-bit mode; without `REX.W`, the return is to compatibility mode. The source prints that the new flags register is loaded from R11. For compatibility-mode return, CS comes from `IA32_STAR` bits 48-63 and SS is formed by adding 8 to CS. For 64-bit return, CS comes from `IA32_STAR` bits 48-63 plus 16, while SS uses `IA32_STAR` bits 48-63 plus 8. RIP is loaded from RCX, or ECX for compatibility-mode return. `IA32_EFER.SCE` must be set.

*Editorial note: the printed register name is R11, with a lowercase letter l, rather than R11. It is preserved here as a source typo.*

## C.3. Working with the Processor Cache

1. **INVD** - invalidates the processor's internal cache. The instruction takes no operands. Modified data is not written back to main memory. The instruction does not affect the TLB and may be executed only at privilege level 0.
2. **INVLPG** - removes the TLB entry associated with the specified address. Syntax: `invlpg <address>`. The instruction may be executed only at privilege level 0.
3. **WBINVD** - invalidates the processor's internal cache with write-back. All modified data in the processor cache is written to main memory. The instruction does not affect the TLB and may be executed only at privilege level 0.

## C.4. Additional Instructions

1. **ARPL** - raises the Requested Privilege Level (RPL). Syntax: `arpl <destination>, <source>`. The first operand may be a 16-bit register or memory location; the second may only be a 16-bit register. ARPL compares the low two RPL bits of the destination and source. If the destination RPL is numerically smaller, the destination RPL is replaced by the source RPL and ZF is set. Otherwise ZF is cleared. The source says ARPL may execute at any privilege level, but is practically useful only at levels 1 and 2. It does not operate in 64-bit mode.

2. **CLI** - clears IF. The instruction takes no operands. If the current privilege level is numerically greater than IOPL, #GP is generated. If CR4.VME is set while the processor is in virtual-8086 mode and IOPL is less than 3, VIF is cleared instead of IF and #GP is not generated. If CR4.PVI is set, CPL is 3, and IOPL is less than 3, VIF is likewise cleared instead of IF without #GP.

3. **STI** - sets IF to one. The source states that the instruction is identical to CLI.

*Editorial note: 'identical to CLI' is retained from the source. In context, it refers to the corresponding privilege/virtual-interrupt rules, while STI sets rather than clears the interrupt flag.*

4. **INT3** - invokes the breakpoint exception handler (#BP) using the short opcode. The instruction takes no operands. The source distinguishes it from INT 3 by saying INT3 uses a one-byte opcode whereas INT 3 occupies two bytes. It also says that when CR4.VME or CR4.PVI is set, the breakpoint handler is invoked regardless of whether it is prohibited.

5. **LAR** - loads descriptor access rights. Syntax: `lar <destination>, <selector>`. The selector may be supplied in a register or memory; the destination must be a register. The selector may identify a descriptor in either the GDT or LDT. If the destination is a 16-bit register, the source says the third word of the descriptor, bytes 5 and 6, is extracted, ANDed with 0FF00h, and placed in the destination. If the destination is a 32-bit or 64-bit register, the source says the second doubleword, bytes 5-8, is extracted, ANDed with 00FFFF00h, and placed in the destination.

6. **LSL** - loads a segment limit. Syntax: `lsl <destination>, <selector>`. The selector may be in a register or memory; the destination must be a register. The selector may identify a descriptor in the GDT or LDT. The instruction loads the segment limit into the destination, with the limit calculated according to the granularity bit.

7. **UD2** - generates the invalid-opcode exception (#UD). It takes no operands and may be executed in any processor mode and at any privilege level.

8. **VERR** - checks whether a segment can be read. Syntax: `verr <selector>`. The selector may be supplied in a register or memory. If the current code is permitted to read the segment described by the selector, ZF is set; otherwise it is cleared.

9. **VERW** - checks whether a segment can be written. The source says its operation is identical to VERR.

# Alphabetical Index

Page references correspond to this 153-page revised English edition. Page numbers are clickable.

## A

|                                                  |            |
|--------------------------------------------------|------------|
| Accident                                         | 42         |
| Address                                          |            |
| virtual. See Address, linear                     |            |
| linear                                           | 12, 47     |
| logical                                          | 12, 31, 47 |
| physical                                         | 12, 47     |
| AP (application processor)                       | 138        |
| APIC, Advanced Programmable Interrupt Controller | 129        |
| I/O APIC                                         | 129, 132   |
| local                                            | 129, 131   |
| APIC, local. See APIC, local                     |            |
| API functions                                    | 74         |
| ARPL, instruction                                | 148        |

## B

|                             |    |
|-----------------------------|----|
| Bit map                     | 53 |
| BSOD, critical system error | 97 |

## C

|                                                   |     |
|---------------------------------------------------|-----|
| CLI, instruction                                  | 148 |
| CLTS, instruction                                 | 147 |
| Command. See also Instruction; System instruction |     |
| Controller                                        | 7   |
| CPL, privilege level                              | 67  |
| CR0, register                                     | 143 |
| CR2, register                                     | 143 |
| CR3, register                                     | 144 |
| CR4, register                                     | 144 |
| CR8, register                                     | 146 |
| CreateFile, function                              | 79  |
| Critical section                                  | 138 |

## D

|                                              |     |
|----------------------------------------------|-----|
| DDK, driver-development toolkit              | 88  |
| DeviceIoControl, function                    | 80  |
| DFR, register                                | 137 |
| DLL, library                                 | 82  |
| creation                                     | 86  |
| DMA, direct memory access                    | 92  |
| DOS program                                  | 80  |
| DPL, privilege level                         | 68  |
| Driver                                       |     |
| video adapters                               | 88  |
| startup and control                          | 88  |
| streaming devices                            | 88  |
| legacy                                       | 88  |
| device                                       | 88  |
| file systems                                 | 88  |
| Dynamically linked library. See DLL, library |     |

## E

|                    |          |
|--------------------|----------|
| EBP, register      | 76       |
| ENDP, macro        | 77       |
| ENTER, instruction | 76       |
| EOI, operation     | 131, 132 |
| EOI, register      | 132      |
| Error              | 41       |

## F

|                       |    |
|-----------------------|----|
| FreeLibrary, function | 83 |
|-----------------------|----|

|                      |    |
|----------------------|----|
| Function             |    |
| prologue             | 77 |
| epilogue             | 77 |
| Function epilogue    | 77 |
| Function prologue    | 77 |
| Function stack frame | 76 |

## G

|                                  |            |
|----------------------------------|------------|
| Gate                             |            |
| call                             | 68         |
| descriptor                       | 39         |
| task                             | 39, 61, 68 |
| trap                             | 39, 68     |
| trap descriptor                  | 40         |
| interrupt                        | 39, 68     |
| interrupt descriptor             | 40         |
| GDTR, register                   | 33, 145    |
| GDT. See Global Descriptor Table |            |
| GetProcAddress, function         | 83         |
| Global Descriptor Table          | 33         |
| GUI program                      | 79, 81     |

## H

|                            |        |
|----------------------------|--------|
| Handle (object descriptor) | 79     |
| module                     | 83, 85 |
| access level               | 79     |
| HLT, instruction           | 148    |
| HyperThreading, technology | 136    |

## I

|                                     |            |
|-------------------------------------|------------|
| ICR, register                       | 136        |
| IDTR, register                      | 145        |
| IDT. See Interrupt Descriptor Table |            |
| Import                              |            |
| dynamic                             | 83         |
| static                              | 83         |
| Instruction                         | 7          |
| INT3, instruction                   | 148        |
| Interrupt                           | 12, 39, 89 |
| SMI                                 | 131        |
| maskable                            | 39         |
| nonmaskable                         | 131        |
| from system timer                   | 131        |
| Interrupt Descriptor Table          | 39         |
| Interrupt handlers                  | 39         |
| Interrupt request levels            | 89         |
| INVD, instruction                   | 148        |
| INVLPG, instruction                 | 148        |
| I/O APIC, input/output controller   | 129, 132   |
| I/O manager                         | 90         |
| IRET, instruction                   | 148        |
| IRP, -                              | 90         |
| IRQL, interrupt request levels      | 89         |
| IST, mechanism                      | 111        |
| ITD, table                          | 111        |

## L

|                  |         |
|------------------|---------|
| Label            | 18      |
| anonymous        | 21      |
| near and far     | 19      |
| LAR, instruction | 148     |
| LDR, register    | 137     |
| LDTR, register   | 60, 145 |

|                                 |        |                                   |          |
|---------------------------------|--------|-----------------------------------|----------|
| LDT. See Local Descriptor Table |        |                                   |          |
| LEAVE, instruction              | 76     | Program                           | 80       |
| LGDT, instruction               | 147    | DOS                               | 79, 81   |
| LIDT, instruction               | 147    | GUI                               | 79, 80   |
| LLDT, instruction               | 147    | console                           |          |
| LMSW, instruction               | 147    | <b>R</b>                          |          |
| LoadLibrary, function           | 83     | RDMSR, instruction                | 148      |
| Local Descriptor Table          | 57, 59 | RDPMC, instruction                | 148      |
| LOCK, prefix                    | 138    | RTDSC, instruction                | 148      |
| Long mode                       | 10     | RTDSCP, instruction               | 148      |
| LSL, instruction                | 149    | Read File, function               | 80       |
| LTR, instruction                | 147    | Register                          | 8        |
| LVT, Local Vector Table         | 130    | DFR                               | 137      |
| <b>M</b>                        |        | GDTR                              | 33       |
| Macro                           | 28     | ICR                               | 136      |
| MDL, list                       | 92     | LDR                               | 137      |
| Memory                          | 8      | LDTR                              | 60       |
| cache memory                    | 8      | task (TR)                         | 60       |
| register memory                 | 8      | general-purpose                   | 15       |
| segment                         | 31     | memory-mapped                     | 129      |
| segment descriptor              | 31     | privileged                        | 15       |
| segment selector                | 31     | flags                             | 13       |
| Message                         | 82     | Relocation                        | 73       |
| MONITOR, instruction            | 148    | Return address                    | 9        |
| MOV (CRn), instruction          | 147    | RPL, privilege level              | 67       |
| MOV (DRn), instruction          | 148    | RSM, instruction                  | 148      |
| Multicore processor             | 136    | <b>S</b>                          |          |
| Multiprocessor systems          | 136    | SCM, Service Control Manager      | 88       |
| Multitasking                    | 57     | SCP, service-control program      | 88       |
| MWAIT, instruction              | 148    | Segment                           | 31       |
| <b>O</b>                        |        | descriptor                        | 31       |
| Object                          |        | selector                          | 31       |
| incomplete name                 | 80     | SEH, exception-handling mechanism | 83       |
| descriptor (handle)             | 79     | Selector                          | 33       |
| full name                       | 80     | Services                          | 88       |
| user                            | 79     | SGDT, instruction                 | 147      |
| system                          | 59     | Shift                             | 16       |
| kernel                          | 79     | SIDT, instruction                 | 147      |
| Opcode                          | 8      | SLDT, instruction                 | 147      |
| OpenFile, function              | 80     | SMM, System Management Mode       | 131      |
| <b>P</b>                        |        | SMSW, instruction                 | 147      |
| Page                            | 47     | Spin lock                         | 138      |
| Page directory                  | 47     | Stack                             | 9        |
| Privilege level                 |        | STI, instruction                  | 148      |
| CPL                             | 67     | STR, instruction                  | 147      |
| descriptor                      | 68     | Structure                         | 28       |
| requested                       | 67     | SWAPGS, instruction               | 148      |
| Processor                       | 7      | Symbolic link                     | 91       |
| AP (application processor)      | 138    | SYSCALL, instruction              | 117, 148 |
| BSP (bootstrap processor)       | 138    | SYSENTER, instruction             | 74, 148  |
| multicore                       | 136    | SYSEXIT, instruction              | 148      |
| submodes                        | 11     | SYSRET, instruction               | 117, 148 |
| mode - long mode                | 10     | System object                     | 59       |
| mode - protected                | 10     | <b>T</b>                          |          |
| mode - real                     | 10     | Table                             |          |
| mode - system management        | 11     | ITD                               | 111      |
| core                            | 136    | import                            | 73, 75   |
| Processor core                  | 136    | local vector                      | 130      |
| PROC, macro                     | 77     | redirection                       | 132      |
|                                 |        | pages                             | 47       |
|                                 |        | handles                           | 79       |
|                                 |        | export                            | 73, 82   |

|                                                              |     |
|--------------------------------------------------------------|-----|
| Task                                                         | 57  |
| I/O permission map                                           | 57  |
| Local Descriptor Table                                       | 57  |
| Task State Segment                                           | 57  |
| TEB, data area                                               |     |
| data area                                                    | 83  |
| Thread context switch                                        | 89  |
| Timer                                                        | 131 |
| Trap                                                         | 41  |
| TR, register                                                 | 145 |
| TSS. See Task State Segment                                  |     |
| TSS, Task State Segment                                      |     |
| 64-bit                                                       | 111 |
| <b>U</b>                                                     |     |
| UD2, instruction                                             | 149 |
| UnhandledExceptionFilter; operating-system supplied function | 83  |
| <b>V</b>                                                     |     |
| VERR, instruction                                            | 149 |
| VERW, instruction                                            | 149 |
| <b>W</b>                                                     |     |
| WBINVD, instruction                                          | 148 |
| WDK, driver-development toolkit                              | 88  |
| WDM driver                                                   | 88  |
| Win32 operating system                                       |     |
| thread                                                       | 73  |
| process                                                      | 73  |
| WriteFile, function                                          | 80  |
| WRMSR, instruction                                           | 148 |
| <b>X</b>                                                     |     |
| xAPIC, Advanced Programmable Interrupt Controller            | 129 |

Ruslan Zufyarovich Ablyazov

## **Programming in Assembly on the x86-64 Platform**

**Editor-in-chief** D. A. Movchan

dm@dmk-press.ru

**Literary editor** O. V. Gotlib

**Layout** A. Yu. Tatarinov

**Cover design** A. G. Movchan

Signed for printing 02.02.2011. Format 70 x 100 1/16.

Petersburg typeface. Offset printing.

Conventional printed sheets: 24.6. Print run: 1000 copies.

No.

Publisher's electronic address: [www.dmk-press.ru](http://www.dmk-press.ru)