N.G.Hitrovo

First steps

towards system programming

under MS-DOS7

2009

UDC 004.451.9dos

N.G. Hitrovo

First steps towards system programming under MS-DOS7.

This book is for those who intend to seize the most intimate secrets of modern computer technology. In order to be understood easily, narration starts with some elementary explanations and examples, but its main part presents professional level data with all relevant latest updates. These data will be beneficial for both students and teachers on computer technology as well as for computer maintenance staff, system programmers and developers of x86-platform microprocessor equipment.

Table of contents

Introduction

Nowadays the windows of Windows-2000/XP operating system have become familiar in home interior. You look there for work and for fun. One easily gets an impression of ability to access just everything. But it's a deceptive impression: the actual origin of all computer's wonders can't be seen via these windows.

Each advanced user once comes to a sense of a virtual boundary, arranged by the Windows operating system to prevent intrusions into its internal affairs. If you want to understand computer more thoroughly, you have to choose a way leading beyond the familiar windows. But outside the windows there is darkness with no visible fulcrum for support. If you are still eager to go further, then this book is just for you.

0.01 Applied and system software

All computer's wonders are performed by programs. More accurately, there is a complex interaction of applied and system software, each playing its own role. For example, the well-known WORD editor program is a typical application, because it needs run-time environment, provided by the Windows operating system. On the other hand, system software is the one that creates, configures and maintains the operating environment.

Much attention is now given to application programming. Microsoft corporation, the owner of Windows operating system, actively promulgates its VISUAL STUDIO packet, comprising several high-level languages for writing applications. Microsoft's interest is obvious: the more applications will require Windows operating system, the more Windows usage licenses will be sold.

Attitude to system programming is quite opposite: leading software vendors are determined to prevent any appearance of rival system products. The right to buy some system data is given to reliable business partners only, and not every company can afford such expense. Accepted information policy doesn't foster attention to system programming, but the latter doesn't become less significant because of that. One can't be a true professional in computer technology without some knowledge and experience in system programming.

System software is not limited to operating systems only. It also includes hardware drivers, fixed BIOS programs, various diagnostic and recovery services. Study of these software functions has always been an important element of computer technology education. Traditionally system programming is taught on the basis of low-level assembler language (MASM or TASM) under documented versions of Microsoft's DOS (MS-DOS).

Now all the documented versions of MS-DOS have become so obsolete, that can't even give access to large storage media used in modern computers. More recent operating sys-

tems protect themselves against any attempt of intrusion into their affairs and respond with a known error message "Your program has performed an illegal operation and has to be shut down". Nevertheless an acceptable solution exists. It is an undocumented operating system, which can be installed on modern computers and allows to do just everything. Main purpose of this book is to make you acquainted with that operating system and with it's usage for solving simple system tasks.

0.02 Real mode and protected mode

Protection against execution of inadmissible functions is hardware implemented inside processors (CPU), and it is active while the CPU operates in protected mode. Besides the protection itself, protected mode gives several important advantages, and this is why it has become the main operating mode for modern CPUs. For application programs and for the user the protected mode resembles a virtual shell, disabling all actions, which may inflict any harm to vital functions of operating system. This is the main factor of high reliability, inherent to modern operating systems.

Contrary to the protected mode, real mode is a "defenseless" mode, making CPU to emulate archaic processor i8086. One may wonder why this obsolete feature is kept supported in each next CPU generation and is not abandoned yet? There is the only reason: real mode is necessary for modern computer systems. First, real mode is required by BIOS: it must have free access to computer hardware for performing the POST tests. Just because of this mission all processors are automatically set into real mode each time the power supply is switched on. Operating systems too can't obtain control over computer's hardware unless protection is disabled, and therefore must start while CPU runs in a "defenseless" real mode.

Having got total control over computer, operating systems of the Windows clone prepare protection data structure so that after switching to protected mode the highest privilege level is granted to Windows OS itself, whereas both the user and applications are given the third (the lowest) privilege level. Since then and for ever the user wouldn't be allowed to change this allocation of rights. Because of the same reason all opportunities beyond the limits of Windows' API become inaccessible for the user and for his application programs in protected mode.

Nowadays ordinary user's practice doesn't imply experience to act beyond the restrictions, inherent to protected mode. Fundamental concept of modern operating systems is distinction between the user's and system's spheres of responsibility. Excessive user's curiosity is considered destructive and should be suppressed. Protected mode hasn't proved its efficiency against malignant viruses; it provides an effective protection ... against you.

0.03 Why MS-DOS 7 ?

If an operating system permits to run applications in real mode, it can't be as stable as those using protected mode. This is the main reason why old-fashioned DOS-like operating systems have been ousted by more modern ones. But ordinary ratings are not valid for emergency services, which often require unlimited access rights. Then the main drawback of DOS-like operating systems turns into their unique advantage. Therefore computer professionals haven't forgotten DOS. All bootable diskettes (reparatory, diagnostic, disk service, etc.) need an operating system and load just DOS. Most part of bootable compact disks is DOS-based as well. Besides all, DOS is the simplest operating system, and just for that it is the most suitable for primary study of system functions.

Though DOS is often considered a real-mode operating system, this opinion is not completely true. Initially DOS works in real mode, but it wouldn't object against mode change undertaken either by a driver (5.04-02) or by the user. Thus you get a unique opportunity to assign the highest privilege level to yourself. Only in the latter case the CPU would obey to just any your command, including those which are allowed for execution exclusively in protected mode at the highest privilege level. No one other operating system would transfer its highest privileges to the user. Only DOS can give you full freedom of action in both real and protected modes.

Practical need of real-mode access forces software vendors to continue development of DOS-like operating systems. Independent FreeDOS project slowly advances towards completion (http://www.freedos.org/). One more commercial version of ROM-DOS recently has emerged (http://www.datalight.com/). Some less fresh shareware versions also have found their interested consumers. A lot of drivers are written in order to supply obsolete DOS versions (MS-DOS 6.22, IBM PC DOS 2000, etc.) with new properties, including access to disks with widespread file systems FAT-32 and NTFS. But drivers, not integrated into the DOS's core, give no opportunity to install DOS onto such disks.

If you have ever dared to buy a computer without preinstalled commercial operating system, you almost certainly found there a typical DOS prompt and a HDD (hard disk drive) formatted with FAT-32. Most probably neither of the mentioned DOS versions has any relation to that. Computer initialization is usually done with tools, supplied by Microsoft on emergency diskettes for Windows-95/98 operating systems. Formally it is known as "command line only" loading mode.

Simple examination reveals, however, that "command line only" mode doesn't resemble the Windows operating system, but rather is a typical version of DOS. Inside almost every file's code on emergency diskette you can find the proof - a string "MS-DOS Version 7...". The 7-th version of MS-DOS is just that Microsoft's undocumented operating system, which is the main subject of this book. For MS-DOS taken from Windows-95 OSR2 release, function INT 21\AH=30h (8.02-22) reports exact version number 07.0Ah, or decimal 7.10. It is just this version which here and forth is referred to as MS-DOS7.

MS-DOS7 is not the latest version of MS-DOS. Windows-ME release is based on MS-DOS8. Having been recompiled for modern CPUs, MS-DOS8 became more compact, but has lost compatibility with some 486 CPU clones. Besides this, it doesn't play an active role during loading of Windows-ME and thus is unable to implement different loading schemes. But other features of MS-DOS8 are similar to those of MS-DOS7, so most data in this book are equally true for both. Each exception is explicitly noted.

0.04 What is this book about?

System programming is a vast theme; its narration tends to grow beyond all affordable limits. Therefore this book doesn't pretend to completeness: several large topics (in particular, networks) have been deliberately left aside. Some other items are touched in short, as far as it is enough for comprehension.

Chapters 1 - 4 of this book make the reader acquainted with keyboard, with command line composition and with internal commands. These short chapters are addressed to newbies who had no deal with MS-DOS7 ever before.

Chapter 5 describes important drivers for computer's hardware, including the most recent ones, developed by various software vendors during 2004 – 2008. Chapter 6 presents a survey of selected utilities for MS-DOS7. Special attention is paid to programming instrument DEBUG.EXE – the worst documented utility inside the undocumented DOS. Chapter 7 is devoted exclusively to DEBUG's assembler commands. The 8-th chapter describes various services, which can be called for in MS-DOS7 via interrupts.

The 9-th chapter presents examples of programming with the tools from Windows-95/98 standard release. Examples will help you to write your own interpretable and executable files according to your needs. Presented selection of examples illustrates the scope of opportunities, which can become available in MS-DOS7, if it is properly asked for.

The last "A"-th chapter consists of 16 thematic appendixes with a lot of data tables, concerning both MS-DOS7 and AT-compatible PCs. The last 16-th appendix is a vocabulary, explaining the abbreviations used in this book.

0.05 Some more remarks

For a long time leading software vendors inculcate hard selling of operating systems, which inhibit user's access to real mode. But sales of both OS/2 (IBM, 1989) and Windows-NT (Microsoft, 1994) went bad. The next attempt – Windows-2000 – opened an opportunity to employ DOS's services on disks formatted with FAT-32. As soon as success of Windows-2000 became obvious, Microsoft decided to kill the whole rival Windows-95/98/ME brood. However, this decision doesn't eliminate necessity of real mode and provides no alternative real-mode tools. I suggest to regard this Microsoft's decision as a

guarantee that your study of MS-DOS7 today will not go in vain tomorrow due to advent of any better version of MS-DOS.

One more challenge to renewal arose in 2002, when Intel has developed and began to produce the Itanium CPU, providing no support for old-fashioned 16-bit machine code. All former real-mode tools, including MS-DOS7, could be turned into trash by forthcoming emergence of new PCs with 32-bit BIOS code, and then this book were not worth writing. Seven years have elapsed since, but the expected marvel hasn't happened. Public PCs with Itanium CPU have not emerged. One has to admit, that preservation of support for 16-bit code in all newer CPUs must be caused by firm reasons. Until these reasons persist, experience in DOS will still be beneficial for you.

How to get MS-DOS7 launched? If Windows-95/98 is installed already, just keep the F8 key pressed while PC starts to boot operating system, and you'll get into boot menu; then choose "command line only" option, and you're there. Otherwise you'll have to get a Windows-95/98 emergency diskette, and boot your PC with this diskette. Suitable images of bootable diskettes can be found in many internet sites, for example, in http://www.bootdisk.com/ . Standard loading procedure leaves you with a "raw" DOS's command line. MS-DOS7 may appear much more convenient and friendly, if you'll follow the advices given in parts 6.25 and 9.01 of this book. Part 9.11 suggests some other ways of launching MS-DOS7, including those sharing common disk with Windows OS.

This book is not based on legal OEM description of MS-DOS7, since it hasn't been published yet. Author of this book is the only one responsible for all its contents. Despite genuine desire to check every data item, the 100% veracity of data can't be guaranteed: author's resources are not infinite. Besides that, evolution wouldn't stop, it permanently engenders something new. Therefore the author can't admit responsibility for consequences of your actions, even those inspired by this book. Any risk-bearing affair requires some caution and competence. However there is a hope that owing to this book your actions will become competent, effective and safe.

Your remarks and corrections concerning this book should be sent by e-mail to For-H@yandex.ru. Each relevant message will be accepted with gratitude.

Chapter 1 Let's get acquainted with keyboard

For all DOS-like operating systems the main input means is keyboard. Each keystroke invokes execution of at least one preloaded (resident) software module. Sometimes this execution doesn't reveal itself at all, but more frequently it brings about an appearance of the corresponding character on the screen. Several keys (the "hot" keys) may be charged with more complex missions. Resident software modules, which define key's missions, are loaded in memory by either

– BIOS (Basic Input-Output system);
– DOS loader from IO.SYS file (active during loading only);
– the CON device (console) driver from DOS's core;
– command interpreter (usually COMMAND.COM).

This chapter doesn't describe other "hot" key functions, which may be assigned by other TSR (TSR = terminate and stay resident) utilities or file managers: Norton Commander, Volcov Commander, etc. Being loaded later, TSRs intercept some original keyboard functions – as most file managers do – or substitute with partial (not exact) emulation. Command line presented by file managers is NOT the same as original DOS's command line: a large part of original keyboard functions is intercepted and made inactive.

The following text describes functions of the most common "enhanced" keyboard, which usually has from 101 to 108 keys. Evolution of key's functions is shown just as they are activated while loading MS-DOS7. The assumed final stage of this evolution is that implemented by command interpreter COMMAND.COM.

1.01 Keyboard functions of PC's BIOS.

When switched on, computer starts under control of its BIOS system. BIOS loads the INT 09 handler (8.01-09) and thus becomes able to sense keyboard controller calls sent via the IRQ 1 line. Every keystroke is sensed, but only a few keys and key combinations invoke a certain response. Common BIOS versions assign special mission to the following keys and key combinations:

Ctrl-Alt-Delete – initiates a "warm" reboot.

Delete – launches BIOS Setup procedure, which enables to set BIOS' parameters (see notes 1 and 2). This function is deactivated after about 2 seconds, so you have to keep the "Delete" key pressed just when computer starts.

Pause/Break (or Ctrl-Break) – causes temporary stop until any other key is pressed, thus giving an opportunity to read screen messages.

Shift-PrtScr – sends current screen to printer via LPT1 port. The printer must be ready and must be able to respond properly (those "designed for WINDOWS" or for USB port wouldn't suit!).

There is no uniform standard for BIOS' keyboard functions, so they may somewhat differ. For example, the 8-th BIOS version from American Megatrends provides supplementary PC's loading menu, invoked with the F8 key. Nevertheless some BIOS' keyboard functions de-facto have become standard, in particular, these of the DELETE key (note 2) and of the CTRL-ALT-DELETE key combination.

Keyboard functions set by BIOS may be deactivated later, just as any software implemented function. In particular, the Shift-PrtScr keystroke combination is often deactivated by embedded software of the video card. Other BIOS' functions may fail because of INT 09 interception, or because of software crash, affecting data in interrupt table (from 0000:0000h to 0000:0400h) or in BIOS memory area (A.01-1). Therefore the most important reboot function in some PCs is hardware duplicated by a RESET button.

Note 1: BIOS Setup program may make active some other keys – it depends on BIOS version. These keys enable to change some parameter settings, including the one preventing BIOS' logo display. Without BIOS' logo you will be able to see current BIOS' messages.

Note 2: some (largely obsolete) BIOS versions launch BIOS Setup procedure with F1, F2, F10, ESC keys or with F3-F2, Ctrl-Alt-S, Ctrl-Alt-Ins, Ctrl-Alt-Esc key combinations.

Note 3: some models of "enhanced" keyboard, developed in 1990-ties, have a supplementary TURBO key. TURBO-F11 key combination toggles keyboard lock ON/OFF, and TURBO-F12 key combination toggles beeper sound lock ON/OFF. Most modern keyboards have no TURBO key.

Note 4: for power supply control via keyboard many keyboard models are equipped with three special keys: POWER, SLEEP and WAKE UP. These keys are intended to be served by BIOS, but their functions are intercepted, in particular, by Windows-XP and by Windows Vista.

1.02 Keyboard functions of the DOS loader

At some moment during PC's loading the BIOS' logo is replaced by the logo of the operating system. This logo change signifies termination of BIOS's loading mission. Since that moment control is transferred to operating system's loader.

Both Windows-95/98 and MS-DOS7 operating systems have the same primary loader, contained as a part inside the IO.SYS file. The loader begins its mission with reading parameters stored in MSDOS.SYS file (5.01-01). Then according to these parameters the DOS loader activates for a time some more "hot" keys and key combinations:

F5 – turns subsequent loading into Safe Mode with WINDOWS' GUI (graphic user's interface) and default settings, ignoring commands in configuration files (CONFIG.SYS and AUTOEXEC.BAT).

SHIFT-F5 – turns subsequent loading into "command line only" mode with all drivers, but without WINDOWS' GUI. This is in fact loading of MS-DOS7.

F6 – turns subsequent loading into Safe Mode (just as F5) plus adds network support.

F8 – invokes display of WINDOWS' standard boot menu and stops further process until user makes his choice. In MS-DOS8 the CTRL key duplicates the function of F8 key.

SHIFT-F8 – turns execution of configuration files into step-by-step mode; this enables to skip some operations according to the user's choice.

The listed key functions in MSDOS7 and MSDOS8 differ from those implemented in previous DOS versions. Activation of the mentioned "hot" keys may be affected by parameters (BOOTDELAY, BOOTKEYS, BOOTMULTI), specified in configuration file MSDOS.SYS (5.01-01). Of course, loading of WINDOWS' GUI can't be performed, when MS-DOS7 is used as a stand-alone operating system, and the GUI software isn't physically accessible.

For the time of displaying boot menu the DOS loader activates cursor keys (up, down), the ENTER key and numerical keys (0 – 9) in the common part of keyboard. NUMLOCK key is activated too; when NUMLOCK is set ON, numerical (rightmost) keypad may be used to choose menu items by number. After the choice is made, all menu keys are deactivated. If the chosen menu item doesn't imply anything else, then the DOS loader begins interpretation of commands from CONFIG.SYS configuration file (example – in 9.01-01).

During step-by-step confirmation the choice may be made by Y (yes), N (no), ENTER (= yes) and A keys (A = yes for all following lines). Normal (non-step-by-step) execution doesn't produce any effect on the screen, because at that time Window's logo is displayed. However, the logo display may be suppressed by setting the "Logo=0" parameter in the MSDOS.SYS file (5.01-01). Then on the screen you will see rapidly scrolling messages from the drivers being loaded. In order to be able to examine these messages carefully, you may suspend loading process with a PAUSE/BREAK keystroke or with CTRL-S keystroke combination. Then after any other keystroke the loading process will be resumed.

Having interpreted all the lines in CONFIG.SYS configuration file, the DOS loader deactivates all its "hot" keys and transfers control to command interpreter COMMAND.COM.

Note 1: in version 7.00 of MS-DOS the DOS loader activates the F4 key in order to start previous DOS version loading. Since version 7.10 of MS-DOS this option has been abolished.

1.03 Keyboard functions during batch file execution.

Control over the PC is given to the main command interpreter when DOS configuration procedure is not finished yet. The first mission of COMMAND.COM is interpretation of commands in lines of the last configuration file AUTOEXEC.BAT (example – in 9.01-02). During interpretation COMMAND.COM is quite self-sufficient, but at the same time the user's role is not diminished to zero. If Window's logo display is turned off, then user can monitor current messages. Besides that, command interpreter keeps active some "hot" keys, which enable user to suspend or to stop interpretation of batch command file.

Both suspension and stoppage of execution initiate a complex succession of calls (see 8.01-95 for details), involving resident modules of DOS's core as well as those installed by PC's BIOS. Therefore exact action of some "hot" keys may depend on BIOS version. Nevertheless the activated "hot" keys are the same in all AT-compatible computers.

The Break/Pause key provides a temporary stop, enables to read messages, but gives no opportunity to terminate execution of batch file: any following keystroke resumes execution.

The user can terminate batch file execution with "hot" keystroke combinations CTRL-C, CTRL-BREAK and ALT-03 (the latter digits "03" must be typed via the rightmost numerical keypad). Action of these keystroke combinations depends on the way the command interpreter has been launched. When COMMAND.COM starts permanently with /K or /P parameter (6.04), the mentioned keystroke combinations stop execution and display an offer

```
"Terminate batch job? Y/N"
```

thus providing an explicit choice. But if COMMAND.COM starts with /C parameter (6.04) for execution of a single job (as inside TSR shells), then CTRL-C, CTRL-BREAK and ALT-03 keystroke combinations terminate batch file interpretation at once, giving no chance to resume.

The CTRL-S keystroke combination provides a temporary stop, but (unlike the BREAK key) always enables to have a choice. You may resume execution by pressing any other key, except CTRL-C, CTRL-BREAK, ALT-03 and CTRL-2. Action of CTRL-C, CTRL-BREAK and ALT-03 depends on the way COMMAND.COM has been launched just as it was described above. CTRL-2 acts similarly, but only when execution has been suspended already by CTRL-S. Normal batch interpretation will not be affected by CTRL-2 keystroke combination.

Keyboard functions, breaking interpretation of a batch file, can be disabled by CTTY NUL command (see 3.07) in a line of the same batch file. But this trick may be justified in special circumstances only (example – in 9.03-02).

Having finished interpretation of commands from AUTOEXEC.BAT file, command interpreter COMMAND.COM by default must transfer control to the Windows' GUI loader - the WIN.COM file. But this wouldn't happen, if

– the WIN.COM file isn't found (for example, on a bootable diskette);
– the SHIFT-F5 key combination has been pressed at start (1.02);
– in Windows' boot menu the "command line only" option has been chosen;
– in MSDOS.SYS file (5.01-01) the "BootGUI=0" parameter is specified;
– the SINGLE parameter is given to the DOS command (note 1 to 4.08).

In any of the listed cases MS-DOS7 is loaded instead of the Windows operating system, command interpreter COMMAND.COM begins to accept commands from keyboard, displays its prompt on the screen, and since that moment works with keyboard in a quite different way.

1.04 Keyboard input of commands and textual lines.

When COMMAND.COM presents its command line, it accepts input via the CON device driver. The latter enables to type characters, digits and special symbols according to keyboard layout table resident in memory. Symbols may be entered either by corresponding keys or by ASCII decimal symbol's numbers. The latter should be typed via numerical keypad while the ALT key is kept pressed. Each next character appends the current command line and increments by 1 current position pointers for the current line and for the internal memory buffer, where the previous line is automatically stored.

Selection of symbols, typed with literal keys, depends on the state of SHIFT and CAPSLOCK keys. Pressing CAPSLOCK toggles keyboard into upper case letters selection and back. Pressing SHIFT turns keyboard into upper case selection for a while until SHIFT key isn't released. Besides that, DOS presents some limited opportunities of editing current command line (1.05).

All mentioned features are equally true for typing textual lines. Transition from command line input to textual line input is initiated by an explicit request to the CON device driver, for example, with command

```
COPY CON Filename.txt
```

where

FILENAME.TXT is an arbitrary name of a file to store the text.

The COPY command is not a separate utility; it is an internal command of the interpreter COMMAND.COM (3.06). Textual line input also can be implemented by other utilities, which are able to address the CON device driver in the same way.

Differences between textual input and command line input evince in what happens with the typed line when user confirms that typing the line is finished with the ENTER key-

stroke or with equivalent CTRL-M keystroke combination. Textual lines are written into an internal buffer, replace there the previous line, and then a new line is opened for input. The end of each textual line is marked with bytes 0Dh 0Ah, and in this form it is appended to previous lines in memory area, devoted for text storage. The COMMAND.COM interpreter enables to send this text into a specified file or into a selected channel (see 3.06 for details). Return from textual input back to command line input can be performed with keystroke combinations F6-ENTER, CTRL-Z-ENTER or ALT-26-ENTER. In the latter combination the digits must be typed via the rightmost numerical keypad.

When the ENTER keystroke signifies entering of a command line, this line is also written into an internal line buffer, just as textual line, but the following events occur otherwise. In buffer the line is parsed in order to extract the command name and to discriminate whether it is an internal command or a separate utility. While parsing MS-DOS7 regards upper case letters and lower case letters as identical. Separate utilities are searched for (2.02-02), read from their media and prepared for execution. When execution terminates, control is returned back to command interpreter. The latter displays it's prompt in a new line and begins to wait for the next command line input.

If the main object of the parsed line is a name of a command file, then command interpreter begins to read commands from this file and interprets them line-by-line. When execution of the last command line terminates, command interpreter returns to waiting for command line input from keyboard.

Specific contents of a command line, of course, depend on the command interpreter it is addressed to, and this is a subject of a separate consideration, which begins in chapter 2 of this book and is continued along all the following chapters.

1.05 Functional keys for line editing.

The most "hot" input key, no doubt, is the ENTER key. But besides it there are several other "hot" keys, charged with editing and control functions, which have a long history in former computer generations. Both interactive command interpreters in DOS – COMMAND.COM and DEBUG.EXE – inherit these functions. Some of them seem rudimentary, but some are still widely used.

BACKSPACE key (left arrow) decrements by 1 current position pointer for the current line and in internal memory buffer as well. Contents of internal memory buffer remain intact, but the last character in the current line looks lost; in fact this character is prepared to be overwritten with the following keystroke. Left arrow among cursor keys, left arrow in the numerical keypad, CTRL-H and ALT-08 keystroke combinations do just the same.

CTRL-2, CTRL-C, CTRL-BREAK and ALT-03 – all these key combinations erase contents of the current line and open an empty line to be typed anew. All previous textual input becomes lost.

CTRL-ENTER, CTRL-J, and ALT-10 key combinations wrap current input line and give an opportunity to continue it in the next lower row. This enables to see the whole line (up to 128 characters long by default) within the screen, which is usually 80 characters wide. The wrapped command line is always accepted just as if it were continuous.

CTRL-G and ALT-07 key combinations insert code 07h "Beep". It does nothing in textual and command lines, but having been sent to the screen, produces a vile sound signal instead.

CTRL-P and ALT-16 key combinations toggle data output from screen to printer and back. This is dangerous when printer is not ready, or is not connected, or is connected not to the default port (LPT1). In any such case INT 24 (8.02-84) is called for with its "Abort, Retry?" offer, but the "Abort" choice doesn't reset the output state (this seems to be a bug). In order to get back to normal command line you have to press CTRL-P once more, otherwise the offer is repeated indefinitely.

DEL (DELETE) key increments by 1 the pointer in internal memory buffer. This looks as though the previous line in the buffer has been shifted one position to the left. If then copying into the current line is performed, one character from the preceding line becomes skipped. Formerly this function was known as "SKIP1".

ESC, CTRL-ESC, CTRL-[and ALT-27 key combinations cancel the current line and open a new one. Formerly this function was known as "VOID". Contents of internal memory buffer remain intact. The cancelled command line is marked on the screen with a backslash "\", but the backslash itself doesn't cause the "VOID" action: you are free to type it in an ordinary way.

F1 keystroke appends current line with one character, copied from the same position in the previous line, which is kept stored in internal memory buffer. Formerly this function was known as "COPY1". Right arrow key in numerical keypad and right arrow cursor key do just the same.

F2 keystroke causes a pause, waiting for one character input. If the inputted character isn't present in the rest part of previous line in internal memory buffer, the F2 keystroke is aborted; but if the character is present, a number of characters preceding the inputted one are copied from internal buffer, appending the current line. Formerly this function was known as "COPYUP".

F3 keystroke fills line's empty space with characters from internal memory buffer, thus copying the previous command line. If current line contains several symbols already, then only the rest part of characters from inter-

nal memory buffer will be appended to existing part of the current line. Formerly this function was known as "COPYALL".

F4 keystroke affects internal memory buffer only. When pressed, F4 causes a pause, waiting for one character input. If the inputted character isn't present in rest part of internal memory buffer, the F4 keystroke is aborted; but if the character is present, preceding characters in memory buffer are deleted. This looks as though the contents of internal buffer became shifted to the left until the inputted character gets the same position as the cursor in the current line. The F4 function, formerly known as SKIPUP, enables to skip a part of preceding line from being copied into the current line by F1 or by F3 keystroke.

F5 keystroke copies current line into internal memory buffer, closes the copied line on the screen with "@" symbol, and opens a next (empty) command line, which is to be typed anew. The described action isn't caused by symbol "@" itself, so the latter may be used in an ordinary way.

F7 and ALT-00 insert code 00h, marked by symbol pair ^@. Code 00h interrupts parsing of command line: all characters after 00h will be ignored. But 00h doesn't interrupt text input: code 00h itself and all the following characters remain in the saved text.

INS (INSERT) keystroke toggles a bit (see A.02-3, 0040:0017), which controls incrementation of current position pointer in internal memory buffer. If you have copied a part of previous line, then stop incrementation, type in some new characters, and then copy the rest part of previous line, the result will look as if the new characters were inserted between adjacent characters of the previous line. To restore normal incrementation you have to press the INS key once more.

CTRL-I, ALT-09 and TAB keys insert code of horizontal tabulation 09h, which is expanded by the displaying procedure into 8 empty spaces. When text is saved into a file, tabulation symbol is not expanded. Some text editing programs can expand the 09h code into other number of spaces.

The listed original key functions are often intercepted by TSR (terminate and stay resident) programs, which can be loaded later. TSR shells (Norton Commander, Volcov Commander, etc.) usually intercept INS, DEL, F1 – F7 and several other keys, make them inactive or active in another way (see 6.25 for examples). Nevertheless original command line editing functions always remain active for input of textual lines and in DEBUG's (6.05) interactive sessions as well.

1.06 Keyboard layouts and character sets.

By default MS-DOS7 employs American English (US') set of characters, represented by codepage 437, but gives an opportunity to replace it according to national demands. Implementation of this opportunity involves several operations for changing both keyboard and display settings. Microsoft proposes the following sequence of operations:

- change limitations for names and some other setting by loading data from COUNTRY.SYS file (5.02-01, an example – in 9.01-01);
- prepare a memory place for additional codepage(s) by loading DISPLAY.SYS driver (5.02-02, an example – in 9.01-01);
- launch MODE.COM utility in order to load codepages and to make one of them active (6.18, an example – in 9.01-02);
- launch KEYB.COM driver in order to load alternative keyboard layouts (see 5.02-04, an example – in 9.01-02);
- load NLSFUNC.EXE driver (5.02-03) in order to enable switching between prepared codepages and national data sets.

National codepages inside WINDOWS-95 OS release are packed into four data files: EGA.CPI, EGA2.CPI, EGA3.CPI, ISO.CPI. Each national codepage contains 256 characters in two character sets: american english set (characters 32 – 127) and a national one (characters beyond the 128-th). Therefore switching between any national language and english language doesn't require swapping of codepages, it can be performed within any single national codepage. This is quite enough for all that limited variety of tasks, which are solved nowadays with MS-DOS7. Because of this reason swapping of codepages has come out of use.

Switching between different character sets (inside one codepage) is done with "hot" keys, arranged by KEYB.COM (5.02-04) or by any other keyboard driver (for example, KEYRUS.COM, see 5.02-05). In particular, KEYB.COM activates CTRL-rightSHIFT key combination for switching to a national character set and CTRL-leftSHIFT key combination to switch back to common english character set. KEYRUS.COM enables to activate various key combinations, including the mentioned ones.

Microsoft supplies keyboard layout tables packed in special keyboard data files KEYBOARD.SYS, KEYBRD2.SYS, KEYBRD3.SYS. Among these the KEYBOARD.SYS is the only keyboard data file supporting typewriter form of keyboard layout. Details concerning the choice of a proper keyboard layout for a particular country (and of a proper code page too) are shown in appendix A.02-2. An implementation example of Microsoft's proprietary national adaptation is shown in articles 9.01-01 and 9.01-02.

Though Microsoft's keyboard files and codepages include national data for the most part of the globe, they are no more supported by Microsoft and are not opened for updating. Because of this at least 5 other keyboard drivers have been developed for DOS up to now. Only one of those alternatives is described in this book – the KEYRUS.COM driver

(5.02-05). Contrary to most other keyboard drivers, KEYRUS.COM is an open project, supplied with means for creating new keyboard layouts and for making corrections in co-depages. Unfortunately, format of Microsoft's keyboard data files and codepages can't be accepted by KEYRUS.COM. Examples of national adaptation with KEYRUS.COM are shown in articles 9.04-01 and 9.09-01.

Chapter 2 Command line

Command line in MS-DOS begins with a machine-generated prompt and then is implied to be filled with symbols and words, which altogether must be suitable for machine interpretation according to certain syntax conventions. The final ENTER keystroke initializes interpretation of command line by a resident program – the command interpreter. Implementation of syntax conventions by different interpreters is not just the same.

MS-DOS7 provides 3 interpreters: IO.SYS, COMMAND.COM, and DEBUG.EXE. Each interpreter accepts its own set of commands, described in chapter 4 for IO.SYS, in chapter 3 for COMMAND.COM and in part 6.05 for DEBUG.EXE. Since the loader IO.SYS deals with command lines in configuration files only, the user first encounters command line, presented by COMMAND.COM. The latter is often called resident interpreter, because it is permanently loaded into computer's memory and therefore is always ready to perform commands entered from keyboard.

By means of input redirection (see 2.04-02 for details) one can force the interpreter to accept commands not from keyboard, but from command files. Each line in command files is in fact a separate command line. Sending command files via input redirection is the only way to automate execution of command sequences by the DEBUG.EXE interpreter (examples – in 9.02).

The COMMAND.COM interpreter too can accept commands via input redirection, but it is not the best way to execute command sequences for COMMAND.COM, because it is able to handle a special sort of command files – batch files – without redirection. From batch files the COMMAND.COM interpreter accepts a number of supplementary commands, which can't be executed from command line or from ordinary command files. These supplementary commands include substitution of dummy parameters and names of variables with their values (2.03-03), search for labels and some other commands (3.02, 3.14, 3.21, 3.27). Here and forth in this book the term "batch files" is applied only to this special sort of command files. Examples of batch files are shown in part 9.03. Configuration file AUTOEXEC.BAT (9.01-02, 9.04-02, 9.09-02) also represents an example of a typical batch file.

This chapter describes the most essential conventions, which define command line composition both in separate command lines and in command files. These conventions are common to some extent for all the three interpreters in MS-DOS7. Endemic features of interpreters are presented too. If not specified otherwise, interpretation of commands by the main interpreter – COMMAND.COM – is implied.

2.01 Named objects and their names

Each command line addresses one or more objects. An object may be, for example, a separate utility, which is to perform the desired action, or an internal command of command interpreter. The data in command line must be ample enough to identify exactly the addressed object(s). For this purpose inside any common directory identical names of objects are not allowed: all files and subdirectories must be given different names. In order to identify any object outside any particular directory the path to this object is taken into account. The path may be specified explicitly or set by default.

The main object's name together with its optional path should be specified the first in command line and may be followed by other items, including name(s) of other object(s), parameter(s), reference(s), syntax mark(s), etc. Each line of command files, presented in part 9.03, can be taken as example of command line.

There are objects, however, – internal commands, ports and some others – which can't be defined by path specification. The names of such objects must be unique, reserved words, which shouldn't be assigned to any other object. Therefore the first theme to consider is which names may be assigned to objects by the user, and which can't.

2.01-01 Reserved words

Reserved words represent names of internal commands, specific for each interpreter, and names of devices, claimed as existing in a particular computer. Names of these objects can't be altered or assigned to other objects. Nevertheless the reserved words should be known in order to prevent attempts to assign such names to other objects, which can be named by the user.

Internal commands are those performed by a particular interpreter itself. All those names of internal commands, described in chapter 3, are regarded as reserved words as long as command line is interpreted by COMMAND.COM. For example, you can't assign the name PROMPT to a file, since COMMAND.COM, having encountered this name in command line, "understands" it as its own internal command and has to do nothing but perform this command. Other command interpreters similarly don't allow to assign names of their internal commands to other objects.

Reserved names of devices in a PC define sources for obtaining data or targets for sending data. Most known device names are:

AUX – first serial port
COM1 – first serial port (equivalent to reserved name AUX)
COM2 – second serial port
CON – console, i.e. keyboard for input and display for output
LPT1 – first parallel port
NUL – virtual "nowhere" output port

PRN – first parallel port (equivalent to reserved name LPT1)

Besides these some other device names are regarded as reserved: CLOCK$, COM3, COM4, CONFIG$, LPT2, LPT3. These words are reserved by device drivers, integrated into DOS's core, which are loaded always, even if corresponding device is not physically present. The entire list of devices, which are claimed present in your PC, can be displayed by the MEM.EXE utility (6.17), if it is launched with /D parameter. This utility displays a table, and all registered names are shown in its 4-th column. Reserved words are shown shifted by 3 spaces to the right against other names.

Several installable software drivers also can be identified by name and reserve words for this purpose. For example, SETVER.EXE driver (5.01-02) reserves word SETVERXX, HIMEM.SYS driver (5.04-01) reserves word XMSXXXX0, EMM386.EXE driver (5.04-02) reserves word EMMXXXX0. Moreover, when you specify an arbitrary identifier for a CD-ROM drive, for example, MSCD001 (see 5.09-01, 5.09-02, 5.09-03), this name becomes registered by DOS as a reserved word. If later you'll try to assign this name to a file, DOS will reject your attempt.

2.01-02 Names and suffixes

Naming and renaming operations most often are applied to directories (3.19, 6.20) and to files (3.24, 3.25), both ordinary and executable ones. Names in DOS consist of no more than 8 characters and may be appended with an extension (suffix) of no more than 3 characters long. Suffix is separated from the name with a dot. When suffix is assigned, but is not required for a particular operation and is omitted, then the dot separator must be omitted too.

Both names and suffixes may be composed of letters, digits and several signs, which have no dedicated functional mission in parsing of command lines: number sign (#), dollar sign ($), ampersand (&), minus (–), exclamation mark (!), underscore (_) and a few others (2.04-06). Upper case and lower case letters are regarded equivalent in almost all operations, except letters in strings to be compared with commands IF (3.15-02), SEARCH (6.05-16) and FIND (6.14).

Since dot (.) is used to separate a name from suffix, it is not allowed to be employed as an ordinary character. Other prohibited signs are comma (,), colon (:), semicolon (;), equality sign (=), question mark (?), plus (+), left arrow (<), right arrow (>), asterisk (*), pipe (|), slash (/), backslash (\ ), and double quotes ("). National language symbols also can't be employed, unless restrictions on their usage are taken off by the COUNTRY command (4.05, 5.02-01, A.02-5).

Directory names usually have no suffix. Suffixes in names of files are used to indicate file's type or origin. Three suffixes – BAT, COM and EXE – have special status, since are recognized by the COMMAND.COM interpreter as belonging to executable files. When a

filename with either of these suffixes is specified the first in command line, COMMAND.COM automatically tries to execute the file's code. If there is something else except valid executable code, the PC most probably would get hanged. These suffixes shouldn't be assigned to files, which are not definitely executable and valid.

Assignment of other suffixes is not so critical, but nevertheless should be submitted to common restrictions. File managers are able to link files with certain extensions to devoted applications, for example, files with suffix BAS may be automatically sent for execution to BASIC language interpreter. It's convenient to have every file automatically directed to appropriate program (examples – in 6.25-03, 6.25-04). Suffixes also may be helpful for visual recognition of essential file's classes. The list below shows some suffix associations, commonly used in DOS environment.

BAK – archive or old version of a file
BAT – batch file, interpretable by COMMAND.COM
BIN – executable file, which requires fixed placement
BMP – raster image file (Bit-Map-Picture)
CAB – compressed file for software distribution
COM – executable file without a header
CPI – library file with font data for DOS
DAT – files with non-textual data of various kind
DOC – textual file, produced by the WORD editor
DLL – Dynamically Linked Library of executable codes
EXE – executable file with control data in header
EXT – specifications of functional extensions
GIF – Graphic Image File
HTM – file written in HyperText Markup language
INI – file with initialization data
JPG – image file, compressed according to JPEG specification
RAR – compressed archive file, packed by RAR.EXE
RTF – textual file of Rich Text Format
SCR – file with command lines, interpretable by DEBUG.EXE
SYS – system file, either textual or installable
TMP – temporary file
TXT – non-formatted textual file
ZIP – compressed archive file, packed by PKZIP.EXE

Associations for a lot of other suffixes can be found in internet site http://www.openwith.org/ .

2.01-03 Filemasks and wildcards

Question mark (?) and asterisk (*) are known as substitution symbols (or wildcards), which can't be employed in names and extensions, but can be used to replace characters in specifications of names and suffixes in command line(s). Name specification with wildcard(s) is a filemask – a means to address several files at once.

While parsing a command line with a filemask, COMMAND.COM may invoke wildcard expanding function, which searches for filenames, satisfying to the encountered filemask. If such file is found, its name is substituted for the filemask, and then command line with this name is executed. If more than one such file is found, similar procedures are performed sequentially with each of the found filenames.

Whether words with wildcards will be interpreted – it depends on the command to be executed: some commands invoke a call for the mentioned wildcard expanding function, some commands don't. Generally wildcards are not ignored, but there are some exceptions:

- wildcards are not expanded by internal commands ECHO, SET, TYPE and IF with equality comparison function (see chapter 3).
- wildcards in parameters of a batch file are not expanded (it doesn't matter whether the CALL command is used or not);
- input redirection notation (2.04-02) doesn't expand wildcards;
- the FOR command (3.13) expands wildcards inside brackets only, but outside brackets wildcards are transferred "as they are" to the command to be executed inside the FOR cycle, and further fate of wildcards depends on that command.

Wildcard (?) means a symbol, which induces comparison procedure(s) to give positive response for any single letter or digit. When the (?) wildcard in a filename or in a suffix is not followed by any explicitly specified letter or digit, comparison procedures also give positive response to absence of any symbol in that place. For example, in a command

```
DEL readme.??
```

the object will match all files in the current directory having name "readme" and not more than two-character suffix: readme.ru, readme.en, readme.f, and so on. Hence the DEL command (3.09) will delete all such files from the current directory.

Asterisk (*) means a symbol, which induces comparison procedure(s) to give positive response for any number of any following letters or digits up to the end of the word, which may be marked with either a dot or a space. For example, asterisks in command

```
DEL C:\TEMP\*.*
```

match any filename with any suffix, so this DEL command will be applied to all the files inside the specified directory.

Numerous examples of wildcard usage are shown in 2.02-03, 9.09-02 and in several other batch files in chapter 9.

Note 1: whether the attributes will be taken into consideration during a search for files, satisfying to a filemask, – it depends on the command to be executed. For example, wildcard expansion by the ATTRIB command (6.01) includes all files without exceptions.

Note 2: interpreters IO.SYS and DEBUG.EXE don't accept wildcards. While CONFIG.SYS file is interpreted by IO.SYS, the question mark is treated not as a wildcard, but as a prompt (see 4.06, 4.07, 4.15, 4.16, 4.25).

2.02 Paths

Exact placement of each file on a disk is specified in a certain directory. Numerous directories are usually arranged in a hierarchical structure: each directory of higher rank may contain data not only about files, but also about placement of several lower rank directories (subdirectories). In order to access any file you have to point out the disk and the path – that is a directory and the whole chain of subdirectories, leading to the one, which contains exact placement data for this particular file.

DOS gives you an opportunity (2.04-01, 3.03) to specify any one particular path as the default path. This path will be stored in internal DOS's data table (A.03-3). Each time you enter a command line without path specification, DOS takes into account the stored default path. The disk and the final directory (subdirectory), specified in the default path, are known as the current disk and the current directory. DOS's prompt (3.22) is usually adjusted to show the current path.

2.02-01 Typical path structure

By default the command interpreter searches for the addressed object in the current directory. If the object must be found in any other directory, object's name should be preceded by a path, for example

```
C:\DOS\MS7\Edit.com
```

where:

Edit.com – is the addressed executable file;
C:\DOS\MS7\ – is an example of a path to this file.

The shown path specification directs the search process: first it should be switched to the root of disk C:, there the DOS directory should be entered, then – its MS7 subdirectory, and at last in the latter directory the specified executable file should be searched for. If you happen to have other directory structure, you'll have to specify other names, but the principle remains the same: first the disk's letter-name, followed by a colon,

then a chain of directory names, separated with backslashes, and finally a name of the addressed object.

Specifications terminated with a backslash are considered in MS-DOS as having no target object and hence not complete. Such paths are either ignored (see 2.04-01) or regarded as not valid, except one special case – the path, reduced to a single backslash. Single backslash after a disk's letter-name means a path to the root directory of the specified disk. For example,

```
A:\
```

means a path to the root directory of disk A.

Paths without disk's letter-name (for example: \DOS\MS7\Edit.com) are reckoned from the root directory of the current disk, whichever disk is current at the moment. This is important for writing media-independent batch files (9.01-01, 9.04-01, 9.09-01). Single backslash is interpreted as a path to the unnamed root directory of the current disk. Command to change the current directory (3.02) with the following backslash

```
CD \
```

performs a jump to the root directory of the current disk, whichever it is.

When there is no both disk and root specification at the beginning of the path, it is reckoned taking the current directory as the start point. For example, the path

```
DOS\VC4\Vc.com
```

implies that the first subdirectory DOS exists inside the current directory, otherwise an error message will be displayed.

2.02-02 The PATH variable

In order to make command line usage easier DOS provides one more mechanism of path specification – via the PATH environmental variable. This mechanism comes into action when 4 conditions are met:

— command is addressed to the COMMAND.COM interpreter;
— PATH variable is present yet in the environment;
— path is not specified in command line;
— the first addressed object isn't found in the current directory.

If all these conditions are met, then DOS will search for the first addressed object along all the paths, which are implied to constitute the value of the PATH variable. The PATH variable should be set beforehand with internal PATH command (3.20). Examples of paths specification with PATH command are shown in 9.01-02, 9.04-02, 9.09-02.

Owing to the PATH variable you can call utilities for execution as easily as though these were always present in any directory you may choose. This is convenient, but only

until it happens to encounter in the current directory a synonymous version-specific utility, belonging to other version of DOS. Then the latter utility will be found first and will fail, leaving an error message. Different approaches to the problem of avoiding such conflicts are discussed in article 5.01-02 and in introduction article to chapter 6. This problem also may be solved by specifying full paths in command files or in configuration files. Examples of such solutions are given in part 9.03.

2.02-03 Dot(s) in path specifications

A dot (.) in path specifications is interpreted as an alias for the current directory. Note, for example, the trailing dot, which replaces the target path in the following copy command:

```
Copy /B A:\MyDir\*.* .
```

Sometimes batch files must be written so as to do their job in any current directory (which may be not known beforehand), and then the dot alias is the only allowable replacement for required specification of target path to the current directory. Addressing to the current directory with a dot is also needed when any other (not current) directory will be implied otherwise.

Dot as the first symbol of a path statement means that the path should be reckoned taking the current directory as the start point:

```
.\VC4\Vc.com
```

Such path is equivalent to the one without preceding backslash (2.02-01), but nevertheless this form of path may be practical in batch files, because MS-DOS provides no other means to get rid of preceding backslash. One more reason to use commands with preceding dot alias and a backslash is to prevent search for synonymous utilities throughout all the paths, enlisted in the PATH environmental variable, when execution of a synonymous utility may inflict unwanted consequences.

Some utilities return directory specifications with final backslash, for example:

```
C:\DOS\MS7\
```

Such paths are regarded by MS-DOS as invalid, and MS-DOS doesn't provide means to get rid of the final backslash. Appending a dot to such path solves the problem:

```
C:\DOS\MS7\.
```

- this directory specification is regarded valid.

Double dot, or dot-dot (..) may be used in any place within path statements just like the single dot, but dot-dot is an alias for the parent directory. If, for example, you are given specification C:\DOS\MS7\ and want to address its parent directory, then the

```
C:\DOS\MS7\..
```

specification is exactly equivalent to C:\DOS.

While parsing a path, containing a double dot alias, DOS simply throws out the preceding element of path's chain (the \MS7\ in the last example). DOS doesn't check whether the thrown element exists, whether it represents a file or a directory. This gives an opportunity to address a new file in a directory, unknown beforehand, on a basis of a path to another file, obtained during execution. Examples of such addressing are shown in article 6.25-03.

Double dot without any preceding path is interpreted as an alias to the parent directory with respect to the current one, for example, in a command to change current directory (3.03):

```
Cd ..
```

In order to climb two levels up the tree of directories structure, you have to combine double dot twice:

```
Cd ..\..
```

Some even more complex double dot combinations may be used to navigate and to explore directory trees.

2.03 Tips on command line parsing syntax

2.03-01 Separation symbols

Words in command line are separated with separation symbols: space ( ), comma (,), equality sign (=) and semicolon (;). Though space is the one most universally used, either of these is ignored at the beginning of command line and will act as separation symbol in parsing operations, including parsing of an object list within the FOR command (3.13). Because of the same reason separation symbols can't be transferred from parameters of command line into internal dummy parameters of a batch file (2.03-03).

There are some exceptions, though. Commands SET and IF use equality sign (=) in a special way and don't allow to use it for separation. When symbols comma, semicolon or equality sign precede to ECHO command, then displayed part the line will begin just after the separation symbol, including possible preceding spaces and the word ECHO itself. Semicolon (;) is used as special separator in PATH command (see 3.20).

The IO.SYS interpreter deals with semicolon otherwise. Being placed first in a line within Config.sys or Msdos.sys file (5.01-01), semicolon is interpreted as a command to jump to the next line, ignoring the rest part of current line (usually containing commentaries). The DEBUG.EXE interpreter in assembler mode deals with semicolon in a

similar way: it ignores the rest part of a line after semicolon and thus enables to supplement DEBUG's command files with remarks (7.01-05).

2.03-02 A slash

A slash (/) in MS-DOS command line specifications is a sign to interpret the following letter (or word) as a parameter. For example, in command

```
DEL C:\TEMP\*.* /P
```

(see 3.07) slash (/) forces to interpret the letter P as parameter, inducing a prompt on whether each file in the specified directory should be deleted. Exact place and form of the parameter(s) are specific and must conform to particular command's specifications.

Sometimes a slash (/) is used within the FOR command as a functional separator, causing conversion of letters in the following word to the upper case (see 3.13 for details).

2.03-03 Percent sign

Percent sign (%) in batch files means substitution of the name of a dummy parameter or of a variable with the value of the same parameter or variable. These substitutions are performed before execution of the specified command(s) and redirection(s).

Dummy parameters are named with digits from 0 to 9. Value of the 0-th dummy parameter is always the name of the batch file itself; other dummy parameters take their values from the items, specified after the name of batch file in the command line, from which the batch has been started. So %3 ,for example, will be replaced with the third item, following the name of batch file in command line. If several dummy parameters are specified without a space in between, then after substitution their values will be concatenated. Examples of assigning values to dummy parameters are shown in 2.03-04 and in 9.03-01.

If total number of words, following the name of a batch file in command line, is less than 3, then the designation %3 of a dummy parameter will be replaced with nothing and disappears without any error message. If total number of words, following the name of a batch file, is greater than 9, then the values of the rest can be accessed after applying a shift to dummy parameter numeration (3.27). There is one exception, though: the Autoexec.bat file (for example, 9.01-02), automatically executed during DOS loading procedure, has no dummy parameters at all, so the CALL %0 command (3.02) doesn't induce its recursive execution.

Names of variables must be a single word with a letter (not a digit!) as their first character. Values of variables either are assigned by special command SET (3.23) or are inherited from parent environment, belonging to the program, which has launched execution of the current program (see 6.04). For performing substitution the name of the

variable in command line must be surrounded with percent signs on both sides (%VAR%, for example). Numerous other examples of command lines with environmental variables are shown in part 9.03.

Note 1: if percent sign is to be transferred without substitution, you have to specify it twice (%%). During interpretation of command line the doubled percent sign doesn't induce replacement, it is simply transformed into a single percent sign.

Note 2: the FOR command (3.13) employs its own local variable; its name outside batch files must be specified with only one (preceding) percent sign. Being used within a batch file, name of that variable needs two preceding percent signs, for example, %%A, because substitution shouldn't be performed in advance to the FOR command (see 3.13 for details).

Note 3: other interpreters (IO.SYS and DEBUG.EXE) ignore percent sign and don't replace variables and dummy parameters with their values.

.

2.03-04 Double quote sign (")

Double quote (") disables interpreter's function of parsing the command line until the next double quote is met in the same line. Thus any group of words between opening and closing double quotes (possibly including separation symbols, redirections, etc) will be interpreted as one item. Enclosing double quotes themselves are considered belonging to the enclosed item. For example, execution of a line

```
C:\>Batch.bat 1 " 2 3 " 4 ""
```

creates a new set of dummy parameters, in which the value of the first parameter is single digit 1, the value of the second is a string " 2 3 ", the value of the third parameter is digit 4, and the value of the last fourth parameter is an empty pair of double quotes. This set of dummy parameters will exist until execution of Batch.bat terminates. Enclosing a group of words in double quotes is a way to include this group of words as a whole (together with any symbols inside) in a value of one dummy parameter. This trick is used, in particular, to preserve integrity of long names in DOS.

An empty pair of double quotes ("") is regarded as a special void symbol, enabling to preserve parsing sequence unchanged. Most internal commands in MS-DOS7 (except ECHO, IF and SET) ignore empty pair of double quotes as a separate symbol, but accept the results of parsing affected by double quotes. For example, command

```
C:\>cd ""
```

is executed just as though there were no following symbols at all. When a parameter is enclosed in double quotes, command is executed as though there were no double quotes:

```
C:\>cd "C:\dos"
```

Existence of the closing double quote is not checked during parsing of command line, except parsing parameters for FIND (6.14) and FOR (3.13) commands. Both FIND and FOR commands perform parsing in a slightly different way: any group of words enclosed in double quotes is still regarded as one item, but double quotes are not considered belonging to this item. Therefore empty paired double quotes ("") may be used in FIND command for counting total number of lines in textual files. For the same reason the FOR command enables to get rid of enclosing double quotes, when these are no more needed.

2.03-05 Square brackets

Square brackets [ ] are used as a special sign in files, which are to be executed by DEBUG.EXE and by IO.SYS. Data in square brackets are interpreted by DEBUG.EXE as references to operands (see introduction article to chapter 7 for details).

In configuration files MSDOS.SYS and CONFIG.SYS, which are to be executed by the IO.SYS loader, the words enclosed in square brackets are interpreted otherwise: as headers, marking beginning of a separate configuration block and at the same time as signs, enclosing name of this configuration block. There are two reserved words, which denote special configuration blocks in CONFIG.SYS file: [menu] and [common]. The [menu] block represents multiconfiguration menu; if it exists, it must be placed the first in CONFIG.SYS file. The menu block is eminent because of a special subset of allowed commands: the MENUCOLOR (4.19), MENUDEFAULT (4.20), MENUITEM (4.21) and SUBMENU (4.29) commands may be used only in blocks, announced as menu or submenu. All other configuration commands, described in chapter 4, except the NUMLOCK command (4.23), can't be used in menu and in submenu blocks.

Commands to be executed in all configurations are grouped in one or more blocks with the same name [common]. Outside block headers the names of configuration blocks are referenced without square brackets (4.14). Examples of configuration file CONFIG.SYS with blocks [menu], [common] and some others are shown in 9.04-01 and in 9.09-01.

2.04 Syntactic marks with command mission

2.04-01 Colon

Interpretation of a colon (:) depends on its place in command line. Being used as the first character in a line (in batch files only), colon (:) forces to interpret the nearest following word as a label, marking a target address point for a jump. There may be more than one word in this line, but all the rest words and symbols will be ignored. Double colon

(::) at the beginning of a line in a batch file is sometimes used in order to disable all operations, specified in this line, including redirection operations (2.04-02 – 2.04-05).

A colon (:) in the second place in command line forces to interpret the preceding letter as a letter-name of a disk. If the following part of command line is empty, or is reduced to a single backslash, or is enclosed in backslashes, then the whole line is interpreted not as a path, but as a command to change current disk (make specified disk the default one). For example, in order to make disk A: the current default disk you may type either

```
A:
```

or

```
A:\
```

or

```
A:\WINDOWS\
```

and then press the ENTER key. A change of the current disk doesn't alter the default directory on this disk. If the current directory on the target disk is, for example, A:\DOS, it will preserve its status after default disk change in each of the examples above. In such commands any path, enclosed in backslashes, is not checked and even may not exist. In fact any full address, appended with a backslash, will fit as current disk change command.

2.04-02 Left Arrow

Left Arrow (<) denotes an operation of input redirection, prepared by COMMAND.COM interpreter for execution of the program, specified in command line to the left of Left Arrow sign. By default the standard input channel (STDIN) is linked with console (CON) and accepts input from keyboard. In fact Left Arrow is a command to link STDIN (handle 0000h) with the data source, specified to the right of input redirection sign. When the utility, specified to the left of input redirection sign, will ask DOS for data input via the STDIN channel, it will get data from this source. Of course, this works only if the utility asks for input via the STDIN channel. For example, command

```
MORE < C:\DOS\Filename.txt
```

supplies the filter utility MORE.COM (6.19) with data read from specified textual file in C:\DOS directory. If directory specification is omitted, then current directory is implied. In any case the file to be read will not be searched for along the paths, enlisted in the PATH variable. Filemasks instead of the source filename are not allowed, wildcards (2.01-03) are not expanded.

Besides files, ports (LPT1, LPT2, COM1, COM2, COM3, COM4) may be used as sources of input data. If default links are disrupted by the CTTY NUL command (3.07), then for performing input from keyboard you have to use input redirection with explicit specification of the console (CON) as the data source (example – in 3.07).

In any case of input redirection you must be sure, that specified source will be ready to provide the required data. Waiting for data input from an empty, idle or defective source most probably will cause hanging.

Note 1: redirections are arranged by data substitution in JFT table (note 3 to A.07-1). Redirections, prepared by COMMAND.COM for execution of a program, may be cancelled by the program itself (example – in 9.07-02).

Note 2: other command interpreters (DEBUG.EXE and IO.SYS) ignore left arrow sign as well as other redirection signs (2.04-02 – 2.04-05). However, the redirections, prepared by COMMAND.COM, are accepted by DEBUG.EXE (examples – in 9.02).

2.04-03 Right Arrow

Right Arrow (>) denotes an operation of output redirection, prepared by COMMAND.COM interpreter for execution of the program, specified in command line to the left of the Right Arrow sign. By default the standard output channel (STDOUT, handle 0001h) is directed to the console device (CON) - that is to the display's screen. Output redirection forces to direct STDOUT to another target - the one specified to the right of Right Arrow sign. For example, the DEL /? command (3.09) normally presents on-line help to display, but when it is followed by output redirection sign

```
DEL /? > Filename.txt
```

its output wouldn't reach the screen, it will be written into the specified file. A new file with the specified name (Filename.txt) will be created automatically in order to write there the redirected output. If a synonymous file exists yet, it will be overwritten without prompt, and its former contents will be lost.

Besides files, allowable targets for redirection may be ports (LPT1, LPT2, COM1, COM2, COM3, COM4), printer PRN (which is usually equivalent to LPT1), display (CON) and special quasi-device NUL, which acts just as a "black hole": any output becomes lost there for ever (example – in 3.21). This is often used in order to get rid of unwanted messages.

If the default link between the STDOUT channel and the display device is disrupted by the CTTY NUL command (3.07), then sending output to display is still possible, but it needs explicit output redirection to CON device (examples – in 9.03-02).

Output redirection can intercept only those data, which are sent via DOS's normal STDOUT channel. Data sent via BIOS' interrupts (8.01-17, 8.01-21, 8.01-33), via DOS' INT 29 (8.02-88) and via STDERR channel for error messages (handle 0002h) can't be affected by STDOUT redirection.

If you combine input and output redirections in one line, then the main executable command must be first followed by input redirection with full source specification. Output redirection should be specified afterwards. Examples of combined redirections are shown in 6.14, 6.25-03 and 9.03-02.

All symbols of redirection (2.04-02 - 2.04-05) are assigned higher priority, than common operations, except labels (2.04-01) and double quotes (2.03-04). For example, after string input commands (ECHO, SET) all redirection symbols will not be processed as a member of the string, but rather will cause redirection. Because of the same reason redirection ignores conditions, set by IF command, and the only way to execute redirection conditionally is to bypass its line with conditional "IF ... GOTO" jump (3.15, 3.14).

Redirection will be executed even if the main operation in the same line is invalid, disabled or provides no output at all. Empty output redirection after the REM command (3.24) is often used to create a file of zero length.

Output of a batch file as a whole may be redirected only via loading a separate module of command interpreter COMMAND.COM with /C parameter (6.04) in order to execute this batch file (examples - in 3.22, 9.01-03). Without launching a new interpreter's module the output of a batch file may be redirected in a line-by-line manner from inside of this batch file, but not as a whole.

Output redirection should be used with caution, because together with anticipated output it may affect some warnings or invitations for certain action(s) sent to the user. For example, the DIR /P command stops its output after each screenful and sends a message via STDOUT, reminding that output will be resumed after any keystroke. When such messages and warnings are redirected, the screen remains empty, and PC seems having got hanged.

2.04-04 Double Right Arrow

Double Right Arrow (>>), just as single Right Arrow (2.04-03), is also a sign of output redirection, but its action is different, when target file for redirection exists yet. Rather than overwrite the target file anew, redirection with Double Right Arrow appends new data to former contents of the existing target file. All other peculiarities of output redirection with the Right Arrow (2.04-03) are applicable to Double Right Arrow as well.

2.04-05 Vertical bar (or "pipe")

Vertical bar or "pipe" separator (|) is a sign of intermediate redirection, i.e. data transfer from one utility (or command) to another. Especially for this purpose COMMAND.COM interpreter creates a temporary file. Utility placed to the left of "pipe" is executed first, and its output via STDOUT channel (via handle 0001) is written into this temporary file. Then control is transferred to the utility (or command) placed to the right of

"pipe". When this utility issues a request for reception of data via STDIN channel (via handle 0000), these data are taken from the prepared temporary file. When execution of the latter utility terminates, temporary file is automatically deleted.

For example, following command sequence enables to avoid query on whether the user really wants to delete all the files in specified directory:

```
ECHO Y | DEL C:\TEMP\*.*
```

First an empty temporary file is created. Then the ECHO command is performed, and its output message (a single letter Y) is written into that temporary file. Then the DEL command is executed. Having found the *.* filemask, it issues a request for the user's permission and begins to wait for a reply from STDIN channel. But since its STDIN channel is redirected, execution will not be suspended: the reply – letter Y – will be automatically read at once from the prepared temporary file.

There may be more than two commands linked by "pipes" in one command line. Examples of such command lines are shown in 3.08 and in 3.28.

If command, specified to the right of "pipe" sign, doesn't need the contents of temporary file, then the command to the left of the "pipe" sign is not obliged to send a message into STDOUT channel. Hence the "pipe" sign can potentially act as a separator, enabling to specify several commands in one line. However, such usage of "pipes" can't be recommended: the FOR cycle (3.13) can do just the same much faster and without access to a writable disk for creation of temporary file(s).

Note 1: the "pipe" separator implies creation of a temporary file either in current directory or in the directory, pointed out by environmental variable TEMP. However, both these attempts may fail when DOS is loaded from a non-writable optical disk or from any write-protected media. In such cases an error message informs that the intermediate redirection can't be performed. Then command specified to the right of "pipe" sign will not be executed too.

2.04-06 The "at" (@) sign

Being used as the first symbol in batch file's command line, the "at" sign (@) is interpreted as a command to prevent display of this line on the screen. Therefore almost each batch file begins with the "at" sign, followed by the ECHO OFF command. Sometimes such action is necessary not only in the first line (see 3.13, 6.25-02, 6.25-03 for examples).

Note 1: DOS doesn't restrict usage of the "at" sign in filenames, but wrong interpretation of filenames with the "at" sign in a place of the first letter may lead to harmful consequences.

Chapter 3 Internal commands

Internal commands are those executed by command interpreter itself. Contrary to other utilities, which are to be searched for and read from their media, the interpreter is permanently present in memory, and therefore internal commands are executed much faster. Since internal commands are not to be searched for, the path for internal commands shouldn't be specified. One more common property of all internal commands is that after execution they don't leave errorlevel code.

This chapter presents internal commands of Microsoft's proprietary command interpreter COMMAND.COM (file size 93812 bytes, file date 12.06.1996). Localized versions of this interpreter usually are slightly larger and have somewhat later date, but nevertheless execute the same set of internal commands. Among the described internal commands are those intended for batch files only (3.02, 3.14, 3.21, 3.27).

The COMMAND.COM interpreter provides a short help for its internal commands. In order to display help text you have to type the command's name, followed by a space and the /? parameter, and then press the ENTER key. But the provided short help is often found too short. This chapter presents a lot of clarifications, which will help you to avoid common mistakes and to make internal commands usage much more effective.

3.01 BREAK — disk access intercept control

The BREAK command is similar to synonymous configuration command (see 4.02 for details). They both affect the same binary flag, which controls checks of BREAK and CTRL-C keystrokes during disk access operations. This binary flag doesn't lose its state when current resident module of command interpreter finishes its job, and local environment becomes lost. Contrary to the IO.SYS loader (4.02), the COMMAND.COM interpreter shows current state of the mentioned binary flag in response to the BREAK command entered without parameters.

3.02 CALL – batch file execution with return

CALL is a command to execute one batch file (the secondary batch) from another batch file (the primary batch). Batch files are non-formatted textual command files, from which the COMMAND.COM interpreter accepts an extended set of commands. The CALL command is just one of those intended to be entered not from keyboard, but only from lines of batch files.

Each line in a batch file may include a name of an internal command, or a name of an external utility, or a name of another batch file as well. When command interpreter encounters a name of external utility, it transfers control to this utility, but takes control back after the utility finishes its job, and proceeds to the next line in the same batch file. However, batch files don't behave like ordinary utilities. Having found a name of a batch file (the secondary batch) in a line of another batch file (the primary batch), the COMMAND.COM interpreter begins execution of the secondary batch and doesn't return to the primary one. In order to enable a return to execution of the primary batch file, the secondary one should be launched with CALL command, for example:

```
CALL C:\DOS\VC4\Help.bat J 96
```

where:

C:\DOS\VC4\ – an example of a path to HELP.BAT file. If path is omitted, the file will be searched for inside current directory and then according to all path(s), specified by PATH variable.

Help.bat – an example of a name for the secondary batch file.

J 96 – specific parameters to be transferred to HELP.BAT file (other batch files may need other parameters or may not need them at all).

In fact the CALL command prevents closure of the primary batch file, keeps stored segment of the primary batch file, its dummy parameters and its file pointer position (as a target to return), lets command interpreter to execute another (secondary) batch file, specified after the CALL command in the same line, and then restores access to requisites of the primary batch file, thus enabling a return back to execution of the next operation in the primary batch file.

Note 1: the secondary batch file inherits not a copy of primary environment, but an access to the same primary environment. The values of variables, which have been assigned during secondary batch file execution, will become accessible after resumption of primary batch file execution.

Note 2: execution of batch files with the CALL command is allowed to be nested more than twice.

Note 3: the CALL command enables to perform recursive calls, that is a batch file may be called from a line inside the same batch file. But it's the user's responsibility to prevent uncontrolled deepening of recursion nesting level.

Note 4: if the secondary batch file is not found in the specified directory or in the current directory and throughout all the paths, defined by the PATH variable (2.02-02), then execution will be transferred to the next line of the primary batch file without any error message.

Note 5: redirection signs (2.04-02 - 2.04-05) are not allowed in lines with the CALL command.

Note 6: the described CALL command, intended for COMMAND.COM interpreter, shouldn't be mixed up with synonymous assembler command (7.03-08) which is to be interpreted by DEBUG.EXE debugger.

3.03 CD – change directory

When disk and path are not specified in command line, then MS-DOS implies presence of the addressed object in default (current) directory of the default (current) disk. Default disk assignment has been described in article 2.04-01. Default directory assignment is performed by CD command.

The CD command enables to change current directory on any disk (but not the current disk itself) according to a specified path. For example, command

```
CD C:\DOS
```

will set current directory \DOS, if the current (default) disk is disk C:. CD command may be addressed to a disk, which is not the current one, but then specified path is taken into account as a preset, which will become the current (default) directory later, when the disk, specified in CD command, will be given status of the current disk (2.04-01).

The path in CD command may be expressed in any of its allowed forms (2.02-01). The final word in the path must be either a name of target directory or any combination of alias signs (2.02-03, 2.03-02). Such combinations enable to perform transition into the root directory (CD \ ), into parent directory (CD ..), to climb two levels up along the directory structure (CD ..\..) and so on.

CD command without path specification, for example

```
CD C:
```

shows a path to the current directory on the specified disk. When disk is not specified too, then the current disk is implied.

Note 1: CHDIR is another valid name for the same CD command.
Note 2: the directory, which is assigned current by CD command, can't be deleted with RD command (3.23), even if this directory doesn't belong to the current disk.
Note 3: preset paths to current directories for all logical disks are stored in CDS table structure (A.03-3). By default it is filled with paths to root directories.

3.04 CHCP – change code page

Codepage is an array of characters and symbols used to display messages on the screen. When the CHCP command is used without following codepage number, it shows the number of the codepage, which is currently active. Codepage changing function is usually disabled,

– unless 2 or more memory buffers for codepages have been prepared by DISPLAY.SYS driver (5.02-02),
– unless these buffers are filled yet with different codepages by the MODE.COM CON CP PREP command (6.18-03),
– and unless NLSFUNC.EXE driver (5.02-03) is loaded beforehand.

Common practice is to load only one national codepage, because each national codepage contains not only national character set, but american english character set too. Switching between these character sets doesn't imply codepage change: it is performed inside any single national codepage.

Loading several national codepages becomes needed when there is no one codepage, containing character sets for the languages, which you intend to use. For example, if you prepared norwegian codepage 865 and russian codepage 866, then commands CHCP 865 and CHCP 866 will perform switching between these codepages.

Note 1: codepages may be changed with MODE.COM CON CP SEL command (6.18-03), which doesn't need to have NLSFUNC.EXE driver loaded.
Note 2: the CHCP command can't cope with national codepages, loaded by non-Microsoft's drivers, for example by KEYRUS.COM (5.02-05).
Note 3: if you use national version of an important utility or of a TSR shell (for example, Norton Commander with russian notation), then changing of the codepage will make this national notation totally non-readable! Only american english notation (characters 32 – 127, common for all codepages) will not be affected by codepage change.

3.05 CLS – clear screen

The CLS command erases all contents of the current screen page in video card memory and also sets colors to default values, thus providing for white text display on a black background.

3.06 COPY – make copy of a file

The COPY command is used to copy one file or several files at once. It also enables to concatenate files and to combine copying with renaming. Here is an example of copying one file into another location:

```
COPY /A C:\DOS\MS7\Trial.txt A:\DOS /V /Y
```

where:

/A – parameter, indicating that the specified source file must be copied no further than the first end-of-file mark (1Ah). For copying only one file into a target file, the opposite /B parameter is taken instead by default. It forces to copy a file as a whole, ignoring end-of-file byte 1Ah, which might play quite different role(s) inside executable files and in binary data files.

C:\DOS\MS7\ – is an example of a path to the source file. For allowable alternative forms of path specifications see 2.02-01 - 2.02-03. If path to the source file is not specified, this source will be searched for within the current directory only (paths inside the PATH variable are ignored by COPY command).

Trial.txt – an example of a name for the source file to be copied. If source file name has an extension, it must be specified. Source file may have "A" (Archive) and "R" (Read-only) attributes. Source files with "H" (Hidden) and "S" (System) attributes are not copied.

A:\DOS – an example of a path to the existing target directory, where the copy should be placed. But if the \DOS directory doesn't exist, DOS will be interpreted as a new name for the copy, and a copy named DOS will be created in the root directory of disk A:. If you don't intend to rename the copy, target path must differ from the source path. If the source is not in the current directory, target path may be omitted, and in this case the current directory will be implied as the target.

/V – an optional parameter, inducing verification of the copy. It makes copying more slow and is not needed, when the target media is a hard disk. But when the target media is a floppy, the /V parameter may be worth the delay.

/Y – an optional parameter, giving permission to overwrite any synonymous file in the target directory without prompt. The /Y parameter may be

preset in the COPYCMD environment variable (by command SET COPYCMD= /Y), and then you will not need to specify it in command line. This preset may be overridden with /–Y parameter in command line, when prompt is really needed.

When the /A parameter is the last item in command line, its action is quite different: it doesn't prevent from copying the file as a whole, but rather forces the COPY command to append the copy with end-of-file mark (1Ah), if the latter isn't there yet.

The first path, specified in the COPY command, is always interpreted as a source path, and the last - as the target path. Specifying more than two paths (or filenames) within one COPY command is regarded as an error, except specifying intermediate source(s), preceded by plus symbol (+) for copying with concatenation, for example:

```
COPY /B T1.dat + T2.dat /A + Remark.txt C:\DOS
```

where:

/B – copying mode parameter, which precedes the first source name, retains its effect for all following source names until an opposite parameter specification is encountered (/A in this example). The latter spreads its effect for all remaining source names, if there are any. Since concatenation is applied mainly to non-executable files, the /A parameter is taken as default for copying with concatenation.

T1.dat, T2.dat, Remark.txt – these are three source files for copying with concatenation. Since the paths to the source files are omitted, all these files are implied to be present in the current directory. The following source file names, except the first, must be preceded by plus (+) sign of concatenation.

C:\DOS – is an example of a path to the target directory. When the target directory exists, the result of concatenation will be placed there, and it will inherit the name of the first source (T1.DAT). But if the C:\DOS directory doesn't exist, then the last name DOS will be interpreted as a new name for the combined file, and the result of concatenation named DOS will be written into the root directory of disk C:.

A filemask instead of the source filename is allowed, but such specifications should be used with great caution. Let's consider an example:

```
COPY /B T*.dat  C:\DOS\Concat.dat
```

The COPY command checks the last word in the target path (to the right of the last backslash) on whether it is a name of an existing subdirectory or not. If subdirectory CONCAT.DAT doesn't exist, all copies of files, conforming to the given filemask, will be concatenated into a new file, which will be created in the C:\DOS directory and will be given name Concat.dat. But if a subdirectory with CONCAT.DAT name exists yet, all files, conforming to the filemask, would not be concatenated, but rather would be copied

one-by-one separately into this subdirectory. The latter example shows that the COPY command gives you an opportunity to obtain quite different results with just the same command line. This is why you must know exactly, whether the last name you specify does or doesn't coincide with the name of an existing subdirectory.

For copying with concatenation the same file may be specified as a target and as the first source, and this file will be appended with contents of the following source file(s). Specifying a non-first source as the target is not allowed (contents of this source will be lost). When new name for the copy can't be misinterpreted, then you may specify the same path to the source and for the target or even omit both paths at all (current directory will be implied).

Copying of one file into itself is qualified as an error. Nevertheless the form of copying with concatenation allows to specify the same paths for a non-renamed copy and for the source. At the same time the name of the second file for concatenation may be omitted, for example:

```
COPY /B \DOS\File.ext +,,\DOS
```

This form of pseudocopying doesn't change file's contents, but is used in order to assign current date to the file or in order to delete the file if it is empty (see notes 1 and 2 below).

The reserved word CON (= console) in place of source in COPY command causes the command interpreter not to parse the following input lines as command lines, but to accept the following input as text:

```
COPY CON C:\DOS\Remarks.txt
```

This command enables to write text typed from keyboard into Remarks.txt file, created in C:\DOS directory, until F6-ENTER key combination is pressed to return to command line input (see 1.04 for more details).

Reserved words PRN (printer), LPT1 - LPT4 (parallel port), COM1 - COM4 (serial port) or NUL (virtual "black hole") may be used as targets for copying instead of a file. The command

```
COPY CON PRN
```

turns computer into a typewriter. Of course, the chosen target must be properly configured, and the connected terminal device must be able to respond to DOS's request. When the target is a device, then the /A parameter (copy as ASCII text) is taken by default. Contrary to real terminal devices, virtual device NUL is always configured properly. Copying of a real file from physical media (for example, from a floppy) into virtual "black hole" NUL is sometimes used to test whether this file is not empty or whether it is readable.

Note 1: empty files (having zero length) are not copied.
Note 2: if both target and source(s) are empty, target file is deleted.

Note 3: the result of copying can't be redirected, redirection affects screen messages only.
Note 4: attributes of the source file are not copied.
Note 5: the copy is always given the "A" attribute.

3.07 CTTY – redirection of I/O links

The CTTY command changes settings for all the tree standard I/O channels: STDIN, STDOUT and STDERR. Initial default I/O settings are equivalent to those set by the CTTY CON command: all channels are linked with the CON device (console), that is with the keyboard for input and with display for output. Instead of the console (CON) the CTTY command may accept one of following ports: COM1 (AUX), COM2, COM3, COM4, LPT1 (PRN), LPT2 and also virtual device NUL (for output into "nowhere").

CTTY is an archaic command. Its name (CTTY = Change TeleTYpewriter) reminds about the times when there were no displays, and the I/O consoles resembled old-fashioned teletypes.

Nowadays there are 2 reasons to use CTTY in batch files. The first is to prevent accidental interruption of execution; the second is to prevent indication of undesirable error messages, sent via DOS's STDERR channel, which can't be redirected otherwise. In both cases the problem is solved by redirection into nowhere, represented by virtual NUL device: CTTY NUL precedes the group of commands to be protected, and later CTTY CON restores normal communication with keyboard and display. Inside the protected group of commands (between CTTY NUL and CTTY CON) both internal and interactive interruptions of execution are not allowed, because otherwise no message will be displayed, no input will be accepted, and PC may seem to get hanged. Reboot via CTRL-ALT-DELETE usually remains accessible, though.

CTTY command affects only implicit I/O settings, but it doesn't affect redirections, which are specified in command lines explicitly (2.04-02 - 2.04-05). For example, let's consider the following piece of a batch file:

```
@ctty nul
copy /B Trial.dat Suit.dat
echo Press any key to exit > con
pause < con
ctty con
```

Here a message from the COPY command will not reach the screen, even if it will be an error message. But the message "Press any key to exit" will be shown, because it is directed to the CON device explicitly. The next PAUSE command will work properly too, because its input is explicitly linked with keyboard. This form of CTTY usage needs some caution, but opens attractive opportunities to affect interaction with the user. An example of a batch file with this form of CTTY usage is shown in article 9.03-02.

Note 1: having been banned by CTTY NUL command, the STDERR (error) messages can't be redirected explicitly and are lost.

3.08 DATE – date display and reset

In order to set a new date one has to specify this new date after the name of the DATE command in command line, for example:

```
DATE 11.07.2002
```

When date is not specified, the current date will be shown, and then you will be offered to input a new date via keyboard (pay attention to note 2 below). If you don't want to change the date, just respond to the offer by pressing the ENTER key.

Letters and other textual files are often appended with a line with data signature. For this purpose the DATE command should be used, for example, in the following way:

```
ECHO= | DATE | Find.exe "Current" >> Anyfile.txt
```

Here the first redirection (ECHO= | DATE) automatically responds to the displayed offer and enables non-stop action, the second redirection (DATE | Find.exe) excludes undesirable output lines, and the third redirection (>> Anyfile.txt) appends date signature to the specified file.

Of course, all conditions for performing redirections (2.04-05) and for finding files (Find.exe and the one to be appended) should be met.

Note 1: date and month data order is country-specific and should be set by COUNTRY command (4.05).

Note 2: the offer for date change is supplemented with a prompt for a two-digit year data, but it is a bug: MS-DOS7 demands 4-digit year data.

3.09 DEL – file(s) deletion

The DEL (DELete) command doesn't physically erase files, but rather disables their entries in directory specification. The clusters, occupied by a file with invalid entry, are considered free and may be overwritten during following operations. But until these clusters are not overwritten, the deleted file may be restored, for example, by the UNDELETE.EXE utility from Norton Utilities release.

Here is an example of a command line with the DEL command:

```
DEL D:\TEMP\Filename.ext /P
```

where:

D:\TEMP\ – an example of disk and path specification for the directory, containing the file(s) to be deleted. If path is omitted, then current directory is implied.
Filename.ext – a name example of a file to be deleted;
/P – optional parameter, causing a prompt for confirmation before deletion of each file.

When a filemask (2.01-03) is specified instead of a filename, then all files conforming to this filemask will be deleted. But an attempt to delete all files in a directory by means of filemask *.* causes a stop and a query to the user on whether all files really should be deleted. Execution will be stopped even if the /P parameter is not specified. The user has to respond to the query with Y (yes) or N (no) keystroke.

In batch files a non-stop operation is often desirable, without any prompts and queries. This may be achieved, for example, by execution of the DEL command within a FOR cycle:

```
FOR %%Z in (*.*) do DEL %%Z
```

This form of cycle always displays a list of deleted files (even despite redirection to NUL). You may avoid undesirable messages by implementing the DEL command in another way:

```
ECHO Y | IF EXIST D:\TEMP\*.* DEL D:\TEMP\*.* > NUL
```

In the example above the ECHO Y command provides an automatic response to the query, and the only reason for IF EXIST condition is to avoid the "File not found" error message, when the specified directory (D:\TEMP\) initially is empty.

Note 1: ERASE is another valid name for the same DEL command.
Note 2: files with attributes R (read-only), H (hidden), S (system) and directories can't be deleted with the DEL command. To delete directories the RD command (3.23) should be used instead.
Note 3: the DEL . command (appended with a dot) is equivalent to DEL *.*.
Note 4: the DEL \ command (appended with a backslash) deletes in the root directory of the current disk all the files, which are not protected by attributes.

3.10 DIR – display of directory contents

In MS-DOS the DIR (DIRectory) command is the main instrument of exploring the directories' contents. Here is an example of a command line with the DIR command:

```
DIR C:\DOS /P /A:HS /O:GN /S /L /V
```

where:

C:\DOS – an example of a path to the directory to explore. If not specified, current directory is implied.

/P – optional parameter causing a stop after each screenful of output data until any key is pressed by the user.

/A:HS – a parameter specifying permission to show the items with particular attributes: H (hidden), S (system), A (new or changed files, not saved in an archive yet), R (read-only), D (directories). Prefix "–" may be used to reverse the choice: –H (except hidden), –D (except directories), and so on. Parameter /A without following attributes forces to show all the items in the directory. When parameter /A is omitted, hidden and system files are not displayed.

/O:GN – specification of the sorting order for the displayed items: G – directories first, N – by name (the default), S – by size (smallest first), E – by extension, D – by date and time (earliest first), A – by last access date (earliest first). Prefix "–" may be used to reverse the order: –N – by name with inverse alphabetic order, –S – by size with largest the first, and so on.

/S – an optional parameter, forcing to show contents of subdirectories too.

/L – an optional parameter, forcing conversion of the shown filenames to lower case, otherwise these names will be shown just as they were originally specified.

/V – an optional parameter, causing display of supplementary data: attributes, time of the last access, allocated disk space, total disk space and its usage. Other parameters, having preference over /V, are:

- /W – show item names in 5 columns,
- /B – show item names in one column, without disk summary; no time loss for preparing summary makes the DIR /B command much more fast.

The DIR command can be used to show data about a particular file or about all files conforming to a given filemask:

```
DIR *.txt /P /S /B
```

Absence of path specification in combination with /S parameter in the latter example means that the DIR command in fact will perform a search for files with *.txt suffix in the current directory and in all its subdirectories. If current directory is the root, the search will proceed throughout the current disk. If you are interested in finding a forgotten file only, the /B parameter will make the result more concise and easy to apprehend.

Parameters for the DIR command may be preset in the DIRCMD environmental variable (for example, with command SET DIRCMD= /P /S /B), and then you will get the desirable action of DIR command by default. If needed, you may later override any preset parameter by prefixing a hyphen "–" to it in command line (for example, /–P).

When DIR command is executed with /A parameter, then the *.* filemask (all files) doesn't exclude directories. If you need to display files only, you ought to provide /A: –D

parameter instead. This feature enables to explore whether a directory (or a disk) is empty or not. Consider, for example, the following lines from a batch file:

```
@echo off
set DIRCMD=/a /b
dir *.* > C:\Temp\Found.lst
copy C:\Temp\Found.lst NUL | Find.exe "0 file" > nul
if errorlevel 1     echo Current directory is NOT empty
if not errorlevel 1 echo Current directory is empty
```

The second line specifies parameters so that DIR command will display nothing, if the directory under test is empty. In third line the output of DIR command is redirected into file FOUND.LST. COPY command in the fourth line wouldn't copy an empty file and in this case issues a message "0 files copied". Having caught this message via redirection, the FIND.EXE utility sets errorlevel to zero. The rest two lines are used to sense errorlevel and to display appropriate conditional response.

When the DIR command is executed without /W or /B parameters, it may display filenames in two different ways, depending on operating system environment. Inside DOS window of WINDOWS operating system filenames are displayed "as they are", but in MS-DOS7 environment names of files and their suffixes are displayed separately without a dot between them. This feature may be used as a simple test to determine current operating environment.

Note 1: the DIR \ command shows all files in the root directory.
Note 2: during the DIR command display scrolling of output lines on the screen may be stopped by pressing CTRL–S or BREAK keys; then after any other keystroke scrolling will be resumed.

3.11 ECHO – string output via STDOUT

The words, specified in the same command line after the name of ECHO command are sent as a message into the STDOUT channel; unless redirected, its default terminal point is the CON (console) device, displaying the message on the screen. Examples of string output with the ECHO command are shown, in particular, in previous article 3.10.

The message to display may be up to 123 characters long. Actual message length is limited by the line itself or by the first encountered redirection symbol (2.04-02 – 2.04-05). Message string may include ASCII service marks, shown in appendix A.02-8. But message string must not be empty or begin with words ON or OFF. These and several other exceptions are used to perform special functions:

ECHO ON – switches ON the ECHO flag, enabling to display batch file lines as they are executed (this is the default status).

ECHO OFF– switches the ECHO flag OFF (no lines display). Outside batch files this causes disappearance of command prompt.

ECHO – (without following message) shows current status of ECHO flag.

ECHO= – (appended with equality sign) – sends bytes 0Dh 0Ah via STDOUT channel, just as if the ENTER key were pressed (example – in article 3.08). On the screen or in a file this causes insertion of an empty line.

ECHO+ – sends bytes 0Dh 0Ah via STDOUT, but also sends words specified after plus sign, if there are any, including words ON and OFF. The ECHO command acts similarly, when it is appended with slash or dot

ECHO flag is a local flag, maintaining its state until batch files share common environment, but this state is not inherited and is reset to default each time the command interpreter creates a derived (child) environment.

Command lines displayed in the default ECHO ON state are not the same as original command lines in batch files: in displayed lines all aliases are replaced with their values. This is helpful for debugging. But it is not needed, when file has proved to have no errors. Therefore almost each completed batch file starts with @ECHO OFF command. The "@" character, preceding the ECHO command, prevents display of this command itself.

3.12 EXIT – closure of current interpreter session

Command interpreter COMMAND.COM (6.04) is a resident program, which arranges environment for execution of other programs. On the other hand, command interpreter itself may be launched just as an ordinary program in order to arrange a separate (local) environment, if it is required. In such cases several resident modules of command interpreter may coexist in memory simultaneously, but only one of them may be active – the one which is loaded the last. The EXIT command closes current session of the active resident module, releases the memory occupied by the module, and transfers control to the parent program (which has launched the mentioned resident module).

Just as current interpreter session is closed, its local environment with all values and variables becomes lost. At the same time the former environment of the parent program becomes accessible again, and execution of this program is automatically resumed.

The very first resident module of command interpreter is launched by the IO.SYS loader with the SHELL command (4.26) in CONFIG.SYS file (9.01-01). This first resident module can't transfer control to its "parent", since the IO.SYS loader is not resident and has finished its job yet. If the first resident module of command interpreter were able to execute the EXIT command, then computer were get hanged. To prevent such outcome the COMMAND.COM interpreter must be launched for the first time with the /P parameter (6.04), which disables EXIT command.

3.13 FOR – cycle operator

The FOR cycle operator arranges cyclic execution of other command(s). Assume, for example, that we need to display three short files: First.txt, Second.txt and Third.txt. Instead of sending these files to display with separate commands, the FOR operator enables to do the same in one line:

```
FOR %Z IN (First Second Third) DO TYPE %Z.txt
```

where:

- %Z – an example of a name for cycle variable, which is sequentially set equal to each of the items specified in parenthesis. Cycle variable name shouldn't commence with a digit. Usually it is a one-letter name, preceded by a percent symbol (%), when the cycle is executed from command line, or by double percent symbol (%%), when the cycle is executed from a batch file.
- IN – a required reserved word, introducing the following list of variable's values, specified inside parenthesis.
- DO – a required reserved word, introducing the following name of the command, which is to be executed in the cycle as many times as many values are specified for the cycle variable. In each iteration a new value is substituted for the name of cycle variable.

Item(s) in parenthesis may be any word(s), including environmental variable substitutions (such as %TEMP%) and dummy parameter substitutions (2.03-03). Items in parenthesis may be separated by spaces, or by semicolons (;), or by commas (,). Note, that paths in PATH variable value are separated by semicolons, hence the PATH's value will be disintegrated by the FOR cycle into a group of separate paths. This operation is often used in order to determine accessibility of a given file or in order to provide explicit path specification for it. The following piece of a batch file shows a typical path determination example:

```
@echo off
set P=
FOR %%Y IN (. %PATH%) DO if exist %%Y\Fc.exe set P=%%Y\Fc.exe
if %P%"==" echo Requested file hasn't been found!
if not %P%"==" echo Path to the requested file is %P%
```

Here in the second line an auxiliary variable P is assigned an empty value, and in the third line - the path to the specified file, if the latter happens to be found. Note, that cycle variable name (%%Y) is preceded by double percent symbol, as it must be in batch files. The last two lines check presence of auxiliary variable's value and issue a corresponding message according to the result.

Item(s) in parenthesis may include wildcards (2.01-03), but items with wildcards become interpreted as filenames, which should be searched for in the current directory or according to the preceding path, if it is specified. Automatic search along all the paths in the PATH variable is not implied. Each of several items in parenthesis may include wildcards, for example:

```
FOR %%X IN (A:\*.txt A:\*.doc) DO COPY /B %%X C:\DOS
```

Sometimes it is desirable to display separately each operation performed within FOR cycle, but not the cycle itself. This opportunity may be illustrated by the following modification of the preceding example:

```
ECHO ON
@FOR %%X IN (A:\*.txt A:\*.doc) DO COPY /B %%X C:\DOS
@ECHO OFF
```

If the file, specified in parenthesis with wildcards, is not found, then corresponding operation is skipped without any error message. This is why the FOR cycle sometimes is used as a means to get rid of undesirable error messages (see 3.09 for an example). One more "side effect" of the FOR cycle is that it enables to parse a multi-word value of an environmental variable and to get rid of extra spaces, which may precede or follow separate words within this value.

Inside parenthesis a string, including separation symbols, may be regarded as one item, if it is enclosed in double quotes (absence of the closing quote is qualified as an error). Double quotes themselves are not regarded as belonging to the item. This enables to specify several different commands in one line, for example:

```
FOR %%Z IN ("set E=%W%" "echo E is set" "goto L23") DO %%Z
```

There are sequences of operations, which can't be performed in separate lines, but can be performed within the FOR cycle. Examples of such sequences are shown in 46-th line of batch file in article 9.03-02, and also in the 6-th line of batch file in article 9.01-03.

Inside parenthesis the commands, enclosed in double quotes, may include substitutions of variable's values (as %W%), conditional commands (IF), jump commands (GOTO) and redirections (2.04-02 - 2.04-05). When a jump to a label is performed from within the FOR cycle, the next operations (placed to the right, if such exist) will be skipped.

Redirections inside the FOR cycle spread their action on all following commands. For example, in a cycle

```
For %%Z in ("echo 1-st line >> Q.txt" "echo 2-nd line") do %%Z
```

the words "2-nd line" will not be shown on the screen, but rather will be appended to the Q.txt file. Redirection for the following operation(s) can be changed, but it must be specified explicitly.

The ability of FOR cycle to take away enclosing double quotes from its arguments is essential not only for execution of commands, but also for displaying messages, protected from parsing by double quotes.

Nested FOR cycles are not allowed, but an internal FOR cycle may be executed by a separate resident module of command interpreter (COMMAND.COM), which itself is launched within an external FOR cycle, specified in the same command line. If inside a batch file the FOR cycle is used to execute another (secondary) batch file, this secondary batch file may contain its own internal FOR cycles.

Note 1: the name of the cycle variable must be chosen so as to prevent interference with any of currently used other variables.

Note 2: inside parenthesis the forward slash (/) is regarded as separator, but any single item preceded by the forward slash will be appended to this slash and converted to the upper case (in earlier DOS versions slash acts otherwise).

Note 3: attempts to redirect all messages from FOR cycle affect only the first operation within this cycle. If the following operations have no explicit redirections, they will be subjected to default settings.

Note 4: when interpreter COMMAND.COM is explicitly launched from command line in order to execute a FOR cycle, the cycle variable must be preceded by double percent signs, just as in batch files.

3.14 GOTO – jump to a label

The GOTO command performs a jump within batch file to a label, which must be specified in any line of the same batch file. A line with label begins with a colon (:), and after the colon an arbitrary label's name follows. The same name must be specified after GOTO command. When label's name is long, its first 8 characters only are taken into account. Synonymous labels in one batch file are not allowed. If, for example, there is a label :L36 in some line of a batch file, then a jump to this label is performed by command

```
GOTO L36
```

The GOTO command may be preceded in the same line by conditional operator IF (3.15). A lot of jump examples, both conditional and unconditional, can be found in articles 9.03-02, 9.09-02.

Note 1: label name after the GOTO command can be obtained by substitution for a variable's name (%VAR%, for example).

Note 2: a value of dummy parameter (%1, %2..) cannot be used after the GOTO command as name of a label, but it may constitute a part of this name after any preceding letter(s).

Note 3: the GOTO command doesn't affect errorlevel, thus giving an opportunity to continue errorlevel checks after the jump.

3.15 IF – condition operator

Condition operator enables to perform three types of condition checks: existence check, equality check and errorlevel check.

Command line with conditional execution of any operation must commence with conditional operator IF, followed by condition type definition, condition specification and full specification of the command, which should be executed if the condition is met. Several condition operators with separate condition definitions and specifications may be written sequentially in one line, and then logical operation AND is applied implicitly to results of separate condition checks. Examples of combining several condition checks in one line are shown in articles 3.15-03, 9.03-01, 9.09-02. Peculiar composition of command line for each type of condition checks is described below in detail.

3.15-01 Existence condition check

Existence condition check, performed by IF EXIST command, can be applied to files, directories and logical devices. Inversion of existence condition – absence condition check – is performed by IF NOT EXIST command. Here are two examples of existence check usage:

```
IF EXIST C:\DOS\Format.com C:\DOS\Format.com A: /S
IF NOT EXIST C:\DOS\Format.com ECHO Format.com isn't found!
```

where:

EXIST – a reserved word, defining type of check and forcing to interpret the following element as a name or a mask (may be with path) of the object to be searched for.

C:\DOS\Format.com – an example of a filename to be searched for; preceded by a path. This filename will be searched for in the specified directory only. When path is omitted, then current directory is implied.

C:\DOS\Format.com A: /S – an example of a utility to be launched if the preceding condition is met (if the check returns TRUE). Note, that utility name should be followed by all necessary parameters. When path is not specified, the utility will be searched for in current directory and then along all the paths, stored in PATH variable's value.

NOT – a reserved word, meaning a logical inversion of the result (TRUE or FALSE), returned by a check of any condition type.

ECHO Format.com isn't found! – another example of a command to be executed depending on preceding condition (when the utility is NOT found).

Given examples enable to execute a specified utility (FORMAT.COM), when it is found in its proper place, or to have an intelligible message displayed, when this utility isn't found.

Names of logical devices are reserved words (2.01-01), assigned by DOS's core or by drivers during loading procedure. By existence check, applied to a logical device name, one can clear up whether a particular driver is loaded. Full list of logical devices in your computer is shown by the MEM.EXE utility being launched with /D parameter (6.17). For example, the EMM386.EXE (5.04-02) driver reserves logical device name EMMXXXX0. Hence the loading check for this driver can be done with the following line:

```
IF NOT EXIST EMMXXXX0 ECHO The EMM driver isn't loaded!
```

When you use a filemask with wildcards instead of utility name to be searched for, it will be referred to conforming files only, but not to directories. Existence condition with filemask *.* is true, when there is at least one file (or more, example – in 3.09). In order to check existence of a directory you should append the directory name with a name of virtual file NUL:

```
IF EXIST C:\DOS\NUL ECHO The C:\DOS directory exists!
```

However, the shown check for existence of a directory may fail on CD-ROMs because of peculiarity of their file system (ISO 9660).

It is often important to provide non-stop execution of batch files. The existence check provides non-stop execution on accessible media only: the file or directory may not exist, but disk must exist (must be inserted) and must be formatted with a file system accessible for MS-DOS7 (FAT12, FAT16, FAT32) or accessible by means of installed drivers. When it is not known whether the media is accessible, a non-stop execution of the existence check still may be performed, but it needs special measures to prevent critical error handler calls (8.02-84) and to avoid appearance of undesirable messages (example – in 9.03-02).

3.15-02 Equality condition check

Equality condition check is applied to two words, separated by double equality symbol (= =). Since there is no sense in comparing a-priori known words, equality condition implies using aliases, which may be dummy parameter(s) or variable's value(s) (2.03-03). Wildcard symbols (? and *) in words to compare are allowed, but these are interpreted as ordinary symbols and are not expanded as wildcards. Contrary to ordinary DOS' practice, upper and lower case letters in words to compare are regarded as NOT equal. Here are two examples of equality check usage in batch files:

```
IF %VAR%==%2 GOTO L23
IF NOT %VAR%==%2 GOTO HELP
```

where:

%VAR% – an alias to be replaced with value of environmental variable VAR;
%2 – an alias (dummy parameter) to be replaced with the second parameter's value of the batch file;
GOTO L23 – a command to be executed if the %VAR%= =%2 condition is met;
NOT – a reserved word, meaning a logical inversion of the result (TRUE or FALSE), returned by condition check;
GOTO HELP – a command to be executed if the %VAR%= =%2 condition is NOT met.

Of course, any one of the words to compare may be specified directly, without aliases. Each of the words to compare may combine one or more aliases with directly specified part(s). But words to compare are not allowed to be empty: this is regarded as syntax error. Since any batch file may be executed without parameters, it must be prepared to the case when its dummy parameter (%2 in the example above or any other) becomes empty. The simplest way to solve the problem is to append any certain symbol (for example, a dot) to both left and right parts of the equation:

```
IF %VAR%.==%2. GOTO L23
IF NOT %VAR%.==%2. GOTO HELP
```

These equality checks do the same as in the previous example, but are immune against the empty value of the aliases. The same principle is used to compose a check on whether the value of a variable (or of a dummy parameter as well) is empty:

```
IF .==%CASH%. ECHO The CASH variable has an empty value
```

Special care should be taken when the value of a variable may include spaces. The word to the right of the double equality symbol doesn't allow space(s) in its value. If the value of the variable CASH (in the example above) includes spaces, it will be interpreted as syntax error. But the word to the left of double equality symbol is allowed to have spaces inside. When this word consists of several items, separated by spaces, only the first (leftmost) item will be taken into account. This is illustrated by the following three examples (all three are valid):

```
IF NOT A: B: C:.==. ECHO Compared items are not equal
IF .A: B: C:==.A: ECHO Compared items (.A: and .A:) are equal
IF . B: C:==. ECHO Compared items (dots) are equal too
```

In all these three examples presence of the B: and C: items is ignored.

3.15-03 Errorlevel condition check

When execution of any utility in DOS is about to terminate, it may leave errorlevel code, which is in fact a message to the following algorithms. Errorlevel informs whether the terminated execution has been successful or not, and if not, then what kind of obstacle

has been encountered. Errorlevel is a 8-bit binary code in DOS's swappable data area (offset 14h in A.01-03), but it is presented as decimal number from 0 to 255 (without sign). Errorlevel 0 means successful termination, other errorlevel values usually mean different kinds of errors, interpreted specifically for each utility.

Errorlevel condition is identified by presence of the word ERRORLEVEL. It gives an opportunity to check whether the errorlevel code is equal or greater than a specified decimal number, for example:

```
IF ERRORLEVEL 1 ECHO Execution has failed
IF NOT ERRORLEVEL 1 ECHO Execution has terminated successfully
```

The check in the first line of the example above returns TRUE for all errorlevel values from 1 to 255, that is for all possible unsuccessful outcomes. The check in the second line includes reserved word NOT and this is why acts as logical inversion, returning TRUE for errorlevels lower than 1, that is for a single value 0, which means successful termination.

When you need to execute a separate procedure for a single type of erroneous outcome, you may concatenate errorlevel checks. Suppose, for example, that in the case of errorlevel 15 you need to perform a jump to label ERROR15. This may be achieved by the following command line:

```
IF NOT ERRORLEVEL 16 IF ERRORLEVEL 15 GOTO ERROR15
```

The first check in the line above returns TRUE for all errorlevels from 0 to 15, and the second - for all errorlevels from 15 to 255. The result is that errorlevel 15 becomes the single errorlevel value, which enables to execute the following GOTO command.

Note 1: non-zero errorlevel code is used by some utilities to indicate different circumstances of normal (not erroneous) outcome.
Note 2: all internal commands (3.01 - 3.34) don't return and don't change errorlevel code.
Note 3: mixed successive concatenation of several existence checks, equality checks, and errorlevel checks in one line is allowed in the same way as that shown above for two errorlevel checks.

3.16 LFNFOR – long filenames display mode

LFNFOR is an undocumented local switch, which may be set ON (LFNFOR ON) or OFF (LFNFOR OFF). Being used without parameters, LFNFOR command shows the state of this switch. It's default state is OFF. Under "bare" MS-DOS7 the state of LFNFOR is ignored, but inside DOS box under Windows OS switching LFNFOR ON enables non-truncated treatment of long file names by the FOR command (3.13), for example, by

```
FOR %%Z in (*.*) do echo %%Z
```

When LFNFOR is switched OFF, the FOR command truncates long file names to 8 characters, just as it always does in MS-DOS7.

3.17 LH – load beyond conventional memory

The LH command (LH = Load High) loads drivers and TSR utilities beyond the 640 kb boundary of conventional memory in computers having the 80386 or higher processor. Access beyond the 640 kb boundary must be enabled in advance by command DOS=UMB (4.08) in CONFIG.SYS file and by loading memory managers: the HIMEM.SYS driver (5.04-01) and then either EMM386.EXE (5.04-02) or UMBPCI.SYS (5.04-04) driver. In both cases access to the TSR modules, loaded by the LH command, will be performed via the UMB region (640 - 1024 kb) of address space. When there is no more free space in UMB region, the LH command issues no error message and continues to load drivers and TSR utilities into conventional memory below 640 kb.

LH acts similar to the INSTALLHIGH command (4.16); the main difference is that INSTALLHIGH command is performed by the IO.SYS loader and can't take part in memory optimization procedure (5.04-03). The LH command is performed by COMMAND.COM interpreter from ordinary command line or (preferably) from a line in AUTOEXEC.BAT file. Here is an example of a line from AUTOEXEC.BAT file, in which the LH command is used to load MSCDEX.EXE driver:

```
LH /L:1,23680 \DOS\DRV\Mscdex.exe /D:CD1 /E /S /V /L:0 /M:32
```

The name of the driver is preceded by a path (\DOS\DRV\ ), which may take any of its allowed forms (2.02-01). If the path is omitted, the driver will be searched for in the current directory and throughout all the paths, specified in PATH variable's value (2.02-02). All items following the driver's name are not identified by LH command, but rather are transferred to the driver as its specific parameters.

Between the name of LH command and specification of the software to be loaded there may be an optional /L parameter, which gives an opportunity to point out a particular part of USB memory region, which should be devoted for access to each TSR module (see also 4.07). In the example above this parameter looks as /L:1,23680, where /L:1 means addressing via the first part of UMB region, and the number 23680 is the size of space (in bytes) required for TSR module of MSCDEX.EXE driver.

Size specification after the /L parameter is optional, but when the size is specified, the LH command can accept one more parameter /S, for example:

```
LH /L:1,2160 /S \DOS\COM\Escape.com
```

The /S parameter means that allocated UMB block should be truncated to the specified size. This results in the most efficient usage of address space, but doesn't guarantee from a crash, if size specification is not quite correct. It is not recommended to

use the /S parameter apart from memory optimization procedure, performed by memory optimization utility MEMMAKER.EXE (5.04-03). During this procedure the /L and /S parameters together with exact size specification will be automatically inserted in all those lines of AUTOEXEC.BAT file with LH command.

Note 1: if access beyond conventional memory is opened by UMBPCI.SYS driver (5.04-04), then LH command loads TSR modules into UMB memory region (640 - 1024 kb). But EMM386.EXE driver (5.04-02) acts otherwise: it adjusts processor's address translation table (TLB) so that access to memory beyond 1088 kb is performed via the same UMB region of address space. This is why in the latter case the same LH command physically loads TSR modules not into the UMB region, but elsewhere beyond 1088 kb.

3.18 LOCK – forbid concurrent access

Requests of various programs for disk access are controlled by MS-DOS7 in order to provide proper order of access and effective buffering. Programs, which require direct access to a disk, must coordinate their operations with MS-DOS7 by means of the INT 13\AH=45h interrupt (8.01-58). But some programs don't do that, for example, the program for recovering deleted files UNDELETE.EXE from MS-DOS6.22 release. In order to permit such programs to do their job the COMMAND.COM interpreter in MS-DOS7 provides the LOCK command. It gives exclusive rights to access the requested disk for the program, which will be launched next.

Arguments for the LOCK command are one or more letter-names of the disks, which should acquire exclusive treatment. During execution of LOCK command you will be asked for confirmation (Y or N) from keyboard. In order to avoid stops for confirmation in batch files you have to prepare a response in advance, for example

```
ECHO Y | LOCK C: D:
```

The result of the shown command line will be excusive non-interrupted access to disks C: and D: for the program which will be launched next. When this program terminates, the locked state of disk(s) should be turned off by UNLOCK command (3.31). Since direct access operations may be nested, up to 256 lock levels are allowed. Of course, each involved program must be supported by proper sequence of LOCK and UNLOCK commands.

3.19 MD – make directory

The MD command enables to create a new directory or subdirectory, for example:

```
MD  C:\DOS\ARC
```

where:

C:\DOS\ – an example of a path to an existing directory, where the new subdirectory should be created;
ARC – an example of a unique name for the new subdirectory to be created. Backslash after this name is not allowed.

The unique new name is a required argument, but the preceding path is optional. When it is omitted, new (sub)directory will be created in the current directory of the current disk.

Note 1: MKDIR is another valid name for the same MD command.

3.20 PATH – search path(s) specification

The PATH command defines the default paths for searching programs, which have no prescribed path and are not present in the current directory. The paths, specified by PATH command, constitute the value of synonymous environmental variable PATH. Its value may be defined by SET command (3.26) as well. But there is an important difference: unlike SET command, PATH command automatically converts all the characters in the specified path(s) into upper case (otherwise search procedure may go wrong). The PATH command must be followed by one or more existing paths, separated by semicolon(s), for example:

```
PATH  C:\DOS\VC4;C:\DOS\MS7;C:\WINDOWS\COMMAND
```

Ability of PATH command to convert arbitrary word(s) into upper case is sometimes used in order to avoid ambiguities further in course of case-sensitive equality check.

Note 1: within a string of path specifications there must be no spaces on both sides of each semicolon separator.
Note 2: there must be no semicolon at the end of a string of path specifications.
Note 3: the PATH command with no following arguments just displays the defined paths and leaves them unchanged.
Note 4: the PATH ; command (followed by one semicolon only) deletes all previously defined paths.
Note 5: the name of PATH command may be separated from the following string of path specifications by either a space or by equality sign (=) as well.

3.21 PAUSE – temporary stop

The PAUSE command, being encountered in a line of a batch file, stops execution of this batch file and displays message "Press any key to continue...". This message is not quite true, because CTRL-C, CTRL-BREAK and ALT-03 keystroke combinations terminate execution, enabling to bypass all the rest lines of batch file (this action doesn't depend on the BREAK status). When PAUSE's message is not desirable, it may be redirected:

```
PAUSE > NUL
```

PAUSE command may be followed in the same line by a commentary string, just like the REM command. This commentary will not be displayed unless the ECHO flag (3.11) is set ON.

When default communication with console is halted by the "CTTY NUL" command (3.07), then PAUSE command must be given explicit input redirection:

```
PAUSE < CON
```

In batch files redirection of the 03h symbol (shown as ♥ , see A.02-8) to PAUSE command provides the shortest way to quit execution of the batch file at once, without any pause:

```
ECHO ♥| PAUSE > NUL
```

Symbol 03h may be inserted into command line with ALT-03 keys, the digits should be entered via numerical keypad while the ALT key is kept pressed. But you shouldn't dare to disable such line by a preceding REM command (3.24): the ECHO command only will be disabled, redirected symbol wouldn't be received by PAUSE command, and the computer will get hanged.

3.22 PROMPT – prompt specification

Command PROMPT redefines the value of synonymous environmental variable PROMPT, which specifies the form of DOS's command prompt. Usually the PROMPT command is written in a line of AUTOEXEC.BAT file, but it also may be entered from an ordinary command line. The name PROMPT should be followed by the suggested prompt text. In this text pairs of characters, beginning with a dollar symbol ($), are interpreted in a special way and are substituted with other data, which can't be written into prompt text directly. Here is a table of correspondence between character pairs and the substituting data:

$Q	equality sign (=)
$$	single dollar symbol ($)
$T	current time
$D	current date
$P	current disk’s letter-name and path to current directory
$V	Windows' version number
$N	current disk’s letter-name
$G	right arrow (or greater-than) sign (>)
$L	left arrow (or less-than) sign (<)
$B	vertical bar (or pipe) sign (|)
$H	the 08h code "Backspace" (A.02-8)
$_	carriage return (0Dh) and linefeed (0Ah)

$E code 1Bh "Escape" (A.02-8)

The PROMPT command without arguments deletes the variable PROMPT, and then DOS's prompt would present current disk's letter-name appended with right arrow sign (>), the same prompt as after command PROMPT NG. By default MS-DOS7 assigns to the PROMPT variable another value (PROMPT PG), which corresponds to most common form of command prompt: full path to the current directory appended with right arrow sign.

The data, displayed by prompt, can be written into a file and then assigned as a value to an environmental variable. Consider the following example of batch file lines:

```
prompt @echo off$_Set Ret$q$p
C:\Command.com /c Ret.bat > Ret.bat
Call Ret.bat
```

The first line in the example above sets a complicated form of a prompt, and the second line writes this prompt into a new batch file RET.BAT. Obtained RET.BAT file's contents may look as follows:

```
@echo off
Set Ret=D:\BACKUP
```

Note, that prompt parameter "Set Ret$q$p" has been transformed into "Set Ret=D:\BACKUP", where "D:\BACKUP" is an example of current disk's letter-name followed by actual current path at the moment of batch line execution. If you execute the RET.BAT file with CALL command (3.02), then actual full path will become written into value of environmental variable RET. After that you can delete RET.BAT file and may use the RET variable whenever necessary in order to return to the former disk and directory:

```
%Ret%\
CD %Ret%
```

Another example of PROMPT command usage for obtaining current disk's letter-name is shown in article 9.01-03. Current time, date and OS version number can be written into environmental variable(s) in a similar way.

3.23 RD – remove directory

The RD command (Remove Directory) enables to delete a directory, if the following conditions are met:

– the directory to be deleted is empty;
– the directory exists on a writable disk;
– the directory is not the root directory of a disk;

– the directory is not the current directory on its disk, even if the addressed disk is not the current disk.

Here is a RD command usage example:

```
RD D:\TEMP\NOTES
```

where:

NOTES – is the name of a directory to be deleted;

D:\TEMP\ – is an example of a path to the directory to be deleted. The path may be specified in any of its allowed forms (2.02-01, 2.02-03). If path is omitted, a subdirectory of the current directory is implied.

Note 1: RMDIR is another valid name for the same RD command.

3.24 REM – remark line

The REM command (REMark) forces COMMAND.COM interpreter to ignore all the following character(s) in the same command line up to any nearest sign of redirection (2.04-02 - 2.04-05) or up to the end of the line. Main mission of the REM command is to provide an opportunity to insert lines of commentaries in batch files. Commentaries may be up to 123 characters long in one line. The REM command is used for those commentaries, which shouldn't be displayed during normal execution of a batch file, but rather are to be displayed only for tracing while the ECHO flag is kept ON (3.11).

The REM command is sometimes used to disable an executable line in a batch file, but it can't disable redirection. Double colon "::" (2.04-01) is more suitable for this purpose.

The other mission of REM is an "empty" command, which is accepted as valid and formally is executed, but does nothing (see VCEDIT.EXT in 6.25-03 for an example). Since the REM command sends no output into STDOUT channel, redirection

```
REM > Anyfile.ext
```

is used to create an empty file with specified name. If a synonymous file exists there yet, it will be overwritten and will become empty. Overwriting of a real file with a file of zero length erases address of its first cluster in directory specification. Therefore files, overwritten by a file of zero length, can't be restored by UNDELETE.EXE or by other similar utilities.

Note 1: REM command shouldn't be used to disable command lines with intermediate redirection (2.04-05). The REM command disables only the first command in such line, its output is not sent, and command in the rest part of command line never gets the awaited data. Therefore computer may get hanged (example – in article 3.21).

Note 2: inside "DOS box" of Windows OS empty redirections are not performed, so there the REM command can't be used to create a file of zero length.

3.25 REN – rename a file

The REN command (REName) enables to rename one file or several files at once if their names conform to a certain mask. Here is an example of renaming one file with REM command:

```
REN C:\DOS\Notes.txt Notes.old
```

where:

C:\DOS\ – an example of a path to the file to be renamed; the path may be specified in any of its allowed forms (2.02-01, 2.02-03) or may be omitted.

Notes.txt – current name of the file to be renamed.

Notes.old – an example of new name for the same file; the new name must be specified without preceding path, even if the file to be renamed exists anywhere else beyond the current directory.

Specifying wildcards within the first (old) name only is allowed, but often leads to an error: attempt to create several synonymous files in one directory. Therefore it is recommended to specify wildcards in the same positions in both old and new filenames, so as to retain unique features of each filename. Characters hidden under wildcards will not be changed. Suppose there is a group of files Part_01.txt – Part_12.txt, which should be renamed into Chap_01.txt – Chap_12.txt. This operation is performed with one command:

```
REN Part_??.txt Chap_??.txt
```

Note 1: directories and files with H (hidden) attribute can't be renamed by the REN command.

Note 2: attributes of the renamed file(s) remain unchanged.

3.26 SET – value assignment to a variable

When SET command is specified without parameters, it displays all variables of the current environment together with their values. But if name of the SET command is followed by any word, this word is interpreted as a name of environmental variable, which should be assigned a new value, for example:

```
SET  TEMP=D:\Temp
```

where:

TEMP – is an example of variable's name;

D:\Temp – is the value to be assigned to the variable TEMP. Equation sign(s) and redirection symbols (2.04-02 – 2.04-05) inside value specification are forbidden.

Note 1: any space to the right of the equation sign between valid characters, preceding or following valid characters (up to the end line mark) will be included in variable's value.
Note 2: if a part to the right of the equation sign is empty, then specified variable will be deleted (will cease to exist).
Note 3: the SET command is able to expand environmental space, when it is not enough to accommodate the new value of a variable.
Note 4: in batch files the value to the right of the equation sign may include substitutions for other variable's names (for example, %VAR%) and for dummy parameters (2.03-03). All such aliases will be replaced with their values before defining the new variable.
Note 5: a synonymous command SET (4.25) in lines of CONFIG.SYS file is interpreted by IO.SYS loader. The latter doesn't perform substitutions and redirections, but thus it gives an opportunity to include corresponding symbols in variable's value(s).

3.27 SHIFT – dummy parameter's order shift

The SHIFT command shifts by –1 (minus one) the numerical order of dummy parameters (2.03-03) in a batch file, so that former %0 is lost, former %1 becomes %0, former %2 becomes %1, and so on. It's important to notice, that the dummy parameter, which becomes the 9-th, previously was the 10-th and couldn't be accessed. Thus the SHIFT command gives an opportunity to specify more than 9 dummy parameters for a batch file and to access them sequentially, shifting their numeration from one iteration to each next. An example of such numeration shift is presented by a subroutine in lines 29 – 38 of DISK.BAT file in article 9.03-02. When the shifted address order comes to the end of parameter's sequence, the last dummy parameter becomes empty, and this is a sign to terminate execution of the whole cycle containing the SHIFT command.

3.28 TIME – time display and reset

In order to set a new time one has to specify this new time after the name of the TIME command in command line, for example:

```
TIME 11:39:23,24
```

where the successive numbers mean hours, minutes, seconds and the hundredth parts of a second. Separation symbols within the shown time specification are colons and a comma, but it depends on what national conventions are set by the COUNTRY command (4.05).

When time is not specified, the current time will be shown, and then you will be offered to input a new time via keyboard. If you don't want to change time, just respond to the offer by pressing the ENTER key.

In order to append a textual file with a time signature, the TIME command should be used, for example, in the following way:

```
ECHO= | TIME | Find.exe "Current" >> Anyfile.txt
```

Here the first redirection (ECHO= | TIME) automatically responds to the displayed offer and enables non-stop action, the second redirection (TIME | Find.exe) excludes undesirable output lines and the third redirection (>> Anyfile.txt) appends time signature to the specified file. Of course, all conditions for performing redirections (2.04-05) and for finding files (FIND.EXE and the one to be appended) should be met.

3.29 TRUENAME – canonical form for path and name

Any specification, following TRUENAME command in the same command line, is interpreted as name of an object (file or directory), which may be preceded by a path. The TRUENAME command doesn't check whether the given specification corresponds to a real file or to a real directory structure, but rather tries to translate it into canonical form, which must commence with disk's letter-name and must include full path to the object. If the given specification isn't complete, it may be automatically supplemented with letter-name of the current disk and a path to the current directory. If the given specification contains dot aliases (2.02-03), these will be expanded to full paths which they denote. Besides this, all letters are uppercased, forward slashes are converted to backslashes, asterisks (2.01-03) are converted into appropriate number of question marks, long names are truncated to 8 characters, long suffixes – to 3 characters.

If the original specification points out a fake path, arranged by utilities ASSIGN.COM, JOIN.EXE or SUBST.EXE, then the TRUENAME command returns the true path. When used without following specification, the TRUENAME command returns full path to the current directory, just as CD command (3.03).

Note 1: the TRUENAME command doesn't display error messages.
Note 2: the TRUENAME command can't be applied to network paths, unless a LAN driver is installed.
Note 3: the action of TRUENAME command is based on INT 21\AH=60h (8.02-72).

3.30 TYPE – read a file to STDOUT

The TYPE command reads contents of a specified file and sends it line-by-line into STDOUT channel, which has the CON device (display) as its default terminal point. Sending may be terminated by CTRL-C or CTRL-BREAK keystrokes, or suspended by pressing CTRL S or BREAK keys and then resumed by any other keystroke. Usage example:

```
TYPE  C:\DOS\Notes.txt
```

where:

Notes.txt – a name example of the file to be displayed; wildcards in a name are not allowed by TYPE command.

C:\DOS\ – is an example of a path to the file to be displayed; the path may take any of allowed forms (2.02-01, 2.02-03) or be omitted. In the latter case the file is implied to exist in the current directory.

Output of the TYPE command may be redirected, for example, to the printer, connected to LPT1 port:

```
TYPE A:\Config.sys > PRN
```

The TYPE command is often used together with the MORE.COM viewer (6.19), which displays long STDOUT messages page-by-page.

3.31 UNLOCK – concurrent access permission

After termination of each program, which has been given direct access to a disk with LOCK command (3.18), the original access state should be restored by the UNLOCK command, applied to the same disk, for example:

```
UNLOCK C:
```

In fact the UNLOCK command doesn't permit concurrent access, but rather decreases by 1 the count of nested ban levels, forbidding concurrent access. Thus a proper treatment of nested program calls is provided. But if the original ban level was 1, then UNLOCK command will reactivate a queue of requests for access to the disk. This is important in multitasking operating environment, for example, in "DOS box" under Windows95/98 OS. If a disk originally is not locked, no action will be taken by the UNLOCK command. In any case no message will be displayed.

3.32 VER – operating system version display

In MS-DOS7 and in MS-DOS8 the VER command (VERsion) shows version number of the corresponding WINDOWS software release. Having been supplied with /R parameter

```
VER /R
```

the VER command appends its version message with a remark about whether the DOS's kernel is loaded into high memory area or not.

3.33 VERIFY – verification function control

VERIFY command, being launched without parameters, indicates the state of verification function, which defines whether or not each written file should be re-read and compared with its origin. By default the verification function is turned OFF, and verification is not performed. You can change the state of verification function with VERIFY ON or VERIFY OFF commands (more about this – in note 2 to 8.02-60).

Note 1: high reliability of modern HDDs makes verification unnecessary, it leads to time loss and to excess disk wear. It is better to initialize verification by the /V parameter of COPY command (3.06) for copying file(s) only to diskette(s).

Note 2: the VERIFY function acts as a global switch: it doesn't loose its state when current resident module of command interpreter finishes its job and local environment becomes lost.

3.34 VOL – disk's label and serial number

Disk's label is a string of up to 11 characters long, chosen by the user. If the user wouldn't define label while formatting, disk will be given the NO NAME label. Later disk's label may be changed with the LABEL.EXE utility (6.16) or by means of some TSR shells (Norton Commander, Volcov Commander, etc.).

Serial number is a 8-digit hexadecimal identifier, which is automatically assigned to a disk during formatting procedure. Diskettes may have no serial number, if their production technology employs formatting by magnetic contact copying. Contrary to this during ordinary copying procedures both label and serial number are inherited by diskette-copy from diskette-origin.

In order to display label and serial number of a particular disk, you have to specify its letter-name after the name of the VOL (= VOLume) command in the same command line, for example:

```
VOL A:
```

If disk's letter-name is omitted, then volume label and serial number of the current disk will be displayed.

Chapter 4 Configuration commands

MS-DOS7 loading configuration is prescribed by configuration options in three non-formatted textual files, which must be present in root directory of the bootable disk: MSDOS.SYS (5.01-01), CONFIG.SYS (9.01-01) and AUTOEXEC.BAT (9.01-02). Among these the CONFIG.SYS file has the most long history in previous versions of DOS. It defines a number of very important parameters and a set of software drivers to be loaded at boot time. Each line in CONFIG.SYS is a command to the IO.SYS loader (5.01-01). Interpretation of commands by the latter differs considerably from that by the most known command interpreter, the COMMAND.COM (6.04). There must be other commands and other syntax in CONFIG.SYS file.

Though several configuration commands (4.02, 4.24, 4.25) are synonymous to those executed by the COMMAND.COM interpreter, the IO.SYS loader deals with them in a different way. The loader doesn't allow to omit file's suffixes, doesn't execute redirections, doesn't substitute variable's values for their names. Command execution order in CONFIG.SYS depends not only on line's order, but also on command's priorities (see 4.15 and 4.25 for details). There is a group of commands (4.19, 4.20, 4.21, 4.29), which can be specified in menu and submenu blocks only, and other commands (except 4.23) in these blocks can't be used.

Several commands (4.01, 4.04, 4.11, 4.13, 4.18, 4.28) have no equivalents in previous versions of MS-DOS; some other commands (4.08, 4.30) have been changed in MS-DOS7, new parameters have been added. Inherited configuration commands, which invoke

loading procedures, in MSDOS7 are treated otherwise – as loading beyond conventional memory by default: DEVICE is executed just as DEVICEHIGH, BUFFERS – as BUFFERSHIGH, and so on. If a certain driver must be loaded into conventional memory, implicit defaults should be discarded by specifying the NOAUTO parameter in DOS command (4.08). These and other peculiarities of configuration commands and of their interpretation by IO.SYS loader are described in detail in articles below.

4.01 ACCDATE – registration of last access date

The ACCDATE command (ACCess DATE) enables or disables writing the date of last access into a directory entry (A.09-1), related to the accessed file. By default the last access date is registered in hard disk drives, but is not written on floppies. In a command line after the name of ACCDATE command you may specify any number of disk's letter-names with following plus sign (= permit registration) or minus sign (= prohibit registration), for example:

```
ACCDATE C+ D- E- R-
```

When access date registration is disabled, disk access operations are performed faster.

4.02 BREAK – disk access intercept control

The BREAK command affects the state of a binary flag, which controls disk access intercepts. By default the BREAK flag is turned off, and then the BREAK and CTRL-C keystrokes (1.03) are checked only during the CON driver addressing operations. Hence the user can suspend or terminate execution of current procedure only when the latter addresses the CON driver: sends output to screen or waits for input from keyboard.

The BREAK flag can be turned on with command:

```
BREAK ON
```

Since that moment the BREAK and CTRL-C keystrokes will be checked during disk access operations too. This gives an opportunity to suspend or to terminate execution of current procedure at the moments it addresses disk drives. Additional check makes disk operations a little slower. To disable the check the BREAK flag should be turned off with command:

```
BREAK OFF
```

After successful termination of IO.SYS loader's mission the BREAK command remains supported by command interpreter COMMAND.COM (3.01).

4.03 BUFFERS – number of buffers

Command BUFFERS reserves memory for buffers each 512 bytes long, which serve as a cash for sectors read from disks. By default MS-DOS7 arranges 30 primary buffers and 0 secondary buffers. The BUFFERS command enables to create from 1 to 99 primary buffers and from 0 to 8 secondary buffers. Secondary buffers are needed when double buffering should be arranged by the DBLBUFF.SYS driver (5.06-02). For example, command

```
BUFFERS=12,6
```

reserves 9 kb of memory for 12 primary and 6 secondary buffers. Disk reading and writing operations may become slow when number of buffers is less than 30. But when the SMARTDRV.EXE driver (5.06-01) is installed, then the number of buffers can be reduced to 10.

Note 1: by default the buffers are created beyond the 640 kb boundary of conventional memory, but may be arranged below 640 kb, if in DOS command (4.08) the NOAUTO parameter is specified, and also if the address space in UMB region is insufficient or unavailable (availability conditions – in 4.07). In any case no error message will be displayed.

Note 2: in some computers the DMA controller can't provide access to UMB region or to its part, though the whole UMB region is opened by UMBPCI.SYS driver (5.04-04). In such computers it's better to arrange buffers outside UMB region: in conventional memory or in a space beyond 1088 kb, opened by EMM386.EXE driver (5.04-02). Sometimes this problem can be solved by auxiliary LOWDMA.SYS driver, supplied together with UMBPCI.SYS.

4.04 BUFFERSHIGH – number of buffers in UMB address space

The BUFFERSHIGH command is almost equivalent to BUFFERS command (4.03), except that BUFFERSHIGH command attempts to arrange buffers beyond conventional memory despite presence of NOAUTO parameter in DOS command (4.08). All other information in article 4.03 is equally applicable to BUFFERSHIGH command.

4.05 COUNTRY – loading national adaptation data

The COUNTRY command initializes selective copying of national adaptation data from COUNTRY.SYS data file (5.02-01) into internal DOS's data tables (A.02-4, A.02-5). Thus DOS settings become adapted to localized rules of a particular country. Besides other features, national adaptation enables access to files and directories having specific national characters inside their names. Here is an example of a line with COUNTRY command:

```
COUNTRY=007,866,C:\DOS\DRV\Country.sys
```

where:

007 – country code, in particular for Russia
866 – number of codepage with Russian character set
C:\DOS\DRV\ – example of a path to Country.sys file

Note 1: for other country codes and codepage numbers see appendix A.02-2

4.06 DEVICE – loading a device driver

The DEVICE command is used to load those drivers, which have a header of special format (A.05-1) and should be loaded into memory when DOS's system structure arrangement is not finished yet. Most often (but not necessarily) these drivers are given the *.SYS suffix. Drivers with *.COM and *.EXE suffixes may have no special header, and then such drivers should be loaded not by DEVICE command, but by INSTALL command (4.15).

Here is an example of a line with DEVICE command loading a driver with *.SYS suffix:

```
DEVICE=C:\DOS\DRV\Himem.sys /EISA /V
```

where:

Himem.sys – an example of driver's name
C:\DOS\DRV\ – an example of a path to the driver
/EISA /V – an example of a parameters group for the driver; it must conform to parameters specifications for this particular driver.

Here is one more example of a DEVICE command loading another driver:

```
DEVICE?=\DOS\DRV\Emm386.exe RAM /V
```

Besides the driver itself, the latter string presents two differences. First, the path (\DOS\DRV\) without preceding disk's letter-name is suitable for loading from any disk, even when the disk's letter-name is not known beforehand. The second difference is an optional question mark "?", appended to the DEVICE command. This mark forces the IO.SYS loader to suspend further execution and to display the line, followed by a query whether to load the specified driver or not:

```
[Enter=Y, Esc=N]?
```

Thus the DEVICE command may be used to compose selective loading configurations.

Note 1: by default the drivers are loaded beyond conventional memory (above 640 kb), but may be loaded below 640 kb, if in DOS command (4.08) the NOAUTO parameter is specified, and also if address space in UMB region is insufficient or

unavailable (availability conditions – in 4.07). In any case no error message will be displayed.

Note 2: DEVICE command can't be involved in address space optimization procedure in UMB memory region. If this feature is significant, the DEVICEHIGH command (4.07) should be used instead.

4.07 DEVICEHIGH – loading a driver via UMB address space

The main purpose of DEVICEHIGH command is almost the same as that of the DEVICE command (4.06), but DEVICEHIGH command attempts to load drivers beyond conventional memory despite presence of NOAUTO parameter in DOS command (4.08). Of course, the UMB region address space must be made accessible beforehand. Therefore the following conditions must be met:

– computer must be equipped with 80386 or newer processor;
– the UMB parameter must be specified in DOS command (4.08);
– HIMEM.SYS driver (5.04-01) must be loaded yet by DEVICE command;
– either EMM386.EXE (5.04-02) or UMBPCI.SYS (5.04-04) driver must be loaded yet by DEVICE command in the following line of CONFIG.SYS file.

When DEVICEHIGH command is used in its simplest form, for example

```
DEVICEHIGH=C:\DOS\DRV\Setver.exe
```

the specified driver (Setver.exe) will be loaded so that it could be addressed via that part of UMB region, which has the largest free block of address space, provided this free block is sufficient for the specified driver.

Just as DEVICE command, the DEVICEHIGH command can be appended by a question mark: DEVICEHIGH?=... . It will cause this line to be displayed, then execution will be suspended with an offer for the user to decide whether the shown driver is to be loaded or not.

The DEVICEHIGH command gives an opportunity to specify a particular area of UMB address space for access to the loaded driver, for example:

```
DEVICEHIGH /L:1,15792 =C:\DOS\DRV\Display.sys CON=(EGA,,1)
```

where:

/L:1 – an example of UMB address space area number: it may be found from listing, shown by the MEM.EXE utility (6.17), being executed with /F parameter.

15792 – optional size of address space to be devoted to the specified driver (generally it is not equal to driver's file size).

If a particular driver is composed of several parts, which can be addressed via different areas of UMB address space, then several address space areas may be allocated in one command, with or without size specification for each area, for example

```
/L:2;3
```

or else

```
/L:2,12192;3,3600
```

Note, that size is preceded by a comma, and different area specifications are separated by a semicolon.

If size of the devoted area is specified, then the DEVICEHIGH command can accept optional /S parameter:

```
DEVICEHIGH /L:1,35008 /S =C:\DOS\DRV\MOUSE.SYS
```

The /S parameter means that allocated UMB block should be truncated to specified size. This leads to the most efficient address space usage, but doesn't guarantee from a crash, if size specification is not quite correct. Both block size and the /S parameter shouldn't be specified unless a memory allocation optimization procedure is performed by the MEMMAKER.EXE utility (5.04-03). As a result of this procedure both /S and /L parameters together with exact area specifications will be automatically written into each line with DEVICEHIGH command.

Note 1: if address space in UMB region is insufficient or unavailable, then the DEVICEHIGH command will load driver(s) into conventional memory (below 640 kb), and no error message will be displayed.

4.08 DOS – introduction of DOS's loading options

By default the core of MS-DOS7 is loaded into conventional memory. If the HIMEM.SYS driver is already installed, the DOS's core may be loaded into high memory region 1024 – 1088 kb. For this purpose the CONFIG.SYS file must contain a line with command

```
DOS=HIGH
```

Furthermore, if UMB address space is opened yet by either the EMM386.EXE driver (5.04-02) or by UMBPCI.SYS driver (5.04-04), then MS-DOS7 may be allowed to use UMB space for addressing DOS's system structures and drivers. This is achieved by command:

```
DOS=UMB
```

In MS-DOS7 one more optional parameter NOAUTO is introduced. It means that the IO.SYS loader must disable its defaults for implicit loading of several drivers (HIMEM.SYS, DBLBUFF.SYS, IFSHLP.SYS, DBLSPACE.SYS) as well as for loading beyond 640 kb with ordinary loading commands DEVICE, INSTALL and some others

(4.03, 4.06, 4.10, 4.12, 4.15, 4.17, 4.27). In fact the NOAUTO parameter enables to configure MS-DOS7 as a separate operating system. All parameters of the DOS command may be specified in one line:

```
DOS=HIGH,UMB,NOAUTO
```

Note 1: the DOS command can accept one more parameter SINGLE, which enables to load MS-DOS7 when otherwise the Windows-95/98 OS were loaded. But this way to load MS-DOS7 is accompanied with inappropriate questions to the user and entails increased risk to fall into reboot. Therefore other methods of loading MS-DOS7 should be preferred, those which are enlisted in article 1.03.

4.09 DRIVPARM – replacement of drive's parameters

The DRIVPARM command (DRIVe PARaMeters) is a means to provide access to storage media in those devices, which can't be identified properly by PC's BIOS. In fact those devices are implied, which were known yet to MS-DOS7 at the moment of it's release in 1996, but were not known to BIOS of obsolete computers, produced in early 1990-ties or even earlier. Here is an example of DRIVPARM command usage for providing access to a 3,5-inch floppy drive in an old PC with BIOS support only for 5,25-inch floppy drives:

```
DRIVPARM /D:1 /c /f:7 /h:2 /i /s:18 /t:80
```

where:

- /D:1 – specifies physical drive number, "1" means drive B:, "0" should be used for drive A:, "2" - for drive C:, and so on.
- /c – optional parameter, enabling removable media change detection. For non-removable media the "/n" parameter should be specified instead.
- /f:7 – defines type number of the drive:
  - 0 – 160/180/320/360 kb 5,25 inch drive;
  - 1 – 1.2 Mb 5,25-inch drive;
  - 2 – 720 kb 3.5-inch drive;
  - 5 – hard disk drive;
  - 6 – magnetic tape device (streamer);
  - 7 – 1.44 Mb 3.5-inch drive;
  - 8 – optical disk drive;
  - 9 – 2.88 Mb 3.5-inch drive.
- /h:2 – defines number of heads, default is 2 for double-sided diskettes.
- /i – support for 3.5-inch drives if these are not supported by BIOS.
- /s:18 – specifies number of sectors per track
  - 8 – for old 320 kb 5.25-inch diskettes;
  - 9 – for 360 kb and 720 kb diskettes;
  - 15 – for 1.2 Mb 5.25-inch diskettes;

18 – for 1.44 Mb 3.5-inch diskettes.

/t:80 –specifies number of tracks

40 – for old 360 kb diskettes;

80 – for 720 kb and 1.44 Mb diskettes.

Note 1: default settings for /f and /s parameters correspond to 5.25-inch diskettes with 9 sectors per track.

4.10 FCBS – number of file control blocks

The FCBS command (File Control BlockS) reserves memory for a specified number of file control blocks – from 1 to 255, each 80 bytes long. FCBS is an obsolete form, providing access to open files in current directory only. FCBS can't be applied to media having FAT32 file system. Modern DOS programs use "file handles" (4.12) instead of FCBS. Nevertheless MS-DOS7 supports FCBS, because they are used by some old-fashioned programs and network services (INTERLNK.EXE, SHARE.EXE, etc.). In most cases the default value

```
FCBS=4
```

is quite enough.

Note 1: by default a place for FCBS is reserved beyond conventional memory (above 640 kb), but may be reserved below 640 kb, if in DOS command (4.08) the NOAUTO parameter is specified, and also if the address space in UMB region is insufficient or unavailable (availability conditions - in 4.07). In any case no error message is displayed.

Note 2: the FCBS specification doesn't limit available number of "unopened" file control blocks (A.09-5), which are used inside PSP (A.07-1), and also by some file search procedures.

4.11 FCBSHIGH – file control blocks in UMB address space

The FCBSHIGH command is almost equivalent to FCBS command (4.10), except that FCBSHIGH command attempts to arrange file control blocks beyond conventional memory despite presence of NOAUTO parameter in DOS command (4.08). All other information in article 4.10 is equally applicable to FCBSHIGH command.

4.12 FILES – reservation of SFT entries

The FILES command reserves address space for a specified number of SFT table entries (A.01-4). Each entry defines state of an opened object – a file or a channel, and also defines association of this object with its numerical reference, the so called "handle" used

to address the object. By default there is 60 entries in SFT. This number of entries is often excessive. For ordinary work you may need

```
FILES=30
```

SFT with 30 entries takes about 1800 bytes. For operating with large databases the number of SFT entries should be increased to 40.

Note 1: by default a place for SFT entries is reserved beyond conventional memory (above 640 kb), but may be reserved below 640 kb, if in DOS command (4.08) the NOAUTO parameter is specified, and also if address space in UMB region is insufficient or unavailable (availability conditions – in 4.07). In any case no error message is displayed.

4.13 FILESHIGH – number of SFT entries in UMB address space

The FILESHIGH command is almost equivalent to FILES command (4.12), except that FILESHIGH command attempts to arrange SFT entries beyond conventional memory despite presence of NOAUTO parameter in DOS command (4.08). All other information in article 4.12 is equally applicable to FILESHIGH command.

4.14 INCLUDE – reference to a block of commands

The INCLUDE command inserts a named block of commands into general succession of configuration commands to be executed. The inserted block of commands may be placed elsewhere inside the same CONFIG.SYS file, but it must be announced in its first line by a unique block name – a word or a number, enclosed in square brackets, for example, [L055] (example is taken from 9.09-01). End of the block must be marked with similar line, containing a name of the next block. If there is no need to specify other blocks, a line with reserved name [common] should be specified afterwards. All commands following a line with this name (if there are any) will be executed in each specified configuration.

In order to execute commands, in particular, of the block [L055], in the desired position in CONFIG.SYS file there must be a line

```
INCLUDE=L055
```

Note that name of the block should be specified to the right of equality sign without enclosing square brackets.

If the same block of commands should be executed in several loading configurations, then each configuration specification must contain identical lines with INCLUDE command. Segregation of repeated command sequences into separate blocks makes the structure of CONFIG.SYS file simpler and more clear. Examples of such CONFIG.SYS file's structures are shown in articles 9.04-01, 9.09-01 and 9.11-03.

4.15 INSTALL – loading a TSR executable

The INSTALL command is used to load program's resident modules and those drivers, which have no special header (A.05-1) and therefore can't be loaded with DEVICE (4.06) or DEVICEHIGH (4.07) commands. Most often such drivers are marked with suffix *.COM or *.EXE. Drivers and utilities to be loaded with INSTALL command must meet the following conditions:

– they must not need environment space;
– they must not send calls for handling critical errors;
– they must not rely on services of COMMAND.COM interpreter, which is not loaded yet at that time.

The mentioned conditions are satisfied, though, by a large part of those drivers and TSR utilities, which are designed to be loaded from AUTOEXEC.BAT file or from command line. Loading from CONFIG.SYS file with INSTALL command is considered more reliable and takes a little less memory space.

Lines with INSTALL command are interpreted after all lines with DEVICE, DEVICEHIGH and SET commands, but before the line with SHELL command, even if succession of lines in CONFIG.SYS file is different. Contradiction between sequence of lines and sequence of execution can cause confusion and therefore should be avoided. Preferable line's order must correspond to sequence of execution.

Here is an example of a line with INSTALL command:

```
INSTALL=\DOS\DRV\Mkecdex.com /B /L:0
```

In the shown line to the right of equality sign there are path to the driver, driver's name and a group of its parameters. Just like the DEVICE command (4.06), the INSTALL command may be appended with question mark (INSTALL?=...), enabling the user to see this line on the screen and make a choice ([Enter=Y, Esc=N]) about whether to execute it or to skip.

The INSTALL command can be used for temporary loading of those modules, which are to be uninstalled after termination of their mission. Peculiarity of such operations is that DOS automatically releases occupied memory space within conventional memory only, below 640 kb. When the NOAUTO parameter is specified in DOS command (4.08), then the INSTALL command will load modules into conventional memory, and then, therefore, you can afford temporary loading of command interpreter module in order to suspend execution for a while and give an opportunity to read displayed messages:

```
INSTALL=C:\Command.com /low /c pause
```

One more example of command interpreter's temporary loading with INSTALL command is shown in article 9.09-01.

Note 1: unlike the DEVICEHIGH command, INSTALL command is not involved in memory allocation optimization procedure, performed by MEMMAKER.EXE

(5.04-03). Because of this reason the INSTALL command doesn't allow to affect memory allocation with auxiliary parameters /L and /S (4.07).

Note 2: by default the drivers and TSR modules are loaded beyond the conventional memory (above 640 kb), but may be loaded below 640 kb, if in DOS command (4.08) the NOAUTO parameter is specified, and also if address space in UMB region is insufficient or unavailable (availability conditions – in 4.07). In any case no error message is displayed.

4.16 INSTALLHIGH – loading a TSR via UMB address space

The INSTALLHIGH command is almost equivalent to INSTALL command (4.15), except that INSTALLHIGH command attempts to load drivers and TSR modules beyond conventional memory despite presence of NOAUTO parameter in DOS command (4.08). The INSTALLHIGH command shouldn't be used for loading those modules, which are to be uninstalled after termination of their mission. All other information in article 4.15 is equally applicable to INSTALLHIGH command.

4.17 LASTDRIVE – reservation of CDS entries

The LASTDRIVE command defines number of entries in DOS's CDS table (A.03-03). Entries in CDS table store names of current directories for logical disks, both real and virtual. At boot time MS-DOS7 creates one valid CDS entry record per each logical disk, identified by PC's BIOS system, and then appends CDS structure with dummy entries (reservations). The last entry corresponds to the last disk's letter-name, specified by LASTDRIVE command. By default MS-DOS7 assumes

```
LASTDRIVE=Z
```

Such CDS table with 26 entries occupies 2288 bytes. If you consider this excessive, you may specify other disk's letter-name by LASTDRIVE command, but in any case there must be enough CDS entries for all logical disks, including those which will be made accessible later, after installation of CD-ROM drivers, network services, etc.

Note 1: by default CDS table is arranged beyond conventional memory (above 640 kb), but may be arranged below 640 kb, if in DOS command (4.08) the NOAUTO parameter is specified, and also if address space in UMB region is insufficient or unavailable (availability conditions – in 4.07). In any case no error message is displayed.

4.18 LASTDRIVEHIGH – CDS entries in UMB address space

The LASTDRIVEHIGH command is almost equivalent to LASTDRIVE command (4.17), except that LASTDRIVEHIGH command attempts to arrange CDS table beyond conventional memory despite presence of NOAUTO parameter in DOS command (4.08). All other information in article 4.17 is equally applicable to LASTDRIVEHIGH command.

4.19 MENUCOLOR – menu color choice

The MENUCOLOR command may be used optionally only in those blocks of configuration commands, which are named [menu] or are announced as submenu in their parent menu or submenu. Color palette for each menu or submenu may be set by a separate MENUCOLOR command, specified in this menu or submenu. The default color settings are equivalent to command:

```
MENUCOLOR=7,0
```

In the shown example the first digit means code of text color (7 – white), and the second digit (after comma) means code of background color (0 – black). You are allowed to choose other colors. All permissible color codes together with their meanings are enlisted in appendix A.10-5.

4.20 MENUDEFAULT – choice of default menu item

The MENUDEFAULT command may be used only in those blocks of configuration commands, which are named [menu] or are announced as submenu in their parent menu or submenu. The MENUDEFAULT command is placed in the last line of menu block and defines that menu item, which should be chosen automatically if the user has not made his choice during a prescribed time, for example:

```
MENUDEFAULT=L007,20
```

In the shown example the first group of characters just after equality sign means a name of configuration block [L007], which is to be chosen by default, and a number after comma means 20 seconds waiting for the user's choice before automatic default choice will be made. Delays from 0 to 99 seconds are allowed. Examples of configuration menu with MENUDEFAULT command are shown in articles 9.04-01, 9.09-01, 9.11-03.

4.21 MENUITEM – specification of menu entry

The MENUITEM command may be used within the first menu block of commands (named [menu]) in CONFIG.SYS file and in those blocks, which are announced as

submenu in their parent menu or submenu. Each alternative, presented in menu or in submenu, must be linked with corresponding block of configuration commands and must be given an intelligible title. This is just what is done by MENUITEM command, for example:

```
MENUITEM=L007, Relocate DOS to 5600 kb RAM-disk R:
```

In the example above the "L007" is a name of a block of commands (taken from 9.09-01), which must be present in CONFIG.SYS file and must be preceded by a header line with the same name [L007] in square brackets (2.03-05). The rest part of the shown command line (after comma) is a title text, representing this menu entry on the screen. Title text may include spaces between words, but square brackets [ ], semicolons (;) and slashes (/ \ ) are not allowed.

During interpretation of menu block the IO.SYS loader creates an environmental variable CONFIG and assigns as its value the name of the chosen configuration block (L007 in the shown example). This value may be used later in order to adapt execution of AUTOEXEC.BAT file (or any other batch file as well) according to the chosen configuration.

Note 1: total number of MENUITEM and SUBMENU (4.29) commands in each menu block must not exceed 9.

4.22 MULTITRACK – drive addressing mode

For access to a disk, DOS must specify starting sector and a number of sectors to be read (or written). In obsolete PCs produced in early 1980-ties disk drives and BIOS versions performed access operations within one track at a time, so that the sum of starting sector number and the number of sectors to be accessed could not exceed total number of sectors on a track (otherwise the process was "wrapped" to the beginning of the same track). Being forced to cope with obsolete hardware, MS-DOS7 also couldn't afford multitrack addressing, and then the CONFIG.SYS file must contain a line

```
MULTITRACK OFF
```

Since late 1980-ties all disk drives and BIOS versions are able to prevent the mentioned wrapping: access automatically is switched to the next track. This gives an opportunity to address several tracks in one operation and makes disk access much faster. MS-DOS7 performs multitrack access and in fact takes MULTITRACK ON option as default. Therefore now the MULTITRACK command is almost always omitted.

4.23 NUMLOCK – numerical keypad state control

The NUMLOCK command (NUMerical keypad LOCK) defines status of numerical keypad (at the right edge of standard keyboard). Most often the NUMLOCK switch is kept

OFF, and then the numerical keypad keys duplicate functions of keys in main part of keyboard (arrows, PgUp – PgDn, etc.). If other status should be avoided, the CONFIG.SYS file must contain command

```
NUMLOCK OFF
```

In order to enable input of digits and arithmetic symbols the NUMLOCK status should be reversed with command

```
NUMLOCK ON
```

Either state of NUMLOCK switch is suitable for choosing items in configuration menu. While NUMLOCK is OFF, you may scroll selection up-down as with arrow keys, but when NUMLOCK is set ON, you may select items in configuration menu by their number, entered via numerical keypad.

Note 1: NUMLOCK switch state affects key codes, returned by INT 16\AX=10h (8.01-83). More about this – in note 6 to appendix A.02-1.

4.24 REM – remark line

REM command (REMark) forces the IO.SYS loader to ignore all following words up to the end of line. Main purpose of REM command is to provide an opportunity to specify comments, which should not be displayed on the screen. Insertion of the REM command at the start of any command line enables to skip this line during interpretation of CONFIG.SYS file.

Note 1: during interpretation of lines in CONFIG.SYS file the IO.SYS loader doesn't provide those extra opportunities, which are provided by the COMMAND.COM interpreter for its synonymous REM command (3.24).

4.25 SET – value assignment to a variable

The SET command in CONFIG.SYS file is used to assign or to redefine a value of an environmental variable, for example:

```
SET  Var_Name=New_Var_Value
```

where:

Var_name – a name example of environmental variable; it must begin with a letter and may contain digits.

New_Var_Value – an example of a string value for the named environmental variable; it must not contain equation sign(s) inside. If there are spaces in the value, preceding the value or following the value (up to the end of line mark), all these spaces will be included in the assigned value.

The SET command, executed by IO.SYS within CONFIG.SYS file, causes a bit different effect than synonymous command (3.25), executed by COMMAND.COM interpreter from command lines or in batch files. The differences are:

– when used without following variable's parameters, the SET command in CONFIG.SYS file doesn't display the current environment.

– in CONFIG.SYS file the SET command, appended with a question mark (SET?=...), causes execution to stop with a query [Enter=Y, Esc=N], thus enabling the user to make a choice on whether to execute this line or not.

– SET commands in CONFIG.SYS file are performed after all DEVICE and DEVICEHIGH commands, but before execution of INSTALL, INSTALLHIGH and SHELL commands. This is a reason to prefer this order of lines in CONFIG.SYS file.

– aliases (2.03-03), redirections (2.04-02 – 2.04-05) and value substitutions in CONFIG.SYS file are not performed. Hence any symbol of redirection and substitution can be included in a value of a variable, assigned by the SET command in CONFIG.SYS file.

4.26 SHELL – interpreter shell specification

The SHELL command is used to launch an executable file, which will not return control back to IO.SYS loader. This is why SHELL command is executed the last in CONFIG.SYS file, and preferably should be specified in its last line. Executables loaded with the SHELL command may be loaders of other operating systems (as LOADLIN.EXE for LINUX OS) or command interpreters, which are to take control over PC after the IO.SYS loader finishes its job. Here is an example of control transfer to command interpreter NDOS.COM:

```
SHELL=C:\DOS\NU\Ndos.com /f @C:\DOS\NU\Ndos.ini
```

where:

C:\DOS\NU\ – is an example of a path to command interpreter file;

/f @C:\DOS\NU\Ndos.ini – an example of parameters string for NDOS.COM command interpreter.

When the SHELL command isn't present in CONFIG.SYS file, the IO.SYS loader attempts to find the MS-DOS's proprietary command interpreter – COMMAND.COM – in the root directory of the current disk. In this case the COMMAND.COM interpreter will be launched with default parameters. It is better, though, to specify parameters explicitly, for example:

```
SHELL=Command.com A:\ /e:1008 /p
```

In the latter example absence of a path before interpreter's name implies its presence in the current directory. Parameter's assignments for COMMAND.COM interpreter are described in detail in 6.04. Other examples of control transfer to the COMMAND.COM interpreter are shown in articles 9.01-01, 9.04-01 and 9.09-01.

4.27 STACKS – number of auxiliary stacks

The STACKS command in CONFIG.SYS file specifies number of auxiliary DOS's stacks used for treatment of nested interrupts. Parameters of STACKS command define a number of auxiliary stacks and amount of address space devoted for each stack. The default parameters are equivalent to command

```
STACKS=9,256
```

where:

9 – number of auxiliary stacks (from 8 to 64 and 0 are allowed);
256 – size of each auxiliary stack in bytes (from 32 to 512 and 0 are allowed).

Since each stack overflow fault forces to reboot the PC and may cause data loss, the actual size of auxiliary stacks should not be less than the default value.

Note 1: by default auxiliary stacks are arranged beyond the conventional memory (above 640 kb), but may be arranged below 640 kb, if in DOS command (4.08) the NOAUTO parameter is specified, and also if address space in UMB region is insufficient or unavailable (availability conditions – in 4.07). In any case no error message is displayed.

4.28 STACKSHIGH – number of stacks in UMB address space

The STACKSHIGH command is almost equivalent to the STACKS command (4.27), except that STACKSHIGH command attempts to arrange auxiliary stacks beyond conventional memory despite presence of NOAUTO parameter in DOS command (4.08). All other information in article 4.27 is equally applicable to STACKSHIGH command.

4.29 SUBMENU – introduction of a submenu

The SUBMENU command announces a menu entry, just as the MENUITEM command (4.21), but unlike the latter, the SUBMENU command forces to treat the corresponding block of commands as a submenu. Of course, the SUBMENU command itself may be used only in those blocks of configuration commands, which are named [menu] or are announced as submenu in their parent menu or submenu, for example:

```
SUBMENU=6000, Relocation to RAM-disk
```

In the shown example "6000" means a name of corresponding configuration block, and words after comma denote title text for the proposed menu entry. This text is subjected to the same restrictions as text in MENUITEM command (4.21). Submenu configuration block (named [6000] in the shown example) must have the same structure as the main menu block and may include up to 9 entries, each represented by a separate line with MENUITEM or SUBMENU command. The main specific feature of submenu block is its name: it must be unique and must differ from reserved names [menu] and [common].

4.30 SWITCHES – supplementary options

The SWITCHES command enables to specify up to four optional configuration items, for example:

```
SWITCHES= /K /N /F /E:64
```

where:

- /K – enables old programs, consigned for 86-key keyboard, to cope with newer "enhanced" 101/108-key keyboard.
- /N – disables opportunity to skip execution of configuration files (CONFIG.SYS and AUTOEXEC.BAT) with F5 and F6 keys.
- /F – excludes two-second delay after displaying the message "Starting WINDOWS...".
- /E:64 – allots 64 bytes of conventional memory (from 48 to 1024 bytes allowed) to a "handle" for EBIOS, which is BIOS system extension, enabling LBA access to HDDs (see note 4 to A.13-6). If the number after /E parameter is omitted, then the whole EBIOS code, if it will be found necessary, will be loaded into conventional memory. In modern PCs the /E parameter is not needed, because LBA access is supported by their main BIOS system.

Chapter 5 Selected drivers for MS-DOS7

Drivers are files with executable resident code inside. Resident code is the code adapted for being written into RAM (random accessed memory) and left there, waiting for its chance of being requested on certain occasion(s). When this particular occasion happens, driver's code is executed, performs its mission and then again is left waiting for the next request. This mode of action is similar to the "life" of operating system's core. MS-DOS combines a limited number of main core's functions with various functional extensions, provided by drivers. Proper choice and renewal of drivers is an important factor for DOS's survival amongst ever changing PC's hardware.

Drivers may be presented in a form of files with special header (A.05-1), most often marked with *.SYS suffix, or in a form of ordinary executable files (*.COM or *.EXE) having a TSR (Terminate and Stay Resident) part. Drivers with *.SYS suffix must be loaded by DEVICE (4.06) or DEVICEHIGH (4.07) commands from lines of configuration file CONFIG.SYS. Composition examples for CONFIG.SYS file are shown in articles 9.01-01, 9.04-01 and 9.09-01. Loading by DEVICE or DEVICEHIGH commands gives more chances to affect DOS system structures construction, because it is not finished yet at that moment.

Drivers with *.COM and *.EXE suffixes usually are loaded later either from CONFIG.SYS file with INSTALL (4.15) or INSTALLHIGH (4.16) commands, or from AUTOEXEC.BAT file (9.01-02, 9.04-02, 9.09-02), or from command line - directly or with LH command (3.17). Loading from CONFIG.SYS file is less subjected to mutual software interference and therefore is considered more safe. On the other hand, INSTALL and INSTALLHIGH commands can't be involved in memory optimization procedure by MEMMAKER.EXE optimizer. If memory optimization is significant, loading with LH command from AUTOEXEC.BAT file should be preferred.

The main group of drivers for MS-DOS7 constitute those supplied within WINDOWS-95/98 operating system release and on its rescue diskette. In case of standard operating system installation all drivers for MS-DOS7 are in directories \WINDOWS and \WINDOWS\COMMAND. But if MS-DOS7 is installed as an independent operating system, it's better to arrange a separate directory for drivers, for example, C:\DOS\DRV. This path is shown in the most part of presented examples. When you will intend to follow these examples in practice, it's important to remember: the path you specify must not be necessarily C:\DOS\DRV, it must be exactly the one that leads to each particular driver in your particular computer.

Beside "native" Microsoft's drivers, a lot of drivers for MS-DOS7 have been developed since 1995 by other software vendors and by manufacturers of PC's hardware. Existing

drivers are too numerous, and only a limited selection of them can be described here. The following text doesn't include descriptions of some other Microsoft's drivers (that can be found in MSDOSDRV.TXT, supplied with WINDOWS-95 software release) as well as of drivers for some non-common equipment (MO, PD and ZIP drives, LS120 floppies, streamers, etc.). Preference has been given to those drivers which are the most necessary for reparatory works.

5.01 DOS's system services

5.01-01 The core file IO.SYS and parameters file MSDOS.SYS

In the root directory of the disk used to load WINDOWS-95/98 or MS-DOS7, there are two hidden system files: IO.SYS and MSDOS.SYS. These files are there since installation of the operating system, they implement its loading. File IO.SYS combines the core of MS-DOS7 with interpreter and with DOS loader. The MSDOS.SYS file contains a set of loading parameters. If you have no such files, you may get them from a disc with WINDOWS-95/98 release, or else you may download them, for example, packed into archive DOS7.ZIP from internet site http://www.micosyen.com/msdos.php .

In order to make a disk bootable with MS-DOS7, presence of the mentioned system files is necessary, but not sufficient. Boot-sector's executable code will not "know" where control should be transferred, unless the name of loader is written into boot-sector. This is why copying of system files is combined with boot-sector updating in joint mission of SYS.COM utility (6.24).

Having got control over booting process, DOS loader reads loading parameters from MSDOS.SYS file. These parameters define the alternatives, described in article 1.02, and also define which operating system should be loaded: MS-DOS7 or WINDOWS-95/98. If the HRS attributes (H = Hidden, R = Read-only, S = system) of the MSDOS.SYS file are taken off by the ATTRIB.EXE utility (6.01), then MSDOS.SYS becomes an ordinary non-formatted textual file, which can be opened, changed and rewritten with editor utility, for example, with EDIT.COM (6.09).

Several or even all parameters may be not specified in MSDOS.SYS file, and then the omitted parameters will be given the default values. Contrary to other system files, MSDOS.SYS is not copied by SYS.COM utility, but rather is created empty anew, and this doesn't hamper normal loading of WINDOWS-95/98 operating system. At least some of the default parameter's values wouldn't fit, though, if you intend to load MS-DOS7 as a stand-alone operating system. Suitable values of all parameters are shown in an example of MSDOS.SYS file presented below.

```
[Paths]
WinDir=C:\WINDOWS
; path for environmental variables TMP, TEMP and PATH
WinBootDir=C:\WINDOWS
; path for environmental variable WINBOOTDIR
HostWinBootDrv=C
; announcement of the disk used to boot the PC
[Options]
Logo=0
; hide loading messages under logo (= 1) or not (= 0)
BootMenu=0
; display Windows's boot menu (= 1) or not (= 0)
BootMenuDelay=20
; delay in seconds of default menu item choice
BootMenuDefault=1
; menu item number to be chosen as default
BootKeys=1
; enable (= 1) or disable (= 0) the "hot keys" F5,
; Shift-F5, F6, F8 and Shift-F8, described in article 1.02
BootDelay=2
; waiting time (in seconds) for "hot key" keystroke
BootMulti=0
; disable the F4 key (1.02) for previous DOS version
BootWin=1
; load MS-DOS7 and Windows (= 1) or previous DOS (= 0)
BootSafe=0
; load Windows in ordinary (= 0) or in safe mode (= 1)
BootWarn=0
; don't warn about loading Windows in safe mode
BootGUI=0
; load Windows with its GUI (= 1) or MS-DOS7 (= 0)
LoadTop=0
; load Command.com and Dblspace.bin below 640 kb
AutoScan=0
; Scandisk.exe automatic launching conditions:
;       - never launch automatically (= 0),
;       - launch after each identified failure (= 1)
;       - launch during each PC loading procedure (= 2)
DBLSpace=0
DRVSpace=0
; don't load compression on-the-fly drivers. Otherwise
;  value = 1 may be specified for one such driver only
Network=0
```

```
; load (= 1) or don't load (= 0) network support
DoubleBuffer=0
; cancel default loading of Dblbuff.sys driver (5.06-02)
DisableLog=1
; don't write loading report into Bootlog.txt file.
```

Lines of the MSDOS.SYS file are read by an interpreter, integrated into the same IO.SYS file. The interpreter ignores all lines starting with semicolon, and this is why such lines are used to insert comments. Of course, lines with comments can be omitted, but there is one argument against it: for compatibility with obsolete antivirus programs the length of hidden system files must be not less than 1024 bytes. For most modern antivirus programs this restriction is not significant.

All parameter's values in the shown example of MSDOS.SYS file are compatible with variants of configuration files, presented in articles 9.01, 9.04, 9.09 and 9.11. Settings in the [PATHS] section are taken into account by WINDOWS operating system only; for MS-DOS7 the corresponding values of environmental variables should be ignored or reassigned later, during interpretation of the second configuration file - AUTOEXEC.BAT. A large part of parameters in the [OPTIONS] section can be omitted too. Nevertheless the shown complete list of parameters will help you to decide, whether any particular parameter should be specified and which value it should be given. Having prepared your own version of MSDOS.SYS file, don't forget to return back its original attributes HRS (Hidden, Read-only, System).

In accordance with the prepared parameters the DOS loader loads the core of MS-DOS7. The core is responsible for system DOS services and for main INT 21 handlers, described in part 8.02. The last loader's mission is execution of commands from configuration file CONFIG.SYS, which define loading of most drivers. Several variants of CONFIG.SYS file are shown in articles 9.01-01, 9.04-01 and 9.09-01. Later MS-DOS7 never calls the IO.SYS file for execution, but its presence is nevertheless necessary for copying each time, when you have to make bootable any other disk.

Note 1: in earlier versions of MS-DOS the MSDOS.SYS file was not a textual file, it contained the core of DOS and was loaded almost as an ordinary driver.

Note 2: In 2001 a bug has been revealed in the core of MS-DOS7: it didn't cope properly with HDD's LBA-errors. Therefore Microsoft issued a patched core file IO.SYS inside SFX-archive 311561usa8.exe. The latter can be downloaded from http://support.microsoft.com/kb/311561/en-us?spid=6519&sid=global . WINRAR version 3.2 (or higher) unpacks 311561usa8.exe as CAB-archive. Two subversions of IO.SYS file are hidden there under nicknames Winboot.98s and Winboot.98g. If the VER command (3.32) reports version 4.10.2222, then IO.SYS should be obtained by renaming Winboot.98s. If reported version is 4.10.1998, then Winboot.98g should be similarly renamed and used as IO.SYS.

5.01-02 DOS version number substitution: driver SETVER.EXE

Evolution of OS versions is complicated by a problem of application programs, which have been developed for previous versions of DOS. Compatibility of any application with future OS versions is a matter of faith rather than prediction. Microsoft suggested to solve this problem by informing definitely compatible application programs via INT 21\AH=30h (8.02-22) about not the actual, but the required DOS version number. SETVER.EXE is just that driver, which is charged with mission of deceiving application programs by substitution of DOS version number.

In case of standard installation of WINDOWS-95/98 the SETVER.EXE driver is in the \WINDOWS directory. But if you want to use MS-DOS7 separately, it's better to have a copy of SETVER.EXE in common directory with other DOS drivers. It has to be loaded by DEVICE (4.06) or DEVICEHIGH (4.07) command from a line of CONFIG.SYS file:

```
DEVICEHIGH=C:\DOS\DRV\Setver.exe
```

where:

C:\DOS\DRV\ – is a path example to SETVER.EXE driver stored in separate directory for DOS drivers.

Application program will receive the substituted DOS version number only if the name of this program together with the required DOS version number is in advance registered in internal versions table inside the loaded TSR module of SETVER.EXE driver. Though such substitution doesn't guarantee a proper outcome, nevertheless most old programs are able to operate properly in MSDOS7.

The SETVER.EXE driver can be launched from command line just as an ordinary utility, for example, in order to display its short help text:

```
Setver.exe /?
```

Being run from command line without parameters, SETVER.EXE driver displays its internal table of versions. Originally table of versions is not empty: its entries reflect Microsoft's recommendations. In order to append the table of versions with one more entry you should type

```
Setver.exe Qbasic.exe 6.22
```

where:

Qbasic.exe – name example of the utility to be "deceived". The name must have suffix *.COM or *.EXE.

6.22 – an example of required DOS version number.

Command to delete the same entry from internal table of versions requires the /D parameter and looks as

```
Setver.exe Qbasic.exe /D
```

After any operation with its internal table the SETVER.EXE driver leaves one of the following errorlevel values (more about errorlevels – in 3.15-03 and in 9.07-03):

0 – successful completion
1 – invalid command line switch specification
2 – invalid filename specification
3 – no memory enough to carry out the command
4 – invalid version number format
5 – specified entry isn't found in the table of versions
8 – too many parameters in command line
9 – one or more necessary parameters are missing
10 – an error detected while reading the table of versions
11 – the table of versions is corrupt
13 – no more space in table of versions for a new entry
14 – an error while writing SETVER.EXE with new table

All successful operations affecting contents of internal table are finished in the same way: former internal table on a disk is overwritten with its updated variant, containing changed entries. But writing the changes in a file on a disk is not enough to make them active. They will become active when the changed table of versions is transferred from a file into driver's TSR module, loaded into memory by DEVICE or by DEVICEHIGH command. This is why the changes come into effect only after PC's reboot.

5.02 National adaptation drivers

5.02-01 COUNTRY.SYS – specifications data file

In case of standard installation of WINDOWS-95/98 OS the COUNTRY.SYS file can be found in \WINDOWS\COMMAND directory. COUNTRY.SYS is in fact a set of data tables, one of which is to be loaded from CONFIG.SYS file with special COUNTRY command (4.05), for example:

```
COUNTRY=007,866,C:\DOS\DRV\Country.sys
```

where:

007 – country code (A.02-2), in particular for Russia;
866 – number of codepage (A.02-2), defining character set for Russia;
C:\DOS\DRV\ – path example to file COUNTRY.SYS, copied into common directory for DOS drivers.

Having been loaded, data from COUNTRY.SYS change several internal DOS settings, related to country-specific conventions on displaying time, dates, currency and punctuation

symbols, character sorting and national restrictions on character set for names (A.02-5). The latter feature is of special importance, since otherwise the files, having national characters in their names, may become inaccessible.

5.02-02 DISPLAY.SYS – character generator driver

DISPLAY.SYS driver prepares memory buffers for one or more national codepage tables, specifying character set and outline (A.02-2). Normally the DISPLAY.SYS driver can be found in \WINDOWS\COMMAND directory. This driver must be loaded by a DEVICE (4.06) or DEVICEHIGH (4.07) command from a line in CONFIG.SYS file:

```
DEVICE=C:\DOS\DRV\Display.sys CON=(EGA,866,2,1)
```

where:

C:\DOS\DRV\ – example of a path to DISPLAY.SYS driver, copied into special directory for DOS drivers.

CON – (console) – specification of display as output device (no alternative).

EGA – means that PC is equipped with EGA, VGA or SVGA video card; alternatives to EGA type are:

- LCD – video module used in obsolete portable PCs (notebooks);
- CGA – obsolete color video card without codepage switching;
- MONO – obsolete monochrome MDA video card, also without codepage switching.

When type of video card is omitted, then DISPLAY.SYS driver will try to determine it, but this leaves a chance for mistake.

866 – number of primary codepage (A.02-2), this one is in particular for Russia. Several codepages are supplied in each of the files EGA.CPI, EGA2.CPI, EGA3.CPI and ISO.CPI. Later the MODE.COM utility (6.18) enables to select proper codepage and write it into memory buffer prepared by DISPLAY.SYS driver.

2 – number of memory buffers to be prepared for character sets, besides the one prepared for the primary character set, specified by preceding codepage number. From 0 to 6 auxiliary buffers are allowed for EGA, VGA and SVGA video cards, no more than 1 – for LCD video modules, and only 0 – for CGA and MDA video cards.

1 – number of hardware supported fonts for each codepage. This number may be omitted together with preceding comma. Default is 1 for LCD type and 2 for EGA, VGA and SVGA video cards.

Note 1: all parameters in parenthesis may be omitted (brackets left empty), and then DISPLAY.SYS driver will appoint default settings.

Note 2: TSR module of DISPLAY.SYS driver is opened for interaction with programs via INT 2F\AX=AD00-AD03h (8.03-26, 8.03-27).

5.02-03 NLSFUNC.EXE – codepage switch

The NLSFUNC.EXE driver activates CHCP command (3.04) for switching between different codepages, coordinated with changing of other national settings. Normally this driver can be found in \WINDOWS\COMMAND directory. The NLSFUNC.EXE driver may be loaded directly or with LH command (3.17) from command line or from AUTOEXEC.BAT file, or else from CONFIG.SYS file with INSTALL (4.15) or INSTALLHIGH (4.16) command, for example:

```
INSTALLHIGH=C:\DOS\DRV\Nlsfunc.exe C:\DOS\DRV\Country.sys
```

where:

C:\DOS\DRV\ – path example to NLSFUNC.EXE driver, copied into common directory for DOS drivers.

C:\DOS\DRV\Country.sys – path and name examples for a file with national specifications data (5.02-01).

Note 1: switching between american english and any other national notation doesn't imply changing codepages: american english notation is accessible within any single national codepage (A.02-02).

Note 2: switching between different national codepages can be performed by MODE.COM utility too (6.18-03). The latter is usually preferred, because MODE.COM is not a TSR program and releases occupied memory after termination.

5.02-04 KEYB.COM – keyboard driver

Control over keyboard layouts is provided by Microsoft's keyboard driver KEYB.COM. Normally it can be found in \WINDOWS\COMMAND directory. The KEYB.COM driver can be loaded directly or with LH command (3.17) from AUTOEXEC.BAT file, or else from CONFIG.SYS file with INSTALL (4.16) or INSTALLHIGH (4.17) command, for example:

```
INSTALL=C:\DOS\DRV\Keyb.com UK,850,C:\DOS\DRV\Keybrd3.sys /E /ID:168
```

where:

C:\DOS\DRV\ – path example to KEYB.COM driver, copied into common directory for DOS drivers.

UK – example of a two-letter national keyboard layout code (other layout codes - in appendix A.02-2).

850 – example of national codepage number (A.02-2). When it is specified just here, it will not be changed automatically following display codepage change by CHCP command. If synchronous change of codepages (on both display and keyboard) is preferable, then codepage

number here should be omitted, but both enclosing commas must remain intact (. . .UK, ,C:\DOS\DRV\Keybrd3.sys. . .).

C:\DOS\DRV\KEYBRD3.SYS – path and name examples for keyboard data file. Each such file contains several national layout tables (A.02-2).

/E – this parameter specifies that keyboard layout should be adapted to "enhanced" 101/108-key keyboard.

/ID:168 – identifier of a certain keyboard subtype, needed for those countries only, where more than one keyboard layout is used (A.02-2). Most countries, including Russia, use only one keyboard layout, and then the subtype identifier /ID should be omitted.

When loaded, KEYB.COM driver activates the following "hot" key combinations:

CTRL-RightSHIFT – to type symbols 128-255, specific for each selected national codepage;

CTRL-LeftSHIFT – to type symbols 032-127 (english letters, digits and punctuation symbols), common for all codepages;

CTRL-ALT-F1 – to activate original american codepage 437 (selected national codepage becomes disabled);

CTRL-ALT-F2 – to return from american codepage 437 back to selected national codepage;

CTRL-ALT-F7 – to enable typewriter keyboard mode, if it is supported by the loaded keyboard layout table.

All those "hot" keys switchings are accompanied with lousy beep sound, and there is no simple way to get rid of it.

Note 1: TSR module of KEYB.COM driver is opened for interaction with programs via INT 2F\AX=AD80h-AD83h (8.03-28, 8.03-30).

5.02-05 KEYRUS.COM – combined keyboard and display driver

KEYRUS.COM driver (written by D.Gurtjak, Donetzk) is a combined keyboard and character generator driver. It is popular among russian users, because it is originally supplied with internal 866 (russian) codepage and with russian keyboard layout. But the original release of KEYRUS.COM also contains supplementary programs, which enable the user to write, to install and to activate any codepage and any keyboard layout. The KEYRUS.COM driver can be downloaded for free from many russian internet sites. The last version 8.0b (1994), packed into keyrus8b.zip archive, is available from author's site http://www.gurtjak.skif.net/pages/programs.htm .

The KEYRUS.COM driver may be loaded from CONFIG.SYS file with INSTALL (4.15) or INSTALLHIGH (4.16) command, or else from command line or from a line of AUTOEXEC.BAT file - directly or with the LH command (3.17), for example

```
LH  C:\DOS\DRV\Keyrus.com
```

When default settings don't suffice, options should be specified after the driver name in the same command line. As options may be numerous, they can be specified inside a separate file; arbitrary name of this file must be preceded by symbol "@" (at):

```
LH  C:\DOS\DRV\Keyrus.com  @Opt_file.ext
```

KEYRUS.COM is not loaded and doesn't affect its loaded TSR part when it is executed in command line to show default options:

```
Keyrus /?
```

or to extract internal fonts and keyboard layout:

```
Keyrus /FILES
```

or to change the default specifications. In the latter case the driver name (KEYRUS.COM) must be followed by a group of options, and the last option in this group must be /SAVE.

KEYRUS.COM driver consists of three TSR modules: keyboard module, character generator module and interface module. Each module accepts its own set of options. If not specified otherwise, acceptance of option "ON" by KEYRUS.COM instead of option "OFF" (and vice versa) everywhere in examples below is always allowed and leads to reverse results.

Options for keyboard module are:

/KEYBOARD=Off – don't load keyboard module, use BIOS' layout.

/BASE_KEYS – enables key reallocation (default is OFF).

/KEYS=filename.ext – take keyboard layout from a file. This file (either 212 or 318 bytes long) may be created by KEYEDIT utility, supplied together with the driver. When KEYRUS.COM is launched with /SAVE option, then specified file is not loaded, but rather becomes KEYRUS's internal default layout.

/BUFFER=ON – expands keyboard's buffer up to 31 characters.

/FAST=ON,10,1 – set keyboard's speed (0 – the fastest, then 1, 2, 4, 8, 10, 13, 16, 20, 31 – the slowest) and delay from 0 (0,25 s) to 3 (1 s).

/RUSALT=ON – enables typing symbols [ ] ; ' , . / in any national keyboard layout while the ALT key is kept pressed.

/BEEP=OFF,rus– no beep at keystroke in RUS (any national) keyboard layout. Instead of RUS the LAT (latin) and ALT (pseudographic) layouts may be specified.

/CLICK=OFF,rus – no click at keystroke in RUS (any national) keyboard layout. Instead of RUS the LAT (latin) and ALT (pseudographic) layouts may be specified.

/LAMP=ON,rus – lit ScrollLock lamp in RUS (any national) keyboard layout. Instead of RUS the LAT (latin) and ALT (pseudographic) layouts may be specified.

/COLOR=0,2 – indicate keyboard layout with border color. Left color code (0 = black) is for RUS (any national) keyboard layout. Right color code (2 = dark green) is for ALT (pseudographic) keyboard layout. Other allowable color codes – in appendix A.10-5.

/ALT=87,4 – set a "hot" key for switching to pseudographic keyboard layout. Shift 4 and scancode 87 mean switching by CTRL-F11 key combination (see notes 3 and 4 below). /ALT=OFF means no access to pseudographic keyboard layout.

/SCAN=54,4 – set a "hot" key for switching to RUS (any national) keyboard layout. Shift 4 and scancode 54 mean switching by CTRL-RightSHIFT key combination (see notes 3 and 4 below).

/LAT=42,4 – set a "hot" key for switching to latin keyboard layout. Shift 4 and scancode 42 mean switching by CTRL-LeftSHIFT key combination (see notes 3 and 4 below). /LAT=OFF means using one "hot" key for toggling to national keyboard layout and back.

/MODESHIFT=87,1 – set a "hot" key for temporary switching keyboard layouts while this "hot" key is kept pressed. Shift 1 and scancode 87 correspond to RightSHIFT-F11 key combination (see notes 3 and 4 below). /MODESHIFT=OFF disables temporary switching.

/CLRSCAN=ON – reset all "hot keys" to original defaults.

Reconfiguration of "hot" keys requires KEYRUS's interface module to be loaded. The hooked "hot" keys must be chosen so as to be not intercepted later by resident shells or by other TSRs.

Character generator's module accepts the following options:

/BLANK=ON,9,ON,ON – blank screen after, for example, 9 minutes idle, the second "ON" means enabling to feel mouse's movement, the rightmost "ON" means to sense screen output.

/SWITCH=22,6 – set a "hot" key to switch character generator to DOS's default codepage 437; scancode 22 corresponds to letter "U", shift 6 means keeping pressed both CTRL and LeftSHIFT keys (see notes 3 and 4 below). Switching of codepages doesn't affect keyboard's layout. /SWITCH=OFF means no access to codepage 437.

/EGA – assume the PC is equipped with EGA-compatible video card. Instead of /EGA you may specify /VGA to assume a VGA-compatible video card. Both /EGA and /VGA options are not saved by /SAVE operation.

/8x8=ON – load 8x8 font for 80x43, 80x50 and similar textual videomodes. This option may be set to OFF and AUTO, the latter means loading on software request.

/8x14=ON – load 8x14 font (used by MS WORD for DOS). This option also may be set to OFF and AUTO, just as the /8x8 option.

/8x16=ON – load 8x16 font for 80x25 main DOS's videomode 03. This option also may be set to OFF and AUTO, just as the /8x8 option.

/FULL – load all 3 sorts of internal fonts.

/ROM – don't load internal fonts, use DOS's default fonts.

/FONT=filename.ext – load an external font given in a separate file; when used together with /SAVE option, the font is not loaded into memory, but becomes accepted as the default internal font of KEYRUS driver.

/DELETEFONT – delete the font, which was installed the last.

/COMPRESS=OFF – don't compress fonts (ON is allowed for textual videomodes only). Choose OFF for graphic videomodes and for "DOS window" of WINDOWS operating system.

/ALL – load all symbols 0 - 255. Implies no font compression (/COMPRESS=OFF)

/128 – load symbols 128 - 255. Implies no font compression (/COMPRESS=OFF)

/RANGE=128-175,224-239 – load symbols within 2 specified ranges, shown as an example.

/RUSSIAN – load the same ranges as in the example above, but with font compression (/COMPRESS=ON). To switch compression OFF the option /COMPRESS=OFF must be specified explicitly.

Interface module of KEYRUS.COM driver accepts the following set of options:

/ANYWAY – allow to load driver's TSR modules into memory once more.

/DELAY_INIT – suspend initialization of driver's TSR modules until the KEYRUS.COM driver is launched from command line.

/INTERFACE=OFF – keep interface module inactive. When it is inactive, programmable reconfiguration is disabled, the KEYRUS.COM driver can't detect presence of its TSR modules in memory and can't unload them from memory.

/RELEASE – unload TSR modules of the KEYRUS.COM driver from memory. This operation needs the interface module to be active (by default /INTERFACE=ON).

Note 1: the KEYRUS.COM driver is not compatible with Microsoft's drivers DISPLAY.SYS (5.02-02) and KEYB.COM (5.02-04), as well as with Microsoft's keyboard layout tables KEYB*.SYS (A.02-2). Those files shouldn't be used, if you intend to load KEYRUS.COM driver.

Note 2: resident interface module of the KEYRUS.COM driver interacts with programs via INT 2F\AX=4352h (8.03-24).

Note 3: the shift here is a sum of digit 1 for the RightSHIFT key, digit 2 for LeftSHIFT key, digit 4 for CTRL key, digit 8 for ALT key, number 16 for ScrollLock switch, number 32 for NumLock switch, number 64 for CapsLock switch and number 128 for Insert switch (see 8.01-85, the byte returned in AL). For example, shift 12 means holding CTRL and ALT keys while pressing the main "hot" key, defined by its scancode.

Note 4: the KEYRUS.COM driver accepts decimal scancodes, so hexadecimal values from the second column in table A.02-1 must be translated into decimal form. Besides this, KEYRUS.COM can't discriminate properly scancodes with and without prefix E0h.

Note 5: inside "DOS window" of Windows-95/98 operating systems the KEYRUS.COM driver operates properly only in textual videomodes, that is when this "DOS window" is extended to whole screen with ALT-ENTER keystroke. The "DOS window" of Windows-2000/XP operating systems can be served by KEYRUS.COM driver too, but in this case it must be loaded either from CONFIG.NT file or from AUTOEXEC.NT file, which both are present in \WINDOWS\SYSTEM32 directory.

5.03 Drivers for "mouse" pointing devices

Functions of mouse pointing device become accessible, when four conditions are met. First, you must have a mouse, which is able to perform the desired function(s) and can be connected to your computer. Second, you must load a driver, supporting execution of the desired function(s) by the mouse with appropriate type of connection. Third, the desired functions must be called by the program, which you want to employ these functions. Fourth, both the driver and the program must be able to operate under the operating system, which is installed in your computer.

As to the mouse pointing devices, the MS-DOS7 operating system seems inconsistent. On one hand, the Windows-95/98 release provides no mouse driver for MS-DOS7. On the other hand, mouse's functions are needed, for example, for editor utility EDIT.COM (6.09), and mouse drivers from previous versions of DOS operate properly under MS-DOS7. All those mouse drivers for DOS interact with application programs in the same way – via interrupt INT 33 (8.03-31 – 8.03-55).

Connection of mouse pointing device to a PC is defined by port address (A.14-1) and by IRQ – interrupt request line number. Mouse drivers, described below in part 5.03, are able to search for mouse pointing devices throughout all relevant ports and IRQ numbers. Though connection specification is not necessary, it nevertheless may be given in order to avoid search and make loading faster.

Since new millennium another breed of "mice" has appeared – those with USB port connection. Such "mice" rely on pointing device BIOS interface (INT15\AX=C2xx) and on driver's interaction with this interface (5.03-3). In obsolete PCs, where BIOS system doesn't respond to INT15\AX=C2xx calls, necessary support may be provided by a choice of appropriate driver for USB controller (5.07-05), and then any mouse driver, described below in part 5.03, will be able to suffice a mouse, connected via USB port.

5.03-01 GMOUSE.COM – mouse driver from KYE Systems Corp.

The GMOUSE.COM driver is supplied in software packets for mouse pointing devices with a well-known trade mark GENIUS, belonging to KYE Systems Corporation. This driver also can be found on compact discs, containing collections of drivers. Version 10.20 of GMOUSE.COM driver (1996) packed into GMOUSE.ZIP archive is available for free downloading from internet site http://www.dosbootsector.narod.ru/drivers.htm .

The GMOUSE.COM driver can be loaded either directly or by LH command (3.17) from command line or from AUTOEXEC.BAT file, or else by INSTALL (4.15) or INSTALLHIGH (4.16) command from CONFIG.SYS file, for example:

```
INSTALLHIGH=C:\DOS\DRV\Gmouse.com
```

where:

C:\DOS\DRV\ – an example of a path to GMOUSE.COM driver.

Normally the GMOUSE.COM driver doesn't needs parameter specifications, but nevertheless it can accept any one of the following mutually excludible parameters:

/1	– mouse connection via serial port COM1
/2	– mouse connection via serial port COM2
/3	– mouse connection via serial port COM3
/4	– mouse connection via serial port COM4
/P	– mouse connection via PS2 Port.
/P2	– 2-button mouse, connected via PS2 port.
/P3	– 3-button mouse, connected via PS2 port.
/U	– unload GMOUSE.COM driver from memory.
/?	– display short help.

The listed parameters are also suitable for newer versions of GMOUSE.COM driver, which have increased file size and several superfluous functions, such as automatic determination of language for messages on basis of the loaded codepage. Contrary to newer versions, version 10.20 of GMOUSE.COM driver can accept some more parameters from a special parameter file GMOUSE.INI, if it is present in the same directory with the driver.

Note 1: if during loading a search for mouse pointing device is stopped by BREAK keystroke, for example, in order to read the displayed messages, then after any

other keystroke the GMOUSE.COM driver is unable to resume the search, and PC gets hanged.

Note 2: when mouse pointing device is connected to port COM3 or COM4, the GMOUSE.COM driver doesn't provide compatibility with Windows OS.

5.03-02 MOUSE.COM – "mouse" driver from Microsoft Corp.

The MOUSE.COM name was common for several mouse drivers, written at different times by different authors. Here the version 8.20 is described of Microsoft's MOUSE.COM driver (file size 37681 bytes, file date 29.06.93). It was supplied within MS-DOS6.22 release, but was found compatible with MS-DOS7 too. Archive MOUSE.ZIP containing MOUSE.COM driver can be downloaded from internet site http://www.computerhope.com/download/hardware.htm#05 .

The MOUSE.COM driver may be installed either from CONFIG.SYS file with INSTALL (4.15) or INSTALLHIGH (4.16) command, or else directly or with LH command (3.17) from AUTOEXEC.BAT file or later from command line:

```
C:\DOS\DRV\Mouse.com /C1 /R1 /S50 /P2 /N50 /Y
```

where:

C:\DOS\DRV\ – an example of a path to MOUSE.COM driver.

/C1 – select COM1 port. Instead of /C1 there may be:
- /C2 – for mouse connected to COM2 port;
- /Z – for mouse connected to PS2 port;
- /I1 – for mouse on LPT1 port;
- /I2 – for mouse on LPT2 port;
- /B – for bus mouse.

/R1 – specifies interrupt rate 30 Hz (the default). Alternatives are:
- /R2 – interrupt rate 50 Hz,
- /R3 – interrupt rate 100 Hz,
- /R4 – interrupt rate 200 Hz.

/S50 – specifies sensitivity; the number after /S is allowed within range 0 - 100, 50 is the default. Instead of /S50 there may be separate /H50 – horizontal and /V50 – vertical sensitivity specifications.

/P2 – slow acceleration profile number. Alternatives are:
- P1 – profile without acceleration
- P3 – moderate acceleration profile;
- P4 – fast acceleration profile.

/N50 – number after /N specifies cursor redraw rate (0 – 255 is allowed).

/Y – use hardware cursor feature.

In most cases all optional specifications are not needed, since the default values are sufficient, and the driver is able to search for mouse pointing devices throughout all relevant ports.

In order to disable mouse you may unload this driver by launching it with single OFF parameter:

```
C:\DOS\DRV\Mouse.com OFF
```

5.03-03 CTMOUSE.EXE – freeware "mouse" driver

The CTMOUSE.EXE driver is developed since 2002 as a part of FreeDOS project, but it is made compatible with MS-DOS7. Version 2.1 of CTMOUSE.EXE (dated 2007), packed in archive CTMOUS21.ZIP, may be downloaded from internet site http://www.ibiblio.org/pub/micro/pc-stuff/freedos/files/dos/mouse/ .Another address, where recent versions of this driver can be found, is http://cutemouse.sourceforge.net/ .

Functionally CTMOUSE.EXE is similar to other mouse drivers for DOS, but it occupies less memory and supports pointing devices equipped with a wheel. One more important feature of this driver is its interaction with pointing device BIOS interface (INT15\AX=C2xx). Therefore it provides better chances to suffice USB mice in modern PCs. CTMOUSE.EXE driver has an elaborated set of defaults, which make command line parameters in most cases unnecessary. It may be loaded either from command line or from AUTOEXEC.BAT file – directly or by LH command (3.17), or else from CONFIG.SYS file by INSTALL (4.15) or INSTALLHIGH (4.16) command, for example:

```
INSTALL=\DOS\DRV\Ctmouse.exe /S14 /3 /R33 /N
```

where:

\DOS\DRV\ – path example to Ctmouse.exe driver.

/S14 – search for mouse pointing device through serial ports only. The first digit denotes serial port number (=COM1), the second – interrupt request line number (=IRQ 4). If the /S parameter is not specified, driver will examine PS2 port first, and then – all serial ports.

/3 – activate mouse's third button. This function can't be applied to mice of Mouse System type and to mice with a wheel instead of middle button.

/R33 – sensitivity settings: first digit is a grade of horizontal sensitivity, second digit – of vertical sensitivity. If only one digit is specified, it is applied to both coordinates. Default /R parameter value is /R33.

/N – load Ctmouse.exe driver even if some mouse driver has been loaded yet. This parameter enables to compose batch files so that after unloading of Ctmouse.exe driver the configuration returns back to its original state, whichever this state was.

Besides the shown parameters, Ctmouse.exe driver can accept the following options:

/? – display a short help;
/P – examine the PS2 port only;
/Y – don't search for "mouse" of Mouse System type;
/V – reverse search order: examine serial ports first;
/O – disable mouse wheel detection;
/L – adapt buttons order for left-handed people;
/B – don't load Ctmouse.exe, if some mouse driver is loaded yet;
/W – load Ctmouse.exe into conventional memory;
/U – unload CTMOUSE.EXE driver from memory (this can't be done if functions of the driver are intercepted).

5.04 Memory managers

In AT-compatible computers the 640 – 1024 kb memory region is hardware consigned for special usage as BIOS code "shadow", as a "window" into video memory, as a page frame "window", etc. (see notes 2 and 3 to A.12-1 for more details). Because of 16-bit addressing and because of special status of 640 – 1024 kb memory region the accessible memory space for DOS programs is considerably limited. Since early 1980-ties researchers in all leading software companies attempted to overcome memory space restrictions. Nowadays these attempts are continued by many independent software vendors. The best solutions of this problem are presented here in the following articles.

5.04-01 HIMEM.SYS – extended memory driver

HIMEM.SYS is Microsoft's memory driver for PCs with CPU 80386 or higher, providing machine-dependent access to memory beyond the 1024 kb boundary according to eXtended Memory Specification (XMS). Therefore the memory, opened by HIMEM.SYS, is often called XMS memory. Since version 3.10 of HIMEM.SYS driver the upper limit of XMS memory is 65535 kb.

Besides access to XMS memory, the HIMEM.SYS driver performs two important functions: control over switching of A20 line gate and allocation of memory space, requested by different programs. Both these functions are necessary in order to prevent conflicts between programs, addressing memory beyond conventional memory limit (640 kb). Due to HIMEM.SYS driver each program, requesting some space in UMB region or in XMS memory, is given exclusive access rights to the allocated part of memory space.

Official explanations of HIMEM's operation principle are not known, but analysis of its code gives some grounds to suppose, that it uses 32-bit linear addressing in real mode (more about that – in 9.10). In any case, HIMEM's operation principle gives no chance to execute programs in XMS memory, it gives only an opportunity to copy executable code together with data from conventional memory into XMS memory and back.

Version 3.95 of HIMEM.SYS driver (1995) is supplied in WINDOWS-95/98 release and normally may be found in \WINDOWS directory. The HIMEM.SYS driver should be loaded with DEVICE command (4.06) preferably in the first executable line of CONFIG.SYS file, for example:

```
DEVICE=C:\DOS\DRV\Himem.sys /V
```

where:

C:\DOS\DRV\ – path example to HIMEM.SYS driver;

/V – display loading information. The same may be achieved if the ALT key is kept pressed while HIMEM.SYS is being loaded.

HIMEM.SYS has internal default settings, which are suitable for a wide variety of PC's hardware. In rare cases you may have to specify other settings by optional parameters. Allowable optional parameters are:

/a20control:OFF – allows HIMEM.SYS to take control over A20 bus if it is not active only, thus preventing A20 bus interruptions (default is to take control always).

/cpuclock:ON – activate CPU clock frequency control, when this frequency becomes affected by upper memory access. This parameter makes HIMEM.SYS work slower.

/eisa – provides access beyond 16 Mb on PCs with EISA bus.

/hmamin=40 – prevents reserving of HMA area (1024 – 1088 kb) to any program demanding less than 40 kb (range is 0 – 63 kb). By default HMA area is allocated to the first program that requests it, regardless of how much of HMA space the program intends to use.

/int15=128 – reserve 128 kb (range 64 – 65535, default is 0) for programs, accessing memory via interrupt INT 15\AH=87h (8.01-76).

/machine:AT – specification of PC types, which HIMEM.SYS can't identify properly, by number (from 1 to 17) or by literal identifier (see table A.11-2). Default is number 1 or identifier AT (both correspond to IBM's PC/AT).

/noabove16 – don't use INT 15\AX=E801h (Compaq Bigmem support) to scan for memory beyond 16 Mb. In this case HIMEM.SYS can't provide XMS access beyond 16 Mb.

/noeisa – prohibit search for memory extension cards on EISA bus.

/numhandles=32 – maximum number of active references (handles) to XMS memory blocks, which can be kept open simultaneously; allowed range is from 1 to 128, 32 is the default.

/shadowRAM:ON – don't prevent BIOS to RAM transfer into segment address space F000 - FFFFh. When HIMEM.SYS detects RAM size 2 Mb or less, it tries to prevent BIOS to RAM transfer in order to get more memory free.

/testmem:OFF – bypass XMS memory test.

/X – don't explore memory with INT 15\AX=E820h (8.01-80).

Note 1: HIMEM.SYS driver gives no access to memory beyond 16 Mb, if CMOS BIOS parameter "Memory hole 15 – 16 Mb" is enabled.

Note 2: HIMEM.SYS driver accepts program's requests via CALL FAR commands (7.03-08). Address for CALL FAR commands can be obtained with INT 2F\AX=4310h function (8.03-23). A list of operations, which can be requested by programs, is shown in table A.12-3.

Note 3: several similar XMS memory drivers have been developed by different authors. J.R.Ellis wrote QHIMEM2.SYS driver, which extends XMS memory limit to 4 Gb. Version 3.1 of this driver (dated 2005) is available inside SFX diskette image USB18.EXE from site http://johnson.tmfc.net/dos/usbdrv.html . The most recent 4 Gb XMS memory driver XMGR.SYS can be downloaded from page http://johnson.tmfc.net/dos/driver.html . One more similar driver – Himem.exe – is developed as a part of FreeDOS project and is supplied together with Emm386.exe (see note 4 to 5.04-02). Alternative XMS memory drivers accept different sets of options, described in attached files.

Note 4: in MS-DOS8 the XMS access code is integrated into the core, hence there is no need to load Himem.sys.

5.04-02 EMM386.EXE – memory mapping driver and VCPI server.

EMM drivers (EMM = Expanded Memory Manager) originally were developed in 1983 – 1984 in order to provide access to memory banks on expansion cards for IBM PC, so that any memory bank can be addressed through the same "window" in address space (usually in segment addresses E000 – EFFFh). This method of addressing memory banks was institutionalized by LIM EMS specification. Among other details, LIM EMS specification stipulates division of the address space "window" into several pages of 16 kb each, which independently direct addressing to different memory banks.

Later the main computer's memory has grown far above 1 Mb, and expansion cards with memory banks have come out of use. But some popular programs still relied on old-fashioned LIM EMS access, and compatibility with these programs had to be preserved. Therefore the fundamental principle of EMM driver's operation has been changed: control over expansion cards has been replaced by control over address translation mechanism, which is embedded in all CPU's since 80386. This is why such memory managers are named EMM386. They enable to address the main computer's memory above 1 Mb through the same LIM EMS "window" in address space.

Here is just a place to remind, that main computer's memory above 1 Mb is controlled by XMS driver HIMEM.SYS (5.04-01), and memory space allocation according to program's requests is just its prerogative. Hence the EMM386 driver plays a role of an intermediary transactor: having got a program's request for LIM EMS access to a memory

space, it asks the HIMEM.SYS driver and gets a suitable piece of memory beyond 1 Mb. Then EMM386 rewrites data in CPU's TLB table so that the caller program becomes able to access the allocated piece of memory via the LIM EMS "window" in address space. When there is no a single piece of requested size, the EMM386 driver combines several pieces of smaller size. Due to address translation in CPU, the LIM EMS "window" in address space enables not only access, but execution of programs too.

Mission of EMM386 driver is complicated by the fact that address translation mechanism is activated only when CPU works in protected mode. Therefore the EMM386 driver switches CPU into virtual 8086 (V86) mode, which is a variant of protected mode, and then all DOS programs are executed in this mode at the third (the lowest) privilege level. Of course, PC must have a CPU of 80386 type or newer, which is able to implement V86 mode. EMM386 driver gives to DOS programs a permission for I/O operations at the lowest privilege level, so that conditions of access to disks and to ports are the same as in real mode. Additional features, inherent to the V86 mode, are used by EMM386 driver for loading other drivers "through" free areas of UMB address space (640 – 1024 kb) and for control over attempts of other programs to affect implementation of protected mode.

At least three types of programs pretend to control protected mode implementation in DOS. These are extender programs (for example, DOS4GW.EXE), DPMI servers (for example, CSWDPMI.EXE), and loaders of operating systems (for example, the Windows' GUI loader WIN.COM). Such contenders can't perform their mission in V86 mode at the lowest privilege level. In order to enable smooth control transfer to such programs the VCPI protocol (VCPI = Virtual Control Program Interface) has been adopted in 1989. It was developed jointly by Phar Lap Software and Quarterdeck Office Systems. According to VCPI protocol, the program, which switches CPU into V86 mode, must perform several extra functions on request, including transition to the highest privilege level. Since EMM386 implements these extra functions, it is also called VCPI server. Some of VCPI server's functions are described in articles 8.03-71 - 8.03-73.

Direct execution of programs beyond 1088 kb boundary is necessary for loading drivers there, but that's not enough. Ordinary driver's code is not adapted to page-by-page addressing. Therefore for loading drivers the EMM386 manager arranges static mapping of memory beyond 1088 kb onto UMB blocks, which can be squeezed in all free stretches of address space between 640 and 1024 kb. If EMM386 memory manager is loaded yet, and then is launched from command line once more (without parameters), it will show on the screen the exact allocation of entrusted address space.

Windows-95/98 release includes version 4.95 of EMM386.EXE driver (date 6/12/1996, size 125495 bytes). This version is not intended, though, for service to a separate operating system MS-DOS7. Version 4.95 transfers exception calls to protected mode handlers, installed by Windows OS. But when MS-DOS7 runs alone, exception calls don't find their handlers, and PC gets hanged. This is why earlier versions 4.49 or 4.50 of EMM386.EXE driver are more suitable for MS-DOS7. These versions are loaded in the

same way and accept the same set of options. Version 4.49 (date 31/05/1993, size 120926 bytes) can be downloaded from server ftp://ftp.vgt.ru/dos/ inside diskette image Dos622_1.img, which can be written to diskette by IMG.EXE (6.06). Then file EMM386.EX_ should be unpacked by EXPAND.EXE (6.10). Version 4.50 (date 30/04/1998, size 119390 bytes) from IBM's PC DOS 2000 release can be downloaded inside SFX archive Pcdos2k.exe from server ftp://ftp.eesnet.ru/dos/ .

Loading of DOS structures and of other drivers beyond conventional memory by LH (3.17), DEVICEHIGH (4.07) and other commands, ending with ...HIGH, requires memory manager to be loaded yet. But access beyond conventional memory must be provided in advance, before the EMM386.EXE driver is loaded. Therefore EMM386.EXE driver should be loaded by DEVICE command (4.06) from that line of CONFIG.SYS file, which follows HIMEM.SYS (5.04-01) loading line, but precedes to all lines with commands, having the ...HIGH ending. A line loading EMM386.EXE driver may look, for example, as

```
DEVICE=C:\DOS\DRV\Emm386.exe RAM V
```

where:

C:\DOS\DRV\ – path example to EMM386.EXE driver;

RAM – allow arrangement of EMS pages and UMB blocks (another form of RAM parameter usage is shown further);

V – display allocation of EMS pages and UMB blocks.

Besides the shown options, EMM386.EXE driver can accept optional parameters from the following list:

OFF – suspend activation of EMM386.EXE until command "EMM386.EXE ON" is launched from command line. Parameter AUTO (instead of OFF) also allows activation on software request. While EMM386.EXE is not active, it doesn't support loading beyond conventional memory by LH command and by all other commands ending with "…HIGH".

8196 – an example of requested EMS-memory size in kb, from 16 kb up to actual size of free XMS-memory (latter is the default), but not more than 32768 kb. If NOEMS switch is specified (see further), the default value is set to zero. EMM386 rounds any requested value down to the nearest multiple of 16 kb.

min=256 – don't establish EMS-memory, if its available amount doesn't exceed this minimum value, ranging from 0 up to requested memory size, 256 kb is the default (unless the NOEMS switch is specified).

W=ON – enable Weitek coprocessor support (default is OFF). When EMM386 is active, it accepts from command line commands

```
EMM386 W=OFF
EMM386 W=ON
```

M4 – an example of start segment address specification for page frame placement. Page frame is a region for successive continuous placement of EMS pages 0 – 3 (16 kb each). The number after "M" is a code (from 1 to 14), which denotes segment addresses
1 = C000h, 2 = C400h, 3 = C800h, 4 = CC00h,
5 = D000h, 6 = D400h, 7 = D800h, 8 = DC00h,
9 = E000h (the default setting), 10 = 8000h,
11 = 8400h, 12 = 8800h, 13 = 8C00h, 14 = 9000h

FRAME=CC00 – an example of direct specification for the same page frame start address. Other start addresses, except those shown above, are not allowed. In order to prohibit frame page you may specify FRAME=NONE, and this will be equivalent to the NOEMS parameter (see further).

/PCC00 – one more example of another form for the same page frame start address specification; the same 14 values can be specified after the /P parameter.

P4=DC00– an example of specification for adding one more EMS page P4 (the fifth) to the page frame CCOOh – DBFFh. There may be several "P" parameters for EMS pages denoted by numbers from 4 to 14. When page frame start address is not specified otherwise, numbers 0 – 3 are allowed too, but sequential continuous allocation of EMS pages 0 – 3 must be preserved.

X=F000-FFFF – a specification example for a prohibited region of segment addresses (within A000h – FFFFh range), which must be left intact. Specified segment addresses will be rounded down to the nearest 4-kilobyte multiple.

I=BC00-BFFF – a specification example for a region of segment addresses to be allocated for UMB blocks. Specified segment addresses will be rounded down to the nearest 4-kilobyte multiple. When "I" and "X" specifications overlap, the "X" option takes preference.

B=4000 – lowest boundary segment address (within range 1000h – 4000h) of allowed EMS pages placement region. Default lowest boundary is 4000h.

L=256 – size of memory space (in kb) reserved for XMS-type access (default is 0 kb).

A=7 – number of register banks to provide support for multitasking (0 – 254 allowed, default is 7). Each register bank takes about 200 bytes.

h=64 – number of references (handles) to EMS memory areas, which can be kept open for access simultaneously (2 – 255 allowed, default is 64).

d=32 – size of reserved Direct Memory Access (DMA) buffer (16 – 256 kb allowed, default is 32 kb).

RAM=C000-EFFF – a specification example of region boundaries for UMBs and EMS pages placement. When "RAM" parameter is not followed by boundary segment addresses, all available memory space is regarded as allowable.

WIN=E000-EFFF – an example of segment addresses, specifying boundaries of address space region, reserved for being used by Windows OS.

ROM=F000-FFFF– load BIOS shadow into a part of address space, specified by segment addresses of its boundaries, if this is not done by PC's BIOS itself. Copying of BIOS code can make your PC more fast.

NOEMS – don't create EMS page frame, don't provide LIM EMS memory access, but use all available address space in 640 – 1024 kb region for arranging UMBs and for loading drivers.

NOHI – load the whole resident module of EMM386.EXE driver into conventional memory below 640 kb.

NOMOVEXBDA – cancel default loading of BIOS shadow.

NOTR – exclude network card search (for EMM386.EXE versions 4.45 – 4.95)

NOVCPI – disable support for VCPI functions (see note 2 below), requested by other programs. This switch should be used together with the NOEMS switch.

HIGHSCAN – launch a thorough search for free stretches of address space, thus making its usage more efficient. But there is a chance to consider engaged space as free, and then your PC may get hanged.

ALTBOOT – replace hot reboot handler (only if CTRL-ALT-DEL "hot key" combination stops responding or goes wrong after loading of EMM386.EXE driver).

NOBACKFILL – prevent conventional memory from being backfilled, when EMM386 attempts to create UMBs in a PC having total memory 640 kb or less.

When the EMM386.EXE driver is loaded yet and active, it can accept commands "EMM386 OFF" and "EMM386 AUTO" from command line. But these commands will not be executed, if at that moment UMB address space has been used already for loading other drivers, which must remain available for DOS.

Note 1: the EMM386.EXE driver accepts requests from other programs via INT 67 (8.03-57 – 8.03-74).

Note 2: VCPI services, provided via INT 67\AX=DE00h-DE0Ch, give a chance to perform their mission for programs, which have to affect V86 mode of CPU operation, set by EMM386.EXE. Due to VCPI services, in particular, the Windows OS can be loaded even if the V86 mode is set yet. But the Windows OS itself inside its "DOS window" disables VCPI services, thus excluding risk to lose control over the PC.

Note 3: a version of EMM386.EXE from MS-DOS8 is not compatible with some model types of obsolete 486 processor and may cause hanging.

Note 4: several similar EMM386 managers are known to be developed. The most advanced is JEMM386.EXE driver, written by a group of authors under auspices of Tom Ehlert. Now, in december 2007, archive JEMM568.ZIP with version 5.68 of this driver can be downloaded from internet site http://japheth.de/ . One more version of EMM386.EXE driver is developed as a part of FreeDOS project; it can be downloaded from server ftp://ftp.devoresoftware.com/downloads/ . Both mentioned drivers are more compact, than original EMM386.EXE, and accept different sets of options, described in attached files.

5.04-03 Memory allocation optimization: CHKSTATE.SYS

CHKSTATE.SYS is a special service utility for memory optimization procedure, developed for MS-DOS6, but proved to be suitable in MS-DOS7. A line loading CHKSTATE.SYS is inserted into CONFIG.SYS file automatically by memory optimization manager MEMMAKER.EXE, and is deleted in the same way when memory optimization procedure terminates. Being executed, CHKSTATE.SYS utility forms a textual report file, containing exact data about memory areas, allocated for each loaded driver.

Note 1: for performing memory optimization procedure the main optimization manager MEMMAKER.EXE must be in one directory with auxiliary files CHKSTATE.SYS, EMM386.EXE, HIMEM.SYS, MEMMAKER.HLP, MEMMAKER.INF and SIZER.EXE. All these files are present in MS-DOS6.22 release, and also are contained in SFX archive OLDDOS.EXE, freely available from server ftp://ftp.microsoft.com/softlib/mslfiles/ .

Note 2: the MEMMAKER.EXE optimizer is unable to make proper corrections in AUTOEXEC.BAT file and in CONFIG.SYS file, if the latter contains a menu with several loading alternatives. Each alternative variant should be optimized separately. Nevertheless the data, obtained for each loading alternative separately, enable to compose optimized configuration files with several loading alternatives.

5.04-04 UMBPCI.SYS – freeware UMB region driver

In order to open UMB address space (C000h – EFFFh) for loading drivers, the EMM386.EXE memory manager (5.04-02) arranges address translation via CPU's TLB tables and forces to pay for that with switching CPU into V86 mode. But when it is necessary to stay in real mode, access to UMB address space may be opened by reprogramming memory controller, which is a part of chipset on PC's motherboard. This alternative way of access to UMB address space is implemented by UMBPCI.SYS driver.

The UMBPCI.SYS driver has a long development history with a lot of participants. Now it is continued by Uwe Sieber. The most recent version of UMBPCI.SYS can be found in internet site http://www.uwe-sieber.de/ , packed in archive UMBPCI.ZIP with attached texts in german, or else in archive UMBPCI_E.ZIP with attached texts in english. The description below is based on version 3.66 of UMBPCI.SYS, dated march 2006.

By no means, UMBPCI.SYS is not a replacement of EMM386.EXE, it does not implement LIM EMS specification. UMBPCI.SYS performs only functions, stipulated by XMS specification, but just those, which are incumbent upon EMM386.EXE driver (see note 6 to A.12-3).

In order to affect memory controller's settings, the UMBPCI.SYS driver calls BIOS's interrupt handlers INT 1A\AH=B1h, related to PCI bus services. This is why the UMBPCI.SYS driver can perform its mission only in computers, which are equipped with PCI bus, controlled by PC's BIOS. These conditions are met by almost all AT-compatible computers, produced after 1996. One exception is represented by computers with AMD-K7 processor: in such computers the UMB blocks, created by UMBPCI.SYS driver, can't be used for loading those drivers, which control expansion cards on PCI bus.

The UMBPCI driver provides access to free parts of "shadow" memory area, intended for copying executable codes from slow fixed storage BIOS chips. Not every chipset is able to provide direct memory access (DMA) into this "shadow" memory region. The DMA access is necessary for floppy disks controller, for SMARTDRV.EXE driver (5.06-01), and even for the WINDOWS OS, if it would be loaded later. The archives with UMBPCI.SYS driver contain some auxiliary utilities, enabling to avoid undesirable consequences of restrictions on DMA access to "shadow" memory areas. In the same archives there are files with advices, helping to overcome problems, specific for some particular chipsets. Of course, relevant peculiarities must be taken into account.

The UMBPCI.SYS driver should be loaded by DEVICE command (4.06) from that line of CONFIG.SYS file, which follows loading of HIMEM.SYS driver (5.04-01) and precedes all lines with commands LH and those having the ...HIGH ending. The line loading UMBPCI.SYS driver may look, for example, as

```
DEVICE=\DOS\DRV\Umbpci.sys  /I=D000-DFFF
```

where

\DOS\DRV\ – path example to UMBPCI.SYS driver.

/I=D000–DFFF – an optional specification of segment address space, which should be allocated for UMB blocks. Boundaries of the allocated space must be multiples of 16 kb (that is C800, CC00, D000, D400, D800 and so on).

If the /I= parameter is specified, then UMBPCI.SYS will not examine address space areas on whether they are free or not. Presence of several /I= parameters in one command line is allowed, and then each UMB area will be given a serial number. The

DEVICEHIGH (4.07) and LH (3.17) commands accept serial number of UMB area after their /L: parameter, and due to that there is an opportunity to load any particular driver through the prescribed part of UMB address space. The latter is important for those drivers, which use direct memory access (DMA), because some chipsets – for example, i430TX – provide DMA access only inside the E000 – EFFFh part of UMB region. Most often the /I= parameter is not needed, since popular modern chipsets (i815, i820, i845, i850, i855 and many others) apply no restrictions on DMA access to UMB region.

Note 1: in obsolete computers, having no PCI bus, access to UMB address region may be provided by HIRAM.EXE driver (written in 1993) and its auxiliary utilities. This set of software can be downloaded as one archive from internet site http://www.uwe-sieber.de/files/hiram.zip .

5.05 RAM-disk drivers

RAM-disk drivers occupy a part of computer's random access memory (RAM) in order to simulate a virtual disk drive. This provides access to writable disk space during exploratory and reparatory works, when physical disk space may not exist or must be preserved free from traces of access. Virtual RAM-disks are much faster, than any physical disk. Since contents of RAM-disk is lost each time the PC is switched off, RAM-disks sometimes are used as a place for temporary files in order to avoid pollution of physical disk space.

A common drawback of many RAM-disk drivers is that they are unable to assign a certain letter-name for the created virtual disk. DOS always assigns to virtual disk the first free letter-name, which can't be known beforehand. One solution, presented in article 9.04-02, is a search for a particular letter-name, assigned to the arranged virtual disk. Another solution is to write a special driver, which will deceive DOS with a false number of registered disks, so that the next disk could be given an arbitrary prescribed letter-name. An example of the latter solution is presented in article 9.08.

5.05-01 RAM-disk driver RAMDRIVE.SYS

RAMDRIVE.SYS is Microsoft's RAM-disk driver for DOS, supplied within WINDOWS-95/98/ME releases. Normally it can be found in \WINDOWS directory.

RAMDRIVE.SYS should be loaded by a DEVICE (4.06) or DEVICEHIGH (4.07) command from a line of CONFIG.SYS file, for example:

```
DEVICE=C:\DOS\DRV\Ramdrive.sys 16000 512 256 /E
```

where:

C:\DOS\DRV\ – a path example to RAMDRIVE.SYS driver.

16000 – an example of requested RAM-disk size in kilobytes, 4 – 32767 kb range is allowed, 64 kb is the default value.

512 – an example of sector size in bytes; 128, 256 and 512 bytes are allowed, 512 is the default value. If sector size is specified, RAM-disk size must be specified too.

256 – an example specification of entries number in the root directory of RAM-disk, 2 – 1024 are allowed, 64 is the default value. If number of entries is specified, sector size must be specified too.

/E – optional parameter, forcing to arrange RAM-disk in XMS memory, provided by HIMEM.SYS driver. Instead of /E you may specify /A parameter, forcing to use EMS memory access, provided by EMM386.EXE driver.

You may set up as many RAM-disks as free memory space allows: just add one such line to your CONFIG.SYS file per each extra RAM-disk.

Note 1: RAM-disk driver RDISK.COM with 2 Gb size limit may be downloaded from http://johnson.tmfc.net/dos/driver.html . This driver should be used together with XMS memory managers providing 4Gb XMS memory limit (note 3 to 5.04-01).

5.05-02 Reconfigurable drivers TDSK.EXE and BITDISK.EXE

Freeware drivers TDSK.EXE and BITDISK.EXE enable to define RAM-disk size not only at the moment of loading, as RAMDRIVE.SYS driver does, but also enable to postpone RAM-disk size definition, and even enable to do it repeatedly. This feature is very important for installation of MS-DOS7 on an unknown computer, because you need to determine available amount of memory before a decision can be made about whether to arrange a RAM-disk and of what size.

Both TDSK.EXE and BITDISK.EXE drivers were written by Garcia de Celis, Valladolid, Spain, in 1992 – 1995, and then the last was version 2.3. Later these drivers were modernized and became a part of FreeDOS project. Now both these drivers can be downloaded from http://www.ibiblio.org/pub/micro/pc-stuff/freedos/files/dos/ramdisk/ . Original drivers of Garcia de Celis are there in archive TDSK23.ZIP, and modernized version 2.42 of the TDSK.EXE driver – in archive TDSK242B.ZIP.

BITDISK.EXE is a simplified version of TDSK.EXE. While the latter is able to employ EMS, XMS or conventional memory, BITDISK.EXE can cope with XMS memory only. Therefore it always needs the HIMEM.SYS driver to be loaded in advance. TDSK.EXE can accept all BITDISK's options and, besides that, some options of its own.

Original drivers of Garcia de Celis should be initialized by DEVICE (4.06) or DEVICEHIGH (4.07) command in a line of CONFIG.SYS file. Modernized version 2.42 of the TDSK.EXE driver must be loaded in conventional memory only by DEVICE

command (4.06), provided that CONFIG.SYS file also contains a line with DOS=NOAUTO command (4.08). RAM-disk initialization line may include all required parameters (see further), and then RAM-disk will be created at once. But when creation of RAM-disk should be retarded, command line in CONFIG.SYS file must not include particular parameter's values; it may look, for example, as

```
DEVICEHIGH=C:\DOS\DRV\Bitdisk.exe 0
```

or else as:

```
DEVICE=C:\DOS\DRV\Tdsk.exe 0
```

where:

C:\DOS\DRV\ – path example to the driver;
0 – zero size denotes that RAM-disk shouldn't be arranged at once.

The shown command doesn't cause memory allocation for the RAM-disk, but induces loading and initialization of driver's TSR module (about 600 bytes). At this stage DOS registers new RAM-disk and assigns its letter-name. If you repeat this line in CONFIG.SYS file more than once, you can initialize several RAM-disks.

In order to make RAM-disk accessible, the same driver must be invoked later by a command, written into a line of AUTOEXEC.BAT file or directly into DOS' command line, for example:

```
C:\DOS\DRV\Bitdisk.exe R: 5600 512 384 4 /F:2
```

or else

```
C:\DOS\DRV\Tdsk.exe R: 5600 512 384 4 /F:2 /E /M /I=1
```

where:

R: – letter-name of the addressed RAM-disk, if there are several RAM-disks. Letter-name of a single RAM-disk may be omitted.
5600 – an example of required disk's size specification (in kb). Minimum size is 4 kb, maximum – 32766 kb for BITDISK.EXE and 65534 kb for TDSK.EXE. Command with size 0 releases all occupied memory, except that for driver's TSR module: it remains loaded and ready to rearrange RAM-disk on request.
512 – optional specification example for sector size (in bytes); sector sizes 64, 128, 256 and 512 bytes are allowed, the latter is the default.
384 – optional number of entries in the root directory. Allowed values are from 1 to disk size limit, 384 is the default. When number of entries isn't omitted, sector size must be specified too.
4 – optional number of sectors per cluster, in MS-DOS7 must be a power of 2. Default is the minimum number corresponding to RAM-disk's size. When number of sectors per cluster isn't omitted, then number of entries in the root directory must be specified too.

/F:2 – specifies creation of TWO copies of FAT, as in real disks (default is creation of a single FAT table). In rare cases some disk addressing utilities can't work properly with virtual disks, having only one FAT.

Command line, addressed to TDSK.EXE, contains some additional options, accepted by TDSK.EXE only. Allowable additional options for version 2.42 of TDSK.EXE driver are:

/E – arrange RAM-disk in XMS memory (the default), provided by HIMEM.SYS driver, which has to be loaded yet.

/C – arrange RAM-disk in conventional memory.

/X – (or equivalently /A) – arrange RAM-disk in EMS memory, provided by EMM386.EXE driver. In this case both EMM386.EXE and HIMEM.SYS memory drivers must be loaded in advance.

/B – don't load TDSK.EXE driver, if at least one real disk drive is found (this option isn't accepted by original version 2.3 of TDSK.EXE driver).

/M – display screen messages in monochrome (default is color).

/I=1 – display messages in english (the default); besides this, spanish (/I=34) and german (/I=49) languages may be chosen.

Auxiliary options /E, /C, /X, /A are mutually exclusive and usually are omitted, because TDSK.EXE driver itself is able to choose optimum place to arrange RAM-disk.

The shown form of command can be repeatedly used to call either TDSK.EXE or BITDISK.EXE in order to change RAM-disk's size. Each time after this command RAM-disk is rearranged anew, and all its contents becomes lost. RAM-disk data are not affected by either driver in two cases:

– if it is called without parameters to show current status;

– if it is called with a single /? parameter to show help.

In order to solve the problem of RAM-disk letter-name determination, version 2.42 of TDSK.EXE driver is able to assign this letter-name as a value to environmental variable named TURBODSK or RAMDRIVE. An environmental variable with any of the mentioned names should be prepared in advance with SET command (3.26), and it should be given a question mark and a colon as its preliminary value:

```
SET RAMDRIVE=?:
```

If a variable with the same name has another value, it will be preserved intact. But if this variable has just the shown value, and if at that moment RAM-disk is arranged already, then after execution of TDSK.EXE driver the question mark in variable's value will be replaced by letter-name of the RAM-disk. When RAM-disk’s letter-name is known yet, then it's easy to compose AUTOEXEC.BAT file so that this letter-name is automatically written into all relevant paths (an example – in article 9.09-02).

After execution the TDSK.EXE driver returns errorlevel code (3.15-03, 9.07-03). Errorlevel values from 1 to 128 denote a reference number (handle) for XMS or EMS access to allocated memory area; other errorlevel values mean the following:

0 – RAM-disk is not arranged since it isn't defined;
252 – syntax error;
253 – attempt to define a RAM-disk under Windows OS;
254 – incorrect specification of disk's letter-name;
255 – TSR module of TDSK.EXE driver is not loaded.

5.06 HDD services

5.06-01 SMARTDRV.EXE – cache buffer driver

The SMARTDRV.EXE driver arranges and maintains disk's cache buffer. Data transfer operations in this buffer are performed via DMA controller. DMA access gives some relief to CPU and makes transfer operations faster for both HDDs and CD drives. Time saving effect becomes essential in case of large data amounts transfer, for example, during installation of operating systems Windows ME/2000/XP under DOS.

The SMARTDRV.EXE driver is a part of Windows-95/98 release and must be present in \WINDOWS directory. But WINDOWS OS itself performs buffering otherwise and doesn't need SMARTDRV.EXE. It is needed only in DOS, and in MS-DOS7 as well.

Since the cache buffer is expedient beyond conventional memory, access above 640 kb must be opened in advance. Therefore appropriate memory manager(s) (5.04-01, 5.04-02, 5.04-04) must be loaded before SMARTDRV.EXE. If you intend to access CD-ROMs with MSCDEX.EXE (5.08-03), the latter also should be loaded in advance. Most often SMARTDRV.EXE is loaded either by INSTALL command (4.15) from CONFIG.SYS file or from a line in AUTOEXEC.BAT file, for example:

```
C:\DOS\DRV\Smartdrv.exe /X A- B- C+ /U /N /L /V 128 /E:2048 /B:4096
```

where:

C:\DOS\DRV\ – path example to SMARTDRV.EXE driver.
/X – disable write-behind caching for disks, except those enlisted after the /X parameter with plus sign (as C+). On disks followed by minus sign (as A–, B–) read-ahead buffering is disabled too. If the /X parameter is omitted, then write-behind caching is enabled only for HDDs.
/U – don't load CD-ROM caching module.
/N – allow to accept the next command when write cache is not written yet onto disk. This parameter affects those drives only, where write-behind caching is enabled. If the /N parameter is omitted, command prompt does not appear until cache writing to disk operation terminates.

/L – forces to arrange cache buffer in conventional memory. This is needed, when DMA controller has no access to UMB address space.

/V – display status message (by default nothing is displayed, when SMARTDRV.EXE is launched for the first time). Instead of /V there may be /S – display status message and cache statistics.

128 – cache size in kilobytes. It must be specified, if a large part of XMS-memory is to be used for other purposes, for example, as a RAM-disk.

/E:2048 – size in bytes of transferred data blocks; 1024, 2048, 4096 and 8192 are allowed, 8192 is the default.

/B:4096 – size in bytes of read-ahead buffer, it must be a multiple of transferred data block, 16384 is the default.

When SMARTDRV.EXE is running yet, it may be called from command line with other set(s) of options for execution of a command or for reconfiguration, for example:

```
C:\DOS\DRV\Smartdrv.exe /X C- D+ /R /F /Q
```

where:

/X – disable write-behind caching for all disks (if not followed by particular disk's letter-names). This parameter may be applied, if write-behind caching was not disabled in current state of SMARTDRV.EXE. Here C– and D+ letter-names mean disable write-behind caching for disk C: and enable it for disk D:.

/R – clear the cache and restart SMARTDRV.EXE. One more operation to be performed immediately (instead of /R) is /C – write information currently present in write-behind cache to disk.

/F – discard the /N parameter, if it is active in current state, and return to default state, when command prompt doesn't appear until cache writing operation terminates. If the state should be reversed in opposite direction, then /N parameter should be specified instead of /F.

/Q – cancel default status message display. Instead of /Q there may be /S – display status message plus cache statistics.

Note 1: /C operation (writing cache) ignores /V and /S parameters, nothing is displayed.

Note 2: DMA data transfer, arranged by SMARTDRV.EXE driver, doesn't involve operations, performed by SHSUCDX.COM (5.08-04).

Note 3: in order to cope with SATA drives and with UltraDMA data transfer the UIDE.SYS cache driver should be preferred. This driver can be downloaded from site http://johnson.tmfc.net/dos/driver.html .

5.06-02 Double buffering driver DBLBUFF.SYS

The DBLBUFF.SYS driver provides compatibility for some HDD controllers, which otherwise may be unable to work with EMS memory and with WINDOWS OS. In

particular, double buffering may be needed for HDDs with SCSI interface. The DBLBUFF.SYS driver is included in WINDOWS-95/98 release and normally can be found in \WINDOWS directory.

A way of loading DBLBUFF.SYS depends on contents of configuration parameter file MSDOS.SYS (5.01-01). If there is a line with "DoubleBuffer=1" parameter, then MS-DOS7 will attempt to load DBLBUFF.SYS by default. Otherwise it should be loaded explicitly by DEVICE command (4.06) from a line in CONFIG.SYS file:

```
DEVICE=C:\DOS\DRV\Dblbuff.sys /D+
```

where:

C:\DOS\DRV – path example to DBLBUFF.SYS driver.

/D+ – an optional parameter, instructing to arrange permanent double-buffering for all disks. Otherwise only I/O to UMB blocks (5.04-02) will be double-buffered, and it wouldn't be done automatically if it appears to be unnecessary.

Note 1: status message, displayed by SMARTDRV.EXE driver (5.06-01), contains a column "buffering". If there is at least one "yes" in this column, then DBLBUFF.SYS driver must be loaded.

Note 2: if double buffering is needed, then a non-zero number of secondary buffers should be reserved by BUFFERS (4.03) or by BUFFERSHIGH (4.04) command in CONFIG.SYS file.

5.06-03 DRVSPACE.SYS – compressed drive's shell loader

DRVSPACE.SYS is a loader for TSR program DRVSPACE.BIN, which provides on-the-fly compression and decompression for logical disks. Data flow is processed just in the course of access operations. Compression improves disk space usage, since compressed data are written continuously, and free disk space is not lost in partially filled clusters.

Files DRVSPACE.SYS and DRVSPACE.BIN are included in WINDOWS-95 release and normally must be written in the same directory outside compressed region of logical disk (most often in the root directory). When MS-DOS7 discovers presence of a compressed region on the bootable disk, both DRVSPACE.SYS and DRVSPACE.BIN are loaded by default. Default loading may be discarded either by a line with parameter "DRVSPACE=0" in MSDOS.SYS file (5.01-01) or by a "DOS=NOAUTO" command in CONFIG.SYS file (4.08, 9.01-01).

When DRVSPACE.SYS is to be loaded explicitly, it should be loaded before memory managers (5.04-01, 5.04-02) just as a driver by DEVICE command (4.06) from a line of CONFIG.SYS file, for example:

```
DEVICE=C:\Drvspace.sys /MOVE /NOHMA /LOW
```

where:

C:\ – path example to DRVSPACE.SYS loader.

/MOVE – initiates relocation of DRVSPACE.BIN from upper part of conventional memory into UMB or HMA areas, because original placement may be incompatible with other software. Relocation takes place after execution of all DEVICE and DEVICEHIGH commands in CONFIG.SYS file. If address space beyond 640 kb is inaccessible, the /MOVE option induces relocation to lower part of conventional memory.

/NOHMA– don't relocate TSR compression module into HMA (1024 – 1088 kb).

/LOW – forces relocation of TSR compression module into lower part of conventional memory even when address space beyond 640 kb is accessible.

When DRVSPACE.BIN is loaded, the DRVSPACE word becomes a command, invoking a dialog shell. This shell gives an opportunity to convert ordinary disks (and diskettes) into compressed ones and back, to create new compressed disks, to test and defragment compressed disks, to arrange protection of compressed disk space, etc.

Disk compression with DRVSPACE hasn't got much success because of several reasons. The first is incompatibility with other versions of DOS and with other operating systems (WINDOWS-95 is an exception). The second reason is that compression makes disks more vulnerable to possible errors and less accessible for data recovery procedures. The third reason is that nowadays the problem of disk's space is not so acute, as in early 1990-ties. Large and fast modern disk drives make compression "on the fly" not worth that loss of access speed, which is caused by compression and decompression operations. This is why DRVSPACE usage is not included in examples of configuration files, presented in chapter 9.

Note 1: the DRVSPACE.BIN program provides compression for logical disks formatted with file systems FAT-12 or FAT-16. Disks formatted with FAT-32 can't be compressed by DRVSPACE.BIN.

5.07 Interface controller's drivers

5.07-01 ATAPI interface BIOS extension: ATAPIMGR.SYS

Disk drives with Integrated Drive Electronics (IDE) were developed by Western Digital Corp. and were employed for the first time in IBM's PC-AT in 1984. Since then such drives became the most widely used. Protocol of IDE drive's interaction with HDD

controller has been named ATA (= Advanced Technology Attachment). In 1994 it has got status of standard ANSI X3.221-1994. With advent of CD-ROMs and other types of removable drives the ATA protocol was supplemented with new commands, including those for packet data transfer, and since 1998 became known as ATA Packet Interface, or ATAPI.

In computers, produced after 2003, the ATAPI interaction protocol is implemented by PC's BIOS. A specific symptom of ATAPI support is BIOS's ability to load OS from optical DVD discs. In those PCs, which can't load OS from DVD disks, the ATAPI functions may be provided by interface driver ATAPIMGR.SYS, developed under "Panasonic" trade mark by Matsushita Corp. Due to this driver a standard (obsolete) IDE controller becomes able to give access to DVD disks, to removable magnetic, magnetooptical and solid-state media of high capacity. SFX archive 85X_DOS.EXE, containing version 2.04 of ATAPIMGR.SYS driver, can be downloaded from internet site http://panasonic.co.jp/pcc/products/drive/internal/support/info_dd2.html .

The ATAPIMGR.SYS driver should be loaded from CONFIG.SYS file before any other driver, which will need ATAPI interface support. A line loading the ATAPIMGR.SYS driver with DEVICE (4.06) or DEVICEHIGH (4.07) command may look like this:

```
DEVICE=DOS\DRV\Atapimgr.sys /P:170,15 /W:2 /NDR /NRS /C:2 /T:5 /LUN
```

where:

DOS\DRV\ – path example to ATAPIMGR.SYS driver.

/P:170,15 – hexadecimal address of I/O port and decimal interrupt request line number (IRQ). Allowable port addresses are 1F0, 170, 1E8, 168 (A.14-1). Allowable IRQ numbers are 10, 11, 12, 14, 15. If /P parameter is omitted, the driver will search for disk drives throughout all mentioned port addresses and IRQ lines.

/W:2 – number of waiting cycles (0 – 99 allowed) for data I/O operations in old PCs, not supporting readiness confirmation (IOCHRDY) check. Recommended number of waiting cycles depends on CPU clock frequency: for 50 MHz – /W:2, for 75 MHz – /W:6, for 100 MHz – /W:9, for 166 MHz – /W:19, for 240 MHz – /W:30. For PCs with higher CPU clock frequency the /W parameter is not needed.

/NDR – don't issue the reset command to CD/DVD-ROM drives. This option is essential for booting from a CD/DVD disc, since otherwise the reset command will disrupt booting process.

/NRS – don't return request sense confirmation, when drive sends a request for conditions check.

/C:2 – recalculate PIO mode for a particular drive:

0 – for master drive at primary IDE controller,

1 – for slave drive at primary IDE controller,

2 – for master drive at secondary IDE controller,
3 – for slave drive at secondary IDE controller.

/T:5 – set timeout (in seconds) for waiting drive's response; default is 30 seconds.

/LUN – selectively support addressing to a device with LUN (= Local Unit Number) zero. The LUN numbers enable to regard multifunctional disc drives as different devices, for example, a drive with inserted DVD-RAM disc – as equivalent of hard disk drive, and the same drive with inserted CD-ROM disc – as an ordinary CD-ROM drive. Addressing with LUN numbers is not supported by default.

If "above" MS-DOS7 with loaded ATAPIMGR.SYS the Windows-95/98 OS is launched, the latter will run in "MS-DOS compatibility mode" unless you add the following line into the \Windows\IOS.INI file:

```
ATAPIMGR.SYS  ; MKE ATAPI Manager
```

If ATAPIMGR.SYS driver will find, that ATAPI interface functions are supported yet by PC's BIOS, it wouldn't be loaded. When ATAPIMGR.SYS is not loaded, some CD-ROM drivers (in particular, SR_ASPI.SYS) wouldn't be loaded too. Hence, in cooperation with ATAPIMGR.SYS those CD-ROM drivers should be preferred, which use ATAPI functions regardless of their origin, for example, as OAKCDROM.SYS (5.09-01) and VIDE-CDD.SYS (5.09-02). Such drivers provide access to DVD-ROM discs in any case, whether ATAPIMGR.SYS will "agree" to be loaded or not.

5.07-02 PCMCIA interface drivers

Since early 1990-ies portable PCs of "notebook" class were equipped with special 68-pin slot for external memory expansion cards. Interface for these cards was standardized in 1990 by PC Memory Card International Association (PCMCIA). In 1995 it was renamed into PC card interface, because at that time it was also used for a variety of peripheral equipment: modems, external disk drives, etc. The last 8-th version of PC card interface standard was adopted in 2001. Later PC card interface was ousted by USB 2.0 interface (5.07-05).

In early 1990-ies there were several mutually exclusive types of PCMCIA controllers, and then external devices with PCMCIA interface had to be supplied with a number of PCMCIA drivers for DOS. Thus, CD-ROM drive Panasonic KXL-DN720A (1995) was supplied with 3 drivers for different PCMCIA controllers. These drivers enable to address the device with unified ASPI commands, just as devices with SCSI (5.07-03) or USB interface (5.07-05). This set of drivers, packed into SFX archive 720PCM32.EXE, can be downloaded from server ftp://ftp.panasonic.com/pub/Panasonic/Drivers/CDROM/ .

Later PCMCIA controllers of i82365 type and those compatible with i82365 have ousted all other types. Most modern portable PCs are sold with pre-installed WINDOWS operating system, and are not supplied with PCMCIA drivers for DOS. In order to enable usage of external disk drives with PCMCIA interface for emergency maintenance, some hardware vendors have developed PCMCIA drivers, which directly address from DOS to ports of i82365 controller. In particular, such drivers are proposed by Novac Co: driver NVIHD.EXE for non-optical storage devices and driver NVICDF.EXE for optical disc drives. From site http://www.driver.novac.co.jp/driver/hd150p/hd150p_drv.html you can download archive compact_PCMCIA.zip with version 4.0 (2000) of NVIHD.EXE driver. Version 4.0 (2001) of NVICDF.EXE driver, packed in archive FDOS.ZIP, can be downloaded from http://www.driver.novac.co.jp/driver/sta_PCMCIA/pcm_drv.html .

Both mentioned Novac's PCMCIA drivers should be loaded without parameters by DEVICE (4.06) or DEVICEHIGH (4.07) command from a line of CONFIG.SYS file. Then, after a storage device card is inserted into a PCMCIA slot, the same driver should be launched from command line once more, for example

```
Nvicdf.exe /E
```

or

```
Nvihd.exe  /E  /I
```

where parameters mean:

/E – initialize PCMCIA card;
/I – spin drive's motor.

The NVICDF.EXE driver doesn't need the /I parameter, because motor in optical disc drives is activated automatically, just after insertion of optical disc. Given information about the mentioned PCMCIA drivers should be treated with some caution, since the author had no opportunity to test these drivers.

Note 1: Windows-ME release includes a DOS PCMCIA driver CARDDRV.EXE for external storage devices. However, exact information about it hasn't been found.

5.07-03 SCSI interface drivers

Small Computer's System Interface (SCSI) has been developed in 1982 by Shugart Co. and became widely known due to its implementation in PCs of Apple Co., which were popular then. In 1986 SCSI interface has got status of standard ANSI X3.131-1986. Since then it has been modernized several times. For AT-compatible PCs the SCSI bus is a rare requisite, it is used predominantly in servers. Because of this reason SCSI bus driver's features are not described here in detail. Nevertheless a short survey of SCSI bus access principles is given below.

SCSI bus access may be provided either by PC's BIOS or by special drivers. Server's motherboards usually have embedded SCSI controller and BIOS extensions, enabling to access SCSI devices just as IDE devices, without special SCSI drivers. Known BIOS extensions need at least one active device to be present on SCSI bus when PC starts, and wouldn't be loaded otherwise. Removable disk drives without media inside are considered inactive. If a particular removable SCSI drive must be taken under BIOS control (for example, in order to be formatted as a HDD), you should manage to insert a media in this drive before BIOS launches its start tests. If BIOS senses a valid media inside a SCSI drive, the latter can be used to boot the PC, but beforehand you have to mark this drive as bootable in BIOS setup options. Those SCSI drives, which are initially registered by BIOS as inactive, can't be taken under BIOS control, even if SCSI BIOS extensions are loaded and provide access to other drives.

Some SCSI controllers on extension boards have internal read-only memory, containing BIOS software extensions for SCSI bus, and then SCSI bus access is just as that provided by embedded controllers. You can easily determine presence of BIOS software extensions by exploring BIOS setup options. However, most cheap SCSI extension boards have no BIOS software extensions. SCSI devices, which are not supported by BIOS software extensions, can't be used to boot the PC, and access should be arranged by drivers. This form of access is preferable for removable disk drives, because many drivers are able to register each current media partition structure (while most BIOS versions register media partition structure only once at start-up).

SCSI driver sets for DOS have been developed by several vendors; the most known are Adaptec, DTC, Mylex, Tekram. Each driver set consists of SCSI controller driver (ASPI8U2.SYS from Adaptec, AS80DOS.SYS from DTC, etc.), HDD driver for SCSI (ASPIDISK.SYS from Adaptec, DISKDOS.SYS from DTC, etc.) and a CD-ROM driver for SCSI (ASPICD.SYS from Adaptec, CDDOS.SYS from DTC, etc.). Drivers of either set must be loaded by DEVICE or DEVICEHIGH command from a line in CONFIG.SYS file. SCSI controller driver always must be loaded in advance, before drivers for other devices connected to SCSI bus.

Drivers for HDDs and CD-ROMs, proposed by SCSI controller vendors, accept standardized ASPI command codes and are suitable for almost any device of the same class with SCSI interface. This is not true for all devices: those having unique functions definitely need proprietary drivers. Most SCSI controller drivers are suitable for series of SCSI controller types. Emergency diskettes for Windows-95/98 include two sets of SCSI drivers: from Adaptec and from Mylex. This is often thought to be enough for almost any PC with SCSI bus. In any case configuration file(s) on these diskettes may be taken as example of loading drivers for SCSI bus.

5.07-04 IEEE1394 (FireWire) interface drivers

The FireWire interface has been developed by Apple Company in 1987 for connecting devices having large data transfer rates, up to 393 Mb/sec. Such data transfer rates are typical for video cameras, for video storage devices, for external hard disk drives. Since 1995 the FireWire interface specification has been adopted as standard IEEE1394. For AT-compatible PCs expansion cards with IEEE1394 interface controllers are produced, but up to now the FireWire interface hasn't become widely used.

Now two drivers for IEEE1394 interface controllers are known, which are claimed to be suitable under MS-DOS7. The first is ASPI1394.SYS driver, developed by Iomega Company. From site http://www.stefan2000.com/darkehorse/PC/DOS/Drivers/USB/ you can download archive iomega_usb_firewire_dos_driver_boot_disk.zip with version 1.01 of ASPI1394.SYS driver (dated 2002). Just there is an example of loading this driver, but explanation of parameters isn't given. Version 1.02 of another driver – SBP2ASPI.SYS, developed by Medialogic Company, is contained in SFX archive DAT.EXE, which can be downloaded from site http://www.datoptic.com/fw25fr.html .

Since both mentioned drivers implement the same unified set of ASPI commands, hence for HDDs and for CD-ROM drives, connected via IEEE1394 bus, potentially just those drivers are suitable, which are used for devices of the same class on SCSI bus (5.07-03) and on USB bus (5.07-05). But this feature hasn't been tested by the author, and is reported here for contemplation only.

5.07-05 USB interface drivers

Universal Serial Bus (USB) is a joint development of Compaq, Intel, Microsoft and NEC. In 1996 these companies have adopted version 1.0 of USB bus specification. Since then an embedded USB controller has become ordinary requisite of motherboards in almost all AT-compatible computers. Practical importance of USB bus has increased even more in 2002 with adoption of USB 2.0 specification, stipulating data transfer rates up to 480 Mb/s. Now a vast majority of external devices is designed for connection via USB bus.

USB controllers implement three different types of interaction with PC's software:

– Open Host Controller Interface (OHCI),
– Universal Host Controller Interface (UHCI),
– Enhanced Host Controller Interface (EHCI).

Each type of interaction needs a specific treatment. All modern USB controllers match USB 2.0 specification and implement EHCI type of interaction, but are able to simulate USB controllers of previous generation, implementing either OHCI or UHCI type of interaction. UHCI controllers transfer data via an allotted port, whereas OHCI controllers

transfer data via a buffer zone in memory. The OHCI type of interaction is implemented by chipsets from SIS and ALI, whereas UHCI – by chipsets from INTEL, VIA and some others.

Access from DOS to peripheral devices, connected via USB bus, can be provided either by PC's BIOS system or by loaded drivers. Access provided by BIOS is necessary, when operating system must be loaded from an external storage device. Solutions of this problem are considered in detail in article 9.11-01. But PC's BIOS is not necessarily ready to cope with any external device: some BIOS systems accept external devices of a certain class only, for example, floppy drives with USB interface. Besides that, BIOS systems register properties of storage media only once, just after PC is switched on. If, for example, in a USB adapter slot the originally inserted storage card will be replaced by some other card, then BIOS wouldn't provide access to that other card. An opportunity to access removable storage media can be obtained by loading an appropriate driver for storage devices of a particular class.

In order to avoid possible conflicts between driver(s) and PC's BIOS, parameter "Legacy USB support" on page "Advanced" of BIOS Setup settings should be set to "Disabled". If there is no similar parameter(s) in BIOS Setup settings of your computer, then, most probably, this BIOS system doesn't support USB storage devices. Access to media in these devices is still possible, but only by means of appropriate driver(s). Generally, USB controller driver should be loaded first, and then – drivers for all USB devices, which you intend to use.

The very first drivers for USB controllers – UHCI.EXE and OHCI.EXE – were developed in 1998 – 2001 by SoftConnex Co. Version 2.3 of these drivers is present in bootable diskettes, formed by a known software packet Norton Ghost (up to it's version 8.0). Version 2.5 of the same drivers can be downloaded from internet site http://www.stefan2000.com/darkehorse/PC/DOS/Drivers/USB/ . In order to suffice any USB controller, the mentioned drivers should be loaded sequentially, one after the other, but actually only one of them will be loaded – the one that corresponds to interaction type, implemented by a particular USB controller. Loading may be performed either just from command line, or by LH command (3.17) from a line of AUTOEXEC.BAT file, or else by DEVICE (4.06) or DEVICEHIGH (4.07) command from a line of CONFIG.SYS file, for example:

```
DEVICEHIGH = \DOS\DRV\Uhci.exe
DEVICEHIGH = \DOS\DRV\Ohci.exe
```

Drivers from SoftConnex Co. are devised for USB keyboards, for USB mouse pointing devices and for external ports on USB bus. Obviously, a driver for a particular external device should be loaded afterwards. In particular, for mouse pointing devices any of those drivers will suit, which are described in part 5.03. But you have to connect your external device to port of the first USB controller, because all other USB controllers, which may be present in modern computers, can't be detected by drivers from SoftConnex Co. Besides

that, these drivers don't provide functions for access to storage cards and to external disk drives.

Problem of access from DOS to USB storage devices has become especially urgent now. There is a lot of different advices about that in various internet sites. Author of this book had to undertake some experiments in order to form his own opinion. In these experiments the role of a drive under test was played by USB adapter ImageMate-2 for Compact Flash cards.

Among USB controller's drivers the USBASPI.SYS driver from Matsushita (trade mark Panasonic) has been acknowledged the best. Latest version 2.27 of this driver (dated 22.10.2008), packed into SFX archive F2H_USB.EXE, can be downloaded from site http://panasonic.co.jp/pcc/products/drive/other/f2h_usb.html . This driver is able to detect all active USB controllers of any type. Then it scans each USB bus, and registers all connected devices with all their LUN numbers (see note 1 to appendix A.03-2 for details). Unlike most other USB drivers, version 2.27 of this driver doesn't cause conflicts with PC's BIOS when "Legacy USB support" parameter is enabled. Complete specification for USBASPI.SYS driver isn't made public. Nevertheless the following options have been discovered:

/e	– activate USB controllers of EHCI class only.
/o	– activate USB controllers of OHCI class only.
/u	– activate USB controllers of UHCI class only.
/nocbc	– don't search for USB adapters in PCMCIA slots.
/w	– display a prompt for the user, that he has to connect and to switch on external device(s), which should be registered on USB bus.
/slow	– decrease speed of polling the connected devices, so that even the most lazy devices have enough time to respond.
/v	– display data about registered USB controllers and about all devices, detected on USB bus(es).
/r	– don't decline loading of the driver because of errors and in case of BIOS's control over USB controller(s).
/norst	– don't send the RESET command to USB devices in order to prevent disruption of OS loading process from one of those devices.

The /e, /o, /u parameters give a chance to save some time, if class of USB controller(s) in a particular computer is known in advance. The last two parameters (/r and /norst) are necessary, when operating system is to be loaded from a USB device under BIOS control, because otherwise either the loading process will be disrupted or else other USB devices will be left inaccessible. One more warning: the USBASPI.SYS driver doesn't allow loading of drivers for optical disc drives in advance, even if these disc drives have a non-USB interface.

The best driver for mass storage devices is still ASPIDISK.SYS, originally developed by Adaptec for hard disk drives with SCSI interface. But ASPIDISK.SYS has no direct

relation to interface type, because it addresses storage devices via standardized set of ASPI functions, provided by SCSI controller's driver. Since just this set of ASPI functions is provided by USBASPI.SYS driver too, the ASPIDISK.SYS driver is able to cope with storage devices, connected to USB bus.

Version 4.01b of the ASPIDISK.SYS driver (size 15060 bytes, dated 02.12.1998) can be got inside SFX archive DOSDRVR.EXE from Adaptec's internet site http://www.adaptec.com/en-US/speed/scsi/dos/dosdrvr_exe.htm . This driver gives access to logical disks with file systems FAT-12, FAT-16, FAT-32 and "Big Floppy". If removable media is not present in the storage device at the moment of initialization, then this storage device will be given one of reserved letter-names. But there are removable media, which may be formatted as hard disk drives with several FAT-16 partitions, each representing a separate logical disk. The ASPIDISK.SYS driver can provide access to all such partitions (including those beyond the first), if an appropriate number of disk's letter-names is reserved beforehand by specifying this number after the /r parameter in command line. The whole set of allowed command line options includes the following:

/id=2:0+1 – an example of specification for devices, which should be taken under driver's control: the device number 2, connected to USB bus of the first USB controller, and devices number 0 and 1, connected to USB bus of the second USB controller.

/nospinup – don't issue command to switch on drive's motor at the moment of driver's initialization.

/d – display status message about devices, which are taken under driver's control.

/pause – suspend further execution until any keystroke in order to give an opportunity to read the status message.

/r4 – an example of request to reserve 4 disk's letter-names for non-first partitions on removable media in drives, taken under driver's control. From 1 to 24 letter-names can be reserved.

If specification for controlled devices is not given in command line, then the ASPIDISK.SYS driver will examine each encountered device and will attempt to take under it's control all mass storage devices, even those having no removable media inside at that moment. When specification for controlled devices is given, no time will be lost for examination of unsuitable devices. In the example above the specification of device number 2 means that driver will not examine devices 0 and 1, connected to bus of the first USB controller, since these devices may happen to be, for example, a scanner and a printer.

For external optical CD/DVD-ROM drives with USB interface the NJUSBCDA.SYS driver from Workbit Corp. may suffice. Archive BST_DOS.ZIP, containing version 3.9 of NJUSBCDA.SYS driver (dated 2000), can be downloaded from internet site http://www.driver.novac.co.jp/driver/sta_black/bst_drv.html . As many other CD/DVD-ROM drivers, NJUSBCDA.SYS accepts from command line the /D: parameter, followed

by an arbitrary identifier (for example, /d:USBCD001) of up to 8 characters long. It is used for drive's identification by file system translation utility – either MSCDEX.EXE (5.08-03) or SHSUCDEX.EXE (5.08-04), one of which should be loaded afterwards. It should be given exactly the same identifier in its command line.

The described drivers of USB devices should be loaded by DEVICE (4.06) or by DEVICEHIGH (4.07) commands from lines of CONFIG.SYS file. Let's assume, that all mentioned drivers are present in the \DOS\DRV directory of bootable disk; then lines for loading these drivers may look, for example, as

```
DEVICEHIGH=\DOS\DRV\Usbaspi.sys /slow /v
DEVICEHIGH=\DOS\DRV\Aspidisk.sys /nospin /d /pause
DEVICEHIGH=\DOS\DRV\Njusbcda.sys /d:USBCD001
```

Parameters in the shown example are selected for loading MS-DOS7 from a device with some other (non-USB) interface, when it is not known in advance, how many USB controllers there are in the PC and which devices are connected to USB bus(es). But when PC's hardware is known, and MS-DOS7 loading is performed not for the first time, then parameters /SLOW and /PAUSE may be omitted. Besides that, loading goes faster, if parameters specify the type of USB controller and particular devices, which should be registered.

The described USBASPI.SYS driver, developed by Matsushita (version 2.27, file length 39729 bytes), is often messed with synonymous driver, supplied by Novac (version 1.07, file's size 43528 bytes). The latter driver detects only the first USB controller, works according to the USB 1.1 specification exclusively, ignores all devices with non-zero LUN number (note 1 to appendix A.03-2), and accepts from command line a somewhat different set of options:

- /w – display a prompt for the user, that he has to connect and to switch on external device(s), which should be registered by USB controller.
- /v – display data about detected devices on USB bus of the first USB controller.
- /r – don't unload driver's resident module when the first USB controller is under control of PC's BIOS.
- /m=D0 – an example of contact memory zone specification for USB controller of OHCI class. D0 denotes segment addresses zone D000 – DFFFh.
- /p=A400 – an alternative example of port address specification for USB controller of UHCI class.

The Novac's USBASPI.SYS driver, packed into archive HD352u_dos.zip, can be downloaded from site http://www.driver.novac.co.jp/driver/hd352u/hd352u_drv.html . Besides the USBASPI.SYS driver itself, the HD352u_dos.zip archive contains version 2.0 of the Novac's driver Di1000dd.sys for mass storage devices. The Di1000dd.sys driver accepts from command line parameter

/dS – an example of disk's letter-name ("S"), which should be appointed to the logical disk, opened for access by the Di1000dd.sys driver.

However, letter-name appointment by the Di1000dd.sys driver goes "dirty", created dummy disks (phantoms) are not hidden. The Di1000dd.sys driver can't cope with multiple LUN numbers, can't provide access to disks with FAT-32 file system, to devices beyond the bus of first USB controller, needs a media to be present in storage device(s) at the moment of initialization. Though capabilities of both mentioned Novac's drivers seem substantially limited, they may be sufficient for performing main data transfer operations in computers with known hardware.

One more way of access to USB storage devices from DOS is opened by combined driver DUSE.EXE, developed by Cypress Semiconductor. Version 4.9 of this driver (dated 2003), packed into archive Duse_4_9.zip, is available in subdirectory "downloads" of site http://www.pocketec.net/support.taf . The DUSE.EXE driver combines USB controller driver with driver for CD-ROMs and with driver for non-optical storage devices, including HDDs and solid-state storage cards.

The opportunity to get all-in-one seems attractive, but is accompanied with unexpected and unpleasant surprise. Because of a strange reason DUSE.EXE requires DOS working in real mode only, that is without the EMM386.EXE driver (5.04-02), which gives access to UMB memory region. Having no access to UMBs, DOS is forced to load all drivers into conventional memory (below 640 kb), and then resident module of DUSE.EXE with default settings occupies yet more 233 kb of precious conventional memory, so that no free space is left there for normal operation of other programs.

Some tolerable configuration can be obtained, if access to UMB region is provided in real mode by UMBPCI.SYS driver (5.04-04) and if you decline loading of DUSE's CD/DVD-ROM support module. Thus total size of DUSE's resident module has been reduced to 153 kb, but all attempts to load it beyond conventional memory have failed. For comparison: resident modules of USBASPI.SYS and ASPIDISK.SYS drivers provide nearly the same functions, but together occupy 45 kb only and can be loaded beyond conventional memory either in CPU's real mode or in V86 mode as well. If circumstances will force you to use the DUSE.EXE driver, then a line in CONFIG.SYS file for loading DUSE.EXE may look, for example, as

```
DEVICE=\DOS\DRV\Duse.exe NOC EMU XFER=8
```

where the shown options mean::

NOC – decline loading of CD/DVD-ROM support module.
EMU – emulate IRQ calls in order to prevent driver's mutual incompatibility.
XFER=8 – reduce buffer size to 8 kb (1 to 64 kb are allowed, default is 64 kb).

Whole list of DUSE's options can be found in file DUSEUsersGuide.pdf inside the same archive Duse_4_9.zip. The INT option, proposed there among other options, deserves special notice. It claims to provide an opportunity to divide external HDDs into

partitions with FDISK.EXE utility (6.13), but in fact doesn't. Moreover, making HDD partitions with ASPI functions (note 5 to 6.13) was not available too, either with or without the INT option, whereas the USBASPI.SYS driver enables to do it easily.

Two personal attempts to create sets of USB drivers for DOS became known recently. Version 1.0 of driver's set, developed by G.Potthast, appeared in december 2006 at internet site http://www.georgpotthast.de/usb/ . Later, in august 2009, B.Johnson has uploaded version 0.08 of another USB driver's set at his site http://bretjohnson.us/ .

The set, developed by G.Potthast, presents drivers for USB controller, for non-optical storage devices and for some types of USB printers. Besides drivers, it includes a number of service utilities, enabling to detect and to fix errors of USB interface configuration. This set of drivers can't determine actual number of USB storage devices, can't detect devices on bus(es) of non-first USB controller(s), requires a media to be present in storage device at the moment of initialization, provides access to the first partition only of a physical disk and has some other considerable drawbacks.

The set, developed by B.Johnson, besides the same main drivers and some service utilities, includes drivers for USB keyboard, for USB mouse and provides access to disks formatted with FAT-32. USB controller must be either of UHCI or EHCI type and can be used with 12 Mb/s transfer speed only. Probably, some drawbacks are not revealed yet, because this set of drivers appeared in the last moment and wasn't tested thoroughly.

Despite all drawbacks of early versions, those sets of drivers, presented by G. Potthast and by B.Johnson, are worth serious attention. Both sets are still under development, and their future versions may be much better. But the most valuable feature of these driver's sets is their partially opened executable code. There is a lot of interesting details for all those who intend to have a closer deal with USB interface.

5.08 Services for installable file systems

5.08-01 IFSHLP.SYS – IFS service driver

Auxiliary driver IFSHLP.SYS provides service functions for IFS (= Installable File Systems) procedures. IFS file system is a form of hiding real data structures and real ways of access, enabling to avoid explicit technical complications and implement selective access rights.

Original 16-bit access to storage devices is performed by BIOS's INT 13 handlers, which require CPU's real mode and can't do their job indirectly, for example, via a network. When CPU works in protected mode or in V86 mode, each call to INT 13 handler implies a callback with switches to real mode and back. These switches make a slow 16-bit

access even more slower. Service functions, provided by IFSHLP.SYS driver, enable a much faster 32-bit direct and indirect disk access without callbacks in both protected and V86 modes. But If you intend to work in real mode only and without network communications, then, most probably, the IFSHLP.SYS driver will not be needed.

The IFSHLP.SYS driver is present in Microsoft Network Client packet, and also in Windows-3.11\95\98\ME releases. MS-DOS7 searches in the \Windows directory for the IFSHLP.SYS driver and loads it by default, unless default loading is forbidden by DOS=NOAUTO command (4.08) in CONFIG.SYS file.

5.08-02 NTFSDOS.EXE – NTFS file system translator

An opportunity of access from DOS to disks with NTFS file system is provided by driver NTFSDOS.EXE, written by Mark Russinovich and Bryce Cogswell. Full-functional versions of this driver (latest the 5-th) were not free. Here a freeware version 3.02 (dated 2001) is described, which enables only reading NTFS volumes, including reading programs into memory for execution. However, NTFSDOS driver and several other helpful items disappeared from their author's personal site since 2006, when the authors joined Microsoft's developers team. Now archive Ntfs30r.zip containing NTFSDOS.EXE driver is still available from, for example, server ftp://ftp.uni-koeln.de/pc/msdos/diskutils/ and from http://web.archive.org/web/20020123013310/www.sysinternals.com/new.shtml .

It's just the moment to remind, that Windows-2000/XP installation program, being launched from a CD-ROM disc, gives an opportunity to open the so called Recovery Console. Default settings of the Recovery Console enable writing to NTFS volumes, but prohibit copying of file(s) from NTFS volumes. Consequently, version 3.02 of the NTFSDOS.EXE driver enables to do just what is not allowed by Recovery Console.

The NTFSDOS.EXE driver uses XMS-memory and therefore needs the HIMEM.SYS driver (5.04-01) to be loaded beforehand. Besides that, NTFSDOS.EXE needs much space in conventional memory. In particular, for access to a 10 Gb NTFS disk the NTFSDOS.EXE driver occupies 285 kb of conventional memory. Because of such memory requirements it's not reasonable to keep the NTFSDOS.EXE driver constantly loaded. Therefore it is loaded from command line just before access to NTFS disk becomes necessary, for example:

```
C:\DOS\DRV\Ntfsdos.exe /L:K /C:1024 /N /X /U /V
```

where:

C:\DOS\DRV\ – path example to NTFSDOS.EXE driver.

/L:K – an option, assigning the "K" letter-name to the first found NTFS disk. The next NTFS disks, if there are any, will be given next letter-names

after "K". If /L: parameter is omitted, NTFS disk(s) name assignment will start from the first free letter-name.

/C:1024 – an option, forcing to create a 1024 kb cache buffer in XMS memory. Default cache buffer size is 500 kb.

/N – this option prevents loading of decompression module. It is not needed, when NTFS volume contains no compressed fragments. The /N option decreases conventional memory requirements.

/X – this option forbids usage of extended INT 13 functions (8.01-55). The /X parameter should be specified for old PCs, produced before 1996 and having a HDD not larger than 8.4 Gb.

/U – this option enables translation of names written in unicode (two bytes per character).

/V – this option forces to display driver's memory usage. By default assigned disk's letter-names only are displayed.

Note 1: NTFSDOS.EXE driver loads handlers, enabling to read long names of files and directories in NTFS volume(s). Non-truncated names are displayed by Volcov Commander file manager (6.25) and can be got by any program, which is able to address the mentioned handlers. However, file copying commands and utilities in DOS truncate long names.

Note 2: attempts of access to disk(s) with damaged NTFS file system may cause PC's hanging. Suspected NTFS file system should be checked and fixed in advance by CHKDSK procedure, provided by Recovery Console.

5.08-03 MSCDEX.EXE – optical disc's file system translator

MSCDEX.EXE is a resident program, which cooperates with one or more CD/DVD-ROM drivers, enables disk's letter-names assignment and access to the associated logical disks. In fact it is a file system translator for CD-ROM's file systems "High Sierra" and ISO 9660, which differ from those "understandable" to DOS' core.

The MSCDEX.EXE program is contained in Windows-95/98 releases and normally can be found in \Windows\Command directory. MSCDEX.EXE should be loaded after all necessary CD-ROM drivers, but before SMARTDRV.EXE (5.06-01) cache driver, if the latter is used. Most often the MSCDEX.EXE program is loaded from CONFIG.SYS file with INSTALL (4.15) or INSTALLHIGH (4.16) command, but may be launched from AUTOEXEC.BAT file with LH command (3.17) or just from command line, for example:

```
C:\DOS\DRV\Mscdex.exe /D:MSCD001 /e /k /s /v /L:N /M:12
```

where:

C:\DOS\DRV\ – path example to MSCDEX.EXE file.

/D:MSCD001 – is an example of arbitrary identifier ("MSCD001") of up to 8 characters long for association with a particular CD/DVD-ROM driver, which must be loaded yet and must be given the same identifier in its command line. If there are several CD/DVD-ROM drivers, these must be given different identifiers, and each must be specified after a separate /D: parameter in command line loading MSCDEX.EXE.

/e – this option gives preference to arrangement of buffers beyond conventional memory, provided that access beyond conventional memory is opened yet by EMM386.EXE memory manager (5.04-02).

/k – this option gives preference to secondary volume descriptor with Japanese characters (Kanji), if it can be found. By default primary descriptor is used, secondary descriptor even is not searched for.

/s – this option forces to make preparations for subsequent loading of network server software; this helps to avoid letter-name allocation conflicts and enables sharing of CD-ROM logical disks.

/v – display information about status of CD-ROM drives.

/L:N – optional assignment of letter-name "N:" to the drive, which corresponds to the first /D:MSCD001 identifier; drive(s) related to the following identifier(s), if there are any, will be given the next letter-names. By default CD/DVD drive(s) are given the nearest spare letter-names, but MSCDEX.EXE can't exceed the limit, set by LASTDRIVE command (4.17) in CONFIG.SYS file.

/M:12 – is an example of reserving 24 kb of memory for arranging 12 buffers (2048 bytes each) in order to increase reading speed; figures between 4 and 64 are recommended, 12 is the default.

Note 1: resident module of MSCDEX.EXE program accepts other program's requests via INT 2F\AX=1500–150Fh (8.03-13 – 8.03-19).

5.08-04 SHSUCDX.COM – optical disc's file system translator

In recent years several amendments to CD/DVD-ROM file system specifications have been adopted, stipulating usage of long filenames. The MSCDEX.EXE program (5.08-03) can't cope with modified file systems on optical discs: it shows truncated long names, but gives no access to such files. This drawback isn't inherent to SHSUCDX.COM program; except this, SHSUCDX.COM is a close functional equivalent of MSCDEX.EXE. Development of SHSUCDX.COM program has been started by John McCoy and now is continued by Jason Hood. Archive Shcdx302.zip, containing version 3.02 (dated 2005) of SHSUCDX.COM program, can be downloaded from http://www.shsucdx.adoxa.cjb.net/ .

The SHSUCDX.COM program can be loaded from a line of AUTOEXEC.BAT file by LH command (3.17) or just from command line, but usually it is loaded by INSTALL (4.15) or by INSTALLHIGH (4.16) command from a line of CONFIG.SYS file, for example:

```
INSTALLHIGH=\DOS\DRV\Shsucdx.com /D:?CD001,N,0,2 /~+ /R /Q
```

where

\DOS\DRV\ – path example to SHSUCDX.COM file

/D:?CD001 – specification of an arbitrary identifier ("CD001") of up to 8 characters long for association with a particular CD/DVD-ROM driver, which must be loaded yet and must be given the same identifier in its command line. If there are several CD/DVD-ROM drivers, these must be given different identifiers, and each must be specified after a separate /D: parameter in command line loading SHSUCDX.COM. The question mark preceding the identifier marks those drives, which may be absent in some configurations, since PC's hardware may be not known beforehand.

,N,0,2 – a group of options for that CD/DVD-ROM driver, which is given the same identifier after the /D: parameter. The first letter ("N") is an example of letter-name to be appointed to the corresponding logical disc. The second digit ("0") is an example of disc drive number in a list of disc drives, controlled by this driver. The third digit ("2") is number example of disc drives, accepted by SHSUCDX.COM program from this driver. By default all disc drives will be accepted, and next disc drives will get letter-names, following the specified letter-name or the first spare letter-name.

/~+ – this option forces to insert the tilde sign (~) as the last character in long names, truncated to standard length (8 characters). Action of this option may be cancelled later by launching the SHSUCDX.COM program from command line with inverse parameter /~–.

/R – this option prescribes to take off the "Read only" attribute from files, copied from non-writable optical discs.

/Q – this option forbids display of status message, the letter-name appointment(s) only will be displayed. If you want just nothing to be displayed, you should specify the /QQ parameter instead.

Besides the parameters shown above, SHSUCDX.COM program can accept the following options:

/L:N – an alternative form of prescription to appoint the specified letter-name ("N") to the first optical disc drive, taken under control of SHSUCDX.COM program. The next disc drives, if there are any, will be given next letter-names. The /L: parameter is equivalent to synonymous parameter of MSCDEX.EXE program (5.08-03). The

SHSUCDX.COM program accepts the /L: parameter if only additional group of options to the /D: parameter is not specified.

/C – load resident module of SHSUCDX.COM program into conventional memory. By default resident module is loaded in UMB address space, if it is accessible.

/V – show some more details in displayed status message. Having been loaded in advance, SHSUCDX.COM program can be launched with this parameter once more from command line. Obviously, the /V parameter is incompatible with /Q and /QQ parameters.

/U – unload resident module of SHSUCDX.COM program from memory. This parameter is accepted from command line only, provided that resident module is loaded yet. Unloading releases occupied memory and disables all logical disks, controlled by that resident module.

Note 1: data transfers, arranged by SHSUCDX.COM program, are not supported by SMARTDRV.EXE cache buffer driver (5.06-01), but are supported by UIDE.SYS cache buffer driver (note 3 to 5.06-01).

5.09 Drivers for optical disc drives

Optical disc drives with different interfaces usually need different drivers. Some drivers for optical disc drives with SCSI and USB interfaces are mentioned in articles 5.07-03 and 5.07-05. But in AT-compatible PCs a vast majority of internal disc drives has the IDE (ATA) interface, and drivers for such disc drives are presented here, in part 5.09.

Popular types of motherboards are equipped with two IDE controllers. Each of them enables to connect two devices, which may be optical disc drive(s) and magnetic hard disk drive(s) as well. Connection of optical and magnetic drives to one IDE controller is allowed, but can't be recommended, since relatively slow optical disc drives cause reduction of total data transfer rate. It's better to connect optical device(s) to a separate IDE controller, which must be enabled in settings of BIOS Setup program.

In modern PCs the BIOS Setup programs propose several operation modes for IDE controllers, including that with polling serial ATA buses only (S-ATA). Since most optical disc drives have parallel ATA interface, a compatible operation mode should be preferred. If BIOS Setup program enables to affect direct memory access (UltraDMA), then an ordinary mode of IDE access should be set ("Legacy IDE mode"). These settings guarantee proper operation of optical disc drives and drivers in modern PCs under MS-DOS7.

5.09-01 OAKCDROM.SYS – CD-ROM driver from OTI Corp.

The OAKCDROM.SYS driver is developed by Oak Technology for CD-ROM disc drives with standard IDE interface. OAKCDROM.SYS file, dated 1997, is present in WINDOWS-95/98 emergency diskettes. The same version of OAKCDROM.SYS can be downloaded from site http://www.computerhope.com/download/hardware.htm#02 .

In modern PCs, equipped with a DVD drive, the OAKCDROM.SYS driver enables access to both CD and DVD discs. But in PCs, produced before 2003, presence of a DVD drive is not enough, since BIOS system may provide no support for ATAPI protocol (see 5.07-01 for details), and then OAKCDROM.SYS will enable access to CD discs only. In such computers access to DVD discs is still possible, but requires ATAPI protocol support module to be installed in advance by ATAPIMGR.SYS driver (5.07-01).

The OAKCDROM.SYS driver should be loaded by DEVICE (4.06) or DEVICEHIGH (4.07) command from a line of CONFIG.SYS file, for example:

```
DEVICEHIGH=\DOS\DRV\Oakcdrom.sys /D:MSCD001 /V
```

where:

\DOS\DRV\ – path example to OAKCDROM.SYS driver.

/D:MSCD001 – announcement of an arbitrary identifier up to 8 characters long. This identifier enables recognition of the driver by MSCDEX.EXE program (5.08-03) or by SHSUCDX.COM program (5.08-04). One of these should be loaded afterwards, and it must be given the same /D: parameter followed by exactly the same identifier in its command line.

/V – this option causes status message display.

The OAKCDROM.SYS driver is able to search for CD/DVD-ROM disc drives throughout all IDE-controllers having typical port addresses (1F0h, 170h) and typical interrupt request (IRQ) line numbers. If PC is equipped with several such disc drives, the OAKCDROM.SYS driver will take under its control all drives, which will be found.

5.09-02 VIDE-CDD.SYS – CD-ROM driver from Acer Co.

Version 2.14 of the VIDE-CDD.SYS driver, dated 1998, is a close functional equivalent of OAKCDROM.SYS driver (5.09-01), but VIDE-CDD.SYS is more compact (size 11.8 kb) and can accept port address(es) together with IRQ line number(s) from command line. The latter feature is important when non-standard interface specifications are used, and search for disc drives must be avoided. SFX archive Apicd214.exe, containing VIDE-CDD.SYS driver, can be downloaded via internet, for example, from ftp://ftp.benq.co.uk/cd-rom/drivers/apicd214.exe .

If PC's BIOS doesn't support ATAPI protocol, then VIDE-CDD.SYS can provide access to DVD discs, but needs the ATAPIMGR.SYS driver (5.07-01) to be loaded in

advance. VIDE-CDD.SYS must be loaded from a line in CONFIG.SYS file with DEVICE (4.06) or DEVICEHIGH (4.07) command, for example:

```
DEVICEHIGH=\DOS\DRV\Vide-cdd.sys /D:MSCD001 /P:170,15
```

where:

\DOS\DRV\ – path example to VIDE-CDD.SYS driver.

/D:MSCD001 – announcement of an arbitrary identifier up to 8 characters long. This identifier enables recognition of the driver by MSCDEX.EXE program (5.08-03) or by SHSUCDX.COM program (5.08-04). One of these should be loaded afterwards, and it must be given the same /D: parameter followed by exactly the same identifier in its command line.

/P:170,15 – an optional specification for port base address and for interrupt request line number (IRQ).

There may be more than one /P parameter specified in one line. When at least one /P parameter is given, all other port addresses and IRQ line numbers will not be examined. When /P parameter is omitted, a search for CD/DVD-ROM drives will be initialized throughout all typical IDE port addresses and IRQ line numbers: /P:1F0,14, /P:170,15, /P:1E8,12, /P:168,10 (A.14-1). The VIDE-CDD.SYS driver will take under its control all optical disc drives, which will be found.

5.09-03 QCDROM.SYS – a freeware CD/DVD-ROM driver

Version 4.2 of QCDROM.SYS driver was developed in 2007 by J. R.Ellis on the basis of his earlier driver XCDROM.SYS. Contrary to the latter, the QCDROM.SYS driver is able to provide access to DVD discs and doesn't need ATAPI protocol support neither from PC's BIOS, nor from ATAPIMGR.SYS driver. The QCDROM.SYS driver can control up to three CD/DVD-ROM drives, connected to IDE controller(s) with standard port base address(es) and standard interrupt request line(s): 1F0h with IRQ 14 and/or 170h with IRQ 15.

The QCDROM.SYS driver, packed into archive QCDROM42.ZIP, can be downloaded from internet site http://cyberia.dnsalias.com/Cyb.05.Htm .

QCDROM.SYS should be loaded by DEVICE (4.06) or DEVICEHIGH (4.07) command from a line of CONFIG.SYS file, for example:

```
DEVICEHIGH=\DOS\DRV\Qcdrom.sys /D:MSCD001 /L
```

where:

\DOS\DRV\ – path example to QCDROM.SYS driver.

/D:MSCD001 – announcement of an arbitrary identifier up to 8 characters long. This identifier enables recognition of the driver by MSCDEX.EXE program (5.08-03) or by SHSUCDX.COM program (5.08-04). One of

these should be loaded afterwards, and it must be given the same /D: parameter followed by exactly the same identifier in its command line. If the /D: parameter is omitted, QCDROM.SYS will appoint default identifier QCDROM1.

/L – this option prescribes to refrain from direct memory access (DMA) beyond conventional memory (640 kb). This may be necessary, when PC's memory controller doesn't support DMA access to UMB memory region, while UMBs are opened for ordinary access, for example, by UMBPCI.SYS driver (5.04-04). If the /L option is specified, then data transfer is performed via a buffer in XMS-memory; therefore in this case the HIMEM.SYS driver (5.04-01) must be loaded in advance.

Besides the shown parameters, QCDROM.SYS driver can accept from command line the following options:

/A – forces to use old alternate IDE controller addresses 01E8h–01EFh on primary channel, and 0168h-016Fh on secondary channel. This may be required for "odd" SATA BIOS or other unusual cases.

/I – forces QCDROM.SYS to arrange its own XMS buffer. This enables to avoid possible conflicts with "odd" DMA and BIOS services.

/UF – prescription to enable accelerated direct memory access (UltraDMA). Feasibility of UltraDMA is confirmed for many motherboard's chipsets, but not for all. A check whether the /UF parameter is permissible can be recommended for each particular case.

/UX – refrain from UltraDMA usage, even if both disc drive and motherboard claim support for UltraDMA. The /UX option is used for diagnostics and testing purposes. When /UX option is specified, the QCDROM.SYS driver doesn't need XMS memory.

/PM – (= Primary Master): don't search for disc drives, but check presence of master disc drive only on the bus of primary IDE controller.

/PS – (= Primary Slave): don't search for disc drives, but check presence of slave disc drive only on the bus of primary IDE controller.

/SM – (= Secondary Master): don't search for disc drives, but check presence of master disc drive only on the bus of secondary IDE controller.

/SS – (= Secondary Slave): don't search for disc drives, but check presence of slave disc drive only on the bus of secondary IDE controller.

Up to three parameters, prohibiting search for disc drives, may be specified in one command line. Disc drives will be numerated according to order of corresponding parameters in command line. If specified checks don't reveal presence of disc drive(s), then all other optical disc drives will be ignored, and resident module of QCDROM.SYS driver wouldn't be loaded.

5.09-04 DVS.SYS – CD/DVD-ROM driver from DVS Corp.

The DVS.SYS driver has been developed in 1999 by Digital Video Systems Corp. DVS.SYS driver is able to provide access to DVD discs and doesn't need ATAPI protocol support neither from PC's BIOS, nor from ATAPIMGR.SYS driver. Version 1.1 of DVS.SYS driver, packed into SFX archive Drdvdwd.exe, can be downloaded from internet site http://web.archive.org/web/20030212210152/www.dr-tech.com/drivers/cdroms.html .

The DVS.SYS driver should be loaded by DEVICE (4.06) or DEVICEHIGH (4.07) command from a line of CONFIG.SYS file, for example:

```
DEVICEHIGH=\DOS\DRV\Dvs.sys /D:MSCD001
```

where:

\DOS\DRV\ – path example to DVS.SYS driver.

/D:MSCD001 – announcement of an arbitrary identifier up to 8 characters long. This identifier enables recognition of the driver by MSCDEX.EXE program (5.08-03) or by SHSUCDX.COM program (5.08-04). One of these should be loaded afterwards, and it must be given the same /D: parameter followed by exactly the same identifier in its command line.

The DVS.SYS driver is able to search for disc drives, but performs search more slowly, than other similar drivers. An experiment has confirmed, that DVS.SYS driver can take under its control at least two optical disc drives, connected to either of IDE controllers with standard specifications of port base addresses and interrupt request line numbers (1F0h with IRQ 14 and/or 170h with IRQ 15).

Chapter 6. Selected utilities for MS-DOS7

Functions of command interpreter in MS-DOS7 are complemented and extended by separate executable files (utilities). Several utilities for DOS are supplied within Windows-95/98 OS release. If the origin of utility is not specified explicitly in this chapter, hence, it is one of these Microsoft's utilities, which normally can be found in \Windows\Command directory. Besides these, many other utilities can be successfully used in MS-DOS7, including those from previous versions of DOS and those written by various private and non-private software vendors.

MS-DOS utilities contain internal help text. It can be displayed, when the utility is launched from command line with a single "/?" parameter. Rarely other utilities can be encountered, which display help text when are launched with "–h" parameter or without parameters at all.

Some MS-DOS utilities request DOS version number and wouldn't perform their mission, if the number, returned by DOS, isn't equal to the expected one. It is not necessarily caused by incompatibility: most such version-specific utilities successfully operate under MS-DOS7, when the Setver.exe driver (5.01-02) substitutes expected version number from its table for the actual DOS version number.

Mismatch of DOS version numbers may cause a problem of other kind: when a synonymous utility, belonging to a different DOS version, is encountered in the current directory, the proper utility from MS-DOS7 can't be addressed via the PATH environmental variable (2.02-02), because DOS begins its search in the current directory and finds the improper utility first. In MS-DOS7 version-specific utilities are: Attrib.exe,

Chkdsk.exe, Command.com, Debug.exe, Diskcopy.com, Doskey.com, Fc.exe, Find.exe, Format.com, Label.exe, Mem.exe, Mode.com, Sort.exe, Subst.exe, Xcopy.exe. There is one obvious solution for this problem: name of each version-specific utility in command line should be preceded by full path. It will be wise to prepare special batch files or file manager's menu entries for execution of version-specific utilities with properly specified paths (examples –- in 9.03).

Most examples of utilities usage, presented in this chapter, contain no path specifications. It's implied, that the Setver.exe driver, the PATH environmental variable value and file's placements are prepared in advance so, that each particular utility can be called just by name.

6.01 ATTRIB.EXE – attribute changing utility

In directories each file is represented by a record (A.09-1). Attribute byte 0Bh in this record specifies file's status and rights of access (A.09-2). Attribute byte 0Bh in record(s), related to particular file(s), can be changed by ATTRIB.EXE utility, launched, for example, in the following way:

```
Attrib.exe +R -A C:\DOS\COM\*.txt /S
```

where:

+R –A – set attribute R (read-only) and remove attribute A (prescribed for archiving); there may up to four attributes specified: A, H (hidden), R and S (system), each preceded by "+" (to be set up) or "–" (to be removed). Attributes, which are not mentioned, remain unchanged.

C:\DOS\COM*.txt – an example of path and mask specifications for the files to be processed (= all textual files in C:\DOS\COM directory). If path is not specified, files in current directory are implied. A particular filename may be used instead of a mask.

/S – this option prescribes to continue the search for file(s) to be processed in subdirectories of the specified (or of implied current) directory.

Note 1: if attributes are not specified in command line, the ATTRIB.EXE utility shows which of the requested files have been found and displays a summary of their features, including attributes.

6.02 CHKDSK.EXE – disks checking tool

CHKDSK.EXE is a utility for analyzing and repairing FAT (File Allocation Tables) on diskettes and on hard disk drives, formatted with file systems FAT12, FAT16 or FAT32. By comparing data in first and second FAT tables (and in directories as well), CHKDSK.EXE reveals crosslinks and lost clusters. The latter are transformed into files

with *.CHK suffix, written in the root directory of the same disk. Having finished the check, CHKDSK.EXE shows a summary of results.

Being executed without parameters, CHKDSK.EXE checks the current disk with default settings. Besides this, you may specify the following options:

```
Chkdsk.exe C: /F /V
```

where:

C: – specification example for the disk to be checked.
/F – permission to fix the found errors at once.
/V – prescription to display name of every processed file with full path.

Note 1: the SCANDISK.EXE utility (6.21) performs the all the checks which CHKDSK.EXE does, and some extra checks besides that. Therefore CHKDSK.EXE is worthy only as an information utility, displaying summary of disk's usage.

Note 2: CHKDSK.EXE shouldn't be applied to network drives, CD-ROMs, and to virtual disks, arranged by utilities Assign.com, Subst.exe and Join.exe.

Note 3: the CHKDSK.EXE utility doesn't check whether the files are readable or damaged.

Note 4: the CHKDSK.EXE utility with /F parameter shouldn't be applied to disks suspected of being infected by virus: an antivirus program should be applied first.

Note 5: the CHKDSK.EXE utility shouldn't be applied to disks having from 4085 to 4087 clusters, because it may report nonexistent errors in such disks, and an attempt to fix these errors may destroy stored data.

6.03 CHOICE.COM – choice input utility

The CHOICE.COM utility is intended for arranging interactive menu in the course of batch file(s) execution by command interpreter. CHOICE.COM accepts a character, sent from keyboard or via redirection, and then sets ERRORLEVEL value (3.15-03, 9.07-03) according to the number of the accepted character in a succession of prescribed alternatives. Here is an example of CHOICE.COM usage, charged with its main mission:

```
Choice.com /C:YNC /T:C,10 Yes, No or Continue
```

where the options are::

/C:YNC – the /C: parameter introduces a list of characters to be accepted. Returned errorlevel value corresponds to the order of accepted character in this sequence example: Y – 1, N – 2, C – 3. When the /C: option is not specified, default is YN (Y – 1, N – 2).

/T:C,10 – set waiting time limit 10 seconds (0 – 99 allowed) and then, if no key is pressed, take the C choice as default (errorlevel 3 in this example). When time limit is not set, CHOICE.COM will wait indefinitely. Time

limit 0 forces to make default choice at once: thus the CHOICE.COM utility can be used to set errorlevel.

Yes, No or Continue – an example of a optional prompt to be displayed just before waiting time begins. If final part of line contains a group of words, not preceded by a slash, this group of words will be displayed as prompt message.

Just before prompt message two more optional parameters may be specified:

/S – treat choice characters as case sensitive.

/N – don't append displayed prompt message with a list of alternative choices and with question mark.

Errorlevel values, returned by CHOICE.COM, remain intact during execution of interpreter's internal commands and may be taken into account in order to affect further course of batch file interpretation. Errorlevel values can be examined in the following lines of batch file by a succession of "if errorlevel" conditions (3.15-03) or else within a FOR cycle (3.13), assigning errorlevel value to an environmental variable, for example:

```
FOR %%Z in (1 2 3) DO if errorlevel %%Z set Err=%%Z
```

In a similar way the FOR cycle can be used to perform conditional jumps with "GOTO L%%Z" command, but for this purpose the expected errorlevel values in parenthesis should be enlisted in reverse order.

Quite different application for CHOICE.COM utility is word parsing. The word to be parsed may be, for example, a path, if you have to check writability of specified disk, existence of each directory in the path, etc. Let's assume that file CHECK.BAT contains an analyzing program, which needs the word to be presented letter-by-letter. For this purpose the CHOICE.COM utility should be used as follows:

```
ECHO ; | Choice /S /C:;Anyword; Call Check.bat > Temp.bat
Call Temp.bat
```

Here the word to be parsed (example: Anyword) is enclosed in semicolons, which guarantee proper separation of first and last letters. Besides this, semicolons guarantee non-stop execution because of presence of redirected answer (ECHO ;) among allowable alternatives. Prompt message is represented by group of words "Call CHECK.BAT". Being sent to STDOUT, this output message will be redirected into temporary file TEMP.BAT. After execution of the first line the created TEMP.BAT file will contain the following string:

```
Call Check.bat [;,A,n,y,w,o,r,d,;]?
```

The second of the shown lines calls execution of the TEMP.BAT file. During this execution all letters of the word to be parsed will be presented to CHECK.BAT as its parameters (from %2 and on) and can be analyzed separately one-by-one.

Note 1: if user breaks batch execution by pressing CTRL-BREAK or CTRL-C key combinations, then CHOICE.COM utility returns errorlevel 0.
Note 2: when user presses any other key instead of those expected, CHOICE.COM sends a beep signal (the 07h control symbol) to the console.
Note 3: when CHOICE.COM encounters any error, it returns errorlevel 255.

6.04 COMMAND.COM – command interpreter

Command interpreter COMMAND.COM is a resident program, which presents command prompt, enables execution of commands from command line and automatic execution of commands from batch files as well. File COMMAND.COM contains code of all internal commands, specified in chapter 3.

COMMAND.COM gets control over PC at final booting stage, when IO.SYS interpreter executes the SHELL command (4.26) in CONFIG.SYS file. This first execution loads resident module of COMMAND.COM and arranges its primary (parent) environment with global variables. After that each next execution COMMAND.COM creates a derived (child) environment, inheriting copies of all the variables from the former (parent's) environment.

Contrary to ordinary utilities, COMMAND.COM leaves no reference in its PSP (= Program Segment Prefix, A.07-1) to the parent's PSP, so the parent's environment is preserved hidden with no legal access. Repeated execution of COMMAND.COM enables to change local environmental variables and execute application files otherwise (run step-by-step, for example). Having accomplished its mission, the last loaded resident module of COMMAND.COM can be unloaded out of memory with EXIT command (3.12). Then its derived (child) environment becomes lost with all its variables, active state of the former (parent) resident module is restored, and the former (parent) environment again becomes accessible.

Here is an example of loading the command interpreter, in particular, for step-by-step execution of a single batch file:

```
Command.com C:\dos\ CON /E:1008 /L:512 /U:255 /Y /C R:\Trial.bat
```

where:

C:\dos\ – an example of a path to COMMAND.COM file. This path (with final backslash!) is used to compile a value of %COMSPEC% environmental variable in the new environment. The path item should precede all other parameters. If the path is omitted, then %COMSPEC% variable will inherit its value from the parent environment.

CON – an example of device specification for I/O operations. CON stands for console, that is display for output plus keyboard for input. CON is the

default device, therefore its specification may be omitted. Other device specifications (3.07) are allowed too, but the chosen device must be ready to support interactive activity of COMMAND.COM.

/E:1008 – optional prescription to reserve 1008 bytes of memory for environmental variables (256 – 32768 allowed). Default environment's size is that sufficient for all inherited variables, but not less than 160 bytes.

/L:512 – optional specification of internal buffer's size in bytes (128 – 1024 is allowed, 256 is the default). This size must be large enough for placement of command line with all substitutions of explicit values and commands.

/U:255 – optional specification of input buffer size in bytes (128 – 255 is allowed, 128 is the default). This size sets maximum length of original command line (before alias substitutions are made).

/Y – this option forces step-by-step interpretation of lines in batch files. When used together with /P parameter (permanent loading, see below), the /Y option is ignored.

/C – this optional parameter announces, that the following name is a name of a program to be executed, and that the interpreter's resident module must automatically unload itself after execution of this program. /K or /P parameters should be used instead of /C if the interpreter's resident module ought to stay loaded. The /K parameter does the same, but enables to unload resident module later with EXIT command (3.12). The /P parameter doesn't announce a name of a program, it denotes permanent loading of interpreter's resident module with EXIT command disabled. The /P parameter must be the last in command line, when COMMAND.COM is loaded for the first time with SHELL command (4.26) from a line of CONFIG.SYS file.

R:\Trial.bat – a name example for a file to be executed by COMMAND.COM interpreter. Name of an executable file may be specified after the /C or /K parameter, which must be the last COMMAND.COM's parameter in command line. All following parameters, if there are any, will be regarded as belonging to the specified executable file.

Besides the shown parameters, COMMAND.COM interpreter accepts the following options:

/MSG – load error message texts into memory. In case of an error, which hinders reading of message texts from disk, this option ensures display of an adequate error message.

/LOW – this option forces to load command interpreter's resident module into conventional memory (below 640 kb). The /LOW parameter is used together with /P parameter for permanent loading.

/F – this option prescribes to skip query on possible error(s) and to go on as in the case of user's answer "Fail". The /F parameter is active only when used together with /C parameter for execution of a single command. But the FOR command (3.13) keeps the /F parameter active over all operations of a cycle, and the CALL command (3.02) keeps it active over all operations in a secondary batch file.

/Z – this option forces to display errorlevel value after execution of any utility, which returns errorlevel value.

Note 1: the /C parameter affects keyboard functions used to terminate execution of batch files (1.03).

Note 2: when COMMAND.COM is loaded permanently with /P parameter, its first default task is interpretation of AUTOEXEC.BAT file, which is implied to exist in the root directory of the current disk.

Note 3: when execution of AUTOEXEC.BAT is launched implicitly via the SHELL command (4.26), then the %0 parameter inside AUTOEXEC.BAT can't be used to initiate recursion.

Note 4: having been loaded with /K parameter, COMMAND.COM is able to accept commands from other processes or from command file(s) via input redirection (see 2.04-02, 2.04-05, and also note 1 to part's 6.05 introduction article).

Note 5: being executed, batch files (with *.BAT suffix) share a common environment with their interpreter COMMAND.COM. Ordinary executable files (with *.COM or *.EXE suffix) get a copy of that environment.

6.05 DEBUG.EXE – debugger and mini-assembler

DEBUG.EXE is a specialized command interpreter (debugger), written by Tim Patterson as an instrument for creation of that operating system, which later has been bought by Microsoft and became known as first version of MS-DOS. DEBUG.EXE helps to find out and to fix errors in both program's executable code and PC's hardware settings. Capabilities of DEBUG.EXE are not limited to a set of its internal commands, because these commands enable to assemble just any machine code and to execute it at once. Of course, DEBUG.EXE is not a tool for writing complex programs; it can't compete with high-level languages. But circumstances dictate other criteria, when you can't rely on known compilers, when you have to examine something or to clear up. If you do it with DEBUG.EXE in CPU's real mode, then all AT-compatible computers obediently submit themselves at your disposal. You'll get direct access to disks, to ports, to memory and to code inside executable files.

Command line for launching DEBUG.EXE may include name of a file to be loaded just at start and prepared for debugging, for example:

```
Debug.exe  Trial.com /B /S
```

where:

Trial.com – an example of a file to be examined, and all following items ("/B /S" in this example) are regarded as options for this file (but not for debugger itself!). These options are written into file's PSP (= Program Segment Prefix) just as it is done when the file is loaded for execution by interpreter COMMAND.COM. Exact layout of the file in RAM depends on file's suffix (more about that – in article 6.05-10).

If the file to be loaded is not specified in command line, it may be defined and loaded later with debugger's internal commands "N" (6.05-12) and "L" (6.05-10). The retarded loading gives an opportunity to start code's layout from an arbitrary address. This is important, in particular, for assembling and debugging drivers (details – in article 6.05-18).

Just as COMMAND.COM interpreter, DEBUG.EXE activates command line editing keys (1.05) and accepts commands from command line. If there is any redirected input (2.04-02), DEBUG accepts it as a sequence of commands instead of commands from keyboard. This feature enables to write command sequences into textual command file(s) and then send it to DEBUG.EXE for automatic execution:

```
Debug.exe < Cmnd_txt.scr
```

All specifications of the code to be loaded or written may be included into such command files (examples – in articles 9.02, 9.06, 9.08, 9.10).

Note 1: when interpreter accepts redirected commands from a command file, it loses communication with keyboard via the STDIN channel, and hence data input can't be performed by interrupts INT 21\AH=01h, 06h, 07h, 08h, 0Ah (8.02-02, 8.02-04, 8.02-06). Because of the same reason the interpreter may hang, if there is no command to restore communication with keyboard in the last line of command file. Most often this mission is performed by the "Q" command, terminating each debugger's session. Another way to restore communication with keyboard without termination of debugger's session is shown in article 9.07-02.

Note 2: an enhanced modification of DEBUG.EXE is developed by Paul Vojta. The main difference from original Microsoft's debugger is that enhanced modification is able to "understand" machine commands of modern processors. Archive Debug113.zip, containing version 1.13 (2008) of the enhanced debugger, can be downloaded from site http://www.japheth.de/dwnload4.html . If not specified otherwise, all following articles in part 6.05 are equally applicable to Paul Vojta's enhanced debugger.

6.05-01 DEBUG.EXE: commands and addresses

Having been started, DEBUG.EXE presents its "–" (hyphen) prompt. It means that DEBUG.EXE is ready to accept a command.

Debugger's commands consist of a one-letter or two-letter instruction name, which may be followed by parameters, separated by space(s) or by comma(s). Separators may be omitted, if their absence in any particular position doesn't cause ambiguity. DEBUG.EXE treats numbers as hexadecimal, upper and lower case letters as identical; the only exception from this rule is data comparison with SEARCH command (6.05-16).

Most often the first parameter following command's name specifies a start point memory address. It may be presented in full form – as a segment address and offset (for example, 1FA5:0100), or with an explicit reference to a segment register (for example, CS:0100), or in a short form – as an offset only (for example, 0100). In the latter case segment will be defined by CS (Code Segment) register for "A" (Assemble), "G" (Go), "L" (Load), "P" (Proceed), "T" (Trace), "U" (Unassemble) and "W" (Write) commands; for other commands it will be defined by DS (Data Segment) register. Offset specifications less than 4 digits long are regarded as having preceding zero(s). For example, offset 100 is interpreted as 0100h.

Initial segment address in all segment registers is the same, allocated by DOS on request of DEBUG.EXE in order to lay the code, which is to be debugged. This code will be written not just from the start of the allocated segment, but from a certain shifted point, defined by offset in IP (=Instruction Pointer) register. Initial setting for offset is 0100h (i.e. 256 decimal). Reserved 256 bytes at the start of allocated segment are known as PSP = Program Segment Prefix (details – in appendix A.07-1).

If action of debugger's command is directed to a group of bytes, then this group is defined either by its start point address and length or by start point address (segment:offset) and end point offset. Parameter, specifying size (length) of a group, is marked by preceding letter "L" – for example, L20 specifies a group of 20h bytes. Sum of start offset and length must not exceed FFFFh. If a hexadecimal number in place of length specification is not preceded by letter "L", it is interpreted as end point offset inside the same segment. Separate segment specification for the end point is not allowed. Obviously, end point offset must be greater than start point offset.

Having typed a command, the user initiates its execution by pressing the ENTER (or "CR") key.

The simplest debugger's commands consist of nothing more than a single character and don't need much comments:

Q – ("Quit") – terminate debugger's session, exit to DOS.
? – display a list of debugger's commands.

However, the displayed list of commands presents no intelligible guidelines for their usage. All necessary guidelines are given in this book in the following articles of part 6.05.

Note 1: if you have mistaken typing a command, DEBUG.EXE displays a message "Error" in the following line, and points with a ^ (caret) to the first character it is unable to "understand". Most often just this character is the cause of the error, but sometimes the cause may be a mistake in preceding part of command line.

Note 2: absolute memory address is calculated as a sum of offset with segment address, multiplied by 10h (i.e. 16 decimal). If segment address is, for example, 1FA5h, and offset is 0100h, then absolute memory address will be

```
(1FA5h x 10h) + 0100h = 1FB50h.
```

6.05-02 DEBUG.EXE: the "A" (= Assemble) command

The "A" command switches DEBUG.EXE to translation of assembler language instructions (see chapter 7) into executable machine code. This code is not executed at once, but rather is written into memory, thus forming a succession of machine commands. When translation is finished, the formed succession can be executed or saved in a file. Here is an example of the "A" command specification in debugger's command line:

```
A 0100
```

Letter-name of the ASSEMBLE command is followed by address of a memory cell, where the first byte of the formed machine code should be written. In the shown example this start address is represented by an offset only, but it may be specified in any of allowable forms, enlisted in article 6.05-01. If segment is not specified explicitly, then it is defined by segment register CS:. Start address may be omitted at all, and then the start cell is pointed at by contents of registers CS:IP.

Having accepted the "A" command, DEBUG.EXE shows full address of a memory cell, where the first byte of translated machine code will be written, and displays a blinking cursor, thus inviting you to type an assembler instruction, which should be translated. At that moment DEBUG.EXE doesn't accept commands, described in part 6.05, but accepts only assembler instructions, described in chapter 7. The user confirms an end of each assembler instruction by ENTER keystroke. DEBUG.EXE translates the entered instruction, writes its machine code into memory cell(s), and shows a new line with a new full address, inviting to type the next assembler instruction.

In order to terminate translation of assembler instructions you have to ignore debugger's invitation and just press ENTER, while the line is left empty. Then DEBUG.EXE returns to normal interactive operation, shows its hyphen prompt and again becomes ready to accept commands, described in part 6.05. If DEBUG.EXE received assembler instructions from a file via input redirection, then an empty line should be left in

this file (7.01-04), and this empty line will force DEBUG.EXE to terminate translation of assembler instructions.

6.05-03 DEBUG.EXE: the "C" (= Compare) command

The "C" command shows non-coincident bytes from two successions of memory cells. Identical bytes are skipped. Command line with a call for the "C" command may look, for example, as follows:

```
C 113 L8 153
```

After the letter-name "C" the first and the third parameters are start addresses of those memory cell successions, where are the bytes to be compared. Start addresses may be specified in any of their allowable forms (6.05-01). In the shown example both start addresses are represented by offsets only, and then default segment for both sequences is defined by segment register DS:. The second parameter after letter-name "C" defines either offset of the last memory cell in the first succession or the length of byte successions to be compared (if the number is marked by preceding letter "L"). In the shown example L8 means length 8 bytes. All three parameters after the "C" command are required.

6.05-04 DEBUG.EXE: the "D" (= Dump) command

The "D" command displays hexadecimal contents of a group of memory cells, and at the same time in the right part of display screen shows the same contents, represented by characters and symbols of ASCII code. Here is an example of debugger's command line with a call for the "D" command:

```
D 19A9:02E0 L10
```

Just after the letter-name "D" there is a full address of the first memory cell in a group to be displayed. Address may be specified in any of allowable forms (6.05-01). If segment is not specified, then it will be defined by segment register DS:. The second parameter after the letter-name "D" defines either offset of last memory cell in a group or a length of that group (if the number is marked by preceding letter "L"). In the shown example L10 means length 10h (i.e. 16 decimal) bytes. If the second parameter is not specified, 80h bytes (i.e. 128 decimal) will be shown. If both parameters are omitted, then 80h bytes will be shown, starting from current offset, which is increased by length of the shown bytes group at each execution of the "D" command. This is why each next execution of the "D" command without parameters will show not the same, but the next group of bytes. Examples of dumps, displayed by the "D" command, are shown in fig. 8 – 12 (in chapter "Appendixes").

6.05-05 DEBUG.EXE: the "E" (= Enter) command

The "E" command writes new contents into one or more memory cells. Debugger's command line with a call for the "E" command may look like this:

```
E 0211
```

In the shown example the letter-name "E" is followed by address (offset) of the memory cell, where writing of new data should start. When segment is not specified, then it is defined by DS: segment register. Address may be specified in any of its allowable forms (6.05-01), but it can't be omitted. Having accepted this command, DEBUG.EXE shows full address of the requested memory cell and shows the data byte, present in this memory cell at that moment. The shown line ends with a dot, which should be regarded as a prompt to type a new data byte for this memory cell. Data in ASCII codes are not accepted by "E" command via keyboard, data should be presented in a form of two hexadecimal digits per byte. Then a SPACEBAR keystroke should follow. If a new data byte has been typed in, it will be written into memory cell, but if a new byte hasn't been typed, the former contents of this memory cell is preserved. In any case DEBUG.EXE will show contents of the next memory cell, inviting to change it in the same way. If instead of the SPACEBAR keystroke the "–" (minus, or hyphen) key will be pressed, then the user will be given an opportunity of another attempt to change contents of the previous memory cell. In order to terminate data input, you have to press ENTER instead of SPACEBAR.

The "E" command is executed otherwise, without waiting for data input via keyboard, if new input data are specified in the same command line after the memory cell address, for example:

```
E 03E0 'Data error' 0D 0A
```

New input data in command line may be represented either by two hexadecimal digits per byte, or as string(s) of ASCII characters, enclosed at both sides in quotes or in double quotes. Both forms of representation may interlace in one line in arbitrary order. Before being written into memory, ASCII characters are translated into hexadecimal form byte-by-byte, except the enclosing quotes, which are not translated and are not stored. If a byte is represented by a single hexadecimal digit, it will be interpreted as the lower half-byte, for example, the "A" digit will be interpreted as 0Ah. In any case data will be written into sequential memory cells, starting from the specified address, in the order of their placement in command line.

6.05-06 DEBUG.EXE: the "F" (= Fill) command

The "F" command fills a succession of memory cells with a repeated record of a single byte or of any given sequence of bytes. In debugger's command line a call for the "F" command may look, for example, like this:

```
F 03E0 L2E 0D 0A 'Reserved' 90 90
```

The first parameter after the letter-name "F" is start address of memory cell succession which is to be filled. Though start address may be specified in any of its allowable forms (6.05-01), in the shown example it is represented by an offset only, and then segment is defined by DS: segment register. The second parameter specifies either a length of memory cells succession (if the number is marked by preceding letter "L") or offset of the last cell in that succession. In the shown example length 2Eh bytes (46 decimal) is declared. The third and the following parameters represent that sequence of bytes, which should be used to fill memory cell succession. At least one byte of this sequence must be specified.

Data in command line may be represented either by two hexadecimal digits per byte or by a group(s) of ASCII characters, enclosed at both sides in quotes or in double quotes. Both forms of representation may interlace in one line in arbitrary order. Before being written into memory, ASCII characters are translated into hexadecimal form byte-by-byte, except the enclosing quotes, which are not translated and are not stored. If the data, presented in command line, are not enough to fill the whole succession of memory cells, the process is repeated: the next cells are filled with a copy of the same sequence of bytes. On the contrary, if succession of memory cells is too short, filling process terminates at the last cell without error message.

6.05-07 DEBUG.EXE: the "G" (= Go) command

The "G" command initiates execution of machine code commands, prepared in memory beforehand. Here is an example of debugger's command line with "G" command:

```
G =100 143
```

Optional parameters following letter-name "G" are addresses. The first address, preceded by equality sign, points at the first byte of that machine command, which is to start the execution procedure. Though start address may be specified in any of its allowable forms (6.05-01), in the shown example it is represented by offset only; hence, default segment will be defined by CS: segment register. If an address, marked by preceding equality sign, in not present in command line, then the start memory cell will be pointed at by contents of CS:IP registers.

Those addresses, which are not preceded by equality sign, are breakpoints. Up to 10 breakpoints may be specified in command line for different branches of the program under test. In those memory cells, which are pointed at by breakpoints, DEBUG.EXE replaces original machine code by code CCh of INT 03 interrupt (8.01-04).

Interrupt INT 03 handler returns control back to DEBUG.EXE, and then debugger restores original machine code in breakpoint memory cells, and also stores the state of registers and flags, thus preparing further continuation of debugging procedure. For successful termination of the described operations two conditions must be met. First, breakpoint address(es) must correspond to first byte(s) of machine command(s), since

otherwise the CCh code will be interpreted not as INT 03 call, but as a part of previous machine command. Second condition is that execution shouldn't be stopped in any other way, besides the prepared breakpoints, because otherwise DEBUG.EXE will not store states of registers and flags, will not restore the replaced contents of memory cells, and the program under test will become corrupted. You'll have nothing to do but reload it with "L" command (6.05-10).

If you have no intention to apply breakpoints, there are two methods to stop execution, initiated by the "G" command. The first is a call for INT 21\AH=4Ch handler (8.02-55), which stops execution, terminates debugger's session and returns control back to DOS. The second method is a call for INT 20 handler (8.02-01), which stops execution, but doesn't terminate debugger's session. The RET command (7.03-73) at original stack's position also induces a INT 20 call with the same consequences. DEBUG.EXE gets control back and begins to accept next command(s) either from keyboard or from redirection – according to the way it was launched. After a stop of execution via a INT 20 call the code under test most probably wouldn't be corrupted, but final states of registers and flags will be lost.

6.05-08 DEBUG.EXE: the "H" (= Hexadecimal) command

The "H" commands calculates and displays a sum and a difference of two hexadecimal numbers, each of up to four digits long. In debugger's command line it may look like this:

```
H 12BA 00AE
```

In the shown example two parameters after letter-name "H" are original numbers presented for calculation. Any of them or both may comprise less than four digits, and then zero(s) in senior positions will be implied. Presence of both parameters is required.

6.05-09 DEBUG.EXE: the "I" (= Input) command.

The "I" command reads a byte from the specified port and shows it on the screen. The shown byte can't be written elsewhere or stored. A single parameter in command line after the letter-name "I" is port address, for example:

```
I 03f8
```

Unlike ordinary memory addresses, port addresses have no relation to segments and segment registers. Port address is just a hexadecimal number of up to four digits long. Zeros in senior positions may be omitted. Addresses of several common ports are enlisted in appendix A.14-1.

6.05-10 DEBUG.EXE: the "L" (= Load) command

The "L" command reads a succession of codes from disk and loads it into memory, starting from specified address and on. Succession of codes to be read may be defined either by a name of a file or by a number of disk's sector. Here is an example of "L" command usage for loading a succession of codes from a file:

```
L 0100
```

Single parameter in the shown example is a memory cell address, where writing of code succession should start. Address may be specified in any of its allowable forms (6.05-01), but when segment is omitted, it is defined by CS: segment register. It is assumed, that the name of a file to be read is written already into PSP (A.07-1) by "N" command (6.05-12) or was initially transferred there from parameter of that command line, which has launched the debugger. Moreover, it is assumed, that contents of DS: segment register has not been changed since this name was written into PSP, and hence the "L" command can still refer to segment address in DS: for reading this name from PSP.

Address after the letter-name "L" may be omitted, and then succession of codes will be loaded from address CS:0100 an on, except loading from files with suffixes *.EXE and *.HEX. These files contain a header with supplementary loading specifications. Supplementary offset from the header of *.HEX files is added to that taken by default (0100h) or to that specified after the "L" command. For *.EXE files the offset specified after the "L" command is ignored, header of the *.EXE files is not loaded. In order to see "as they are" those files loaded without a header or not loaded at all, their suffix should be replaced (preferably with *.BIN). In any case length of the file is stored in CX register. If length exceeds 64 kb, then senior digits of length are written into BX register.

For loading a succession of codes from sectors of logical disk, command line must contain four required parameters, for example:

```
L 0100 2 0 1
```

The first parameter, just as in previous example, is address of that memory cell, where writing of code succession should start. The second parameter is interpreted as logical disk number: 0 – disk A:, 1 – disk B:, 2 – disk C:, and so on. The third parameter represents actual hexadecimal number of the first sector to be read, the fourth parameter – total hexadecimal number of sectors to be read. Up to 80h sectors can be read in one operation. In particular, the shown example specifies reading of a single sector number 0 (the boot-sector) from logical disk C: and writing its contents into memory starting from address CS:0100. Contrary to loading from files, loading from disk's sectors doesn't induce writing of loaded succession's length into registers. Physical disk's sectors beyond logical disk(s) are not accessible to "L" command.

6.05-11 DEBUG.EXE: the "M" (= Move) command

The "M" command copies a block of data from one place in memory into another. In debugger's command line a letter-name "M" must be followed by three required parameters, for example:

```
M 0100 L20 0180
```

In the shown example the first parameter specifies start address of the source data block, and the third parameter – a similar start address of the target memory space. Both these addresses may be specified in any of their allowable forms (6.05-01). If address is represented by offset only, then segment is defined by DS: segment register. The second parameter in command line represents either the length of the source data block (if the number is marked by preceding letter "L") or offset of its final byte. In the shown example second parameter L20 denotes length 20h bytes (32 decimal) of the source data block.

If the source data block and target memory space are not superimposed, then data in the source block remain unchanged. But when target memory space overlaps source data block, data in the overlapped part of this block will be overwritten without error message. In any case the order of copying is chosen so that the bytes to be overwritten are copied first.

6.05-12 DEBUG.EXE: the "N" (= Name) command

The "N" command declares a name of a file, which is to be loaded later by "L" command (6.05-10) or written later by "W" command (6.05-19). Letter-name "N" must be followed by a name of a file, for example:

```
N Trial.com /S /D
```

If suffix of the specified name exists, it can't be omitted. Name of a file may be preceded by a path and may be followed by a group of parameters (as "/S /D" in the shown example).

All data, declared by the "N" command, are written into PSP area (A.07-1) at address DS:0081 and on. Besides that, length of the written string is stored at DS:0080, file's name without suffix – at DS:005D, suffix – at DS:0065. As far as the mentioned addresses refer to DS: segment register, its contents must not be changed up to the moment when the written data will be requested by "L" or by "W" command. When the "N" command is executed without parameters, it overwrites data in DS:005C – DS:0080 memory cells and writes code 0Dh in memory cell DS:0081.

Note 1: initial settings in CS: and DS: registers are the same. Therefore loading of the program under test below default address CS:0100 may cause overwriting of PSP data (A.07-1), including the stored filename(s). Moreover, loaded code itself may be damaged later because of writing data into PSP by "N" command.

6.05-13 DEBUG.EXE: the "O" (= Output) command

The "O" command sends a data byte into specified port. Letter-name "L" in debugger's command line must be followed by two parameters, for example:

```
O 0378 00
```

The first parameter represents port address, the second parameter is the byte to be sent. Unlike ordinary memory addresses, port addresses have no relation to segments and segment registers. Port address is just a hexadecimal number of up to four digits long. Zeros in senior positions may be omitted. Addresses of several common ports are enlisted in appendix A.14-1.

Note 1: incautious usage of the "O" command may affect important PC's settings and cause serious malfunctioning.

6.05-14 DEBUG.EXE: the "P" (= Proceed) command

The "P" command initiates execution of a prescribed number of machine instructions, prepared or loaded into memory beforehand. If execution is initiated by the "P" command, then loops (7.03-55 – 7.03-57), subroutine calls (7.03-08), repetitions (7.02-03, 7.02-04) and interrupts (7.03-28) are not traced step-by-step, but rather are executed as though it were one machine instruction. This feature is the main difference between "P" and "T" (6.05-17) commands. Debugger's command line with "P" command may look like this:

```
P =0100 5
```

The first parameter after letter-name "P" is address of machine instruction, intended to start execution. This address may be specified in any of its allowable forms (6.05-01), but it must be preceded by equality sign. If address is represented by offset only, as in the shown example, then default segment is defined by CS: segment register. The second parameter specifies the number of machine instructions to be executed. Both parameters may be omitted, and then only one instruction will be executed, the one pointed at by CS:IP.

After execution of machine instructions the "P" command displays current states of registers and flags, and also presents result of unassembling the next, not executed yet machine instruction.

Note 1: the "P" command shouldn't be applied to machine instructions, which are read directly from fixed storage chips or from read-only memory (ROM). For such purposes the "T" command (6.05-17) should be used instead.

6.05-15 DEBUG.EXE: the "R" (= Register) command

If you type into command line a single letter R and press ENTER, then DEBUG.EXE will show you current states of CPU's main registers and flags. Typical initial states of registers and flags, set by DEBUG.EXE at the moment when it is launched, are shown below in fig.1.

```
R:\DOS\MS7>debug.exe
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=195B  ES=195B  SS=195B  CS=195B  IP=0100   NV UP EI PL NZ NA PO NC
195B:0100 D7            XLAT
-
```

Fig 1

Let's notice, that in all shown segment registers (DS, ES, SS, CS) the same hexadecimal number is written: it is segment address of memory space, allocated by DOS for debugging experiments. Other registers and flags acquire predetermined initial states. Bi-letteral designations of flag's initial states have the following meaning:

NV	No oVerflow;
UP	count UP, or incremental calculation of offsets;
EI	Enable Interrupts;
PL	Positive number;
NZ	Non-Zero value or inequality;
NA	No Auxiliary carry in the 4-th digit position;
PO	Parity Odd, odd sum of bits in least significant byte;
NC	No Carry in most significant digit position.

The last line of displayed message (fig.1) shows machine instruction code in memory cell, pointed at by CS:IP, together with unassembled representation of that code. Just this machine instruction will be executed, if DEBUG.EXE will be given "T" (6.05-17) or "P" (6.05-14) command with default parameters.

You may get an opportunity to change flag's states, if letter-name "R" in debugger's command line will be followed by parameter "F":

```
R F
```

DEBUG.EXE responds to this command by showing current flag's states together with a hyphen, which is an invitation to input bi-letteral designations of desired flag's states. Flag's states, opposite to those shown in fig.1, can be set by typing the following input:

OV	OVerflow after arithmetic operation;
DN	count DowN, or decremental calculation of offsets;
DI	Disable Interrupts;
NG	NeGative number;
ZR	ZeRo value or equality;
AC	Auxiliary Carry in the 4-th digit position;

PE Parity Even, even sum of bits in least significant byte;
CY CarrY in most significant digit position.

The order of flag's states input is indifferent. If new state for some flags is not specified, those flags preserve their former state.

In order to get an opportunity to change a state of a register, letter-name "R" in debugger's command line must be followed by a name of that register, for example:

```
R AX
```

The shown command opens the AX register for writing new value. The "R" command also enables to open registers BX, CX, DX, BP, SP, DI, SI, CS, DS, ES, SS, IP (and PC – it is another name for the same IP register). Having got the shown command, DEBUG.EXE displays former contents of that register and a colon, thus inviting you to input new value of up to four hexadecimal digits long. If you wouldn't type new data and just press ENTER, the former value in this register will be preserved. The "R" command doesn't provide separate access to senior byte and to least significant byte of 16-bit registers, for example, to AH and to AL inside the AX register. If you want to change the state of only one byte, you have to specify unchanged former value for the other byte of the same register.

Note 1: the 8 flags, mentioned in this article, are not all the flags in modern CPU's. For a more complete list of flags see appendix A.14-4.

6.05-16 DEBUG.EXE: the "S" (= Search) command

The "S" command performs a search for a specified sequence of bytes throughout a limited search region. Both sequence of bytes and search region must be defined by parameters, following the "S" command in debugger's command line, for example:

```
S 0100 L200 20 'st'
```

The first parameter after the letter-name "S" is start address of search region. Though start address may be specified in any of its allowable forms (6.05-01), in the shown example it is represented by an offset only, and then default segment is defined by DS: segment register. The second parameter specifies either a length of search region (if the number is marked by preceding letter "L") or offset of the last cell of that region. In the shown example length 200h bytes (512 decimal) is declared. The third and all the following parameters represent that sequence of bytes, which is to be searched for. At least one byte of this sequence must be specified.

Bytes of the sequence may be represented either by two hexadecimal digits per byte or by a group(s) of ASCII characters, enclosed at both sides in quotes or in double quotes. Both forms of representation may interlace in one line in arbitrary order. Before being taken into account, ASCII characters are translated into hexadecimal form byte-by-byte,

except the enclosing quotes, which are not included in the prepared sequence sample. In the course of search similar letters in upper case and in lower case will be considered different.

When the search is finished, DEBUG.EXE shows all addresses in the search region, where the sought sequence has been found. It's important to notice, that DEBUG.EXE doesn't analyze the mission of the found bytes sequence(s). Data strings and machine instructions may include just the same sequences of bytes. Solving of such uncertainties is a user's prerogative.

6.05-17 DEBUG.EXE: the "T" (= Trace) command

The "T" command initiates execution of a prescribed number of machine instructions, prepared or loaded into memory beforehand. The "T" command can trace execution of machine instructions inside loops, subroutine calls, interrupt handlers, etc. This feature is the main difference between "T" and "P" (6.05-14) commands. Here is an example of debugger's command line with "T" command:

```
T =0100 5
```

The first parameter after letter-name "T" is address of machine instruction, intended to start execution. This address may be specified in any of its allowable forms (6.05-01), but it must be preceded by equality sign. If address is represented only by offset, as in the shown example, then default segment is defined by CS: segment register. The second parameter specifies number of machine instructions to be executed. Both parameters may be omitted, and then only one instruction will be executed, the one pointed at by CS:IP.

After execution of machine instructions the "T" command displays current states of registers and flags, and also presents result of unassembling the next, not executed yet machine instruction.

Note 1: as far as the "T" command traces execution of machine instructions inside loops, subroutine calls and interrupt handlers, it may involve very long sequences of instructions, taking too much time to trace. In such cases the "P" command (6.05-14) should be preferred.

6.05-18 DEBUG.EXE: the "U" (= Unassemble) command

The "U" command displays assembler instructions obtained by translation of executable machine code from a group of memory cells. A debugger's command line with "U" command may look like this:

```
U 014B L10
```

The first parameter of the "U" command is an address of the first memory cell in selected group. Address may be specified in any of its allowable forms (6.05-01). If

segment is not specified, then it is defined by CS: segment register. The second parameter after the letter-name "U" defines either offset of the last memory cell in selected group or a length of that group (if the number is marked by preceding letter "L"). In the shown example L10 means length 10h (i.e. 16 decimal) bytes. If parameters are omitted, then code of 20h bytes will be translated, starting from current offset inside CS: segment. Current offset is increased by length of the translated bytes group at each execution of the "U" command, so that each next execution of the "U" command without parameters shows translation of code from not the same, but from the next group of memory cells.

It's important to notice, that the "U" command can't discriminate between executable code and other data, can't find the first byte of long machine instructions. When specified start address doesn't correspond to the first byte of instruction, or when unassembling is applied to data, then translation produces garbage. Two examples of unassembling are shown below in fig.2: the first corresponds to proper specification of start address (0181h), the other shows consequences of improper specification.

```
R:\DOS\MS7>debug.exe  fit.com
-u181 L15
197C:0181 8E06FC00      MOV     ES,[00FC]
197C:0185 BA5F03        MOV     DX,035F
197C:0188 8CC0          MOV     AX,ES
197C:018A 26            ES:
197C:018B 3B061600      CMP     AX,[0016]
197C:018F 7419          JZ      01AA
197C:0191 26            ES:
197C:0192 8E061600      MOV     ES,[0016]
-u180 L16
197C:0180 028E06FC      ADD     CL,[BP+FC06]
197C:0184 00BA5F03      ADD     [BP+SI+035F],BH
197C:0188 8CC0          MOV     AX,ES
197C:018A 26            ES:
197C:018B 3B061600      CMP     AX,[0016]
197C:018F 7419          JZ      01AA
197C:0191 26            ES:
197C:0192 8E061600      MOV     ES,[0016]
-
```

Fig. 2

In case of improper start address specification (0180h) translation of bytes at offsets 0180h and 0184h produces invalid results, but after that the "phase" of unassembling comes to a proper steady state. If translated machine instructions are correct and are known to DEBUG.EXE, then stochastic process of reaching the proper steady state usually takes up to 10h bytes and doesn't affect further translation.

Of course, commands should be unassembled just as they are apprehended by processor. However, the same machine code may be interpreted in different ways. One reason of differences is that machine codes of short jump instructions don't comprise jump target address, but rather contain a target offset relative to current contents of IP register. Debugger calculates target addresses by addition of these offsets to IP register contents. The latter depends on start offset, where the first byte has been loaded of that code, which is to be unassembled. Therefore unassembling of short jump instructions can't be

performed correctly, unless the code, which is to be unassembled, is loaded just as it must be loaded for execution, i.e. starting at CS:0000 address for drivers and at CS:0100 address for programs with suffixes *.COM and *.EXE.

Another reason of differences between processor's and debugger's interpretations of machine codes is the effect of processor's type and state. Interpretation depends on operand size byte in code segment descriptor (note 5 to A.12-2). Besides that, there are several machine codes, which are interpreted in a special manner by 64-bit processors. However, proper processor's type and state for a particular presented machine code isn't "known" to DEBUG.EXE. It is able to unassemble those machine codes only, which are designed for 16-bit execution. There is no sense in forcing DEBUG.EXE to unassemble other machine codes.

Invalid results of unassembling may be caused by those machine instructions, which are not "known" to debugger. Because of this reason Microsoft's version of DEBUG.EXE can't be recommended for unassembling modern programs. Another version of DEBUG.EXE, proposed in note 2 to part's 6.05 introduction article, should be preferred. Of course, recent versions of "respectable" unassemblers (IDA, SoftICE, etc.) are much more "clever", but can't give comparable freedom of access and even can't work under DOS. For those constrained to use Microsoft's version of DEBUG.EXE some "unknown" machine codes are given in the table below. Its first column presents first bytes of machine codes, which are shown by DEBUG.EXE as parameters of DB command (7.01-01) in a separate line. First byte together with second byte, given in the second column, are often enough to understand the type of operation. If necessary, names and references from the fourth column enable to find more information about particular instruction(s).

First byte	Second byte	Data bytes	Operation, instruction
0F	00	1–3	loading of task register (LTR)
0F	01	1–3	appeals to registers GDTR, IDTR, MSWR
0F	02	1–3	load access rights (LAR)
0F	03	1–3	load segment limit (LSL)
0F	05		loading of system registers (LOADALL)
0F	2(0–3)		MOV CR, MOV DR (note 1 to 7.03-58)
0F	4(0–F)	1–3	conditional copying (CMOV)
0F	8(0–F)	2	conditional jumps inside one segment
0F	9(0–F)		bit's conditional set/reset (SET)
0F	A(0,8)		PUSH FS, PUSH GS (7.03-69)
0F	A(1,9)		POP FS, POP GS (7.03-67)
0F	A2		identification of CPU (CPUID)
0F	A(4–7)	1–2	double word shift (SHLD, SHRD)
0F	B(2,4,5)	0–2	load segment registers (LSS, LFS, LGS)
60			copying of AX – DI into stack (PUSHA)

Continuation of table 6.05-18

61			popping stack data into AX – DI (POPA)
62		3	check of array's bounds (BOUND)
63		2	adjustment of access rights (ARPL)
6(4,5)			segment prefixes FS:, GS: (7.02-01)
66			operand size prefix (7.02-06)
67			address size prefix (7.02-07)
6(8,A)		1–2	push a number into stack (PUSH, 7.03-69)
6(9,B)		1–3	multiplication (IMUL, note 1 to 7.03--25)
6(C,D)			group input from port (INSB, INSW)
6(E,F)			group output into port (OUTSB, OUTSW)
8(C,E)	E(0–F)		MOV FS, MOV GS (note 2 to 7.03-58)
C(0,1)	E(0–7)	1	SHL bl,0f, SHL bx,0f (7.03-82)
C(0,1)	E(8–F)	1	SHR bl,0f, SHR bx,0f (7.03-83)

6.05-19 DEBUG.EXE: the "W" (= Write) command

The "W" command copies a code's succession from PC's memory onto a logical disk, either in a form of a file or just into disk's sectors. Here is an example of "W" command usage for writing a succession of codes into a new file:

```
W 0100
```

A single parameter in the shown example is a memory cell address, where reading of code succession should start. Address may be specified in any of its allowable forms (6.05-01), but when segment is omitted, default segment is defined by CS: segment register. If address is not specified, reading of code succession starts at CS:0100. Length of code succession, counted from start address, must be prepared beforehand: least significant two bytes of length – in CX register, most significant byte – in BX register.

It is assumed, that name of new file to be created is written already into PSP (A.07-1) by "N" command (6.05-12). The name may have any suffix except *.HEX and *.EXE, because DEBUG.EXE is unable to compile a header for such files. One more assumption is that contents of DS: segment register has not been changed since name of new file was written into PSP, and hence the "W" command can still refer to segment address in DS: for reading this name from PSP. If the prepared name is not preceded by a path, new file will be created in the current directory. If a file with the same name exists yet, it will be overwritten without error message.

For writing into sectors of a logical disk the letter-name "W" must be followed by four required parameters, for example:

```
W 0100 0 0 1
```

Here the first parameter presents start address of code succession, just as in previous example of writing into a file. The second parameter is interpreted as logical disk number:

0 – disk A:, 1 – disk B:, 2 – disk C:, and so on. The third parameter represents actual hexadecimal number of the first sector to be written, the fourth parameter – total hexadecimal number of sectors to be written. Up to 80h sectors may be written in one operation. Total number of sectors defines the length of written code succession, contents of CX and BX registers are ignored. In particular, the shown example specifies writing of a single sector number 0 (the boot-sector) onto logical disk A:. Physical disk's sectors beyond logical disk(s) are not accessible to the "W" command.

Note 1: when it is necessary to write a code succession from PSP region (A.07-1) into a file, you can't declare file's name by "N" command before a spare memory space for writing file's name is prepared. This may be done either by moving code succession out of PSP memory region or by changing DS: segment register contents so that file's name wouldn't overwrite code succession in PSP (example – in article 9.08).

6.05-20 DEBUG.EXE: the "XA" (= Allocate) command

The "XA" command appeals to EMM386.EXE memory manager (5.04-02, 8.03-59) with a request to allocate a specified memory space beyond conventional memory and to assign a hexadecimal reference number – a handle – to the allocated memory space. Requested operation can't be performed unless the EMM386.EXE memory manager is loaded yet and unless memory above 1088 kb is available. In debugger's command line the name "XA" must be followed by requested number of logical memory pages of 16 kb each, for example:

```
XA 1A
```

The shown requested memory space is 1Ah (26 decimal) logical pages, which will be enumerated from 00h to 19h. DEBUG.EXE responds to the request with a message, for example, "Handle created 0006". Hence, the allocated memory space can be referenced with hexadecimal number 0006h. But this is not enough for access to the allocated memory space, one more operation must be performed: a limited number of selected logical pages from the allocated memory space must be mapped onto physically addressable memory space by "XM" commands (6.05-22).

6.05-21 DEBUG.EXE: the "XD" (= Deallocate) command"

The "XD" command appeals to EMM386.EXE memory manager (5.04-02, 8.03-61) with a request to deallocate a particular memory space, which has been allocated yet by "XA" command (6.05-20). It is assumed, that EMM386.EXE memory manager is loaded and that memory beyond 1088 kb is available. In debugger's command line the name "XD" must be followed by a hexadecimal reference number (a handle), identifying the memory space, which is to be deallocated, for example:

```
XD 0006
```

DEBUG.EXE responds to the shown example: "Handle 0006 deallocated". Since that moment the 0006h handle becomes invalid, access to logical pages in deallocated memory space is lost, and this space is regarded as free.

6.05-22 DEBUG.EXE: the "XM" (= Map) command

The "XM" command appeals to EMM386.EXE memory manager (5.04-02, 8.03-60) with a request to map a 16 kb logical page in allocated memory space onto a "physical" page of the same size. Here the term "mapping" means adjustment of address translation mechanism in CPU so that ordinary 16-bit addresses within "physical" page address space are translated by CPU into 32-bit addresses of real memory cells in that part of address space, which corresponds to specified logical page. It is assumed, that PC has a 32-bit CPU, that amount of memory exceeds 1 Mb, that EMM386.EXE memory manager is loaded yet and that reference number (a handle) is assigned yet to the allocated memory space by "XA" command (6.05-20).

In debugger's command line the name "XM" must be followed by three required parameters, for example:

```
XM 0B 03 0006
```

The first parameter – 0Bh in the shown example – is a number of requested logical page in allocated memory space. The second parameter – 03h in the shown example – is a number of "physical" page. The third parameter is a reference number (a handle), which is assigned to that allocated memory space, where the requested logical page belongs.

Numbers of "physical" pages may be selected from 00h to 1Bh, but it should be taken into account, that "physical" pages from 04h and on occupy address space in conventional memory. This is why the most actively used are "physical" pages 00h – 03h, which by default correspond to segment addresses E000h, E400h, E800h and EC00h accordingly. But their placement in address space may be affected by PC's BIOS requirements and by EMM386.EXE (5.04-02) memory manager's settings. Therefore in each particular case placement of "physical" pages should be checked by the "XS" command (6.05-23) or by a call to INT 67\AX=5800h handler (8.03-70).

6.05-23 DEBUG.EXE: the "XS" (= Show) command

The "XS" command appeals to EMM386.EXE memory manager (5.04-02, 8.03-70) with a request to display data, related to memory usage and to placement of "physical" pages. It is assumed, that PC's amount of memory exceeds 1 Mb and that EMM386.EXE memory manager is loaded yet. Since "XS" command needs no parameters, command line with "XS" command looks like this:

```
XS
```

DEBUG.EXE responds to "XS" command with display of a table showing valid reference numbers (handles) and how many logical pages is associated with each handle. Then segment addresses are shown for each "physical" page. The last lines display statistics: total number of logical pages, number of free logical pages, maximum available number of handles and a number of handles, which are assigned yet. However, displayed table will be too long and wouldn't fit the screen, if it will comprise data about all 28 "physical" pages, stipulated by EMS 4.0 specification. In order to see the whole length of the table, you may either redirect output into a file (2.04-03), or set video mode 108h (A.10-1), or else limit actual number of "physical" pages by means of EMM386.EXE driver's settings (5.04-02).

6.06 DISKCOPY.COM – diskette copying utility

DISKCOPY.COM copies the whole contents of one floppy disk onto another, including volume label and serial number. Both floppies (diskettes) must be of the same type. Here is an example of command line for copying a diskette in a PC equipped with two floppy drives:

```
DISKCOPY.COM  A: B: /V
```

where:

A: – letter-name example of a drive with source diskette;
B: – letter-name example of a drive with target diskette;
/V – optional prescription to verify the copy.

Besides the shown options, DISKCOPY.COM can accept also

/1 – prescription to copy one side only of a diskette (for obsolete diskettes with a single writable side).
/M – prescription to perform multi-pass copying via memory, when one floppy drive only is available and a buffer-file on a HDD shouldn't be arranged.

If computer is equipped with only one floppy drive, then the same letter-name must be specified both for source and for target. In this case DISKCOPY forms a temporary buffer-file on a HDD in a directory, specified by environmental variable %TEMP%, then suggests to replace the source diskette with target diskette, and copies data from buffer-file onto target diskette. After that buffer-file is automatically deleted. But when the %TEMP% variable is not defined (or when /M option is specified), the source diskette is partially copied into available memory space, and therefore you'll have to exchange source and target diskettes several times in the same floppy drive.

DISKCOPY.COM is not designed to create a non-temporary file-image of a diskette. Besides that, DISKCOPY.COM can't cope reasonably with damaged sectors. Because of these drawbacks nowadays other copying programs are preferred. For example, the IMG.EXE utility can be recommended. Archive file IMG.ARJ, comprising the IMG.EXE utility, can be downloaded from server ftp://ftp.elf.stuba.sk/pub/pc/utildisk/ .

Note 1: DISKCOPY.COM and similar diskette copying utilities can't be applied to hard disk drives, to network drives, to CD/DVD-ROMs and also to virtual disks, arranged by utilities ASSIGN.COM, JOIN.EXE and SUBST.EXE.

6.07 DOSKEY.COM – command line manager

DOSKEY.COM is a resident supplement to command line functions of COMMAND.COM interpreter. DOSKEY.COM enables to edit command line text, to recall former commands, to create and execute macrocommands, comprising simple sequences of operations. Procedures, specified via macrocommands, become available after loading of their definitions by DOSKEY.COM into special memory buffer. Besides this buffer, resident module of DOSKEY.COM occupies about 4 kb.

DOSKEY.COM in MS-DOS7 has been changed relative to its former versions: it has been given an ability to load a list of macrocommand's definitions from a textual file and to increase sizes of both keyboard buffer and command line editing buffer.

DOSKEY.COM can be loaded from command line or from a line of a batch file (AUTOEXEC.BAT or other), either directly or with LH command (3.17), for example:

```
LH Doskey.com /bufsize:1024 /insert /file:C:\DOS\MS7\Macro.scr
```

where:

/bufsize:1024 – macrocommand buffer's size definition (minimum is 256 bytes, default is 512). This option should be used when DOSKEY.COM is loaded for the first time or is reloaded with /reinstall option (see further). Free part of the same buffer is used to store previous command lines (the history).

/insert – turns command line into character insertion mode, enabling to insert each next character between those typed earlier. Default is overstrike mode, when next characters replace the former ones.

/file:C:\DOS\MS7\Macro.scr – is a specification example for loading macrocommand's definitions from a textual file (an example of such file is given further). The path, preceding filename, may be omitted, if this file exists in the current directory. Filename and suffix of this file are arbitrary, but if suffix exists, it must be specified.

Other options, allowable for loading operations, are:

/echo:off – disable display of executed macrocommands on the screen.

/keysize:31 – increase size of keyboard's type-ahead buffer (default is 15 bytes).
/line:256 – set size of a buffer for editing command line (default is 128 bytes).
/reinstall – install a new resident module of DOSKEY.COM. This may be needed in order to change buffer's size or in order to revive functions, lost because of interference with other TSR utilities. Each reinstallation increases amount of occupied memory by about 4 kb (since former resident module of DOSKEY.COM can't be unloaded).

When DOSKEY.COM is loaded yet, it may be called again with /H option in order to show stored previous commands (the history):

```
Doskey /H
```

or else with /M option in order to show stored macrocommands:

```
Doskey /M
```

Besides this, DOSKEY.COM may be called again in order to load one more definition of a macrocommand into memory buffer from command line, for example:

```
Doskey count=C:\DOS\COM\Find.exe /v /c "" $1
```

Here the word "count" is interpreted as a name of new macrocommand. This name, being typed into command line close to command prompt, will initiate execution of command(s), specified within macrocommand's definition to the right of equality sign. The shown definition consists of one command, which counts lines in any textual file, represented here by dummy parameter $1. In the course of macrocommand's execution dummy parameter is replaced by actual filename, specified after macrocommand's name in the same command line.

Commands within macrocommand's definition may contain equality signs and substitutions of variable's values (such as %Temp%). When symbol "$" is encountered within macrocommand's definition, it is interpreted together with the following letter (or digit) in a special way as:

$G – output redirection sign ">" (right arrow);
$L – input redirection sign "<" (left arrow);
$B – intermediate redirection sign "|" (the "pipe");
$T – a separator between commands within one macrocommand;
$1-$9 – dummy parameters, equivalent to %1 – %9 in batch files;
$* – all words following macrocommand's name on command line;
$$ – one character "$" (dollar sign).

Textual file for loading macrocommand's definitions must contain one definition per line, which may look, for example, as follows:

```
count=C:\DOS\COM\Find.exe /v /c "" $1
newbat=echo @echo off$G %Temp%\New.bat $T Edit.com %Temp%\New.bat
```

You may delete any loaded macrocommand from buffer by specifying its name followed by nothing more than equality sign, for example:

```
Doskey count=
```

When DOSKEY.COM is loaded, it activates the following "hot" keys:

Left and right arrows	– shift cursor to the left and to the right
CTRL-left arrow	– shift cursor to the left by one word
CTRL-right arrow	– shift cursor to the right by one word
HOME	– shifts cursor to the beginning of command line
END	– shifts cursor to the end of command line
INS	– toggles overstrike and insert keyboard modes
ESC	– clears current command line
ALT+F10	– deletes all macrocommands from memory buffer
F7	– displays a list of previous commands (the history)
PageUp	– recalls the oldest line from the history list
PageDown	– recalls the newest line from the history list
UP and DOWN arrows	– shift command's selection along history list
F9	– selects a command from history list by number
Alt+F7	– clears list of previous commands (the history)
F8	– appends character(s) in command line with the rest part of command's name, if suitable name is present in history list.

Note 1: macrocommand wouldn't be executed, if its name in command line is preceded by at least one space.

Note 2: when macrocommand is synonymous to a command, it disables the latter (macrocommand will be executed instead).

Note 3: DOSKEY.COM is NOT compatible with TSR file managers (Norton Commander, Volcov Commander, etc). File managers usually are preferred.

6.08 DELTREE.EXE – directories eraser

DELTREE.EXE is an extremely dangerous utility, because it enables to delete directories with all their subdirectories and files, regardless to file's attributes. Only root directories of logical disks can't be deleted with DELTREE.EXE. An example of its usage may look like this:

```
Deltree /Y C:\TEMP\TDIR1
```

where:

/Y – an option, prescribing to delete without prompts and confirmations.

C:\TEMP\TDIR1 – an example specification of a directory to be deleted. There may be several such specifications in one line, but wildcards are not allowed.

6.09 EDIT.COM – editor utility

EDIT.COM is a popular double-window editor program mainly for textual files, but also with limited capabilities for editing binary files. EDIT.COM enables to keep open up to ten files simultaneously. Fig.3 shows two different files opened in separate windows of EDIT.COM editor.

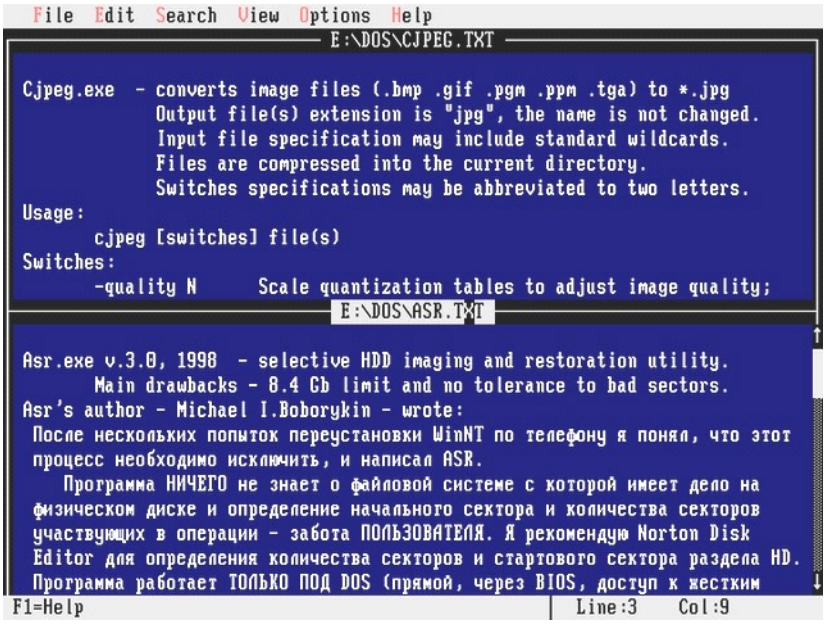

Fig. 3

The file(s) to be edited may be as large as PC's memory allows. EDIT.COM arranges a clipboard, enabling to send selected parts of text between opened files. Besides that, EDIT.COM is able to cooperate with mouse device drivers, thus making editing much easier and faster.

The EDIT.COM editor may be launched from command line without parameters, and then it will show an empty window. The user will have to open file(s) to be edited via menu "FILE". Otherwise the file(s) to be edited may be specified just in command line, for example:

```
Edit.com  C:\DOS\Addons.txt Part6.txt
```

Name of the first file to be edited is preceded by a path; without a path the EDIT.COM editor can open those file only, which are present in the current directory. Thus, the second file (Part6.txt), specified without path, must be present in the current directory. By default

both files will be opened for textual editing, but only the first opened file will be shown in displayed editor's window. The user is given an opportunity either to open second window by CTRL-F6 keystroke or to switch the initial window with F8 keystroke to editing of the second opened file.

Some presets and help texts for EDIT.COM are stored in files EDIT.INI and EDIT.HLP (implied to be present in the same directory), but EDIT.COM can cope without help and with default settings only. When EDIT.INI is absent, the user is still allowed to save optional presets via the OPTIONS menu, and this will induce EDIT.COM to regenerate EDIT.INI file anew.

EDIT.COM preserves functions of some "hot" keys, which were active in DOS. Among these are the BACKSPACE key and combinations of ALT keystroke with digit(s) in numerical keypad, enabling to enter characters by their ASCII code. But most other "hot" key functions are redefined. The ALT keystroke makes active literal "hot" keys for selecting items in upper bar menu, and also makes active digit keys in literal part of keyboard for selecting opened files by their number, if more than one file is opened. Arrow keys shift cursor anywhere inside editor's window. If the SHIFT key is kept pressed while cursor is shifted over a part of text, then this part of text becomes selected (highlighted) for being deleted or copied into clipboard. The same effect is achieved when cursor is dragged over a part of text by mouse, while its left button is kept pressed.

Besides the mentioned key functions, EDIT.COM activates the following "hot" keys and key combinations:

CTRL C	– copies selected part of text into clipboard (CTRL-INS key combination acts similarly);
CTRL F4	– closes active window (if both windows are opened);
CTRL F6	– splits window in two (if one window only was opened);
CTRL F8	– resizes both windows (if both windows are opened);
CTRL Home	– shifts cursor to the start of opened file;
CTRL End	– shifts cursor to the end of opened file;
CTRL P	– enables to insert non-literal symbols of ASCII code (further prompt is shown in the bottom line);
CTRL-PageDown	– shifts window by line's length to the right;
CTRL-PageUp	– shifts window by line's length to the left;
CTRL Q	– access to find/replace/delete functions (further prompt is shown in the bottom line);
CTRL-Space	– deletes selected part of text;
CTRL V	– pastes clipboard's contents into cursor's position;
CTRL W	– scrolls the text one line down (key combination CTRL-UpArrow acts similarly);
CTRL X	– cuts selected block of text into clipboard buffer;
CTRL Y	– deletes the line, pointed at by cursor, even if this line is

not selected;
CTRL Z – scrolls the text one line up (key combination CTRL-DownArrow acts similarly);
Delete – deletes one character at cursor's position, if no text is selected, otherwise deletes selected part of text;
F3 – finds next item, specified beforehand via CTRL-Q-F key combination;
F6 – toggles active and passive windows (if both are opened);
F8 – switches active window to display of the next file (if several files are opened);
PageUp – shifts the displayed text one page up;
PageDown – shifts the displayed text one page down.

Several "hot" key combinations are shown in upper bar menus. These menus and separate items in these menus can be opened not only with keystrokes, but also with mouse's left button as well.

As far as binary files have no line structure, line wrapping specification in command line will make their editing more convenient, for example:

```
Edit.com /78 Sc_sct.dat
```

where

/78 – an option for wrapping lines to 78 characters wide. Instead of 78 any number up to 255 may be specified, but 78 corresponds to actual editor's window width in 80x25 screen display mode.

EDIT.COM is able to accept from command line also the following options:

/B – turn display to monochrome;
/H – turn display to maximum allowable number of screen lines;
/R – open file(s) for reading only, exclude risk of change(s);
/S – force usage of short filenames;
/? – display on-line help.

6.10 EXPAND.EXE – unpacker for compressed files

EXPAND.EXE is unpacker for files compressed with Microsoft's COMPRESS.COM utility. Each such compressed file contains only one original file and inherits its name, except that the last character in suffix is replaced with underscore, for example *.TX_ instead of *.TXT. Releases of previous MS-DOS versions and several other software packets comprise such compressed files together with EXPAND.EXE unpacker utility. In particular, EXPAND.EXE is present in SFX archive DOS62SP.EXE, which can be downloaded from Microsoft's FTP-server ftp://ftp.microsoft.com/softlib/mslfiles/ .

Name of EXPAND.EXE unpacker utility must be followed in command line by name of that compressed file, which is to be unpacked, and then by a name, which should be appointed to the restored original file, for example:

```
Expand.exe E:\DOS\MSDOS622\Country.tx_ C:\DOS\MS6\Country.txt
```

Both paths in the shown example are optional. The path preceding a name for the restored file is interpreted as a prescription to write the result of unpacking into that directory. If paths are omitted, then compressed file will be searched for inside current directory only, and the result will be placed just there, but in this particular case it is permissible, because compressed file and original file have different names.

Wildcards (2.01-03) in filenames are not allowed by EXPAND.EXE, but there may be several names of compressed files in one command line. In this case the last specified name must be a directory name (without final backslash), where unpacked files should be written. Name of the current directory with this mission can't be omitted, but may be represented with a dot (2.02-03).

Source files, compressed with old versions of COMPRESS.COM, comprise no information about original names of compressed files. Such files will be unpacked properly, but their original names will not be restored, if explicit new name specification is not given in command line. Therefore sharing of a common directory as both source and target should be avoided because of possible name conflicts.

Note 1: compressed files in Windows-2000/XP releases also are marked with underscore at the end of suffix, but these files are compressed by other algorithm. Their unpacking in MS-DOS7 can be performed by EXTRACT.EXE (6.11).

6.11 EXTRACT.EXE – unpacker for *.CAB files

The EXTRACT.EXE utility unpacks files, compressed by MAKECAB.EXE packer. Microsoft uses this algorithm for compression of separate files in Windows-2000/XP releases, and also for compiling large multi-volume archives with *.CAB suffix in releases of Windows-95/98/ME. EXTRACT.EXE enables to extract separate files from large *.CAB archives and to display their contents. Here is an example of command line for extracting a list of contents from a multi-volume *.CAB archive:

```
Extract.exe /A /D E:\Win95\OSR2.PE\Win95_21.cab > C:\Temp\List.txt
```

where:

E:\Win95\OSR2.PE\ – example of a path to one *.CAB archive from Windows-95 release. Other volumes of the same archive are implied to exist in the same directory. If path is omitted, archive is searched for in current directory only.

/D – an option, prescribing to display a table of contents without unpacking of archive's volume(s).

/A – an option, prescribing to apply the same operation to all following volumes of the same archive. In this particular case it induces processing of CAB-archive volumes from 22-nd to 26-th.

As far as *.CAB archives contain a large number of files, it's difficult to perceive their long tables of contents from a scrolling screen. This is why in the shown example output is redirected into file List.txt, which may be analyzed more conveniently with a viewer (6.19) or with an editor program (6.09).

Here is one more example of EXTRACT.EXE usage for extracting of a single file from a multi-volume archive, when it is not known beforehand, which particular volume contains the required file:

```
Extract.exe /A /Y /L C:\Windows\System Win95_02.cab Msvcrt40.dll
```

where:

/Y – an option, prescribing to overwrite any synonymous file in the target directory without prompt.

/L – an option, prescribing to interpret the following item (C:\Windows\System) as a target path for unpacking. When /L parameter is omitted, default target for unpacking is the current directory.

Msvcrt40.dll – a name example for the file to be unpacked (there may be several filenames specified for unpacking).

In the shown example the first specified filename (Win95_02.cab) is interpreted as a name of that archive volume, where a search for the required file should start. All following filenames are interpreted as names of those files, which should be extracted and decompressed. Execution of the shown command line initiates a search procedure through volumes from 02 to 26 for the specified file. In fact it will be found in the 13-th volume and will be unpacked into C:\WINDOWS\SYSTEM directory, overwriting there a corrupted sample of the same file. Such procedures are much more fast, than total unpacking of a software release.

EXTRACT.EXE doesn't allow wildcards (2.01-03) in filenames, but accepts one more optional parameter /E , which forces to unpack all contents of the specified CAB-file. In this case a particular name(s) for the files to be unpacked shouldn't be specified. The /E parameter is expedient for unpacking non-cabinet compressed files (those marked with underscore at the end of suffix) from Windows-2000/XP releases. But total unpacking of large *.CAB files in DOS takes too much time. A combination of /A and /E parameters deserves special caution, because it may initiate a process, which will take hours of time and a lot of disk's space.

Note 1: some software vendors supply *.CAB archives, which are produced by other compression algorithm. Besides that, several mutually incompatible versions of EXTRACT.EXE are known. The most suitable unpacker for any given software release is that one, which is supplied within this release.

6.12 FC.EXE – files comparison utility

The FC.EXE utility enables to compare two files, either binary or textual. Binary comparison is expedient for finding changed bytes in nearly identical binary files: FC.EXE displays byte number for each pair of mismatched bytes and these bytes themselves, taken from each of the compared files.

Here is an example of command line for performing binary comparison:

```
Fc.exe /B Trial2.com D:\Temp\Trial1.com
```

where:

/B – an option, prescribing binary comparison.
Trial2.com – an example of the first file to compare; since its name isn't preceded by a path, it is implied to exist in the current directory.
D:\Temp\Trial1.com – a name example of the second file with preceding path.

Binary comparison doesn't imply searching for match in mutually shifted successions of bytes.

Textual comparison is based on line structure of textual files. FC.EXE compares textual files line-by-line and displays mismatched lines side-by-side. When order of correspondence between line successions becomes disrupted, FC.EXE can restore it by searching for a next group of matching lines. Both comparison and search conditions are specified by options in command line, for example:

```
Fc.exe /A /C /L /LB9 /N /T /W /1 A:\Config.sys C:\Config.sys
```

where:

/A – display 2 lines only (the first and the last) from each mismatched group of lines.
/C – disregard letter's case (for ASCII codes up to 127).
/L – prescription for ASCII textual comparison.
/LB9 – limit example (9 consecutive mismatched lines) for a region of match search; default is 100 lines.
/N – prescription to display line numbers.
/T – prescription to avoid expansion of tabulation codes (09h) into spaces.
/W – prescription to ignore empty space (tabulation codes and spaces).
/1 – a number example of consecutive lines that must match after a group of mismatched lines.

Note 1: binary comparison is the default for files having suffixes BIN, COM, EXE, LIB, OBJ, SYS. Other files are compared as textual by default.

Note 2: in special applications one of the files to compare (or both) may be replaced by other sources (2.01-01), for example, by virtual device NUL or by CON device – the console. The latter enables to perform comparison with keyboard input (1.04).

6.13 FDISK.EXE – partitioning tool for HDDs

Disks, known by their letter-names, are logical disks. Contrary to those, physical hard disk storage devices have no letter-names. Writable storage space in physical storage devices may be divided into several partitions, and each partition may represent a separate logical disk. The FDISK.EXE utility is a proprietary MS-DOS7's tool for arranging partition structures in physical HDDs (Hard Disk Drives).

Being launched without parameters, FDISK.EXE asks the user whether large disk support should be provided or not. Rejection of the offer means that arranged partitions within range 512 – 2048 Mb will be of FAT-16 type, otherwise FAT-32 type will be preferred. FAT system for partitions outside the mentioned range is assigned by default: FAT-32 for partitions larger than 2048 Mb, FAT-16 for partitions between 512 and 16 Mb, FAT-12 – for 16 Mb and smaller volumes.

Those apt to trust theirselves may specify command line parameters, enabling to get rid of undue questions and restrictions:

```
Fdisk.exe /fprmt /actok
```

The /fprmt parameter cancels query about large disks support and gives all rights on file system choice to the user. The /actok parameter enables to arrange an active partition in any physical hard disk drive (otherwise an active partition may be arranged in the first physical HDD only). Being launched in the shown way, FDISK.EXE presents a menu to select an operation: to display current partition structure, to delete or to create a partition or to make it active. The user is allowed to compose partition structure, but the latter is not written to disk at once. FDISK.EXE gives a chance to correct it. If a partition ought to be bootable, don't forget about making it active. Then you may exit FDISK.EXE.

If partition structure has been changed, FDISK.EXE writes it to disk and exits into reboot, because created partitions must be registered by BIOS. PC's BIOS system appoints letter-names to registered partitions. Since that moment all registered partitions become "visible" in DOS as logical disks, but new logical disks are not yet formatted and therefore are inaccessible. New logical disks become accessible after formatting by FORMAT.COM utility (6.15).

Besides interactive arrangement of partition structures, FDISK.EXE may be used as ordinary console utility with the following options:

`Fdisk.exe /?`	– display a short help.
`Fdisk.exe /status`	– show available physical HDDs and logical disk's allocation.
`Fdisk.exe /mbr`	– write or rewrite MBR (master boot record) on the first HDD, leaving its partition table intact (note 1). Confirming message is not displayed.
`Fdisk.exe /cmbr 2`	– acts just as /mbr, but enables to specify the addressed HDD by physical number: 1, 2, 3 and on are allowed, if these HDDs exist. Confirming message is not displayed.

Regardless to purpose of FDISK.EXE usage, its command line may be complemented with /X option, which forces to avoid extended support functions for access to disk(s). The /X option should be remembered, when ordinary usage attempts fail: the HDD either is not recognized, or is found inaccessible, or the PC gets hanged with "Stack overflow" message. Such outcomes may take place because of incorrect configuration, hardware faults, virus MBR infection. Not necessarily, but in some such cases the /X option may help.

If an identical partition structure should be formed on several new HDDs, then automatic (non-interactive) arrangement of partitions should be preferred. Automatic writing of a partition structure onto a HDD can be initialized by the following example of command line:

```
Fdisk.exe 1 /PRI:2000 /EXT:8000 /LOG:8000 /Q
```

where:

1	– addressed physical HDD number (1, 2, ...).
/PRI:2000	– arrange a primary partition, for example, 2000 Mb. If FAT-16 file system should be formed in this partition, then /PRIO: parameter (instead of /PRI:) should be specified.
/EXT:8000	– arrange extended partition, for example, 8000 Mb.
/LOG:8000	– arrange logical disk, for example, 8000 Mb, inside the extended partition. For sizes not larger than 2000 Mb parameter /LOGO: (instead of /LOG:) means that FAT-16 file system should be formed.
/Q	– "quiet" option, i.e. run without screen messages.

Having finished its job successfully, FDISK.EXE initializes reboot.

Note 1: overwriting of MBR enables to get rid of MBR bugs, including those inflicted by viruses. However, some BIOS extensions and boot managers use non-standard types of MBR, which can't be restored by FDISK.EXE. In such circumstances other ways of MBR restoration should be preferred (example – in 9.02-03).

Note 2: Microsoft's FDISK.EXE doesn't support non-sequential placement of partitions and therefore gives no opportunity to bypass worn-out regions of physical disk's surface. FDISK.EXE can't arrange correctly those partitions, which cross the

8.4 Gb boundary in physical disk's space. FDISK.EXE often can't cope properly with partitions, arranged by other operating systems: either can't delete such partitions, or may arrange new partitions, overlapping the former ones.

Note 3: most significant drawbacks of original Microsoft's FDISK.EXE have been corrected in its new unofficial version (dated 2006), which can be downloaded from site http://radified.com/Files/FDISK.EXE . Besides that, completely new synonymous utility has been compiled by Brian E. Reifsnyder. Version 1.30 (dated 2003) of this utility, packed into archive fdisk130.zip, can be downloaded from server ftp://ftp.uni-koeln.de/pc/msdos/diskutils/

Note 4: any change of existing partitions structure, performed by FDISK.EXE, causes total data loss within affected partition(s). Partition(s) rearrangement without data loss may be performed by more powerful tools, for example, by PowerQuest's Partition Magic utility (bought now by Symantec Co.).

Note 5: FDISK.EXE can't arrange partitions in HDDs, accessed via a network or controlled by drivers. Because of the latter reason FDISK.EXE often can't cope with external storage devices having SCSI or USB interface. In such cases other utilities may suit, for example, BTFDISK.EXE (from Buslogic) or TFDISK.EXE (from TEKRAM). Archive DC390FBW.ZIP, comprising TFDISK.EXE, can be found in internet site http://www.neuron.alt.ru/drivers/Driver/Controllers/ , in its subdirectory TEKRAM. A SFX archive DOSASPI.EXE, comprising BTFDISK.EXE, is present in subdirectory BUSLOGIC of the same site.

6.14 FIND.EXE – word(s) searching filter

The FIND.EXE utility acts as a filter for textual data: it receives lines of text from a file or via redirection, selects lines with or without a certain combination of characters, and sends selected lines into standard output channel (STDOUT) for being displayed on the screen by default. Here is an example of word(s) filter usage for searching lines with a specified string of characters:

```
Find /N /I " INT 13 " C:\DOS\SRV\Drives.txt
```

where:

/N – an option, prescribing to accompany the displayed lines with their line numbers.

/I – an option, prescribing to ignore the case of letters in specified string of characters.

" INT 13 " – an example of characters string to be searched for, enclosed in double quotes. Note spaces between words and each of adjacent double quotes – this guarantees finding of whole words, but not parts of other words.

C:\DOS\SRV\Drives.txt – an example of a file to be analyzed (with preceding path). If path is not specified, the file is implied to exist in the current

directory. There may be several filenames specified in one line one-by-one or by means of wildcards (2.01-03).

The shown example of command line will display on the screen all lines of the analyzed file, containing words INT 13, and a short message reminding which file has been analyzed. Lines will be shown preceded by their numbers: this will help you to find them later while editing the same file. If you expect that the displayed listing may happen to be too long, you may send output into a file (2.04-03) or to the viewer MORE.COM (6.19).

Having finished the search, FIND.EXE returns errorlevel 0, if it has found the specified string of characters at least once, or errorlevel 1, if this string hasn't been found in the analyzed text. Errorlevel may be used to determine the outcome of search (see articles 3.15-03 and 6.03).

The next less typical example shows FIND.EXE counting lines in a textual file:

```
Find.exe  /V /C "" < Draft.txt
```

where:

/V – an option, prescribing to display all lines NOT containing the specified string.

/C – an option, prescribing to display only the count of lines meeting the specified condition.

"" – an empty specification of a string to be searched for.

< Draft.txt – an example of a file to be analyzed, sent via input redirection (2.04-02). Since the filename is not preceded by a path, this file is implied to exist in the current directory. Having got input via redirection, FIND.EXE doesn't add its reminding message to the displayed result.

Void string to be searched for is regarded by FIND.EXE as a special non-existing object. Therefore FIND.EXE will simply count all the lines, including the empty ones. After counting the lines FIND.EXE always returns errorlevel 0.

The third usage example presents lines of a batch file. Suppose that the target path is given as a value of environmental variable %P%, and it is not known, whether it has a final backslash or not. Since DOS is not indifferent to absence of a the final backslash, you need to append it, if it isn't specified yet. This may be done in the following way:

```
echo %P%\\ | Find.exe "\\\" > nul
if errorlevel 1 set P=%P%\
```

In the first line FIND.EXE analyses redirected output of the ECHO command and tries to find specified combination of symbols, which will be present there if the given path contains final backslash. STDOUT output of FIND.EXE presents no interest and is sent to NUL (into nowhere). The result becomes known via the errorlevel value, left by FIND.EXE. It is analyzed in the second line of the presented example. Errorlevel 1 means

that FIND.EXE has found no final backslash in the given path, and then the SET command (3.26) will append the missing backslash.

The last (fourth) usage example is also a part of a batch file. Suppose that you have activated a procedure, which prepares a list of files (%Temp%\Files.lst) to be packed into an archive, and want to prevent twice-fold packing, when this list contains nothing more except archives of the same kind (RAR, for example). This can be achieved by the following command lines:

```
Find /C /I /V ".rar" < %Temp%\Files.lst | Find ": 0" > nul
if not errorlevel 1 echo Chosen file(s) - already RAR-archive(s)
if not errorlevel 1 goto NO_PACK
```

In the first line the leftmost call for FIND.EXE reads filenames from the list line-by-line via input redirection, and counts only those lines, which don't contain filenames with ".RAR" suffix. The count result is redirected via STDOUT to the second (rightmost) call for FIND.EXE, which is "waiting" for zero count. When the count result really is zero (i.e. there are no other files except RAR archives), the rightmost call for FIND.EXE leaves errorlevel 0. This errorlevel value is checked in second and third lines of the example. The second line displays an error message, and the third line performs a jump to label "NO_PACK", thus enabling to bypass packing operation.

6.15 FORMAT.COM – formatting tool for disks

The FORMAT.COM utility arranges logical disks on storage media, including diskettes and partitions of hard disks. Formatting includes testing of each sector's readability, writing sector's headers, creation of boot sector, of file allocation table (FAT) and of root directory. Clusters found to have non-readable sectors are marked in FAT as "BAD" and thus get out of use. Disks of 16 Mb and smaller are formatted with file system FAT-12. Choice of file system (FAT-16 or FAT-32) for partitions of hard disks is made with respect to FAT type identifier (A.13-6), assigned to particular partition beforehand by FDISK.EXE (6.13). Cluster size is calculated automatically as allowable minimum for given disk's size and FAT type.

Formatting of floppy disks (diskettes) implies low-level recalibration of tracks, so that actual capacity of a diskette may be altered. An example of command line for formatting a diskette may look like this:

```
Format.com A: /V:Archives /Q /F:1.44 /B
```

where:

A: – a required letter-name specification of the disk to be formatted. Valid letter-names are appointed to disks at boot time, when disks are registered by BIOS. Those letter-names, which are appointed or

changed later by software means, are regarded by FORMAT.COM as invalid.

/V:Archives – an example of optional specification for volume's label: any word(s) of up to 11 characters long altogether. If this option is omitted, FORMAT.COM offers to specify volume's label at the final stage of formatting procedure. The user may reject the offer and just press ENTER; then this logical disk is given the NO_NAME label.

/Q – "quick format" – an option prescribing to skip most time consuming operations: sectors test and headers writing. Formatting with "Q" option is used to delete all contents of those disks, which have been formatted yet and have sector headers as well as sectors themselves in good condition. If there is a doubt about it, quick formatting can't be recommended.

/F:1.44 – optional size specification (for floppy disks only). Allowable sizes are: 160, 180, 320, 360, 720 (kb) and 1.2, 1.44, 2.88 (Mb). Instead of the /F option you may use other forms of size specification, which are shown in note 1 below. When neither of allowable size options is specified, FORMAT.COM is able to determine suitable size based on BIOS' CMOS settings and on drive's sensor signals.

/B – an option prescribing to reserve space for system files in order to make the disk bootable later. Instead of /B you may specify /S, which means the same plus copying of DOS's system files into the root directory of the formatted disk. System files (COMMAND.COM, IO.SYS) are implied to exist in the root directory of PC's main bootable disk. Contrary to other system files, MSDOS.SYS is not copied: FORMAT.COM creates an empty sample of this file anew.

Modern hard disk drives have a fixed track structure. It can't be altered by formatting procedure. Therefore disk's size, number of sectors and other similar format parameters shouldn't be specified for formatting HDD's partitions, for example:

```
Format.com D: /s /c /z:64
```

where:

/c – an option prescribing to test clusters that are currently marked "BAD". Sometimes this enables to revive disks and diskettes, which have got a marked "BAD" sector in their first track. But testing of numerous bad clusters may considerably increase the time spent for formatting.

/z:64 – an option forcing to arrange 32-kb clusters, each containing 64 sectors. It may be needed, if you intend to expand this partition later. The /z:1 option enables to arrange FAT-16 file system on small disks (from 4 to 16 Mb). But most often the "/z:" specification is omitted, because the default cluster size is considered the best.

Note 1: instead of total size of a floppy disk you may specify number of tracks and number of sectors in each track, for example:

/T:80 /N:18.

One more alternative is to use the following parameters (instead of the /F option):

/1 – format an obsolete one-sided diskette;
/4 – format a 360 kb floppy in 1.2 Mb 5.25-inch drive;
/8 – format 8 sectors per track.

Note 2 original Microsoft's FORMAT.COM (dated 1998) can't cope properly with formatting partitions beyond 64 Gb. Corrected version of FORMAT.COM (dated 2006), packed into SFX archive FDSKFRMT.EXE, can be downloaded from http://www.mdgx.com/files/FDSKFRMT.EXE .

6.16 LABEL.EXE – volume's label changing tool

Volume's label is a word or a group of words of up to 11 characters long, used as a storage media identifier. Volume's label is written into boot sector (A.03-4) and, besides that, into a hidden entry of the root directory. Most often volume's label is appointed during formatting procedure, but it may be assigned or changed later by the LABEL.EXE utility, for example:

```
Label.exe  R:RAMDRIVE
```

where:

R: – a specification example for a disk which should have its volume label altered; when disk is not specified, the current disk is taken by default.
RAMDRIVE – an example of volume label to be assigned to specified disk.

When volume label is omitted, LABEL.EXE shows current label of specified disk and offers to type a new one. New label specification may be accepted via redirection too (2.04-02, 2.04-05). If user rejects the offer, he is prompted to delete (or to preserve) the current label.

Having finished its job, the LABEL.EXE utility leaves behind the following errorlevel values:

082 – label writing attempt has failed.
015 – invalid letter-name, no corresponding drive.
002 – drive has no removable storage media inside.
000 – disk is either accessible or non-formatted. Anyway, writing of volume's label hasn't been attempted.

The returned errorlevel value may be determined just as it is shown in articles 3.15-03 and 6.03. As far as the returned errorlevel value is informative, the LABEL.EXE utility is sometimes used to test disk's accessibility (example - in article 9.03-02).

6.17 MEM.EXE – memory allocation explorer

The MEM.EXE utility is called in order to display information about amount of available memory, about its allocation, about loaded resident modules, for example:

```
Mem.exe /A /C /P
```

where options are:

- /A – append summary with data about free space in HMA area.
- /C – show a summary table of memory usage. The first column of this table enlists names of resident modules, the rest columns – details of memory allocation. Instead of the /C option there may be specified:
  - /D – show status of resident modules and drivers;
  - /F – show nothing but amount of free memory;
  - /M:HIMEM – show details about a particular resident module (HIMEM in this example). Its name, specified after the /M option, must be separated by a colon. Names of modules should be taken from the first column of summary table.
- /P – make a pause after each screenful of information.

6.18 MODE.COM – peripherals interface tuner

The MODE.COM utility implements a number of interface configuration functions for ports, for video card and for character generator. A summary table of current subordinate equipment settings is displayed by MODE.COM utility, when it is called with /STATUS option:

```
Mode.com  /STATUS
```

6.18-01 MODE.COM: ports tuning operations

Interface tuning operations for serial ports (COM1, COM2, COM3, COM4) are performed by MODE.COM as it is shown in the following example:

```
Mode.com  COM1:11,E,8,2,B
```

where:

- 11 – set the 110 baud rate; values 11, 15, 30, 60, 12, 24, 48, 96, 19 are allowed and correspond to baud-rates 110, 150, 300, 600, 1200, 2400, 4800, 9600 and 19200 accordingly.
- E – implement even parity check; instead you may specify:
  - O – implement odd parity check;
  - N – no parity check.
- 8 – set 8 information bits per codeword (8 or 7 allowed).
- 2 – set number of stop bits (1 or 2 allowed).

B – means normal reaction on port's errors. Instead of B there may be specified:
- E – return ERROR message if connected device is busy;
- N – don't repeat appeals to port after an error;
- P – appeal to port repeatedly until CTRL-BREAK keystroke;
- R – force switching to a new job even if the connected device didn't finish its previous job.

Interface tuning operations for parallel ports (LPT1, LPT2, LPT3) imply that the connected device is a printer. This is why for parallel ports the MODE.COM utility accepts other parameters, for example:

```
Mode.com  LPT1:80,6,B
```

where:

80 – number of characters per line (80 or 132 allowed);
6 – number of lines per inch (6 or 8 allowed),
B – means normal reaction on printer errors. Instead of B there may be specified just those options as for serial ports in the previous example.

The MODE.COM utility can readdress messages from parallel port (LPT1 – LPT3) to any serial port (COM1 - COM4), thus enabling to use a printer with serial interface:

```
Mode LPT1:=COM2
```

In order to cancel readdressing the MODE.COM utility should be called once more without further specifications after the same parallel port's name:

```
Mode.com LPT1:
```

6.18-02 MODE.COM: textual video modes switching

The MODE.COM utility allows two forms of commands for switching textual video modes (A.10-1). Here is an example of the first (most obsolete) form of command:

```
Mode.com  co80
```

where:

co80 – means color video mode with 80 characters per line. Instead of co80 there may be specified:
- bw40 – monochrome video mode, 40 characters per line;
- bw80 – monochrome video mode, 80 characters per line;
- co40 – color video mode, 40 characters per line.

The second form of command for switching textual video modes looks like this:

```
Mode.com  80,25
```

where:

80 – number of characters in a line (40 or 80 allowed);
25 – number of lines per screen height (25, 43 or 50 allowed).

Note 1: switching of textual video modes is accompanied with loading fonts anew. If current font has been installed by some other program (except MODE.COM), then change of a video mode by MODE.COM may cause a return from national font to default american font codepage 437.

6.18-03 MODE.COM: codepage selection

Commands for codepage preparation and selection operations, performed by MODE.COM utility, usually are written into the AUTOEXEC.BAT file (9.01-02). All these operations need the DISPLAY.SYS driver (5.02-02) to be loaded yet. The first of these operations is preparation of several codepages for further activation:

```
Mode.com CON CP PREP=((437,850,866) EGA3.CPI)
```

where:

CON – is a specification of the device, addressed by this operation. Instead of CON (console, i.e. keyboard and display) ports LPT1, LPT2 and PRN (printer) may be specified.
CP PREP – abridged name of operation "Code Page PREPare".
(437,850,866) – numbers of codepages (A.02-2) to be prepared. Only these codepages will be available for switching by CHCP command (3.04).
EGA3.CPI – name example of a file, containing all those codepages, which should be prepared.

Then one of prepared codepages should be made active by means of selection operation:

```
Mode.com CON CP SEL=866
```

where:

CP SEL – abridged name of operation "Code Page SELect".
866 – number example of the codepage to be activated.

The codepage, activated for the CON (console) device, defines visual presentation of letters and symbols on the screen. In order to determine which codepage is currently active for a particular output device, you should call for the MODE.COM utility with device specification (CON, PRN, LPT1 or LPT2), with CP (Code Page) operation name and with the /STATUS option:

```
Mode.com CON CP /STATUS
```

Any loaded codepage may become corrupted because of malfunction or because of undue actions of other software. In such cases the MODE.COM utility enables to load the active codepage once more with command:

```
Mode.com CON CP REF
```

where:

CP REF – abridged name of operation "Code Page REFresh".

6.19 MORE.COM – page-by-page viewer

When long messages are sent to display as a whole, text is scrolled over the screen too fast to be perceived. In order to make long messages readable, the MORE.COM viewer sends them to display in a page-by-page manner. Each time the screen is full with text, the MORE.COM viewer suspends output, giving an opportunity to read the text. Each user's keystroke initiates display of the next text page.

The MORE.COM viewer intercepts the messages, sent into standard output channel (STDOUT), either by means of input redirection (2.04-02):

```
More.com < D:\MyDocs\Part2.txt
```

or by means of intermediate redirection (2.04-05):

```
Type D:\MyDocs\Part2.txt | More.com
```

where:

D:\MyDocs\Part2.txt – is a name example of a file to be displayed, preceded by full path. When the path is omitted, the file is implied to exist in the current directory.

Type D:\MyDocs\Part2.txt – is an example of a command (3.30), sending its output via STDOUT output channel. Any other command, sending its output to STDOUT, may be specified instead.

Note 1: messages sent to display via BIOS' interrupts or via STDERR output channel can't be intercepted by MORE.COM.

Note 2: when the MORE.COM viewer accepts a message by means of intermediate redirection (2.04-05), this message is written into a temporary file, and access to a writable media is necessary. If current disk is non-writable or write-protected, and if a path to a writable media is not defined in environmental variable %TEMP%, then the MORE.COM viewer can't perform this mission. Therefore interception by means of input redirection should be preferred.

6.20 MOVE.EXE – copying and renaming utility

The MOVE.EXE utility is used to rename directories and to move files from one directory into another.

When source and target directories belong to different logical disks, moving is equivalent to copying, followed by deleting the copied file in the source directory. But when both source and target directories belong to the same logical disk, moving can be performed much faster by rearranging directory entries: reference record to file's first cluster is moved from one directory table into another without touching the file itself. The MOVE.EXE utility is able to decide, which way of moving should be implemented in each particular case.

Moving of file(s) may be accompanied by renaming. Renaming of a directory is also performed by correcting an entry in parent directory table.

Here is an example of MOVE.EXE usage for moving files from one directory into another:

```
Move.exe /Y  D:\MyDocs\Part*.txt C:\Dos\Chap*.txt
```

where:

/Y – an option prescribing to overwrite synonymous file(s) in the target directory without prompt. When this option is not present in command line, MOVE.EXE tries to find it in the value of COPYCMD environmental variable. If COPYCMD value contains this option, you may overcome its action by specifying an opposite /-Y option in command line.

D:\MyDocs\Part*.txt – specification example for source file(s) with preceding path. Path may be omitted, if the file(s) reside in current directory. There may be several source specifications in one command line.

C:\Dos\Chap*.txt – target specification example, including a mask for renaming the files to be moved. The last of all such specifications in command line is interpreted as target directory specification. If the last name in this specification is not a name of an existing object, then this new name is assigned to the moved file. But when the last name in target specification is a name of an existing directory, file(s) are moved into that directory without renaming.

In order to rename a directory, the last name in source specification must be not a mask and not a filename, but a name of an existing directory. In this case the target specification must consist of a new name only without path, even if the directory to be renamed is not a subdirectory inside the current directory.

6.21 SCANDISK.EXE – disk(s) repairing tool

SCANDISK.EXE is a program for examination and repairing diskettes and hard disk drives, formatted with FAT12, FAT16 or FAT32 file systems. SCANDISK.EXE checks boot sector, reveals lost clusters, corrects crosslinks in FAT table, examines writeability of disk's surface in each cluster. Non-readable clusters are marked in FAT as BAD and therefore get out of further use. The course of testing and location of bad clusters are displayed in a chart, shown below in fig.4.

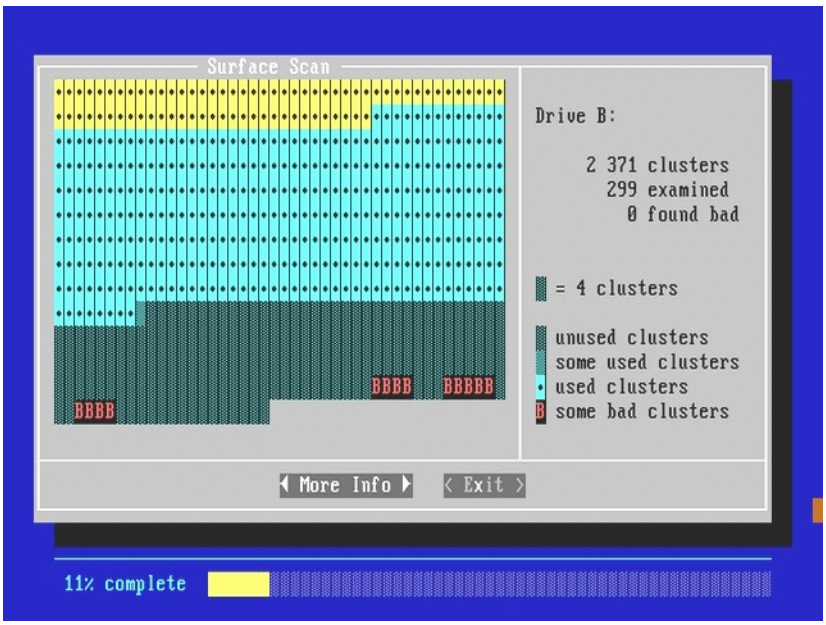

Fig. 4

SCANDISK.EXE tries to rewrite information from bad clusters into good ones so as to restore integrity of each file. Being launched regularly, SCANDISK's maintenance procedures enable to keep your computer in good health for a long time.

Details of test's structure are defined by parameters, stored in a separate file SCANDISK.INI, which must be present in the same directory with main SCANDISK.EXE file. SCANDISK.INI is a textual file. It contains comprehensive commentaries and can be edited by the user. If you have to check and repair a drive according to all parameters in SCANDISK.INI, you have to launch SCANDISK.EXE with /CUSTOM option in command line. When /CUSTOM option is not specified, only those parameters in [ENVIRONMENT] section of SCANDISK.INI file will be taken into account.

Parameters in command line have priority over those specified in SCANDISK.INI file. SCANDISK.EXE is able to fix automatically most part of the errors it finds, but for that it has to be launched with parameters in command line, for example:

```
Scandisk.exe C: /AUTOFIX /NOSAVE /NOSUMMARY /SURFACE
```

where:

C: – specification example of a disk to be examined. If you want all accessible disks to be processed, you should specify /ALL instead of particular letter-name. Compressed disks should be specified with their volume-name, for example: C:\DRVSPACE.000

/AUTOFIX – fix errors without prompt. On the contrary, when automatic repairing isn't desirable, you may specify the /CHECKONLY option instead of /AUTOFIX.

/NOSAVE – delete lost clusters rather than save them as files. This option can be used only if the /AUTOFIX option is specified too. By default lost clusters are transformed into files *.CHK in the root directory of the same disk.

/NOSUMMARY – don't stop at summary screens. This option can be used together with either /CHECKONLY or /AUTOFIX options.

/SURFACE – perform a surface scan after other checks (surface scan shouldn't be applied to compressed disks).

When /AUTOFIX option is not specified, then before making any changes SCANDISK.EXE offers you to write disk's current state onto UNDO-diskettes. This gives an opportunity to restore current disk's state later, if fixes will happen to inflict something wrong. While making a decision about UNDO diskettes, it should be taken into account, that amount of data in current state record may be enormous. Large partitions of modern HDDs may require hundreds of UNDO-diskettes and a lot of time. This is why SCANDISK's offer about UNDO-diskettes most probably should be rejected.

If nevertheless original disk's state was saved in UNDO-diskette(s), and restoration of this state has been found necessary, then restoration procedure must be performed at once, before any new files are written onto this disk:

```
Scandisk.exe /UNDO B:
```

where:

/UNDO – parameter initiating restoration procedure.

B: – specification example of a drive, which will be used to read the UNDO-diskette(s).

One more operation, performed by SCANDISK.EXE, is exploration of whether a particular file is fragmented or not:

```
Scandisk.exe /FRAGMENT C:\WINDOWS\SYSTEM\Kernel32.dll
```

where:

/FRAGMENT – an option, inducing exploration of file's fragmentation.

C:\WINDOWS\SYSTEM\KERNEL32.DLL – a specification example for a file to be explored. If path is omitted, this file will be searched for in current directory only.

Regardless to performed operation, SCANDISK.EXE accepts from command line one more option – the /MONO option, prescribing to configure its output for a monochrome display (by default the output messages are displayed in color).

Note 1: SCANDISK's procedures can't be applied to network drives, to CD/DVD-ROMs, to RAM-disks and to virtual disks, created by utilities ASSIGN.COM, JOIN.EXE and SUBST.EXE. SCANDISK.EXE also can't cope with those damages of disk's data, which make this disk inaccessible. However, inaccessible disks with damaged boot sector sometimes may be restored with NDD.EXE or with DISKEDIT.EXE utilities from Symantec Co.

Note 2: SCANDISK.EXE needs direct access to the processed disk and therefore can't be used inside a multi-tasking environment. A call for SCANDISK.EXE inside the DOS "window", arranged by WINDOWS OS, in fact invokes another program – the SCANDSKW.EXE from the \WINDOWS directory.

Note 3: SCANDISK.EXE with /AUTOFIX option shouldn't be applied to disks, suspected of being infected by virus. This may cause irreparable data loss. Such disks should be processed by an antivirus scanning program first (for example, by DRWEB.EXE).

Note 4: by default SCANDISK.EXE regards as an error each long filename, appointed by WINDOWS OS. Attempts to fix all such errors may have awful consequences. SCANDISK.EXE wouldn't do harm to long names, if section [ENVIRONMENT] of SCANDISK.INI file contains lines:

```
LfnCheck = Off
SpaceCheck = Off
```

Improved version of SCANDISK.EXE from WINDOWS-ME software release is more tolerant to long names and can be used under MS-DOS7.

Note 5: data rewriting from bad clusters into good ones can't be performed, unless there are some free clusters on the same disk. Of course, you can remove any file in order to get some free space, but there are utilities (version 2.50 of PKZIP.EXE archiver, for example), which fill diskette(s) with a single file leaving no free space at all. Because of one corrupted sector such file can't be neither read nor repaired, since there are no spare sectors. Filling of the whole media with one file should be avoided.

6.22 SORT.EXE – lines sorting filter

The SORT.EXE utility accepts textual lines from a file or via redirection and sends the same lines in another – sorted – order via STDOUT channel to the screen (the default), or to another file or device. Sorting is defined by the order of characters in ASCII code table.

The following example shows SORT.EXE usage for displaying a changed order of textual lines on the screen:

```
Sort.exe  /R /+12 D:\MyDocs\Unsort.txt
```

where:

/R – an option, prescribing to reverse sorting order, i.e. from Z to A and then from 9 to 0.

/+12 – specification example for sorting lines according to characters in column 12. If this option is omitted, characters in column 1 will be taken by default.

D:\MyDocs\Unsort.txt – specification example for the file to be processed (with preceding path). File without preceding path will be searched for in current directory only.

Second usage example shows input from a source file via redirection and output redirected into another (target) file:

```
Sort.exe /+9 < D:\MyDocs\Unsort.txt > D:\MyDocs\Sort.txt
```

where:

D:\MyDocs\Sort.txt – specification example for the target file. If this file exists, it will be overwritten without prompt. Specification of the same file for both the source and the target is not allowed (data will be lost!).

The last example shows input via intermediate redirection from another command and output of sorted lines to a printer:

```
Type D:\MyDocs\Unsort.txt | Sort /+3 > PRN
```

Note 1: redirection to a printer shouldn't be attempted unless you are sure that your printer is properly connected to LPT1 port, is switched on and is able to cooperate with MS-DOS.

Note 2: both output redirection (>) and redirection via a "pipe" (|) need access to a writable media and can't be performed, if this condition is not met (2.04-03 – 2.04-05).

6.23 SUBST.EXE – arrangement of virtual disks

The SUBST.EXE utility arranges virtual disks, used as an equivalent replacement for real paths. Originally SUBST.EXE has been developed as a means of access to subordinate directories for old programs (from DOS versions earlier than 3.0). Early DOS versions didn't "know" hierarchical directory structures, all files were stored in a single root directory. The SUBST.EXE utility enables to address a file in any subdirectory as though it were in the root directory of a disk. Nowadays the SUBST.EXE utility is used

rarely in order to squeeze more data into a limited length of command line and of environmental variable %PATH%.

Here is an example of arranging a virtual disk with SUBST.EXE:

```
Subst.exe  V: D:\DATA386\For_K\MyDocs
```

where:

V: – example of a letter-name for the virtual disk to be arranged. Letter-name must be free (not belonging to any real drive) and must be chosen within the limit, set by LASTDRIVE command (4.17, 4,18) in CONFIG.SYS file.

D:\DATA386\For_K\MyDocs – example of real disk and path to be assigned to virtual disk. If path is not specified, current disk and path are implied.

In order to delete virtual disk, command line with a call for SUBST.EXE utility may look like this:

```
Subst V: /D
```

where:

V: – letter-name example for the virtual disk to be deleted.

/D – an option, prescribing to delete virtual disk.

Being executed without parameters, the SUBST.EXE utility displays a list of currently active virtual disks.

Note 1: virtual disks, arranged by SUBST.EXE, can't be subjected to actions of the following utilities: Assign.com, Backup.exe, Chkdsk.exe, Defrag.exe, Diskcomp.com, Diskcopy.com, Fdisk.exe, Format.com, Label.exe, Mirror.exe, Recover.exe, Restore.exe, Scandisk.exe, Sys.com, Undelete.exe, Unformat.com.

Note 2: the paths assigned to virtual disks remain accessible in an ordinary way.

Note 3: virtual disks can be accepted by WINDOWS OS, but can't be arranged, when WINDOWS OS is loaded yet. If necessary, virtual disk(s) should be arranged before loading WINDOWS, preferably by SUBST.EXE utility launched from a line of AUTOEXEC.BAT file.

6.24 SYS.COM – utility making disks bootable

At a certain stage of PC's booting process control is transferred to executable code, read from boot sector of the boot disk. In its turn, this executable code transfers control to a loader file, specified inside the same boot sector. In order to make a disk bootable with MS-DOS7, an appropriate executable code and the name IO.SYS of MS-DOS7 loader file must be written into disk's boot sector beforehand. Besides that, the IO.SYS loader itself together with command interpreter COMMAND.COM and MSDOS.SYS file (5.01-01)

must be present in the root directory of bootable disk. These boot ability conditions are prepared by SYS.COM utility.

Command line for making a disk bootable may look like this:

```
Sys.com  C:\  A:
```

where:

C:\ – example of a path to the source directory with those system files (IO.SYS and COMMAND.COM), which should be copied to new disk. If source path is omitted, system files will be searched for in the root directory of that disk, which has been used to boot the PC.

A: – letter-name example of that disk, which is to be made bootable. It must be recognized by PC's BIOS and must be formatted yet under the same version of DOS.

Note 1: SYS.COM utility can't be applied to network disks, to CD/DVD-ROMs and to virtual disks, created by RAM-disk drivers or by utilities ASSIGN.COM, JOIN.EXE and SUBST.EXE.

Note 2: the operations, performed by SYS.COM utility, are necessary, but are not sufficient for making bootable a HDD's partition. This partition must be marked active in disk's MBR by FDISK.EXE (6.13) or by some other similar program. Executable code of boot sector will be read for execution just from that single active partition.

Note 3: contrary to IO.SYS and_COMMAND.COM system files, the MSDOS.SYS file (5.01-01) is not copied. SYS.COM creates it empty anew. Loading from a new disk with default settings is considered more safe, because former settings may be unsuitable in other circumstances.

Note 4: in the 8-th version of MS-DOS the SYS.COM utility has been changed: it always uses the default path to the source directory, and can't accept this path from command line.

6.25 VC.COM – file manager

6.25-01 Main properties of Volcov Commander shell

File manager Volcov Commander (VC), written by Vsevolod V. Volcov (Kiev, Ukraine), resembles the known Norton Commander file manager, but VC is more compact and more flexible for adaptation to user's demands. Though VC is an unfinished project and can't be considered quite good, nevertheless it is the best for making reparatory work comfortable under MS-DOS7. Version 4.99.07 of VC, packed into archive vc499.zip, can be downloaded from internet site http://www.fdd5-25.net/shells.php . Just this version, dated 1998, is described below. The latest alfa-version 4.99.08 of VC Shell is dated 2000.

This latest version, packed into archive vc49908a.zip, can be downloaded from site http://vvv.kiev.ua/download/ .

VC.COM is not a file manager itself, it is just a start file. The whole VC release includes the main overlay VC.OVL, a number of code translation tables *.TBL, and the following configuration files:

VC.INI	– non-textual file with general configuration settings.
VC.MNU	– configuration of a menu, invoked by F2 keystroke.
VC.EXT	– suffix-defined services, invoked by ENTER keystroke.
VCARCH.EXT	– specifications of archive services.
VCEDIT.EXT	– suffix-defined services, invoked by F4 (Edit) keystroke.
VCVIEW.EXT	– suffix-defined services, invoked by F3 (View) keystroke.

All VC's configuration files, except VC.INI, are ordinary textual files, which can be edited with EDIT.COM (6.09) or with any other editor program for non-formatted textual files. Examples of several VC's configuration files are shown in articles 6.25-02 – 6.25-04.

All files of the VC release must be stored in one directory, and the path to this directory should be assigned as a value to environmental variable VC. Therefore file AUTOEXEC.BAT should include a line, for example:

```
set VC=C:\DOS\VC4
```

VC Shell may be launched from command line as an ordinary program, but it is usually launched automatically from the last line in AUTOEXEC.BAT file, for example:

```
C:\DOS\VC4\Vc.com /TSR /no2E /noswap /nozoom
```

where options are:

/TSR – activate TSR supervisor, which monitors all loaded later TSR modules and unloads them at VC Shell shutdown. This may help to release memory and sometimes enables to avoid necessity of PC's reboot. On the contrary, when TSR supervisor is not desirable, the /noTSR parameter should be specified instead.

/no2E – exclude execution of programs via undocumented INT 2E interrupt (8.02-89), use legal INT 21\AX=4B00h (8.02-53) for this purpose. INT 2E is shorter and faster, but is not reentrant and gives no access to local variables of the caller. If nevertheless INT 2E is preferred, you may specify /2E parameter instead.

/noswap – prescription to avoid swapping of VC's data from memory to a file on a HDD. VC's data swapping is based on CHS access, which is not compatible with large modern HDDs. On the contrary, in obsolete PCs data swapping may be desirable, and then you may specify the opposite /swap option.

/nozoom – don't zoom message windows (zooming is adopted as the default).

Other allowable options for VC.COM are:

/BW – set black and white palette for monochrome displays. An alternative to /BW is /LCD – set special palette for LCD displays. Default is 16-color palette, typical for textual video mode 03h (A.10-1).

/std – load VC into conventional memory. By default VC Shell prefers other choices, if there are any. When VC loads itself into conventional memory, you may specify also

/big – load the whole VC's resident module;

/small – load a part of module's code, requiring periodic replenishment from disk.

/XMS – prescription to load VC into XMS memory, if XMS memory manager HIMEM.SYS (5.04-01) is installed yet. Alternatives to /XMS option are:

/noXMS – avoid XMS memory usage;

/EMS – prefer EMS memory, if EMM386.EXE memory manager (5.04-02) is installed yet;

/noEMS – avoid EMS memory usage.

/ini:Alter.ini– use another configuration file with arbitrary name (Alter.ini in this example) instead of VC.INI. You may rename current VC.INI file, change VC configuration and press Shift-F9 "Save Setup": VC will be forced to create a new VC.INI file. Thus you may store a number of alternative settings in different *.INI files.

/? – display a short help.

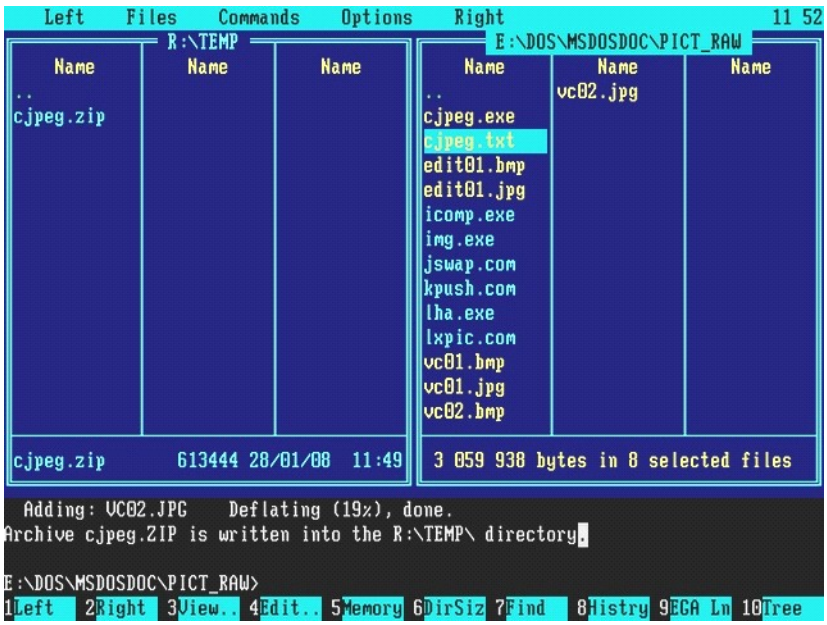

Fig 5

Behind all parameters in command line VC.COM allows to specify a command, which should be executed just after launching the VC Shell itself. This opportunity may be expedient, for example, for refreshing a record of interrupt table contents, saved beforehand by ESCAPE.COM utility.

When VC Shell is running, its appearance on the screen may be different, depending on initial settings, stored in VC.INI file. Usually directory's contents are displayed in one or two panels, as it is shown in fig.5.

Main functions of VC Shell are accessible via "hot keys" and via mouse's buttons clicks as well. Those "hot keys", which are kept active by VC Shell, are enlisted below in decreasing importance order.

Ctrl B – toggles on/off the bottom keybar
F9 – activates the upper functional bar
Ctrl F1 – toggles on/off the left panel
Ctrl F2 – toggles on/off the right panel
Tab – toggles active left panel/active right panel
Alt F1 – enables to choose disk in the left panel
Alt F2 – enables to choose disk in the right panel
Ctrl O – toggles on/off both panels
Ctrl Q – toggles on/off Quick View window in non-active panel
Ctrl L – toggles disk info on/off in non-active panel
Ctrl [– types the path from left panel into command line
Ctrl] – types the path from right panel into command line
Ctrl i – types name of a selected file into command line
Ctrl P – toggles non-active panel on/off
Ctrl U – swaps panels
Arrows – shift menu item or selected file in active panel
Home – shifts selection to the start of current directory
End – shifts selection to the end of current directory
F2 – displays a user-defined menu (6.25-02)
F3 – invokes viewer to read a selected file
F4 – invokes text editor, if it is defined by the user
F5 – copies selected file or group into opposite panel
Ctrl F5 – as F5, but copies only one file, ignores group list
F6 – moves or renames selected file or prepared group
Ctrl F6 – as F6, but moves only one file, ignores group list
F7 – enables to create a directory
F8 – deletes a selected file or prepared file group
Ctrl F8 – as F8, but deletes only one file, ignores group list
Ins – adds highlighted item to a list for group operation
Shift F9 – writes current VC's settings into VC.INI file
Ctrl A – shows file attributes and enables to alter them

Ctrl C – invokes directories comparison, marking endemic files
Ctrl E – moves last line from history table back into command line
Ctrl F – enables to define suffix(es) for selection with a mask
Ctrl N – toggles appearance of file's suffixes in panels
Ctrl R – causes re-reading of current directory contents
Ctrl F4 – enables to alter volume label for the active disk
Ctrl F9 – changes display video mode (Alt-F9 changes it too)
Ctrl \ – causes a jump to the root of disk in the active panel
Alt-letter – search for a file in the current directory by name
Alt F6 – size of selected directory (in panel's "full" mode)
Alt F8 – displays previous command lines (the history)
Alt F10 – shows active panel tree, enables jump to any directory
Alt F11 – enables to make panels shorter with ARROW UP key
Alt F12 – enables to make panels longer with ARROW DOWN key
F10 – exits current session and unloads Volcov Commander.

Functions of some "hot keys" are duplicated. Ctrl-J and Ctrl-Enter act just as Ctrl-i: write name of the selected file into command line. Ctrl-Z acts as ALT F10 – shows active panel tree. "0" in the numerical keypad acts just as INS: adds the selected item to a list of items, being prepared for performing a group operation. Items, included in the list, are highlighted with colour. Several keys in the numerical keypad are charged with auxiliary functions for preparing group operations:

Ctrl + – includes all files of the current directory in group list
\+ – includes in group list the files, selected with a mask
Ctrl – – discards current selection of files
– – excludes files selected with a mask (wildcards allowed)
Ctrl * – inverts files selection in current directory
* – acts as "+", if list of files in current directory is not formed, or as "–", if group list of files exists yet

When a list of highlighted items is prepared, then operations F5 (copy), F6 (move), F8 (delete) involve all the items in the list. List of highlighted items exists as a temporary file VCLIST.000 in the directory, defined by environmental variable %Temp%. Inside VC configuration files (*.MNU, *.EXT) the prepared list of highlighted items is accessible via special macro commands !~@ (for the active panel) and %~@ (for the passive panel). Thus the user is given an opportunity to define its own group operations (examples – in article 6.25-02).

If both VC's panels are switched OFF, VC shell provides more opportunities to navigate along the command line with LEFT ARROW and RIGHT ARROW keys, to insert characters into command line, and enables to access its history list with UP and DOWN arrow keys. When NUMLOCK key is switched OFF, arrow keys in numerical keypad act in the same way.

Development of the described VC's version (4.99.07) has not been completed. Some functions, defined by the user via configuration files, may go wrong, when there are alien *.EXT or *.MNU files in the current directory, and also when an archive file is left opened in the non-active panel. Several "hot keys" are claimed active, but in fact are not or act in a wrong way:

Alt F4 – (edit) – has no effect, use F4 instead
Alt F5 – (memory usage) – has no effect
Alt F7 – (file's search) – implemented since version 4.99.08
Ctrl M – (restore group) – has no effect
Ctrl N – (long file names) – in rare cases may cause hanging
Ctrl Z – (show tree) – may go non-stop until ESC keystroke
F1 – (VC's internal help) – isn't available
F6 – (move and rename) – being applied to a directory, disables mouse, compels to reload mouse driver.

6.25-02 Volcov Commander's menu file VC.MNU.

VC's wide capabilities of adaptation are achieved due to a variety of macrocomands, which are available inside its menu files (*.MNU) and inside its extension files (*.EXT). Certain groups of syntax symbols and digits, being encountered in VC's configuration files, are interpreted by VC Shell as a call for a macrocommand, and then are replaced by the result, returned by the same macrocommand. This result may be a name, a path, a line, etc. Here is a list of syntax symbol groups and corresponding macrocommands, provided by VC file manager version 4.99.07:

![prompt] – invitation to input word(s) from keyboard. The other word(s), specified inside the square brackets, constitute a prompt, displayed before this invitation. Up to 10 independent similar invitations are allowed, implicitly enumerated from 0-th to 9-th.
!0...!9 – mark for a place to insert a copy of the word(s), which were typed in by the user in response to an invitation, enumerated by the specified number, from 0-th to 9-th.
!: – mark to insert letter-name of the disk, opened in active panel.
%: – mark to insert letter-name of the disk, opened in passive panel.
!~\ – mark to insert the path, opened in VC's active panel. Returned path ends with a backslash.
%~\ – mark to insert the path, opened in VC's passive panel. Returned path ends with a backslash.
!~ – mark to insert short name (without suffix) of the file, selected with left mouse's button in VC's active panel.
%~ – mark to insert short name (without suffix) of the file, selected with left mouse's button in VC's passive panel.

!~.!	– mark to insert short name (with suffix) of the file, selected with left mouse's button in VC's active panel.
%~.%	– mark to insert short name (with suffix) of the file, selected with left mouse's button in VC's passive panel.
!~@	– mark to insert name of temporary file, containing a list of filenames, highlighted with right mouse's button in VC's active panel.
%~@	– mark to insert name of temporary file, containing a list of filenames, highlighted by right mouse's button in VC's passive panel.
!!	– mark to be replaced with a single exclamation sign.
%%	– mark to be replaced with a single percent sign.

Combinations of symbols, including the tilde sign (~), invoke those macrocommands, which truncate "long" filenames to 8 characters, and "long" suffixes – to 3 characters. In all such combinations of symbols the tilde sign (~) can be omitted, and then the same macrocommands, being executed in "DOS window" of WINDOWS-95\98\ME OS, return long filenames "as they are", without truncation.

The user is given an opportunity to specify groups of symbols, invoking any macrocommand, in any command line inside menu file (*.MNU) and inside extension files (*.EXT). All lines in these files can be typed with editor program for non-formatted textual files (6.09). Each item of menu is represented in VC.MNU file by a header line and by at least one command line. VC file manager considers command lines are those non-empty lines, which begin with at least one space (ASCII code 20h). On the contrary, the header lines must not begin with a space. In the header lines the first group of characters, followed by a colon, is interpreted as specification of the "hot key" to invoke function of this menu item. The rest group of words to the right of colon is interpreted as a name, representing this item of menu, when menu is displayed on the screen.

When the user has chosen a menu item for execution, the VC file manager replaces all groups of symbols, used to call any macrocommand, by those data returned by corresponding macrocommand. This is done at once in all command lines, following the header line of the chosen menu item. Then these command lines are sent one-by-one for execution to COMMAND.COM interpreter, which is called anew for each command line. Because of this reason jumps between command lines are not allowed, and the values of environmental variables can't be transferred from one command line to the following command lines within the same menu item. Of course, all conditions for proper execution of batch files should be met (see introduction article to chapter 9).

An example of VC.MNU file is shown below. Mentioned there files Edit.com (6.09), Fc.exe (6.12), Scandisk.exe (6.21) are programs taken from Windows-95/98 release. Files Arc.bat and Turn_off.com are those described in articles 9.03-01 and 9.05-02 of this book. All these files must be present inside directories, specified in the value of the %PATH%

environmental variable (2.02-02). As far as VC file manager has no internal help, file HELP.TXT, mentioned in the first item of the menu, is implied to contain a text written by you yourself. A path to HELP.TXT file is required to exist on one disk with COMMAND.COM interpreter.

```
F1:  Help
     @for %%Z in (%%comspec%%) do %%Z\
     @Edit.com /R \DOS\VC4\Help.txt
     @!:\
F4:  Search for a file or filemask in the current subtree
     @rem ![Type filename or mask to search for & press ENTER:]
     @echo.
     @if not !0"==" Dir !0 /P /A:-D /S /B
F5:  Compare a file with synonymous file in non-active panel
     @if !:!~\==%:%~\ echo Other panel must show other folder!!
     @if not !:!~\==%:%~\ Fc /L /N !~.! %:%~\!~.! > %%Temp%%\Y.t
     @if not !:!~\==%:%~\ Edit.com %%Temp%%\Y.t
     @if not !:!~\==%:%~\ del %%Temp%%\Y.t
F6:  Pack file(s) marked in the active panel into a ZIP-archive
     @Arc.bat ZIP !: !~\ !~ !~@ %: %~\
F7:  Pack file(s) marked in the active panel into a RAR-archive
     @Arc.bat RAR !: !~\ !~ !~@ %: %~\
F8:  Repair a disk with Scandisk
     @rem ![Type letter-name of the disk to repair & press Enter]
     @if not !0"==" Scandisk.exe !0: /custom
F10: Switch the PC off
     cls
     @Turn_off.com
```

Each group of command lines related to one menu item can be regarded almost as a batch file. The purpose of the first item is just display of a prepared text. But when DOS is transposed to a RAM-disk (9.04, 9.09), letter-name of this RAM-disk can't be known beforehand. Letter-name problem is solved otherwise: the first command line makes current the disk, specified in a reference to command interpreter. If DOS is transposed, the HELP.TXT file is copied to the same disk too. Therefore the path to HELP.TXT file in the second command line refers to the root directory of a current disk, whichever it is. After that the last command line of the first menu item restores former appointment of the current disk.

The second menu item, called with "hot key" F4, presents a simple example of a query, performed by VC file manager, and also an example of conditional execution of a command, if user's response to the query isn't empty. Just the same can be said about the sixth menu item, called with "hot key" F8. Appeal to SCANDISK.EXE program via menu

is expedient because otherwise a very important /CUSTOM parameter (6.21) is often forgotten, and therefore testing procedures may go wrong.

The third menu item, called with "hot key" F5, presents a useful procedure of finding differences between synonymous textual files, shown in the left panel and in the right panel simultaneously. The first command line checks, whether different directories are opened in VC's panels. If yes, the FC.EXE utility in the second command line compares selected files and sends a report into a temporary file. The third command line displays this report on the screen, and the fourth command line deletes temporary file.

Unfortunately, the described procedure of finding differences doesn't include all the checks, which should be conducted. The reason is that VC file manager performs substitutions of macrocommand's data in its own buffer. As far as this buffer is only 128 bytes long, returned data increase risk of truncating command line before it is sent to command interpreter COMMAND.COM for execution. The threat of truncation is quite real, in particular, for the second command line with a call to FC.EXE. Besides that, a call to a version-specific utility via the %PATH% environmental variable is not reliable enough (see introduction article to chapter 6), and separate loading of command interpreter for each command line makes execution slow, especially when command interpreter is loaded from a diskette.

It's better to arrange complex procedures in a form of separate batch files. An example of this alternative approach is represented by batch file ARC.BAT (9.03-01), called from command lines in the 4-th and 5-th menu items. Batch files decrease risk of command line's truncation, eliminate restrictions on jumps, on searchless addressing, on usage of environmental variables.

At last, a batch file, launched from VC.MNU, is executed much faster, than equivalent separate command lines.

Note 1: in command lines of VC's configuration files, including VC.MNU, VC file manager allows to specify other menu files with *.MNU suffix, i.e. a submenu may be specified instead of a call for an executable utility. Potentially this gives an opportunity to compose hierarchical menu structures.

Note 2: in VC's configuration files, including VC.MNU, all lines, beginning with a quote sign ('), are skipped. The rest part of these lines may be filled with commentaries.

6.25-03 Extension files VC.EXT and VCEDIT.EXT.

Volcov Commander is able to arrange specific treatment for selected types of files according to their suffixes. Suffix-dependent VC's reaction on ENTER keystroke (and equally on mouse's left button double-click) is defined by textual configuration file VC.EXT. Similarly suffix-dependent reaction on F3 (VIEW) keystroke is defined by

textual file VCVIEW.EXT, and reaction on F4 (EDIT) keystroke - by textual file VCEDIT.EXT.

Suffix definitions in these files are specified just at the left edge of a line and must be followed by a colon. To the right of a colon there is specification of a command to be performed. If a keystroke must invoke more than one command, lines with the next commands follow just below; these lines must begin with one or more spaces (instead of suffix definition). If suffix definition is not followed by a command, then a file with this suffix invokes command(s) specified in the nearest line(s) below. Suffix definitions may include wildcards (? and *).

Each user has to choose particular programs, called via extension files, according to his own needs. The following example of VC.EXT file is not intended to show as much functions as can be invented, it just reflects the author's taste.

```
bas: Qbasic.exe /run !~.!
bmp:
gif:
jpg:
pcx: Lxpic.com !~.! /A
1st:
diz:
doc:
lsm:
me:
rus:
txt:
?!!: @echo _______ > !@\..\t.txt
      @copy !@\..\t.txt /B + !~.! !@\..\.
      @Emagic.exe -q -n3 !@\..\t.txt
      @Svtxt.com !@\..\t.txt
htm: @echo _______ > !@\..\t.txt
      @Html2txt.com < !~.! >> !@\..\t.txt
      @Emagic.exe -q -n3 !@\..\t.txt
      @Svtxt.com !@\..\t.txt
scr: @rem ![Press ENTER to get listing or ESC to quit]
      @echo Processing goes on. Wait...
      @Debug.exe < !~.! > !@\..\Listing.txt
      @Edit.com !@\..\Listing.txt !~.!
ima:
wbt: Diskimg.mnu
```

The shown example of VC.EXT file begins with a simple line: command files written in QBASIC language are directed to their interpreter. The latter can be found in MS-DOS6.22 release or in SFX archive OLDDOS.EXE, available at Microsoft's ftp-server

ftp://ftp.microsoft.com/softlib/mslfiles/ . You have to remember, that a corresponding record must be written into versions table of SETVER.EXE driver (5.01-02) in order to enable QBASIC.EXE usage under MS-DOS7.

The next group of lines in VC.EXT file directs picture files (*.BMP, *.GIF, *.JPG, *.PCX) to picture viewer LXPIC.COM, developed by Stefan Peichl. Version 7.3 of this viewer (dated 2002), packed into archive LXPIC.ZIP, can be downloaded from internet site http://hplx.pgdn.de/ .

Further in VC.EXT file a large group of suffixes is enlisted (*.1ST, *.DIZ, etc.), which usually denote textual files. The main problem of reading textual files in russian is caused by a large variety of possible codes. This problem is solved by utility EMAGIC.EXE, written by Sergey Gernshtein. EMAGIC.EXE analyses statistics of text, determines the type of code and then automatically translates text into codepage CP866. Archive EMAGIC.ZIP with EMAGIC.EXE utility inside can be downloaded from internet ftp-server ftp://ftp.botik.ru/pub/msdos/convert/ . Those, who don't use codepage CP866, are suggested to delete a line with a call for EMAGIC.EXE out of VC.EXT file.

Display of textual files on the screen is a mission of viewer SVTXT.COM, written by Lo Hung Che for FreeDos project. This viewer, packed in archive PG116.ZIP, can be downloaded from server ftp://sunsite.unc.edu/pub/micro/pc-stuff/freedos/files/util/file/pg/ . The SVTXT.COM viewer shows long textual lines in wrapped form and is able to cope with the first 64 kb of indefinitely long files. The SVTXT.COM viewer can't extract parts of text, but this can be done later with EDIT.COM editor (6.09), because a copy of the shown text, translated into codepage CP866, will be saved as T.TXT file in a directory intended for temporary files.

Hypertext files with *.HTM suffix may be treated just as ordinary textual files, if their hypertext markup is taken off beforehand. The latter mission is performed by HTML2TXT.COM utility, written by Shin Chan. Archive HTML2T08.LZH, containing the HTML2TXT.COM utility, can be downloaded from internet site http://www.vector.co.jp/download/file/dos/net/fh050307.html . The shown sequence of hypertext file's translations can't be recommended for those who don't read in russian. It's better for them to send *.HTM files directly to hypertext viewer VH.EXE. This viewer, packed into VIEWHT25.ZIP archive, can be downloaded from internet server ftp://ftp.bu.edu/pub/mirrors/simtelnet/msdos/html/ .

Suffix *.SCR most often marks command files for debugger DEBUG.EXE, but there may be exceptions. Besides that, execution of some debugger's command files may inflict undesirable consequences. Therefore for files with *.SCR suffix a query is issued by Volcov Commander. If the user responds to the query with ENTER keystroke, command file will be sent to debugger, and the returned listing will be presented for editing on the screen. This procedure is very convenient for correcting those errors in command files, which are revealed in returned listing (9.07-01).

Files with *.IMA suffix are images of diskettes. These files may be either subjected to unpacking or written back onto a diskette. An appeal to user's choice compels to create a new submenu. Let it be named DISKIMG.MNU. In this submenu writing to disk is performed by utility IMG.EXE, and unpacking – by utility DDI2HDD.EXE. Both these utilities can be downloaded from internet ftp-server ftp://ftp.elf.stuba.sk/pub/pc/utildisk/ inside archives IMG.ARJ and DDI2HDD.ZIP accordingly. An example of submenu is shown below; its text must be stored as unformatted textual file DISKIMG.MNU in common directory with other VC's configuration files. In DISKIMG.MNU, just as in VC.MNU, names of "hot keys" must be typed just at the left edges of lines without preceding spaces.

```
F4:  Write the image onto diskette in drive A:
     @Img.exe !~.! A:
F8:  Unpack files from the diskette image
     @if not %~\"==" Ddi2hdd.exe !~.! %:%~\. /d /s
     @if %~\"==" echo Archive in passive panel must be closed
     @if not %~\"==" echo Unpacked files are in %:%~\!~ folder
```

While composing your own samples of VC's extension files you have to keep in mind, that besides explicitly specified operations Volcov Commander performs some implicit suffix-dependent processing. In particular, after interpretation of all lines in VC.EXT file, selected files with *.BAT, *.COM and *.EXE suffixes are by default transferred to command interpreter for execution, and archive files are further processed according to specifications in VCARCH.EXT file (6.25-04). If the mentioned suffixes are specified inside VC.EXT file, then the default treatment of the corresponding files will be intercepted and wouldn't be performed. Because of this reason suffixes of archives and of executable files are not specified in VC.EXT.

Similarly, after F3 (VIEW) keystroke Volcov Commander transfers to its internal viewer only those files, which are not intercepted by suffix specifications in extension file VCVIEW.EXT. Internal VC's viewer shows contents of any file in binary and textual forms. The opportunity to view any file "as it is" is itself valuable, and therefore the VCVIEW.EXT file, in my opinion, is not needed. If your opinion is different, you may write your own version of VCVIEW.EXT, taking the shown VC.EXT file as example. Syntax of records in VCVIEW.EXT and VC.EXT files is identical.

When a file, which is to be viewed or edited, is selected inside an opened archive in active panel, then both F3 and F4 keystrokes invoke file's unpacking from that archive into temporary directory %TEMP%\VC.000. Then the `!~.!` mark in lines of VCVIEW.EXT and VCEDIT.EXT files is replaced by corresponding macrocommand with name of the unpacked file, preceded by path to that temporary directory. Thus the F3 keystroke enables viewing inside archives without their explicit unpacking. Similarly the F4 keystroke subjects the unpacked file to just any operation, specified in lines of VCEDIT.EXT file.

Editing operation isn't as safe as viewing: the editor program may inflict damage to the subjected file, especially harmful, when the subjected file is not an unpacked copy. Succession of commands in VCEDIT.EXT file must protect susceptible non-textual files against accidental damage. Therefore binary files and archives in the example below are intercepted by an empty operation (REM) and thus escape the risk of being damaged.

```
bas: Qbasic.exe !~.!
bin:
cab:
com:
exe:
rar:
zip: @rem
bmp:
gif:
jpg:
pcx: Lxpic.com !~.! /A
*:    Edit.com !~.!
```

Picture files are protected by separate interception: it enables to view pictures inside archives. All the rest files are directed to editor program by asterisk wildcard specification (instead of suffix) in the last line. As far as version 4.99.07 of Volcov Commander provides no internal editor, in the proposed example of VCEDIT.EXT file the EDIT.COM editor (6.09) is charged with this mission.

Note 1: all *.EXT files are read by VC shell once when it starts. Changes in *.EXT files will not come into effect unless VC shell is shut down and then launched anew.

Note 2: in VC's extension files all lines, beginning with a quote sign ('), are skipped. The rest part of these lines may be filled with commentaries.

6.25-04 Archives processing configuration file VCARCH.EXT

In panels of VC's version 4.99.07 archives can be opened just as directories. This feature is achieved by intercepting STDOUT output of archiver utilities and parsing it word-by-word according to a predetermined order. If the output line is found conforming to the order, then selected words from this line can be identified as a name of a file, its date, length, etc. These data are displayed in VC's panel just as it is done for an ordinary directory.

As far as output data display format is not the same for different archiver utilities, VC shell has to adapt the parsing order according to type of archive file, defined by its suffix. Parsing order models for different archive types are specified in configuration file VCARCH.EXT. Moreover, VCARCH.EXT enables to specify several parsing order

models for archives of one type, and the output line will be parsed if it is found conforming to at least one model.

VCARCH.EXT consists of separate entries for each archive type, each entry is up to 5 lines long. The first line in each entry defines archive suffix (it must be specified just at the left edge of the line) and corresponding parsing order model(s). Suffix is separated from parsing order model(s) with a colon. Definitions of parsing order models are composed of space separators and upper case characters, interpreted in the following way:

A	– attributes of a file, stored inside archive;
C	– size of a file after compression;
D	– day or date of the last change;
L	– line feed to continue parsing in the following line;
M	– month;
N	– name of a file, stored inside archive;
P	– path, stored inside archive;
S	– original size of a file, stored inside archive
T	– time of the last change;
W	– a word to be ignored;
Y	– year;
;	– semicolon separator between parsing order models;
' or "	– quotes to enclose any recognizable symbol(s) or text;
\	– backslash as the first symbol of a parsing order model signifies that this model refers to a directory.

The next four lines in each entry commence with at least one space and define calls to the corresponding archiver utility for performing particular operations:

2-nd line:	send table of archive's contents into STDOUT. This operation is initiated automatically each time when panels are redrawn.
3-rd line:	unpack selected archive file(s). Unpacking is initiated by F5 (copy) keystroke, if archive is opened in active panel.
4-th line:	add file(s) to the archive. Adding is initiated by F5 (copy) keystroke, if archive is opened in passive panel.
5-th line:	delete file(s) from archive. This operation is initiated by F8 (delete) keystroke.

Unpacking operation, specified in the 3-rd line, can also be initiated by F3 (view) and F4 (edit) keys. In this case the result of unpacking is written into directory for temporary files and is immediately presented for viewing or editing according to specifications in extension files VCVIEW.EXT and VCEDIT.EXT.

Out of a large variety of archive types only a few are actively used for packing. Most part of other entries in VCARCH.EXT file is preserved for a single purpose: for access to archives types, which may be occasionally encountered in software stores. Entries for such archive types in VCARCH.EXT file may be reduced to three lines.

During interpretation of command lines in VCARCH.EXT file Volcov Commander gives no access to environmental variables, but provides specific macrocommands, which differ from those available in other extension files. Macro command calls in VCARCH.EXT file are induced by the following marks:

!A	– mark to be replaced by archive file name with preceding path.
!T	– mark to be replaced by path to directory for temporary files.
!F	– mark to be replaced by a filename, selected with left mouse's button click.
!M	– mark to be replaced by one or more filenames, selected by right mouse's button click.
!@	– mark to be replaced by a name of temporary file, containing a list of filenames, selected by right mouse's button click.

Proposed examples of entries for VCARCH.EXT file are shown below. All executable files, mentioned in entry examples, must be accessible along path(s), specified in value of environmental variable %PATH% (2.02-02).

```
ARC: N S W C W D T W
     Pkxarc.exe -v !A
     Pkxarc.exe -e !A @!@
BSA:
BSN: W S C W W D "-" M "-" Y T A L N; W S C W W D "-" M "-1" Y T A L N
     Bsa.exe v !A
     Bsa.exe x -S !A @!@
CAB: M "-" D "-" Y T A S N; N ":" W W W W W W W
     Extract.exe /d !A
     Extract.exe /e !A !M
RAR:
R0?:
R1?:
R2?: "*" N L S C W D T A W W W; N L S C W D T A W W W
     Unrar.exe v -r -w!T !A
     Unrar.exe x -r -w!T !A @!@
     Rar.exe a -std -ds -r -rr -ems- -w!T !A @!@
     Rar.exe d -std -r -rr -ems- -w!T !A @!@
TAR: A W S N L
     Tar.exe -tvf !A
     Tar.exe -xnf !A !M !F
```

```
UHA: N S D "-" M "-" Y T A
     Uharcd.exe  l -y+ !A
     Uharcd.exe  x !A !M !F
ZIP:
??!:
?$:
??$: S W C W D T W A N
     Pkunzip.exe -v !A
     Pkunzip.exe -d !A @!@
     Pkzip.com -b!T -P -wHS !A @!@
     Pkzip.com -b!T -d !A @!@
1:
2:
3:
4:
Z:   M "-" D "-" Y T S A C N
     Command.com /c Icomp.exe -l -h !A
     Command.com /c Icomp.exe -d -i -h !A !F
```

Volcov Commander's version 4.99.07 release is supplemented with VCARCH.EXT file, containing entries for many archive types. Examples of entries, shown in this article, shouldn't be regarded as a complete VCARCH.EXT file. These examples represent separate entries, which differ from those in original release. Proposed entries may be added to original VCARCH.EXT file or may replace original entries in this file. All archiver programs, mentioned in proposed entries, can be downloaded from a vast collection of archivers in ftp-server ftp://ftp.elf.stuba.sk/pub/pc/pack/ .

6.26 XCOPY.EXE – copying utility

The XCOPY.EXE utility copies files and can copy directories with all their contents, including subdirectories. XCOPY.EXE needs an auxiliary file XCOPY32.EXE to be present in the same directory. A command line with a call for XCOPY.EXE may look like this:

```
Xcopy.exe D:\TEMP\*.* C:\DOS /A /D /P /S /V /W
```

where:

D:\TEMP*.* – path and mask examples for the files to be copied.
C:\DOS – example of a target path for copying.
/A – copy those files only, which have the "A" attribute set, and leave this attribute intact. The /M option means the same, but the "A" attribute will be taken off.
/D – copy those files only, whose source is newer than synonymous sample in the target directory. If after the /D parameter a date is specified (for

example, /D:18/12/2003), then those files only will be copied, which are newer than that date (see note 2 below).

/P – prompt before copying each file into target directory.

/S – copy directories and subdirectories except empty ones. Specification of both /S and /E options forces to copy empty directories too.

/V – verify each copied file.

/W – wait for permission to copy, confirmed by keystroke (in order to give time for changing either source or target storage media)

Note 1: being launched under MS-DOS, XCOPY.EXE can't copy files having H (Hidden) or S (System) attributes. This and some other functions can be performed by XCOPY.EXE inside Windows's "DOS box" only.

Note 2: data format, specified after the /D parameter, depends on national settings, defined by COUNTRY command (4.05). In any case data format must be just that returned by DATE command (3.08).

Note 3: in MS-DOS8 auxiliary file XCOPY32.EXE is renamed to XCOPY32.MOD.

Note 4: a synonymous utility XCOPY.EXE has been written as a part of FreeDOS project in 2003 by Rene Ableidinger. This utility needs no auxiliary files and is able to copy files with HS attributes under MS-DOS7. Archive RXCOPY2.ZIP, containing this utility, can be downloaded from internet server ftp://sunsite.unc.edu/pub/micro/pc-stuff/freedos/files/dos/ .

Chapter 7 Debugger's assembler commands

Are commands of archaic DEBUG.EXE worth the time to get acquainted with? This question invokes recollections. More than a half century ago computers have implemented time sharing for several terminals. Each terminal was used to launch a separate program, and computer's memory had to be distributed between these programs. For claiming memory requirements program's file headers were introduced. Since then those assemblers, which couldn't automatically compile file's headers, have been regarded obsolete. Just that attitude has been inherited by DEBUG.EXE.

More sophisticated assemblers automatically compile headers and jump target addresses. This great convenience, however, has its back side: freedom of manipulation with address space and with segment registers becomes lost. Applications don't suffer because of that, but for research and system tasks it may become a sufficient obstacle. Just because of this reason the MASM assembler wouldn't compile program examples shown in parts 9.06, 9.08, 9.10 of this book. Similar unacceptable fragments were found in BIOS codes and even in loader modules of Windows-XP operating system. Codes that can't pass through MASM, can be compiled by DEBUG.EXE.

For firsthand acquaintance with programming the most simple assembler tool should be preferred. Besides simplicity, DEBUG.EXE has a number of advantages, which make it the best suitable choice for this book:

– first, it gives the most clear notion of machine codes usage;
– second, it combines assembling and debugging capabilities;
– third, it provides access to hardware and to OS structures;
– fourth, it is able to interact with the user.

Though DEBUG.EXE can send to CPU for execution just any machine code, variety of all x86 machine commands is too vast and may bewilder a newbie. Therefore the 7-th chapter presents an easily comprehensible selection: command set of that DEBUG.EXE version, which is supplied within Windows-95/98 release. These commands constitute a subset of all modern CPU's commands, but this subset plays a very important role. It includes the most widely used commands. About 96% of the whole currently active computer park "understands" these commands. Compatibility of machine codes for the majority of modern computers is based on just that subset, which is presented here.

More recent commands in modern CPU's may differ. Nevertheless a number of such commands is widely acknowledged and is necessary for tuning modern computers. As exception, the 7-th chapter includes descriptions of several commands, which are not "known" to DEBUG.EXE. Descriptions elucidate the ways to employ these commands in a form of machine codes. In any case this wouldn't be an obstacle for debugging programs with these codes.

DEBUG.EXE accepts assembler commands, when it is switched to translation of assembler language into executable machine codes (6.05-02). Though each dialect of assembler language has its peculiarities, general composition of assembler commands is common. A line begins with command name, followed by operand(s). Operands in some commands must be preceded by a marker of operand's type. If an assembler command contains several operands, these are separated by a comma. Commands, which produce any numerical result, overwrite their first (leftmost) operand with this result. Former contents of this register (or memory cell) become lost. Examples of command files with switching DEBUG.EXE to assembler language translation and with a lot of assembler commands are presented in articles 9.05, 9.06, 9.08, 9.10.

As far as alternative command's specifications are too numerous, some variables and operands in chapter 7 are given easily recognizable fixed lower case designators, just the same in all examples. Allowable substitutions for these designators together with some other terms, used in DEBUG's assembler dialect, are shown and explained in the table below.

Table 7.00

Designators	Explanations and allowed alternatives
`bl`	any of 8-bit registers: AL, CL, DL, BL, AH, CH, DH, BH.
`bx`	any of 16-bit registers: AX, CX, DX, BX, SP, BP, SI, DI.
`ss`	any of segment registers: CS, DS, ES, SS.
`ST`	coprocessor's top stack register ST(0) (see note 1 below)
`ST(0-7)`	any coprocessor's stack register from ST(0) to ST(7)
`far`	marker of a 4-byte address (segment + offset).
`byte ptr`	marker of a one-byte operand (ptr = "pointer")
`word ptr`	marker of a 2-bytes operand (a word)
`dword ptr`	marker of a 4-bytes operand (a double word)
`qword ptr`	marker of a 8-bytes operand (a quadruple word)
`tbyte ptr`	marker of a 10-bytes operand
`f`	any single hexadecimal digit from "0" to "F"
`±7f`	any signed hexadecimal number from –7Fh to +7Fh
`ff`	any hexadecimal number from 00h to FFh
`ffff`	any hexadecimal number from 0000h to FFFFh
`aaaa`	address for "short" jumps (see note 2 below)
`[bp+si+ffff]`	expression enclosed in square brackets means addressing to operand in memory. Offset of the corresponding memory cell(s) is to be found by evaluating the given expression. Two groups of expression forms are allowed (see notes 3 and 4 below).

Note 1: coprocessor's top stack register is normally named ST(0), but in several positions, where no other register can be addressed instead, DEBUG.EXE accepts abridged name ST.

Note 2: offset is specified as "aaaa" for control transfer commands with one address byte. This offset must be within ±7Fh vicinity from the next machine command. If vicinity condition is violated, DEBUG.EXE responds with error message.

Note 3: expressions of the first group produce offset, alluding by default to segment address in DS. These expressions are either

just an offset specification: [ffff].
or a single register reference: [BX], [DI], [SI],
or a sum of 2 registers: [BX+DI], [BX+SI],
or a sum with displacement: [BX±7f], [BX+ffff],
[BX+DI±7f], [BX+DI+ffff], [BX+SI±7f], [BX+SI+ffff],
[DI±7f], [DI+ffff], [SI±7f], [SI+ffff].

Note 4: expressions of the second group include a reference to BP register and produce offset, alluding by default to segment address in SS:
[BP+DI], [BP+SI], [BP±7f], [BP+ffff], [BP+DI±7f],
[BP+DI+ffff], [BP+SI±7f], [BP+SI+ffff].

Note 5: register names in expressions may be separated by minus sign, but this doesn't affect the result: sum is counted in any case.

Note 6: the default segment register may be changed by explicit specification of a prefix byte (7.02-01).

Note 7: when operand's type marker ("byte ptr", "word ptr"...) isn't present in assembler command, DEBUG.EXE determines type of operand according to the register, containing the other operand: 16-bit register (AX, BX...) sets word type, 8-bit register (AL,AH,BL...) sets byte type. If a command doesn't address to a register, then operand's type marker becomes a required item. In any case these markers may be truncated to 2 characters: "by", "wo", etc.

7.01 Control instructions

Having been switched to translation of assembler commands, DEBUG.EXE accepts commands and also control instructions. The latter are not translated into machine code, but rather affect the process of translation and may play a very important role.

7.01-01 DB – data bytes insertion instruction

The DB (= Data Byte) instruction informs DEBUG.EXE, that the following string of bytes must be treated not as an assembler command, but rather as separate bytes of data, which should to be written into assembled code "as they are". Each byte is represented by

two hexadecimal digits without the trailing "h". String of bytes may include group(s) of ASCII characters, enclosed in quotes or in double quotes. ASCII characters (except the enclosing quotes) are translated into hexadecimal code byte-by-byte. Comments after the DB instruction are not allowed. Here is a DB instruction usage example:

```
DB 71 6C 65 'data array'
```

Note 1: while unassembling machine codes there may be encountered byte(s), which can't be identified as machine commands, "known" to DEBUG.EXE. Then "DB" is displayed as a prefix to each such byte.

7.01-02 DW – data words insertion instruction

The DW (= Data Word) instruction informs DEBUG.EXE, that the following string must be treated not as an assembler command, but rather as separate words of data, which should be written into assembled code "as they are". Each word consists of up to 4 hexadecimal digits. If there are less than 4 digits in a word, it will be automatically supplemented to 4 digits with higher order zeros. In assembled machine code the least significant 2 digits of each word constitute the first byte, and the most significant 2 digits constitute the following byte. String of words after the "DW" instruction may include group(s) of ASCII characters, enclosed in quotes or in double quotes. These characters (except the enclosing quotes) are translated into hexadecimal representation one byte per character, just as after the "DB" instruction (7.01-01). Comments after the DW instruction are not allowed. Here is a DW instruction usage example:

```
DW 71A0 F01 06D5 "other key" 0FFF
```

7.01-03 ORG – target address change instruction

The ORG instruction informs DEBUG.EXE, that machine codes of the assembled commands, from the next one and on, must be written into other place, starting from that address, which is specified after the ORG instruction. Processor's registers are not affected by ORG instruction. In course of interactive debugger's sessions the ORG instruction enables to navigate along the assembled code in order to correct errors and those references forward, which couldn't be specified in advance.

In debugger's trial command files the ORG instruction is used to fix positions of restart points and of jump target points (example – in 9.02-02). Reserved free space, preceding positions of fixed points, helps to avoid tedious address recalculations, which otherwise would be necessary each time you have to add or delete bytes of code in any previous part of command file. Comments after the ORG instruction are allowed. Usage examples are shown in the following table:

Examples	Performed action
`ORG ffff:ffff`	Set explicitly both segment address and offset
`ORG fff`	Leave segment address intact, set offset 0fffh
`ORG ss:ffff`	Refer to segment register "ss:", set offset ffffh
`ORG ss:`	Refer to segment register "ss:", offset 0000h is implied

7.01-04 Instruction "Empty line"

Empty line, containing no commands, no instructions, no data and no comments, is itself regarded by DEBUG.EXE as an instruction, forcing to return from translation of assembler commands, described in chapter 7, to normal control with those commands, described in articles 6.05-02 – 6.05-23. In order to input this instruction from keyboard you have to leave the last presented line empty and just press ENTER. When assembler commands are received from a command file via input redirection, return to normal control is induced by the first encountered empty line in this command file. Therefore a care should be taken about absence of empty lines inside any block of assembler commands in command file(s), and equally about presence of an empty line, marking the end of each block of assembler commands.

7.01-05 Semicolon – comments insertion instruction

Being encountered in any position in a line with assembler command, semicolon (;) is interpreted by DEBUG.EXE as instruction to proceed to the next assembler command line at once, skipping translation into machine codes of the semicolon itself and of all following characters in the rest part of current command line. This mission of semicolon is used in order to append comments to assembler commands.

Note 1: a line beginning with semicolon is not regarded empty (7.01-04) and doesn't force DEBUG.EXE to cease translation of assembler commands into machine code. This gives an opportunity to insert headers and multi-line commentaries.

Note 2: DEBUG's message " ^ Error" pointing upwards just towards semicolon in the previous line means that an error is present in preceding part of this line. Most probably command line is considered not complete because of absence of some required parameter.

Note 3: semicolon can't be used to append commentaries to those assembler command lines, which begin with control instruction DB or DW, and also when DEBUG.EXE is not switched to translation of assembler commands.

7.02 Prefixes

In machine code of x86 platform CPU's several particular bytes are given a special prefix status. Each such byte is not a separate machine command. But a prefix byte, being encountered preceding a machine command, forces CPU to change the manner of it's interpretation or execution. Effect of a prefix byte isn't spread beyond the nearest following command.

DEBUG.EXE allows to specify a prefix byte in a separate line before that assembler command, which is to be affected by prefix, for example:

```
CS:
ADD byte ptr [BX],0F
```

Specification of a prefix byte just before the affected command in the same line is equally allowable:

```
CS: ADD byte ptr [BX],0F
```

Machine command's code may be preceded by up to four prefixes, if their effects don't contradict to each other. All prefixes in one command must be different, duplicate prefixes are not allowed.

7.02-01 Segment override prefixes

Most assembler commands don't include explicit segment address specification. Absolute memory addresses (see note 2 to 6.05-01) for such commands are calculated by CPU on the basis of default segment register assignment (see notes 3 and 4 to table 7.00). Segment override prefixes force CPU to read segment address from another segment register instead of the default one for the nearest following command. Naturally, segment override prefixes can be applied to those commands only, which appeal to memory and therefore imply calculation of absolute address.

DEBUG.EXE "knows" four segment override prefixes, inheriting names of corresponding segment registers (see second column of the table below). Examples of CS: prefix usage have been shown in article 7.02. Numerous other similar prefix usage examples can be found in assembler texts, presented in articles 9.06 and 9.08.

Modern CPUs have not four, but six segment registers. Segment override prefixes for auxiliary segment registers FS and GS are not "known" to DEBUG.EXE, but it allows to specify these prefixes as data by means of DB instruction (7.01-01) and doesn't hinder to proper interpretation of these prefixes by CPU. Of course, for proper interpretation of these prefixes the 80386 or newer CPU is required.

Codes	Examples	Comments
2E	CS:	
3E	DS:	
26	ES:	
36	SS:	
64	DB 64	relative to FS: segment register
65	DB 65	relative to GS: segment register

Note 1: segment override prefixes can't affect default assignment of ES: register, in particular, for string commands CMPSB, CMPSW, INSB, INSW, MOVSB, MOVSW, SCASB, SCASW, STOSB, STOSW.

7.02-02 LOCK – system bus lock prefix

Prefix LOCK corresponds to prefix byte F0h, which induces CPU to send "Bus busy" signal and to keep it active until execution of the following command terminates. This is necessary in computers with several processors in order to prevent uncoordinated access to shared memory resources. For ordinary computers with a single processor prefix LOCK is not needed.

The LOCK prefix can be specified for writing into memory with commands ADC, ADD, AND, DEC, INC, NEG, NOT, OR, SBB, SUB, XCHG, XOR. But when the same commands perform reading only or operate with registers, then LOCK prefix shouldn't be specified. In such cases CPU responds to byte F0h with exception 06h, inducing a call for interrupt INT 06 handler (8.01-07).

Code	Example
F0	LOCK

Note 1: Intel's CPUs don't allow combinations of LOCK prefix with any of repetition prefixes (7.02-03, 7.02-04) in one command.

Note 2: modern processors, having more than 8 control registers, allow prefix byte F0h for access to control registers with MOV command (note 1 to 7.03-58). In this case F0h byte is interpreted not as LOCK prefix, but as a prefix for access to registers CR8 – CR15.

7.02-03 Repetition prefix REPNZ

Name REPNZ stands for "REPeat while Not Zero". The REPNZ prefix induces cyclic execution of the nearest following command. Repetition cycle terminates, when at least one of two conditions is met:

- the executed command finds equal operands and therefore sets zero flag (ZF) into ZR state (6.05-15).
- a number in CX register becomes zero because the prescribed number of repetitions in CX register has exhausted.

Prefix name REPNE, standing for "REPeat while Not Equal", is accepted by DEBUG.EXE as equivalent of REPNZ. Both correspond to the same prefix byte F2h, which forces CPU to perform cyclically the following operations:

- first, to check the CX= = 0 condition. If it is met, then terminate the cycle. If it is not met, then decrement by a unity the number in CX register.
- second operation is to reset zero flag into NZ state.
- the next operation is to perform that command, which is preceded by repetition prefix.
- the last operation is to check whether the zero flag is set into ZR state. If yes, then terminate the cycle. If no, then return to the first operation, to CX= = 0 check.

As long as both described conditions are not met, CPU continues to perform the cycle. As soon as at least one condition is met, CPU leaves the cycle and proceeds to the next command, following the one executed within the cycle.

Repetition prefixes are used with string commands, which automatically increment or decrement contents of index register (SI, or DI, or both) at each iteration. Hence the actual address of their operand(s) is shifted from one iteration to the other. String commands CMPSB, CMPSW, SCASB, SCASW affect not index only, but also the state of ZF flag. Therefore repetition prefixes with these commands enable to analyze string(s) of bytes or words. When repetition prefix is used with commands, not affecting the ZF flag (INSB, INSW, MOVSB, MOVSW, OUTSB, OUTSW, STOSB, STOSW), then these commands will be just repeated that number of times, which is preset in CX register.

Code	Examples
F2	REPNE
F2	REPNZ

Note 1: when a command is preceded by several prefixes, including a repetition prefix, then archaic processors (more obsolete, than 80386) sometimes can't resume execution of the repetition cycle after interrupts. If such combination of prefixes can't be avoided, one has either to recheck explicitly the conditions of cycle's termination or else inhibit interrupts with CLI command (7.03-12) for the time of cycle execution.

Note 2: repetition prefixes shouldn't be applied to non-string commands, because it results in those byte combinations, which may be interpreted by modern processors as operation code extensions (7.02-08), in particular, denoting the SSE commands.

Note 3: when repetition prefix is used in combination with operand size override prefix (7.02-06), the prescribed number of repetitions is read not from 16-bit CX register, but from 32-bit ECX register. For such cases it's important to remind about preparation of a proper value in upper part (bits 31 – 16) of ECX register.

7.02-04 Repetition prefix REPZ

Name REPZ stands for "REPeat while Zero". The REPZ prefix causes iterative execution of the command it precedes. Repetition cycle terminates, when at least one of the following two conditions is met:

- the executed command finds different operands and therefore resets zero flag (ZF) into NZ state (6.05-15).
- a number in CX register becomes zero because the prescribed number of repetitions in CX register has exhausted.

Prefix names REP (=REPeat), REPE (= REPeat while Equal) and REPZ are regarded by DEBUG.EXE as equivalent: either of them corresponds to the same prefix byte F3h. Having encountered byte F3h, processor performs the same sequence of operations as for prefix REPNZ (7.02-03), except that initial state of ZF flag is reversed to ZR, and the target state of ZF flag is the opposite (NZ). All other peculiarities of command's execution with REPNZ prefix, described in article 7.02-03 and in the following notes, are equally inherent to execution of commands with prefix REPZ.

Code	Examples
F3	REP
F3	REPE
F3	REPZ

Note 1: prefix REPZ is often used with commands CMPSB and CMPSW for comparison between two strings of characters – names, paths, signatures. When comparison cycle terminates, the result is expressed by state of ZR flag: the set state (ZR) proves identity, the reset state (NZ) is an evidence of difference.

7.02-05 Synchronizing prefixes WAIT and FWAIT

Prefix names WAIT and FWAIT correspond to the same prefix byte 9Bh. It is used with commands, which imply code transfer between CPU and asynchronous coprocessor. Byte 9Bh forces CPU to wait for arrival of readiness confirmation signal from coprocessor to CPU's pin "BUSY". In particular, prefix byte 9Bh should precede to ESC commands (7.03-22) and to coprocessor's commands (7.04), if machine code is to be executed by a CPU without internal arithmetic coprocessor.

For modern CPUs, comprising an integrated arithmetic coprocessor with hardware synchronization means, former mission of prefix byte 9Bh is not needed. Byte 9Bh is ignored, if bit 01h "coprocessor synchronization" in control register CR0 (A.11-4) is reset to zero. By default bit 01h is set, and then byte 9Bh may induce a call for INT 07 handler, if at the same time task switch flag (bit 03h in CR0) is set too. Task switching is conjoined with necessity to process exceptions, registered by coprocessor. The INT 07 handler can be charged with this mission, and then it's fulfillment will be ensured by pertinent usage of the WAIT prefix.

Code	Examples
9B	FWAIT
9B	WAIT

7.02-06 Operand's size override prefix

In real mode modern CPUs by default emulate 16-bit operations of obsolete processor 8086. But in fact modern CPUs have 32-bit general purpose registers. Sometimes it is desirable to get access to the whole 32-bit register, while CPU is still kept in real mode. This can be achieved with operand size override prefix byte 66h, which is properly "understood" by all 32-bit CPUs, belonging to x86 platform.

As far as DEBUG.EXE doesn't "know" operand size override prefix, it should be introduced by DB instruction (7.01-01), for example:

```
DB   66
SHR  AX,CL
```

In the shown example presence of prefix byte 66h modifies action of SHR command (7.03-83) so that it will affect the whole 32-bit register EAX. In particular, if the number of shifts, preset in CL register, is 10h (= 16 decimal), then contents of bits 31 - 16 in EAX will be shifted into bits 15 - 0 and will become accessible in AX as an ordinary 16-bit operand.

Being preceded by prefix byte 66h, stack commands PUSH, PUSHF, POP, POPF operate with four bytes at once. Bytes are pushed into stack in descending significance order: from the most significant to the least. Contents of bits 31 - 16 in EAX register may be read via stack in the following way:

```
DB   66
PUSH AX
POP  BX
POP  BX
```

In this example prefix byte 66h forces to push into stack the whole contents of 32-bit register EAX. Then first POP command moves into BX register two least significant bytes

of EAX, but these are available just from AX and therefore are not needed. The next POP command overwrites former data in BX register with the required most significant bytes of EAX contents.

The CMP command (7.03-14) with operand size override prefix compares four-byte operands, including those stored in 32-bit registers. For commands with operand size override prefix the operands stored in memory as well as operands, specified directly in executable code, also must be of the DWORD type, i.e. 4 bytes long. Unfortunately, DEBUG.EXE can't assemble machine commands for CPU with attached operands of DWORD type. If necessary, extra data bytes may be appended by DB instruction (7.01-01).

Note 1: programs using operand size override prefix can't be executed by 16-bit CPUs.

Note 2: operand size override prefix can't be specified before commands with one-byte operand(s), including those in one-byte registers (AH, AL, BH, etc.), and also before one-byte string commands (CMPSB, INSB, LODSB, MOVSB, OUTSB, SCASB, STOSB). These combinations of codes may be interpreted by modern processors as SSE commands.

Note 3: operand size override prefix can't be specified before commands with operands in segment registers, since these are 16-bit registers in both 16-bit and 32-bit processors. However, this restriction doesn't relate to commands, which read segment address for access to a particular memory cell.

Note 4: when a program is tested with commands "Proceed" (6.05-14) or "Trace" (6.05-17), then DEBUG.EXE doesn't show as the next machine command that one, which follows prefix byte 66h. Nevertheless 32-bit CPUs always accept prefix byte 66h together with the following machine command and execute both at one step.

Note 5: operand size override prefix always forces the non-default size of operands. When bit 6 in byte 06h of code segment descriptor (note 5 to A.12-2) specifies default 32-bit size of operands, then prefix byte 66h forces 16-bit operand's size. Here and further in this book default 16-bit operand's size is implied, just as it is automatically set by CPU's "shadow" registers, when CPU begins to work in real mode after being switched on.

7.02-07 Address size override prefix

For testing memory and for several other tasks it is necessary to get access to the whole address space without those restrictions, which are inherent to 32-bit addressing in protected mode. In real mode access to the whole address space is possible, but it needs the utmost 4-Gb segment size to be set (example - in article 9.10-01) and, besides that, non-default 32-bit addressing to be allowed. Address size override prefix byte 67h allows non-default 32-bit addressing to a single nearest following command.

Prefix byte 67h is not "known" to DEBUG.EXE. Therefore it has to be introduced as data by DB instruction (7.01-01). When a command is preceded by several prefixes, then prefix byte 67h is specified before operand size override prefix (7.02-06), but after segment override prefix (7.02-01). Naturally, there is no sense in combination of prefix byte 67h with those commands, which have no deal with memory cells.

Prefix byte 67h affects code length of many machine commands and changes interpretation of all indirection expressions, listed in notes 3 and 4 to table 7.00. Only commands with implicit indirection remain unchanged: CMPSB, CMPSW, LODSB, LODSW, MOVSB, MOVSW, SCASB, SCASW, STOSB, STOSW. For assembling all other commands with prefix byte 67h capabilities of DEBUG.EXE are insufficient.

The table below shows original interpretation of indirection expressions (in the left column) in comparison with interpretation, affected by address size override prefix byte 67h (in the right column). The table lists only those interpretations, which correspond to machine commands of the same length, performing the same operation(s). Hence the shown interpretations can be exchanged, and thus DEBUG.EXE can be "deceived". You are free to specify to DEBUG.EXE that indirection expression from the left column, which corresponds to the desirable CPU's interpretation, chosen in the right column.

Original form	Affected by prefix 67h
[BP+DI]	[EBX]
[BX]	[EDI]
[BP+DI±7f]	[EBX±7f]
[BX±7f]	[EDI±7f]
[BP±7f]	[ESI±7f]
[DI±7f]	[EBP±7f]

Note 1: programs using address size override prefix can't be executed by 16-bit CPUs.

Note 2: when a program is tested with commands "Proceed" (6.05-14) or "Trace" (6.05-17), then DEBUG.EXE doesn't show as the next machine command that one, which follows prefix byte 67h. Nevertheless 32-bit processors always accept prefix byte 67h together with the following machine command and execute both at one step.

Note 3: address size override prefix always forces the non-default address size. When bit 6 in byte 06h of code segment descriptor (note 5 to A.12-2) specifies default 32-bit address size, then prefix byte 67h forces 16-bit address size for the nearest following command. Here and further in this book default 16-bit address size is implied, just as it is automatically set by CPU's "shadow" registers, when CPU begins to work in real mode after being switched on.

7.02-08 Operation codes extension prefixes

Operation codes of x86 platform CPUs have been formed as a result of long evolution. At each stage of evolution a set of machine commands has been supplemented with new commands. Therefore several bytes have been used as operation code extension prefixes, affecting interpretation of operation codes in CPU's instruction decoder. These prefixes have no common specific function, except that each such prefix is able to introduce a separate group of various machine commands.

Long ago, at times of huge mainframes, byte FFh has been charged with mission of operation code extension prefix. Now it is regarded as first byte in operation codes of many different machine commands, described in part 7.03. Later bytes D8h – DFh were devoted to introduce arithmetic coprocessor's commands, described in part 7.04. In 1990-ties new commands for Pentium processor were introduced by prefix byte 0Fh; several of these commands are mentioned in table 6.05-18.

Nowadays there is no more free bytes for being charged with prefix mission. For modern processors the SSE group of new commands was introduced by those combinations of operation codes with prefixes 66h (7.02-06), F2h (7.02-03), F3h (7.02-04), which previously were regarded invalid. New 64-bit processors transfer into prefix class the bytes 40h – 4Fh, which are interpreted by all other x86 platform processors as commands DEC (7.03-20) and INC (7.03-27). Such changes can't be ignored, even if the goal of this book is limited to acquaintance with 16-bit programming under DOS. Recommendations for ensuring compatibility of 16-bit codes with modern processors are given in notes to descriptions of affected machine commands.

7.03 Commands for CPU

7.03-01 AAA – decimal correction of unpacked sum

AAA name stands for Adjust After Addition. The AAA command transforms a binary sum, obtained in AX register after addition of unpacked decimal digits, into a proper unpacked decimal word, containing one decimal digit per byte (for correction of a sum in packed decimal format see 7.03-18).

The AAA command checks whether a binary sum in AX violates decimal overflow condition. Violation is expressed in that either flag AF is set into AC state, or a number in four least significant bits of AL (junior nimble) exceeds 9. If decimal overflow hasn't happened, then AAA command does nothing but clears CF flag (resets it into NC state). Otherwise AAA command adds AL =(AL + 6), AH =(AH + 1), and sets flags AF and CF into states AC and CY respectively. In any case four most significant bits in AL (senior nimble) are cleared to zero. Flags OF, SF, ZF and PF acquire indefinite state.

Code	Example
37	AAA

7.03-02 AAD – decimal preparation to division

The AAD command (AAD = Adjust AX for Division) implies, that AX register contains an unpacked decimal word, i.e. one decimal digit per byte. This decimal word is transformed by AAD command into binary form, so that it may be subjected to binary division (7.03-21).

The AAD command calculates AL = AL+(10•AH), and then clears AH to zero. Flags SF, ZF, PF are set according to the result. Flags OF, AF, CF acquire indefinite state.

Code	Example
D5 0A	AAD

Note 1: machine codes "D5 (1-F)(0-9,B-F)" are improperly unassembled by DEBUG.EXE as "AAD ff" command.

Note 2: numbers in packed decimal format can't be processed by AAD command, these must be unpacked beforehand.

7.03-03 AAM – decimal correction of a product

AAM stands for Adjustment After Multiplication. It is implied, that AX register contains a binary product of two bytes, each representing an unpacked decimal digit and having four most significant bits (senior nimble) zeroed. The AAM command divides the product in AX by 10, writes quotient into AH, and the remainder – into AL, so that result is a proper unpacked decimal word, containing one decimal digit per byte. Flags SF, ZF, PF are set according to the result. Flags OF, AF, CF acquire indefinite state.

In a similar way the AAM command is able to transform any binary number (up to 63h) into an unpacked decimal word.

Code	Example
D4 0A	AAM

Note 1: machine codes "D4 (1-F)(0-9,B-F)" are improperly unassembled by DEBUG.EXE as "AAM ff" command.

Note 2: a product of packed decimal bytes can't be corrected by AAM command. Packed decimal numbers must be unpacked before multiplication.

7.03-04 AAS – decimal correction of unpacked remainder

The AAS command (AAS = Adjust After Subtraction) transforms a binary remainder, obtained in AX register after subtraction of unpacked decimal digits, into a proper unpacked decimal word, containing one decimal digit per byte (for correction of a remainder in packed decimal format see 7.03-19).

The AAS command checks whether a binary remainder in AX violates decimal overflow condition. Violation is expressed in that either the AF flag is set into AC state, or a number in four least significant bits of AL (junior nimble) exceeds 9. If decimal overflow hasn't happened, then AAS command does nothing but clears CF flag (resets it into NC state). Otherwise AAS command subtracts AL =(AL – 6), AH = AH – 1, and sets flags AF and CF into states AC and CY respectively. In any case four most significant bits in AL (senior nimble) are cleared to zero. Flags OF, SF, ZF and PF acquire indefinite state.

Code	Example
3F	AAS

7.03-05 ADC – binary addition with carry

The ADC command performs addition of specified integers, taking into account a carry in the least significant bit. Carry reflects the result of previous operation, and since then it must be preserved, being represented by a state of CF flag. After addition flags OF, SF, ZF, AF, PF, CF acquire new states according to the sum, which replaces the first operand of ADC command.

ADC is a binary operation, but there are two exceptions. If the first operand is in AX register, then ADC command may be applied to unpacked decimal numbers, and the obtained binary sum in AX should be transformed into proper unpacked decimal number by AAA command (7.03-01). If the first operand is in AL register, then ADC command may be applied to packed decimal bytes, and the obtained binary sum in AL should be transformed into proper packed decimal byte by DAA command (7.03-18).

First byte	Second byte	Data bytes	Examples
10	(0-B)(0-F)	0-2	`ADC [bp+si+ffff],bl`
10	(C-F)(0-F)		`ADC bl,bl`
11	(0-B)(0-F)	0-2	`ADC [bp+si+ffff],bx`
11	(C-F)(0-F)		`ADC bx,bx`
12	(0-B)(0-F)	0-2	`ADC bl,[bp+si+ffff]`
13	(0-B)(0-F)	0-2	`ADC bx,[bp+si+ffff]`
14		1	`ADC AL,ff`

Continuation of table 7.03-05

15		2	`ADC AX,ffff`
80	(1,5,9)(0-7)	1-3	`ADC byte ptr [bp+si+ffff],ff`
80	D(1-7)	1	`ADC bl,ff`
81	(1,5,9)(0-7)	2-4	`ADC word ptr [bp+si+ffff],ffff`
81	D(1-7)	2	`ADC bx,ffff`
83	(1,5,9)(0-7)	1-3	`ADC word ptr [bp+si+ffff],±7f`
83	D(1-7)	1	`ADC bx,±7f`

Note 1: machine codes "1(2,3) (C-F)(0-F)" and "82 (1,5,9,D)(0-7)" are also unassembled by DEBUG.EXE as ADC command.

7.03-06 ADD – binary addition

The ADD command performs addition of specified integers, ignoring carry in the least significant bit, represented by a state of CF flag. After addition flags OF, SF, ZF, AF, PF, CF acquire new states according to the sum, which replaces the first operand of ADD command.

ADD is a binary operation, but there are two exceptions. If the first operand is in AX register, then ADD command may be applied to unpacked decimal numbers, and the obtained binary sum in AX should be transformed into a proper unpacked decimal number by AAA command (7.03-01). If the first operand is in AL register, then ADD command may be applied to packed decimal bytes, and the obtained binary sum in AL should be transformed into proper packed decimal byte by DAA command (7.03-18).

First byte	Second byte	Data bytes	Examples
00	(0-B)(0-F)	0-2	`ADD [bp+si+ffff],bl`
00	(C-F)(0-F)		`ADD bl,bl`
01	(0-B)(0-F)	0-2	`ADD [bp+si+ffff],bx`
01	(C-F)(0-F)		`ADD bx,bx`
02	(0-B)(0-F)	0-2	`ADD bl,[bp+si+ffff]`
03	(0-B)(0-F)	0-2	`ADD bx,[bp+si+ffff]`
04		1	`ADD AL,ff`
05		2	`ADD AX,ffff`
80	(0,4,8)(0-7)	1-3	`ADD byte ptr [bp+si+ffff],ff`
80	C(1-7)	1	`ADD bl,ff`
81	(0,4,8)(0-7)	2-4	`ADD word ptr [bp+si+ffff],ffff`
81	C(1-7)	2	`ADD bx,ffff`
83	(0,4,8)(0-7)	1-3	`ADD word ptr [bp+si+ffff],±7f`
83	C(1-7)	1	`ADD bx,±7f`

Note 1: machine codes "0(2,3) (C-F)(0-F)" and "82 (0,4,8,C)(0-7)" are also unassembled by DEBUG.EXE as ADD command.

7.03-07 AND – logical "AND" operation

he AND command analyses pairs of corresponding bits in two operands. If FALSE (zero) state is found in at least one bit in a pair, then corresponding bit of the result is cleared to FALSE (zero). If both bits in a pair are in TRUE state, then corresponding bit of the result is set to TRUE state too. Result replaces the first operand. Flags SF, ZF, PF acquire new states according to the result. Flags CF and OF are cleared to states NC (No Carry) and NV (No oVerflow) respectively.

First byte	Second byte	Data bytes	Examples
20	(0-B)(0-F)	0-2	`AND [bp+si+ffff],bl`
20	(C-F)(0-F)		`AND bl,bl`
21	(0-B)(0-F)	0-2	`AND [bp+si+ffff],bx`
21	(C-F)(0-F)		`AND bx,bx`
22	(0-B)(0-F)	0-2	`AND bl,[bp+si+ffff]`
23	(0-B)(0-F)	0-2	`AND bx,[bp+si+ffff]`
24		1	`AND AL,ff`
25		2	`AND AX,ffff`
80	(2,6,A)(0-7)	1-3	`AND byte ptr [bp+si+ffff],ff`
80	E(1-7)	1	`AND bl,ff`
81	(2,6,A)(0-7)	2-4	`AND word ptr [bp+si+ffff],ffff`
81	E(1-7)	2	`AND bx,ffff`
83	(2,6,A)(0-7)	1-3	`AND word ptr [bp+si+ffff],±7f`
83	E(1-7)	1	`AND bx,±7f`

Note 1: machine codes "2(2-3) (C-F)(0-F)" and "82 (2,6,A,E)(0-7)" are also unassembled by DEBUG.EXE as AND command.

7.03-08 CALL – call for a subroutine

The CALL command saves return address in stack and then performs a jump to subroutine according to specified target address. States of flags are not altered by CALL command.

Several forms of CALL command should be distinguished. A call to subroutine outside code segment of caller program is a CALL FAR, it operates with full 4-byte target address (segment : offset). A call to subroutine within code segment of caller program is a "near" CALL, it doesn't change current code segment and operates with 2-byte target offset only.

There are two different forms of "near" CALL command with machine codes FFh and E8h.

The "near" CALL command with machine code FFh pushes current contents of IP (Instruction Pointer) register into stack and then overwrites IP register with target offset, read either from memory or from a general-purpose register.

The "near" CALL command with machine code E8h, followed by a data word, acts otherwise: after saving IP's contents in stack, it adds this data word to offset in IP. Having found explicit target offset in command line after the CALL command, DEBUG.EXE automatically calculates difference between the given target offset and offset of the next machine command, which is currently present in IP register. This difference constitutes just that data word, which is written after E8h byte into assembled executable code.

Since both forms of "near" CALL command execute jumps inside current code segment, return back to the caller program from a subroutine, called with a "near" CALL command, must be performed by RET command (7.03-73), which restores from stack the contents of IP register only.

The CALL FAR command pushes into stack contents of both CS (Code Segment) and IP registers. Double-word operand of CALL FAR command replaces former contents in both CS and IP registers. Thus a jump to other segment is performed. Therefore a return back to the caller program from a subroutine, called with CALL FAR command, must be performed by RETF command (7.03-74), which restores from stack the contents of both CS and IP registers.

First byte	Second byte	Data bytes	Examples	Comments
9A		4	`CALL FAR ffff:ffff`	note *1
E8		2	`CALL ffff`	
FF	(1,5,9)(0-7)	0-2	`CALL [bp+si+ffff]`	note *2
FF	(1,5,9)(8-F)	0-2	`CALL FAR [bp+si+ffff]`	note *2
FF	D(0-7)		`CALL bx`	note *3

Note 1: in the shown example the first number is target segment address, the second number – target offset. Specification of marker FAR in this line is allowed, but isn't necessary: in any case a call FAR is performed.

Note 2: when target address is read from memory, operation depends on presence (or absence) of marker FAR. If it is present, a four-byte target address is read, and CALL FAR is performed. If marker FAR is not specified, then a 2-byte target offset is read, and a "near" call is performed.

Note 3: if operand of CALL command is in a register, then target offset must be written into this register beforehand. An appeal of CALL command to a 16-bit register always causes a "near" call.

Note 4: machine code "FF D(8-F)" is unassembled by DEBUG.EXE as "CALL far bx".

Note 5: repetition prefixes F2h (7.02-03), F3h (7.02-04) can't be applied to the CALL command.

7.03-09 CBW – byte to word conversion

The CBW command (CBW = Convert Byte to Word) converts a signed byte in AL register into a signed word (2 bytes) in AX register by filling the AH part of AX with sign bit of original signed byte. States of flags are not altered by CBW command.

Code	Example
98	CBW

7.03-10 CLC – carry flag reset

The CLC command clears carry flag CF to the default "NC" (No Carry) state, which is often referred to as CF=0.

Code	Example
F8	CLC

7.03-11 CLD – direction flag reset

The CLD command (CLD = CLear Direction) resets direction flag DF into it's default state "UP". This causes ascending direction of offset count in index registers (DI and/or SI) during execution of string operations (CMPSB, LODSB, MOVSB, SCASB, STOSB, etc.).

Code	Example
FC	CLD

7.03-12 CLI – interrupt flag reset

The CLI command (CLI = Clear Interrupt Flag) resets interrupt flag IF to the "DI" (= Disable Interrupts) state. The CLI command forces CPU to ignore external interrupts, except the non-maskable interrupt INT 02 (8.01-03).

Code	Example
FA	CLI

Note 1: programmable interrupts are performed by INT command (7.03-28) in any case, regardless of the IF flag state.
Note 2: the CLI command wouldn't be executed, if privilege level of the current program is lower than privilege level for I/O operations, defined by bits 0Ch and 0Dh in flags register (A.11-4).

7.03-13 CMC – reversion of carry flag state

The CMC command (CMC = CompleMentary Carry) changes any current state of carry flag CF to the reverse state: NC (No Carry) to CY (CarrY) or vice versa.

Code	Example
F5	CMC

7.03-14 CMP – comparison of operands

The CMP command sets flags OF, SF, ZF, AF, PF, CF according to difference between the first operand (the minuend) and the second (the subtrahend). The difference itself is not saved. Both operands are preserved intact.

Interpretation of flag's states, left by CMP command, depends on whether the operands were signed or unsigned numbers. Conditional jump commands JA, JB, JBE, JNB should be used after comparison of unsigned numbers. Other conditional jump commands JG, JGE, JL, JLE should be used after comparison of signed numbers. Full names of all conditional jump and loop commands reflect status relation of the first (left) operand of CMP command to the second (right) operand. For example, JA = "jump if above" means that the left operand of CMP command must be above, or greater than the right operand.

First byte	Second byte	Data bytes	Examples
38	(0-B)(0-F)	0-2	`CMP [bp+si+ffff],bl`
38	(C-F)(0-F)		`CMP bl,bl`
39	(0-B)(0-F)	0-2	`CMP [bp+si+ffff],bx`
39	(C-F)(0-F)		`CMP bx,bx`
3A	(0-B)(0-F)	0-2	`CMP bl,[bp+si+ffff]`
3B	(0-B)(0-F)	0-2	`CMP bx,[bp+si+ffff]`
3C		1	`CMP AL,ff`
3D		2	`CMP AX,ffff`
80	(3,7,B)(8-F)	1-3	`CMP byte ptr [bp+si+ffff],ff`
80	F(9-F)	1	`CMP bl,ff`
81	(3,7,B)(8-F)	2-4	`CMP word ptr [bp+si+ffff],ffff`

Continuation of table 7.03-14

81	F(9-F)	2	`CMP bx,ffff`
83	(3,7,B)(8-F)	1-3	`CMP word ptr [bp+si+ffff],±7f`
83	F(9-F)	1	`CMP bx,±7f`

Note 1: machine codes "3(A,B) (C-F)(0-F)" and "82 (3,7,B,F)(8-F)" are also unassembled by DEBUG.EXE as CMP command.

7.03-15 CMPSB – serial comparison of byte pairs

Though the name CMPSB stands for "CoMPare Strings of Bytes", the CMPSB command in fact compares one pair of bytes. Addresses of bytes to compare must be loaded beforehand into DS:SI and ES:DI pairs of registers. If bytes are equal, CF (carry flag) is cleared to NC (No Carry) state, ZF (zero flag) is set to ZR state. If bytes are not equal, CF flag is set to CY state, ZF flag is cleared to NZ (No Zero) state. Flags OF, SF, AF, PF acquire the states corresponding to the difference between compared bytes, but this difference itself is not saved.

After comparison both offsets – in SI (source index) register and in DI (destination index) register – are incremented by 1 or decremented by 1: it depends on the state ("UP" or "DN") of direction flag DF. The state of DF flag can be altered by CLD (7.03-11) and STD (7.03-85) commands. Automatic change of index register(s) contents prepares conditions for comparison of the next bytes pair.

The CMPSB command is often preceded by repetition prefixes F2h (7.02-03) or F3h (7.02-04), which enable to execute it cyclically and thus compare strings of bytes. The CMPSB command also may be preceded by a segment override prefix (2Eh or 26h or 36h, see 7.02-01); it enables to refer to other segment register instead of segment register DS for one of compared bytes. For the other compared byte the default segment register ES can't be overridden by prefix.

Code	Example
A6	CMPSB

Note 1: when CMPSB command is preceded by repetition prefixes F2h or F3h, the order of operations within the cycle includes assignment of flags states, then incrementation (or decrementation) of index register's contents, and after that cycle termination condition check. Therefore, the offsets in index registers at the moment of cycle termination are pointing not to those bytes, which have caused cycle termination, but rather to the next pair of bytes.

7.03-16 CMPSW – serial comparison of word pairs

The CMPSW command (CMPSW = CoMPare Strings of Words) compares a pair of words and then increments (or decrements) contents of SI and DI index registers by 2, thus preparing addresses to comparison of the next words pair. The operand size override prefix 66h (7.02-06) forces CMPSW command to compare pairs of four-byte operands (of DWORD type) and to increment (or decrement) contents of index registers by 4. All other peculiarities of CMPSW command execution are the same as those for CMPSB command (7.03-15).

Code	Example
A7	CMPSW

7.03-17 CWD – word to double word conversion.

The CWD command (CWD = Convert Word into Double word) converts a signed word in AX register into a four-byte signed number (of DWORD type). The DX register is dedicated for the two most significant bytes of dword operand. Conversion is performed by filling the DX register with the sign bit of original signed word. States of flags are not altered by CWD command.

Code	Example
99	CWD

7.03-18 DAA – decimal correction of packed sum

The DAA command (DAA = Decimal Adjustment after Addition) transforms a binary sum of packed decimal bytes in AL register into a proper packed decimal byte, representing 2 decimal digits of the sum (for decimal correction of unpacked sum see 7.03-01).

Binary sum of packed decimal bytes may violate decimal overflow condition in both junior and senior 4-bit parts (nimbles) of AL register. The junior part is checked first: if the value there exceeds 9 or flag AF is set into AC state, then DAA command adds AL = (AL + 6). After that similar check is applied to senior 4-bit part (nimble) of AL register: if the value there exceeds 9Fh or CF flag is set into CY state, then DAA command adds AL = (AL + 60h). Flags AF, CF, SF, ZF, PF acquire new states according to the result. The OF flag is left in indefinite state.

Code	Example
27	DAA

7.03-19 DAS – decimal correction of packed remainder

The DAS command (DAS = Decimal Adjustment after Subtraction) transforms a binary difference of packed decimal bytes in AL register into a proper packed decimal byte, representing 2 decimal digits of the remainder (for decimal correction of unpacked difference see 7.03-04).

Binary difference of packed decimal bytes may violate decimal overflow condition in both junior and senior 4-bit parts (nimbles) of AL register. The junior part is checked first: if the value there exceeds 9 or flag AF is set into AC state, then DAS command subtracts AL = (AL – 6). After that a similar check is applied to senior 4-bit part (nimble) of AL register: if the value there exceeds 9Fh or CF flag is set into CY state, then DAS command subtracts AL = (AL – 60h). Flags AF, CF, SF, ZF, PF acquire new states according to the result. The OF flag is left in indefinite state.

Code	Example
2F	DAS

7.03-20 DEC – a unity decrement

The DEC command decrements its operand by 1. Flags OF, SF, ZF, AF, PF acquire new states according to the result. The CF flag preserves its former state.

First byte	Second byte	Data bytes	Examples
4(8-F)			`DEC bx`
FE	(0,4,8)(8-E)	0-2	`DEC byte ptr [bp+si+ffff]`
FE	C(8-F)		`DEC bl`
FF	(0,4,8)(8-E)	0-2	`DEC word ptr [bp+si+ffff]`

Note 1: bytes 48h – 4Fh can be interpreted by 64-bit processors as prefixes. Therefore 2-byte codes "FF C(8-F)" should be given preference over 1-byte codes 4(8-F). Codes "FF C(8-F)" are interpreted as "DEC bx" command by all x86 platform processors and are properly unassembled by DEBUG.EXE, but during assembling these codes should be presented to DEBUG.EXE as data by DB instruction (7.01-01).

7.03-21 DIV – division of unsigned integers

The DIV command performs division of unsigned binary integers (for division of signed integers see 7.03-24). Explicit operand is the divisor. If divisor is a byte, dividend is implied to exist in AX register, quotient is left in AL, and the remainder is placed in AH. If divisor is a word, dividend is implied to exist in DX register (most significant 2 bytes) and in AX register (less significant 2 bytes), quotient is left in AX, the remainder is placed in DX. Flags OF, SF, ZF, AF, PF, CF acquire indefinite state.

Though DIV is a binary operation, unpacked decimal words may be subjected to binary division, if they are transformed in advance into acceptable quasi-binary form by AAD command (7.03-02).

First byte	Second byte	Data bytes	Examples
F6	(3,7,B)(0-7)	0-2	`DIV byte ptr [bp+si+ffff]`
F6	F(0-7)		`DIV bl`
F7	(3,7,B)(0-7)	0-2	`DIV word ptr [bp+si+ffff]`
F7	F(0-7)		`DIV bx`

Note 1: if division operation causes overflow in quotient register, CPU automatically generates an exception: a call for INT 00 handler (8.01-01). Outcome depends on that handler.

7.03-22 ESC – code transfer to asynchronous coprocessor

Originally the ESC (= ESCape) command has been used to send data and commands from CPU to external asynchronous coprocessor. Each ESC command has been preceded by WAIT prefix (7.02-05), forcing CPU to wait for arrival of readiness confirmation signal from coprocessor to CPU's pin "BUSY". Later arithmetic coprocessor's commands have got their specific names (7.04), but the rest machine codes, starting with bytes D8h - DFh, are still unassembled by DEBUG.EXE as ESC command:

D9 (0,4,8)(8-F), D9 D(1-7), DA (C-F)(0-F), DB (0,4,8,C,D,F)(8-F),
DB (2,3,6,7,A-D,F)(0-7), DB E(4 – F), DD (0,2,6)(8-F),
DD (E,F)(0-F), DE D(8,A-F), DF (0,4,8)(8-F), DF (E,F)(0-F).

Some of the mentioned codes are assigned yet to new commands of modern arithmetic coprocessors. As all coprocessor's commands, starting with bytes D8h – DFh, the ESC command is executed by CPU, if in control register CR0 (A.11-4) it's bit 02h ("Coprocessor emulation") is cleared to zero. But if bit 02h is set, then CPU responds to each such command with a call to INT 07 handler (8.01-08). It is implied, that a special INT 07 handler should be loaded, which is able to emulate functions of arithmetic coprocessor or other asynchronous device(s).

Potentially the ESC command can be used to send data and commands to external asynchronous device(s), but for modern processors this opportunity is not documented. The first operand of ESC command is a hexadecimal number, the second is read from the specified source. Interpretation of both operands is a prerogative of each particular target device. States of flags are not altered by ESC command.

First byte	Second byte	Examples
DA	C(0-7)	`ESC 10,bl`
DA	C(8-F)	`ESC 11,bl`
DA	D(0-7)	`ESC 12,bl`
DA	D(8-F)	`ESC 13,bl`
DA	E(0-7)	`ESC 14,bl`
DA	E(8-F)	`ESC 15,bl`
DA	F(0-7)	`ESC 16,bl`
DA	F(8-F)	`ESC 17,bl`

7.03-23 HLT – set CPU to a standstill

The HLT command (= HaLT) forces processor to stop. Being stopped, processor preserves contents of CS:IP registers and states of flags, so that an opportunity of proper activation is secured. Processor can be turned back to normal functioning either by a reboot or by external interrupt signal, received either via NMI pin (8.01-03) or via interrupt controller (8.01-09).

Code	Example
F4	HLT

Note 1: the HLT command can be used in programs, which are to be executed at the highest privilege level. Beyond the highest privilege level the HLT command is ignored.

Note 2: a way to determine that particular external interrupt, which has turned CPU out of a standstill state, is shown in article 8.01-09.

7.03-24 IDIV – division of signed integers

The IDIV (= Integer DIVision) command performs division of signed binary integers (for division of unsigned integers see 7.03-21). Explicit operand is the divisor. If divisor is a byte, dividend is implied to exist in AX register, quotient is left in AL, the remainder is placed in AH. If divisor is a word, dividend is implied to exist in DX register (more significant 2 bytes) and in AX register (less significant 2 bytes), quotient is left in AX, the

remainder is placed in DX. Sign of the remainder is always the same as that of dividend. Flags OF, SF, ZF, AF, PF, CF acquire indefinite state.

Though IDIV is a binary operation, unpacked decimal words may be subjected to binary division, if they are transformed in advance into acceptable quasi-binary form by AAD command (7.03-02).

First byte	Second byte	Data bytes	Examples
F6	(3,7,B)(8-F)	0-2	`IDIV byte ptr [bp+si+ffff]`
F6	F(8-F)		`IDIV bl`
F7	(3,7,B)(8-F)	0-2	`IDIV word ptr [bp+si+ffff]`
F7	F(8-F)		`IDIV bx`

Note 1: if division operation causes overflow in quotient register, CPU automatically generates an exception: a call for INT 00 handler (8.01-01). Outcome depends on that handler.

7.03-25 IMUL – multiplication of signed integers

The IMUL (Integer MULtiplcation) command multiplies signed integers (for unsigned integers see 7.03-61). Explicit operand of IMUL command represents multiplier. If this operand is a byte, then the other operand is implied to exist in AL register; after multiplication the product is left in AX register. If explicit operand is a word, then the other operand must exist in AX register; after multiplication the less significant 2 bytes of product are left in AX register, the most significant 2 bytes of product – in DX register.

If most significant part of product in AH or in DX register represents non-zero values, then flags OF and CF are set by IMUL command to OV and CY states correspondingly. On the contrary, clear states NV and NC of these flags indicate, that most significant part of product is filled with sign bits only. Flags SF, ZF, AF, PF acquire indefinite state.

The IMUL command can be applied to binary integers and to unpacked decimal numbers. Packed decimal operands must be unpacked beforehand. Product of unpacked decimal numbers needs to be transformed into unpacked decimal format by AAM command (7.03-03).

First byte	Second byte	Data bytes	Examples
F6	(2,6,A)(8-F)	0-2	`IMUL byte ptr [bp+si+ffff]`
F6	E(8-F)		`IMUL bl`
F7	(2,6,A)(8-F)	0-2	`IMUL word ptr [bp+si+ffff]`
F7	E(8-F)		`IMUL bx`

Note 1: other forms of IMUL command with 2 explicit operands (codes 69h and 6Bh) are not supported by DEBUG.EXE.

7.03-26 IN – data input from port

While performing the IN command, CPU generates a signal, which switches CPU's buses from memory to I/O ports and enables asynchronous data transfer. First operand of IN command specifies the register, where the received data should be written. This register must be chosen according to format of received data: a byte register AL, if a byte is to be received, or a double-byte register AX, if a word is to be received. The second operand of IN command defines port address. The latter may be specified either explicitly as a double-digit hexadecimal number or indirectly – as contents of DX register. States of CPU's flags are not altered by IN command.

First byte	Second byte	Data bytes	Examples
E4		1	`IN AL,ff`
E5		1	`IN AX,ff`
EC			`IN AL,DX`
ED			`IN AX,DX`

Note 1: selected port addresses are shown in appendix A.14-1. Direct forms of IN command don't allow port addresses above FFh. Indirect addressing via DX register is not subjected to this restriction.

Note 2: the IN command wouldn't be executed, if privilege level of the current program is lower than privilege level for I/O operations, defined by bits 0Ch and 0Dh in flags register (A.11-4).

7.03-27 INC – a unity increment

The INC command increments its operand by 1. Flags OF, SF, ZF, AF, PF acquire new states according to the result. The CF flag preserves its former state.

First byte	Second byte	Data bytes	Examples
4(0-7)			`INC bx`
FE	(0,4,8)(0-7)	0-2	`INC byte ptr [bp+si+ffff]`
FE	C(0-7)		`INC bl`
FF	(0,4,8)(0-7)	0-2	`INC word ptr [bp+si+ffff]`

Note 1: bytes 40h – 47h can be interpreted by 64-bit processors as prefixes. Therefore 2-byte codes "FF C(0-7)" should be given preference over 1-byte codes 4(0-7).

Codes "FF C(0-7)" are interpreted as "INC bx" command by all x86 platform processors and are properly unassembled by DEBUG.EXE, but during assembling these codes should be presented to DEBUG.EXE as data by DB instruction (7.01-01).

7.03-28 INT – a call for interrupt handler

The INT (= Interrupt) command transfers control to that interrupt handler, which number is defined by operand of INT command. But before control is transferred, the INT command prepares conditions for further return back to the current program after termination of interrupt handler's job. Therefore the following actions are undertaken by INT command:

– current state of flags register is saved in stack;
– current state of CS register (segment address) is saved in stack;
– offset of the next command is calculated and saved in stack in order to enable further restoration of IP register state;
– the IF flag is cleared to DI state, so that intake of interrupt requests via interrupt controller is blocked;
– queue of prefetched commands in CPU is reset;
– multiplication of interrupt number by 4 gives address of that memory cell, where interrupt handler's address is stored;
– copying of interrupt handler's address (segment and offset) from memory cell into registers CS:IP transfers control to the handler.

The order of data, left in stack by INT command, enables a return back to current program by means of IRET command (7.03-30), which must be the last command, performed by each interrupt handler. Resumed execution of interrupted program will start from that command, which follows the INT command.

Almost each interrupt handler can't perform its mission unless some specific conditions are met or unless some required data are present in CPU's registers or in memory. These conditions and data must be prepared in advance, before execution of INT command. Relevant requirements of selected interrupt handlers are described in chapter 8 of this book.

First byte	Second byte	Data bytes	Examples
CC CD		 1	`INT 3` `INT ff`

Note 1: original state of interrupt flag IF doesn't affect execution of INT command. Flag IF affects only those external interrupt requests, which are received via interrupt controller.

Note 2: a unique feature of INT 3 command (code CCh) is that it doesn't depend on privilege level: at any privilege level it is executed just as in real mode.

Note 3: offset of the next command is stored in stack by commands INT and INTO (7.03-29) only. All other internal interrupts (exceptions) leave in stack the current command's offset.

Note 4: data in stack are available to interrupt handlers. If the state of stack pointer (SP) is saved in BP register just after control transfer, then [BP+00] address points at return offset, [BP+02] address points at return segment, [BP+04] address points at flag's states of interrupted program.

7.03-29 INTO – a call for overflow handler

Immediate response to overflow via INT 00 (8.01-01) sometimes isn't expedient. More flexible and retarded response to overflows can be provided by INTO command (INTO = INTerrupt if Overflow). If OV (= OVerflow) state of OF flag indicates fact of overflow, then INTO command calls for interrupt INT 04 handler (8.01-05), which must be designed for handling overflow errors. A call for INT 04 handler by INTO command includes all those precautions, which are undertaken by INT command (7.03-28).

Code	Example
CE	INTO

Note 1: the default INT 04 handler does nothing but returns control to the caller program. In order to obtain a desirable response to overflows the user has to prepare another INT 04 handler instead of the default one. New handler becomes active since its address is written into interrupt table (8.02-18).

7.03-30 IRET – return from interrupt handler

The IRET (= Interrupt RETurn) command restores from stack all the data, providing a return to execution of the caller program: its segment address in CS register, the former states of flags, and prepared offset of the next command in IP register. The IRET command must be the last executed by every interrupt handler.

Code	Example
CF	IRET

Note 1: restoration of flag's states by IRET command is not subjected to those restrictions, which are imposed on POPF command (7.03-68). Thus IRET command gives a chance to bypass these restrictions.

Note 2: the IRET commands resets a queue of prefetched commands in CPU. This is done because commands decoding rules for interrupt handler may differ from those accepted for the caller program.

7.03-31 JA – Jump if above

The JA command adds its data byte to contents of IP register, if both flags CF and ZF are cleared to states NC (No Carry) and NZ (No Zero) correspondingly. As far as data byte represents difference between target and current offsets, its addition causes a "short" transition (jump) to appointed target offset within ±7Fh vicinity of the nearest next command.

Most often the JA command is used after operations with unsigned integers, in particular, after commands CMP, SBB, SUB (after operations with signed integers the same condition "above" is checked by JG command, 7.03-35).

DEBUG.EXE accepts one more name for JA command: JNBE – "jump if not below or equal", but code 77h is always unassembled as JA.

First byte	Second byte	Data bytes	Example ("aaaa" – target offset)
77		1	`JA aaaa`

7.03-32 JB – Jump if below

The JB command adds its data byte to contents of IP register, if flag CF is set to CY (CarrY) state. As far as data byte represents difference between target and current offsets, its addition causes a "short" transition (jump) to appointed target offset within ±7Fh vicinity of the nearest next command.

The JB command is used for performing jumps after various failures, marked by setting CF flag into CY state. JB command is also used after operations with unsigned integers, in particular, after commands CMP, SBB, SUB (after operations with signed integers the same condition "below" is checked by JL command, 7.03-37).

DEBUG.EXE accepts two more names for JB command: JNAE –"jump if not above or equal" and JC – "jump if carry", but code 72h is always unassembled as JB.

First byte	Second byte	Data bytes	Example ("aaaa" – target offset)
72		1	`JB aaaa`

7.03-33 JBE – Jump if below or equal

The JBE command adds its data byte to contents of IP register, if either flag CF is set to CY (CarrY) state or flag ZF is set into ZR (ZeRo) state. As far as data byte represents difference between target and current offsets, its addition causes a "short" transition (jump) to appointed target offset within ±7Fh vicinity of the nearest next command.

The JBE command is used after operations with unsigned integers, in particular, after commands CMP, SBB, SUB (after operations with signed integers the same condition "below or equal" is checked by JLE command, 7.03-38).

DEBUG.EXE accepts one more name of this command: JNA – "jump if not above", but code 76h is always unassembled as JBE.

First byte	Second byte	Data bytes	Example ("aaaa" – target offset)
76		1	`JBE aaaa`

7.03-34 JCXZ – Jump if CX is Zero

The JCXZ command adds its data byte to contents of IP register, if value in CX register is zero. As far as data byte represents difference between target and current offsets, its addition causes a "short" transition (jump) to appointed target offset within ±7Fh vicinity of the nearest next command.

As far as CX register is often used as iterations counter, the JCXZ command enables to bypass loops, if necessary condition is not met before entering the loop.

First byte	Second byte	Data bytes	Example ("aaaa" – target offset)
E3		1	`JCXZ aaaa`

7.03-35 JG – Jump if Greater

The JG command adds its data byte to contents of IP register, if flag ZF is cleared to NZ (No Zero) state and at the same time flags SF and OF have the same state, i.e. either both are cleared or both are set. As far as data byte represents difference between target and current offsets, its addition causes a "short" transition (jump) to appointed target offset within ±7Fh vicinity of the nearest next command.

The JG command is used after operations with signed integers, in particular, after commands CMP, SBB, SUB (after operations with unsigned integers the same condition "greater" is checked by JA command, 7.03-31).

DEBUG.EXE accepts one more name for JG command: JNLE – "jump if not lower or equal", but code 7Fh is always unassembled as JG.

First byte	Second byte	Data bytes	Example ("aaaa" – target offset)
7F		1	`JG aaaa`

7.03-36 JGE – Jump if Greater or Equal

The JGE command adds its data byte to contents of IP register, if flags SF and OF are in the same state, i.e. either both are cleared or both are set. As far as data byte represents difference between target and current offsets, its addition causes a "short" transition (jump) to appointed target offset within ±7Fh vicinity of the nearest next command.

The JGE command is used after operations with signed integers, in particular, after commands CMP, SBB, SUB (after operations with unsigned integers the same condition "greater or equal" is checked by JNB command, 7.03-40).

DEBUG.EXE accepts one more name of this command: JNL – "jump if not lower", but code 7Dh is always unassembled as JGE.

First byte	Second byte	Data bytes	Example ("aaaa" – target offset)
7D		1	`JGE aaaa`

7.03-37 JL – Jump if Lower

The JL command adds its data byte to contents of IP register, if flags SF and OF are in different states, i.e. when any of them is cleared, while the other is set. As far as data byte represents difference between target and current offsets, its addition causes a "short" transition (jump) to appointed target offset within ±7Fh vicinity of the nearest next command.

The JL command is used after operations with signed integers, in particular, after commands CMP, SBB, SUB (after operations with unsigned integers the same condition "lower" is checked by JB command, 7.03-32).

DEBUG.EXE accepts one more name of this command: JNGE – "jump if not greater or equal", but code 7Ch is always unassembled as JL.

First byte	Second byte	Data bytes	Example ("aaaa" – target offset)
7C		1	`JL aaaa`

7.03-38 JLE – Jump if Lower or Equal

The JLE command adds its data byte to contents of IP register, if either flag ZF is set to ZR (ZeRo) state or flags SF and OF are in different states, i.e. when any of them is cleared, while the other is set. As far as data byte represents difference between target and current offsets, its addition causes a "short" transition (jump) to appointed target offset within ±7Fh vicinity of the nearest next command.

The JLE command is used after operations with signed integers, in particular, after commands CMP, SBB, SUB (after operations with unsigned integers the same condition "lower or equal" is checked by JBE command, 7.03-33).

DEBUG.EXE accepts one more name for JLE command: JNG – "jump if not greater", but code 7Eh is always unassembled as JLE.

First byte	Second byte	Data bytes	Example ("aaaa" – target offset)
7E		1	`JLE aaaa`

7.03-39 JMP – unconditional jump

The JMP command performs a transition (jump) to execution of another machine command by changing former contents either in IP (instruction pointer) register only or simultaneously in both CS (code segment) and IP registers. If JMP command is given a double-word operand, it is executed as "JMP FAR": the first word replaces former segment address in CS register, the second word replaces former offset in IP register. The JMP command with one-word operand replaces offset in IP register only, thus performing a "near" jump within the same segment. The JMP command with a single byte of data performs a "short" jump otherwise: it adds its data byte to current offset in IP register.

While CPU operates in real mode, the JMP command doesn't affect flags.

First byte	Second byte	Data bytes	Examples	Comments
E9		2	`JMP ffff`	note *1
EA		4	`JMP ffff:ffff`	note *2
EB		1	`JMP aaaa`	note *1
FF	(2,6,A)(0-7)	0-2	`JMP [bp+si+ffff]`	note *3
FF	(2,6,A)(8-F)	0-2	`JMP FAR [bp+si+ffff]`	note *3
FF	E(0-7)		`JMP bx`	note *4

Note 1: having been given a target offset in an assembler command line, DEBUG.EXE automatically calculates difference between specified target offset and offset of the next command. If this difference doesn't exceed ±7fh, JMP command is translated into machine code EBh ("short" jump), otherwise it is translated into machine code E9h ("near" jump).

Note 2: in the shown example the first number is segment address, the second number – the target offset. Marker FAR in such command lines is allowed, but isn't necessary: a FAR jump will be performed in any case.

Note 3: when JMP command gets target address via indirection, then the type of jump depends on presence of marker FAR: if it is specified, a 4-byte full address will be read from memory, and a far jump will be performed. If marker FAR is absent, then a 2-byte word will be read from memory. This word will be interpreted as target offset, and a "near" jump will be performed.

Note 4: if JMP command appeals to a register, then target offset must be prepared in this register beforehand. An appeal of JMP command to a 16-bit register always causes a "near" jump.

Note 5: almost each switching of CPU from real mode and back is followed by a JMP FAR command, transferring control to the next command in the same code segment. This JMP command is needed not for a jump, but for other purposes. First, it brings status of a word in CS register (segment address or selector) in accordance with CPU's mode. Second, this JMP command resets a queue of prefetched commands in CPU, because these commands were decoded according to the rules of former CPU's mode.

Note 6: codes "FF E(8-F)" are unassembled by DEBUG.EXE as command "JMP far bx".

7.03-40 JNB – Jump if Not Below

The JNB command adds its data byte to contents of IP register, if flag CF is cleared to NC (No CarrY) state. As far as data byte represents difference between target and current offsets, its addition causes a "short" transition (jump) to appointed target offset within ±7Fh vicinity of the nearest next command.

The JNB command is used for performing jumps after successful terminations, marked by clearing CF flag to NC state. JNB command is also used after operations with unsigned integers, in particular, after commands CMP, SBB, SUB (after operations with signed integers the same condition "greater or equal" is checked by JGE command, 7.03-36).

DEBUG.EXE accepts two more names for JNB command: JAE – "jump if above or equal" and JNC – "jump if not carry", but code 73h is always unassembled as JNB.

First byte	Second byte	Data bytes	Example ("aaaa" – target offset)
73		1	`JNB aaaa`

7.03-41 JNO – Jump if No Overflow

The JNO command adds its data byte to contents of IP register, if OF flag is cleared to NV (No oVerflow) state. As far as data byte represents difference between target and current offsets, its addition causes a "short" transition (jump) to appointed target offset within ±7Fh vicinity of the nearest next command.

First byte	Second byte	Data bytes	Example ("aaaa" – target offset)
71		1	`JNO aaaa`

7.03-42 JNS – Jump if No Sign

The JNS command adds its data byte to contents of IP register, if SF flag is cleared to PL state, which indicates positive integer result of previous operation. As far as data byte represents difference between target and current offsets, its addition causes a "short" transition (jump) to appointed target offset within ±7Fh vicinity of the nearest next command.

First byte	Second byte	Data bytes	Example ("aaaa" – target offset)
79		1	`JNS aaaa`

7.03-43 JNZ – Jump if No Zero

he JNZ command adds its data byte to contents of IP register, if ZF flag is cleared to NZ (No Zero) state, which indicates inequality or non-zero result of previous operation. As far as data byte represents difference between target and current offsets, its addition causes a "short" transition (jump) to appointed target offset within ±7Fh vicinity of the nearest next command.

DEBUG.EXE accepts one more name for JNZ command: JNE – "jump if not equal", but code 75h is always unassembled as JNZ.

First byte	Second byte	Data bytes	Example ("aaaa" – target offset)
75		1	`JNZ aaaa`

7.03-44 JO – Jump if Overflow

The JO command adds its data byte to contents of IP register, if OF flag is set to OV (OVerflow) state. As far as data byte represents difference between target and current offsets, its addition causes a "short" transition (jump) to appointed target offset within ±7Fh vicinity of the nearest next command.

First byte	Second byte	Data bytes	Example ("aaaa" – target offset)
70		1	`JO aaaa`

7.03-45 JPE – Jump if Parity Even

The JPE command adds its data byte to contents of IP register, if PF flag is set to PE (Parity Even) state, which indicates an even sum of bits in the least significant byte of previous operation result (other bytes of the result are not taken into account). As far as data byte represents difference between target and current offsets, its addition causes a "short" transition (jump) to appointed target offset within ±7Fh vicinity of the nearest next command.

DEBUG.EXE accepts one more name for JPE command: JP – "jump if parity", but code 7Ah is always unassembled as JPE.

First byte	Second byte	Data bytes	Example ("aaaa" – target offset)
7A		1	`JPE aaaa`

7.03-46 JPO – Jump if Parity Odd

The JPO command adds its data byte to contents of IP register, if PF flag is cleared to PO (Parity Odd) state, which indicates an odd sum of bits in the least significant byte of previous operation result (other bytes of the result are not taken into account). As far as data byte represents difference between target and current offsets, its addition causes a "short" transition (jump) to appointed target offset within ±7Fh vicinity of the nearest next command.

DEBUG.EXE accepts one more name for JPO command: JNP – "jump if not parity", but code 7Bh is always unassembled as JPO.

First byte	Second byte	Data bytes	Example ("aaaa" – target offset)
7B		1	`JPO aaaa`

7.03-47 JS – Jump if Sign

The JS command adds its data byte to contents of IP register, if SF flag is set to NG state, which indicates negative integer result of previous operation. As far as data byte represents difference between target and current offsets, its addition causes a "short" transition (jump) to appointed target offset within ±7Fh vicinity of the nearest next command.

First byte	Second byte	Data bytes	Example ("aaaa" – target offset)
78		1	`JS aaaa`

7.03-48 JZ – Jump if Zero

The JZ command adds its data byte to contents of IP register, if ZF flag is set to ZR (ZeRo) state, which indicates equality or zero result of previous operation. As far as data byte represents difference between target and current offsets, its addition causes a "short" transition (jump) to appointed target offset within ±7Fh vicinity of the nearest next command.

DEBUG.EXE accepts one more name for JZ command: JE – "jump if equal", but code 74h is always unassembled as JZ.

First byte	Second byte	Data bytes	Example ("aaaa" – target offset)
74		1	`JZ aaaa`

7.03-49 LAHF – copying flags into AH register

The LAHF command (LAHF = Load AH with Flags) copies into AH register the states of flags from the lower byte of flags register. Bit 0 in AH corresponds to CF (carry flag), bit 2 – to PF (parity flag), bit 4 – to AF (auxiliary flag), bit 6 – to ZF (zero flag), bit 7 – to SF (sign flag). Bits 5, 3, 1 have no corresponding flags. Bit 1 is always set to binary unity, bits 5 and 3 are always cleared to zero.

Code	Example
9F	LAHF

7.03-50 LDS – Loading of DS register

The LDS command regards its second operand as address of a double word. Bytes 1 and 2 in this double word are interpreted as offset, bytes 3 and 4 – as segment address. LDS command copies this segment address into DS segment register, and offset – into that register, which is specified as the first operand of LDS command. Thus this register together with segment register DS become ready to be referenced as segment: offset pair. States of flags are not altered by LDS command.

First byte	Second byte	Data bytes	Example
C5	(1,5,9)(8-F)	0-2	`LDS bx,[bp+si+ffff]`

Note 1: codes "C5 (C-F)(0-F)" are also unassembled by DEBUG.EXE as LDS command.
Note 2: by default the DS:SI pair of registers represents source address; therefore SI register is the most frequent substitution for "bx" in the shown example of LDS command.
Note 3: both segment in DS register and offset in specified register may be used for addressing and be reassigned in the same operation; for example, command DS: LDS SI,[SI] is valid.

7.03-51 LEA – offset calculation

The LEA command (LEA = Load Effective Address) calculates expression in square brackets, given as the second operand. Result of calculation represents a certain offset. This offset is written into that register, which is specified as the first operand of LEA command. States of flags are not altered by LEA command.

First byte	Second byte	Data bytes	Example
8D	(0-B)(0-F)	0-2	`LEA bx,[bp+si+ffff]`

7.03-52 LES – Loading of ES register

The LES command regards its second operand as address of a double word. Bytes 1 and 2 in this double word are interpreted as offset, bytes 3 and 4 – as segment address. LES command copies this segment address into ES segment register, and offset – into that register, which is specified as the first operand of LES command. Thus this register together with segment register ES become ready to be referenced as segment: offset pair. States of flags are not altered by LES command.

First byte	Second byte	Data bytes	Example
C4	(1,5,9)(8-F)	0-2	`LES bx,[bp+si+ffff]`

Note 1: codes "C4 (C-F)(0-F)" are also unassembled by DEBUG.EXE as LES command.
Note 2: by default the ES:DI pair of registers represents destination address; therefore DI register is the most frequent substitution for "bx" in the shown example of LES command.
Note 3: both segment in ES register and offset in specified register may be used for addressing and be reassigned in the same operation; for example, command ES: LES DI,[DI] is valid.

7.03-53 LODSB – serial copying of bytes

Though the name LODSB stands for "LOaD String of Bytes", the LODSB command in fact copies into AL register a single byte, read out of memory according to address, which is written beforehand into DS:SI pair of registers. After copying offset in SI (source index) register is incremented by 1 or decremented by 1: it depends on state ("UP" or "DN") of direction flag DF. The state of DF flag can be altered by CLD (7.03-11) and STD (7.03-85) commands. Automatic change of SI register contents prepares conditions for copying of the next byte. States of flags are not altered by LODSB command.

The LODSB command may be preceded by a segment override prefix (7.02-01); it enables to refer to other segment register instead of default source segment register DS.

Code	Example
AC	LODSB

7.03-54 LODSW – serial copying of words

The LODSW command (LODSW = LOaD a String of Words) copies into AX register a single word and then increments (or decrements) contents of SI index register by 2, thus preparing offset to copying of the next word. The operand size override prefix 66h (7.02-06) forces LODSW command to copy a four-byte operands (of DWORD type) and to increment (or decrement) contents of SI register by 4. All other peculiarities of LODSW command execution are the same as those for LODSB command (7.03-53).

Code	Example
AD	LODSW

7.03-55 LOOP – arrangement of a cycle

The LOOP command first decrements an integer in CX register by 1, and then checks whether the remainder is zero. Until the remainder is not zero, the LOOP command adds its data byte to current offset in IP register. Thus a "short" jump is performed within ±7fh vicinity of the nearest next command. But when the remainder in CX register becomes zero, then LOOP command does nothing, so that CPU exits the cycle and just proceeds to execution of the nearest following command beyond the cycle's body. States of flags are not altered by LOOP command.

Count of iterations in CX register is based on supposition, that LOOP command follows the cycle's body. In this case cycle's body is executed once before cycle entering condition is checked by LOOP command for the first time. In order to prevent uncontrolled execution the cycle's body should be preceded by JCXZ command (7.03-34). The same result can be obtained by cycles with exported body, but in the latter case the preset integer in CX register must be by a unity greater, than required number of iterations.

First byte	Second byte	Data bytes	Example ("aaaa" – target offset)
E2		1	`LOOP aaaa`

7.03-56 LOOPNZ – cycle with ZF = ZR exit condition

The LOOPNZ command (LOOPNZ = Loop if Not Zero) first decrements an integer in CX register by 1, not affecting flags, and then checks two conditions: whether the remainder in CX register is zero and whether ZF flag is set into ZR (ZeRo) state. Until both conditions are not met, LOOPNZ command adds its data byte to current offset in IP register. Thus a "short" jump is performed within ±7fh vicinity of the nearest next command. But when either of the mentioned conditions is met, then LOOPNZ command does nothing, so that CPU exits the cycle and just proceeds to execution of the nearest following command beyond the cycle's body. Other peculiarities of cycle's arrangement with LOOPNZ command are the same as for LOOP command (7.03-55).

DEBUG.EXE accepts one more name for LOOPNZ command: LOOPNE (= loop, if not equal), but code E0h is always unassembled as LOOPNZ.

First byte	Second byte	Data bytes	Example ("aaaa" – target offset)
E0		1	`LOOPNZ aaaa`

7.03-57 LOOPZ – cycle with ZF = NZ exit condition

The LOOPZ command (LOOPZ = Loop if Zero) first decrements an integer in CX register by 1, not affecting flags, and then checks two conditions: whether the remainder in CX register is zero and whether ZF flag is cleared to NZ (No Zero) state. Until both conditions are not met, LOOPZ command adds its data byte to current offset in IP register. Thus a "short" jump is performed within ±7fh vicinity of the nearest next command. But when either of the mentioned conditions is met, then LOOPZ command does nothing, so that CPU exits the cycle and just proceeds to execution of the nearest following command beyond the cycle's body. Other peculiarities of cycle's arrangement with LOOPZ command are the same as for LOOP command (7.03-55).

DEBUG.EXE accepts one more name for LOOPZ command: LOOPE (loop, if equal), but code E1h is always unassembled as LOOPZ.

First byte	Second byte	Data bytes	Example ("aaaa" – target offset)
E1		1	`LOOPZ aaaa`

7.03-58 MOV – data copying command

The MOV command copies a byte or a word, specified directly or indirectly by the second operand, into register or memory cell, specified by the first operand. Explicit specification of data size to be copied (byte or word) is not needed, when it can be determined by the size of involved register. States of flags are not altered by ordinary forms of MOV command, except forms appealing to control, debugging and test CPU's registers. These forms, shown in note 1 below, may leave flags OF, SF, ZF, AF, PF, CF in indefinite state.

First byte	Second byte	Data bytes	Examples
88	(0-B)(0-5, 7-F)	0-2	`MOV [bp+si+ffff],bl`
88	(C-F)(0-F)		`MOV bl,bl`
89	(0-B)(0-5, 7-F)	0-2	`MOV [bp+si+ffff],bx`
89	(C-F)(0-F)		`MOV bx,bx`
8A	(0-B)(0-5, 7-F)	0-2	`MOV bl,[bp+si+ffff]`
8B	(0-B)(0-5, 7-F)	0-2	`MOV bx,[bp+si+ffff]`
8C	(0,1,4,5,8,9)(0-F)	0-2	`MOV [bp+si+ffff],ss`
8C	(C,D,E)(0-F)		`MOV bx,ss`
8E	(0,1,4,5,8,9)(0-F)	0-2	`MOV ss,[bp+si+ffff]`
8E	(C,D,E)(0-F)		`MOV ss,bx`
A0		2	`MOV AL,[ffff]`

Continuation of table 7.03-58

A1		2	`MOV AX,[ffff]`
A2		2	`MOV [ffff],AL`
A3		2	`MOV [ffff],AX`
B(0-7)		1	`MOV bl,ff`
B(8-F)		2	`MOV bx,ffff`
C6	(0,4,8)(0-7)	1-3	`MOV byte ptr [bp+si+ffff],ff`
C7	(0,4,8)(0-7)	2-4	`MOV word ptr [bp+si+ffff],ffff`

Note 1: DEBUG.EXE doesn't "know" those forms of MOV command, which appeal to debugging and control registers of 32-bit CPUs, but codes of these commands may be entered as data by DB instruction (7.01-01). Codes of these commands are 3 bytes long, commencing with OFh byte. The second byte defines direction of copying:

20h – from control register (CR0, CR2 – CR4)
21h – from debugging register (DR0 – DR3, DR6, DR7)
22h – into control register (CR0, CR2 – CR4)
23h – into debugging register (DR0 – DR3, DR6, DR7)

The third byte in such codes defines particular register:

C0h – CR0 or DR0, for example, 0F 20 C0 = MOV EAX,CR0
C8h – DR1, for example, 0F 23 C8 = MOV DR1,EAX
D0h – CR2 or DR2, for example, 0F 20 D0 = MOV EAX,CR2
D8h – CR3 or DR3, for example, 0F 20 D8 = MOV EAX,CR3
E0h – CR4, for example, 0F 22 E0 = MOV CR4,EAX
F0h – DR6, for example, 0F 21 F0 = MOV EAX,DR6
F8h – DR7, for example, 0F 23 F8 = MOV DR7,EAX

In order to use other register instead of EAX, you have to add to the third byte the number of that register (from 00h to 07h) in the following list: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, for example: 0F 20 C3 = MOV EBX,CR0
DEBUG.EXE can't unassemble these codes, but doesn't hamper debugging of programs with these codes, if programs are executed by 32-bit CPU. Operand size override prefix (7.02-06) before these commands isn't needed.

Note 2: DEBUG.EXE doesn't "know" commands appealing to segment registers GS and FS of 32-bit CPUs, but codes of these commands can be entered as data by DB instruction (7.01-01). Codes of these commands are 2 bytes long:

8C E0 = MOV AX,FS
8C E8 = MOV AX,GS
8E E0 = MOV FS,AX
8E E8 = MOV GS,AX

In order to use other register instead of AX, you have to add to the second byte the number of that register (from 00h to 07h) in a list, given in second line of table 7.00, for example:

8E E3 = MOV FS,BX

DEBUG.EXE improperly unassembles codes of these commands as related to ES and CS segment registers, but this doesn't hamper debugging of programs with these codes, if programs are executed by a 32-bit CPU.

Note 3: the MOV command can't copy data into CS segment register; this can be done by control transfer commands only (CALL, JMP, RETF, etc.).

Note 4: the MOV command copying data into SS register induces hardware blocking of external interrupts for the time of execution of one next command. It is implied, that the next command must write new offset into SP register. Only this order of commands excludes failures, caused by external interrupts, at the moment of transition to other stack.

Note 5: codes 8(A,B) (C-F)(0-F), 8(C,E)(2,3,6,7,A,B,F)(0-F) and C(6,7) (C-F)(0-F) are also unassembled by DEBUG.EXE as MOV command.

7.03-59 MOVSB – serial copying of bytes

Though the name MOVSB stands for "MOVe String of Bytes", the MOVSB command in fact copies a single byte. The source byte address must be loaded beforehand into DS:SI pair of registers, the destination address – into ES:DI pairs of registers. After copying both offsets – in SI (source index) register and in DI (destination index) register – are incremented by 1 or decremented by 1: it depends on the state ("UP" or "DN") of direction flag DF. The state of DF flag can be altered by CLD (7.03-11) and STD (7.03-85) commands. Automatic change of index registers contents prepares conditions for copying of the next byte in the next memory cell. States of flags are not altered by MOVSB command.

The MOVSB command is often preceded by repetition prefixes F2h (7.02-03) or F3h (7.02-04), which enable to execute it cyclically and thus copy a string of bytes. The MOVSB command also may be preceded by a segment override prefix (7.02-01); it enables to refer to other segment register instead of default source segment register DS. Destination segment register ES can't be changed by prefix.

Code	Example
A4	MOVSB

7.03-60 MOVSW – serial copying of words

The MOVSW command (MOVSW = Move String of Words) copies a word and then increments (or decrements) contents of SI and DI index registers by 2, thus preparing source and destination offsets to copying the next word into the next pair of memory cells. Operand size override prefix 66h (7.02-06) forces MOVSW command to copy four-byte operands (of DWORD type) and to increment (or decrement) contents of index registers

by 4. All other peculiarities of MOVSW command execution are the same as those for MOVSB command (7.03-59).

Code	Example
A5	MOVSW

7.03-61 MUL – multiplication of unsigned integers

The MUL command (MUL = MULtiplcation) multiplies unsigned integers (for multiplication of signed integers see 7.03-25). Explicit operand of MUL command represents a multiplier. If this operand is a byte, then the other operand is implied to exist in AL register; after multiplication the product is left in AX register. If explicit operand is a word, then the other operand is implied to exist in AX register; after multiplication the less significant 2 bytes of product are left in AX register, and the most significant 2 bytes of product – in DX register.

If the most significant part of product in AH or in DX register represents non-zero values, then flags OF and CF are set by MUL command to OV and CY states correspondingly. On the contrary, cleared states NV and NC of these flags indicate, that the most significant part of product is filled with zeros. Flags SF, ZF, AF, PF acquire indefinite state.

The MUL command can be applied to binary integers and to unpacked decimal bytes. Packed decimal operands must be unpacked beforehand. Product of unpacked decimal bytes needs to be transformed into unpacked decimal format by AAM command (7.03-03).

First byte	Second byte	Data bytes	Examples
F6	(2,6,A)(0-7)	0-2	`MUL byte ptr [bp+si+ffff]`
F6	E(0-7)		`MUL bl`
F7	(2,6,A)(0-7)	0-2	`MUL word ptr [bp+si+ffff]`
F7	E(0-7)		`MUL bx`

7.03-62 NEG – operand's sign reversal

The NEG command (NEG = NEGate) subtracts its operand from zero. Thus the sign of non-zero operands is reversed, but zero operands are left unchanged. Flags (OF, SF, ZF, AF, PF, CF) acquire new states according to the result.

First byte	Second byte	Data bytes	Examples
F6	(1,5,9)(8-F)	0-2	`NEG byte ptr [bp+si+ffff]`
F6	D(8-F)		`NEG bl`
F7	(1,5,9)(8-F)	0-2	`NEG word ptr [bp+si+ffff]`
F7	D(8-F)		`NEG bx`

7.03-63 NOP – a void operation

Though the NOP command (NOP = No operation) is known to do nothing, it in fact increments IP (instruction pointer) by 1, so since that IP points at the next command.

Code	Example
90	NOP

7.03-64 NOT – inversion of operand's bits

The NOT command subjects to logical NOT operation every bit in its operand. States of flags are not altered by NOT command.

First byte	Second byte	Data bytes	Examples
F6	(1,5,9)(0-7)	0-2	`NOT byte ptr [bp+si+ffff]`
F6	D(0-7)		`NOT bl`
F7	(1,5,9)(0-7)	0-2	`NOT word ptr [bp+si+ffff]`
F7	D(0-7)		`NOT bx`

7.03-65 OR – logical OR operation

The OR command analyses pairs of corresponding bits in two operands. If TRUE state is set in at least one bit in a pair, then corresponding bit of the result is set to TRUE state too. If both bits in a pair are cleared to FALSE state, then corresponding bit of the result is also cleared to FALSE state. Result replaces the first operand. Flags SF, ZF, PF acquire new states according to the result. Flags CF and OF are cleared to states NC (No Carry) and NV (No oVerflow) respectively. Flag AF acquires indefinite state.

First byte	Second byte	Data bytes	Examples
08	(0-B)(0-F)	0-2	`OR [bp+si+ffff],bl`
08	(C-F)(0-F)		`OR bl,bl`
09	(0-B)(0-F)	0-2	`OR [bp+si+ffff],bx`
09	(C-F)(0-F)		`OR bx,bx`
0A	(0-B)(0-F)	0-2	`OR bl,[bp+si+ffff]`
0B	(0-B)(0-F)	0-2	`OR bx,[bp+si+ffff]`
0C		1	`OR AL,ff`
0D		2	`OR AX,ffff`
80	(0,4,8)(8-F)	1-3	`OR byte ptr [bp+si+ffff],ff`
80	C(9-F)	1	`OR bl,ff`
81	(0,4,8)(8-F)	2-4	`OR word ptr [bp+si+ffff],ffff`
81	C(9-F)	2	`OR bx,ffff`
83	(0,4,8)(8-F)	1-3	`OR word ptr [bp+si+ffff],±7f`
83	C(9-F)	1	`OR bx,±7f`

Note 1: codes "0(A,B) (C-F)(0-F)" and "82 (0,4,8,C)(8-F)" are also unassembled by DEBUG.EXE as OR command.

Note 2: when OR command is applied to equal operands, these operands wouldn't be changed. For example, OR AX,AX command is often used just to set flags.

7.03-66 OUT – data output to port

While performing the OUT command, CPU generates a signal, which switches CPU's buses from memory to I/O ports. First operand of OUT command specifies target port address either explicitly as a double-digit hexadecimal number or indirectly – as contents of DX register. The second operand of OUT command defines data source register: a byte register AL, if a byte is to be sent, or a double-byte register AX, if a word is to be sent. States of CPU's flags are not altered by OUT command.

First byte	Second byte	Data bytes	Examples
E6		1	`OUT ff,AL`
E7		1	`OUT ff,AX`
EE			`OUT DX,AL`
EF			`OUT DX,AX`

Note 1: selected port addresses are shown in appendix A.14-1. Direct forms of OUT command don't allow port addresses above FFh. Indirect addressing via DX register is not subjected to this restriction.

Note 2: the OUT command wouldn't be executed, if privilege level of the current program is lower than privilege level for I/O operations, defined by bits 0Ch and 0Dh in flags register (A.11-4).

7.03-67 POP – data ejection out of stack

The POP command copies a data word (2 bytes) from stack's top into specified register or memory cell, and then shifts stack's top by incrementing offset in SP register (stack pointer) by 2. States of flags are not altered by POP command.

First byte	Second byte	Data bytes	Examples	Comments
07			POP ES	
0F	A1		DB 0F A1	= POP FS
0F	A9		DB 0F A9	= POP GS
17			POP SS	
1F			POP DS	
5(8-F)			POP bx	
8F	(0,8)(0-7)	0-2	POP [bp+si+ffff]	

Note 1: commands popping data from stack into FS and GS segment registers are not "known" to DEBUG.EXE, but may be entered by DB instruction (7.01-01). DEBUG.EXE can't unassemble codes of these commands. Nevertheless DEBUG.EXE enables to debug programs with such codes, if programs are executed by a 32-bit processor.

Note 2: codes "8F (C-F)(0-F)" are unassembled by DEBUG.EXE as "POP bx".

7.03-68 POPF – restoration of flag's states out of stack

The POPF command copies data word (2 bytes) from stack's top into flags register, and then shifts stack's top by incrementing offset in SP register (stack pointer) by 2. Flags acquire new states, defined by bits of the ejected data word.

Code	Example
9D	POPF

Note 1: the POPF command can't alter states in I/O privilege level field (bits 0Ch and 0Dh) of flags register (A.11-4), if current program is executed at any non-highest privilege level.

Note 2: the POPF command can't alter state of IF flag, if privilege level of current program is lower than privilege level for I/O operations, defined by bits 0Ch and 0Dh in flags register (A.11-4).

Note 3: being preceded by operand size override prefix 66h (7.02-06), the POPF command pops 4 bytes from stack into extended 32-bit flags register. However, this way of access to V86 mode flag is blocked by hardware (more about that – in notes 4 and 5 to A.11-4).

7.03-69 PUSH – copying of a data word into stack.

The PUSH command decrements SP register (stack pointer) by 2, thus extending stack by two memory cells for new data. Then data are copied into these memory cells from that source, which is defined by operand of PUSH command. States of flags are not altered by PUSH command.

First byte	Second byte	Data bytes	Examples	Comments
06			PUSH ES	
0E			PUSH CS	
0F	A0		DB 0F A0	= PUSH FS
0F	A8		DB 0F A8	= PUSH GS
16			PUSH SS	
1E			PUSH DS	
5(0-7)			PUSH bx	
68		2	DB 68 ff ff	= PUSH ffff
6A		1	DB 6A ff	= PUSH 00ff
FF	(3,7,B)(0-7)	0-2	PUSH [bp+si+ffff]	

Note 1: commands pushing explicit integers and segment addresses from FS and GS registers are not "known" to DEBUG.EXE, but may be entered as data by DB instruction (7.01-01). DEBUG.EXE can't unassemble codes of these commands. Nevertheless DEBUG.EXE enables to debug programs with such codes, if programs are executed by a 32-bit processor.

Note 2: it is recommended to avoid PUSH SP operation. Obsolete CPUs first decrement SP, and then copy its value. Most modern CPUs store original SP contents. Therefore in some computers PUSH SP operation may cause unpredictable program's behavior.

Note 3: code "FF F(0-7)" is also unassembled by DEBUG.EXE as "PUSH bx".

7.03-70 PUSHF – copying of flag's states into stack

The PUSHF command (PUSHF = PUSH Flags) copies two bytes from flags register into stack, just as PUSH command (7.03-69) copies a data word. All peculiarities of execution are the same.

Code	Example
9C	PUSHF

Note 1: being preceded by operand size override prefix 66h (7.02-06), the PUSHF command copies into stack 4 bytes from extended 32-bit flags register (see note 4 to A.11-4). However, copying of the V86 mode flag state by PUSHF command is blocked by hardware.

7.03-71 RCL – leftward shift through carry flag

The RCL command (RCL = Rotate through Carry Leftward) arranges a circular shift of its first operand through carry flag to the left, towards more significant bit positions. At each step the most significant bit becomes the state of CF flag, while the least significant bit of operand acquires previous state of CF flag. State of OV flag may be altered too, but other flags preserve their former states.

The second operand (1 or CL) defines the number of shift steps to the left. When the number of shift steps is read from CL register, only 5 less significant bits are taken into account; hence maximum number of shift steps is 31. The preset number of shift steps in CL register is preserved intact.

First byte	Second byte	Data bytes	Examples
D0	(1,5,9)(0-7)	0-2	`RCL byte ptr [bp+si+ffff],1`
D0	D(0-7)		`RCL bl,1`
D1	(1,5,9)(0-7)	0-2	`RCL word ptr [bp+si+ffff],1`
D1	D(0-7)		`RCL bx,1`
D2	(1,5,9)(0-7)	0-2	`RCL byte ptr [bp+si+ffff],CL`
D2	D(0-7)		`RCL bl,CL`
D3	(1,5,9)(0-7)	0-2	`RCL word ptr [bp+si+ffff],CL`
D3	D(0-7)		`RCL bx,CL`

7.03-72 RCR – shift through carry flag to the right

The RCR command (RCR = Rotate through Carry to the Right) arranges a circular shift of its first operand through carry flag to the right, towards less significant bit positions. At each step the least significant bit becomes the state of CF flag, while the most significant bit of operand acquires previous state of CF flag. State of OV flag may be altered too, but other flags preserve their former states.

The second operand (1 or CL) defines the number of shift steps to the right. When the number of shift steps is read from CL register, only 5 less significant bits are taken into

account; hence maximum number of shift steps is 31. The preset number of shift steps in CL register is preserved intact.

First byte	Second byte	Data bytes	Examples
D0	(1,5,9)(8-F)	0-2	`RCR byte ptr [bp+si+ffff],1`
D0	D(8-F)		`RCR bl,1`
D1	(1,5,9)(8-F)	0-2	`RCR word ptr [bp+si+ffff],1`
D1	D(8-F)		`RCR bx,1`
D2	(1,5,9)(8-F)	0-2	`RCR byte ptr [bp+si+ffff],CL`
D2	D(8-F)		`RCR bl,CL`
D3	(1,5,9)(8-F)	0-2	`RCR word ptr [bp+si+ffff],CL`
D3	D(8-F)		`RCR bx,CL`

7.03-73 RET – return within the same segment

The RET command performs a return to the caller program from subroutines, which are present inside the same code segment and are called by CALL command with double-byte target address (for those called by CALL FAR command with 4-byte target address the RETF command must be used, 7.03-74).

The RET command implies, that top stack register contains return offset, i.e. offset of the next command in the caller program. Operand for RET command is not needed, if terminating subroutine leaves nothing in stack. However, subroutines may accept parameters via stack, and at the moment of termination these parameters must be deleted. Therefore operand of RET command defines the number of bytes, which are to be deleted from stack. The RET command pops return offset from stack into IP (instruction pointer) register, and then adds its operand to contents of SP (stack pointer) register. Thus a return is executed to the caller program, and original position of stack's top is restored. States of flags are not altered by RET command.

First byte	Second byte	Data bytes	Examples
C2		2	`RET ffff`
C3			`RET`

Note 1: stack's top position can be preserved at return offset, if return is executed by RET FFFE command.

Note 2: if code, assembled by DEBUG.EXE, is to be executed inside debugger's environment, then the RET command may be used to terminate execution of this code (example – in 9.02-03).

7.03-74 RETF – return from another segment

The RETF command (RETF = RETurn Far) performs all operations of RET command (7.03-73) and, besides that, restores segment address in CS register from stack. Therefore a return to the caller program from other code segment is implemented.

The RETF command is used as exit in those subroutines and drivers, which are called from other code segments by CALL FAR command (7.02-08) with full 4-byte address, so that original segment address of the caller program is saved in stack.

First byte	Second byte	Data bytes	Examples
CA CB		2	`RETF ffff` `RETF`

7.03-75 ROL – circular shift leftward

The ROL command (ROL = ROtate Leftward) arranges a circular shift of its first operand to the left, towards more significant bit positions. At each step the least significant bit acquires the "ejected" former state of the most significant bit. States of CF and OV flags are altered according to result, but all other flags preserve their former states.

The second operand (1 or CL) defines the number of shift steps to the left. When the number of shift steps is read from CL register, only 5 less significant bits are taken into account; hence maximum number of shift steps is 31. The preset number of shift steps in CL register is preserved intact.

First byte	Second byte	Data bytes	Examples
D0	(0,4,8)(0-7)	0-2	`ROL byte ptr [bp+si+ffff],1`
D0	C(0-7)		`ROL bl,1`
D1	(0,4,8)(0-7)	0-2	`ROL word ptr [bp+si+ffff],1`
D1	C(0-7)		`ROL bx,1`
D2	(0,4,8)(0-7)	0-2	`ROL byte ptr [bp+si+ffff],CL`
D2	C(0-7)		`ROL bl,CL`
D3	(0,4,8)(0-7)	0-2	`ROL word ptr [bp+si+ffff],CL`
D3	C(0-7)		`ROL bx,CL`

7.03-76 ROR – circular shift to the right

The ROR command (ROR = ROtate to the Right) arranges a circular shift of its first operand to the right, towards less significant bit positions. At each step the most significant

bit acquires the "ejected" former state of the least significant bit. States of CF and OV flags are altered according to result, but all other flags preserve their former states.

The second operand (1 or CL) defines the number of shift steps to the right. When the number of shift steps is read from CL register, only 5 less significant bits are taken into account; hence maximum number of shift steps is 31. The preset number of shift steps in CL register is preserved intact.

First byte	Second byte	Data bytes	Examples
D0	(0,4,8)(8-F)	0-2	`ROR byte ptr [bp+si+ffff],1`
D0	C(8-F)		`ROR bl,1`
D1	(0,4,8)(8-F)	0-2	`ROR word ptr [bp+si+ffff],1`
D1	C(8-F)		`ROR bx,1`
D2	(0,4,8)(8-F)	0-2	`ROR byte ptr [bp+si+ffff],CL`
D2	C(8-F)		`ROR bl,CL`
D3	(0,4,8)(8-F)	0-2	`ROR word ptr [bp+si+ffff],CL`
D3	C(8-F)		`ROR bx,CL`

7.03-77 SAHF – copying AH into flags register

The SAHF command (SAHF = Store AH in Flags) copies a byte from AH register into lower part of flags register. Bit 7 will define the state of SF (sign flag), bit 6 – the state of ZF (zero flag), bit 4 – the state of AF (auxiliary flag), bit 2 – the state of PF (parity flag) and bit 0 – the state of CF (carry flag). Though SAHF command doesn't define the state of overflow flag OF, nevertheless the latter may acquire indefinite state. Bits 5, 3, 1 in AH register don't correspond to real flags, their states are ignored.

Code	Example
9E	SAHF

7.03-78 SAR – signed integer shift to the right

The SAR command (SAR = Shift Arithmetic to the Right) shifts its signed integer operand to the right, towards less significant bit positions. At each shift's step the state of the rightmost bit becomes lost, and the leftmost bit acquires state of sign bit. Flags ZF, PF, CF acquire new states according to the result. Flags OF and AF acquire indefinite state.

The second operand (1 or CL) defines the number of shift steps to the right. When the number of shift steps is read from CL register, only 5 less significant bits are taken into account; hence maximum number of shift steps is 31. The preset number of shift steps in CL register is preserved intact.

First byte	Second byte	Data bytes	Examples
D0	(3,7,B)(8-F)	0-2	`SAR byte ptr [bp+si+ffff],1`
D0	F(8-F)		`SAR bl,1`
D1	(3,7,B)(8-F)	0-2	`SAR word ptr [bp+si+ffff],1`
D1	F(8-F)		`SAR bx,1`
D2	(3,7,B)(8-F)	0-2	`SAR byte ptr [bp+si+ffff],CL`
D2	F(8-F)		`SAR bl,CL`
D3	(3,7,B)(8-F)	0-2	`SAR word ptr [bp+si+ffff],CL`
D3	F(8-F)		`SAR bx,CL`

7.03-79 SBB – binary integers subtraction with borrow

The SBB command subtracts its second operand (the substrahend) from the first operand (the minuend), taking into account the borrow, left after previous operation and represented by state of CF (carry flag). Remainder replaces the first operand. Flags OF, SF, ZF, AF, PF, CF acquire new states according to the result.

Interpretation of flag's states, left by SBB command, depends on whether the operands were signed or unsigned numbers. Conditional jump commands JA, JB, JBE, JNB should be used after subtraction of unsigned numbers. Other conditional jump commands JG, JGE, JL, JLE should be used after subtraction of signed numbers. Full names of all conditional jump and loop commands reflect status relation of the first (left) operand of SBB command to the second (right) operand. For example, JA = "jump if above" means that the left operand of SBB command (the minuend) must be above, or greater than the right operand (the subtrahend).

SBB is a binary operation, but there are two exceptions. If the first operand is in AX register, then SBB command may be applied to unpacked decimal words: binary difference of unpacked decimal words in AX register can be transformed into valid unpacked decimal word by AAS command (7.03-04). If the first operand is in AL register, then SBB command may be applied to packed decimal bytes: binary difference of packed decimal bytes in AL register can be transformed into valid packed decimal byte by DAS command (7.03-19).

First byte	Second byte	Data bytes	Examples
18	(0-B)(0-F)	0-2	`SBB [bp+si+ffff],bl`
18	(C-F)(0-F)		`SBB bl,bl`
19	(0-B)(0-F)	0-2	`SBB [bp+si+ffff],bx`
19	(C-F)(0-F)		`SBB bx,bx`
1A	(0-B)(0-F)	0-2	`SBB bl,[bp+si+ffff]`

Continuation of table 7.03-79

1B	(0-B)(0-F)	0-2	`SBB bx,[bp+si+ffff]`
1C		1	`SBB AL,ff`
1D		2	`SBB AX,ffff`
80	(1,5,9)(8-F)	1-3	`SBB byte ptr [bp+si+ffff],ff`
80	D(9-F)	1	`SBB bl,ff`
81	(1,5,9)(8-F)	2-4	`SBB word ptr [bp+si+ffff],ffff`
81	D(9-F)	2	`SBB bx,ffff`
83	(1,5,9)(8-F)	1-3	`SBB word ptr [bp+si+ffff],±7f`
83	D(9-F)	1	`SBB bx,±7f`

Note 1: codes "1(A,B) (C-F)(0-F)" and "82 (1,5,9,D)(8-F)" are also unassembled by DEBUG.EXE as SBB command.

7.03-80 SCASB – search for a particular byte

Though the name SCASB stands for "SCAn a String of Bytes", the SCASB command in fact compares a byte in AL register with another byte, read out of memory. Address of that another byte must be loaded beforehand into ES:DI pair of registers. If bytes are equal, ZF (zero flag) is set to ZR state. If bytes are not equal, ZF flag is cleared to NZ (No Zero) state. Flags OF, SF, AF, PF, CF acquire the states according to difference between compared bytes, but this difference itself is not saved.

After comparison offset in DI (destination index) register is incremented by 1 or decremented by 1: it depends on the state ("UP" or "DN") of direction flag DF. The state of DF flag can be altered by CLD (7.03-11) and STD (7.03-85) commands. Automatic change of index register contents prepares conditions for comparison of AL contents with a byte in the next memory cell.

The SCASB command is often preceded by repetition prefixes F2h (7.02-03) or F3h (7.02-04), which enable to execute it cyclically and thus search for a particular byte in a string of bytes. Default segment register ES for that string of bytes can't be altered by segment override prefixes.

Code	Example
AE	SCASB

Note 1: when SCASB command is preceded by repetition prefixes F2h or F3h, the order of operations within the cycle includes assignment of flags states, then incrementation (or decrementation) of index register's contents, and after that cycle termination condition check. Therefore, the offset in DI register at the moment of cycle termination is pointing not to that data byte, which has caused cycle termination, but rather to the next byte.

7.03-81 SCASW – search for a particular word

The SCASW command (SCASW = SCAn a String of Words) compares a word in AX register with another word, read out of memory, and then increments (or decrements) offset of that another word in DI index register by 2, thus preparing comparison of AX contents with a word in next memory cells. The operand size override prefix 66h (7.02-06) forces SCASW command to compare a four-byte operand in EAX register with other operand of the same DWORD type, and to increment (or decrement) offset in DI index register by 4. All other peculiarities of SCASW command execution are the same as those for SCASB command (7.03-80).

Code	Example
AF	SCASW

7.03-82 SHL – Shift to the left

The SHL command shifts its first operand step-by-step to the left, towards more significant bit positions. At each step the state of the most significant bit becomes shifted into carry flag CF, and the least significant bit acquires zero (cleared) state. Flags SF, ZF, PF acquire new states according to the result. Flags AF and OV acquire indefinite state.

The second operand (1 or CL) defines the number of shift steps to the left. When the number of shift steps is read from CL register, only 5 less significant bits are taken into account; hence maximum number of shift steps is 31. The preset number of shift steps in CL register is preserved intact.

First byte	Second byte	Data bytes	Examples
D0	(2,6,A)(0-7)	0-2	`SHL byte ptr [bp+si+ffff],1`
D0	E(0-7)		`SHL bl,1`
D1	(2,6,A)(0-7)	0-2	`SHL word ptr [bp+si+ffff],1`
D1	E(0-7)		`SHL bx,1`
D2	(2,6,A)(0-7)	0-2	`SHL byte ptr [bp+si+ffff],CL`
D2	E(0-7)		`SHL bl,CL`
D3	(2,6,A)(0-7)	0-2	`SHL word ptr [bp+si+ffff],CL`
D3	E(0-7)		`SHL bx,CL`
C0	E(0-7)	1	`see note 2`
C1	E(0-7)	1	`see note 2`

Note 1: SHL command is an exact equivalent of SAL command, accepted by other assemblers, but DEBUG.EXE doesn't accept the SAL name.

Note 2: since CPU model 80286, processors execute SHL command with explicit specification of shift steps. This form of SHL command is not known to DEBUG.EXE, but may be entered as data by DB instruction (7.01-01). For example, machine codes of 4-step shift to the left may look as

C0 E0 04 = SHL AL,4
C1 E0 04 = SHL AX,4

The last byte defines the number of shift steps. In order to apply shift to other register you have to add to the second byte (E0h) the number (00h – 07h) of the desired register in lists, presented in first and second lines of table 7.00.

7.03-83 SHR – shift to the right

The SHR command shifts its first operand step-by-step to the right, towards less significant bit positions. At each step the state of the least significant bit becomes shifted into carry flag CF, and the most significant bit acquires zero (cleared) state. Flags SF, ZF, PF acquire new states according to the result. Flags AF and OV acquire indefinite state.

The second operand (1 or CL) defines the number of shift steps to the left. When the number of shift steps is read from CL register, only 5 less significant bits are taken into account; hence maximum number of shift steps is 31. The preset number of shift steps in CL register is preserved intact.

First byte	Second byte	Data bytes	Examples
D0	(2,6,A)(8-F)	0-2	`SHR byte ptr [bp+si+ffff],1`
D0	E(8-F)		`SHR bl,1`
D1	(2,6,A)(8-F)	0-2	`SHR word ptr [bp+si+ffff],1`
D1	E(8-F)		`SHR bx,1`
D2	(2,6,A)(8-F)	0-2	`SHR byte ptr [bp+si+ffff],CL`
D2	E(8-F)		`SHR bl,CL`
D3	(2,6,A)(8-F)	0-2	`SHR word ptr [bp+si+ffff],CL`
D3	E(8-F)		`SHR bx,CL`
C0	E(8-F)	1	`see note 1`
C1	E(8-F)	1	`see note 1`

Note 1: since CPU model 80286, processors execute SHR command with explicit specification of shift steps. This form of SHL command is not known to DEBUG.EXE, but may be entered as data by DB instruction (7.01-01). For example, machine codes of 4-step shift to the right may look as

C0 E8 04 = SHR AL,4
C1 E8 04 = SHR AX,4

The last byte defines the number of shift steps. In order to apply shift to other register you have to add to the second byte (E0h) the number (00h – 07h) of the desired register in lists, presented in first and second lines of table 7.00.

7.03-84 STC – set carry flag

The STC command sets carry flag CF to the "CY" (CarrY) state, which is often referred to as CF=1.

Code	Example
F9	STC

7.03-85 STD – set direction flag

The STD command sets direction flag DF into it's non-default state "DN". This means descending direction of offset count in index registers (DI and/or SI) during execution of string operations (CMPSB, LODSB, MOVSB, SCASB, STOSB, etc.).

Code	Example
FD	STD

7.03-86 STI – set interrupt flag

The STI command sets interrupt flag IF into its default "EI" (= Enable Interrupts) state, thus enabling intake of interrupt requests via interrupt controller.

Code	Examples
FB	STI

Note 1: the STI command wouldn't be executed, if privilege level of the current program is lower than privilege level for I/O operations, defined by bits 0Ch and 0Dh in flags register (A.11-4).

7.03-87 STOSB – filling memory with a byte

Though the name STOSB stands for "STOre String of Bytes", the STOSB command in fact copies a byte from AL register into a memory cell. Address of that memory cell must be loaded beforehand into ES:DI pair of registers. States of flags are not altered by STOSB command.

After copying offset in DI (destination index) register is incremented by 1 or decremented by 1: it depends on the state ("UP" or "DN") of direction flag DF. The state of DF flag can be altered by CLD (7.03-11) and STD (7.03-85) commands. Automatic change of index register contents prepares conditions for copying a byte from AL register into the next memory cell.

The STOSB command is often preceded by repetition prefixes F2h (7.02-03) or F3h (7.02-04), which enable to execute it cyclically and thus fill a succession of memory cells with copies of the same byte. Default segment register ES for these memory cells can't be altered by segment override prefixes.

Code	Example
AA	STOSB

7.03-88 STOSW – filling memory with a word

The STOSW command (STOSW = STOre String of Words) copies a word from AX register into memory according to address in ES:DI pair of registers and then increments (or decrements) offset in DI index register by 2, thus preparing copying of AX contents in next memory cells. Operand size override prefix 66h (7.02-06) forces STOSW command to copy a four-byte operand of DWORD type from EAX register and to increment (or decrement) offset in DI index register by 4. All other peculiarities of STOSW command execution are the same as those for STOSB command (7.03-87).

Code	Example
AB	STOSW

7.03-89 SUB – binary integers subtraction

The SUB command subtracts its second operand (the subtrahend) from the first operand (the minuend), ignoring the state of CF flag (the borrow). Remainder replaces the first operand. Flags OF, SF, ZF, AF, PF, CF acquire new states according to the result.

Interpretation of flag's states, left by SUB command, depends on whether the operands were signed or unsigned numbers. Conditional jump commands JA, JB, JBE, JNB should be used after subtraction of unsigned numbers. Other conditional jump commands JG, JGE, JL, JLE should be used after subtraction of signed numbers. Full names of all conditional jump and loop commands reflect status relation of the first (left) operand of SUB command to the second (right) operand. For example, JA = "jump if above" means that the left operand of SUB command (the minuend) must be above, or greater than the right operand (the subtrahend).

SUB is a binary operation, but there are two exceptions. If the first operand is in AX register, then SUB command may be applied to unpacked decimal words: binary difference of unpacked decimal words in AX register can be transformed into valid unpacked decimal word by AAS command (7.03-04). If the first operand is in AL register, then SUB command may be applied to packed decimal bytes: binary difference of packed decimal bytes in AL register can be transformed into valid packed decimal byte by DAS command (7.03-19).

First byte	Second byte	Data bytes	Examples
28	(0-B)(0-F)	0-2	`SUB [bp+si+ffff],bl`
28	(C-F)(0-F)		`SUB bl,bl`
29	(0-B)(0-F)	0-2	`SUB [bp+si+ffff],bx`
29	(C-F)(0-F)		`SUB bx,bx`
2A	(0-B)(0-F)	0-2	`SUB bl,[bp+si+ffff]`
2B	(0-B)(0-F)	0-2	`SUB bx,[bp+si+ffff]`
2C		1	`SUB AL,ff`
2D		2	`SUB AX,ffff`
80	(2,6,A)(8-F)	1-3	`SUB byte ptr [bp+si+ffff],ff`
80	E(9-F)	1	`SUB bl,ff`
81	(2,6,A)(8-F)	2-4	`SUB word ptr [bp+si+ffff],ffff`
81	E(9-F)	2	`SUB bx,ffff`
83	(2,6,A)(8-F)	1-3	`SUB word ptr [bp+si+ffff],±7f`
83	E(9-F)	1	`SUB bx,±7f`

Note 1: codes "2(A,B) (C-F)(0-F)" and "82 (2,6,A,E)(8-F)" are also unassembled by DEBUG.EXE as SUB command.

7.03-90 TEST – logical test of bit's states

The TEST command sets flags SF (sign flag), ZF (zero flag) and PF (parity flag) according to result of logical AND bit-to-bit operation upon operands, but this result itself is not saved. Both operands of TEST command remain intact. Carry flag CF and overflow flag OF are cleared by TEST command to states NC (No Carry) and NV (No Overflow) correspondingly. AF flag acquires indefinite state.

First byte	Second byte	Data bytes	Examples
84	(0-B)(0-F)	0-2	`TEST [bp+si+ffff],bl`
84	(C-F)(0-F)		`TEST bl,bl`
85	(0-B)(0-F)	0-2	`TEST [bp+si+ffff],bx`
85	(C-F)(0-F)		`TEST bx,bx`

Continuation of table 7.03-90

A8		1	`TEST AL,ff`
A9		2	`TEST AX,ffff`
F6	(0,4,8)(0-7)	1-3	`TEST byte ptr [bp+si+ffff],ff`
F6	C(1-7)	1	`TEST bl,ff`
F7	(0,4,8)(0-7)	2-4	`TEST word ptr [bp+si+ffff],ffff`
F7	C(1-7)	2	`TEST bx,ffff`

7.03-91 XCHG – operands exchange

The XCHG command exchanges contents between specified registers or between a memory cell and a register. States of flags are not altered by XCHG command.

First byte	Second byte	Data bytes	Examples
86	(0-B)(0-F)	0-2	`XCHG [bp+si+ffff],bl`
86	(C-F)(1-7,9-F)		`XCHG bl,bl`
87	(0-B)(0-F)	0-2	`XCHG [bp+si+ffff],bx`
87	(C-F)(1-7,9-F)		`XCHG bx,bx`
9(1-7)			`XCHG bx,AX`

7.03-92 XLAT – tabular translation

The XLAT command calculates a sum (AL + BX) and then copies a byte from DS:(AL + BX) address into AL register, replacing its former contents. States of flags and contents of BX register are not altered by XLAT command.

XLAT command is used for translation of codes via a code table (up to 256 bytes long), which must be loaded beforehand starting at DS:BX address and on. Another segment register may be referred instead of the default segment register DS, if XLAT command is preceded by an appropriate segment override prefix (7.02-01).

Code	Example
D7	XLAT

7.03-93 XOR – exclusive OR logical operation

The XOR command analyses pairs of corresponding bits in two operands. If states of both bits in a pair are identical (both set or both cleared), then corresponding bit of the result is cleared to FALSE (zero). If states of bits in an analyzed pair are different, then corresponding bit of the result is set to TRUE state. Result replaces the first operand. Flags SF, ZF, PF acquire new states according to the result. Flags CF and OF are cleared

to states NC (No Carry) and NV (No oVerflow) respectively. Flag AF acquires indefinite state.

First byte	Second byte	Data bytes	Examples
30	(0-B)(0-F)	0-2	`XOR [bp+si+ffff],bl`
30	(C-F)(0-F)		`XOR bl,bl`
31	(0-B)(0-F)	0-2	`XOR [bp+si+ffff],bx`
31	(C-F)(0-F)		`XOR bx,bx`
32	(0-B)(0-F)	0-2	`XOR bl,[bp+si+ffff]`
33	(0-B)(0-F)	0-2	`XOR bx,[bp+si+ffff]`
34		1	`XOR AL,ff`
35		2	`XOR AX,ffff`
80	(3,7,B)(0-7)	1-3	`XOR byte ptr [bp+si+ffff],ff`
80	F(1-7)	1	`XOR bl,ff`
81	(3,7,B)(0-7)	2-4	`XOR word ptr [bp+si+ffff],ffff`
81	F(1-7)	2	`XOR bx,ffff`
83	(3,7,B)(0-7)	1-3	`XOR word ptr [bp+si+ffff],±7f`
83	F(1-7)	1	`XOR bx,±7f`

Note 1: the XOR command with specifications of the same source as each of two operands is often used in order to clear that source to zero.

Note 2: codes "3(2,3) (C-F)(0-F)" and "82 (3,7,B,F)(0-7)" are also unassembled by DEBUG.EXE as XOR command.

7.04 Commands for arithmetical coprocessor

Those assembler's commands, whose name begins with letter "F" (float), are transferred for execution to arithmetical coprocessor. All modern computers are able to perform these commands, because their arithmetical coprocessor is integrated in the main CPU.

Computers with obsolete processors, including some 486 models, may have no arithmetical coprocessor. Then execution of coprocessor's commands may be performed by software emulation, but for that two conditions must be met:

– first, generation of a call for INT 07 handler (8.01-08) must be ensured in response to each coprocessor's command. This is achieved by setting bit 02h ("coprocessor emulation") in control register CR0 (A.11-4).

– second, an appropriate INT 07 handler must be loaded, which is able to emulate execution of coprocessor's commands.

In some old computers both these conditions are met automatically due to their BIOS system, in some others the user has to bother about that. In any case presence or absence of arithmetical coprocessor is reported by INT 11 handler (8.01-36, A.11-1).

If there is a chance that your program will be executed by old CPUs with a separate coprocessor chip, then each coprocessor's command should be preceded by WAIT prefix (7.02-05), which synchronizes command's transfer from CPU to coprocessor. Modern CPUs have an integrated coprocessor with hardware synchronizing means. Therefore for modern CPUs the WAIT prefix is not needed, its presence is allowed, but most probably is ignored.

7.04-01 F2XM1 – approximation for fractional power of 2

The F2XM1 command calculates a sum of series, used for fractional power of 2 function approximation within limits of power index from –1 to +1. The power index is implied to be prepared in coprocessor's top stack register ST(0). Calculated sum of series replaces power index in ST(0) register. Final result can be expressed by formula

ST(0) = –1+2^ST(0)

Code	Example
D9 F0	F2XM1

7.04-02 FABS – absolute value

The FABS command clears sign bit in coprocessor's top stack register ST(0) to zero, thus making the operand in ST(0) a positive value.

Code	Example
D9 E1	FABS

7.04-03 FADD – addition of real values

The FADD command adds a real value being read out of memory or from any coprocessor's register, if it is specified as the second operand, to another real value in coprocessor's top stack register ST(0) or in any other register ST(1-7), if the latter is specified as the first operand. The sum replaces former value in ST(0) or in ST(1-7), if it is specified as the first operand.

First byte	Second byte	Data bytes	Examples
D8	(0,4,8)(0-7)	0-2	`FADD dword ptr [bp+si+ffff]`
D8	C(0-7)		`FADD ST,ST(0-7)`
DC	(0,4,8)(0-7)	0-2	`FADD qword ptr [bp+si+ffff]`
DC	C(0-7)		`FADD ST(1-7),ST`

7.04-04 FADDP – addition and stack shift up

The FADDP command (FADDP = "ADD and Pop") adds its second operand in coprocessor's top stack register ST(0) to the first operand in any other specified stack register ST(1-7). The sum replaces former value in specified ST(1-7) stack register, and then coprocessor's stack pointer is incremented by 1, so that access to former ST(0) is lost, and all other stack registers ST(1-7), including the one where the sum has been stored, become renamed into ST(0-6), as though their number is decremented by 1.

Code	Example
DE C(0-7)	`FADDP ST(1-7),ST`

7.04-05 FBLD – loading with binary transform

The FBLD command (FBLD = "Binary LoaD") reads from memory, starting at specified address, a 10-byte packed decimal integer, containing two decimal digits per byte. This decimal integer is transformed to real binary value and is loaded into coprocessor's register ST(7), which must be empty at that moment. Then FBLD command decrements coprocessor's stack pointer by 1; therefore registers ST(0-6) are renamed into ST(1-7), and register ST(7) is renamed into ST(0), so that at last the loaded value is found in top stack register ST(0).

First byte	Second byte	Data bytes	Example
DF	(2,6,A)(0-7)	0-2	`FBLD tbyte ptr [bp+si+ffff]`

Note 1: the FBLD command doesn't check, whether its operand is really a packed decimal number or not. If not, the result of performed binary transform is invalid.

Note 2: the default 10-byte binary real format includes mantissa (bits 0 – 63), power index (bits 64 – 78) and sign bit 79. By changing contents of precision control field in CWR register (note 2 to 7.04-35) coprocessor may be turned to 8-byte double precision format (52 bits – mantissa, 11 bits – power index) or to 4-byte single precision format (23 bits – mantissa, 8 bits – power index).

7.04-06 FBSTP – store with decimal transform

The FBSTP command (FBSTP = Binary STore and Pop) transforms a real binary value in coprocessor's top stack register ST(0) into a 10-byte packed decimal integer, containing two decimal digits per byte. Transform includes rounding of fractional part. Transformed integer is written into memory, starting from specified offset and on. Then FBSTP command increments coprocessor's stack pointer by 1, so that access to former ST(0) is lost, and registers ST(1-7) become renamed into ST(0-6).

First byte	Second byte	Data bytes	Example
DF	(3,7,B)(0-7)	0-2	FBSTP tbyte ptr [bp+si+ffff]

7.04-07 FCHS – change sign

The FCHS command reverses sign of real binary value in coprocessor's top stack register ST(0).

Code	Example
D9 E0	FCHS

7.04-08 FCLEX – clear exceptions flags

The FCLEX command clears those bits in coprocessor's status word register SWR, which are used as flags to register coprocessor's states and exceptions. In particular, the following bits are cleared:

bit 0 – flag of invalid operation
bit 1 – flag of denormalized operand
bit 2 – flag of division by zero
bit 3 – coprocessor's overflow flag
bit 4 – antioverflow (lost result) flag
bit 5 – flag of lost precision
blt 7 – interrupt request flag
bit 15 – "coprocessor busy" flag

Code	Example
DB E2	FCLEX

7.04-09 FCOM – comparison of real values

The FCOM command compares a real value in coprocessor's top stack register ST(0) with contents of specified register or memory cells. Flags C0, C2 and C3 in coprocessor's register SWR acquire new states according to the result. First the state of C2 flag should be checked: C2 = 1 marks uncomparable operands, so that there is no sense in further checks. If operands are equal, then C3 = 1. If value in ST(0) is less than the other operand, then C0 = 1. The way of performing checks for flags C0, C2 and C3 in coprocessor's SWR register is described in article 7.04-64.

First byte	Second byte	Data bytes	Examples
D8	(1,5,9)(0-7)	0-2	`FCOM dword ptr [bp+si+ffff]`
D8	D(0-7)		`FCOM ST(0-7)`
DC	(1,5,9)(0-7)	0-2	`FCOM qword ptr [bp+si+ffff]`

Note 1: code "DC D(0-7)" is also unassembled by DEBUG.EXE as FCOM ST(0-7).

7.04-10 FCOMP – compare and shift stack up

The FCOMP command performs comparison just as FCOM operation does (7.04-09), but then increments coprocessor's stack pointer by 1, so that access to former ST(0) is lost, and registers ST(1-7) become renamed into ST(0-6).

First byte	Second byte	Data bytes	Example
D8	(1,5,9)(8-F)	0-2	`FCOMP dword ptr [bp+si+ffff]`
D8	D(8-F)		`FCOMP ST(0-7)`
DC	(1,5,9)(8-F)	0-2	`FCOMP qword ptr [bp+si+ffff]`

Note 1: codes "DC D(8-F)" and "DE D(0-7)" are also unassembled by DEBUG.EXE as FCOMP command.

7.04-11 FCOMPP – compare and twice shift stack up

The FCOMPP command compares operands in coprocessor's stack registers ST(0) and ST(1) and then increments coprocessor's stack pointer by 2, so that access to former operands in both ST(0) and ST(1) is lost. Registers ST(2-7) become renamed into ST(0-5). The result of comparison affects states of flags C0, C2 and C3 in coprocessor's SWR register, just as it is done after FCOM command (7.04-09).

Code	Example
DE D9	FCOMPP

Note 1: DEBUG.EXE unassembles code "DE D9" as "FCOMPP ST(1)", but while assembling doesn't accept the "ST(1)".

7.04-12 FDECSTP – DECrement Stack Top Pointer

The FDECSTP command decrements coprocessor's stack pointer by 1, so that registers ST(0-6) are renamed into ST(1-7). The last stack register ST(7) is renamed into ST(0). All stack register's contents remain accessible and are not altered.

Code	Example
D9 F6	FDECSTP

Note 1: coprocessor's stack pointer is a tree-stage reversible counter, involving bits 11, 12 and 13 of status word register SWR.

Note 2: most commands pushing data into coprocessor's stack are performed by copying data into ST(7) register and decrementing coprocessor's stack pointer by 1, so that ST(7) register becomes renamed into ST(0). All such commands can't be performed, if ST(7) register originally isn't empty.

7.04-13 FDISI – disable interrupts

The FDISI command disables interrupts for obsolete 8087 arithmetical coprocessor. Since model 80287 coprocessors don't need this command and ignore it.

Code	Example
DB E1	FDISI

7.04-14 FDIV – division of real values

The FDIV command divides a real value in coprocessor's top stack register ST(0) or in any non-top register ST(1-7), if it is specified as the first operand, by a real divisor from specified memory cell or other coprocessor's register, if it is specified as the second operand. The quotient replaces dividend in ST(0) or in other stack register ST(1-7), if it is specified as the first operand.

First byte	Second byte	Data bytes	Examples
D8	(3,7,B)(0-7)	0-2	`FDIV dword ptr [bp+si+ffff]`
D8	F(0-7)		`FDIV ST,ST(0-7)`
DC	(3,7,B)(0-7)	0-2	`FDIV qword ptr [bp+si+ffff]`
DC	F(8-F)		`FDIV ST(1-7),ST`

7.04-15 FDIVP – divide and shift stack up

The FDIVP – divide a real value in any coprocessor's non-top stack register ST(1-7) with divisor in top stack register ST(0). The quotient replaces the dividend in non-top stack register ST(1-7). Then FDIVP command increments coprocessor's stack pointer by 1, so that registers ST(1-7) are renamed into ST(0-6), including that one, where the quotient has been written. The ST(0) register is renamed into ST(7) and is announced free. Access to its former contents (the divisor) is lost.

Code	Example
DE F(8-F)	`FDIVP ST(1-7),ST`

7.04-16 FDIVR – divide in reverse order

The FDIVR command divides a real value, being read out of a memory cell or from coprocessor's stack register, if it is specified as the second operand, by a real divisor in coprocessor's top stack register ST(0) or in any non-top stack register ST(1-7), if it is specified as the first operand. The quotient replaces divisor in ST(0) register or in non-top stack register ST(1-7), if it is specified as the first operand.

First byte	Second byte	Data bytes	Examples
D8	(3,7,B)(8-F)	0-2	`FDIVR dword ptr [bp+si+ffff]`
D8	F(8-F)		`FDIVR ST,ST(0-7)`
DC	(3,7,B)(8-F)	0-2	`FDIVR qword ptr [bp+si+ffff]`
DC	F(0-7)		`FDIVR ST(1-7),ST`

7.04-17 FDIVRP – divide in reverse order and shift stack up

The FDIVRP command divides a real value in coprocessor's top stack register ST(0) with divisor in any other stack register ST(1-7), replaces the divisor by the quotient in non-top stack register ST(1-7) and then increments coprocessor's stack pointer by 1, so that registers ST(1-7) are renamed into ST(0-6), including that one, where the quotient has

been written. The ST(0) register is renamed into ST(7) and is announced free. Access to its former contents (the dividend) is lost.

Code	Example
DE F(0-7)	`FDIVRP ST(1-7),ST`

7.04-18 FENI – enable interrupts

The FENI command enables interrupts for obsolete 8087 arithmetical coprocessor. Since model 80287 coprocessors don't need this command and ignore it.

Code	Example
DB E0	FENI

7.04-19 FFREE – announce register as free

The FFREE command marks specified coprocessor's stack register as free by writing "11" binary value into corresponding bits of coprocessor's tag register TWR.

Code	Example
DD C(0-7)	`FFREE ST(0-7)`

Note 1: code "DF C(0-7)" is also unassembled by DEBUG.EXE as FFREE command.

7.04-20 FIADD – addition with an integer

The FIADD command adds an integer from specified memory cell to a real value in coprocessor's top stack register ST(0). The sum is a real value, replacing the former contents in ST(0) register.

First byte	Second byte	Data bytes	Examples
DA DE	(0,4,8)(0-7) (0,4,8)(0-7)	0-2 0-2	`FIADD dword ptr [bp+si+ffff]` `FIADD word ptr [bp+si+ffff]`

7.04-21 FICOM – comparison with an integer

The FICOM command reads an integer from specified memory cell, transforms it into a real value and compares the result with a real value in coprocessor's top stack register ST(0). The comparison itself is performed just as it is done by FCOM command (7.04-09).

First byte	Second byte	Data bytes	Examples
DA DE	(1,5,9)(0-7) (1,5,9)(0-7)	0-2 0-2	`FICOM dword ptr [bp+si+ffff]` `FICOM word ptr [bp+si+ffff]`

7.04-22 FICOMP – compare with an integer and shift stack up

The FICOMP command reads an integer from specified memory cell, transforms it into a real value, and compares the result with a real value in coprocessor's top stack register ST(0). The comparison itself is performed just as it is done by FCOM command (7.04-09), but then FICOMP command increments coprocessor's stack pointer by 1, so that registers ST(1-7) are renamed into ST(0-6). The ST(0) register is renamed into ST(7) and is announced free. Access to its former contents is lost.

First byte	Second byte	Data bytes	Examples
DA DE	(1,5,9)(8-F) (1,5,9)(8-F)	0-2 0-2	`FICOMP dword ptr [bp+si+ffff]` `FICOMP word ptr [bp+si+ffff]`

7.04-23 FIDIV – division by an integer

The FIDIV command divides a real value in coprocessor's top stack register ST(0) by an integer divisor, read from specified memory cell. The quotient replaces the dividend in top stack register ST(0).

First byte	Second byte	Data bytes	Examples
DA DE	(3,7,B)(0-7) (3,7,B)(0-7)	0-2 0-2	`FIDIV dword ptr [bp+si+ffff]` `FIDIV word ptr [bp+si+ffff]`

7.04-24 FIDIVR – integer division in reverse order

The FIDIVR command performs division of integer dividend, read from specified memory cell, by a divisor in coprocessor's top stack register ST(0). The quotient replaces the divisor in top stack register ST(0).

First byte	Second byte	Data bytes	Examples
DA	(3,7,B)(8-F)	0-2	`FIDIVR dword ptr [bp+si+ffff]`
DE	(3,7,B)(8-F)	0-2	`FIDIVR word ptr [bp+si+ffff]`

7.04-25 FILD – loading of an integer

The FILD command transforms an integer, read from specified memory cell, into a real value and loads this value into coprocessor's stack register ST(7), which must be empty at that moment. Then FILD command decrements coprocessor's stack pointer by 1; therefore registers ST(0-6) are renamed into ST(1-7), and register ST(7) is renamed into ST(0), so that at last the loaded value is found in top stack register ST(0).

First byte	Second byte	Data bytes	Examples
DB	(0,4,8)(0-7)	0-2	`FILD dword ptr [bp+si+ffff]`
DF	(0,4,8)(0-7)	0-2	`FILD word ptr [bp+si+ffff]`
DF	(2,6,A)(8-F)	0-2	`FILD qword ptr [bp+si+ffff]`

7.04-26 FIMUL – multiplication by an integer

The FIMUL command multiplies a real value in coprocessor's top stack register ST(0) by an integer value, read from specified memory cell. The product replaces former value in coprocessor's top stack register ST(0).

First byte	Second byte	Data bytes	Examples
DA	(0,4,8)(8-F)	0-2	`FIMUL dword ptr [bp+si+ffff]`
DE	(0,4,8)(8-F)	0-2	`FIMUL word ptr [bp+si+ffff]`

7.04-27 FINCSTP – increment stack top pointer

The FINCSTP command increments coprocessor's stack pointer by 1, so that registers ST(1-7) are renamed into ST(0-6). The top stack register ST(0) is renamed into ST(7). All stack register's contents remain accessible and are not altered.

Code	Example
D9 F7	FINCSTP

Note 1: coprocessor's stack pointer is a tree-stage reversible counter, involving bits 11, 12 and 13 of status word register SWR.

Note 2: when incrementing of coprocessor's stack pointer is performed by other commands, then top stack register ST(0) after being renamed into ST(7) acquires status of empty register (tag 11b). Therefore access to former contents of ST(0) is lost. This operation is often referred to as popping ST(0) contents out of stack.

7.04-28 FINIT – setting coprocessor's initial state

The FINIT command writes initial states into CWR, SWR, TWR, IPR and DPR registers of arithmetical coprocessor. Control word register CWR acquires the state 037Fh: it defines 80-bit format of operands, masking of all exceptions and rounding to nearest integer. Tags register TWR is set to FFFFh state, which means that all coprocessor's stack registers are free. Other coprocessor's registers (SWR, IPR and DPR) are cleared to 0000h.

Code	Example
DB E3	FINIT

7.04-29 FIST – storing of an integer

The FIST command (FIST = Integer STore) reads a real value from coprocessor's top stack register ST(0), translates it into an integer, rounds it according to specified format, and writes the result into specified memory address.

First byte	Second byte	Data bytes	Examples
DB DF	(1,5,9)(0-7) (1,5,9)(0-7)	0-2 0-2	`FIST dword ptr [bp+si+ffff]` `FIST word ptr [bp+si+ffff]`

7.04-30 FISTP – store an integer and shift stack up

The FISTP command stores in memory a translated value from ST(0) register, just as FIST command does (7.04-29). Besides that, FISTP command increments coprocessor's stack pointer by 1, so that registers ST(1-7) are renamed into ST(0-6). Access to former value in top stack register ST(0) becomes lost.

First byte	Second byte	Data bytes	Examples
DB	(1,5,9)(8-F)	0-2	`FISTP dword ptr [bp+si+ffff]`
DF	(1,5,9)(8-F)		`FISTP word ptr [bp+si+ffff]`
DF	(3,7,B)(8-F)	0-2	`FISTP qword ptr [bp+si+ffff]`

7.04-31 FISUB – subtraction of an integer

The FISUB command subtracts an integer subtrahend, stored in specified memory cell, from a real value – the minuend – in coprocessor's top stack register ST(0). The remainder replaces minuend in top stack register ST(0).

First byte	Second byte	Data bytes	Examples
DA	(2,6,A)(0-7)	0-2	`FISUB dword ptr [bp+si+ffff]`
DE	(2,6,A)(0-7)	0-2	`FISUB word ptr [bp+si+ffff]`

7.04-32 FISUBR – subtract in reverse order

The FISUBR command performs reverse order subtraction of a real subtrahend in coprocessor's top stack register ST(0) from an integer minuend, stored in specified memory cell. The remainder replaces former subtrahend in top stack register ST(0).

First byte	Second byte	Data bytes	Examples
DA	(2,6,A)(8-F)	0-2	`FISUBR dword ptr [bp+si+ffff]`
DE	(2,6,A)(8-F)	0-2	`FISUBR word ptr [bp+si+ffff]`

7.04-33 FLD – loading of a real value

The FLD command reads a real value from specified register or from specified memory cell and loads this value into coprocessor's stack register ST(7), which must be empty at that moment. Then FLD command decrements coprocessor's stack pointer by 1; therefore registers ST(0-6) are renamed into ST(1-7), and register ST(7) is renamed into ST(0), so that at last the loaded real value is found in top stack register ST(0).

First byte	Second byte	Data bytes	Examples
D9	(0,4,8)(0-7)	0-2	`FLD dword ptr [bp+si+ffff]`
D9	C(0-7)		`FLD ST(0-7)`
DB	(2,6,A)(8-F)	0-2	`FLD tbyte ptr [bp+si+ffff]`
DD	(0,4,8)(0-7)	0-2	`FLD qword ptr [bp+si+ffff]`

7.04-34 FLD1 – loading of a unity constant

The FLD1 command loads a unity constant into coprocessor's stack register ST(7), which must be empty at that moment. Then FLD1 command decrements coprocessor's stack pointer by 1; therefore registers ST(0-6) are renamed into ST(1-7), and register ST(7) is renamed into ST(0), so that at last the loaded constant becomes stored in top stack register ST(0).

Code	Example
D9 E8	FLD1

7.04-35 FLDCW – loading of CWR register

The FLDCW command (FLDCW = LoaD Control Word) copies a data word, saved by FSTCW command (7.04-55), from specified memory cell into coprocessor's control word register CWR. If selected bits in this data word have been intentionally altered, then after loading with FLDCW command the changes will come into effect.

First byte	Second byte	Data bytes	Example
D9	(2,6,A)(8-F)	0-2	`FLDCW word ptr [bp+si+ffff]`

Note 1: bits 5 – 0 in CWR register represent masks for exception flags in corresponding bits 5 – 0 of SWR register (7.04-08). By default masks in CWR register are set, but if a mask is cleared, then occurrence of an exception invokes a request IRQ 13 for interrupt handler INT 75 (8.03-75), which must be able to cope with the problem.

Note 2: bits 9 and 8 in CWR register represent PC (= precision control) field. Default state 11b of PC field defines 10-byte format of operands. State 10b of PC field defines rounding to 8-byte format, state 00b – rounding to 4-byte format (7.04-05). However, reduction of precision doesn't make calculations faster.

7.04-36 FLDENV – loading into service registers

The FLDENV (= LoaD ENVironment) restores states of coprocessor's service registers (CWR, SWR, TWR, IPR, DPR) according to a record, which is read from memory starting at specified address. This record must be formed and stored beforehand by FSTENV command (7.04-56).

First byte	Second byte	Data bytes	Example
D9	(2,6,A)(0-7)	0-2	`FLDENV word ptr [bp+si+ffff]`

7.04-37 FLDL2E – loading of "log e" constant

The FLDL2E command loads a log e = 1.44269... constant (i.e. base 2 logarithm of e = 2.71828...) into coprocessor's stack register ST(7), which must be empty at that moment. Then FLDL2E command decrements coprocessor's stack pointer by 1; therefore registers ST(0-6) are renamed into ST(1-7), and register ST(7) is renamed into ST(0), so that at last the loaded constant becomes stored in top stack register ST(0).

Code	Example
D9 EA	FLDL2E

7.04-38 FLDL2T – loading of "log10" constant

The FLDL2T command loads log10 = 3.32192... constant (i.e. base 2 logarithm of 10) into coprocessor's stack register ST(7), which must be empty at that moment. Then FLDL2T command decrements coprocessor's stack pointer by 1; therefore registers ST(0-6) are renamed into ST(1-7), and register ST(7) is renamed into ST(0), so that at last the loaded constant becomes stored in top stack register ST(0).

Code	Example
D9 E9	FLDL2T

7.04-39 FLDLG2 – loading of "lg2" constant

The FLDLG2 command loads lg2 = 0.301029... constant (i.e. base 10 logarithm of 2) into coprocessor's stack register ST(7), which must be empty at that moment. Then FLDLG2 command decrements coprocessor's stack pointer by 1; therefore registers ST(0-6) are renamed into ST(1-7), and register ST(7) is renamed into ST(0), so that at last the loaded constant becomes stored in top stack register ST(0).

Code	Example
D9 EC	FLDLG2

7.04-40 FLDLN2 – loading of "ln2" constant

The FLDLN2 command loads ln2 = 0.693147... constant (i.e. base e = 2.71828... logarithm of 2) into coprocessor's stack register ST(7), which must be empty at that moment. Then FLDLN2 command decrements coprocessor's stack pointer by 1; therefore registers ST(0-6) are renamed into ST(1-7), and register ST(7) is renamed into ST(0), so that at last the loaded constant becomes stored in top stack register ST(0).

Code	Example
D9 ED	FLDLN2

7.04-41 FLDPI – loading of "PI" constant

The FLDPI command loads PI = 3.14159... constant into coprocessor's stack register ST(7), which must be empty at that moment. Then FLDPI command decrements coprocessor's stack pointer by 1; therefore registers ST(0-6) are renamed into ST(1-7), and register ST(7) is renamed into ST(0), so that at last the loaded constant becomes stored in top stack register ST(0).

Code	Example
D9 EB	FLDPI

7.04-42 FLDZ – loading of zero constant

The FLDZ command loads 0 (i.e. zero) constant into coprocessor's stack register ST(7), which must be empty at that moment. Then FLDZ command decrements coprocessor's stack pointer by 1; therefore registers ST(0-6) are renamed into ST(1-7), and register ST(7) is renamed into ST(0), so that at last the loaded constant becomes stored in top stack register ST(0).

Code	Example
D9 EE	FLDZ

7.04-43 FMUL – multiplication of real values

The FMUL command multiplies a real value in coprocessor's top stack register ST(0) or in any non-top register ST(1-7), if it specified as the first operand, by another real value – the multiplier, which is read from memory or from other stack register, if it is specified as the second operand. The product replaces former contents in top stack register ST(0) or in other stack register ST(1-7), if it is specified as the first operand.

First byte	Second byte	Data bytes	Examples
D8	(0,4,8)(8-F)	0-2	`FMUL dword ptr [bp+si+ffff]`
D8	C(8-F)		`FMUL ST,ST(0-7)`
DC	(0,4,8)(8-F)	0-2	`FMUL qword ptr [bp+si+ffff]`
DC	C(8-F)		`FMUL ST(1-7),ST`

7.04-44 FMULP – multiply and shift stack up

The FMULP command multiplies a real value in specified coprocessor's non-top stack register ST(1-7) by a real multiplier in top stack register ST(0). The product overwrites former value in specified non-top stack register ST(1-7). Then FMULP command increments coprocessor's stack pointer by 1, so that registers ST(1-7) are renamed into ST(0-6), including that one, where the product has been written. The ST(0) register is renamed into ST(7) and is announced free. Access to its former contents (the multiplier) is lost.

Code	Example
DE C(8-F)	`FMULP ST(1-7),ST`

7.04-45 FNOP – a void operation

Though the FNOP command is known to do nothing, it in fact increments IP (instruction pointer) by 2, because machine code of FNOP command itself takes 2 bytes, so since that IP points at the next command.

Code	Example
D9 D0	FNOP

7.04-46 FPATAN – Partial arctangent.

The FPATAN command divides a positive real dividend in coprocessor's top stack register ST(0) by a positive real divisor in stack register ST(1). The divisor must be equal or greater, than the dividend. The quotient is used to calculate approximation of Arctg(ST(0)/ST(1)) function in radians. Result replaces divisor in stack register ST(1), and then FPATAN command increments coprocessor's stack pointer by 1, so that registers ST(1-7) are renamed into ST(0-6). Former ST(1) register, where the result has been written, becomes top stack register ST(0). Access to former contents of ST(0) register (the divisor) is lost.

Code	Example
D9 F3	FPATAN

7.04-47 FPREM – Partial remainder

The main purpose of FPREM command is reduction of periodic trigonometric function's arguments into limits of their main interval. FPREM command divides a real dividend in coprocessor's top stack register ST(0) by a real divisor in stack register ST(1). The remainder replaces dividend in top stack register ST(0). If this remainder is greater, than divisor in ST(1) register, then this remainder is considered as a partial remainder and is marked by setting flag C2 = 1 in SWR register. In this case the FPREM command should be executed repeatedly until cleared state of flag C2 in SWR register indicates acquisition of final remainder.

When final remainder is obtained, the states of flags C3, C1 and C0 in SWR register point at that circle's sector, which corresponds to final value of trigonometric argument. The way of performing checks for flags C3, C2, C1 and C0 in coprocessor's SWR register is described in article 7.04-64.

Code	Example
D9 F8	FPREM

7.04-48 FPTAN – Partial tangent

The FPTAN command accepts in coprocessor's top stack register ST(0) angular argument in radians for calculation of Tg(ST(0)) value approximation. FPTAN command decrements coprocessor's stack pointer by 1; therefore registers ST(0-6) are renamed into ST(1-7), and angular argument occurs in register ST(1). Calculated value of tangent

function replaces argument in register ST(1), and a unity constant is written in register ST(0). In case of success bit C2 in SWR register is cleared to 0, otherwise it is set to 1.

Code	Example
D9 F2	FPTAN

Note 1: if coprocessor's stack register ST(7) isn't empty, stack pointer can't be decremented, and then FPTAN command wouldn't be executed.

Note 2: obsolete arithmetical coprocessors (up to model 80287) require argument of FPTAN command to be within limits from 0 to PI/4.

7.04-49 FRNDINT – round to integer

The FRNDINT command transforms a real value in coprocessor's top stack register ST(0) into an integer by rounding operation. The way rounding is performed depends on the state of RC (Rounding Control) field in CWR register: if RC=00 – round to the nearest integer, if RC=01 – round to the nearest lower integer, if RC=10 – round to the nearest greater integer, if RC=11 – round by omitting fractional part of original real value.

Code	Example
D9 FC	FRNDINT

Note 1: bits 11 and 10 in coprocessor's CWR register constitute the RC (Rounding Control) field. States of all bits in CWR register can be written into memory by FSTCW command (7.04-55), and then states of desired bits can be intentionally changed. Changed states come into effect after loading back into CWR register by FLDCW command (7.04-35).

7.04-50 FRSTOR – restoration of coprocessor's state

The FRSTOR command (FRSTOR = ReSTORe) loads states of all coprocessor's registers, including control registers and stack, from a record of 96 or 108 bytes long, starting at specified memory address. This record must be stored in memory beforehand by FSAVE command (7.04-51). Actual length of this record depends on CPU's mode: real mode or protected mode. Therefore it is important to perform restoration of coprocessor's state in exactly that CPU's mode, under which this record has been stored.

First byte	Second byte	Data bytes	Example
DD	(2,6,A)(0-7)	0-2	`FRSTOR [bp+si+ffff]`

7.04-51 FSAVE – save coprocessor's state

The FSAVE command stores in computer's memory, starting at specified address, states of all stack registers and control registers in arithmetical coprocessor, and then resets coprocessor's registers CWR, SWR, TWR, IPR, DPR just as FINIT command does (7.04-28). The whole record, formed by FSAVE command, is either 96 or 108 bytes long - it depends on CPU's mode: real mode or protected mode. Later this record may be read by FRSTOR command (7.04-50), which enables to restore coprocessor's former state.

First byte	Second byte	Data bytes	Example
DD	(3,7,B)(0-7)	0-2	`FSAVE [bp+si+ffff]`

7.04-52 FSCALE – multiplication by power of 2

The FSCALE command multiplies a real value in coprocessor's top stack register ST(0) by a power of 2 with integer power index, either positive or negative. Power index must be prepared in ST(1) register as a real value. If it is not an integer, it will be rounded to the nearest lower integer. Final product replaces the first multiplier in top stack register ST(0).

Code	Example
D9 FD	FSCALE

7.04-53 FSQRT – square root

The FSQRT calculates square root of a real positive value in coprocessor's top stack register ST(0). Square root value replaces operand in top stack register ST(0).

Code	Example
D9 FA	FSQRT

7.04-54 FST – store a real value

The FST command copies a real value from coprocessor's top stack register ST(0) into any other stack register ST(1-7) or into memory according to specified address and format.

If specified format is shorter, than original 10-byte format in coprocessor's stack registers, then the stored value is rounded according to specified format.

First byte	Second byte	Data bytes	Example

D9
DD
DD
	(1,5,9)(0-7)
	(1,5,9)(0-7)
	D(0-7)
		0-2
		0-2
			`FST dword ptr [bp+si+ffff]`
			`FST qword ptr [bp+si+ffff]`
			`FST ST(1-7)`

Note 1: code "DF D(0-7)" is also unassembled by DEBUG.EXE as FST command.
Note 2: FST command enables to copy new contents into a stack register, which is not free. Former contents of this register will be overwritten.

7.04-55 FSTCW – store a state of CWR register

The FSTCW command copies current state of coprocessor's control register CWR into a data word, stored in memory according to specified address.

First byte	Second byte	Data bytes	Example
D9	(3,7,B)(8-F)	0-2	`FSTCW [bp+si+ffff]`

Note 1: later the stored state of CWR register can be restored by FLDCW command (7.04-35).

7.04-56 FSTENV – store states of service registers

The FSTENV (= Store environment) command writes into memory, starting from specified address, states of all coprocessor's service registers: CWR (Control Word Register), SWR (Status Word Register), TWR (Tags Word Register), IPR (Instruction Pointer Register), DPR (Data Pointer Register). Contrary to FSAVE command (7.04-51), FSTENV command doesn't reset coprocessor's service registers and doesn't save contents of stack registers. Data from the record, formed by FSTENV command, may be later used to restore former states of service registers by FLDENV command (7.04-36).

First byte	Second byte	Data bytes	Example
D9	(3,7,B)(0-7)	0-2	`FSTENV [bp+si+ffff]`

7.04-57 FSTP – store and shift stack up

The FSTP command copies a real value from coprocessor's top stack register ST(0) into any other stack register ST(1-7), which must not necessarily be free, or into memory according to specified address and format. If specified format is shorter, than original 10-byte format in coprocessor's stack registers, then the stored value is rounded according to specified format. Then FSTP command increments coprocessor's stack pointer by 1, so that registers ST(1-7) are renamed into ST(0-6). The ST(0) register is renamed into ST(7) and is announced free. Access to former contents of ST(0) register is lost.

First byte	Second byte	Data bytes	Examples
D9	(1,5,9)(8-F)	0-2	`FSTP dword ptr [bp+si+ffff]`
DB	(3,7,B)(8-F)	0-2	`FSTP tbyte ptr [bp+si+ffff]`
DD	(1,5,9)(8-F)	0-2	`FSTP qword ptr [bp+si+ffff]`
DD	D(8-F)		`FSTP ST(1-7)`

Note 1: codes "D9 D(8-F)" and "DF D(8-F)" are also unassembled by DEBUG.EXE as FSTP command.

7.04-58 FSTSW – store status word

The FSTSW command copies into specified address the state of coprocessor's status word register SWR, chiefly for analyzing results after comparisons. The role of several bits in SWR register is described in articles 7.04-08 and 7.04-64.

First byte	Second byte	Data bytes	Examples	Comments
DD	(3,7,B)(8-F)	0-2	`FSTSW [bp+si+ffff]`	
DF	E0		`ESC 3C,AL`	see note 1

Note 1: CPU models 80486 and higher perform FSTSW AX operation (code "DF E0"), copying coprocessor's status word into CPU's AX register. This operation is not "known" to DEBUG.EXE, but DEBUG.EXE accepts it under its former name ESC 3C,AL.

7.04-59 FSUB – subtraction of real values

The FSUB command (FSUB = SUBtract) subtracts a real value – the subtrahend, stored in a memory cell or in coprocessor's stack register ST(0-7), if it is specified as the second operand, from a real minuend in coprocessor's top stack register ST(0) or in other stack register ST(1-7), if it is specified as the first operand. The remainder replaces

minuend in coprocessor's top stack register ST(0) or in other stack register ST(1-7), if it is specified as the first operand.

First byte	Second byte	Data bytes	Examples
D8	(2,6,A)(0-7)	0-2	`FSUB dword ptr [bp+si+ffff]`
D8	E(8-F)		`FSUB ST,ST(0-7)`
DC	(2,6,A)(0-7)	0-2	`FSUB qword ptr [bp+si+ffff]`
DC	E(8-F)		`FSUB ST(1-7),ST`

7.04-60 FSUBP – subtract and shift stack up

The FSUBP command subtracts a real value – the subtrahend, stored in coprocessor's top stack register ST(0), from another real value – the minuend, stored in specified coprocessor's non-top stack register ST(1-7). The remainder overwrites the minuend in specified non-top stack register ST(1-7). Then FSUBP command increments coprocessor's stack pointer by 1, so that registers ST(1-7) are renamed into ST(0-6), including that one, where the remainder has been written. The ST(0) register is renamed into ST(7) and is announced free. Access to its former contents (the subtrahend) is lost.

Code	Example
DE E(8-F)	`FSUBP ST(1-7),ST`

7.04-61 FSUBR – subtract in reverse order

The FSUBR command subtracts a real value – the subtrahend, stored in coprocessor's top stack register ST(0) or in any non-top stack register ST(1-7), if it is specified as the first operand, from a real minuend, stored in a specified memory cell or in another coprocessor's stack register, specified as the second operand. The remainder replaces subtrahend in coprocessor's top stack register ST(0) or in other stack register ST(1-7), if it is specified as the first operand.

First byte	Second byte	Data bytes	Examples
D8	(2,6,A)(8-F)	0-2	`FSUBR dword ptr [bp+si+ffff]`
D8	E(0-7)		`FSUBR ST,ST(0-7)`
DC	(2,6,A)(8-F)	0-2	`FSUBR qword ptr [bp+si+ffff]`
DC	E(0-7)		`FSUBR ST(1-7),ST`

7.04-62 FSUBRP – subtract in reverse order and shift stack up

The FSUBRP command subtracts a real value – the subtrahend, stored in any coprocessor's non-top stack's register ST(1-7), from real minuend in top stack register ST(0). Remainder replaces subtrahend in coprocessor's non-top stack's register ST(1-7). Then FSUBRP command increments coprocessor's stack pointer by 1, so that registers ST(1-7) are renamed into ST(0-6), including that one, where the remainder has been written. The ST(0) register is renamed into ST(7) and is announced free. Access to its former contents (the minuend) is lost.

Code	Example
DE E(0-7)	`FSUBRP ST(1-7),ST`

7.04-63 FTST – comparison with zero

The FTST (= TeST) command compares a real value in coprocessor's top stack register ST(0) with zero constant. States of flags C3, C2 and C0 in coprocessor's status word register SWR are altered according to the result. State of flag C2 should be checked first: if C2 = 1, the operands are incomparable, so there is no sense in further checks. Flag C3 = 1 indicates equality, i.e. zero value in top stack register ST(0). Flag C0 = 1 indicates negative value in top stack register ST(0). If top stack register ST(0) contains a positive real value, then all three flags C3, C2, C0 are cleared to zero. The way of performing checks for flags C0, C2, C3 in coprocessor's SWR register is described in article 7.04-64.

Code	Example
D9 E4	FTST

7.04-64 FXAM – operand's type check

The FXAM (= eXAMine) command determines type of operand in coprocessor's top stack register ST(0). States of flags C3, C2, C1 and C0 in coprocessor's status word register SWR indicate the result. Flag C1 reflects the sign of operand. Interpretation for states of flags C3, C2 and C0 is given in the following table:

C3	C2	C0	Operand's type
0	0	0	- unknown format
0	0	1	- any non-numerical format
0	1	0	- correct real number

Continuation of table 7.04-64

0	1	1	- infinity (tag = 10)
1	0	0	- zero (tag = 01)
1	0	1	- empty ST(0) register (tag = 11).
1	1	0	- any denormalized number

In order to analyze the result, status word should be copied by FSTSW command (7.04-58) from coprocessor's SWR register preferably into CPU's AX register. Then states may be either tested by TEST command (7.03-90) or loaded into CPU's flags by SAHF command (7.03-77). As far as flags C0, C1, C2, C3 are represented in SWR register (and in AX register as well) by bits 08, 09, 10 and 14 correspondingly, in AH register the same flags are represented by bits 0, 1, 2, 6. After loading into CPU's flags by SAHF command the state C3 flag is represented by CPU's zero flag ZF, state of C2 flag – by CPU's parity flag PF, state of C0 flag – by CPU's carry flag CF.

Code	Example
D9 E5	FXAM

7.04-65 FXCH – register's contents exchange

The FXCH (= eXCHange) command exchanges contents between coprocessor's top stack register ST(0) and any other specified stack register ST(1-7).

Code	Example
D9 C(8-F)	FXCH ST(1-7)

Note 1: codes "DD C(8-F)" and "DF C(8-F)" are also unassembled by DEBUG.EXE as FXCH command.

7.04-66 FXTRACT – separation of mantissa and exponent

The FXTRACT (= eXTRACT) command decomposes a real value in coprocessor's top stack register ST(0) into its mantissa (i.e. significand) and binary exponent (i.e. binary power index). Mantissa is written into stack register ST(7), which must be empty at that moment. Exponent replaces original value in top stack register ST(0). Then FXTRACT command decrements coprocessor's stack pointer by 1; therefore registers ST(0-6) are renamed into ST(1-7), and register ST(7) is renamed into ST(0), so that at last mantissa occurs in coprocessor's top stack register ST(0), and exponent – in register ST(1).

Code	Example
D9 F4	FXTRACT

7.04-67 FYL2X – logarithm of arbitrary base

The FYL2X command calculates a base 2 logarithm of a positive real value in coprocessor's top stack register ST(0), and then multiplies logarithm by a real multiplier in register ST(1). Multiplication enables to transform a base 2 logarithm to any arbitrary base. Product overwrites multiplier in register ST(1). Then FYL2X command increments coprocessor's stack pointer by 1, so that registers ST(1-7) are renamed into ST(0-6). That register, where the product has been written, becomes top stack register ST(0). The former ST(0) register is renamed into ST(7) and is announced free, access to its contents is lost. Final result of FYL2X command is expressed by formula ST(0)=ST(1)•log(ST(0)).

Code	Example
D9 F1	FYL2X

7.04-68 FYL2XP1 – sum of series logarithm

The FYL2XP1 command calculates a logarithm of arbitrary base, just as FYL2X command does (7.04-67), but FYL2XP1 command implies, that its argument in coprocessor's top stack register ST(0) is a sum of series, calculated by F2XM1 command (7.04-01). For obtaining high precision of calculations this sum of series must be within limits from (–1 + 1/SQRT2) to (–1 + SQRT2), which correspond to base 2 logarithm values from –1/2 to +1/2. Further multiplication of base 2 logarithm value by a multiplier in stack register ST(1) and incrementation of coprocessor's stack pointer is performed just as it is done by FYL2X command (7.04-67). Calculated logarithm is left in top stack register ST(0). Final result of FYL2XP1 command is expressed by formula ST(0)=ST(1)•log(1+ST(0)).

Code	Example
D9 F9	FYL2XP1

Chapter 8. Selected interrupt handlers

There are occasions, when normal succession of CPU's operations has to be interrupted in order to respond to an urgent request. Interruptions can be initiated by both hardware and software. CPU itself invokes interrupts in case of unexpected errors (exceptions). Other hardware devices send their interrupt requests via interrupt controller lines IRQ 00 – IRQ 15. Programs cause "software" interrupts with INT command (7.03-28). In any case interrupt leads to execution of a certain subroutine – interrupt handler, which performs the requested action.

A large number of interrupt handlers is permanently present in computer's memory since computer is switched on. These handlers can be regarded as a library of standard subroutines. Effectiveness of your programming efforts depends largely on your skill in making use of this library.

Entry point addresses of interrupt handlers are stored in interrupt table. Different interrupt tables are used in CPU's real and protected modes.

Protected mode interrupt table is filled with 8-byte descriptors, specifying access rights and entry point addresses of interrupt handlers and API services. Selection of interrupt handlers, their arrangement and placement of interrupt table in memory are arbitrary regulated by that program or by that operating system, which controls computer's functioning in protected mode.

Contrary to that, real mode interrupt table is strictly institutionalized. In all AT-compatible computers it occupies the same memory area: from 0000:0000h to 0000:03FFh. It is filled not by descriptors, but by 4-byte addresses of interrupt handler's entry points. Unified addresses placement for particular BIOS functions constitutes the basis of software compatibility. The latter feature defines an important role of real mode interrupt table and induces our special interest in it.

After switching CPU to protected mode the real mode interrupt table is preserved intact and still may be addressed. Real mode interrupt handlers may be needed for specific motherboard hardware and for execution of DOS functions, called by programs from "DOS box". For the time of execution of real mode interrupt handlers the protected mode operating systems switch CPU back to real mode. Due to these hidden manipulations DOS programs inside "DOS box" are executed considerably slower, than in real mode.

Offset of any required address in real mode interrupt table is calculated automatically by multiplying interrupt number by 4. Each address is a double word pointer, which should be interpreted in reverse order: fourth and third bytes constitute segment address, second and first bytes constitute entry point offset. For example, dump 59 F8 00 F0 corresponds to handler's entry point address F000:F859h.

Any interrupt request implies that submitted address points at an entry point to handler's executable code. There are exceptions, though: offsets 0074h, 0078h, 007Ch, 0104h, 010Ch, 0118h in real mode interrupt table point at non-executable data tables (A.12-1). Corresponding numbers 1Dh, 1Eh, 1Fh, 41h, 43h, 46h can't be used to enumerate interrupts.

Interrupt table is partially filled by BIOS, generally up to INT 1C. The core of MSDOS7 adds its own pointers (most known INT 20 - INT 2E), and later almost each driver sets double-word (dword) pointer(s) to its own handler(s). At each step any previously loaded pointer may be replaced; some are overwritten more than twice. Thus interrupts may become intercepted by new handlers. Most often interrupt interception is intended to provide conditional access to new functions, otherwise readdressing the call to the former handler. For a large part of interrupts one can't be sure whether their handlers belong to BIOS or to DOS or to something else: it may depend on particular configuration settings in your PC.

Since interrupt handling depends on PC's BIOS and may be changed by software, MS-DOS7 can't be responsible for keeping it strictly defined once and for ever. The validity of results has to be checked after almost any interrupt call. Nevertheless there is a large number of API functions, which are preserved intact for compatibility reasons and almost certainly will be encountered in any PC under MSDOS7. Selected interrupt calls invoking such functions are described below in this chapter.

8.01 Interrupt handlers, loaded by PC's BIOS (INT 00 – INT 1C)

8.01-01 INT 00 – divide by zero error

If after DIV (7.03-21) or IDIV (7.03-24) commands the quotient overflows the result register, then CPU generates a call for INT 00h handler. The default INT 00 handler terminates execution of current program, displays a message: "Your program caused a divide overflow error...", and transfers control to DOS.

Note 1: if a call for INT 00 handler is initiated by CPU, then CPU leaves in stack a return address, pointing not to the next command, but to that division command, which has caused the overflow.

8.01-02 INT 01 – single step interrupt

If trap flag TF (A.11-4) in flags register is set, then CPU calls for INT 01 handler after execution of each command. This feature is used in order to trace execution of programs step-by-step. Besides that, modern CPUs, from model 80386 and on, have internal debugging registers (A.11-5), invoking INT 01 handler each time, when CPU addresses

predefined memory region(s) or predefined port(s). INT 01 handler may also be invoked by execution of undocumented code F1h.

Instead of INT 01 handler's address the PC's BIOS system installs a reference to IRET command (7.03-30), which just returns CPU to execution of the next command. Debugging utilities have to replace this reference in interrupt table by an address of their own handler, enabling to transfer control to that process, which has initiated execution of the program under test (for example, to debugger DEBUG.EXE).

Note 1: trap flag TF is cleared after execution of each command, but its original state (the one before it has been cleared) is saved in stack together with return address just when INT 01 handler is called. Therefore INT 01 handler's code itself is performed while TF flag is cleared, but final handler's command IRET restores that original state of TF flag from stack.

Note 2: return address, saved in stack at the moment INT 01 handler is called, usually points at the next command in a program under test. But when a call for INT 01 handler is invoked by a debugging register, then return address points just to that command, which has met the predefined condition. Flags in debugging register DR6 (A.11-5) enable to discriminate the cause of INT 01 call.

8.01-03 INT 02 – non-maskable interrupt

A call for INT 02 handler is caused by a signal sent to NMI (Non-Maskable Interrupt) CPU's pin. Contrary to other hardware interrupts, a call via NMI pin can't be blocked by CLI command (7.03-12) or by a mask in interrupt controller. INT 02 handler has a special mission: it responds to emergency accidents, for example, to memory failures. For most such accidents INT 02 handler displays an error message and halts further execution, bringing CPU to a standstill. In many PCs a signal sent to NMI pin informs about first symptoms of imminent power supply failure. In such cases INT 02 handler undertakes urgent actions, aimed to prevent data loss, and then returns control to interrupted process: it is given a chance to be resumed, if alarm turns to be false, and power supply failure really wouldn't happen.

Note 1: a call to CPU's NMI pin can be temporary blocked, if a byte with 7-th bit set is sent by OUT command (7.03-66) to CMOS memory port 70h. This operation is implied to be followed by sending another byte to port 71h (note 1 to A.14-1). Blocking of NMI requests for the time of access to CMOS memory prevents distortion of CMOS data.

8.01-04 INT 03 – a breakpoint

When code CCh (7.03-28) is encountered in place of the first byte in machine command, then processor generates a call for INT 03 handler. Code CCh is usually

inserted by debugging utilities in order to stop execution of the program under test at the desired point (breakpoint). In particular, just in this way breakpoints are set by debugger DEBUG.EXE.

Instead of INT 03 handler's address the PC's BIOS system installs a reference to IRET command (7.03-30), which just returns CPU to execution of the next command. Debugging utilities have to replace this reference in interrupt table by an address of their own handler, which is to serve the INT 03 calls.

8.01-05 INT 04 – response to overflow error

Contrary to interrupt INT 00, which compels to respond to overflow errors at once, interrupt INT 04 enables a retarded response to overflows, initiated by INTO command (7.03-29). Having encountered code CEh of INTO command, processor checks the state of overflow flag OF: if it is not cleared, the INT 04 handler is called for. The default INT 04 handler just returns control to the caller program. Each program is given an opportunity to replace the default INT 04 handler with its own INT 04 handler, providing a desirable response to overflow errors.

8.01-06 INT 05 – screen dump and boundaries check

Due to IBM's and Intel's mutually inconsistent technical decisions the INT 05 real mode handler has been charged with two different missions.

In IBM-compatible PCs the BIOS system installs interrupt INT 05 handler, which sends active screen page to printer. Printing procedure is initiated by user's Shift-PrtSc keystroke. Having identified this key combination, INT 09 handler (8.01-09) calls for IBM's INT 05 handler. The latter checks printer's status byte at address 0000:0500h in BIOS data area (A.12-1). States of printer status byte have the following meaning:

00h – printer is connected to LPT1 port and is ready;
01h – printer is busy, previous task isn't finished yet;
FFh – previous request to printer has failed.

If status byte has 00h state, then active screen page is sent to LPT1 port and is printed.

Intel's processors, from model i80286 and on, call for INT 05 handler, if violation of bounds is detected by the BOUND machine command (code 62h). Obviously, this mission of INT 05 handler must be quite different.

The BOUND command is not "known" to DEBUG.EXE and this is why it is not described in chapter 7. In order to avoid conflicts, caused by using the BOUND command in real mode, the concerned program must install its own interrupt handlers, which are able to cope with the problem of call source ambiguity. This problem is not inherent to

protected mode, because interrupt table for protected mode is arranged anew so that INT 05 calls are not charged with screen page printing mission.

Note 1: original INT 05 handler is often replaced by another one, supplied by video card and invoked by PrtSc keystroke (8.01-66).

Note 2: if a call for INT 05 handler is induced by BOUND command, then CPU leaves in stack return address not of the next command, but of the BOUND command itself. Such calls shouldn't be directed to IRET command, because PC will get hanged in endless cycle of calls and returns.

8.01-07 INT 06 – invalid code exception

CPU responds with a call for INT 06 handler to invalid code execution attempts, in particular, to

- protected-mode commands while processor works in real mode;
- applying LOCK prefix to commands, which don't write to memory;
- specifying register for commands, accepting memory operands only;
- command codes, which can't be recognized by CPU.

As far as a call for INT 06 handler is induced by any "unknown" machine code, INT 06 handlers are sometimes used in order to emulate commands of modern CPUs in PCs with obsolete CPU. But the default INT 06 handler just returns control back to the next command in the caller program.

8.01-08 INT 07 – coprocessor service exception

CPU calls for INT 07 handler in response to attempts to execute the ESC command (7.03-22) or any coprocessor's command (7.04), if in control register CR0 (A.11-4) its bit 02h ("Coprocessor emulation") is set. Bit 02h may be set intentionally in order to call for a handler, emulating coprocessor's functions. Some BIOS systems do it automatically in those PCs, which are not equipped with arithmetical coprocessor.

Modern PCs have arithmetical coprocessor integrated in main CPU, so that emulation of coprocessor's functions is not needed. Therefore INT 07 handler can be charged with another mission: undertaking of appropriate measures in response to registration of non-masked exceptions in arithmetical coprocessor. For this purpose a call for INT 07 handler may be induced by WAIT prefix (7.02-05), if in control register CR0 (A.11-4) both bits 01h and 03h are set. In order to ignore the WAIT prefix, bit 01h in CR0 register should be cleared.

Implementation of any INT 07 mission implies loading of appropriate handler. The default INT 07 handler just returns control back to the next command in the caller program.

8.01-09 INT 08 – INT 0F: interrupt requests IRQ 0 – IRQ 7

While CPU is in real mode, the INT 08 – INT 0F group of interrupt handlers responds to requests, sent via IRQ 0 – IRQ 7 lines from various devices to the first interrupt controller. Some IRQ lines have dedicated hardware sources, listed in the fourth column of the following table, but some other IRQ lines are free to receive a request from any device, which is tuned to send requests via one of these lines and is supported by a driver, loading a handler for the corresponding interrupt.

Besides responding to external IRQ requests, some interrupt handlers in the same INT 08 – INT 0F group may be called by CPU in order to cope with certain specific errors (exceptions). Those exceptions, which may induce calls for INT 08 – INT 0F handlers in real mode, are described in notes to the following table. Interrupt controller doesn't register interrupt calls, generated by CPU. If a byte 0Ah is sent by OUT command (7.03-66) into port 20h of interrupt controller, then from the same port 20h the IN command (7.03-26) will be able to read a byte, marking reception of a request via each IRQ line by setting TRUE state of corresponding bit, specified in the third column of the following table. Cleared state of corresponding bit is an evidence that this particular interrupt is initiated by CPU.

If a call for interrupt handler is induced by exception, then CPU leaves in stack return address not of the next command, but of that current command, which has induced the exception call. This gives an opportunity to repeat the operation, if the cause of exception can be removed, but, on the other hand, such calls shouldn't be directed to IRET command, because PC will get hanged in endless cycle of calls and returns. If interrupt handler can't emend the situation, repetition must be prevented either by terminating execution or by correction of return offset in stack. Besides that, while the handler performs its job, reception of concurrent interrupt requests via interrupt controller must be blocked. This can be done by OUT command (7.03-66), sending into port 21h a mask byte with TRUE state of that bit, which corresponds to concurrent IRQ line, as it is shown in third column of the following table.

Interrupt	Line	Mask	Source of requests	Comments
INT 08	IRQ 0	bit 0	System timer	Note *1
INT 09	IRQ 1	bit 1	Keyboard controller	Note *2
INT 0A	IRQ 2	bit 2	2-nd interrupt controller	Note *3
INT 0B	IRQ 3	bit 3		Note *4
INT 0C	IRQ 4	bit 4	Serial port COM 1	Note *5
INT 0D	IRQ 5	bit 5		Note *6
INT 0E	IRQ 6	bit 6	Floppy drive controller	Note *7
INT 0F	IRQ 7	bit 7	Parallel port LPT 1	

Note 1: calls for INT 08 handler proceed regularly 18.2 times per second in order to sustain time counting. In its turn, INT 08 handler invokes INT 1C (8.01-96), giving an opportunity to intercept the latter to application programs. INT 08 handler also can be called by CPU in case of "double fault" exception, which usually leads to reboot.

Note 2: the INT 09 handler senses each keystroke, controls keyboard buffer and prepares codes, shown in appendix A.02-1 and presented to application programs via INT 16. But INT 09 handler responds with explicit immediate actions only to a few key combinations, listed in article 1.01.

Note 3: the second interrupt controller accepts interrupt requests via lines IRQ 8 – IRQ 15 and induces calls to INT 70 – INT 77 handlers (8.03-75).

Note 4: most probable source of interrupt requests for the IRQ 3 line is second serial port COM 2 (if it exists).

Note 5: the INT 0C handler also can be called by CPU in case of "stack overflow" exception, when stack grows beyond its predefined limit.

Note 6: the INT 0D handler can be called by CPU in case of segment limits violation exception, i.e. attempt to address code or data beyond predefined segment limits.

Note 7: a call for INT 0E handler may be initiated by CPU in response to an attempt to address "closed" memory pages (those, which can't be found in CPU's TLB buffer).

Note 8: mapping of external requests IRQ 0 – IRQ 7 onto calls for INT 08 – INT 0F handlers can be changed by reprogramming interrupt controller, so that external requests wouldn't invoke those handlers, which are designed to cope with CPU's exceptions. Such reprogramming is usually performed in course of preparation to switching CPU into protected mode in coordination with formation of a new interrupt table for protected mode.

8.01-10 INT 10\AH=00h – setting of a videomode

Prepare:

AH = 00h
AL – code of videomode to be set (A.10-1)

On return:

AL contents may be altered

Note 1: this function switches all video cards, including modern ones, to videomodes, compatible with obsolete VGA video cards. Modern SVGA videomodes should be set by function INT 10\AX=4F02h.

Note 2: code of current videomode is written in BIOS data area (A.10-6) at address 0040:0049h.

Note 3: videomode switching coordinated with mouse pointing device parameters switching may be performed by mouse driver, called via interrupt INT 33\AX=0028h.

Note 4: switching of videomodes causes screen blanking (darkening) for up to 2 seconds, discomfortable for visual perception. Therefore excessive switching of videomodes should be avoided. For screen clearing a call INT 10\AH=06h (8.01-15) should be preferred.

8.01-11 INT 10\AH=01h – cursor's size in textual videomodes

Prepare:

AH	= 01h	
CH	– bits 7, 6, 5:	– cleared to zero
	– bits 0 – 4:	– cursor's topmost line number
CL	– bits 7, 6, 5:	– cleared to zero
	– bits 0 – 4	– cursor's bottom line number

Note 1: lines are counted within current font height from top downwards. Current numbers of topmost and bottom cursor's lines are written into BIOS data area (A.10-6) at address 0040:0060h. Current font height may be determined via a call for INT 10\AX=1130h handler or may be directly read from 0040:0085h memory cell.

Note 2: setting bit 5 in CH register to TRUE state makes cursor invisible.

8.01-12 INT 10\AH=02h – set cursor's position

Prepare:

AH	= 02h
BH	– screen page number (notes 1 and 2 to INT 10\AH=05h)
DH	– row number (counted from 00h – the topmost row)
DL	– column number (counted from 00h – the leftmost column)

Note 1: cursor's position is defined separately for each screen page.

Note 2: pairs of cursor's coordinates for up to 8 screen pages are written into BIOS data area at addresses from 0040:0050h and on (A.10-6).

8.01-13 INT 10\AH=03h – determination of cursor's size and position

Prepare:

AH	= 03h
BH	– screen page number (notes 1 and 2 to INT 10\AH=05h)

On return:

CH	– cursor's size topmost line number
CL	– cursor's size bottom line number
DH	– row number (counted from 00h – the topmost row)
DL	– column number (counted from 00h – the leftmost column)

Note 1: some BIOS versions return zero in AX register.

8.01-14 INT 10\AH=05h – selection of active screen page

Prepare:

AH = 05h
AL – requested screen page number

Note 1: if requested page number is greater than maximum allowable page number for current videomode, then selection will not be confirmed by INT 10\AH=0Fh function. Maximum allowable page number for VGA videomodes is shown in a table, returned by INT 10\AX=1B00h function (A.10-2, offset 29h), and for SVGA videomodes – in a table, returned by INT 10\AX=4F01h function (A.10-7, offset 1Dh).

Note 2: screen pages are numerated from 00h and on, so that maximum allowable page number is by 1 less than total number of screen pages. The most popular textual videomode 03h provides 4 screen pages numerated from 00h to 03h.

Note 3: selection of a screen page, coordinated with mouse cursor switching to the same page, may be performed by mouse driver, called via interrupt INT 33\AX=001Dh (8.03-47).

8.01-15 INT 10\AH=06h-07h – scrolling over screen window

The term "scrolling" denotes moving the displayed contents up or down within the whole screen or inside a "window", i.e. a rectangular region, constituting a part of the whole screen. The lines, appearing from beneath the "window's" border, have no contents and are just filled with prescribed color. Being given AL = 00h, scrolling procedure enables to clear and to fill with uniform color the whole rectangular "window" or the whole screen.

Prepare:

AH – scrolling direction:
= 06h – to scroll up,
= 07h – to scroll down
AL – number of lines to scroll (or 00h – clear the whole window)
BH – color to fill new lines:
bits 0 – 3: – clear to zero,
bits 4 – 7: – color code from column 1 of table A.10-5.
CH – row, CL – column of window's upper left corner
DH – row, DL – column of window's lower right corner

Note 1: scrolling procedure is valid exclusively for textual videomodes.

Note 2: scrolling procedure affects active screen page only, other screen pages are preserved intact.
Note 3: on return from scrolling procedure some BIOS versions may alter contents of BP or DS register.

8.01-16 INT 10\AH=08h – read a character at cursor's position

Prepare:

AH = 08h
BH – screen page number (notes 1 and 2 to INT 10\AH=05h)

On return:

AH – color byte as in table A.10-5 (in textual videomodes only)
AL – ASCII code of the read character

Note 1: in graphic videomodes, only those characters drawn with white foreground pixels can be recognized properly. If character can't be recognized, then AL = 00h is returned.

8.01-17 INT 10\AH=09h-0Ah – character display at cursor's position

Prepare:

AH = 09h
AL – ASCII code of the character to be displayed
BH – screen page number (notes 1 and 2 to INT 10\AH=05h)
BL – color byte, shown in appendix A.10-5.
CX – number of times to repeat writing of the character

Note 1: all symbols are displayed, including 0Dh (CR), 0Ah (LF), 08h (BS) and other special symbols, mentioned in appendix A.02-8.
Note 2: in graphic videomodes with less than 256 colors, if the TRUE state of bit 7 in BL register is set, then specified character is written into video memory by means of XOR logical operation.
Note 3: in graphic videomodes the prescribed number of repetitions in CX register must not exceed the number of character positions in the same row to the right of current cursor's position.
Note 4: the INT 10\AH=0Ah handler displays character(s) in the same way, but ignores color byte in BL. The displayed character(s) will have the same color as all the previous screen contents.
Note 5: cursor's position is not shifted by display operation and doesn't depend on number of repetitions, specified in CX register.
Note 6: for 256-color graphic videomodes (for example, in videomode 13h) BH register must specify background color byte, and BL register must specify foreground color byte.

8.01-18 INT 10\AH=0Bh – background or border color

For graphic videomodes this function defines background color, but for textual videomodes it defines the border color.

Prepare:

AH = 0Bh
BX – bits 3 – 15: – cleared to zero
– bits 2, 1, 0: – correspond to red, green and blue color

Note 1: many modern LCD displays either can't show screen border properly or don't show it at all.

8.01-19 INT 10\AH=0Ch – painting of a dot

Prepare:

AH = 0Ch
AL – pixel's color byte (A.10-5)
BH – screen page number (notes 1 and 2 to INT 10\AH=05h)
CX – pixel's column number
DX – pixel's row number

Note 1: dot painting function is valid for graphic videomodes only.
Note 2: in graphic videomodes with less than 256 colors, if the TRUE state of bit 7 in AL register is set, then dot is written into video memory by means of XOR logical operation.
Note 3: BH contents is ignored, if active videomode supports one screen page only.
Note 4: the INT 10\AH=0Ch function is convenient for drawing lines, but it is too slow for filling screen areas. For the latter purpose more fast direct writing into video memory (8.01-39) should be preferred.

8.01-20 INT 10\AH=0Dh – read pixel's color

Prepare:

AH = 0Dh
BH – screen page number (notes 1 and 2 to INT 10\AH=05h)
CX – pixel's column number
DX – pixel's row number

On return:

AL – color byte (A.10-5)

Note 1: the INT 10\AH=0Dh function is valid for graphic video modes only.
Note 2: BH contents is ignored, if active videomode supports one page only.

8.01-21 INT 10\AH=0Eh – characters display in teletype manner

The INT 10\AH=0Eh function displays a character at current cursor's position and then shifts cursor to the next character cell. If there is no free space in the current row, cursor is transferred to the start of next row.

Prepare:

AH = 0Eh
AL – ASCII code of the character to be displayed
BH – screen page number (notes 1 and 2 to INT 10\AH=05h)
BL – foreground color (in graphic videomodes only)

Note 1: special codes, listed in appendix A.02-8, including 07h (BEL) and 08h (BS), are not displayed, but are executed as commands.
Note 2: in textual videomodes the displayed character inherits the color, which has been specified for the preceding character cells.

8.01-22 INT 10\AH=0Fh – determination of current videomode

Prepare:

AH = 0Fh

On return:

AH – number of character columns in a row or in a line
AL – code of current videomode (A.10-1)
BH – active screen page number (notes 1 and 2 to INT 10\AH=05h)

Note 1: if code of videomode was originally specified with its 7-th bit set to TRUE state (screen not cleared), then the returned code will have its 7-th bit set to TRUE state too.
Note 2: codes of SVGA videomodes can't be determined by INT 10\AH=0Fh function properly: for textual SVGA videomodes it returns either AL=07h (monochrome videomode) or AL=03h (color videomode).

8.01-23 INT 10\AX=1003h – switching of 7-th bit mission

In textual videomodes the 7-th bit in color byte (A.10-5) may define either blinking or background intensity: it depends on state of control bit, which can be changed by INT 10\AX=1003h function.

Prepare:

AX = 1003h
BX – type of operation:
= 0000h – enable background intensity control;
= 0001h – enable foreground blinking control.

Note 1: the state of control bit is expressed by 5-th bit in a byte at address 0040:0065h in BIOS data area (A.10-6), and also by 5-th bit in a byte at offset 2Dh in data block (A.10-2), returned by INT 10\AX=1B00h function.

8.01-24 INT 10\AX=1010h – setting of color intensities

The INT 10\AX=1010h function writes into one of DAC's registers the desired partial intensities (from 00h to 3Fh) of main colors – red, green and blue. Combination of partial intensities results in that color tone, which should be defined by this particular DAC's register.

Prepare:

AX = 1010h
BX – target DAC's register number
CH – partial intensity of green color
CL – partial intensity of blue color
DH – partial intensity of red color

Note 1: all DAC's registers are available for writing, but not all of them are active: it depends on current videomode. In particular, background color palette in 16-color videomodes is defined by DAC's registers 0000h – 0007h.

8.01-25 INT 10\AX=1015h – reading of color intensities

The INT 10\AX=1015h function reads partial intensities of main colors – red, green and blue, stored in one of DAC's registers. Combination of these partial intensities results in that color tone, which is defined by this particular DAC's register.

Prepare:

AX = 1015h
BX – target DAC's register number

On return:

value in AX register may be altered;
CH – partial intensity of green color;
CL – partial intensity of blue color;
DH – partial intensity of red color.

8.01-26 INT 10\AX=1018h – setting of color mask

Prepare:

AX = 1018h
BL – new mask to be set

Note 1: in a mask bits 0 – 2 switch on blue, green and red colors of background. Bits 3 – 5 do the same for foreground colors. States of bits 6 and 7 are indifferent. Normal state of color mask is represented by byte FFh: all colors are switched on.
Note 2: the CLS command (3.05) doesn't reset color mask to its normal state.

8.01-27 INT 10\AX=1100h – font loading for textual videomodes

Prepare:

AX = 1100h
BH – number of bytes per each character pattern
BL – identifier of target memory block (note 1 to 8.01-28)
CX – number of character patterns to be loaded or replaced
DX – offset for loading, counted from the start of target memory block
ES:BP – pointer to table of patterns which is to be loaded

Note 1: it is implied, that the whole font table contains FFh character patterns.
Note 2: each byte in a character pattern represents one screen line, hence the number of bytes in a pattern (the value in BH register) is the same as number of screen lines in font's height.
Note 3: this function sets textual video mode, corresponding to the loaded font, but video buffer is not cleared.
Note 4: if several fonts are to be loaded, then the same number of character generator's memory blocks must be prepared beforehand by DISPLAY.SYS driver (5.02-02). Otherwise only one default memory block with identifier 00h will be available.
Note 5: the INT 10\AX=1110 function also loads a font for textual videomodes and accepts the same specifications, but recalculates current state of video controller. A call for INT 10\AX=1110 must be preceded by setting a textual video mode with active video page 00h.

8.01-28 INT 10\AX=1103h – switching between loaded fonts

The INT 10\AX=1103 function switches character generator to another font, which must be loaded beforehand into one of character generator's memory blocks. Character generators in EGA-compatible and in VGA-compatible videocards are able to keep active two memory blocks simultaneously, thus giving an opportunity to show characters of two fonts. Selection of a font for each character depends on bit 3 in color byte (A.10-5), which is accepted by character displaying functions, in particular, by INT 10\AH=09h and by INT 10\AH=0Eh functions, accepting color byte from BL register.

Prepare:

AX = 1103h
BL – identifier of selected memory block (see note 1 below)

Note 1: identifier of character generator's memory block is a byte with two dedicated fields. One field is represented by bits 4, 1, 0. The other field is represented by bits 5, 3, 2. In each field a number of memory block (from 0 to 7) should be written. A font in this memory block must be loaded yet. If numbers of memory blocks in both fields are equal, then one font will be addressed, and then bit 3 in color byte (A.10-5) will define intensity. In particular, in EGA-compatible video cards, allowing not more than 4 fonts, their memory blocks may be addressed by identifiers 00h, 05h, 0Ah, 0Fh.

Note 2: the possibility to keep active two fonts simultaneously is indicated by the value of 9-th byte in static functionality table (A.10-3). Address of that table is reported by INT 10\AX=1B00h function.

Note 3: in order to activate two fonts simultaneously, in two fields of identifier byte different memory block numbers should be stored. Those characters having bit 3 in color byte cleared will be selected from that memory block, whose number is stored in the first identifier's field (bits 4, 1, 0). Those characters having bit 3 in color byte set will be selected from that memory block, whose number is stored in the second identifier's field (bits 5, 3, 2).

8.01-29 INT 10\AX=1104h – loading of standard 8x16 font

The INT 10\AX=1104 function loads into character generator's memory block that BIOS's default 8x16 font, which represents american codepage CP437. At the same time textual videomode 03h is set, because its format (80x25) corresponds to 8x16 font.

Prepare:

AX = 1104h
BL – identifier of selected memory block (note 1 to 8.01-28)

Note 1: the INT 10\AX=1114 function also loads BIOS's standard 8x16 font for textual videomode 03h and accepts the same specifications, but recalculates current state of video controller. A call for INT 10\AX=1114 must be preceded by setting videomode 03h with active video page 00h.

Note 2: if several fonts are to be loaded, then the same number of character generator's memory blocks must be prepared beforehand by DISPLAY.SYS driver (5.02-02). Otherwise only one default memory block with identifier 00h will be available.

Note 3: 8x8 and 8x14 CP437 fonts also can be loaded from BIOS's read-only memory. The 8x8 font is loaded similarly by INT 10\AX=1102h and by INT 10\AX=1112h functions. Monochrome 8x14 font is loaded similarly by INT 10\AX=1101h and by INT 10\AX=1111h functions. The calls for INT 10\AX=1111h and INT 10\AX=1112h functions cause recalculation of video controller's current state and must be preceded by setting of adequate textual videomode.

8.01-30 INT 10\AX=1121h – font loading for graphic videomodes

Prepare:

AX = 1121h
BL – character rows number specification:
= 01h – 14 rows,
= 02h – 25 rows,
= 03h – 43 rows,
= 00h – number of rows is defined via DL register.
CX – number of bytes per character pattern
DL – number of rows, if BL = 00h (otherwise DL is ignored)
ES:BP – pointer to table of character patterns to be loaded

Note 1: a call for INT 10\AX=1121h function must be immediately preceded by setting (or resetting) of graphic videomode.

Note 2: fonts for graphic videomodes don't require preparation of character generator's memory blocks. Instead of that a pointer to the loaded font must be written into interrupt table at address 0000:010Ch. This pointer is sometimes referred to as vector INT 43.

Note 3: user-defined patterns for characters 80h – FFh for BIOS's 8x8 default font can be loaded similarly by INT 10\AX=1120h. As far as format of this font is known, contents of BL, CX and DL registers are ignored, and font table has a fixed length 400h. A pointer to loaded font must be written into interrupt table at address 0000:007Ch. This pointer is sometimes referred to as vector INT 1F.

8.01-31 INT 10\AX=1124h – loading of standard graphic font

The INT 10\AX=1124h function loads BIOS's standard 8x16 font, representing american codepage CP437 for graphic videomodes. A pointer to that font is written into interrupt table at address 0000:010Ch. This pointer is sometimes referred to as vector INT 43.

Prepare:

AX = 1124h

Note 1: a call for INT 10\AX=1124h function must be immediately preceded by setting (or resetting) of graphic videomode.

Note 2: 8x14 font from BIOS's read-only memory can be loaded similarly by INT 10\AX=1122h function.

Note 3: double-dot 8x8 font from BIOS's read-only memory can be loaded similarly by INT 10\AX=1123h function.

8.01-32 INT 10\AX=1130h – get information about font

Prepare:

AX = 1130h
BH – requested pointer to font table:
= 00h – pointer to 8x8 graphic font (from 0000:007Ch)
= 01h – pointer to current graphic font (from 0000:010Ch)
= 02h – pointer to BIOS's 8x14 textual font (CP437)
= 03h – pointer to BIOS's 8x8 font, characters 00h – 7Fh
= 04h – pointer to BIOS's 8x8 font, characters 80h – FFh
= 05h – pointer to BIOS's alternate 9x14 font
= 06h – pointer to BIOS's standard 8x16 textual font (CP437)
= 07h – pointer to BIOS's alternate 9x16 font

On return:

ES:BP – pointer to the first byte of requested font table
CX – number of bytes per character for the current font
DL – highest character row on the screen for the current font

Note 1: data returned in CX and DL registers relate not to the requested font, but to that one which is currently displayed on the screen.

8.01-33 INT 10\AH=13h – display of a characters string

The INT 10\AX=13h function displays a string of characters, specified either as a succession of ASCII bytes or as a video memory string in textual videomodes, where each ASCII byte is followed by color byte (A.10-5). In the latter case a value in BL register is ignored.

Prepare:

AH = 13h
AL – bits 7 – 2: – cleared to zero
– bit 1 set – string with alternating ASCII and color bytes
– bit 0 set – shift cursor along the displayed characters
BH – screen page number (notes 1 and 2 to INT 10\AH=05h)
BL – color byte (A.10-5), if string consists of ASCII codes only
CX – number of characters in the string to be displayed
DH – character's row to start display
DL – character's column to start display
ES:BP – pointer to first byte of the string to be displayed

Note 1: special codes ASCII, listed in appendix A.02-8, are not displayed, but are executed as commands.

Note 2: some obsolete models of videocards can't execute backspace (BS) and carriage return (CR) special codes properly, if character string is directed to currently inactive screen page.

8.01-34 INT 10\AX=1B00h – get video status information.

Prepare:

AX = 1B00h
BX = 0000h
ES:DI – pointer to 64-byte buffer for data table

On return:

AL = 1Bh, if this function is supported by BIOS
ES:DI – pointer to the first byte of table with video status data. Table's contents are described in appendix A.10-2.

8.01-35 INT 10\AX=4F00h – information about BIOS's VBE extensions

The INT 10\AX=4F00h function reports about available video BIOS extensions and about supported videomodes. If video BIOS extensions are not present in a particular computer, then all INT 10\AX=4Fxxh functions in this computer are not supported.

Prepare:

AX = 4F00h
ES:DI – pointer to a 512-byte buffer for data table

On return:

AL = 4Fh – any other value signifies absence of VBE
AH = 00h – successful termination, data table is written;
= 01h – table writing attempt failed;
= 02h – function is not supported by hardware configuration;
= 03h – function is invalid in current videomode.
ES:DI – pointer to the first byte of returned VBE data table (A.10-4).

8.01-36 INT 10\AX=4F01h – information about SVGA videomode

Prepare:

AX = 4F01h
CX – code of SVGA videomode
ES:DI – pointer to a 256-byte buffer for data table

On return:

AL = 4Fh – any other value signifies that function is not supported
AH = 00h – successful termination
= 01h – operation failed.
ES:DI – pointer to returned videomode data table (A.10-7).

8.01-37 INT 10\AX=4F02h–4F03h – set/get SVGA videomode

Prepare:

AX = 4F02h – set videomode anew;
= 4F03h – get code of current videomode
BX – for AX = 4F02h only: code of videomode (A.10-1), including
bit 14 set: – enable linear frame buffer
bit 15 set: – don't clear video memory

On return:

AL = 4Fh – any other value signifies that function is not supported
AH – termination code, as after INT 10\AX=4F01h (8.01-36)
BX – code of current videomode (A.10-1), including
bit 14 set: – linear frame buffer enabled
bit 15 set: – contents of video memory are preserved

Note 1: switching of videomode, coordinated with switching of mouse control parameters, can be performed by mouse driver, if it is called via interrupt INT 33\AX=0028h.

Note 2: display devices don't necessarily support all SVGA videomodes. Before switching to any videomode you have to know, whether this particular videomode is supported by your display device.

Note 3: videomode switching causes screen blanking (darkening) for up to 2 seconds, discomfortable for visual perception.

8.01-38 INT 10\AX=4F04h – save/restore SVGA videomode

Prepare:

AX = 4F04h
CX – part of configuration saved (or to be saved):
= 0001h – video hardware state (bit 0 in CX is set)
= 0002h – BIOS data (bit 1 in CX is set)
= 0004h – color registers and DACs (bit 2 in CX is set)
= 0008h – SVGA state (bit 3 in CX is set)
= 000Fh – whole video configuration (bits 0 – 3 are set)
DL –subfunction:
= 00h – determination of buffer's size for saving videomode
= 01h – save current state of video controller
= 02h – restore former state of video controller
ES:BX – pointer to the first byte of prepared buffer (for subfunctions 01h and 02h only, for subfunction 02h it must be filled with data).

On return from subfunction DL = 00h:

BX – required number of 64-byte memory blocks

On return from :subfunction DL = 01h:

ES:BX – pointer to buffer with stored videomode data

8.01-39 INT 10\AX=4F05h – control over "windows" into video memory

In obsolete computers video memory occupied a part of address space region A000:0000 – B000:FFFFh. Active regions of address space for EGA-compatible and for VGA-compatible videomodes are shown in table A.10-1. But modern video cards are designed for SVGA videomodes, which require much larger memory. The whole devoted address space region is too narrow for SVGA videomodes. Therefore SVGA BIOS arranges sliding "windows", enabling access to specified areas of video card's large memory via address space A000:0000 – B000:FFFFh.

Quantity, size and place of sliding "windows" in address space may depend on both video card and selected videomode. These data for particular video card and videomode can be found out in table A.10-7, returned by function INT 10\AX=4F01h (8.01-36). Most probably one or two 64-kbyte "windows" are arranged: "window A" A000:0000h – A000:FFFFh and "window B" B000:0000h – B000:FFFFh. The INT 10\AX=4F05h function reports current mapping of video memory onto specified address space "window" and enables to change it.

Prepare:

AX = 4F05h
BH = 00h – set start point of mapped area in video memory
= 01h – get start point of mapped area in video memory
BL = 00h – request for window "A"
= 01h – request for window "B"
DX – pointer to mapped area in video memory (for BH = 00h only)

On return:

AL = 4Fh – any other value signifies that function is not supported
AH – termination code, as after INT 10\AX=4F01h (8.01-36)
DX – pointer to mapped area in video memory (after BH = 01h only)

Note 1: position of mapped area in video memory is expressed in granularity units. Size of one granularity unit is not fixed, but it is given in kilobytes in a word at offset 04h in table A.10-7, returned by INT 10\AX=4F01h function (8.01-36).

Note 2: the INT 10\AX=4F05h function can also be called for by CALL FAR command (7.03-08) with address, given in double word at offset 0Ch in table A.10-7.

Note 3: interpretation of data, sent to video memory via the "window", depends on model type, specified by byte 1Bh in table A.10-7. Besides that, some model types allow variants of interpretation. For example, graphic EGA model implements 3 interpretation modes:

mode 00h – 8 pixels per byte – for overwriting pixel values according to bit mask and color mask;
mode 01h for copying from one video memory address into another, taking into account the addresses only;

mode 02h for filling 8 successive pixels with color, defined by 4 least significant bits of the sent byte.

Some opportunities to alter model type and interpretation mode are described in notes 3 and 4 to table A.14-1.

8.01-40 INT 10\AX=4F06h – logical length of displayed line

The INT 10\AX=4F06h function enables to set logical length of displayed line equal to integer power of 2 or to its multiple. Thus some affordable loss in effective video memory capacity is exchanged for a considerable gain in simplicity and speed of coordinates calculations. This function is equally feasible in both textual and graphic videomodes.

Prepare:

AX = 4F06h
BL = 00h – set logical line's length in pixels
= 01h – get actual length of displayed line
= 02h – set logical line's length in bytes
= 03h – get maximal length of displayed line
CX – line's length (for BL=00h and BL=02h only)

On return:

AL = 4Fh – any other value signifies that function is not supported
AH – termination code, as after INT 10\AX=4F01h (8.01-36)
BX – logical line's length in bytes
CX – logical line's length in pixels
DX – maximum available number of screen lines of specified length.

Note 1: actual line's length may exceed nominal value, but can't exceed maximum value, specific for particular videocard. If requested line length is less than nominal line length for the current videomode, then actual line length may be given the nearest acceptable value, not necessarily equal to the requested line length.

8.01-41 INT 10\AX=4F07h – control over displayed part of video memory

The INT 10\AX=4F07h function enables to implement screen scrolling and also switching to another screen page within available space of video memory. This function is equally feasible in both textual and graphic videomodes.

Prepare:

AX = 4F07h
BX = 0000h – set new start position of displayed part immediately
= 0001h – get start position of displayed part
= 0080h – set new start position during field retrace interval
CX – number of the leftmost pixel in a line (not needed for BX = 0001h)
DX – number of the first displayed line (not needed for BX = 0001h)

On return:

AL = 4Fh – any other value signifies that function is not supported
AH – termination code, as after INT 10\AX=4F01h (8.01-36)
CX – number of the leftmost pixel in a line (after BX = 0001h only)
DX – number of the first displayed line (after BX = 0001h only).

8.01-42 INT 11 – get equipment list

Prepare: nothing

On return:

AX – equipment list word. Explanation is given in appendix A.11-1.

Note 1: while functioning in protected mode or in V86 mode, CPUs may generate calls for INT 11 handler in case of misalignment exception, that is being given operand's address not on a multiple of operand's length. Misalignment monitoring is performed at the third (the lowest) privilege level only, if bit 12h in flags register and bit 12h in control register CR0 are both set to TRUE state (note 6 to A.11-4). Programs using misalignment monitoring must intercept calls for INT 11 handler in interrupt table for protected mode.

8.01-43 INT 12 – size of conventional RAM below 1 Mb

Prepare: nothing

On return:

AX – size of conventional RAM in kilobytes (notes 2 and 3)

Note 1: INT 12 handler reads size of conventional RAM from a word at address 0040:0013h in BIOS data area (A.01-1). Besides this, RAM size is stored in CMOS RAM cells 15h and 16h (note 1 to A.14-1).
Note 2: INT 12 handler doesn't inform about PC's RAM beyond 1 Mb boundary, but these data are reported by functions INT 15\AH=88h, INT15\AX=E801h and INT15\AX=E820h.
Note 3: whole RAM size includes that memory, which is not free or is not accessible for 16-bit addressing. Size of free conventional RAM, which can be allotted by DOS to programs, is reported by INT 21\AH=48h function (note 1 to 8.02-50).

8.01-44 INT 13\AH=00h\0Dh – disk controller reset

Reset operation forces disk controller to fill its internal registers anew with data read from a table of parameters for specified disk drive (A.08-2, A.13-1). Reset operation must

follow each failure of access to a HDD or to a floppy disk, and only after reset the access attempt may be repeated.

Prepare:

AH = 00h – apply to floppy disk controller
= 0Dh – apply to HDD controller
DL – disk drive number (note 1)

On return:

On error CF flag is set, AH returns error code (A.06-1)
Clear state of CF flag signifies successful termination.

Note 1: numeration of floppy disk drives is zero-based: 00h – first, 01h – second, and so on. Numeration of HDDs starts from 80h: 80h – first, 81h – second, and so on. Numbers of disk drives (physical disks) have no relation with letter-names of logical disks: each physical HDD may comprise several logical disks.

Note 2: if two disk drives are connected to one controller, then reset operation causes head shift to zero track (recalibration) in both disk drives. When controller's reset is not needed, recalibration should be initiated by a call for INT 13\AH=11h function (all other specifications are the same).

8.01-45 INT 13\AH=01h – status of the last disk operation

Prepare:

AX = 0100h
DL – disk drive number (note 1 to 8.01-44)

On return:

AH – status code (A.06-1)

Note 1: some obsolete BIOS versions return status code in AL.

8.01-46 INT 13\AH=02h – read disk sector(s) into memory

Prepare:

AH = 02h
AL – number of sectors to be read (note 1)
CH – 8 less significant bits of cylinder (track) number
CL – bits 0-5: start sector number (from 1 to 63),
– bits 6-7: two most significant bits of cylinder (track) number
DH – head number (note 2)
DL – disk drive number (note 1 to 8.01-44)
ES:BX – pointer to buffer for the data to be read

On return:

On error CF flag is set, AH returns error code (A.06-1).

Clear state of CF flag signifies successful termination.
AL – number of sectors which actually have been read (note 5)
ES:BX – pointer to buffer filled with data read from disk.

Note 1: number of sectors specified in AL register must not be zero. Some obsolete BIOS versions don't allow the value in AL exceeding the number of remaining sectors on the current track (more about that – in article 4.22).
Note 2: some obsolete BIOS versions accept only 4 lower bits in DH, thus allowing to specify not more than 16 heads. Modern BIOS systems, marked with signature A0h in data block A.13-1, accept transformed CHS parameters, allowing up to 256 heads. In any case the utmost parameter's values for any particular disk drive are reported by INT 13\AH=08h function (8.01-49).
Note 3: when reading from a floppy disk fails, after that reading attempts should be repeated at least twice with applying reset (8.01-44) to floppy controller before each next attempt.
Note 4: under Windows OS direct access to disk via INT 13 functions is prohibited unless the addressed disk is locked for concurrent access by LOCK operation (note 2 to INT 21\AX=440Dh).
Note 5: some obsolete BIOS versions don't return in AL register the number of sectors, which actually have been read. In case of reading error (error code AH = 11h) modern BIOS systems return in AL register the length of corrected data block.
Note 6: the INT 13\AH=0Ah function (all other specifications are the same) reads HDD's sector(s) along with 22-byte "tail", containing from 4 to 7 bytes of error correcting code. INT 13\AH=0Ah function doesn't correct errors and stops reading after having encountered the first damaged sector.

8.01-47 INT 13\AH=03h – write data into disk's sector(s).

Prepare:
AH = 03h
AL – number of sectors to be written (must be nonzero)
CH, CL, DH, DL – just the same as for INT 13\AH=02h (8.01-46)
ES:BX – pointer to buffer with data to be written

On return:
On error CF flag is set, AH returns error code (A.06-1).
Clear state of CF flag signifies successful termination.
AL – number of sectors which actually have been written

Note 1: notes 1 – 4 to INT 13\AH=02h function (8.01-46) are equally applicable to INT 13\AH=03h function.
Note 2: written data may be verified against buffer contents by INT 13\AH=04h function (all other specifications are the same).

Note 3: sectors with ECC (error correcting code) can be written onto a HDD by INT 13\AH=0Bh function (all other specifications are the same).

8.01-48 INT 13\AH=05h – low-level formatting of a track

Low-level formatting may be applied to those media only, which have no intrinsic track structure. Hence INT 13\AH=05h function can be applied to floppies, but shouldn't be applied to modern HDDs: their original track structure may be damaged by low-level formatting.

Table of parameters for formatting must be prepared in PC's memory beforehand by INT 10\AH=18h function (8.01-54). For obsolete floppy types similar table is prepared by INT 10\AH=17h function. In particular, a pointer to a table with formatting parameters for floppies is stored in 0000:0078h memory cell (A.08-2).

Prepare:

AH = 05h
AL – number of sectors to be formatted
CH – track number
DH – head number
DL – disk drive number (note 1 to 8.01-44)
ES:BX – pointer to data buffer, containing 4 bytes for each sector in a track: first – track number, second – head number, third – sector number, fourth – sector size (00h, 01h, 02h, 03h correspond to size 128, 256, 512, 1024 bytes per sector).

On return:

On error CF flag is set, AH returns error code (A.06-1).
Clear state of CF flag signifies successful termination.

Note 1: in data block, pointed at by ES:DX, count of tracks and heads is zero-based, count of sectors is unity-based.

Note 2: the utmost values of sector number, head number and track number should be determined by INT 13\AH=08h function (8.01-49). Though physical values may be different, but the returned utmost values reflect those transformations (A.13-1), which may be applied to disk parameters by BIOS system of a particular computer.

8.01-49 INT 13\AH=08h – determination of drive's parameters

Prepare:

AH = 08h
DL – disk drive number (note 1 to 8.01-44)

On return:

On error CF flag is set, AH returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then:
BL – drive type (for removable media drives only):
= 01h – drive for 360-kb diskettes,
= 02h – drive for 1.2-Mb diskettes,
= 03h – drive for 720-kb diskettes,
= 04h – drive for 1.44-Mb diskettes,
= 06h – drive for 2.88-Mb diskettes,
= 10h – drive of any other type.
CH – 8 less significant bits of maximum cylinder (track) number
CL – bits 0-5: maximum sector number (from 1 to 63),
– bits 6-7: most significant bits of maximum cylinder number
DH – maximum head number
DL – number of attached drives of the same type (note 2)
ES:DI – pointer to parameters table A.08-2 (returned for floppy drives only).

Note 1: status of successful termination (AH = 00h) may be returned even if the requested drive doesn't exist. In order to be convinced in validity of the returned data, one has to check the state of CF flag and the number, returned in DL register.

Note 2: some BIOS systems can't return in DL register a number greater than 2. Suspicion in presence of more drives should be checked separately by INT 13\AH=15h function (8.01-52).

Note 3: signature A0h in a byte at offset 03h in table A.13-1 signifies, that for HDDs the INT 13\AH=08h function returns not physical, but transformed CHS parameters. In this case just these transformed CHS parameters should be taken into account in calculations of requested values for functions INT 13\AH=02h – INT 13\AH=18h (8.01-46 – 8.01-54).

8.01-50 INT 13\AH=0Ch – move disk drive's head to desired track

Prepare:
AH = 0Ch
CH, CL, DH, DL – just the same as for INT 13\AH=02h (8.01-46)
On return:
On error CF flag is set, AH returns error code (A.06-1).
Clear state of CF flag signifies successful termination.

8.01-51 INT 13\AH=10h – check whether HDD is ready

The status code, returned by INT 13\AH=10h function, signifies whether the requested HDD exists and whether it is ready to perform the next task.

Prepare:

AH = 10h
DL – drive number (80h = first HDD, 81h = second HDD, and so on)

On return:

AH – status byte (table A.06-1).
Clear state of CF flag signifies successful termination.
Set state of CF flag signifies an error.

8.01-52 INT 13\AH=15h – disk drive type check

Prepare:

AH = 15h
DL – disk drive number (note 1 to 8.01-44)

On return:

On error CF flag is set, AH returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then:
AH – disk drive type:
= 00h – requested drive doesn't exist;
= 01h – floppy drive without media change line support;
= 02h – any drive with removable media change line support
= 03h – fixed disk drive (HDD).
CX:DX – 4-byte number of 512-byte sectors (returned for HDDs only).

Note 1: HDDs with removable media may be ranked to type 01h or 02h.
Note 2: the INT 13\AH=15h function doesn't rely upon stored data, it scans controller's bus in order to get valid data anew.

8.01-53 INT 13\AH=16h – media change detection

Prepare:

AH = 16h
DL – disk drive number (note 1 to 8.01-44)

On return:

if disk has not been changed, then CF flag is cleared and AH = 00h;
if CF flag is set, and AH = 06h, then disk has been changed;
if CF flag is set, but AH has any other value, then this value should be interpreted as return code according to table A.06-1.

Note 1: before sending a request to INT 13\AH=16h function about an unknown disk drive, one has to investigate with INT 13\AH=15h function whether this disk drive supports media change line.
Note 2: media change line in most disk drive models is activated by opening (or closing) of disk slot lid, even if disk change event hasn't actually happen.

Note 3: each media change event is reported only once: after a call for INT 13\AH=16h function media change flag is cleared to its original state.

Note 4: extended media change function INT 13\AH=49h (with the same other specifications) may be applied to just any disk drive, including CD drives with physical number 80h and higher. BIOS support for these extended capabilities should be confirmed by INT 13\AH=41h function (8.01-55).

8.01-54 INT 13\AH=18h – set media type for formatting

Prepare:

AH = 18h
CH, CL, DH, DL – just the same as for INT 13\AH=02h (8.01-46)

On return:

AH – status code:
= 00h – requested parameters are supported;
= 01h – requested function isn't available;
= 0Ch – current disk type either isn't supported or is unknown;
= 80h – removable media isn't present in the drive.
ES:DI – pointer to floppy drive's parameters table A.08-2

Note 1: the INT 13\AH=18h function doesn't write the returned pointer to floppy drive's parameter table into memory address 0000:0078h, else known as INT 1E vector (A.12-1). INT 1E vector preparation is considered the caller's responsibility.

Note 2: being applied to a HDD, INT 13\AH=18h function returns CF flag set and status code AH = 01h. To obsolete 5.25" floppies and to 720 kb diskettes the INT 13\AH=17h function should be applied instead.

8.01-55 INT 13\AH=41h – INT 13 extensions check

Owing to INT 13 extensions many customary features of modern computers have been implemented since 1997, in particular, possibility to boot the PC from a CD-ROM and LBA addressing to large HDDs (note 4 to A.13-6).

Prepare:

AH = 41h
BX = 55AAh
DL – disk drive number (note 1 to 8.01-44)

On return:

On error CF flag is set, AH returns error code (A.06-1). Error code 01h means that requested function is not supported.
Clear state of CF flag signifies successful termination, and then:
AH – INT 13 extension version:
= 01h – version 1.x,

= 20h – version 2.0,
= 21h – version 2.1.
BX = AA55h signature confirms support for INT 13 extensions
CX – API subset word's bits signify support for functions:
bit 0 – INT 13\AH=42h-44h,47h,48h;
bit 1 – INT 13\AH=45h,46h,48h,49h, INT 15\AH=52h;
bit 2 – INT 13\AH=48h,4Eh;
bit 3 – extended disk address packet support (note 2)

Note 1: data in AL and DH registers may be lost on return.
Note 2: all versions of INT 13 extensions support 10-byte packet addressing to disks with capacity up to 127 Gb. Besides that, support for 20-byte extended disk address packets enables addressing beyond 127 Gb. Structures of both 10-byte and 20-byte address packets are shown in appendix A.13-4.

8.01-56 INT 13\AH=42h – extended read disk function

Prepare:
AH = 42h
DL – disk drive number (note 1 to 8.01-44)
DS:SI – pointer to disk address packet (A.13-4)

On return:
On error CF flag is set, AH returns error code (A.06-1).
Clear state of CF flag signifies successful termination.

Note 1: on return the number of successfully read data blocks is written into a word at offset 02h inside disk address packet.
Note 2: a pointer to buffer with read data is presented by a double word at offset 04h inside disk address packet.
Note 3: track seek function INT 13\AH=47h, being initiated beforehand with the same other specifications, enables CPU to perform a lot of job while disk drive is moving its head to the specified track. Pertinent usage of track seek function makes actual access to a particular track much faster.

8.01-57 INT 13\AH=43h – extended write to disk function

Prepare:
AH = 43h
AL – flags:
= 00h – skip verification procedure,
= 02h – verify written data.
DL – disk drive number (note 1 to 8.01-44)
DS:SI – pointer to disk address packet (A.13-4)

On return:

On error CF flag is set, AH returns error code (A.06-1).
Clear state of CF flag signifies successful termination.

Note 1: a buffer with the data to be written is pointed at by a dword at offset 04h inside disk address packet.

Note 2: on return the number of data blocks successfully written (or successfully verified) is saved in a word at offset 02h inside disk address packet.

Note 3: versions 1.x of INT 13 extensions used flag AL = 01h for verification of written data. If verification is requested, but is not supported, then INT 13\AH=43h function returns flag CF set and AH = 01h (invalid function).

Note 4: verification may be initiated separately by INT 13\AH=44h; other specifications are the same, except that value in AL is ignored.

8.01-58 INT 13\AH=45h – lock/unlock a drive

Several disk treatment procedures, being interrupted by alien access requests, may inflict severe data loss. Typical example of such procedure is defragmentation. While such procedure lasts, alien access attempts to the subjected disk must be blocked. Having been blocked, removable disk can't be ejected from its drive. Disk blocking in multitasking environment enables to avoid concurrent interventions. Up to 255 levels of nested procedures are allowed, requiring exclusive access to a disk. Having finished its job, each such procedure must release the subjected disk with unlock operation.

Prepare:

AX – subfunction:
= 4500h – lock media in drive: increase lock level by 1
= 4501h – unlock the media: decrease lock level by 1
= 4502h – report media lock level
DL – disk drive number (note 1 to 8.01-44)

On return:

On error CF flag is set, AH returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
AL – media lock level (= 00h if unlocked)

8.01-59 INT 13\AH=46h – eject removable media

Prepare:

AX = 4600h
DL – disk drive number (note 1 to 8.01-44)

On return:

On error CF flag is set, AH returns error code (A.06-1).
Clear state of CF flag signifies successful termination.

Note 1: before ejection the INT 13\AH=46h handler calls for INT 15\AH=52h function in order to be certain, that specified media at the current moment is not engaged in data transfer with cache buffer or with other programs in multi-tasking environment.

8.01-60 INT 13\AH=48h – request for drive's parameters

Prepare:

AH = 48h
DL – disk drive number (note 1 to 8.01-44)
DS:SI – pointer to buffer for data. The first word in buffer must declare its available length, not less than:
001Ah – for INT 13 extensions versions 1.x,
001Eh – for INT 13 extensions versions 2.x,
0049h – for INT 13 extensions versions 3.x.

On return:

On error CF flag is set, AH returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then:
DS:SI – pointer to buffer filled with data table A.13-2.

Note 1: if actually specified length corresponds to more obsolete version, then newer versions truncate the returned table according to format of specified length.

8.01-61 INT 13\AX=4A00h – drive emulation from CD

This function arranges a virtual logical disk copied from a disk image stored in an optical CD or DVD disc.

Prepare:

AX = 4A00h
DS:SI – pointer to boot specification packet A.15-1

On return:

On error CF flag is set, AH returns error code (A.06-1).
Clear state of CF flag signifies successful termination.

Note 1: the INT 13\AX=4C00h function accepts the same other specifications (except AX) and does the same, but then proceeds with booting the PC from the arranged virtual disk.

8.01-62 INT 13\AH=4Bh – drive emulation subfunctions

Prepare:

AH = 4Bh
AL – subfunction:

= 00h – terminate disk emulation
= 01h – get emulation status
DL – emulated disk number (note 1 to 8.01-44)
DS:SI – pointer to 13h-byte buffer for boot data packet

On return:

On error CF flag is set, AH returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then:
DS:SI – pointer to buffer filled with boot data packet A.15-1.

Note 1: having been given DL = 7Fh, subfunction AL = 00h terminates all current emulations.
Note 2: if emulation hasn't been performed, then clear state of CF flag is returned.

8.01-63 INT 13\AH=4Dh – read sectors of optical disc

Prepare:

AX = 4D00h
DS:SI – pointer to command packet (A.15-2)

On return:

On error CF flag is set, AH returns error code (A.06-1).
Clear state of CF flag signifies successful termination.

Note 1: returned data are written into a prepared buffer. A pointer to this buffer must be specified by a dword at offset 02h inside command packet (A.15-2).
Note 2: most often the INT 13\AH=4Dh function is used to read boot catalog of optical disc. Boot catalog structure is shown in table A.15-3.

8.01-64 INT 13\AH=4Eh – control over drive's hardware

Prepare:

AH = 4Eh
AL – subfunction:
= 00h – enable prefetch (reading into drive's buffer)
= 01h – disable prefetch
= 02h – set maximum PIO data transfer mode
= 03h – set PIO data transfer mode 0
= 04h – set default PIO data transfer mode
= 05h – enable INT 13 DMA maximum mode
= 06h – disable INT 13 DMA data transfer
DL – disk drive number (note 1 to 8.01-44)

On return:

On error CF flag is set, AH returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then:

AL = 00h – command affected specified drive only
= 01h – other devices are affected too (note 2)

Note 1: DMA and PIO data transfer modes are mutually exclusive. Selecting DMA disables PIO, and selecting PIO disables DMA.
Note 2: change of data transfer mode may affect other devices, connected to the same controller.

8.01-65 INT 14\AH=00h – initialize serial port

Prepare:
AH = 00h
AL – data transfer parameters:
bits 7–5: values from 000b to 111b correspond to data rates 110, 150, 300, 600, 1200, 2400, 4800, 9600 bits per second
bits 4–3: 00b or 10b – no parity check, 01b – odd parity check, 11b – even parity check.
bit 2: if cleared – 1 stop bit, if set – 2 stop bits.
bits 1–0: values from 00b to 11b correspond to data word lengths 5, 6, 7, 8 bits.
DX – serial port number (0000h – 0003h)
On return:
AH – line status byte (A.14-2)
AL – status of modem, if it is connected to the requested port.

8.01-66 INT 14\AH=01h – send a character to serial port

Prepare:
AH = 01h
AL – character to be sent
DX – serial port number (0000h – 0003h)
On return:
AH – line status byte (A.14-2)

Note 1: transmission error is marked by set state of bit 7 in returned status byte: it means that waiting time for response exceeded a preset time limit (timeout).

8.01-67 INT 14\AH=02h – read a character from serial port

Prepare:
AX = 0200h
DX – serial port number (0000h – 0003h)
On return:
AH – line status byte (A.14-2)

AL – received character, if bit 7 in returned line status byte is clear.

8.01-68 INT 14\AH=03h – get status of serial port

Prepare:

AX = 0300h
DX – serial port number (0000h – 0003h)

On return:

AH – line status byte (A.14-2)
AL – status of modem, if it is connected to the requested port.

8.01-69 INT 15\AH=52h – query whether a drive is busy

The INT 15\AH=52h function reports whether a drive is busy with data traffic at this moment. INT 15\AH=52h function is called for, in particular, by INT 13\AH=46h handler in order to prevent removable media ejection while data transfer procedure isn't finished yet.

Prepare:

AH = 52h
DL – disk drive number (note 1 to 8.01-44)

On return:

if flag CF is set, then drive is busy, and AH returns error code (A.06-1).
Clear state of CF flag signifies that requested drive is not busy.

Note 1: before applying INT 15\AH=52h function, BIOS support for this function should be checked with a call for INT 13\AH=41h.

Note 2: by default BIOS installs a dummy handler for INT 15\AH=52h, which always returns CF flag cleared. Responses will reflect real status of drive's traffic, when calls for INT 15\AH=52h function are intercepted yet by another handler, installed by disk caching driver.

8.01-70 INT 15\AX=5301h – activate APM real mode interface

When computer is switched on, its power management system (APM) stays inactive. While CPU is in real mode, APM interface activation can be initiated by a call for INT 15\AX=5301 function.

Prepare:

AX = 5301h
BX = 0000h (identifier of BIOS's APM extensions)

On return:

On error CF flag is set, AH returns error code (A.06-1).
Clear state of CF flag signifies successful termination.

Note 1: the INT 15\AX=5301h function forces APM system to emulate specifications of APM's version 1.0. In order to perform operations, which are not defined for APM's version 1.0, emulation of newer APM versions should be forced by INT 15\AX=530Eh function (8.01-72).

8.01-71 INT 15\AX=5307h – switching of power supply modes

Computers of ATX form factor can be switched off by a machine command, turning their power supply block into standby mode. Then power is supplied to those blocks only, which enable an opportunity to switch the PC on. Naturally, standby mode of power supply block must be hardware supported by power supply block itself and by PC's motherboard.

Prepare:

AX = 5307h
BX = 0001h (identifier of all APM-controlled devices)
CX = 0003h (code of switch off request)

On return:

On error CF flag is set, AH returns error code (A.06-1).
On success all the rest doesn't matter.

Note 1: switch off operation is defined by specifications of APM version 1.2. If APM system emulates version 1.0 (note 1 to 8.01-70), then an opportunity to switch the PC off should be unblocked by INT 15\AX=530Eh function (8.01-72).

Note 2: in obsolete computers of AT form factor a call for INT 15\AX=5307h function is allowed, but is ignored.

8.01-72 INT 15\AX=530Eh – APM version emulation request

In order to preserve compatibility with operating systems, specifications of newer APM versions stipulate an opportunity to emulate previous APM versions. That program or that operating system, which is to control power management, may request emulation of the desired APM version. In response APM BIOS emulates the requested or the nearest feasible APM version and returns number of that version. Further power management must be performed according to the actually emulated APM version.

Prepare:

AX = 530Eh
BX = 0000h (identifier of BIOS's APM extensions)
CX – requested APM version emulation (note 1)

On return:

On error CF flag is set, AH returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
AX – actually emulated APM version (note 1).

Note 1: integer and fractional parts of APM version number must be specified in separate bytes of CX register. In particular, for version 1.2 request the CX = 0102h value should be specified. Returned in AX version number fits the same format.
Note 2: version emulation request is defined by specifications of APM version 1.1. If version emulation request returns error code 80h or 86h, whereas function INT 15\AX=5301h (8.01-70) has terminated successfully, hence this computer implements APM version 1.0 only.

8.01-73 INT 15\AH=83h,86h – BIOS timer control

Prepare:

AX – subfunction:
= 8300h – initiate delay count and let the caller process to go on
= 8301h – halt count session (CX, DX, ES:BX are ignored)
= 8600h – initiate delay count and suspend the caller process until delay count expires (AL and ES:BX are ignored)
CX – most significant 16 bits of 32-bit delay (in microseconds)
DX – least significant 16 bits of 32-bit delay (in microseconds)
ES:BX – pointer to marker byte: its most significant 7-th bit is set when delay count expires (note 3).

On return:

Clear state of CF flag signifies successful termination.
CF flag is set on error or else when count session is initiated yet, and then AH register returns error code (A.06-1).

Note 1: most probable resolution of time period is 977 microseconds.
Note 2: some obsolete BIOS versions don't support subfunction 8301h.
Note 3: default address of marker byte is 0040:00A0h (A.12-1). When delay count expires, some BIOS versions assign 80h value to marker byte.
Note 4: BIOS timer is inaccessible from "DOS box" under Windows OS.

8.01-74 INT 15\AH=84h – read joystick state

Prepare:

AH = 84h
DX – subfunction:
= 0000h – read states of switches
= 0001h – read signals of position sensors

On return:

On error CF flag is set, AH returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then:

– DX=0000h subfunction returns states of switches in bits 4 – 7 of AL register;
– DX=0001h subfunction returns the following data:
AX – x-coordinate value of the 1-st manipulator
BX – y-coordinate value of the 1-st manipulator
CX – x-coordinate value of the 2-nd manipulator
DX – y-coordinate value of the 2-nd manipulator

Note 1: typical resistance of position sensors is 250 kOhm, and typical limits for coordinate values are 0000h – 01A0h.
Note 2: it is implied, that joystick is connected via a game port. If game port isn't present in a particular PC, then subfunction DX=0001h returns zero coordinate values, and subfunction DX=0000h returns AL=00h, equivalent to disconnected states of switches.

8.01-75 INT 15\AH=85h – PrintScreen key activity hook

The INT 15\AH=85h function is called for by INT09 handler in response to user's PrtScr keystroke. It is implied, that a handler for INT 15\AH=85h function should be installed by videocard driver in order to implement an enhanced Print Screen procedure, based on specific resources of a particular videocard. But until this specific handler is not installed, BIOS's dummy handler just returns control to the caller as if its request were satisfied.

Prepare:
AH = 85h
AL = 00h – procedure initiation when PrintScreen key is pressed
= 01h – procedure initiation when PrintScreen key is released
CF flag must be cleared beforehand

On return:
On error CF flag is set, AH returns error code (A.06-1).
Clear state of CF flag and AH = 00h signify successful termination.

8.01-76 INT 15\AH=87h – copying with access to extended memory

The INT 15\AH=87h function copies a data block within 16 Mb address space. Maximum size of data block is 64 kb. Copying is performed in protected mode while external interrupts are disabled, so that there is no need to prepare interrupt table for protected mode. But GDT table is needed, and it should be prepared according to the following template:

Reserved descriptor:	00 00 00 00 00 00 00 00
Reserved descriptor:	00 00 00 00 00 00 00 00
Source segment descriptor:	ss ss aa aa aa 93 00 00
Destination segment descriptor:	ss ss dd dd dd 93 00 00
Reserved descriptor:	00 00 00 00 00 00 00 00
Reserved descriptor:	00 00 00 00 00 00 00 00

In the shown template letter "a" denotes linear address field for source segment, letter "d" denotes linear address field for destination segment, letter "s" denotes size field for both source and destination segments. As far as size of data block is specified in CX register by number of double-byte words to be copied, both segment sizes (in bytes) must be not less than (2*CX) –1. In size fields and in address fields the first are the least significant bytes; most significant bytes are specified the last. Attribute byte in both source segment descriptor and destination segment descriptor must be 93h. More detailed explanation of descriptor's structure is given in appendix A.12-2.

Reserved descriptors, originally filled with zeros, will be filled with data by INT 15\AH=87h handler and will be used in protected mode. Having finished copying, the INT 15\AH=87h handler switches CPU back into real mode and restores original state for continuation of caller program's execution.

Prepare:

AH = 87h
CX – number of words to be copied, not more than 7FFFh
ES:SI – pointer to the first byte of prepared GDT table

On return:

On error CF flag is set, AH returns error code (A.06-1).
Clear state of CF flag signifies successful termination.

Note 1: as far as external interrupts during copying are disabled, some external interrupt calls may be missed. This wouldn't happen, if copying beyond conventional memory is arranged by HIMEM.SYS driver (A.12-4).

Note 2: HIMEM.SYS driver intercepts INT 15\AH=87h calls and gives no direct access to original BIOS's handler. However, programs may be allowed to access a limited region of extended memory via INT 15\AH=87h calls, if this region is reserved by /INT15 parameter (5.04-01).

8.01-77 INT 15\AH=88h – extended memory size up to 16 Mb

Prepare:

AH = 88h

On return:

On error CF flag is set, AH returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
AX – size in kilobytes of extended memory beyond 1024 kb.

Note 1: if valid address space is intermittent, size of its continuous part from 100000h (1024 kb) up to the first interstice will be reported.
Note 2: INT 15\AH=88h handler reads extended memory size from cells 30h and 31h of BIOS's CMOS data base.
Note 3: default BIOS handler for INT 15\AH=88h function is intercepted by HIMEM.SYS and EMM386.EXE drivers.
Note 4: information about extended memory beyond 16 Mb can be reported by INT 15\AX=E801h\E881h (8.01-79) and by INT 15\AX=E820h (8.01-80).

8.01-78 INT 15\AH=89h – switching CPU to protected mode

The INT 15\AH=89h handler switches CPU into protected mode and performs the most urgent preparations for functioning in protected mode, which include filling of segment registers with segment selectors and reprogramming interrupt controller according to protected mode specifications.

In order to ensure continuation of the caller program after switching CPU into protected mode, interrupt table (IDT) for protected mode and global descriptor table (GDT) should be prepared beforehand. IDT is filled with 8-byte descriptors ("gates"), specifying addresses and call conditions for protected mode handlers. Size and linear address of IDT table itself must be written into IDT segment descriptor inside GDT table.

The GDT table, prepared for INT 15\AH=89h handler, includes 8 descriptors, each 8 bytes long. Arrangement of descriptors in GDT table must correspond to the following template:

Reserved descriptor:	`00 00 00 00 00 00 00 00`
GDT segment descriptor:	`3F 00 aa aa aa 00 00 00`
IDT segment descriptor:	`FF 03 aa aa aa F2 00 00`
DS segment descriptor:	`ss ss aa aa aa 92 0s aa`
ES segment descriptor:	`ss ss aa aa aa 92 0s aa`
SS: segment descriptor	`ss ss aa aa aa 92 0s aa`
CS: segment descriptor	`ss ss aa aa aa 9A 0s 00`
Reserved descriptor:	`00 00 00 00 00 00 00 00`

In the shown template letter "a" denotes linear address fields for each segment, and letter "s" denotes segment size fields. Reserved descriptors must be filled with zeros. As examples the template shows particular values of attribute bytes, and also particular sizes for GDT and IDT segments. Detailed explanation of descriptor's structure is given in appendix A.12-2.

Prepare:

AH = 89h
BL – interrupt number for IRQ 0 (note 2)
BH – interrupt number for IRQ 8 (note 2)
ES:SI – pointer to the first byte of GDT table

On return:

On error CF flag is set, AH returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
AH = 00h, segment registers are filled with segment selectors, interrupt controller is reprogrammed. Former value in BP register may be altered.

Note 1: CS segment descriptor in GDT table must specify just that segment, which was defined by CS segment address in real mode. This condition ensures continuation of caller program's execution in protected mode from the command following the call for INT 15\AH=89h handler.

Note 2: assigned numbers of interrupts in BL and BH registers must be multiples of 8 (three least significant bits must be cleared). Numbers of interrupts, following the value in BL register, will be assigned to request lines IRQ 1 – IRQ 7. Similarly, numbers of interrupts, following the value in BH register, will be assigned to request lines IRQ 9 – IRQ F. An important factor in choice of reprogrammed interrupt numbers is that interrupts 00h – 1Fh may be invoked by CPU's exceptions.

Note 3: direct calls for real mode interrupt handlers become forbidden after switching to protected mode because of changed addressing format and reprogrammed interrupt controller.

8.01-79 INT 15\AX=E801h,E881h – extended memory size

Prepare:

AX = E801h or else = E881h

On return:

On error CF flag is set, AH returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
AX – size (in kilobytes) of extended memory in address space region between 1 Mb and 16 Mb.
BX – size (in 64-kb blocks) of extended memory in address space region beyond 16 Mb.
CX – size (in kilobytes) of configurated memory in address space region between 1 Mb and 16 Mb.
DX – size (in 64-kb blocks) of configurated memory in address space region beyond 16 Mb.

Note 1: if after successful termination AX = BX = 0000h, then size of extended memory should be read from registers CX and DX.
Note 2: unlike INT 15\AX=E801h, the INT 15\AX=E881h handler is able to return values exceeding 4 Gb. Most significant digits are returned in bits 31 – 16 of 32-bit registers EBX and EDX. Ways of access to bits 31 – 16, feasible for real mode programs, are described in article 7.02-06.
Note 3: the INT 15\AX=E801h\E881h functions are not supported by obsolete BIOS versions, developed before 1995.

8.01-80 INT 15\AX=E820h – memory dedication map

Memory dedication map is represented by a succession of 20-byte descriptors, each corresponding to a separate memory region, dedicated for a certain purpose. Each descriptor starts with 8-byte address of the region's first byte, then 8-byte length of that region follows, and the last 4 bytes are that region's dedication code (note 1). One call for INT 15\AX=E820h function returns one descriptor; hence this function has to be called for several times. For the first call a zero value (00000000h) in EBX register should be prepared. After the first call a non-zero EBX value is returned, defining target descriptor for the next call. Termination of calls cycle is marked by return of either a zero value in EBX register or a set state of CF flag.

Prepare:

AX = E820h
EBX – pointer to target descriptor in memory dedication map
ECX – size of ES:DI buffer, not less than 20 (=14h) bytes
EDX = 534D4150h – the "SMAP" signature
ES:DI – pointer to a buffer prepared for the descriptor

On return:

On error CF flag is set, EAX value is not equal to 534D4150h, AH register returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
EAX = 534D4150h "SMAP" – the "SMAP" signature
EBX – pointer to next descriptor in memory dedication map
ECX – actual length of the returned descriptor in buffer
ES:DI – pointer to buffer with the returned descriptor

Note 1: dedication codes should be interpreted as follows:
01h – memory, allocated for operating system;
02h – memory (system ROM), reserved by BIOS;
03h – ACPI tables area (may be free after being read);
04h – nonvolatile memory for system purposes.

Memory areas with other dedication codes should be considered reserved by BIOS system. Intervals of address space, excluded from memory dedication map, are not supported by hardware.

Note 2: some BIOS versions don't ignore contents of bits 31 – 16 in EAX register, so that calls for INT 15\AX=E820h function need these bits to be cleared (i.e. EAX = 0000E820h). Ways of access to bits 31 – 16, feasible for real mode programs, are described in article 7.02-06.

Note 3: some BIOS versions ignore ECX register and return 20 bytes of data after each call for INT 15\AX=E820h function.

Note 4: if a call for INT 15\AX=E820h function is not supported by BIOS in a particular computer, then an attempt should be undertaken to call for INT 15\AX=E801h (8.01-79). If this attempt fails too, then INT 15\AH=88h function (8.01-77) should be called for.

8.01-81 INT 16\AH=03h – keyboard's rate and delay

Prepare:

AX – subfunction:
= 0300h – set default repeat rate and delay values
= 0305h – set rate given in BL, set delay given in BH
= 0306h – get current values for repeat rate and delay
BL – (for AX=0305h only): repeat rate code (note 1)
BH – (for AX=0305h only): delay code (note 2)

On return:

BL – code of current repeat rate (note 1)
BH – code of current delay (note 2)
AH contents may be altered.

Note 1: allowed repeat rate codes from 00h to 1Fh correspond to repeat rates from 30 times per second to 2 times per second.

Note 2: allowed delay codes from 00h to 03h correspond to delays from 0.25 to 1 second.

8.01-82 INT 16\AH=05h – insert key code into keyboard buffer

Prepare:

AH = 05h
CH – scan code of keystroke (table A.02-1)
CL – ASCII code of corresponding character (table A.02-1)

On return:

AL – error code (A.06-1)
AH contents may be altered.

Note 1: obsolete BIOS versions don't support this function.

Note 2: codes of several keys (for example, of the ENTER key), being inserted into keyboard buffer, induce execution of corresponding functions.
Note 3: the INT 16\AH=05h function can't be used to insert codes of "functional" keys (SHIFT, CTRL, ALT).

8.01-83 INT 16\AH=10h – get key code out of keyboard buffer

The INT 16\AH=10h function is adapted for the most widespread 101\108-key "enhanced" keyboards, but also responds to keystrokes on compatible keys of other keyboards. The INT 16\AH=10h handler withdraws the topmost key code out of keyboard buffer and presents this code in AX register. If keyboard buffer is empty at that moment, then INT 16\AH=10h handler starts waiting for user's keystroke.

Prepare:
AH = 10h
On return:
AH – scan code of keystroke (table A.02-1)
AL – ASCII code of corresponding character (table A.02-1)

Note 1: the INT16\AH=20h does the same, but is able to respond to some keys, specific for 122-key keyboards.
Note 2: in obsolete computers similar mission is performed by INT16\AH=00h function. In modern computers this function is still active, but it ignores keys, which were not present in obsolete 84-key keyboards.

8.01-84 INT 16\AH=11h – copy key code from keyboard buffer

The INT 16\AH=11h function is adapted for the most widespread 101\108-key "enhanced" keyboards, but also responds to keystrokes on compatible keys of other keyboards. The topmost key code isn't withdrawn out of keyboard buffer, the INT 16\AH=11h handler just copies this code into AX register and doesn't wait for user's keystroke, if keyboard buffer at that moment is empty.

Prepare:
AH = 11h
On return:
If ZF flag is set to ZR state, then keyboard's buffer is empty.
If ZF flag is cleared to NZ state, then:
AH – scan code of keystroke (table A.02-1)
AL – ASCII code of corresponding character (table A.02-1)

Note 1: the INT16\AH=21h does the same, but is able to respond to some keys, specific for 122-key keyboards.

Note 2: in obsolete computers similar mission is performed by INT16\AH=01h function. In modern computers this function is still active, but it ignores keys, which were not present in obsolete 84-key keyboards.

8.01-85 INT 16\AH=12h – get keyboard status flags

The INT 16\AH=12h function copies into AX register the keyboard status word, stored most probably at address 0040:0017h in BIOS data area (A.02-3).

Prepare:

AH = 12h

On return:

AX – flags:

- bit 0 set: Right Shift key is kept pressed
- bit 1 set: Left Shift key is kept pressed
- bit 2 set: either of CTRL keys (right or left) is kept pressed
- bit 3 set: either of ALT keys (right or left) is kept pressed
- bit 4 set: the Scroll Lock switch is turned on
- bit 5 set: the Num Lock switch is turned on
- bit 6 set: the Caps Lock switch is turned on
- bit 7 set: the Insert switch is turned on
- bit 8 set: Left CTRL key is kept pressed
- bit 9 set: Left ALT key is kept pressed
- bit 10 set: Right CTRL key is kept pressed
- bit 11 set: Right ALT key is kept pressed
- bit 12 set: the Scroll Lock key is kept pressed
- bit 13 set: the Num Lock key is kept pressed
- bit 14 set: the Caps Lock key is kept pressed
- bit 15 set: the SysRq (PrtScr) key is kept pressed (note 2)

Note 1: in obsolete computers with 84\86-key keyboards a similar mission is performed by INT 16\AH=02h function. It returns flag's states in bits 7 – 0 of AL register only. In modern computers this function is still active, but it doesn't return those flag's states, which were not registered in obsolete computers.

Note 2: the pressed state of SysRq (PrtScr) key is not registered, unless either of ALT keys (left or right) is kept pressed too.

8.01-86 INT 17\AH=00h – send a character to LPT port

Prepare:

AH = 00h
AL – code of the character to be sent
DX – LPT port number: 0000h – 0002h correspond to LPT1 – LPT3

On return:

AH – LPT port's and printer's status (A.14-3).

8.01-87 INT 17\AH=01h – initialize printer's port

Prepare:

AH = 01h
DX – LPT port number: 0000h – 0002h correspond to LPT1 – LPT3

On return:

AH – LPT port's and printer's status (A.14-3).

Note 1: reported status in AH may be incorrect. More reliable data are reported after some delay by INT 17\AH=0200h function.

8.01-88 INT 17\AX=0200h – get status of LPT port

The INT 17\AX=0200h function is executed by both conventional LPT BIOS and EPP BIOS (Enhanced Parallel Port BIOS), but in different ways. EPP BIOS is not present in obsolete PCs, and then conventional LPT BIOS reports nothing more than port's status in AH (A.14-3). In modern PCs INT 17\AX=0200h calls can be directed to conventional LPT BIOS and give the same effect, if prepared value in BX register is not 5050h. But calls for INT 17\AX=0200h with BX=5050h and CH=45h are intercepted by EPP BIOS, which returns address of its entry point. A CALL FAR command (7.03-08) to this entry point invokes enhanced I/O functions, stipulated by IEEE 1284 specification (A.14-4).

Prepare:

AX = 0200h
BX = 5050h, if the call is addressed to EPP BIOS
CH = 45h, if the call is addressed to EPP BIOS
DX – LPT port number: 0000h – 0002h correspond to LPT1 – LPT3

On return:

AH – LPT port's and printer's status (A.14-3).
Status AH = 03h and flag CF set to CY state are returned by EPP BIOS, if it doesn't support the requested LPT port.
Status AH = 00h and flag CF cleared to NC state are returned by EPP BIOS, if it supports the requested LPT port, and then:
CX:AL = 4550:50h – signature of EPP BIOS versions 1.x
CX:AL = 5050:45h – signature of EPP BIOS versions 3.x
EPP BIOS versions 1.x and 3x return ES register contents intact and
DX:BX – segment: offset of EPP BIOS far entry point
EPP BIOS revision 7 returns ES contents altered and
DX – base address for EPP BIOS I/O operations
ES:BX – segment: offset of EPP BIOS far entry point.

Note 1: if a call for INT 17\AX=0200h is not addressed to EPP BIOS, then BX and CH registers may contain any values, except 5050h and 45h.
Note 2: a change of segment address in ES register is a specific symptom of the latest 7-th revision of EPP BIOS, supplemented with control functions for LPT multiplexer.

8.01-89 INT 18 – diskless boot hook

Prepare: nothing

When bootable disk can't be found, then PC's BIOS system calls for INT 18 handler. In obsolete PCs INT 18 handler launched BASIC language interpreter, stored in PC's read-only memory (ROM). Default INT 18 handler in modern computers displays a message about absence of BASIC language interpreter and then halts CPU.

However, a call for INT 18 can be intercepted by other handler, supplied in ROM memory of an extension card. In particular, interception may be performed by handlers, supplied with network cards, in order to implement diskless boot via local network.

8.01-90 INT 19 – bootstrap loader

The INT 19 handler performs an important part of PC's booting procedure: it copies boot sector from disk's boot partition into PC's memory at address 0000:7C00h and then transfers control to the copied boot sector's code. That disk is addressed first, which is specified the first in a sequence of boot alternatives, prepared by BIOS Setup program. From floppies their boot sector is copied immediately, HDD's boot partition is determined from partition table in MBR sector (A.13-5). If addressed disk is inaccessible, then an attempt is undertaken to address the next disk in prepared sequence of boot alternatives. If all attempts fail, the last default resort is a call for INT 18 handler (8.01-89).

However, the INT 19 handler doesn't clear memory and doesn't restore interrupt table. Therefore computer most probably will get hanged after a call for INT 19 handler, if appropriate preparations for this call have not been made. When necessary, reboot should be initiated not by a call for INT 19 handler, but otherwise: either by keyboard controller's FEh command (note 3 to A.11-3) or by a CALL FAR command (7.03-08) to boot program's enter point F000:FFF0h (note 4 to A.12-1), which is the same in all AT-compatible computers.

8.01-91 INT 1A\AH=00h – get system ticks count

Prepare:

AH = 00h

On return:

AL – nonzero if midnight passed since last reading of ticks count.
CX – most significant 2 bytes of ticks count, 1800B0h per 24 hours

DX – least significant 2 bytes of ticks count, 18.2 ticks per second

Note 1: tick count is reset at midnight.
Note 2: after midnight DOS must be the first to request system ticks count, otherwise it misses midnight flag and fails to advance the date.
Note 3: system time (in ticks) may be set by INT 1A\AH=01h. Number of ticks to be set must be prepared in CX and DX registers in the same way.

8.01-92 INT 1A\AH=02h – read real-time clock

Prepare:

AH = 02h

On return:

On error CF flag is set, returned data are invalid.
Clear state of CF flag signifies successful termination, and then
CH – hours
CL – minutes
DH – seconds
DL = 00h standard time ("winter time" for northern hemisphere)
= 01h daylight time ("summer time" for northern hemisphere)

Note 1: hours, minutes and seconds are returned in packed decimal format, i.e. two decimal digits per byte.
Note 2: real time may be set with INT 1A\AH=03h. Time data must be prepared in CH, CL, DH and DL registers in the same form.

8.01-93 INT 1A\AH=04h – read real-time date

Prepare:

AH = 04h

On return:

On error CF flag is set, returned data are invalid.
Clear state of CF flag signifies successful termination, and then
CH – century
CL – year
DH – month
DL – day

Note 1: day, month, year and century are returned in packed decimal format, i.e. two decimal digits per byte.
Note 2: new date may be set with INT 1A\AH=05h. The data to be set must be prepared in CH, CL, DH and DL registers in the same form.

8.01-94 INT 1A\AH=06h – preset time for daily event invocation

The INT 1A\AH=06h function specifies moments of regular calls for INT 4A handler by PC's BIOS system. The desired action (clock alarm, for example) is implied to be done by INT 4A handler, which has to be written by the user and has to be loaded yet.

Prepare:

AH – 06h
CH – hour
CL – minutes
DH – seconds

On return:

On error CF flag is set: most probably time preset is active already.
Clear state of CF flag signifies successful termination.

Note 1: hour, minutes and seconds must be prepared in packed decimal format, i.e. two decimal digits per byte.

Note 2: preset FFh discards partial count. For example, in case of CH = FFh the INT 4A handler is invoked once every hour, in case of CH = CL = FFh the INT 4A handler is invoked every minute.

Note 3: preset time remains active until it is disabled by a call for INT 1A\AH=07h function, which needs AH = 07h only to be prepared.

8.01-95 INT 1B – keyboard's "CTRL-Break" hook

Prepare: nothing

The INT 09 handler calls for INT 1B each time when keyboard controller reports about CTRL-Break keystroke. Default INT 1B handler sets TRUE state to a flag in BIOS data area (bit 7 in byte at offset 71h in table A.02-3), and then control is returned to interrupted program. The state of this flag is checked by some MS-DOS handlers, called by the current program. If flag is found set to TRUE state, then INT 23 handler (8.02-83) is called for. The latter suspends execution of current program and is responsible for all consequent events (1.03).

Replacement of INT 1B handler's address in interrupt table by a pointer to IRET command (7.03-30) is a common trick, used in many programs in order to prevent their abortion, which otherwise may be caused by a CTRL-Break keystroke.

8.01-96 INT 1C – system timer tick hook

Prepare: nothing

INT 1C is called by INT 08 handler (8.01-09) at each system timer's tick, i.e. 18.2 times per second. The default INT 1C handler just returns control to the caller. But every

resident program is allowed to replace the default handler with its own INT 1C handler, which will transfer control to program's resident module at each timer's tick. Thus computer systems are enabled to implement monitoring and control over developing processes in real time.

8.02 Interrupt handlers, loaded by MS-DOS7 (INT 20 – INT 2E)

8.02-01 INT 20 – termination of program's execution

The INT 20 handler restores former interrupt table values from program's PSP (A.07-1), releases memory, occupied by the program, restores states of segment registers and stack, and then transfers control to caller's code pointed at by INT 22 (8.02-82). However, INT 20 requires PSP segment address to be present in CS: register, and gives no opportunity to leave errorlevel code (3.15-03), characterizing circumstances of program's termination. Therefore termination of program's execution with a call for INT 21\AH=4Ch function (8.02-55) should be preferred.

Prepare:

segment address in CS: register must point at PSP segment.

On return:

return wouldn't happen.

Note 1: in operating environment of Debug.exe the INT 21\AH=4Ch and INT 20 handlers produce different effects: INT 20 transfers control to Debug.exe, whereas INT 21\AH=4Ch terminates debugger's session and returns control to DOS.

Note 2: INT 21\AH=00h also terminates the caller program and always leaves errorlevel code 00h, just as INT 20 does. MS-DOS7 supports INT 21\AH=00h for preserving compatibility with obsolete software.

8.02-02 INT 21\AH=01h – get a character via STDIN channel

The INT 21\AH=01h function reads character code from STDIN (standard input) channel, writes this code into AL register and sends a copy of this code into STDOUT (standard output) channel. Both STDIN and STDOUT channels may be redirected, but until this is not done, default STDIN source is keyboard, and default STDOUT destination is the screen. Therefore a call for INT 21\AH=01h function induces PC to start waiting for a keystroke, and after keystroke the corresponding character appears on the screen.

Prepare:

AH = 01h

On return:

AL – received character's ASCII code

Note 1: returned ASCII code is represented by two rightmost digits of hexadecimal numbers in table A.02-1.

Note 2: INT 21\AH=01h function shouldn't be called from command files, which are sent to command interpreter via redirection, because in this case INT 21\AH=01h function intercepts STDIN traffic, including those commands, which are to be received by command interpreter.

Note 3: INT 21\AH=01h function checks the state of that flag, which is set after CTRL-C and CTRL-Break keystrokes (8.01-95). If this flag is found in TRUE state, then INT 23 handler is called for.

Note 4: in the same way the INT 21\AH=03h function reads character's code from STDAUX channel, which has the COM1 port as its default source.

8.02-03 INT 21\AH=02h – send a character via STDOUT channel

The INT 21\AH=02h function sends specified ASCII code into STDOUT channel. If STDOUT channel is not redirected, its default destination is display, and therefore a character, corresponding to the sent ASCII code, will appear on the screen.

Prepare:

AH = 02h
DL – ASCII code of the character to be sent into STDOUT

On return:

AH – the sent ASCII code, except TAB code (09h), which is expanded into spaces (20h).

Note 1: if STDOUT channel is redirected into a file, then sending into STDOUT channel doesn't imply checks of media presence in a drive, of whether this media is full, write-protected, etc. But when target drive is busy with previous operation, then INT 21\AH=02h handler will wait for termination of that previous operation.

Note 2: INT 21\AH=02h function checks the state of that flag, which is set after CTRL-C and CTRL-Break keystrokes (8.01-95). If this flag is found in TRUE state, then INT 23 handler is called for.

Note 3: the INT 21\AH=06h also sends ASCII code into STDOUT channel, but doesn't check CTRL-C\CTRL-Break flag. Besides that, INT 21\AH=06h function performs quite different mission, if DL=FFh (8.02-04).

Note 4: similarly to INT 21\AH=02h, the INT 21\AH=04h function sends ASCII code into STDAUX channel (default destination – COM1 port), and INT 21\AH=05h function sends ASCII code to STDPRN channel (default destination – LPT1 port).

8.02-04 INT 21\AH=06h – code copying from STDIN channel

Having been given the DL = FFh specification, INT 21\AH=06h function reads ASCII code from STDIN channel, almost as INT 21\AH=01h function does, but difference is that INT 21\AH=06h function

– doesn't check the CTRL-C\CTRL-BREAK flag,
– doesn't withdraw the copied code from its source,
– doesn't cause waiting, when the source is empty,
– doesn't send code's copy into STDOUT channel,
– is able to cope with extended key codes.

The term "extended" is applied to those key codes, which differ by their scan-code part (left two digits of numbers in table A.02-1), and have their ASCII part values either 00h or E0h. If BIOS reports either of these two ASCII values, then INT 21\AH=06h function clears ZF flag and returns AL = 00h, thus indicating a keystroke with extended key code. In this case INT 21\AH=06h function has to be called once more, and then in AL register it will return scan-code, which enables to discriminate keystrokes with "extended" key codes.

Other ASCII codes (except 00h and E0h) are returned in AL register at once, and then scan-code can't be returned. After that repeated call for INT 21\AH=06h function is allowed, but it will be executed as separate, independent from the previous call.

Prepare:

AH = 06h
DL = FFh (for other values – note 3 to 8.02-03)

On return:

ZR (set) state of ZF flag signifies that the source is empty
NZ (cleared) state of ZF flag signifies presence of data, and then
AL – ASCII code (or scan-code) copied via STDIN channel.

Note 1: INT 21\AH=07h function does almost the same, but withdraws the read code from STDIN source, ignores DL contents, doesn't alter the state of ZF flag, and may cause waiting for a keystroke, if STDIN source is keyboard buffer, and it is empty at that moment.

Note 2: INT 21\AH=08h function does the same as INT 21\AH=07h (see note 1 above), but besides this checks the state of that flag, which is set after CTRL-C and CTRL-Break keystrokes (8.01-95). If this flag is found in TRUE state, then INT 23 handler is called for.

Note 3: INT 21\AH=06h-08h functions shouldn't be called from command files, which are sent to command interpreter via redirection, because in this case INT 21\AH=06h-08h functions intercept STDIN traffic, including those commands, which are to be received by command interpreter.

Note 4: if flag of hieroglyphic languages support (byte at offset 3Ch in table A.07-1) is set to TRUE state, then INT 21\AH=06h-08h functions are able to return partially formed double-byte codes.

8.02-05 INT 21\AH=09h – send a string to STDOUT

Prepare:

AH = 09h
DS:DX – pointer to start byte of a '$'-terminated string

On return:

AL = 24h (code of the "$" character, the last one in the string)

Note 1: output continues until the first character "$" (24h) is encountered; the character "$" itself is not sent to STDOUT and must not be present inside the string.
Note 2: INT 21\AH=09h function checks the state of that flag, which is set after CTRL-C and CTRL-Break keystrokes (8.01-95). If this flag is found in TRUE state, then INT 23 handler is called for.
Note 3: one more function enabling to send a string is INT21\AH=40h (8.02-36).

8.02-06 INT 21\AH=0Ah – buffered input from STDIN channel

When STDIN channel isn't redirected, the INT 21\AH=0Ah function waits for keystroke(s) and after each keystroke writes its code into a prepared buffer. The buffer must not necessarily be empty: new data may be appended to previous buffer's contents. Copies of written codes are sent via STDOUT channel to display. Writing terminates after reception of the 0Dh code, produced by ENTER keystroke. When STDIN channel gets data from a file via redirection, then buffer filling goes non-stop and terminates after the first encountered 0Dh byte. In both cases writing terminates when prepared buffer becomes full.

Prepare:

AH = 0Ah
DS:DX – pointer to start of prepared buffer, where 2 bytes must be filled yet:
at offset 00h: maximum buffer's size in bytes;
at offset 01h: start offset for writing new data.

On return:

DS:DX – pointer to start of filled buffer, where 2 bytes denote:
at offset 00h: maximum buffer's size in bytes;
at offset 01h: number of bytes actually written yet.

Note 1: INT 21\AH=0Ah function doesn't wait for a keystroke and returns control back at once, if buffer size (pointed at by DS:DX) is 00h.

Note 2: count of bytes in prepared buffer starts from offset 02h, excluding byte 0Dh, which terminates writing session.

Note 3: INT 21\AH=0Ah function checks the state of that flag, which is set after CTRL-C and CTRL-Break keystrokes (8.01-95). If this flag is found in TRUE state, then INT 23 handler is called for.

Note 4: INT 21\AH=0Ah function shouldn't be called from command files, which are sent to command interpreter via redirection, because in this case INT 21\AH=0Ah function intercepts STDIN traffic, including those commands, which are to be received by command interpreter.

8.02-07 INT 21\AH=0Bh – get status of STDIN channel

The INT 21\AH=0Bh function is used to determine whether there are any data in STDIN channel source. When STDIN channel is not redirected, the INT 21\AH=0Bh function reports whether keyboard buffer is empty or not. If data are to be received via redirection, then INT 21\AH=0Bh function shows whether there is at least one byte pending in the source.

Prepare:

AH = 0Bh

On return:

AL = 00h, if STDIN channel source is empty, or else
= FFh, if there is at least one byte in STDIN channel source.

Note 1: INT 21\AH=0Bh function checks the state of that flag, which is set after CTRL-C and CTRL-Break keystrokes (8.01-95). If this flag is found in TRUE state, then INT 23 handler is called for.

8.02-08 INT 21\AH=0Ch – clear keyboard buffer and read STDIN

The INT 21\AH=0Ch handler clears keyboard buffer and then calls for a selected STDIN input function, specified by its code in AL register. Allowed codes 01h, 06h, 07h, 08h, 0Ah correspond to functions INT 21\AH=01h, INT 21\AH=06h, INT 21\AH=07h, INT 21\AH=08h, INT 21\AH=0Ah. Other relevant features are defined by the chosen input function.

Prepare:

AH = 0Ch
AL – code of input function: 01h or 06h or 07h or 08h or 0Ah
other registers – as required by the specified input function

On return:

just as the specified input function returns.

Note 1: INT 21\AH=0Ch handler with allowed input function's codes shouldn't be called from command files, sent to command interpreter via redirection, because in this case input function intercepts STDIN traffic, including those commands, which are to be received by command interpreter.

Note 2: if code in AL is not one of allowed codes (01h, 06h, 07h, 08h, 0Ah), then INT 21\AH=0Ch handler clears keyboard buffer, but no STDIN input is attempted.

8.02-09 INT 21\AH=0Dh – write buffer's data to disk

Contents of disk buffers should be written back to current disk each time when current program intends to address another disk. Besides that, a call for INT 21\AH=0Dh function restores default address of DTA area (note 6 to A.07-1).

Prepare:

AH = 0Dh

On return:

Clear state of CF flag signifies successful termination.

Note 1: INT 21\AH=0Dh function writes disk buffers to disk, but doesn't update altered contents of directories. Directories are updated, when one of file's handles is closed by INT 21\AH=3Eh function (8.02-34).

8.02-10 INT 21\AH=0Eh – appointment of current logical disk

The INT 21\AH=0Eh function assigns "current" status to that logical disk, which should be chosen by default in order to enable execution of disk access operations without explicit disk specification.

Prepare:

AH = 0Eh
DL – number of the selected logical disk (note 1)

On return:

AL – maximum number of letter-names, allowed by LASTDRIVE specification (4.17, 4.18)

Note 1: numeration of logical disks follows the order of their letter-names: 00h = A:, 01h = B:, 02h = C:, and so on.

Note 2: being called by drivers at their initiation, the INT 21\AH=0Eh function may return an invalid number in AL register, because it is read from list-of-lists (A.01-2) at offset 21h and may be not written there yet at the moment of initiation.

Note 3: current number of default logical disk is reported by INT 21\AH=19h function (8.02-15).

8.02-11 INT 21\AH=11h – find first matching file using FCB

The INT 21\AH=11h handler uses FCB (File Control Block) – an obsolete form of data specification, unsuitable for access to files on disks formatted with FAT-32 file system. However, FCBs are suitable for matching file search inside current directory, including directories on disks with FAT-32 file system. Allowable FCB structures are shown in table A.09-5. Filename specification in FCB may be a mask with "?" wildcards (2.01-03). FCBs can be conveniently arranged by INT 21\AH=29h function (8.02-19).

Some search data, returned by INT 21\AH=11h handler, supplement data in FCB, as it is shown in table A.09-5. But main part of search results is returned in DTA – Data Transfer Area 128 bytes long. Default DTA position is inside PSP (A.07-1), starting at offset 80h, but DTA may be shifted elsewhere by INT 21\AH=1Ah function (8.02-16). Structure of returned data in DTA depends on type of original FCB – whether it was normal or extended FCB. Both variants of returned data structure are shown in table A.09-1.

Prepare:

AH = 11h
DS:DX – pointer to unopened FCB, normal or extended (A.09-5)

On return:

AL – error code (A.06-1):
= FFh – no matching files found
= 00h – DTA region (A.09-1) is filled with data about first found file.

Note 1: INT 21\AH=11h function enables to obtain information about volume label. For this purpose an extended FCB with attribute byte 08h should be prepared, and current directory must be the disk's root directory.

Note 2: INT 21\AH=11h function enables to obtain information about a directory. For this purpose an extended FCB with attribute byte 10h should be prepared, and current directory must be the parent of requested directory.

Note 3: if search is to be continued with INT 21\AH=12h function (8.02-12), then all data in FCB and in DTA region, including the returned data, must be preserved unchanged.

Note 4: a search for a file without usage of FCB specification is performed by INT 21\AH=4Eh function.

8.02-12 INT 21\AH=12h – find next matching file using FCB

The INT 21\AH=12h function continues search for matching files after successful termination of preceding search iteration, performed either by INT 21\AH=11h (8.02-11) or by INT 21\AH=12h functions. Necessary condition of consistent continuation is

preserving intact those data, returned in FCB and in DTA after preceding search iteration. Search results are returned just as after a call for INT 21\AH=11h (8.02-11).

Prepare:

AH = 12h
DS:DX – pointer to unopened FCB, normal or extended (A.09-5)
DTA region (A.09-1) with data left after previous search iteration

On return:

AL – error code (A.06-1):
= FFh – no more matching files found
= 00h – DTA region (A.09-1) is filled with data about next found file.

Note 1: all notes to INT 21\AH=11h (article 8.02-11) are equally valid for INT 21\AH=12h function.

8.02-13 INT 21\AH=13h – delete matching file(s) using FCB

The INT 21\AH=13h handler uses obsolete FCB (File Control Block) form of specification, which is nevertheless suitable for deletion of file(s) in current directory, including directories on disks formatted with FAT-32 file system. Allowable FCB structures are shown in table A.09-5. For deletion of several files FCB may specify a mask with "?" wildcards (2.01-03). FCB can be conveniently arranged by INT 21\AH=29h function (8.02-19).

Files to be deleted must be closed beforehand (8.02-34), must be free from HSR attributes (A.09-2), and disk with these files must not be write-protected.

Prepare:

AH = 13h
DS:DX – pointer to unopened FCB, normal or extended (A.09-5)

On return:

AL – error code (A.06-1):
= FFh – no matching files found
= 00h – matching files have been deleted successfully.

Note 1: owing to attribute byte in extended FCB (A.09-5) the INT 21\AH=13h function is able to delete volume labels and files with R (read-only) attribute. Deletion of subdirectories is also possible, but then files in these subdirectories are turned into lost clusters.

Note 2: deleted file is not erased physically; rather its directory entry is made invalid: the first character of this entry becomes overwritten with invalidity mark – code E5h.

Note 3: deletion of any file with long filename by INT 21\AH=13h function doesn't affect associated directory entries beyond the main one, whereas these entries contain continuation of long filename.

Note 4: file(s) deletion without FCB specification usage is performed by INT 21\AH=41h function (8.02-37).

8.02-14 INT 21\AH=17h – rename matching file using FCB

Prepare:

AH = 17h

DS:DX – pointer to unopened FCB, normal or extended (A.09-5). Current name is written in FCB at its ordinary place, proposed new name must be written 10h bytes further, just under the current name in next dump line. Particular offset values are given in table A.09-5. Both names must be written in normalized form, which is provided, for example, by INT 21\AH=29h function (8.02-19). Required buffer's length is 28 bytes for normal FCB and 35 bytes for extended FCB.

On return:

AL – error code (A.06-1):
= 00h – specified file has been renamed successfully.
= FFh – failure: either matching file isn't found, or file is protected by HRS attributes, or a file with proposed new name exists yet.

Note 1: filemasks in FCB are allowed, but only with "?" wildcards (2.01-03) and with the same wildcard positions in both filemasks: in that replacing the current filename and in that replacing the proposed new filename. Characters, corresponding to wildcard positions, will be copied from current actual filenames and inserted in the same positions in new assigned filenames.

Note 2: as far as FCB doesn't specify paths, INT 21\AH=17h function can rename files inside current directory only. Renaming outside the current directory may be performed by INT 21\AH=56h function (8.02-62), which doesn't use FCB specifications.

Note 3: extended FCBs with attribute byte 10h enable to rename subdirectories from their parent directory, and extended FCBs with attribute byte 08h enable to rename volume label from root directory of the current disk.

8.02-15 INT 21\AH=19h – report "current" logical disk

The INT 21\AH=19h function reports which logical disk will be taken by default for disk access operations without explicit disk specifications.

Prepare:

AH = 19h

On return:

AL – number of "current" logical disk (note 1 to 8.02-10).

Note 1: appointment of "current" logical disk can be changed by INT 21\AH=0Eh function (8.02-10).

8.02-16 INT 21\AH=1Ah,2Fh – set/get DTA area address

Data Transfer Area (DTA) is a buffer 128 bytes long used by file search functions (8.02-11, 8.02-12, 8.02-57, 8.02-58). Default position of DTA area is inside current program's PSP at offset 0080h (note 6 to A.07-1).

Prepare:

AH – subfunction:
= 1Ah – specify new position for DTA
= 2Fh – report current DTA position
DS:DX – pointer to new DTA position (for AH = 1Ah subfunction only)

On return:

ES:BX – pointer to current DTA position (after AH = 2Fh subfunction only)

Note 1: examples of DTA data structures are shown in table A.09-1.
Note 2: DTA is returned to its default position (PSP:0080h) after each call for INT 21\AH=0Dh function (8.02-09).

8.02-17 INT 21\AH=1Ch – get information about a disk

Prepare:

AH = 1Ch
DL – logical disk number (note 1)

On return:

AH – media identifier (ID byte):
= F8h – fixed disk (HDD),
= F9h – 1.2 Mb or 720 kb floppy diskette,
= FAh – virtual RAM-disk,
= FDh – 360 kb floppy diskette,
= F0h – other media, including 1.44 Mb diskettes.
CX – bytes per sector;
DS:BX – pointer to the same media identifier byte in memory;
DX and AL contents are not preserved.

Note 1: this function uses "shifted" numeration of logical disks: number 00h is the default ("current") logical disk, then follow 01h = A:, 02h = B:, 03h = C:, and so on.
Note 2: being applied to an invalid or non-existing disk, INT 21\AH=1Ch doesn't indicate an error with CF flag, but rather returns values in AH, BX and DS registers unchanged.

Note 3: INT 21\AH=1Bh reports information in the same way about the default ("current") disk only, ignoring DL contents. Both INT 21\AH=1Ch and INT 21\AH=1Bh are obsolete functions, unable to identify most types of modern media.

8.02-18 INT 21\AH=25h – write a pointer into interrupt table

Prepare:

AH = 25h
AL – interrupt number, whose handler's address is to be written
DS:DX – pointer to new interrupt handler

Note 1: this function overwrites the former handler's address. If it shouldn't be lost, you must take care about saving it in advance.
Note 2: INT 21\AH=25h handler, supplied by MS-DOS7, may be used while CPU is in real or in V86 mode, but can't be used in protected mode.

8.02-19 INT 21\AX=2901h – parse a filename into FCB

The INT 21\AX=2901h function arranges an unopened FCB (File Control Block) of normal type (A.09-5), filled with normalized form of given filename. File's name and suffix are written separately in their FCB fields, letters are translated to upper case, asterisk wildcards (2.01-03) are expanded into appropriate number of "?" wildcards. If name is less than 8 bytes long, or suffix is less than 3 bytes long, then the rest free character cells are filled with spaces (20h). Parsing is stopped at the first encountered slash, or at a space, or at 0Dh terminator byte (the 00h terminator byte is not allowed, though).

Filename can't be preceded by a path, but may be preceded by disk's letter-name (for example, A:Config.sys). Disk's letter-name will be transformed into logical disk number, written into a cell in FCB just preceding the filename field. If the second character in presented line is not a colon, then disk's letter-name is considered missing, and disk number cell in FCB will be filled with 00h – number of the default ("current") disk.

Prepare:

AH = 2901h
DS:SI – pointer to filename string to be parsed (wildcards allowed)
ES:DI – pointer to a prepared buffer 21 bytes long for FCB

On return:

AL = 00h – successful parsing, no wildcards encountered
= 01h – successful parsing, wildcards are present
= FFh – parsing failed (invalid specification)
DS:SI – pointer to the character where parsing has stopped
ES:DI – pointer to a buffer filled with unopened FCB

Note 1: buffer must be 36 bytes long, if later FCB has to be transferred into its "opened" form in order to provide access to a file.
Note 2: parsing also can be performed by INT 21\AX=2903h function, which doesn't overwrite previous contents of disk number cell in FCB.

8.02-20 INT 21\AH=2Ah – get system date

Prepare:

AH = 2Ah

On return:

AL – day of the week (00h = Sunday)
CX – year (range 1980 – 2099)
DH – month
DL – day

Note 1: all values are returned in packed decimal format, i.e. two decimal digits per byte.
Note 2: system date can be changed by INT 21\AH=2Bh function. The required values should be prepared in AL, CX, DH, DL registers just as it is shown above. If INT 21\AH=2Bh function fails, it returns AL=FFh and leaves system date unchanged.

8.02-21 INT 21\AH=2Ch – get system time

Prepare:

AH = 2Ch

On return:

CH – hours
CL – minutes
DH – seconds
DL – 1/100 parts of a second

Note 1: all values are returned in packed decimal format, i.e. two decimal digits per byte.
Note 2: some computers count DL values in steps 0.05 s, some other computers always return DL = 00h.
Note 3: system time can be changed by INT 21\AH=2Dh function. The required values should be prepared in CH, CL, DH, DL registers just as it is shown above. If INT 21\AH=2Dh function fails, it returns AL=FFh and leaves system time unchanged.

8.02-22 INT 21\AH=30h – get DOS version

Prepare:

AX = 3000h – return manufacturer's (OEM) identifier in BH
= 3001h – return version flag byte in BH.

On return:

AL.AH – DOS version number
BH – OEM identifier or version flag byte.

Note 1: OEM identifier is 00h for IBM, 66h for PhysTechSoft, EEh for DR-DOS, EFh for Novell, FDh for FreeDOS, FFh for Microsoft.

Note 2: TRUE state of bit 3 in version flag byte marks special DOS versions, designed to be stored in ROM.

Note 3: returned version number is read from a word at offset 40h in PSP (A.07-1) of the caller program. If SETVER.EXE driver is installed, and if name of the caller program is written yet into SETVER's table, then a requested version will be substituted for true DOS version number at offset 40h in caller program's PSP.

Note 4: certainly true DOS version number is reported by INT 21\AX=3306h function (8.02-27).

8.02-23 INT 21\AH=31h – terminate execution, leaving resident module

Prepare:

AH = 31h
AL – hexadecimal errorlevel value (note 3 to 8.02-55)
DX – size of resident module in 16-byte paragraphs, not less then 6 paragraphs, counted from the start of PSP (A.07-1).

Note 1: INT 21\AH=31h function releases main program's memory (except its resident module), restores pointers in interrupt table, restores states of segment registers and stack, and then transfers control to caller's code pointed at by INT 22 (8.02-82). But INT 21\AH=31h function doesn't close opened files, doesn't release environment memory area and that memory, which has been allocated via INT 21\AH=48h (8.02-50). If necessary, these operations must be performed by the program itself before its termination.

Note 2: obsolete INT 27 handler (8.02-86) does the same, but limits resident module's size to 64 kb and doesn't leave errorlevel values.

8.02-24 INT 21\AH=32h – get disk's parameters block (DPB)

Prepare:

AH = 32h
DL – logical disk number (note 1 to 8.02-17)

On return:

AL = FFh, – requested disk is invalid or a network disk
= 00h, – request is performed successfully, and then
DS:BX – pointer to DPB block (A.03-1)

Note 1: a pointer in DS:BX to the default ("current") disk's DPB can also be obtained via INT 21\AH=1Fh; the latter ignores value in DL.
Note 2: both INT 21\AH=32h and INT 21\AH=1Fh try to update the DPB by reading the requested disk. If reading fails, INT 24 handler is called for with its "Abort, Retry, Fail?" question. If disk access attempt at that moment is not desirable, the pointer to any DPB may be found via DOS's list of lists (note 1 to A.01-2).
Note 3: both INT 21\AH=32h and INT 21\AH=1Fh can't be applied to disks formatted with FAT-32: then INT 21\AX=7302h function (8.02-79) should be used instead.

8.02-25 INT 21\AX=3300h – get BREAK flag's state

Prepare:
AX = 3300h
On return:
DL = 00h – BREAK flag is turned OFF
= 01h – BREAK flag is turned ON

Note 1: BREAK flag is in DOS's swappable area at offset 17h (A.01-3).
Note 2: state of BREAK flag can be changed by INT 21\AX=3301 function, it accepts the state to be set from DL register in the same form.

8.02-26 INT 21\AX=3305h – get boot drive

Prepare:
AX = 3305h
On return:
DL – boot drive number (note 1 to 8.02-17)

Note 1: INT 21\AX=3305h function reads number of the disk, used to boot the PC, from DOS's list-of-lists (A.01-2) at offset 43h.

8.02-27 INT 21\AX=3306h – get true version of DOS

Prepare:
AX = 3306h
On return:
AL = FFh, if true version of DOS is less than 5.00
When any other value is returned in AL register, then:
BL.BH – true version of DOS
DH – flags:
bit 3 – DOS is stored in ROM
bit 4 – DOS is loaded into HMA area
DL – DOS revision number.

Note 1: the data returned by INT 21\AX=3306h function can't be affected by SETVER.EXE driver (unlike those data returned by INT 21\AH=30h).

Note 2: INT 21\AX=3306h function is not supported by DOS versions less than 5.00. Besides AL = FFh, symptoms of obsolete DOS version may be values BL < 05h and BH > 64h.

8.02-28 INT 21\AH=34h – get address of InDOS flag

InDOS flag is a byte counter of active DOS functions nesting level: it is incremented whenever any INT 21 function is called for and is decremented when its execution terminates. States of InDOS flag and of critical error flag (note 1) should be checked before each call for a DOS function from any resident module: if either of these flags has a non-zero value, then such call is unsafe. Because of the same reason the INT 21\AH=34h function should be called once at initialization of TSR program, and the returned pointer should be saved. Later, when calls for DOS functions may be unsafe, system flag's states can be read immediately owing to the saved pointer.

Prepare:

AH = 34h

On return:

ES:BX – pointer to one-byte InDOS flag

Note 1: critical error flag is a byte just preceding the InDOS flag (A.01-3). A pointer to critical error flag can be obtained via INT 21\AX=5D06h function.

8.02-29 INT 21\AH=35h – get pointer to interrupt handler

Prepare:

AH = 35h
AL – interrupt number

On return:

ES:BX – pointer to interrupt handler.

8.02-30 INT 21\AH=36h – get disk's free space

Prepare:

AH = 36h
DL – logical disk's number (note 1 to 8.02-17)

On return:

AX – number of sectors per cluster
BX – number of free clusters
CX – number of bytes per sector
DX – total number of clusters on requested disk

Note 1: free space (in bytes) can be found as product AX * BX * CX.
Note 2: whole disk's space (in bytes) can be found as product AX * CX * DX.
Note 3: INT 21\AH=36h function reckons "lost" clusters among those used.
Note 4: after a request for an invalid or non-existing disk the AX = FFFFh value is returned. After a request for a CD\DVD-ROM the returned data are certainly invalid.
Note 5: requests for FAT-32 disks are allowed, but actually reported space is limited to 2048 Mb. For larger disks the INT 21\AX=7303h function (8.02-80) should be used instead.

8.02-31 INT 21\AH=39h-3Ah – create/remove an empty subdirectory

Prepare:

AH = 39h – create a subdirectory
= 3Ah – remove an empty subdirectory
DS:DX – pointer to a string with name of addressed directory. The string must end with 00h byte. Name may be preceded by a path. Maximum length of the string is 64 bytes.

On return:

On error CF flag is set, AL returns error code (A.06-1).
Clear state of CF flag signifies successful termination.
AX contents are not preserved.

Note 1: a directory can't be deleted, if it is the root directory, or if it is not free, or if it is marked as "current" in CDS table (A.03-3).
Note 2: unlike ordinary directories, root directory has a limited capacity: if it is full, then new subdirectory inside this root directory can't be created.

8.02-32 INT 21\AH=3Bh – set current directory

The INT 21\AH=3Bh function rewrites default directory pathname in a CDS table entry (A.03-3) for a particular disk.

Prepare:

AH = 3Bh
DS:DX – pointer to a string with name of new default directory. The string must end with 00h byte. Name may be preceded by a path, optionally with disk's letter-name. Maximum length of the string is 64 bytes.

On return:

On error CF flag is set, AL returns error code (A.06-1).
Clear state of CF flag signifies successful termination.
AX contents are not preserved.

Note 1: if a call for INT 21\AH=3Bh specifies a disk, which is not "current" at that moment, then "current" directory is not changed at once, but the proposed default directory will become "current", when "current" status will be assigned to the specified disk.
Note 2: the INT 21\AH=47h function (8.02-49) reports current assignment of default directory.

8.02-33 INT 21\AH=3Dh – get an access handle

Handle is a hexadecimal identifier of a SFT entry (A.01-4). Each SFT entry is associated with some object: either with an existing file, or with a dedicated region of XMS memory, or with an access channel. Handle may be regarded as a numerical reference to that object. The INT 21\AH=3Dh function investigates whether there is an entry in SFT table, associated with the specified object. If associated entry exists yet, then number of registered references in this entry is incremented by 1, number of this SFT entry is written into a byte cell in JFT table (A.07-1, offset 18h), and ordinal number of that byte cell in JFT table is returned to the caller program as the requested handle. If associated SFT entry for the specified object doesn't exist, then INT 21\AH=3Dh handler creates a new SFT entry for the specified object, and after that performs just the same described sequence of operations, which finally returns a new handle to the caller program. Those files, which have got associated handles, are known as "opened" files.

Prepare:

AH = 3Dh
AL – access and sharing rights (A.09-4)
DS:DX – pointer to object's name (ending with 00h byte)

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
AX – handle – a numerical reference to specified object

Note 1: in order to obtain a handle for driver control channel you have to specify that name, which is stored starting at offset 0Ah inside driver's header (A.05-1). Ways to locate driver's headers are described in note 2 to appendix A.01-2, in article 8.03-12 and in introduction article to part 8.03.
Note 2: wildcards in object's name are not allowed.
Note 3: on opening a file its access point is set at its first byte.
Note 4: a possibility to open a file doesn't depend on its attributes, but INT 21\AH=3Dh function can't associate handles with directories.
Note 5: INT 21\AH=3Dh handler may be called via server function INT 21\AX=5D00h (8.02-68), which gives an opportunity to specify in CL register an attribute mask (A.09-2) for the file, which is to be opened.

Note 6: handles inherited from parent program inherit the same sharing and access restrictions. Access rights byte in AL register (A.09-4) also defines whether a particular handle will be inherited or not.

Note 7: if a file is stored in a logical disk formatted with FAT-32 file system, then associated handle for this file should be provided by INT 21\AX=6C00h function (8.02-78).

8.02-34 INT 21\AH=3Eh – delete an access handle

The INT 21\AH=3Eh function decrements by 1 the number of registered references to the same object in associated SFT entry (A.01-4) and removes number of that entry from a byte cell in JFT table (A.07-1) of the caller program. If object is a file, and if it has been altered, then corresponding directory record is corrected, and contents of disk buffers are written back to disk. If caller program has no duplicate handles for the same file, the latter becomes inaccessible, or "closed" for the caller program. But operating system considers a file closed only in case of zero number of references to this file, registered in associated SFT entry: only then this file can be deleted, and only then this file ceases to be a cause of blocking its removable media in the drive. A SFT entry with zero number of references is also considered closed and invalid.

Prepare:

AH = 3Eh
BX – handle (8.02-33) to be deleted

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies successful termination.
AX contents are not preserved.

Note 1: a handle may be deleted by INT 21\AH=68h and by INT 21\AH=6Ah functions too. Other specifications for these functions (except AH) are the same.

Note 2: when a program is prepared for termination, all files, opened by this program, may be closed at once by INT 21\AX=5D01h function (8.02-69).

Note 3: handles for areas of extended memory, opened by EMM386.EXE driver (5.04-02), should be deleted by INT 67\AH=45h function (8.03-61).

8.02-35 INT 21\AH=3Fh – read data from a source

Prepare:

AH = 3Fh
BX – handle (8.02-33) to source object: file, channel or device
CX – number of bytes to read
DS:DX – pointer to a buffer for read data

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
AX – number of bytes actually read,
DS:DX – pointer to buffer filled with data.

Note 1: each program automatically inherits handle 0000h, which enables to read data from STDIN channel source, naturally, when STDIN channel is ready to supply these data. But if STDIN channel is not redirected, and its buffer is empty, then a request to 0000h handle initiates a buffer filling procedure with waiting for input from keyboard. Inputted characters are displayed on the screen, and their quantity is not limited by a number in CX register. Input terminates after ENTER keystroke, and then that number of characters, which is preset in CX register, is read from STDIN buffer into another buffer, defined by DS:DX pointer. The rest characters in STDIN buffer are available for reading by next calls for INT 21\AH=3Fh function.

Note 2: file data are read from current access position and on, so that access position is updated after each successful read operation.

Note 3: if requested number of bytes in CX register leads access position beyond file's end, then INT 21\AH=3Fh function terminates successfully, but number of read bytes, returned in AX register, will be less, than requested number of bytes in CX register.

8.02-36 INT 21\AH=40h – data transfer to a file or to a device

Prepare:

AH = 40h
BX – handle (8.02-33) to destination object: file, channel or device
CX – number of bytes to transfer
DS:DX – pointer to buffer with data to be transferred

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
AX – number of bytes actually transferred.

Note 1: each program automatically inherits 4 active handles to target channels: 0001h – to STDOUT channel, 0002h – to STDERR channel, 0003h – to STDAUX channel (port COM1), and 0004h – to STDPRN channel (port LPT1). STDOUT channel can be redirected, but its default target is display. STDERR channel can't be redirected; it always displays characters on the screen.

Note 2: writing into a file starts at current access position, defined by access position pointer in SFT (A.01-4). After each successful writing procedure the current access position is automatically updated. If writing procedure leads current access

position beyond file's end, then file's length is automatically increased. However, inflicted changes are not written to disk until file's handle isn't deleted (8.02-48).

Note 3: a call with CX = 0000h doesn't cause data transfer, but causes target file truncation (or extension) to that length, which is defined by current access position.

Note 4: on logical disks with FAT-32 file system files are allowed to grow beyond 2 Gb, if files are opened by INT 21\AX=6C00h function with TRUE state of "extended size" flag.

Note 5: if after data transfer the returned number in AX register is less then requested number in CX register, then the most probable cause of this difference is absence of free space on target disk.

8.02-37 INT 21\AH=41h – delete a closed file

Prepare:

AH = 41h
DS:DX – pointer to a string with filename. Name may be preceded by a path. String must end with 00h byte.

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies successful termination.
AX contents are not preserved.

Note 1: deleted file is not erased physically; rather its directory entry is made invalid: the first character of this entry becomes overwritten with invalidity mark – code E5h.

Note 2: deletion of any file with long filename by INT 21\AH=41h function doesn't affect associated entries beyond the main one, whereas these directory entries contain continuation of long filename.

Note 3: opened files may be deleted, but their handles will be kept active. This may cause FAT corruption and data loss. Each file, which is to be deleted, must be closed in advance (8.02-34).

Note 4: wildcards in filename are not allowed, unless INT 21\AH=41h is called via server function INT 21\AX=5D00h. Server function also allows to specify in CL register an attribute mask (A.09-2) for the files to be deleted.

Note 5: deletion of files from current directory can also be performed by INT 21\AH=13h function (8.02-13).

8.02-38 INT 21\AH=42h – set file's access point

File's access position is defined by a pointer in file's SFT entry (offset 15h in table A.01-4). Position of access point is counted from start of the file. INT 21\AH=42h function affects that pointer in file's SFT entry and thus shifts file access point.

Prepare:

AH = 42h
AL – code of origin for requested access point shift:
= 00h – start of file
= 01h – current access point position (note 1)
= 02h – end of file (note 1)
BX – file's handle (8.02-33)
CX:DX – double word shift of requested access point from that origin, which is specified by code in AL register.

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
DX:AX – new access point position counted from file's start.

Note 1: choice of origin codes 01h and 02h implies, that requested shift in CX:DX is a signed double word. Operations with signed shift potentially may set an access point ahead of file's start. In such cases INT 21\AH=42h function returns CF flag cleared, but error will evince itself at an attempt of access.

Note 2: if new access point position is beyond file's end, then the nearest next writing operation (8.02-36) will automatically extend file's length up to actual access point position.

Note 3: file's size (in bytes) is returned in DX:AX registers after a call for INT 21\AH=42h function with AX=4202h and CX=DX=0000h.

Note 4: on logical disks with FAT-32 file system file access point positions beyond 2 Gb are allowed, if files are opened by INT 21\AX=6C00h function with TRUE state of "extended size" flag.

8.02-39 INT 21\AX=4300h – get file's attributes

Prepare:

AX = 4300h
DS:DX – pointer to a string with filename. Name may be preceded by a path. String must end with 00h byte.

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
CX – file's attributes word (A.09-2)

Note 1: INT 21\AX=4301h function enables to set attributes, prepared in CX register. AX contents may be lost on return. Other features are the same.

8.02-40 INT 21\AX=4400h – get information about a handle

Handle (8.02-33) is a hexadecimal identifier of a SFT entry (A.01-4). Each SFT entry is associated with some object: either with an existing file, or with a dedicated region of XMS memory, or with an access channel. Handle may be regarded as a numerical reference to that object. The INT 21\AH=4400h function enables to elicit some properties of both the given handle itself and the object it is associated with.

Prepare:

AX = 4400h
BX – handle (8.02-33)

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
AX contents are not preserved;
DX – data word (note 1)

Note 1: clear state of bit 7 in returned data word signifies a file handle data word, which should be interpreted according to table A.04-2. Set (TRUE) state of bit 7 in returned data word signifies attribute word of a non-file object. This attribute word should be interpreted according to table A.05-2.

Note 2: attributes of channel's handle in attribute word (A.05-2) may be corrected by INT 21\AX=4401h function. States of bits 7 – 0 for attribute word should be prepared in DX register, bits 15 – 8 must be cleared, AX = 4401h, all other specifications are the same.

8.02-41 INT 21\AX=4402h-4403h – driver control data read/write

Main idea of I/O control (IOCTL) is that DOS allocates some memory space to driver's control data and enables programs to access this memory space in order to read and affect driver's control data. INT 21\AX=4402h-4403h handlers are charged with mission of access to driver's control data.

States of bits 6 and 7 in driver's attribute word (A.05-2) signify whether any particular driver supports programmable I/O control. If programmable I/O control is supported, then control data for this driver can be addressed by a handle, opened either by INT 21\AH=3Dh or by INT 21\AX=6C00h function (note 1 to 8.02-33).

Prepare:

AX = 4402h – to read driver's control data
= 4403h – to send driver's control data
BX – a handle (8.02-33) referencing target driver
CX – number of bytes to be transferred (buffer's length)
DS:DX – pointer to buffer for data or with data to be sent

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
AX – number of bytes actually transferred.

Note 1: format of control data is specific for each particular driver. For example, a set of commands for CD/DVD-ROM drivers is shown in table A.15-4.

Note 2: driver control data for disk drives (block devices) can't be addressed with a handle, but may be read by INT 21\AX=4404h and sent by INT 21\AX=4405h. Data specifications for these functions are similar to those shown above, except registers AX and BX: BL specifies disk's number (note 1 to 8.02-17), BH is ignored.

8.02-42 INT 21\AX=4406h-4407h – check of object's readiness for access

Prepare:

AX = 4406h – input (reading) readiness check
= 4407h – output (writing) readiness check
BX – handle (8.02-33) to target object.

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
AL = FFh – signifies that target object is ready for access
= 00h – signifies that target object is not ready for access.

Note 1: if file's access point is set at file's end by access point shift operation INT 21\AH=42h (8.02-38), then INT 21\AX=4406h check may erroneously report file's readiness for reading (AL = FFh). This error wouldn't happen, when file's access point is set by reading or writing operations (8.02-35, 8.02-36).

Note 2: the INT 21\AX=4407h function doesn't check whether there is a media in the drive and whether this media has free space for writing.

8.02-43 INT 21\AX=4408h – check for a removable media

Prepare:

AX = 4408h
BL – logical disk number (note 1 to 8.02-17)

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
AX = 0000h – requested logical disk is removable,
= 0001h – requested logical disk is fixed (HDD).

8.02-44 INT 21\AX=4409h – logical disk driver's attributes

Driver's attributes enable to determine whether the requested disk is real or virtual, whether it is local or network drive, is it accessible for BIOS functions or not. Interpretation of returned attribute word is shown in table A.05-2.

Prepare:

AX = 4409h
BL – logical disk number (note 1 to 8.02-17)

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
AX contents are not preserved;
DX – driver's attributes word (A.05-2).

Note 1: driver's attribute word reflects driver's capabilities, which don't necessarily correspond to properties of requested disk. For example, TRUE state of bit 11 in attribute word is interpreted as driver's ability to support removable media, but this gives no ground to conclude that the requested disk is a removable disk.

8.02-45 INT 21\AX=440Ah – check whether a handle implies remote access

Prepare:

AX = 440Ah
BX – a handle (8.02-33)

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
DX >= 8000h – signifies access via network..

8.02-46 INT 21\AX=440Dh – generic call for drive's subfunctions

The INT 21\AX=440Dh function is in fact a template for invoking a lot of IOCTL subfunctions. Type of subfunction is defined by its code in CL register. CH register contents define category code: CH = 08h corresponds to disk subfunctions, which are inherited from previous versions of DOS and can be applied to disks with FAT-12 and FAT-16 file systems. Category code CH=48h corresponds to disk subfunctions, which are introduced in MS-DOS7 and can be applied to disks with FAT-32 file system.

This article describes subfunctions with category code CH = 48h, which can be applied to disks with either of FAT-12, FAT-16, FAT-32 file systems and don't require BIOS's support for INT 13 extensions (8.01-55). It may be worth to check with INT 21\AX=4411h function (8.02-47) whether a particular IOCTL subfunction is supported in your computer.

General form of data presentation for all IOCTL subfunctions is data block pointed at by DS:DX registers. Interpretation of data in this data block for some subfunctions is shown in appendix A.04. Specific conditions for several other subfunctions are described in notes below.

Prepare:

AX = 440Dh
BL – logical disk number (note 1 to 8.02-17)
CX – subfunction:
= 4840h – set disk's parameters from a table (A.04-3)
= 4841h – write a track onto logical disk (A.04-4)
= 4842h – format and verify a track on a logical disk (A.04-5)
= 4846h – set volume's serial number (A.04-1)
= 4847h – set access flag (note 1)
= 4848h – set media lock state (note 2)
= 4849h – eject media from drive (no data block required)
= 4860h – read disk's parameters (A.04-3)
= 4861h – read a track from logical disk (A.04-4)
= 4862h – verify logical disk's track (A.04-5)
= 4866h – get volume's label and FAT type (A.04-1)
= 4867h – get access flag (note 1)
= 4868h – determine type of floppy disk (note 3)
DS:DX – pointer to data block, if it is required for subfunction

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
DS:DX – pointer to data block, if it must be returned by subfunction.

Note 1: subfunction CX=4847h accepts and subfunction 4867h returns in DS:DX data block nothing but an access byte at offset 01h. Any nonzero value means that access is allowed.

Note 2: subfunction CL=4848h accepts operation code from byte at offset 00h in DS:DX data block: 00h – lock disk, 02h – unlock disk, 03h – report lock status (just as for INT 13\AH=45h, 8.01-58). Lock status – the number of pending locks on that disk – is returned in byte 01h of the same data block.

Note 3: in byte at offset 00h in DS:DX data block subfunction CX=4868h returns code 01h, if media is of default type for the requested drive, or else code 00h, if media is of any other type. Code of particular media type is returned in byte at offset 01h in the same data block: 02h – a 720 kb diskette, 07h – a 1.44 Mb diskette, 09h – a 2.88 Mb diskette.

8.02-47 INT 21\AX=4411h – check for generic call capability

The INT 21\AX=4411h function is a check whether a specified IOCTL subfunction is supported or not supported by BIOS, by hardware and by installable drivers in a particular PC.

Prepare:

AX = 4411h
BL – logical disk number (note 1 to 8.02-17)
CX – subfunction code, just as for INT 21\AX=440Dh (8.02-46).

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
AX = 0000h – confirms that requested function is supported.

Note 1: support for INT 21\AX=4411h function itself and for some other IOCTL subfunctions can be confirmed by driver's attribute word (A.05-2).

8.02-48 INT 21\AH=45h\46h – duplicate a handle

The INT 21\AH=45h function copies a SFT entry number from one byte cell in JFT table (note 3 to A.07-1) into the nearest free byte cell of the same JFT table. Ordinal numbers of both byte cells became handles, associated with the same SFT entry and with the same "opened" object. Creation of a duplicate handle may be necessary either for saving SFT entry number, or for relocation of entry numbers in JFT table, or just in order to delete a duplicate handle, because deletion of a handle initiates file's saving to disk, while presence of another handle to the same file prevents its closure.

The INT 21\AH=46h function also copies a SFT entry number from a byte cell in JFT table, but stores the copy in a prescribed byte cell, overwriting former SFT entry number in this cell. If, for example, the 0001h handle, associated with STDOUT channel, is made a duplicate of another handle, associated with a file, then output, normally sent to display, will be redirected into this file. Just in this way DOS performs redirection of input and output data traffic (2.04-02 – 2.04-05).

Prepare:

AH = 45h – duplication, preserving current associations
= 46h – duplication, overwriting a selected association
BX – active handle (8.02-33), which is to be duplicated
CX – another handle, which is to acquire association of the handle in BX (for INT 21\AH=46h function only).

On return:

On error CF flag is set, AL returns error code (A.06-1).

Clear state of CF flag signifies successful termination, and then
AX – automatically assigned duplicate handle (after a call for INT 21\AH=45h function only).

Note 1: if a handle, specified in CX register, is associated with an opened file, then after a call for INT 21\AH=46h function this file will become closed automatically.
Note 2: shift of file access point for duplicate handle causes identical shift of file access point for the other handle, because both refer to the same SFT entry (A.01-4).

8.02-49 INT 21\AH=47h – get current directory

Prepare:
AH = 47h
DL – logical disk number (note 1 to 8.02-17)
DS:SI – pointer to 64-byte buffer for pathname.

On return:
On error CF flag is set, AL returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
AX contents are not preserved;
DS:SI – pointer to pathname, ending with 00h byte.

Note 1: returned pathname doesn't include disk's letter-name and initial backslash.
Note 2: default directory assignment may be changed with INT 21\AH=3Bh function (8.02-32).

8.02-50 INT 21\AH=48h – allot a memory block

Prepare:
AH = 48h
BX – requested block's size in 16-byte paragraphs

On return:
On error CF flag is set, AL returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
AX – segment address of the allotted block
BX – size in paragraphs of the largest available block.

Note 1: a request for BX=FFFFh can't be satisfied, but after this request INT 21\AH=48h function returns in BX register the whole available size of free conventional memory.
Note 2: requests to INT 21\AH=48h function from *.COM programs may fail with error code AL=08h ("insufficient memory"), because by default DOS allots to any currently executed *.COM program the whole free space of conventional memory (details – in note 5 to A.12-7). The *.COM programs must declare required

amount of memory with a call for INT 21\AH=4Ah function (8.02-52), otherwise all the rest conventional memory wouldn't be considered free.

8.02-51 INT 21\AH=49h – release memory block

Prepare:

AH = 49h
ES – segment address of the block to be made free

On return:

On error CF flag is set, AL returns error code (A.06-1).
Clear state of CF flag signifies successful termination.

8.02-52 INT 21\AH=4Ah – resize memory block

Prepare:

AH = 4Ah
BX – block's requested size (in 16-byte paragraphs)
ES – segment address of the block to be resized

On return:

On error CF flag is set, AL returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then:
BX – maximum number of paragraphs, available for specified memory block

Note 1: if there is no enough memory to expand the block as requested, the block will be made as large as possible.

8.02-53 INT 21\AH=4Bh – load a program for execution

Prepare:

AH = 4Bh
AL – subfunction:
= 00h – loading and initiation of execution
= 01h – loading without execution initiation
= 03h – overlay loading (exchangeable part of code)
DS:DX – pointer to a string with full specification for program call. Program's name must include suffix. The string must end with 00h byte.
ES:BX – pointer to parameters block A.07-2.

On return:

On error CF flag is set, AL returns error code (A.06-1).
Clear state of CF flag signifies successful termination.
Contents of BX and DX registers are not preserved.

Note 1: subfunction AL=00h creates a new PSP (A.07-1) for the loaded program and fills its environment segment with a copy of the caller's environment. Command line in new PSP ends with byte 00h (whereas in ordinary PSP it ends with byte 0Dh).

Note 2: caller's mission comprises a check whether there is enough memory for the loaded program.

Note 3: loading procedure is performed identically by both AL = 00h and AL = 01h subfunctions, but the latter doesn't initiate execution of the loaded program. In order to enable retarded execution initiation, inside ES:BX parameters block (A.07-2) the AL = 01h subfunction returns enter point address of the loaded program and a pointer to top of its stack.

Note 4: if an executable file starts with signature MZ or ZM, then it is loaded and executed as an *.EXE program. An executable file, which is to be executed as a *.COM program, shouldn't begin with signatures LE, LX, MP, MZ, NE, P2, P3, PE, PL, W3, W4, ZM.

Note 5: in order to execute a batch file, INT 21\AH=4Bh function has to load command interpreter COMMAND.COM (6.04) with /C parameter. Name of batch file should be specified after this parameter inside the same string, pointed at by DS:DX.

Note 6: when subfunction AL = 03h loads an overlay, it doesn't create PSP and environment, doesn't initiate execution of the loaded code, and doesn't check whether target memory area is allotted to the caller program. Subfunction AL = 03h needs other form of ES:BX parameter block: first word must be target segment address, the second word at offset 02h must be overlay relocation factor.

8.02-54 INT 21\AX=4B05h – set execution state

The INT 21\AX=4B05h function is used by programs which intercept calls for INT 21\AX=4B00h in order to prepare programs for execution, including substitution of DOS version number.

Prepare:

AX = 4B05h
DS:DX – pointer to execution state descriptor (A.07-3)

On return:

On error CF flag is set, AL returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then AX = 0000h.

Note 1: no DOS, BIOS or other software interrupts are allowed between return from INT 21\AX=4B05h call and initiation of child process.

Note 2: if DOS is running in HMA area, then A20 line is turned off on return from INT 21\AX=4B05h call.

8.02-55 INT 21\AH=4Ch – terminate execution, leaving errorlevel code

Prepare:

AH = 4Ch
AL – hexadecimal errorlevel code (note 3)

On return:

return wouldn't happen.

Note 1: all network locks, set by executed program, should be removed before applying INT 21\AH=4Ch function.

Note 2: INT 21\AH=4Ch function closes all files, opened by executed program, and releases all its memory, unless the parent PSP pointer (at offset 16h in PSP, A.07-1) points at current PSP itself. This is a symptom of permanently loaded program, for example, of command interpreter COMMAND.COM (6.04).

Note 3: specified errorlevel code is written into a word at offset 14h in DOS's swappable data area (A.01-3). Stored errorlevel code may be read later by INT 21\AH=4Dh function (8.02-56) or may be checked with "If errorlevel..." command (3.15-03).

Note 4: inside operating environment of DEBUG.EXE the INT 21\AH=4Ch function closes debugger's session and transfers control to DOS. If debugger's session has to be continued, then execution of the program under test should be terminated otherwise: either by a breakpoint or by a call for INT 20 (8.02-01).

8.02-56 INT 21\AH=4Dh – read stored errorlevel code

Prepare:

AH = 4Dh

On return:

AH – termination type:
= 00h – normal termination (8.02-01, 8.02-55)
= 01h – abort caused by CTRL-C keystroke (8.01-95, 8.02-83)
= 02h – abort caused by critical error (8.02-84)
= 03h – termination leaving resident module (8.02-23, 8.02-86)
AL – hexadecimal errorlevel code (notes 1 and 2)

Note 1: errorlevel informs about circumstances of previous program termination, except permanently loaded programs and those executed in background mode.

Note 2: errorlevel is stored in a word at offset 14h in DOS's swappable data area (A.01-3). Errorlevel is cleared automatically after each call for INT 21\AH=4Dh function; this is why it can't be read more than once. Multifold reading of errorlevel value can be performed from batch files (3.15-03).

Note 3: internal commands of COMMAND.COM interpreter don't leave errorlevel code and don't affect that code, which is left after termination of preceding program.

8.02-57 INT 21\AH=4Eh – find first matching file

The INT 21\AH=4Eh function leaves returned data in DTA area. Default DTA address is in program's PSP (A.07-1) at offset 0080h, but it may be changed by INT 21\AH=1Ah function (8.02-16). A pointer to actual DTA position is reported by INT 21\AH=2Fh function (8.02-16). Format of returned data in DTA area is shown in column F4E of table A.09-1.

Prepare:

AX = 4E00h
CH = 00h
CL – file's attribute mask (A.09-2)
DS:DX – pointer to a string with name of the file to be searched for. Name may be preceded by a path. Filemask with wildcards is allowed instead of filename. String must end with 00h byte.

On return:

On error CF flag is set, AL returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then data about first found file are returned in DTA area (A.09-1).

Note 1: bits 0 and 5 in file's attribute mask (A.09-2) are ignored. TRUE state of bits 1, 2 and 4 in file's attribute mask doesn't exclude finding of files having no corresponding attributes. TRUE state of bit 3 (volume label) excludes finding files.

Note 2: a search for requested file(s) in current directory can also be performed by INT 21\AH=11h function (8.02-11).

8.02-58 INT 21\AH=4Fh – find next matching file

Prepare:

AH = 4Fh
DTA area (A.09-1) with data left by previous search procedure

On return:

On error CF flag is set, AL returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then data about the next found file are returned in DTA area (A.09-1).

Note 1: for search continuation the INT 21\AH=4Fh function uses those data, which are returned by previous INT 21\AH=4Eh or INT 21\AH=4Fh call and since then are stored in DTA area (8.02-16). Until search isn't completed, one has to abstain from renaming, deleting and file moving operations, which alter directory entries and thus may invalidate the data left in DTA area.

8.02-59 INT 21\AH=52h – address of DOS's "List-of-Lists"

Prepare:

AH = 52h

On return:

ES:BX – pointer to DOS's "List-of-Lists " (A.01-2)

Note 1: segment address, returned in ES: register, points at that data block, which is arranged by DOS's loader IO.SYS.

Note 2: if current program is executed not in real, but in emulated DOS environment, then INT 21\AH=52h function may return an obviously invalid address in ES:BX, for example, 0000:0000h or FFFF:FFFFh.

8.02-60 INT 21\AH=54h – get state of verify flag

Prepare:

AH = 54h

On return:

AL = 00h – verify flag is turned OFF,
= 01h – verify flag is turned ON

Note 1: if verify flag is set ON, then each disk writing operation is followed by verification procedure (details – in article 3.33). Default state of verify flag is OFF.

Note 2: state of verify flag may be changed by INT 21\AH=2Eh function. It accepts the requested state of verify flag from AL register in the same form.

8.02-61 INT 21\AH=55h – create a derived (child) PSP

The INT 21\AH=55h function arranges a new PSP (A.07-1), derived from actual PSP of the caller program. Parent segment field in new PSP is filled with segment address of caller program's PSP. Pointers to INT 22, INT 23 and INT 24 handlers are written into interrupt handler's fields in new PSP. JFT table fields for inherited handles are filled with copied numbers of corresponding SFT entries (A.01-4), and reference counters in these SFT entries are incremented by 1. Arranged new PSP can be used for execution of a *.COM program.

Prepare:

AH = 55h
DX – segment address for new PSP
SI – value for PSP's memory size field at offset 02h.

On return:

AL contents may be altered.

Note 1: memory segment for the new PSP must be allotted by INT 21\AH=48h function (8.02-50). *.COM file, which is to be executed, should be written into this segment starting at offset 0100h. After that a control transfer procedure should follow, which includes redefinition of current process identifier by INT 21\AH=50h (note 1 to 8.02-73) and a CALL FAR command to offset 0100h in allotted segment.

8.02-62 INT 21\AH=56h – correction of directory entries

Correction of directory entries enables to rename files and subdirectories. Moving files from one directory into another within one logical disk also can be performed by moving directory entries. Since this doesn't imply file's copying, moving of directory entries is performed much faster.

Prepare:

AH = 56h
DS:DX – pointer to a string with a name of an existing object – file or directory (note 4). Name may be preceded by a path. String must end with byte 00h.
ES:DI – pointer to a string with a new name (note 4) or with another path. String must end with byte 00h.

On return:

On error CF flag is set, AL returns error code (A.06-1).
Clear state of CF flag signifies successful termination.

Note 1: if a name, pointed at by DS:DX, belongs to an existing file, and a name, pointed at by ES:DI, belongs to an existing directory on the same logical disk, then the entry, corresponding to specified file, will be moved into target directory.
Note 2: if a name, pointed at by ES:DI, doesn't belong to an existing object, while a name, pointed at by DS:DX, belongs to an existing object – closed file or directory, then this existing object will be renamed. Renaming of opened files is not allowed.
Note 3: INT 21\AH=56h function doesn't assign the "A" attribute to files, which have been renamed or moved.
Note 4: wildcards in names are not allowed, unless INT 21\AH=56h function is invoked via a server call INT 21\AX=5D00h (8.02-68). Besides wildcards (2.01-03), server call accepts from CL register an attribute mask (A.09-2), and marks successful renaming (or moving) of file's group by error code AL = 12h (= no more matching files).

8.02-63 INT 21\AX=5700h – date and time of file's last change

Prepare:

AX = 5700h

BX – file's handle (8.02-33)

On return:

On error CF flag is set, AL returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
CX – time of file's last change:
- bits 15 - 11 – hours from 00 to 23;
- bits 10 - 5 – minutes;
- bits 4 - 0 – seconds.

DX – date of file's last change:
- bits 15 - 9 – year, counted from 1980;
- bits 8 - 5 – month;
- bits 4 - 0 – day.

Note 1: date and time of file's last change may be set by INT 21\AX=5701h function: it accepts time from CX register and date from DX register in the same form.

Note 2: date and time of file's creation similarly may be read by INT 21\AX=5706h function and set by INT 21\AX=5707h function. Both these functions use SI register for time code in .01 second units. However, date and time of file's creation are not necessarily registered under DOS.

Note 3: date of last access to a file may be similarly read by INT 21\AX=5704h function and set by INT 21\AX=5705h function (the latter needs CX=0000h to be specified). However, registration of last access date may be prohibited by ACCDATE command (4.01).

8.02-64 INT 21\AX=5800h – memory allocation strategy

Prepare:

AX = 5800h

On return:

On error CF flag is set, AL returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
AX – current strategy code:

bits 7 and 6:
- = 00 – use conventional memory;
- = 01 – use upper memory;
- = 10 – use conventional memory, only if upper memory is unavailable;

bits 1 and 0:
- = 00 – allot first fit space;
- = 01 – allot best fit space;
- = 10 – allot last fit space.

bits 15 - 8 and 5 - 2 must be cleared to zero.

Note 1: the INT 21\AX=5801h function accepts the same strategy code in BX register and enables to set it.
Note 2: former memory allocation strategy must be restored before termination of each program, which has changed strategy settings.
Note 3: proper memory allocation strategy is necessary, but not sufficient for upper memory usage: besides that, command DOS=UMB (4.08) must be present in CONFIG.SYS file.

8.02-65 INT 21\AH=59h – extended information about the last error

Prepare:

AH = 59h
BX = 0000h

On return:

contents of CL, DX, SI, BP and DS registers are not preserved;
BH – error class (A.06-2);
BL – recommended action (A.06-3);
CH – probable error locus (A.06-4);
AX – error code (A.06-1); if AX=0022h, then
ES:DI – pointer to media identifier (note 2 to A.06-1).

Note 1: error information is read from DOS's swappable area (A.01-3).
Note 2: error information is written into DOS' swappable area by INT 21\AX=5D0Ah function; it accepts in DS:DX registers a pointer to parameters list (A.07-4), and gets data from those words in this list, which correspond to registers AX, BX, CX, DI, DX, ES.

8.02-66 INT 21\AH=5Ah – create a temporary file

Prepare:

AH = 5Ah
CX – file's attributes (A.09-2)
DS:DX – pointer to string with a path, ending with a backslash. After that backslash 13 bytes 00h must follow: it is a place for automatically generated filename.

On return:

On error CF flag is set, AL returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
AX – handle (8.02-33) for created temporary file;
DS:DX – pointer to string with path and appended filename.

Note 1: created file automatically gets a unique name, excluding name conflicts.

Note 2: capacity of root directories is limited. If disk's root directory is full, then a file in this directory can't be created.

Note 3: each temporary file must be closed and deleted before termination of that program, which has requested creation of this temporary file.

8.02-67 INT 21\AH=5Bh – create a new file

Prepare:

AH = 5Bh
CX – file's attributes (A.09-2)
DS:DX – pointer to string with a name for new file. Name may be preceded by a path. String must end with byte 00h.

On return:

On error CF flag is set, AL returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
AX – handle (8.02-33) for created new file.

Note 1: the INT 21\AH=5Bh function can't create new file, if a file with the same name exists yet in specified directory.

Note 2: capacity of root directories is limited. If disk's root directory is full, then a file in this directory can't be created.

Note 3: the INT 21\AH=3Ch does the same and accepts the same specifications (except AH), but doesn't return error if a synonymous file is present in the same directory. This synonymous file will be just truncated to zero length, and a pointer to its first cluster will be lost. Therefore restoration of a truncated file is a much more complex procedure, then restoration after occasional ordinary deletion by functions INT 21\AH=13h (8.02-13) or INT 21\AH=41h (8.02-37).

8.02-68 INT 21\AX=5D00h – server function call

The INT 21\AX=5D00h function presents a template, enabling to call any other function of INT 21 handler and execute this function as a separate process with opportunities for selective and repeated execution. In particular, server call for INT 21\AH=3Dh function enables to specify an attribute mask for the target file. Besides that, server calls for INT 21\AH=56h and INT 21\AH=41h functions enable to rename and to delete files, specified by filemasks with wildcards (2.01-03).

Prepare:

AX = 5D00h
DS:DX – pointer to data block (A.07-4), specifying states of all registers, which should be prepared for execution of the requested function.

On return:

– as should be returned by the requested function.

Note 1: validity of initial data in data block is not checked. Computer may get hanged, if data block specifies invalid number of requested function for AH register.
Note 2: filenames, required for INT 21 calls, must be prepared with full path in canonical form by INT 21\AH=60h function (8.02-72).

8.02-69 INT 21\AX=5D01h – close all files for a process

The INT 21\AH=5D01h function closes files, opened by caller process, updates all relevant directory entries and writes disk buffer's contents back to disk. If there are opened files, accessed via a network, then network service function INT 2F\AX=1107h is automatically called for.

Prepare:

AX = 5D01h
DS:DX – pointer to data block (A.07-4), specifying virtual machine identifier at offset 12h and process identifier at offset 14h. Contents of register's data fields in data block are ignored.

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies successful termination.

8.02-70 INT 21\AX=5D06h – address of swappable data area (SDA)

DOS functions store caller's status information: it may be needed later for these or other DOS functions, called by the same process. However, current process may be interrupted by other process – an invoked TSR program or interrupt handler. DOS functions, called by this other process, store other status information and corrupt data, related to interrupted process. Corrupted data prevent proper resumption of interrupted process. In order to avoid such conflicts relevant data should be saved and later restored. Therefore MS-DOS stores relevant data in SDA – Swappable Data Area (A.01-3). INT 21\AX=5D06h function returns address of SDA area and size of that data block, which should be saved in particular circumstances.

Prepare:

AX = 5D06h

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
DS:SI – pointer to start of SDA area (A.01-3),
CX – size in bytes of the whole SDA area, including status data and DOS's stacks. Whole SDA area must be saved in order to enable proper resumption of interrupted DOS function.

DX – size in bytes of status data part of SDA area, which should be saved when interrupted process is not a DOS function.

Note 1: saving of SDA's data is not needed, if interrupting process doesn't call for DOS functions.

Note 2: INT 21\AX=5D06h is also a DOS function, potentially able to damage data in SDA area, when interrupt has happened yet. Therefore INT 21\AX=5D06h function should be called beforehand, during initiation of driver or of resident program. Returned information should be stored and kept ready for future use.

8.02-71 INT 21\AX=5F08h – hide a logical disk

Prepare:

AX = 5F08h
DL – logical disk number (note 1 to 8.02-10)

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies successful termination.

Note 1: INT 21\AX=5F08h function checks disk's parameters in DOS's internal tables. Having found disk's parameters invalid, the INT 21\AX=5F08h function fails.

Note 2: INT 21\AX=5F07h function accepts the same specifications (except AX), but performs reverse operation: turns a hidden disk into valid disk.

8.02-72 INT 21\AH=60h – convert filename or path into canonical form

Effect of INT 21\AH=60h function is identical to effect of TRUENAME command, described in article 3.29. Transformation of filename and path into canonical form is necessary for several DOS functions and enables to avoid errors, which otherwise may cause undesirable consequences.

Prepare:

AH = 60h
DS:SI – pointer to a string with a name, which may be preceded by a path. Maximum length of the string is 64 bytes. String must end with 00h byte.
ES:DI – pointer to a 128-byte buffer for transformed name and path.

On return:

On error CF flag is set, pointers with buffer contents are preserved, and AX register prompts possible cause of error:
AX = 0002h – some component of path is invalid or absent
= 0003h – wrong composition or invalid disk's letter-name.
Clear state of CF flag signifies successful termination, and then

ES:DI – pointer to buffer with transformed specification
AX contents may be lost.

Note 1: actual existence of specified name and path is not checked.
Note 2: INT 21\AH=60h function can't be applied to network paths.

8.02-73 INT 21\AH=62h – get current PSP address

DOS regards current PSP address as an identifier of the current process. Replacement of current process identifier with another one is the key operation in multitasking execution control. Identifier replacement implies reading the current identifier from SDA area (A.01-3) with INT 21\AH=62h function, saving the current identifier, and then writing a new identifier into SDA area. Having finished its job, program must restore the former process identifier before it returns control back. There are some other reasons to call for INT 21\AH=62h function in order to determine current PSP address (an example – in 9.07-02).

Prepare:

AH = 62h

On return:

BX – segment address of PSP (A.07-1) for the current process.

Note 1: new process identifier may be written into SDA area with INT 21\AH=50h; the latter accepts prepared new identifier from BX register. Besides BX and AH=50h, no other data are needed.

8.02-74 INT 21\AX=6501h – country information

Prepare:

AX = 6501h
BX – hexadecimal codepage number, or BX=FFFFh for a request about current codepage.
CX – size of prepared buffer for data, not less than 29h bytes
DX – country identifier, or DX=FFFFh for current country request
ES:DI – pointer to prepared buffer for data

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
ES:DI – pointer to buffer filled with data (A.02-4)
CX – size (in bytes) of buffer's filled part.

Note 1: functions INT 21\AX=6502h, 6504h, 6505h, 6506h use the same specifications (except AX), but return in ES:DI buffer only one 4-byte address at offset 01h:

– INT 21\AX=6502h returns a pointer to uppercase table, which begins with its size (1 word), followed by 128 uppercase equivalents (in there are any) of characters from 80h to FFh.
– INT 21\AX=6504h returns a pointer to filename uppercase table, which has the same structure, but is applied to filenames only.
– INT 21\AX=6505h returns a pointer to filename restrictions table (A.02-5).
– INT 21\AX=6506h returns a pointer to collating sequence table, which begins with its size (1 word), followed by 256 bytes, defining order of sorting for characters from 00h to FFh.

Note 2: information concerning other countries and other code tables, not installed at the current moment, is not available, unless the NLSFUNC.EXE resident program (5.02-03) is installed.

Note 3: country information in MS-DOS7 may be set with INT 21\AX=7002h function. It accepts in DS:SI a pointer to data table (A.02-4), in CX – length of that table (normally 0026h bytes). If CF is clear on return, then CX – length of actually set data. Set state of CF flag signifies an error: AX = 7000h means that function is not supported; other error codes correspond to table A.06-1.

8.02-75 INT 21\AX=6521h – country-dependent string capitalization

Prepare:

AX = 6521h
CX – length of string to capitalize
DS:DX – pointer to the string to capitalize

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies success, submitted string is capitalized.

Note 1: the INT 21\AX=6522h function does the same, but ignores CX and requires the end of string to be marked with byte 00h.

Note 2: INT 21\AX=6520h function is used to capitalize a single character, accepted and returned in DL. CX and DS contents are ignored.

8.02-76 INT 21\AH=67h – set size of handle table

By default not more than 20 handles can be kept active simultaneously, because size of original JFT (offset 18h in A.07-1) is limited to 20 bytes. The INT 21\AH=67h function creates a new JFT table outside PSP, thus removing default restriction, caused by limited length of original JFT.

Prepare:

AH = 67h
BX – number of handles in new JFT table for current process

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies successful termination.

Note 1: if current JFT is inside PSP, and new size in BX is no more than 20 bytes, then no action is undertaken.

Note 2: if current JFT outside PSP contains no more than 20 active handles, and new JFT size in BX is no more than 20 bytes, then JFT is copied back into PSP.

Note 3: if a reduced JFT size is requested, and active handles can't fit into this reduced size, then INT 21\AH=67h function fails with error code 0004h (= too many opened files).

Note 4: number of simultaneously opened files is limited not by JFT table only, but also by SFT table (A.01-4). Length of the latter is defined by command FILES (4.12) in configuration file CONFIG.SYS.

Note 5: irrespective of current JFT size and location, the child process can't inherit more than 20 active handles from its parent process.

8.02-77 INT 21\AX=6900h – get volume label and FAT type

The same data block A.04-1 with disk's volume specifications is returned by both INT 21\AX=440Dh\CX=4866h and INT 21\AX=6900h functions, except that the latter in case of access failure doesn't call for critical error handler INT 24 (8.02-84). All errors are reported via their code returned in AX.

Prepare:

AX = 6900h
BH = 00h
BL – logical disk number (note 1 to 8.02-17)
DS:DX – pointer to buffer 19h bytes long for data block

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
DS:DX – pointer to buffer with written data block (A.04-1)
AH contents may be altered.

Note 1: data starting at offset 02h in returned data block are copied from bytes at offsets 27h – 3Dh in extended BPB block (A.03-4).

Note 2: error code 0005h is reported if extended BPB is not found in the requested disk.

Note 3: the INT 21\AX=6900h function can't be applied to network drives. Such calls return error code 0001h.

Note 4: INT 21\AX=6901h function performs the reverse operation: it accepts data block (A.04-1), pointed at by DS:DX, and copies disk's volume data from this block

into extended BPB of that disk, which is similarly specified by its number in BX register.

8.02-78 INT 21\AX=6C00h – extended get handle function

A handle for access to an object can be obtained with both INT 21\AH=3Dh (8.02-33) and INT 21\AX=6C00h functions, but the latter provides extended capabilities to define handle's properties and prescribed actions.

Prepare:

AX = 6C00h
BH – properties flags:
bit 4 – allow file's size above 2GB (for FAT-32 only)
bit 5 – return error rather than call for INT 24h
bit 6 – write to disk immediately, bypass cache buffer
BL – access and sharing conditions (A.09-4)
CX – file's attributes (A.09-2), if file is to be created
DH = 00h
DL – prescribed action code:
= 01h – open an existing file, fail if it doesn't exist;
= 10h – open a new file, fail if a synonymous file exists;
= 11h – open a file; create it anew, if it doesn't exist;
= 12h – open a new file; if synonymous file exists, remove it.
DS:SI – pointer to a string with filename, optionally preceded by a path. Wildcards in filename are not allowed. The string must end with byte 00h.

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
AX – returned handle to opened file
CX – status code:
= 0001h – file is opened;
= 0002h – file has been created;
= 0003h – file has been replaced.

Note 1: prescribed action DL = 11h is not supported on remote disks.
Note 2: operations with remote disks don't return status code in CX.
Note 3: if a file is created anew, then contents of both BH and BL are copied into a new SFT entry (A.01-4).
Note 4: on opening a file its access pointer is set to file's start.
Note 5: opportunity to open a file doesn't depend on its attributes.

8.02-79 INT 21\AX=7302h – copying of extended DPB

The INT 21\AX=7302h function copies Disk Parameters Block (DPB) into a prepared buffer. Requested DPB (A.03-1) may belong to a disk formatted with either FAT-12, FAT-16 or FAT-32 file system.

Prepare:

AX = 7302h
DL – logical disk number (note 1 to 8.02-17)
CX – length of prepared buffer, not less than 3Dh bytes
ES:DI – pointer to prepared buffer for DPB

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies success, DPB table is copied (note 2).

Note 1: unlike similar functions INT 21\AH=1Fh and INT 21\AH=32h (8.02-24), the INT 21\AX=7302h function is not available in previous DOS versions and fails, if PC's BIOS doesn't support INT 13 extensions (8.01-55).
Note 2: the word at offset 00h in returned data block is its length; a copy of DPB (A.03-1) starts at offset DI+02.
Note 3: place for driver's header address at offset 13h in returned copy of DPB is filled with FFFFh.
Note 4: the INT 21\AX=7302h function attempts to read BPB on the requested disk in order to update DPB. If access fails, INT 24 handler is called for.

8.02-80 INT 21\AX=7303h – get free space table

The INT 21\AX=7303h function reports free space on disks formatted with either FAT-12, FAT-16 or FAT-32 file systems. Returned data block (A.13-7) is copied into prepared buffer. Unlike values, returned by INT 21\AH=36h function (8.02-30), values in data block A.13-7 are not limited to 2048 Mb.

Prepare:

AX = 7303h
CX – length of prepared buffer, not less than 34h bytes
DS:DX – pointer to string, defining the requested disk (note 2)
ES:DI – pointer to buffer for data block; a word at offset 02h in this buffer must be filled with 0000h beforehand.

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
ES:DI – pointer to buffer filled with data block (A.13-7),
AL – size of the returned data block.

Note 1: unlike similar function INT 21\AH=36h, the INT 21\AX=7302h function is not available in previous DOS versions and fails, if PC's BIOS doesn't support INT 13 extensions (8.01-55).

Note 2: prepared string must define the requested disk just as it is defined in CDS table (A.03-3): for example, C:\ for local disks and \\SERVER\Share for disks, accessed via a network. Prepared string must end with byte 00h.

8.02-81 INT 21\AX=7305h – extended read/write operations

The INT 21\AX=7305h function presents 5 subfunctions for performing reading and writing operations inside logical disks, formatted with either FAT-12, FAT-16 or FAT-32 file systems. Similarly to INT 25 and INT 26 handlers (8.02-85), subfunctions of INT 21\AX=7305h ignore file structure and are addressed by sector numbers, counted separately for each logical disk from its start.

Prepare:

AX = 7305h
DL – logical disk number (note 1 to 8.02-17)
DS:BX – pointer to disk address packet (note 2)
SI – subfunction:
= 0000h – reading
= 0001h – writing any non-specific data
= 2001h – writing FAT data
= 4001h – writing directory data
= 6001h – writing file data

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies successful termination.

Note 1: unlike similar functions INT 25 and INT 26 (8.02-85), the INT 21\AX=7302h function is not available in previous DOS versions and fails, if PC's BIOS doesn't support INT 13 extensions (8.01-55).

Note 2: disk address packet is 10 bytes long and contains at offset:
00h – double word: operation start sector number;
04h – a word: number of sectors to be read or written;
06h – double word: address of buffer with data or for data.

8.02-82 INT 22 – address of control return point

Address in INT 22 cell of interrupt table belongs not to a handler, but to a return point inside parent process. When current program terminates, control is to be transferred just to INT 22 address of this return point. Most often INT 22 points at command following that

call for INT 21\AH=4Bh function (8.02-53), which has been used to launch current program.

Interrupt INT 22 shouldn't be called directly, because parent process can't proceed normally without restoration of many necessary conditions, including state of stack and addresses in INT 22 – INT 24 cells of interrupt table. All necessary preparations are performed by program termination handlers INT 20 (8.02-01) and INT 21\AH=4Ch (8.02-55). These handlers read INT 22 address from interrupt table, temporary save it, copy parent's control return point address from offset 0Ah in current program's PSP (A.07-1) into the same INT 22 cell, and then execute a JMP FAR command (7.03-39) to temporary saved address.

8.02-83 INT 23 – CTRL-C/CTRL-Break handler

Each time when keyboard controller reports about CTRL-C/CTRL-Break keystroke, INT 09 handler calls for INT 1B, and the latter sets TRUE state to a flag in BIOS data area (at offset 71h in table A.02-3). State of this flag is checked by some MS-DOS functions, called by current program. If flag is found set to TRUE state, then INT 23 handler is called for, which suspends execution of current program.

Code of INT 23 handler is written in presumption, that its caller is a DOS function. Therefore calls for INT 23 from application programs are not allowed, except indirect calls via INT 1B (8.01-95).

Further events partially depend on circumstances, but in most cases the user has to choose: either continue or terminate execution of the suspended program. Possible user's actions in response are described in article 1.03. If termination alternative is chosen, then INT 23 handler closes all files, opened by suspended program, releases its memory, defines zero errorlevel, sets CF flag into CY state, and after that transfers control to parent process – that one, which has called current program. If user chooses continuation, then states of all registers and flags are restored, and control is transferred to that DOS function, which has called INT 23 handler. Having finished its mission, this DOS function returns control back to its caller – to the current program.

8.02-84 INT 24 – critical error handler

When inadequate responses are returned to DOS function's requests, then these DOS functions call for INT 24 – critical error handler. The INT 24 handler analyses information about each error, but most often doesn't rely upon itself, and addresses its famous question "Abort, Retry, Fail?" to the user. Having got user's decision, INT 24 handler transfers it for execution to that DOS function, which has called for INT 24.

Code of INT 24 handler is written in presumption, that its caller is a DOS function. Therefore direct calls for INT 24 from application programs are not allowed.

On return states of all registers (except AL) are restored from the stack, and AL returns action code:

AL = 00h – return to caller program with error code (fail);
= 01h – make one more attempt of the same request (retry);
= 02h – terminate execution of the caller program (abort);
= 03h – admit system failure and halt the CPU.

Note 1: automatic answer FAIL and non-stop continuation of execution can be ensured by set state of flag at offset 2Ah in SDA (A.01-3). In particular, this flag is set by command interpreter COMMAND.COM, when it is launched with undocumented parameter /f (6.04).

8.02-85 Direct disk read (INT 25) and write (INT 26)

NT 25 and INT 26 handlers enable access to logical disks, formatted with file systems FAT-12 and FAT-16. Disk access is addressed by sector numbers, counted separately within each logical disk from sector number 00000000h and on. Data buffer address and particular sector numbers are transferred inside disk address packet, described in note 2 to 8.02-81.

Prepare:

AL – logical disk number (note 1 to 8.02-10)
CX = FFFFh (note 3)
DS:BX – pointer to disk address packet (note 2 to 8.02-81).

On return:

States of BX, CX, DX, DI, SI registers are not preserved.
On error CF flag is set, AL returns error code (A.06-1), AH returns error status according to table A.06-5.
Clear state of CF flag signifies successful termination.

Note 1: INT 25 and INT 26 handlers leave a word of flag's states in top of stack. The caller program has either to pop this word out of stack or to restore former value in SP register.

Note 2: INT 25 and INT 26 handlers are able to provide access to logical disks with 32-bit data transfer, marked by TRUE state of bit 1 in driver's attribute word (A.05-2).

Note 3: being applied to small logical disks without cluster structure (32 Mb or less, ID 04h in table A.13-6), INT 25 and INT 26 handlers employ different form of specification:

CX – number of sectors to be read or written
DX – number of start sector
DS:BX – pointer to a buffer with data or for data.

Note 4: INT 25 and INT 26 handlers can't be applied to logical disks formatted with FAT-32 file system, the INT 21\AX=7305h handler (8.02-81) should be used instead.

8.02-86 INT 27 – terminate execution, leaving resident module

Prepare:

CS – segment address of current program's PSP
DX – size of resident module, from 60h to FFF0h bytes, counted from start of current program's PSP

Note 1: the INT 27 handler transfers control to the parent process – that one, which has launched the current program. INT 27 handler makes necessary preparations for control transfer: releases main program's memory (except its resident module), restores pointers in interrupt table. But INT 27 handler doesn't close opened files: this must be done by current program itself beforehand.

Note 2: the INT 21\AH=31h function (8.02-23) has the same mission, but doesn't limit resident module's size at 64 kb and enables to leave non-zero errorlevel values.

8.02-87 INT 28 – DOS idle hook for background execution

The INT 28 interrupt is invoked at a pace of timer's tick (18 times per second) while any of DOS's character input functions (INT 21\AH=01h-0Ch) is waiting for input from keyboard. DOS is actually idle during such waiting intervals. Default INT 28 handler is a single IRET instruction (7.03-30), which simply returns control back to the caller. Interception of INT 28 enables to activate a background program or a TSR while the foreground program is waiting for user input. Intercepting INT 28 handler gets SS:SP registers pointing at top of DOS's stack and has to restore states of all registers on return.

Note 1: a program, which intercepts INT 28, must check the state of InDOS flag (at offset 01h in A.01-3). Address of that flag must be found beforehand with a call for INT 21\AH=34h (8.02-28) function and must be stored since program's initialization. At the moment of INT 28 call the value of InDOS flag normally is 01h; if it has larger value, then all calls for DOS functions from intercepting program are prohibited.

Note 2: programs, designed for background execution, when InDOS flag has its normal value 01h, are allowed to call for DOS functions, except INT 21\AH=01h-0Ch functions and except calls, addressed to CON device handles (normally these are handles 0000h – 0002h).

8.02-88 INT 29 – non-redirectable console output

The INT 29 handler is used to display messages on the screen, when STDOUT output is (or may be) redirected elsewhere – to a file or a device, other than display (CON) device.

Prepare:

AL – ASCII code of the character to be displayed

On return:

BX register contents may be altered.

Note 1: default INT 29 handler sends character's code to display via BIOS's function INT 10\AH=0Eh (8.01-21).

Note 2: bit 4 in CON device driver's attribute word (A.05-2) indicates whether the CON device driver supports INT 29.

Note 3: non-redirectable display of a character string also can be performed via STDERR channel (handle 0002h), addressed by INT 21\AH=40h function (8.02-36). Unlike INT 29, STDERR channel is always supported by CON device driver.

8.02-89 INT 2E – transmit a command to interpreter

The INT 2E handler transfers a command line for execution to command interpreter COMMAND.COM, but doesn't load interpreter's module anew. Command line is transferred to that interpreter's resident module, which is loaded yet and which has launched the caller program.

Prepare:

DS:SI – pointer to command line to be executed. Format of command line must be identical to that in PSP (A.07-1, offset 80h): first byte is line's length, then follows command line itself, terminated with byte 0Dh. Length count doesn't include terminating byte.

On return:

AX – error code (A.06-1);

Contents of all registers, except CS:IP, can be altered..

Note 1: resident module of command interpreter may need to load transient portion of COMMAND.COM interpreter for execution of specified command. Caller program must ensure, that DOS will be able to allocate sufficient memory for transient portion of COMMAND.COM interpreter.

Note 2: being invoked by INT 2E, COMMAND.COM interpreter works with its original environment, which may differ from the caller's environment. Hence all changes in environment variables, if there are any, will not be accessible for the caller.

Note 3: INT 2E shouldn't be called by programs, which are launched from a batch file.

8.03 Interrupt handlers, loaded by drivers and TSR programs

When DOS starts execution of application program, then pointers to BIOS's and DOS's interrupt handlers are certainly written yet in their cells of interrupt table. Application programs don't need to check presence of these pointers in their cells. This is not true for those handlers, which are loaded by optional software – drivers and TSR programs. Corresponding cells in interrupt table may be left empty, and application programs must check, whether these cells are filled with valid pointers.

Long time ago, when drivers were not numerous, each driver used to have its own cell in interrupt table. Then validity of handler's address could be confirmed by presence of a specific signature in some vicinity of addressed point. This manner of confirmation may be applied, for example, to resident module of EMM386.EXE driver (note 1 to 8.03-62), whose ancestors are known since 1983.

As more and more drivers appeared, shortage of space in interrupt table began to cause conflicts. Besides that, it was desirable to eliminate threat of hanging, when the addressed resident module isn't loaded. As a solution of both these problems an idea of multiplex interrupt INT 2F has been suggested. Every participating driver had to intercept INT 2F calls, thus forming a link in a chain of interceptors (details – in A.07-5). A call for INT 2F has to pass along this chain from one handler to the other, and each handler analyses an identifier in AH register. If a handler doesn't recognize its own identifier in AH, it leaves AX intact and lets this call to "travel" further along the chain. If specified identifier isn't recognized by all handlers, then the call for INT 2F reaches the last chain's link – a IRET command, initially set by DOS. IRET command returns control to the caller program, so that AX value is returned unchanged. A change of AX contents can be regarded as an evidence of that some handler has recognized its own identifier; hence, the requested resident module is loaded yet.

Naturally, multiplex interrupt is suitable not only for recognition. It enables to address main functions of optional resident modules without risk of hanging when required modules are not loaded. However, tracing a long chain of references takes time and makes execution slower. In order to avoid delays many drivers return addresses of their entrance points via INT 2F. Direct calls to these entrance points invoke specific functions of resident modules much faster, than via multiplex interrupt. For example, multiplex interrupt INT 2F is used by HIMEM.SYS driver (5.04-01) in order to return address of its entrance point (8.03-23).

Practice of INT 2F usage has got a limited success for some time, but it couldn't eliminate conflicts caused by inconsistent appointment of identifiers to different drivers and resident modules. In order to avoid such conflicts one more idea has been suggested – dynamic assignment of identifiers: each resident module should assign to itself the first free identifier it finds. This idea is implemented by multiplex interrupt INT 2D (details – in

A.07-6). Now resident DOS's software has got rid of identification conflicts owing to multiplex interrupt INT 2D.

Meanwhile many drivers with well known identifiers continue to use multiplex interrupt INT 2F. Moreover, several resident modules, which were loaded by drivers in previous DOS versions, now are integrated into DOS kernel together with their INT 2F identifiers for the sake of preserving compatibility. There is no sense in testing presence of resident modules with INT 2F identifiers AH = 05h, 08h and 12h, which actually are now DOS functions. Nevertheless such integrated functions are described below in part 8.03 among functions of optionally loaded software, because otherwise convenience of their ordinal search will be lost.

8.03-01 INT 2F\AX=0501h – convert error code into message

Prepare:

AX = 0501h
BX – error code to be converted (A.06-1)

On return:

Contents of AX, DI, ES registers and states of flags are not preserved.
Set state of CF flag means, that DOS has no message for submitted code.
Clear state of CF flag signifies successful termination, and then
ES:DI – pointer to read-only message, ending with byte 00h;
AL = 00h – message needs to be complemented with disk's letter-name;
= 01h – returned message is complete.

8.03-02 INT 2F\AX=0801h – accept DDT for a logical disk

The INT 2F\AX=0801h function appends DOS's chain of disk data tables (DDT) with a prepared DDT table, and modifies disk description flags in other tables, referencing the same physical drive. Since that moment the added logical disk becomes accessible by means of block device drivers, integrated into the DOS's core.

Prepare:

AX = 0801h
DS:DI – pointer to the prepared DDT table (A.03-2)

On return:

Contents of AX, BX, SI, ES registers are not preserved.

Note 1: the INT 2F\AX=0801h function shouldn't be applied to IFS, remote and other disks, which can't be served by native DOS's block device drivers.
Note 2: an example of DDT table can be obtained via INT 2F\AX=0803h function.

8.03-03 INT 2F\AX=0802h – send a request to block device driver

The INT 2F\AX=0802h function executes requests concerning logical disks, represented by valid disk data tables (A.03-2) and controlled by block device drivers, integrated into DOS's core. Both addressed disk number and code of the requested operation (A.05-3) should be specified inside a request data block (A.05-4).

Prepare:

AX = 0802h
ES:BX – pointer to request data block (A.05-4)

On return:

ES:BX – pointer to request data block, updated according to the requested operation (A.05-3 – A.05-7)

Note 1: INT 2F\AX=0802h function leaves a word of flag's states in top of stack. The caller program has either to pop this word out of stack or to restore former value in SP register.

Note 2: possible errors are announced by error codes, returned in bytes at offsets 03h and 04h in request data block (A.05-4). In any case critical error handler (INT 24) is not invoked.

8.03-04 INT 2F\AX=0803h – get address of the first DDT

Prepare:

AX = 0803h

On return:

DS:DI – pointer to start of the first DDT table (A.03-2)

Note 1: a chain of DDT tables can be easily traced through, since the first double word in each DDT table is a pointer to the next DDT table (A.03-2).

8.03-05 INT 2F\AX=1202h – get a pointer to interrupt handler's address

Prepare:

AX = 1202h
interrupt number should be in top of stack

On return:

ES:BX – pointer to interrupt handler's address;
contents of AX register may be altered;
state of stack is returned unchanged.

Note 1: while CPU is in real mode, the INT 2F\AX=1202h function just multiplies interrupt number by 4.

8.03-06 INT 2F\AX=1212h – determine length of a string

Prepare:

AX = 1212h
ES:DI – pointer to ASCIIZ string, ending with 00h byte.

On return:

CX – length of the string, including terminating 00h byte.

Note 1: the INT 2F\AX=1225h function does the same, but accepts the pointer from DS:SI registers pair.

8.03-07 INT 2F\AX=1213h – uppercase a character

Prepare:

AX = 1213h
ASCII code of the character should be in top of stack.

On return:

AL – upper case ASCII code of the same character.
State of stack is returned unchanged.

8.03-08 INT 2F\AX=1214h – comparison of far pointers

Prepare:

AX = 1212h
DS:SI – first pointer
ES:DI – second pointer

On return:

ZF flag set and CF flag clear if pointers are equal;
ZF flag clear and CF flag set if pointers differ.

8.03-09 INT 2F\AX=1216h – get a pointer to SFT entry

The INT 2F\AX=1216h function accepts a number of SFT entry (A.01-4) and returns a pointer to that SFT entry, thus giving an opportunity of direct access to SFT.

Prepare:

AX = 1216h
BX – number of the requested SFT entry (note 2)

On return:

AX value is not preserved.
Set state of CF flag signifies mission's failure.
Clear state of CF flag signifies successful termination, and then:
ES:DI – pointer to requested SFT entry,
BX – relative number of the same entry in a particular SFT.

Note 1: most probable cause of error is a request for SFT entry number greater than maximum number of SFT entries, specified by FILES command (4.12) in CONFIG.SYS file.

Note 2: a pointer to number of SFT entry, related to a particular handle, is returned by INT 2F\AX=1220h function (8.03-11).

8.03-10 INT 2F\AX=121Eh – comparison of filenames

Prepare:

AX = 121Eh
DS:SI – pointer to the first filename, ending with 00h byte;
ES:DI – pointer to the second filename, ending with 00h byte.

On return:

ZF flag is set if filenames are equivalent,
ZF flag is clear if filenames differ.

8.03-11 INT 2F\AX=1220h – get a pointer to JFT entry

The JFT table (note 3 to A.07-1) is filled with numbers of SFT entries (A.01-4), defining those objects, which are opened for access – channels or files. Access to these objects is performed via handles – numerical references, known to the caller program. A path from a handle to corresponding SFT entry starts with a call for INT 2F\AX=1220h function, returning a pointer to number of that corresponding SFT entry. This number is the main data item in specification of a call for INT 2F\AX=1216h function (8.03-09), which returns address of corresponding SFT entry.

Prepare:

AX = 1220h
BX – an active (opened) handle

On return:

On error CF flag is set, AX = 06h (– invalid handle error).
Clear state of CF flag signifies successful termination, and then:
ES:DI – address of that byte in JFT, where the required number of SFT entry is stored.

Note 1: the FFh value of JFT byte, pointed at by address in ES:DI registers, signifies inactive (closed) state of specified handle.

8.03-12 INT 2F\AX=122Ch – enter device driver chain

The INT 2F\AX=122Ch function enables to trace the whole chain of device driver headers, because the first double word in each header is a pointer to the next header. The last header is marked with first word FFFFh.

Prepare:

AX = 122Ch

On return:

BX:AX – pointer to header of the second driver (the first is NUL device driver in DOS's "List-of-Lists", A.01-2).

8.03-13 INT 2F\AX=1500h – get number of CD/DVD drives

Prepare:

AX = 1500h
BX = 0000h

On return:

AL = FFh – signature of successful termination (note 1);
BX – number of CD/DVD drives;
CX – drive number, assigned to the first CD/DVD drive (0002h = C:, 0003h = D:, and so on.)

Note 1: returned initial value AL = 00h signifies, that resident module of CD/DVD file system translator (5.08-03 or 5.08-04), which has to perform INT 2F\AX=1500h function, is not loaded.

Note 2: the INT 2F\AX=1500h function conflicts with GRAPHICS.COM driver's functions and, besides that, may return incorrect letter-name of the first CD/DVD drive when INTERLINK.EXE network driver is installed.

8.03-14 INT 2F\AX=1501h – get CD/DVD driver header address

Prepare:

AX = 1501h
ES:BX – pointer to a buffer (5 bytes per disc expected)

On return:

ES:BX – pointer to buffer filled with 5-bytes blocks for each drive. In each block the first byte – disc's subunit number, related to a particular driver, the following double word – pointer to header of that driver. Whole length of data in buffer depends on number of CD/DVD drives, returned by INT 2F\AX=1500h function (8.03-13).

Note 1: the INT 2F\AX=1501h function, performed by resident module of CD/DVD file system translator (5.08-03 or 5.08-04), shouldn't be called for, unless INT 2F\AX=150Bh function (8.03-17) confirms in advance, that required module is loaded yet.

Note 2: being called inside "DOS box" under WINDOWS OS, the INT 2F\AX=1501h function returns AX=0000h and invalid header's addresses.

8.03-15 INT 2F\AX=1505h – read CD/DVD's table of contents

Prepare:

AX = 1505h
CX – CD/DVD drive number (0002h = C:, 0003h = D:, and so on)
DX – sector index (note 2)
ES:BX – pointer to a prepared 2048-byte buffer.

On return:

On error CF flag is set, AL – error code:
AL = 15h – invalid drive number;
= 21h – drive is not ready or there is no media in the drive.
Clear state of CF flag signifies successful termination, and then:
AL – volume descriptor type:
= 01h – standard volume descriptor;
= FFh – the last volume descriptor;
= 00h – any other volume descriptor type.
ES:BX – pointer to 2048-byte buffer with table of contents.

Note 1: the INT 2F\AX=1505h function, performed by resident module of CD/DVD file system translator (5.08-03 or 5.08-04), shouldn't be called for, unless INT 2F\AX=150Bh function (8.03-17) confirms in advance, that required module is loaded yet.

Note 2: a CD/DVD disc may have several volume descriptors: sector index 0000h corresponds to the first descriptor, sector index 0001h corresponds to the second descriptor, and so on.

8.03-16 INT 2F\AX=1508h-1509h – absolute CD/DVD read/write

Prepare:

AX = 1508h – to read CD/DVD sectors
= 1509h – to write CD/DVD sectors (note 3)
CX – CD/DVD drive number (0002h = C:, 0003h = D:, and so on)
DX – number of sectors to be read or written
ES:BX – pointer to buffer (with data for write function)
SI:DI – starting sector number

On return:

On error CF flag is set, AL – error code:
AL = 0Fh – invalid drive;
= 15h – drive is busy or there is no media in the drive.
Clear state of CF flag signifies successful termination, and then after reading operation ES:BX buffer is filled with read data.

Note 1: the INT 2F\AX=1508h-1509h functions, performed by resident module of CD/DVD file system translator (5.08-03 or 5.08-04), shouldn't be called for,

unless INT 2F\AX=150Bh function (8.03-17) confirms in advance, that required module is loaded yet.

Note 2: being called inside "DOS box" under WINDOWS OS, the INT 2F\AX=1508h-1509h functions always return error code AL=15h.

Note 3: early versions of CD/DVD file system translator programs (5.08-03 and 5.08-04) didn't support writing operation. Besides that, write function needs to be supported by CD/DVD drive's hardware.

8.03-17 INT 2F\AX=150Bh – request about a CD/DVD drive

Prepare:

AX = 150Bh
CX – CD/DVD drive number (0002h = C:, 0003h = D:, and so on)

On return:

BX = ADADh, – a signature, confirming that CD/DVD file system translator's (5.08-03 or 5.08-04) module is loaded yet and is active.
AX = 0000h – this zero value signifies that CD/DVD file system translator (5.08-03 or 5.08-04) doesn't provide control over the requested drive.

8.03-18 INT 2F\AX=150Dh – get drive numbers of CD/DVD drives

Prepare:

AX = 150Dh
ES:BX – pointer to buffer for drive numbers (1 byte per drive)

On return:

ES:BX – pointer to buffer filled with drive numbers (02h = C:, 03h = D:, and so on). List of drive numbers ends with byte 00h.

Note 1: the INT 2F\AX=150Dh function, performed by resident module of CD/DVD file system translator (5.08-03 or 5.08-04), shouldn't be called for, unless INT 2F\AX=150Bh function (8.03-17) confirms in advance, that required module is loaded yet.

8.03-19 INT 2F\AX=150Fh – copy a CD/DVD directory entry

Prepare:

AX = 150Fh
CH = 00h – direct copy "as it is", without translation
= 01h – copy and translate to canonical form (A.09-6)
CL – CD/DVD drive number (02h = C:, 03h = D:, and so on)
ES:BX – pointer to a pathname, ending with 00h byte
SI:DI – pointer to buffer, minimum 255 bytes for direct copy

On return:

On error CF flag is set, AX returns error code (A.06-1).
Clear state of CF flag signifies successful termination, and then
SI:DI – pointer to buffer filled with directory data
AX = 0000h – if disc is of High Sierra format,
= 0001h – if disc is of ISO 9660 format.

Note 1: the INT 2F\AX=150Fh function, performed by resident module of CD/DVD file system translator (5.08-03 or 5.08-04), shouldn't be called for, unless INT 2F\AX=150Bh function (8.03-17) confirms in advance, that required module is loaded yet.

8.03-20 INT 2F\AX=160Ah – Windows OS operating environment test

Prepare:

AX = 160Ah

On return:

AX = 0000h, if Windows OS responds to this test
BH:BL – version of Windows OS
CX – installation type (0002h = standard, 0003h = enhanced)

Note 1: return of a non-zero value in AX register doesn't signify that the caller program is executed not under Windows OS or in its "DOS box". The reason is that among Windows's settings there is a flag named "Prevent DOS programs from detecting Windows". When this flag is set, Windows OS doesn't respond to INT 2F\AX=160Ah test.

8.03-21 INT 2F\AX=1687h – trial request to DPMI server

DPMI servers provide extended API functions to application programs, designed for execution in CPU's V86 mode. Via a request to DPMI server the INT 31 handler can be activated, which enables to install new protected mode interrupt handlers and to send requests for resources to protected mode operating system. DPMI servers for DOS (QDPMI.SYS, CWSDPMI.EXE, etc.) are not popular now, because Windows OS provides its own DPMI server, and it is always available. Being sent from Window's "DOS box", trial request to DPMI server always gets positive response, even when Windows OS doesn't reveal itself legally (note 1 to 8.03-20). Besides that, positive response to trial request is a sufficient evidence of CPU's V86 mode. Because of these reasons the INT 2F\AX=1687h function may be needed, despite that MS-DOS7 doesn't include DPMI server, and DPMI functions usage is not described in this book.

Prepare:

AX = 1687h

On return:

AX = 0000h – this zero value signifies that DPMI server is loaded yet;
BX – set state of bit 0 signifies support for 32-bit programs;
CL – CPU type (02h – 80286, 03h – 80386, 04h – 80486,...);
DH:DL – DPMI server version;
SI – size of DPMI server's data block (in 16-byte paragraphs);
ES:DI – address of enter point for INT 31 handler activation.

Note 1: difference between DOS's DPMI server's environment and Windows "DOS box" is that the latter gives no access to BIOS timer (8.01-73) and to VCPI functions (8.03-71 – 8.03-73).

Note 2: in "Caldera Open DOS" operating system a DPMI server is integrated into EMM386.EXE driver.

8.03-22 INT 2F\AX=4300h – XMS driver's activity test

Prepare:

AX = 4300h

On return:

AL = 80h – this value confirms, that XMS driver HIMEM.SYS (5.04-01) is loaded and is active. Any other returned AL value signifies that XMS functions are unavailable.

8.03-23 INT 2F\AX=4310h – get XMS driver's entrance point

The INT 2F\AX=4310h function returns address of HIMEM.SYS (5.04-01) driver's entrance point. Various XMS functions, listed in table A.12-3, can be invoked by a CALL FAR command (7.03-08), addressed to this entrance point.

Prepare:

AX = 4310h

On return:

ES:BX – address of XMS driver's entrance point (5.04-01).

Note 1: the INT 2F\AX=4310h function, performed by resident module of HIMEM.SYS driver (5.04-01), shouldn't be called for, unless INT 2F\AX=4300h function (8.03-22) confirms in advance, that required module is loaded yet.

Note 2: calls for XMS functions, including INT 2F\AX=4310h and calls via CALL FAR command, require at least 256 bytes of free stack's space.

8.03-24 INT 2F\AX=4B52h – KEYRUS.COM driver's functions (5.02-05)

Prepare:

AX = 4B52h (= 'KR')

BL – subfunction:
= 00h – installation check;
= 4Ch – switch to US keyboard layout;
= 90h – switch to national (russian) keyboard layout

On return:

If KEYRUS.COM driver (5.02-05) is loaded, then
AL = 82h – signature, confirming driver's active state;
BH:BL – KEYRUS.COM driver's version number;
ES value is not preserved.

8.03-25 INT 2F\AX=AD00h – DISPLAY.SYS driver's installation check

Prepare:

AX = AD00h

On return:

AL = FFh – signature, confirming installation of DISPLAY.SYS driver (5.02-02) and availability of its functions.
BX register contents may be altered.

8.03-26 INT 2F\AX=AD01h-AD02h – set/get active codepage

The INT 2F\AX=AD01h-AD02h functions are performed by resident module of DISPLAY.SYS driver (5.02-02). Before sending a call for either of these functions you have to check with INT 2F\AX=AD00h function (8.03-25) whether the required resident module is installed.

Prepare:

AX = AD01h – replace active codepage with another one
= AD02h – get number of active codepage (A.02-2)
BX – hexadecimal number of new codepage (for AX = AD01h only)

On return:

On error CF flag is set, contents of AX and BX are not preserved.
Clear state of CF flag signifies success, and then after AX=AD02h call only
BX – hexadecimal number of current codepage.

8.03-27 INT 2F\AX=AD03h – get codepage information

The INT 2F\AX=AD03h function is performed by resident module of DISPLAY.SYS driver (5.02-02). Before sending a call for this function you have to check with INT 2F\AX=AD00h function (8.03-25) whether the required resident module is installed.

Prepare:

AX = AD03h

ES:DI – pointer to buffer for codepage information
CX – size of buffer in bytes (A.02-6)

On return:

On error CF flag is set. Probable cause: prepared buffer is too small.
Clear state of CF flag signifies successful termination, and then
ES:DI – pointer to buffer filled with data block (A.02-6).

8.03-28 INT 2F\AX=AD80h – KEYB.COM driver's installation check

Prepare:

AX = AD80h

On return:

AL = FFh – signature, confirming installation of KEYB.COM driver (5.02-04) and availability of its functions.;
BH:BL – version number of KEYB.COM driver;
ES:DI – pointer to KEYB.COM driver's data block;
Contents of AH register may be altered.

8.03-29 INT 2F\AX=AD81h – set codepage for keyboard

The INT 2F\AX=AD81h function is performed by resident module of KEYB.COM driver (5.02-04). Before sending a call for this function you have to check with INT 2F\AX=AD80h function (8.03-28) whether the required resident module is installed.

Prepare:

AX = AD81h
BX – hexadecimal number of proposed codepage (A.02-2)

On return:

On error CF flag is set, and then
AX = 0001h value signifies that proposed codepage isn't available.
Clear state of CF flag signifies successful termination.

8.03-30 INT 2F\AX=AD82h-AD83h – set/get keyboard's layout

The INT 2F\AX=AD82h-AD83h functions are performed by resident module of KEYB.COM driver (5.02-04). Before sending a call for either of these functions you have to check with INT 2F\AX=AD80h function (8.03-28) whether the required resident module is installed.

Prepare:

AX = AD82h – set keyboard's layout
= AD83h – get current keyboard's layout
BL – subfunction (for AX = AD82h function only):

= 00h – switch to american layout (just as can be switched by CTRL-ALT-F1 keystroke)
= FFh – switch to national layout (just as can be switched by CTRL-ALT-F2 keystroke)

On return:

On error CF flag is set. Probable cause: invalid BL value.
Clear state of CF flag signifies success, and then after AX=AD83h call only
BL = 00h – american layout;
= FFh – national layout.

8.03-31 INT 33\AX=0000h – mouse driver status and reset

Reset operation brings mouse driver to its default state: movement and wheel counters are reset to zeros, mouse cursor is made invisible and is placed in the center of screen page 0 (function INT 33\AX=0001h should be called for in order to make cursor visible). Besides reset, the INT 33\AX=0000h function enables to find out whether mouse driver is installed and which mouse pointing device is available.

Prepare:

AX = 0000h

On return:

AX = 0000h value signifies that mouse driver isn't loaded (note 2).
= FFFFh value signifies that mouse driver is loaded, and then
CX = 0000h – number of mouse's buttons is other then two;
= 0002h (and FFFFh too) – a 2-button mouse is used;
= 0003h – Mouse Systems/Logitech 3-button mouse is used.

Note 1: if video mode has been changed, then flag of video mode change should be cleared before applying reset to mouse driver (note 2 to INT 33\AX=0028h, 8.03-52).

Note 2: MS-DOS7 fills free cells in interrupt table (up to INT 3F) with references to a IRET command (7.03-30). When mouse driver is not loaded, this IRET command just returns initial AX value intact. If reset operation is not desirable, then status of mouse driver should be reported by INT 33\AX=0021h function (8.03-49).

8.03-32 INT 33\AX=0001h-0002h – show/hide mouse cursor

Prepare:

AX = 0001h – show mouse cursor
= 0002h – hide mouse cursor

Note 1: if a program has called for INT 33\AX=0001h function to show mouse cursor, then this program before its termination must hide mouse cursor, thus restoring original state. Besides that, mouse cursor should be hidden each time before

image on the screen is redrawn, but in the latter case it's better to hide cursor locally with INT 33\AX=0010h function (8.03-42).

Note 2: multiple calls to hide mouse cursor will require multiple calls to unhide it. Exact number of pending bans on showing mouse cursor is reported by INT 33\AX=002Ah function (8.03-53).

8.03-33 INT 33\AX=0003h – button status, cursor and wheel positions

Prepare:

AX = 0003h

On return:

BH – 8-bit signed wheel movement since last call (note 1 to 8.03-33)
BL – mouse's buttons status byte (note 2 to 8.03-33)
CX – cursor's horizontal X-coordinate (note 1 to 8.03-53)
DX – cursor's vertical Y-coordinate (note 1 to 8.03-53)

Note 1: wheel movement is reported, if wheel is present in mouse device and is supported by mouse driver. Both these conditions should be checked by a call for INT 33\AX=0011h function (8.03-43). Positive movement values correspond to downward wheel movement. Wheel movement counter is reset by a call for INT 33\AX=0003h function, and also by requests to this counter, sent via INT 33\AX=0005h or by INT 33\AX=0006h functions (8.03-35).

Note 2: buttons status byte reports states of those buttons, which are currently not released yet. Set states of bits 0 and 1 in buttons status byte correspond to pressed states of left and right mouse's buttons. Set state of bit 2 corresponds to pressed state of middle button, if it is present in mouse device and is supported by mouse driver.

Note 3: in textual videomodes coordinates are reported as multiples of character cell size.

8.03-34 INT 33\AX=0004h – set location of mouse cursor

Prepare:

AX = 0004h
CX – cursor's X-coordinate (0000h – 0280h in videomode 3)
DX – cursor's Y-coordinate (0000h – 00C0h in videomode 3)

Note 1: maximum coordinate values for any current videomode are reported by INT 33\AX=0026h function (8.03-51) and also by INT 33\AX=0031h function (8.03-54).

Note 2: in textual videomodes coordinate values are automatically rounded to the nearest lower multiple of character cell size.

8.03-35 INT 33\AX=0005h–0006h – button's and wheel's state events

Prepare:

AX = 0005h – request about button press or wheel events
= 0006h – request about button release or wheel events
BX = 0000h – request about left button's events
= 0001h – request about right button's events
= 0002h – request about middle button's events
= FFFFh – request about wheel movement events.

On return:

AH – 8-bit signed wheel movement since last call (note 1 to 8.03-33)
AL – buttons status byte (note 2 to 8.03-33)
BX – after requests for button events: number of button's events (presses or releases) since last call for the same function;
– after requests for wheel movement events: 16-bit signed wheel movement since last call (note 1 to 8.03-33)
CX – mouse cursor's horizontal X-coordinate at the moment of last requested event (press or release or wheel rotation);
DX – mouse cursor's vertical Y-coordinate at the moment of last requested event (press or release or wheel rotation).

Note 1: if no one requested event has happened to the requested button since last call for these functions, then zero values in BX, CX and DX registers are returned.

8.03-36 INT 33\AX=0007h-0008h – define mouse cursor's range

Prepare:

AX = 0007h – define cursor's horizontal range
= 0008h – define cursor's vertical range
CX – lower limit of coordinate value (note 2 to 8.03-34)
DX – upper limit of coordinate value (note 2 to 8.03-34)

Note 1: if mouse cursor is beyond desired range, then INT 33\AX=0007h-0008h functions shift cursor's position to the nearest border within permissible range.

8.03-37 INT 33\AX=0009h – mouse cursor in graphic videomodes

Prepare:

AX = 0009h
BX – horizontal shift of cursor's hot spot (from –16 to +16)
CX – vertical shift of cursor's hot spot (from –16 to +16)
ES:DX – pointer to a block of bitmap masks (note 1).

Note 1: block of bitmap masks includes screen mask and cursor mask, each 16 words long. Screen mask starts at offset 00h, cursor mask starts at offset 20h. Each word in a mask defines sixteen pixels along a screen line. Least significant bit in each word corresponds to the rightmost pixel. Screen mask is superimposed over video memory contents with logical AND operation, and then cursor mask is superimposed over the result with logical XOR operation.

Note 2: current cursor's hot spot position is returned by INT 33\AX=002Ah function (8.03-53).

8.03-38 INT 33\AX=000Ah – mouse cursor in textual videomodes

Prepare:

AX = 000Ah
BX = 0000h – software defined cursor (note 1)
= 0001h – hardware defined cursor (note 2)
CX – screen mask (if BX=0000h), or start scan line (if BX=0001h)
DX – cursor mask (if BX=0000h), or last scan line (if BX=0001h)

Note 1: if software definition is selected, the character/color contents of video memory at cursor's position are subjected to logical AND bit-to-bit operation with screen mask and then the result is subjected to logical XOR bit-to-bit operation with cursor mask. Mask's bytes from CH and DH registers are superimposed over color byte (A.10-5) in video memory. For example, if CX=0000h, then cursor acquires form of character, defined by its ASCII code in DL; its color is defined by bytes 0 – 3 in DH according to table A.10-5. Bytes 4 – 6 in DH define background color, the 7-th byte controls blinking. Nonzero values in bits 4 – 6 of CH register make colors dependent on original video memory contents so that cursor becomes more noticeable against any background image.

Note 2: if hardware definition is chosen, then cursor is a blinking bar or a blinking rectangle. For example, values CX=0002h DX=0003h define mouse cursor as a bar above character's row; values CX=0003h DX=0004h define mouse cursor as an underscore.

8.03-39 INT 33\AX=000Bh – read mouse motion counters

Prepare:

AX = 000Bh

On return:

CX – horizontal shift since last call for INT 33\AX=000Bh
DX – vertical shift since last call for INT 33\AX=000Bh

Note 1: mouse cursor's shifts are counted in steps ("mickeys"), which are the smallest position increments the mouse can sense. Microsoft's drivers assign positive shift

values to downward motion and to rightward motion. Correspondence between pixels and mouse steps ("mickeys") can be set by INT 33\AX=000Fh function (8.03-41).

Note 2 the INT 33\AX=0027h function also accepts AX value only, and returns the same shifts in CX and DX, but, besides that, according to software or hardware cursor definitions (8.03-38) returns: in register AX – screen mask or cursor's start scan line, in register BX – cursor mask or cursor's last scan line.

8.03-40 INT 33\AX=000Ch – mouse driver's call for resident subroutine

Prepare:

AX = 000Ch
CX – mask for subroutine call conditions:
bit 0 set: – call if mouse moves
bit 1 set: – call if left button is pressed
bit 2 set: – call if left button is released
bit 3 set: – call if right button is pressed
bit 4 set: – call if right button is released
bit 5 set: – call if middle button is pressed
bit 6 set: – call if middle button is released
bit 7 set: – call if wheel is rotated (note 1 to 8.03-33)
ES:DX – pointer to subroutine entrance for CALL FAR command. (7.03-08)

Note 1: in CX register several conditions can be specified, and then subroutine will be called by mouse driver when either of specified conditions is met. States of bits 15 – 8 in CX register are ignored. States of bits 7 – 5 in CX register are taken into account by those drivers only, which support corresponding mouse's capabilities.

Note 2: resident subroutine is called with following register states:
AX – call conditions (the same bit assignments as in CX mask)
BH – signed wheel movement since last call (note 1 to 8.03-33)
BL – buttons status byte (note 2 to 8.03-33)
CX – cursor's horizontal coordinate (note 2 to 8.03-34)
DX – cursor's vertical coordinate (note 2 to 8.03-34)
SI – horizontal cursor's shift (note 1 to 8.03-39)
DI – vertical cursor's shift (note 1 to 8.03-39).

Note 3: the INT 33\AX=0014h function enables to replace both call mask and address of subroutine's entrance point with new ones, specified similarly in CX and ES:DX registers; on return CX and ES:DX registers contain corresponding replaced former values.

Note 4: if a program has specified mouse call for resident subroutine, then this mouse call must be disabled before program terminates. For this purpose INT 33\AX=000Ch function should be called once more with 0000h mask in CX register.

Note 5: Microsoft's mouse drivers are able to call up to four different resident subroutines. The first subroutine must be specified by INT 33\AX=000Ch function, and the rest three subroutines may be specified by INT 33\AX=0018h function (8.03-45).

8.03-41 INT 33\AX=000Fh – sensitivity of mouse shift registration

Prepare:

AX = 000Fh
CX – number of horizontal steps per 8 pixels (default is 8)
DX – number of vertical steps per 8 pixels (default is 16).

8.03-42 INT 33\AX=0010h – local ban on cursor's display

While mouse's cursor moves on the screen, mouse's driver restores former screen image in each place where mouse's cursor has been moved from. Restoration of former image contents by mouse driver may interfere with screen image updating procedures, performed by foreground program. Such interference can be avoided, if mouse cursor is hidden in the updated part of screen image. Unlike mouse cursor's hiding in the whole screen by INT 33\AX=0002h function (8.03-32), local ban on cursor's display enables to make cursor's blinking almost unnoticeable. When local image updating procedure expires, mouse cursor should be made visible by a call for INT 33\AX=0001h function (8.03-32).

Prepare:

AX = 0010h
CX – horizontal X-coordinate of ban area upper left corner
DX – vertical Y-coordinate of ban area upper left corner
SI – horizontal X-coordinate of ban area lower right corner
DI – vertical Y-coordinate of ban area lower right corner.

8.03-43 INT 33\AX=0011h – pointing device wheel support check

Prepare:

AX = 0011h

On return:

AX = 574Dh – signature confirming driver's support for wheel (note 1)
CX – bit 0 set signifies that mouse pointing device has a wheel.
BX contents may be altered.

Note 1: GMOUSE.COM driver (5.03-01) versions 9.06+ respond to calls for INT 33\AX=0011h function with AX = FFFFh signature. It means that driver doesn't support pointing device wheel, but returns number of mouse's active buttons in BX register.

8.03-44 INT 33\AX=0016h-0017h – save/restore mouse driver's state

Prepare:

AX = 0016h – write driver's state record into prepared buffer
= 0017h – restore mouse driver's state from data in buffer
BX – size of buffer (note 2)
ES:DX – pointer to a buffer (for AX=0017h it must be filled)

On return:

ES:DX – pointer to a buffer with mouse driver state record.

Note 1: restoration of mouse driver's state must be performed in just that videomode, which was active when driver's state record has been written into buffer.

Note 2: size of buffer needed to store driver's state record should be determined beforehand with INT 33\AX=0015h function: required size (in bytes) will be returned in BX register.

8.03-45 INT 33\AX=0018h – mouse driver's call for resident subroutines

Microsoft's mouse drivers are able to call up to four different resident subroutines. The first subroutine must be specified by INT 33\AX=000Ch function (8.03-40), and the rest three subroutines may be specified by INT 33\AX=0018h function.

Prepare:

AX = 0018h
CX – mask for subroutine call conditions:
bit 0 set: – call if mouse moves
bit 1 set: – call if left button is pressed
bit 2 set: – call if left button is released
bit 3 set: – call if right button is pressed
bit 4 set: – call if right button is released
bit 5 set: – call during pressed state of SHIFT key (note 1)
bit 6 set: – call during pressed state of CTRL key (note 1)
bit 7 set: – call during pressed state of ALT key (note 1)
ES:DX – pointer to subroutine entrance for CALL FAR command. (7.03-08)

On return:

AX = 0018h – signature of successful termination;
AX = FFFFh – signature of a failure.

Note 1: subroutines, registered by INT 33\AX=0018h function, are called if either of "functional" keys (SHIFT, CTRL, ALT) is pressed, therefore at least one of bits 7 – 5 in CX mask must be set. States of bits 15 – 8 in CX register are ignored. States of bits 4 – 0 in CX mask are taken into account according to logical OR function: subroutine will be called when either of these conditions is met, if at the same time the specified "functional" key is kept pressed.

Note 2: when mouse driver calls for subroutine, it leaves in registers those values listed in note 2 to article 8.03-40.
Note 3: in order to cease calls for a certain subroutine, the INT 33\AX=0018h function should be called once more with the same mask word in CX register, but with 0000:0000h entrance point address in ES:DX registers.
Note 4: the INT 33\AX=0018h function is supported by Microsoft's mouse drivers since version 6.0, but isn't necessarily supported by mouse drivers from other vendors.

8.03-46 INT 33\AX=0019h – subroutine entrance point address

In response to a call for INT 33\AX=0019h function Microsoft's mouse drivers return entrance point address of that TSR subroutine, which has been registered yet by INT 33\AX=0018h function (8.03-45) with specified mask for call conditions in CX register. Returned data may be used later by any foreground program in order to restore activity of a particular subroutine.

Prepare:
AX = 0019h
CX – mask for call conditions (8.03-45)
On return:
CX = 0000h – error: TSR with submitted mask hasn't been found.
Any other outcome signifies success, and then
BX:DX – entrance point address for the found TSR subroutine;
CX – actual mask for subroutine's call conditions (8.03-45).

8.03-47 INT 33\AX=001Dh-001Eh – screen page number for mouse cursor

Prepare:
AX = 001Dh – define screen page number for mouse cursor
= 001Eh – report current cursor's screen page number
BX – screen page number to be accessed (for INT 33\AX=001Dh only)
On return:
BX – current screen page number (after INT 33\AX=001Eh only)

8.03-48 INT 33\AX=001Fh-0020h – disable/re-enable mouse driver

Prepare:
AX = 001Fh – disable mouse driver
= 0020h – re-enable mouse driver
On return:
AX = FFFFh – signature of a failure.
On success the AX value is returned unchanged.

Note 1: the INT 33\AX=001Fh function restores addresses in interrupt table of INT 10 and INT 74 handlers, which have been there before mouse driver was loaded.

Note 2: if interrupt handler's addresses, set by mouse driver, were removed from interrupt table by INT 33\AX=001Fh function, then INT 33\AX=0020h function is able to write these addresses back into interrupt table.

Note 3: after successful termination the INT 33\AX=001Fh function returns in ES:BX registers that address of INT 33 handler, which has been in interrupt table before mouse driver was installed. If foreground program writes this address back into interrupt table, then possibility to call mouse driver via interrupts will be completely eliminated.

8.03-49 INT 33\AX=0021h – mouse driver installation test

Prepare:

AX = 0021h

On return:

When mouse driver is not installed, then most probably AX = 0021h.
AX= FFFFh value confirms, that mouse driver is installed, and then
BX – number of mouse's buttons (note 2).

Note 1: MS-DOS7 fills free cells in interrupt table (up to INT 3F) with references to a IRET command (7.03-30). When mouse driver is not loaded, this IRET command just returns initial AX value intact. Status of mouse driver is similarly reported by INT 33\AX=0000h function (8.03-31), which also returns driver to initial state. The INT 33\AX=0021h function doesn't return mouse driver to initial state, but nevertheless resets wheel movement counter.

Note 2: having identified a 2-button mouse, some mouse drivers return BX = FFFFh.

8.03-50 INT 33\AX=0024h – mouse type and IRQ setting

Prepare:

AX = 0024h

On return:

AX = FFFFh – signature of mouse device identification failure.
Any other value in AX signifies success, and then
BH.BL – mouse driver version
CH – mouse pointing device type and connection:
= 00h – mouse pointing device isn't connected;
= 01h – mouse is connected via expansion card;
= 02h – mouse is connected to serial port;
= 04h – mouse is connected to PS/2 port;
= 05h – special Hewlett-Packard's mouse.
CL – hexadecimal IRQ number (except CL=00h for PS/2 port).

8.03-51 INT 33\AX=0026h – cursor's maximum coordinate values

Prepare:

AX = 0026h

On return:

Non-zero value in BX register signifies an error.

BX = 0000h – signature of successful termination, and then

CX – maximum horizontal X-coordinate for current videomode

DX – maximum vertical Y-coordinate for current videomode.

Note 1: the INT 33\AX=0026h function reports cursor coordinate values for the whole screen. If cursor's area is confined within certain ranges (8.03-36), then allowable minimum and maximum coordinate values should be determined with INT 33\AX=0031 function (8.03-54).

Note 2: the INT 33\AX=0026h function shouldn't be called for, unless mouse driver's support for this function is confirmed by INT 33\AX=0032h (8.03-55).

8.03-52 INT 33\AX=0028h – consistent change of videomode

As far as mouse cursor is drawn by means of direct access to video memory, mouse driver's interventions must conform to current format of data in video memory. However, different videomodes define different data formats. Hence each change of videomode must be coordinated with change of video data format, used by mouse driver. The INT 33\AX=0028h function provides an opportunity of consistent videomode change for both video memory and mouse driver, so that appearance and movement of mouse cursor wouldn't be disturbed.

Prepare:

AX = 0028h

CX – code of proposed video mode (A.10-1)

DH – vertical size of character cell (note 1)

DL – horizontal size of character cell (note 1)

On return:

Non-zero value in CL register signifies a failure.

CL = 00h – signature of successful termination.

Note 1: zero values DH = DL = 00h specify default character cell size for any videomode. If proposed videomode doesn't support character cell size control, then any values in DH and DL registers are ignored.

Note 2: if CX = 0000h on call, then videomode is not set, but an internal videomode change flag is cleared. This flag must be cleared before each call for mouse driver's reset function (8.03-31).

Note 3: the INT 33\AX=0028h function shouldn't be called for, unless mouse driver's support for this function is confirmed by INT 33\AX=0032h (8.03-55).

Note 4: a list of videomode codes, supported by mouse driver, can be returned by several sequential calls for INT 33\AX=0029h function. The first call accepts CX = 0000h and returns in CX a code of first supported videomode. The next call, performed with this code in CX intact, returns in CX code of next supported videomode, and so on. Some mouse drivers return in DS:DX a pointer to a string with description of videomode; some other mouse drivers may leave zero value in DS:DX. End of calls cycle is marked by return of zero value CX = 0000h.

8.03-53 INT 33\AX=002Ah – mouse cursor's parameters

Prepare:

AX = 002Ah

On return:

AX – number of pending bans on showing mouse cursor (8.03-32)
BX – horizontal X-shift of cursor's hot spot (note 1)
CX – vertical Y-shift of cursor's hot spot (note 1)
DX – mouse type, as CH returned by INT 33\AX=0024h (8.03-50).

Note 1: cursor's coordinates are counted relative to upper left corner of cursor's block, but cursor points at its hot spot. Shift of hot spot from the upper left corner of cursor's block may range from –128 to +127 for both coordinates.
Note 2: the INT 33\AX=002Ah function shouldn't be called for, unless mouse driver's support for this function is confirmed by INT 33\AX=0032h (8.03-55).

8.03-54 INT 33\AX=0031h – get current coordinate range

The INT 33\AX=0031h function reports current values of range limits, when available mouse cursor's area is confined within a virtual window set by INT 33\AX=0007h-0008h (8.03-36).

Prepare:

AX = 0031h

On return:

AX – minimum value of horizontal X-coordinate
BX – minimum value of vertical Y-coordinate
CX – maximum value of horizontal X-coordinate
DX – maximum value of vertical Y-coordinate

Note 1: when available cursor's area is not confined within a window, then coordinate ranges should be reported by INT 33\AX=0026h function (8.03-51).
Note 2: the INT 33\AX=0031h function shouldn't be called for, unless mouse driver's support for this function is confirmed by INT 33\AX=0032h (8.03-55).

8.03-55 INT 33\AX=0032h – supported functions of mouse driver

Prepare:

AX = 0032h

On return:

AX – a word of flags, where each bit signifies support for mouse driver's functions from INT 33\AX=0034h to INT 33\AX=0025h:
bit 3 set: – INT 33\AX=0031h is supported;
bit 10 set: – INT 33\AX=002Ah is supported;
bit 11 set: – INT 33\AX=0029h is supported;
bit 12 set: – INT 33\AX=0028h is supported;
bit 14 set: – INT 33\AX=0026h is supported.

Contents of BX, CX, DX registers are not preserved.

8.03-56 INT 4A – alarm handler hook

Default INT 4A handler just returns control back to the caller program. Role of the caller program belongs to BIOS's real-time clock alarm, if it is set to a certain time by INT 1A\AH=06h function (8.01-94). Application programs are suggested to load an intercepting INT 4A handler, which will be called at a predetermined time in order to perform a desired action.

Note 1: `the INT 4A interrupt is called from within a hardware interrupt handler. Therefore all necessary precautions against reentering DOS must be taken (8.02-70, 8.02-87).

8.03-57 INT 67\AH=41h – get page frame segment

By default the EMM386.EXE driver (5.04-02) arranges in upper memory (below 1024 kb) 4 "physical" pages, grouped into one 64-kb frame. Access to extended memory is implemented by dynamic mapping of extended memory "logical" pages (beyond 1088 kb) onto these 16-kb "physical" pages inside a page frame. Most often page frame starts at segment address E000h.

Prepare:

AH = 41h

On return:

AH – error code (A.06-1); if AH = 00h, then
BX – segment address of page frame (4 "physical" pages).

Note 1: the INT 67\AH=41h function shouldn't be called for, unless EMM386.EXE (5.04-02) driver's active state is confirmed by INT 67\AH=46h (note 1 to 8.03-62).

8.03-58 INT 67\AH=42h – get number of EMS pages

According to EMS specification the EMM386.EXE driver provides access to selected "logical" 16-kb pages arranged in extended memory from 1088 to 32768 kb, except area reserved by /L parameter (5.04-02) for access arranged by XMS-driver (5.04-01). The INT 67\AH=42h function reports statistics of "logical" EMS pages in extended memory, available for EMM386.EXE driver.

Prepare:

AH = 42h

On return:

AH – error code (A.06-1); if AH = 00h, then
BX – number of unallocated (free) "logical" EMS-pages
DX – total number of "logical" EMS-pages.

Note 1: the INT 67\AH=42h function shouldn't be called for, unless EMM386.EXE (5.04-02) driver's active state is confirmed by INT 67\AH=46h (note 1 to 8.03-62).

8.03-59 INT 67\AH=43h – allot a handle and extended memory

Unlike INT 21\AH=3Dh function (8.02-33), the EMM386.EXE driver (5.04-02) allots only those handles, which refer to areas of extended memory beyond 1088 kb boundary. Each area may comprise an integer number of "logical" 16-kb pages. Specification of any particular "logical" page includes its number in allocated area and the handle, assigned to that allocated area.

Prepare:

AH = 43h
BX – requested non-zero number of "logical" pages, 16 kb each.

On return:

AH – error code (A.06-1); if AH = 00h, then
DX – handle for area, comprising requested number of pages.

Note 1: the INT 67\AH=43h function shouldn't be called for, unless EMM386.EXE (5.04-02) driver's active state is confirmed by INT 67\AH=46h (note 1 to 8.03-62).

Note 2: by default the EMM386.EXE driver may keep active up to 64 handles simultaneously, but the "h" parameter (5.04-02) enables to increase number of active handles to 255.

Note 3: version 4.0 of LIM EMS specification stipulates INT 67\AX=5A00h function with the same mission. The only difference is permission to request zero number of "logical" pages in BX register.

8.03-60 INT 67\AH=44h – map "logical" page to a "physical" page

Here the term "mapping" implies, that appeals addressed to a particular "physical" 16-kb page below 1024 kb will be automatically diverted by CPU to specified "logical" 16-kb page, physically present in extended memory beyond 1088 kb boundary.

Prepare:

AH = 44h
AL – number of a selected "physical" 16-kb page;
BX – number of a requested "logical" 16-kb page, or else
= FFFFh – in order to make specified "physical" page free;
DX – handle of memory area, comprising requested "logical" page.

On return:

AH – error code (A.06-1); AH = 00h signifies successful termination.

Note 1: the INT 67\AH=44h function shouldn't be called for, unless EMM386.EXE (5.04-02) driver's active state is confirmed by INT 67\AH=46h (note 1 to 8.03-62).

Note 2: count of memory pages, both "physical" and "logical", starts from zero number.

Note 3: "physical" pages 00h – 03h constitute page frame. Address of page frame is reported by INT 67\AH=41h function (8.03-57). Location of "physical" pages 04h – 1Bh, if these exist, can be reported by INT 67\AX=5800h function (8.03-70). If a "physical" page is located in conventional memory (below 640 kb), then memory space, occupied by this memory page, must be allotted by operating system to that program, which intends to use this "physical" page.

8.03-61 INT 67\AH=45h – release a handle and memory area

When a part of extended memory, associated with a handle, is no more needed, then handle owner program must inform EMM386.EXE driver (5.04-02) about that. After a call for INT 67\AH=45h function this part of extended memory will be considered free, and associated handle will become disabled.

Prepare:

AH = 45h
DX – handle of that memory area, which is to be released

On return:

AH – error code (A.06-1); AH = 00h signifies successful termination.

Note 1: the INT 67\AH=45h function should be used to disable those handles only, which have been assigned by EMM386.EXE driver (5.04-02) after calls for INT 67\AX=5A00h or for INT 67\AH=43h functions (8.03-59). Other handles should be disabled by INT 21\AH=68h, INT 21\AH=6Ah or INT 21\AH=3Eh functions (8.02-34).

Note 2: repeated request for a handle can't restore access to those pages of extended memory, which were associated with a disabled handle.

8.03-62 INT 67\AH=46h – get version of EMM386.EXE driver

Prepare:

AH = 46h

On return:

AH – error code (A.06-1); if AH = 00h, then
AL – version number of EMM386.EXE driver (5.04-02).

Note 1: address of INT 67 handler, stored in cell 0000:019Eh of interrupt table, points at start of handler's header. At offset 0Ah relative to start of handler's header a signature EMMXXXX0 is written by EMM386.EXE driver. If this signature can't be found, then neither function of INT 67 handler can be called, because INT 67 cell in interrupt table is not filled by default and may contain an invalid pointer (for example, 0000:0000h). Presence of EMMXXXX0 signature in its proper place confirms that EMM386.EXE driver is loaded yet, and then INT 67\AH=46h function should be called for. The AH = 00h value, returned by INT 67\AH=46h function, signifies active state of EMM386.EXE driver and its readiness to perform other INT 67 functions.

Note 2: when it is known for certain, that EMM386.EXE driver is loaded, then any non-zero error code, returned by INT 67\AH=46h function, signifies inactive state of EMM386.EXE driver. In this case the INT 67\AX=FFA5h function (8.03-74) may help: it returns address of EMM386.EXE driver's API entrance point. With a CALL FAR command (7.03-08), addressed to this entrance point, the EMM386.EXE driver may be turned into active state.

8.03-63 INT 67\AH=4Bh – get number of EMM handles

Prepare:

AH = 4Bh

On return:

AH – error code (A.06-1); if AH = 00h, then
BX – number of active handles, assigned by EMM386.EXE driver.

Note 1: the INT 67\AH=4Bh function shouldn't be called for, unless EMM386.EXE (5.04-02) driver's active state is confirmed by INT 67\AH=46h (note 1 to 8.03-62).

Note 2: by default the EMM386.EXE driver may keep active up to 64 handles simultaneously, but the "h" parameter (5.04-02) enables to increase number of active handles to 255.

8.03-64 INT 67\AH=4Ch – get pages associated with a handle

Prepare:

AH = 4Ch
DX – an active handle, assigned to some extended memory area

On return:

AH – error code (A.06-1); if AH = 00h, then
BX – number of "logical" pages, associated with specified handle.

Note 1: the INT 67\AH=4Ch function accepts those active handles only, which have been allotted by INT 67\AH=43h function (8.03-59) of EMM386.EXE driver (5.04-02). The INT 67\AH=4Ch function shouldn't be called for, unless EMM386.EXE driver's active state is confirmed by INT 67\AH=46h (note 1 to 8.03-62).

8.03-65 INT 67\AH=4Eh – save/restore extended memory mapping

If a resident module intends to use EMS access to extended memory, then it should be taken into account, that any foreground program, interrupted by invoked resident module, also could use EMS access to extended memory. Execution of this foreground program can't be resumed, unless interrupting resident module, having finished its mission, restores original mapping of extended memory. For this purpose INT 67\AH=4Eh presents 4 subfunctions, enabling to save current extended memory mapping and later to restore it.

Prepare:

AH = 4Eh
AL – subfunction:
= 00h – save current mapping into a data block;
= 01h – restore mapping from data block;
= 02h – consecutive execution of subfunctions 00h and 01h;
= 03h – determine required buffer size for data block.
ES:DI – pointer to empty buffer (for subfunctions 00h and 02h)
DS:SI – pointer to data block (for subfunctions 01h and 02h)

On return:

AH – error code (A.06-1); if AH = 00h, then
AL – required buffer size in bytes (after subfunction 03h only).

Note 1: the INT 67\AH=4Eh function is executed by EMM386.EXE driver's versions 4.00 and higher (5.04-02). Therefore INT 67\AH=4Eh function shouldn't be called for, unless EMM386.EXE driver's proper version and active state are confirmed by INT 67\AH=46h (note 1 to 8.03-62).

Note 2: starting from EMM386.EXE driver's version 3.00, saving and restoration of extended memory mapping state may be performed by functions INT 67\AH=47h and INT 67\AH=48h correspondingly. These functions restore state of 64-kb

page frame only, don't need explicit data block(s), but require in DX register a handle, allotted by EMM386.EXE driver to the caller module.

8.03-66 INT 67\AX=5000h – change handle's mapping list

Mapping list is a table of correspondence between "physical" and "logical" memory pages, associated with one handle. The INT 67\AX=5000h function enables to replace current mapping list with a new one. Single call for INT 67\AX=5000h function is equivalent to several consecutive calls for INT 67\AH=44h mapping function (8.03-60).

Prepare:

AX = 5000h
CX – number of entries in mapping list, 4 bytes per entry (note 3)
DX – an active handle, assigned to some extended memory area
DS:SI – pointer to start of proposed (new) mapping list

On return:

AH – error code (A.06-1); AH = 00h signifies successful termination.

Note 1: function INT 67\AX=5000h operates with those handles only, which are allotted by EMM386.EXE driver's functions INT 67\AH=43h or INT 67\AX=5A00h (8.03-59). The INT 67\AX=5000h function shouldn't be called for, unless EMM386.EXE (5.04-02) driver's active state is confirmed by INT 67\AH=46h (note 1 to 8.03-62).

Note 2: if a "physical" page is located in conventional memory (below 640 kb), then memory space, occupied by this memory page, must be allocated by operating system to that program, which intends to use this "physical" page.

Note 3: each entry in mapping list is 4 bytes long and consists of two words. The second word in each entry is a "physical" page number (most often 0000h – 0003h). For mapping operation the first word in an entry must be "logical" page number. For opposite unmapping operation the first word in an entry must be FFFFh value: it forces to cancel current association of corresponding "physical" page.

Note 4: the INT 67\AX=5001h function is charged with the same mission and accepts the same specifications (except AX), but operates with other data in mapping list: the second word in each entry must be segment address of corresponding "physical" page.

8.03-67 INT 67\AH=51h – reallocate "logical" pages

Prepare:

AH = 51h
BX – requested number of "logical" pages for the handle
DX – an active handle, assigned to some extended memory area

On return:

AH – error code (A.06-1); if AH = 00h, then
BX – actual number of pages associated with specified handle.

Note 1: the INT 67\AX=51h function operates with those handles only, which are allotted by EMM386.EXE driver's functions INT 67\AH=43h or INT 67\AX=5A00h (8.03-59). The INT 67\AX=51h function shouldn't be called for, unless EMM386.EXE (5.04-02) driver's active state is confirmed by INT 67\AH=46h (note 1 to 8.03-62).

Note 2: if INT 67\AH=51h function is called in order to increase number of associated "logical" pages, then ordinal numbers of new pages will follow ordinal numbers of currently available "logical" pages. If INT 67\AH=51h function is called in order to decrease number of associated "logical" pages, then pages with largest ordinal numbers will be lost first.

8.03-68 INT 67\AH=55h–56h – jump and subroutine call in EMS-memory

When a code is executed in extended memory, then target "logical" page for control transfer operations JMP FAR and CALL FAR should be made accessible in advance. For this purpose the EMM386.EXE driver provides two functions, combining control transfer with replacement of mapping list for specified handle. The INT 67\AH=55h function replaces mapping list and performs JMP FAR operation (7.03-39), the INT 67\AH=56h function replaces mapping list and performs CALL FAR operation (7.03-08). Both these functions accept most part of required parameters from a prepared data block. Structure of this data block is shown in table A.12-6.

Prepare:

AX = 5500h – for JMP FAR operation
= 5600h – for CALL FAR operation
DX – an active handle, assigned to some extended memory area
DS:SI – pointer to start of data block (A.12-6).

On return:

AH – error code (A.06-1); AH = 00h signifies successful termination.

Note 1: the INT 67\AH=55h-56h functions should be used by those programs only, which are designed for execution in extended memory.

Note 2: the INT 67\AX=5501h and INT 67\AX=5601h functions are charged with the same missions and accept the same specifications (except AX), but operate with other data in mapping lists: the second word in each entry must be segment address of corresponding "physical" page.

Note 3: the INT 67\AX=5602h function doesn't need other initial data, except AX value, and returns in BX register number of bytes for return addresses, which are saved in stack by INT 67\AX=5600-5601h functions.

8.03-69 INT 67\AX=5700h-5701h – data copying or exchange

The INT 67\AX=5700h-5701h functions enable to copy data or exchange data between memory areas, either accessed via different handles or belonging to conventional memory. Addressing parameters are specified in a descriptor, shown in table A.12-5.

Prepare:

AX = 5700h – to copy data from one memory area to another
= 5701h – to exchange data between memory areas
DS:SI – pointer to parameters descriptor (A.12-5)

On return:

AH – error code (A.06-1); AH = 00h signifies successful termination.

Note 1: functions INT 67\AX=5700h-5701h operate with those handles only, which are allotted by EMM386.EXE driver's functions INT 67\AH=43h or INT 67\AX=5A00h (8.03-59). The INT 67\AX=5700h-5701h functions shouldn't be called for, unless EMM386.EXE (5.04-02) driver's active state is confirmed by a call for INT 67\AH=46h (note 1 to 8.03-62).

8.03-70 INT 67\AX=5800h – segment addresses of physical pages

Prepare:

AX = 5800h
ES:DI – pointer to a buffer to be filled

On return:

AH – error code (A.06-1); if AH = 00h, then
CX – number of entries in the buffer (4 bytes each entry);
ES:DI – pointer to buffer filled with entries (note 2)

Note 1: the INT 67\AH=5800h function shouldn't be called for, unless EMM386.EXE (5.04-02) driver's active state is confirmed by a call for INT 67\AH=46h (note 1 to 8.03-62).

Note 2: each entry in the buffer is 4 byte long and consists of 2 words: the first is physical page segment, the second is corresponding physical page number.

Note 3: number of physical pages (and, hence, length of the buffer) may be obtained in advance with INT 67\AX=5801h function, which similarly returns number of entries in CX register, but doesn't fill the buffer and ignores contents of ES:DI registers.

8.03-71 INT 67\AX=DE06h – physical address of a 4-kb page

The INT 67\AX=DE06h function is performed by VCPI servers. It helps to perceive the concept of UMB address space transformation, performed by CPU switched into V86 mode. As far as in MS-DOS7 the EMM386.EXE driver (5.04-02) is charged with mission

of VCPI server, the INT 67\AH=DE06h function shouldn't be called for, when EMM386.EXE driver's active state is not confirmed yet by a call for INT 67\AH=46h (note 1 to 8.03-62) and also when VCPI functions are disabled by /noVCPI parameter (note 2 to 5.04-02).

Prepare:

AX = DE06h
CX – number of 4-kb a page below 1024 kb boundary (note 1)

On return:

AH – error code (A.06-1); AH = 8Bh signifies invalid page number.
AH = 00h value signifies success, and then
EDX – physical address of the requested page (note 2).

Note 1: unlike LIM EMS functions, VCPI functions operate with 4-kb pages, processed by TLB address translation mechanism in 32-bit CPUs. Number of a 4-kb page is obtained by shifting its linear address 12 bit rightward. For example, memory cell D400:1ABCh has linear address D5ABCh, so its page number is CX = 00D5h.

Note 2: for DOS programs a way of access to EDX and to other 32-bit registers is shown in article 7.02-06.

8.03-72 INT 67\AX=DE07h – read state of control register CR0

Unlike operation of reading CR0 register's state by MOV command (note 1 to 7.03-58), which requires CPU's real mode or the highest privilege level, the INT 67\AX=DE07h function of VCPI servers is available in CPU's V86 mode at the third (the lowest) privilege level. As far as in MS-DOS7 the EMM386.EXE driver (5.04-02) is charged with mission of VCPI server, the INT 67\AH=DE07h function shouldn't be called for, when EMM386.EXE driver's active state is not confirmed yet by a call for INT 67\AH=46h (note 1 to 8.03-62) and also when VCPI functions are disabled by /noVCPI parameter (note 2 to 5.04-02).

Prepare:

AX = DE07h

On return:

AH – error code (A.06-1); if AH = 00h, then
EBX – current state of control register CR0 (note 1).

Note 1: for DOS programs a way of access to EBX and to other 32-bit registers is shown in article 7.02-06.

8.03-73 INT 67\AX=DE08h–DE09h – access to registers DR0 – DR7

Unlike access to CPU's registers DR0 – DR7 with MOV commands (note 1 to 7.03-58), which require CPU's real mode or the highest privilege level, the

INT 67\AX=DE08h-DE09h functions of VCPI servers are available in CPU's V86 mode at the third (the lowest) privilege level. As far as in MS-DOS7 the EMM386.EXE driver (5.04-02) is charged with mission of VCPI server, the INT 67\AH=DE08h-DE09h functions shouldn't be called for, when EMM386.EXE driver's active state is not confirmed yet by a call for INT 67\AH=46h (note 1 to 8.03-62) and also when VCPI functions are disabled by /noVCPI parameter (note 2 to 5.04-02).

Prepare:

AX = DE08h – reading from DR0 – DR7 registers into a buffer
= DE09h – writing from buffer into DR0 – DR7 registers
ES:DI – pointer to buffer with data or for data (note 1)

On return:

AH – error code (A.06-1); AH = 00h signifies successful termination.

Note 1: buffer is 32 bytes long and is filled with 4 data bytes per register from DR0 to DR7. Data, corresponding to DR4 and DR5 registers, are not read and are ignored by writing operation. Role of DR registers is described in appendix A.11-5.

8.03-74 INT 67\AX=FFA5h – EMM386.EXE driver's API entrance point

The INT 67\AX=FFA5h function, stipulated by LIM EMS specification since version 4.2, differs from other INT 67 functions in that it is executed even when EMM386.EXE driver is inactive and ignores all other requests. However, this feature doesn't exclude necessity to check whether the EMM386.EXE driver is loaded (note 1 to 8.03-62) before INT 67\AX=FFA5h function is called for.

Prepare:

AX = FFA5h

On return:

AH = 84h – signature of successful termination, and then
BX:CX – address of EMM386.EXE API entrance point.

Note 1: a CALL FAR command (7.03-08) addressed to BX:CX entrance point forces EMM386.EXE to perform operation, defined by value in AX register:
if AX = 0100h – to switch itself ON to active state;
if AX = 0101h – to switch itself OFF to inactive state;
if AX = 0500h – to display a message about current state.
When UMB blocks and EMS pages are used yet, then a request for switching OFF to inactive state wouldn't be executed.

8.03-75 INT 70 – INT 77: interrupt requests IRQ 8 – IRQ 15

While CPU is in real mode, the INT 70 – INT 77 group of interrupt handlers responds to requests, sent via IRQ 8 – IRQ 15 lines from various devices to the second interrupt controller, which, in its turn, sends its output to CPU via IRQ 2 line of the first interrupt controller (INT 0A, 8.01-09). Each of IRQ 8 – IRQ 15 input lines may be disabled (masked) by sending a bit, specified in the third column of the following table, to second interrupt controller via port A1h. Some IRQ lines have dedicated hardware sources, listed in the fourth column of the following table, but some other IRQ lines are free to receive a request from any device, which is tuned to send requests via one of these lines and is supported by a driver, loading a handler for the corresponding interrupt.

Interrupt	Line	Mask	Source of requests	Comments
INT 70	IRQ 8	bit 0	Real-time clock	note *1
INT 71	IRQ 9	bit 1	-	
INT 72	IRQ 10	bit 2	-	
INT 73	IRQ 11	bit 3	-	
INT 74	IRQ 12	bit 4	-	note *2
INT 75	IRQ 13	bit 5	Arithmetical coprocessor	
INT 76	IRQ 14	bit 6	1-st IDE controller	
INT 77	IRQ 15	bit 7	-	note *3

Note 1: the INT 70 handler is called 1024 times per second. There are BIOS systems, which call for INT 70 handler in event waiting intervals only (INT 15\AH=83h, 8.01-73).

Note 2: preferable source for IRQ 12 line is PS2 mouse, if it is used.

Note 3: preferable sources for IRQ 15 line are either second IDE controller or SCSI bus controller, if these are present in a particular PC.

Chapter 9 Examples of executable files' composition

All examples of interpretable, executable and configuration files, presented here in the 9-th chapter, have been successfully tested on several different computers with processors from 486SL (dated 1992) to Pentium-D (dated 2006). However, one can't foresee everything in advance. In any case the presented examples should be regarded as advisable schemes, where anyone is allowed to select personally relevant solutions. Their applicability is left for you to decide and to implement.

Though personal problems may various, some elementary operational habits are common for all who dares to descend deeper into low-level programming. First of all – how a program's text, printed in a book, can be transferred into an executable file. For this purpose you have to launch an editor program, capable to save non-formatted textual files. Most widely known WORD and WORDPAD programs wouldn't fit, but NOTEPAD and EDIT.COM are quite suitable. However, in MS-DOS any national (non-american) language commentaries can be inserted by EDIT.COM editor only (6.09), and only if national adaptation drivers are loaded beforehand (examples – in articles 9.01 and 9.04).

When editor program is launched yet, then item NEW in menu FILE enables to open a new empty window, where you may write any desired text. Texts of executable files, presented in this book, normally should be typed line-by-line starting close to the left border of each line. Then text should be saved in a file by SAVE AS command in menu FILE. The SAVE AS command suggests to specify a name and a suffix for new file. Each file should be given just that suffix (2.01-02), which reflects file's status: BAT – for batch-files, SCR – for command files, executed by DEBUG.EXE, SYS, MNU or EXT suffixes – for various configuration files.

Long files may be saved several times: first time by SAVE AS command, and later by SAVE command. If text is to be corrected on-the-fly, then parts of an unfinished file may be tested separately, as it is shown in article 9.07-02. Editor program fulfills its mission, when a textual file with specified name is completely typed, checked and saved.

If not specified otherwise, successful execution or interpretation of all program examples, presented in this chapter, implies that the following normal conditions are met:

– name of current command interpreter (COMMAND.COM or some other) with preceding full path must be defined by a value of environmental variable %COMSPEC%;

– attributes H (hidden) and S (system) shouldn't be assigned to those files or utilities, which are to be called for or referred to by the program under test (note 1 to 9.11-02);

– paths to all those files or utilities, which are to be called for or referred to by the program under test, must be specified in a value of environmental variable PATH (2.02-02);
– synonymous utilities, unsuitable under MS-DOS7, must not be present in the current directory and along all the paths, specified in a value of environmental variable PATH (2.02-02);
– value of the TEMP environmental variable must specify a path to an existing directory, dedicated for temporary files, on a writable media, having enough free space for this purpose.

Note, that status of the %TEMP% directory implies that any file in this directory may be deleted or overwritten. Current values of environmental variables may be checked with SET command (3.26). Original values of environmental variables as well as other execution conditions should be defined in configuration files. Various examples of such definitions are shown in articles 9.04, 9.09. But it's expedient to start from the simplest configuration files, presented in the next article 9.01.

9.01 Simple configuration files

MS-DOS7 loading process is configurable and may lead to different results. The IO.SYS loader accepts loading parameters from MSDOS.SYS file (5.01-01) and takes into account presence of main DOS files in the root directory of boot disk. Complete list of DOS files, which should be present in root directory, is described in article 9.11-02. Among these files there are two optional, but very important files: CONFIG.SYS and AUTOEXEC.BAT. Just these two files define main features of final DOS configuration. Though MS-DOS7 is able to load itself without these two files, but implicit default configuration is too poor and can't be considered sufficient.

MS-DOS7 can be made effective, convenient and friendly, but the user has to bother about that. In modern computers even simple configurations must include drivers for extended memory, for "mouse" pointing device and for CD-ROM drive. DOS can't be convenient without adequate file manager. Examples of configuration files, presented in this book, also load codepages, enabling to use both american and some national (non-american) language.

Simple versions of configuration files may be used for loading MS-DOS7 from removable media, but are most suitable for MS-DOS7 installation on HDDs: either for temporarily installation after formatting or for permanent installation as a possible choice among several alternative operating systems (example – in 9.11-02).

9.01-01 Simple version of CONFIG.SYS file

CONFIG.SYS file is an interpretable command file: each its line is a command. Term "interpretable" implies that commands are executed not directly by CPU, but via a mediator – a command interpreter program. Mission of CONFIG.SYS commands interpretation is incumbent on IO.SYS loader.

The presented example of CONFIG.SYS file shows explicitly all the drivers, which should be loaded, and preferable order of their loading. Drivers are expected to be found in \DOS\DRV directory. If you intend to use other directory structure, path specifications must be corrected accordingly. Note, that the paths are shown without disk's letter-name. Such specifications will suit for loading from any bootable disk.

```
device=\DOS\DRV\Himem.sys
device=\DOS\DRV\Emm386.exe ram v
dos=high,umb,noauto
buffershigh=20,0
fileshigh=30
lastdrivehigh=Z
fcbshigh=1,0
stackshigh=9,256
numlock off
country=007,866,\DOS\DRV\Country.sys
devicehigh=\DOS\DRV\Dblbuff.sys
devicehigh=\DOS\DRV\Ifshlp.sys
devicehigh=\DOS\DRV\Setver.exe
devicehigh=\DOS\DRV\Display.sys con=(ega,,1)
devicehigh=\DOS\DRV\Oakcdrom.sys /D:CD001
installhigh=\DOS\DRV\Mscdex.exe /D:CD001 /E /L:O /M:13
installhigh=\DOS\DRV\Mouse.com
shell=\Command.com \ /E:2016 /L:511 /U:255 /p
```

A help concerning all specified commands and drivers can be found in chapters 4 ("Configuration commands") and 5 ("Selected drivers"). Most drivers can be taken from \WINDOWS and \WINDOWS\COMMAND directories of WINDOWS-95/98 operating system, except MOUSE.COM (5.03-02) from MS-DOS6.22 release and OAKCDROM.SYS (5.09-01), which is copied from WINDOWS-95/98 rescue diskette. There is a lot of other mouse and CD-ROM drivers (GMOUSE.COM, VIDE-CDD.SYS, ECSCDIDE.SYS, etc.), which can work with a variety of device models and can be used here instead of MOUSE.COM and OAKCDROM.SYS.

The order of lines in CONFIG.SYS file must conform to the following general rule: the drivers providing some support must be loaded before a need arises for this support. Upper memory drivers (HIMEM.SYS and then EMM386.EXE) must be loaded with DEVICE commands before upper memory access will be required for DEVICEHIGH and

INSTALLHIGH commands. INSTALL and INSTALLHIGH commands, used to load executable drivers, must be placed after all DEVICE and DEVICEHIGH commands, but before the SHELL command. SETVER.EXE driver must be loaded before any other driver, which needs to be deceived about DOS version number. Though there are no such drivers in the shown example of CONFIG.SYS file, loading of SETVER.EXE gives an opportunity to execute later version-specific utilities from earlier versions of DOS (PRINT.EXE, QBASIC.EXE, TREE.COM, etc.).

Special attention should be paid to the NOAUTO parameter of the DOS command in the 3-rd line: it cancels default loading of some drivers (DBLBUFF.SYS, DRVSPACE.BIN, HIMEM.SYS, IFSHLP.SYS) as well as a search for these drivers along default paths. In fact the NOAUTO parameter enables to use MS-DOS7 as a stand-alone operating system.

Note, that a path to COMMAND.COM file in the last line of CONFIG.SYS file is reduced to a single backslash " \ ". This is enough for finding COMMAND.COM file in the root directory, but is not enough for proper definition of path to COMMAND.COM file in the value of COMSPEC environmental variable. Of course, a particular disk's letter-name may be specified inside CONFIG.SYS file. Such examples are shown in articles 4.26 and 6.04. However, in practice it's not convenient to exchange disk's letter-name in several places each time you have to boot your PC from some other disk. Therefore here definition of a particular disk's letter-name is postponed until execution of the last configuration file AUTOEXEC.BAT (9.01-02). Postponed disk's letter-name assignment enables to correct letter-name in a single place and provides opportunity for automatic disk's letter-name determination (examples – in 9.01-03 and 9.09-02).

9.01-02 Simple version of AUTOEXEC.BAT file

As far as functional capabilities of IO.SYS loader are limited, some configuration operations can't be expressed as commands in CONFIG.SYS file. Such operations constitute another configuration file – AUTOEXEC.BAT. It is also an interpretable command file, but it's mission is to employ capabilities of a more powerful command interpreter – COMMAND.COM – for performing a number of final configuration operations.

Presented version of AUTOEXEC.BAT file implies presence of directory structure on the bootable disk: directory \TEMP for temporary files and directory \DOS with subdirectories \DOS\OTH, \DOS\MS7, \DOS\VC4, \DOS\DRV. This structure can suit for both diskettes and fixed disks, but if you intend to use any other structure, all paths and references must be changed accordingly. The file is devised for booting from disk C:. If it is to be used for loading from any other disk, reference to disk C: in the second line must be replaced with reference to that disk, which actually will be used. Note, that it is a single

reference, which must be corrected; all other references to actual disk will be exchanged automatically.

Here is the proposed simple version of AUTOEXEC.BAT:

```
@echo off
set dsk=C:
set comspec=%dsk%\Command.com
if not exist TEMP\nul %comspec% nul /f /c md TEMP
if exist TEMP\nul set Temp=%dsk%\TEMP
prompt $p$g
set dircmd= /A /O:GNE /P
path ;
path=%dsk%\DOS\VC4;%dsk%\DOS\OTH;%dsk%\DOS\MS7;%dsk%\
Mode.com con codepage prepare=((866) %dsk%\DOS\DRV\Ega3.cpi)
Mode.com con codepage select=866
Keyb.com ru,866,%dsk%\DOS\DRV\Keybrd3.sys
set VC=%dsk%\DOS\VC4
Vc.com /TSR /no2E /noswap
```

Command in the first line of the shown file turns echo flag off, just as it is done in ordinary batch files. The second line specifies letter-name of the current disk and assigns it as a value to environmental variable DSK. Note, that there must be no spare spaces at the end of second line. Multiple references to DSK variable in the following lines insert letter-name of the current disk into all relevant paths. This enables to specify disk's letter-name only once. The third line assigns correct value to COMSPEC environmental variable, which hasn't got proper value during interpretation of CONFIG.SYS file. Since this moment MS-DOS7 is ready to a change of the current disk.

Lines 4 and 5 deal with directory TEMP for temporary files. First, existence of this directory is checked. If it doesn't exist, an attempt is made to create TEMP directory. Then its existence is checked once more, and in case of success a path to TEMP directory is assigned as a value to environmental variable TEMP. This procedure guarantees a valid path for temporary files on any writable disk. On the other hand, absence of TEMP variable after this procedure certainly indicates that current disk is non-writable.

The following group of operations assigns values for other variables: DIRCMD, PROMPT, PATH. The shown values should be regarded as examples, which need to be corrected according to actual directory structure on your disk. It is implied, that value of the PATH variable (2.02-02) contains actual paths to all utilities, specified in the following lines: MODE.COM, KEYB.COM and VC.EXE, as well as other paths which you may want to supplement.

Execution of MODE.COM and of KEYB.COM activates desirable national codepage for display and corresponding keyboard's layout. If russian codepage 866 is not the one you need, you may change it, but beforehand it should be checked in table A.02-2 whether

the associated data tables (EGA3.CPI and KEYBRD3.SYS) contain the data you need. If not, these data tables must be changed too. Of course, the choice of 437-th (american) codepage makes you free to omit all lines with MODE.COM and KEYB.SYS, because this codepage is activated by default.

The last two lines in AUTOEXEC.BAT file serve to launch VC.COM – the Volcov Commander file manager (6.25). Other file managers, for example, Norton Commander (NC.EXE) and Dos Navigator (DN.EXE) may be launched in a similar way. If you don't intend to use file manager, then the shown two last lines are not needed.

9.01-03 Automatic determination of disk's letter-name

It were convenient to have a set of self-adaptive configuration files, which can load MS-DOS7 properly from any disk without manual correction of disk's letter-name. This opportunity can be easily implemented, if you have assembled yet the REASSIGN.COM utility, proposed in part 9.06, or have got any other functionally equivalent utility. All you have to do is to replace fixed assignment ("set dsk=C:") in 2-nd line of AUTOEXEC.BAT file (9.01-02) with the following two lines, performing automatic determination of disk's letter-name:

```
set dsk=33
\DOS\OTH\Reassign.com dsk
```

Of course, the path example ("\DOS\OTH\") must be changed, if necessary, according to the directory, where the utility actually can be found. After execution of these commands the letter-name of current disk becomes a value of DSK variable, and then all other relevant specifications are corrected automatically, as it is shown in article 9.01-02.

Automatic disk's letter-name determination doesn't necessarily need a special utility; it may be achieved exclusively by standard MS-DOS's means. But implementation of this idea is not so simple and, besides that, requiring access to a writable disk. Therefore presented here version of AUTOEXEC.BAT file wouldn't suit for loading MS-DOS7 from a CD-ROM, but it can be used and actually has been used for MS-DOS7 single-fold relocation onto a hard disk after formatting.

```
@echo off
if exist ..\nul goto L19
prompt=@echo off$_Set dsk$q$N:$_goto L7
%comspec% /f /c $.bat > $.bat
type Autoexec.bat >> $.bat
for %%Z in ("del A" "ren $.bat A" "A") do %%Zutoexec.bat
:L7
set comspec=%dsk%\Command.com
set Temp=%dsk%\TEMP
if not exist %Temp%\nul md %Temp%
```

```
prompt $p$g
set dircmd= /A /O:GNE /P
path ;
path=%dsk%\DOS\VC4;%dsk%\DOS\OTH;%dsk%\DOS\MS7;%dsk%\
Mode.com con codepage prepare=((866) %dsk%\DOS\DRV\Ega3.cpi)
Mode.com con codepage select=866
Keyb.com ru,866,%dsk%\DOS\DRV\Keybrd3.sys
set VC=%dsk%\DOS\VC4
Vc.com /TSR /no2E /noswap
:L19
```

Lines from 8 to 18 in this version of AUTOEXEC.BAT file perform ordinary functions, described in part 9.01-02. But lines 2 – 6 and labels are specific for this version. In order to make lines search easier the digits in label names represent ordinal numbers of corresponding lines.

The 2-nd line presents a check whether a parent directory exists for the current one. If it exists, the current directory can't be disk's root directory, and then we exit via L19 label. Hence, this file will do nothing, while it is stored anywhere inside directory structure. It becomes functional after it is moved into the root directory of a disk: then the parent directory doesn't exist, and the check in 2-nd line lets to proceed to the next line.

In the next 3-rd line the PROMPT command (3.22) sets a new prompt, which is issued only once while launching the command interpreter in the 4-th line. There command interpreter formally executes an empty file $.BAT, created just before by DOS while preparing redirection in the same line. The result is written into the same $.BAT file. As far as it was initially empty, it will be filled with nothing but new prompt. Let's assume that current disk is D: ; then contents of $.BAT file will look as follows:

```
@echo off
Set dsk=D:
goto L7
```

Note, that letter-name D: in the second line of $.BAT hasn't been preset beforehand, it is the actual current disk's letter-name, returned by DOS as an element of command prompt.

The TYPE command in next 5-th line reads AUTOEXEC.BAT file and sends its copy via output redirection so that it is appended to the shown three original lines in $.BAT file.

Cycle FOR in the 6-th line of AUTOEXEC.BAT performs three operations by sequential substitutions of three different values for dummy parameter %%Z. First substitution gives

```
del AUTOEXEC.BAT
```

and AUTOEXEC.BAT file ceases to exist. Then the second substitution produces command

```
ren $.BAT AUTOEXEC.BAT
```

and former $.BAT file becomes renamed into AUTOEXEC.BAT. The third substitution gives

```
AUTOEXEC.BAT
```

that is command to execute the new file AUTOEXEC.BAT beginning from its first line. Since name of this new file isn't preceded by the CALL command (3.02), there will be no return back to the executed former batch file, which currently doesn't exist yet. Note, that replacement of currently executed file can't be performed unless all replacement operations are written in one line.

New AUTOEXEC.BAT file begins with those three lines of former $.BAT file, which have been shown above. In course of their execution the letter-name of current disk is assigned as a value to DSK environmental variable and an unconditional jump is performed to label L7. Thus a group of lines, containing operations of self-modification, is bypassed for ever more. Starting from label L7 ordinary operations of AUTOEXEC.BAT file will be performed. During their execution the found letter-name of current disk will be automatically inserted into all paths, where it must be specified.

9.02 Command files, interpreted by debugger DEBUG.EXE

9.02-01 Boot sector saving and restoration

In every logical disk the first sector is boot sector. It contains parameter block BPB (A.03-4), an executable code part and specifications of those files, which should be given control for loading operating system. Boot sector is written by FORMAT.COM utility (6.15) each time the disk is formatted, and may be rewritten by SYS.COM utility (6.24), when disk is made bootable. Several special programs, such as DDO (Dynamic Drive Overlay), use non-standard boot sector configurations, which can't be restored by utilities, supplied with MS-DOS7.

You may need to save boot sector into a file in order to restore it later after occasional data distortion, or virus infection, or overwriting in course of operating system installation (examples – in article 9.11-02).

Though main command interpreter – COMMAND.COM – doesn't provide access to boot sector, for debugger utility DEBUG.EXE the boot sector saving is an easy task, solved exclusively with debugger's internal commands. Here is an example of command file, which induces DEBUG.EXE to do copy boot sector from disk C: into a file:

```
L CS:100 2 0 1
r BX
0000
r CX
200
n Bootsect.dat
w CS:100
q
```

The first line is a command to read boot sector from disk C: and to write its copy into memory starting from address CS:100 and on. Lines 2 – 5 specify length of data area (00000200h), which should be copied into a file. Line 6 specifies a name for the file to be created. Line 7 causes data copying from memory into a file. The last command ("Q") terminates debugger's session.

Text of the shown command file should be typed in editor program's window (NOTEPAD or EDIT.COM) and saved in current directory as a file with any suitable name (let it be named SAVEBOOT.SCR). Then this file should be sent to debugger by a command

```
DEBUG.EXE < SAVEBOOT.SCR
```

Result of execution is a 512-byte file BOOTSECT.DAT, appearing in current directory. This file is an exact image of boot sector on disk C:. You may read boot sector from any other valid logical disk in a similar way (6.05-10). In order to access disk A:, for example, you have to replace the first line in SAVEBOOT.SCR with the following one:

```
L CS:100 0 0 1
```

Of course, you can't address to a drive for removable media unless it has a valid removable media inside.

The reverse operation of writing a boot sector back to disk is performed by an even simpler command file:

```
n Bootsect.dat
L CS:100
w CS:100 2 0 1
q
```

Here the first line announces a name of file, containing boot sector image; this file must be present in current directory. The second line reads sector data from this file into memory, and the third line writes the sector data from memory back onto its original disk (disk C: in this example).

Text of the shown command file should be typed in editor program's window and then saved into a file with any suitable name, for example, RESTBOOT.SCR. As far as

interpretation of RESTBOOT.SCR implies direct access to a disk, this must be done under DOS, but not inside the "DOS box" under WINDOWS OS. Restoration of boot sector will happen, when command file RESTBOOT.SCR will be sent to its interpreter via input redirection:

```
DEBUG.EXE < RESTBOOT.SCR
```

With described command files the boot sector restoration becomes an easy operation. Nevertheless this operation shouldn't be used for boot sector transplantation. Even transplantation between diskettes of the same format raises a number of problems with uniqueness of their serial number and volume label, since the latter is duplicated in the root directory. Generally each boot sector is unique and can be suitable for no other disk except its original disk.

9.02-02 Copying Master Boot Record into a file

Each time you switch your computer on, it loads an installed operating system from its HDD (hard disk drive). This ordinary procedure includes several stages, and one important stage is execution of Master Boot Record (MBR), which specifies HDD's structure and directs loading process further. As any other record on a disk MBR is subjected to natural degradation, it may be damaged by an occasional fault or by virus. In any such case you will have to boot your computer from a recovery disc or from emergency diskette in order to restore MBR.

For Windows-95/98/ME operating systems you may try to restore MBR with FDISK.EXE utility (6.13), but it can't restore partition table, which is written in the same sector together with MBR (A.13-5). Specific forms of MBR, used by some other operating systems, by DDO (Dynamic Drive Overlay) and by boot managers, also can't be restored by FDISK.EXE.

Meanwhile there is a simple and universal solution: to store a copy of MBR together with partition table in a file on a removable media (compact disk or diskette). Then you will be able to restore MBR and partition table whenever you need from this file.

MS-DOS7 doesn't provide special means to copy MBR, but supplies debugger DEBUG.EXE, which enables you to write any succession of machine commands and can execute it at once. This article presents a command file with a succession of commands, inducing DEBUG.EXE to copy MBR into a file. Text of this command file looks as follows:

```
a 100
mov         DX,0080      ;prepare access to head 00h of HDD 80h
mov         CX,0001      ;specify start at cylinder 00h, sector 01h
mov         BX,0200      ;load BX with offset 0200h for data buffer
mov         AX,0201      ;specify function 02 of INT13, copy 1 sector
```

```
int         13          ;call INT13 handler, copy sector to buffer
mov         [01F6],AH   ;save exit code in memory cell 01F6h
ret                     ;quit execution of "G=100" instruction
org         130         ;write next commands from cell 130 and on
mov         AL,[01F6]   ;copy exit code into AL register
mov         AH,4C       ;specify termination function 4C (8.02-55)
int         21          ;call INT21 handler, terminate session

g =100
n Mbr.dat
r CX
0200
r BX
0000
w CS:0200
d 01F6,L1
g =130
```

Proposed commands should be typed in a window of editor program, as recommended in introduction article to chapter 9. Comments to the right of semicolons are not executed by DEBUG.EXE and hence may be omitted. Note an empty line between "INT 21" and "G =100" commands: it is important (7.01-04) and must be present. Then text should be saved in a file, which may be named, for example, READ_MBR.SCR.

If your computer is configured to boot not from logical disk C:, but from any other logical disk (D:, E: or other), you have to check actual number of bootable physical HDD, for example, by command

```
FDISK.EXE /status
```

From displayed table you will be able to determine the number of bootable physical HDD; if it is not number 1, then HDD's code 80h in second line of READ_MBR.SCR file must be corrected. If bootable drive is HDD number 2, then 81h HDD's code should be specified, if bootable drive is HDD number 3 – the 82h HDD's code, and so on.

Now it's time to explain how READ_MBR.SCR works. Its first line "A 100" switches DEBUG.EXE in assembler mode of operation for writing codes of machine commands into memory starting at address CS:0100h. While DEBUG.EXE stays in assembler mode, it allows to insert comments after semicolon sign, so the role of commands in lines 2 – 12 is evident from these comments. More detailed information about each command can be found in chapter 7. Two successions of machine codes are prepared: one starts from memory cell 100h, and the other, announced by ORG command in 9-th line, starts from memory cell 130h. Translation of commands into machine codes continues until empty line 13 is encountered: it forces DEBUG.EXE to leave assembler mode of operation.

Commands in following lines are not added to prepared successions of machine codes, but rather are executed at once.

Command "G" in 14-th line initiates execution of the first prepared machine codes succession from address CS:0100h. In the course of execution the MBR together with partition table are read from HDD and are written into memory from cell 0200h and on. Exit code of reading operation is temporary stored in arbitrary chosen free memory cell 01F6h. Execution of the first prepared machine codes succession is finished by RET command (7.03-73) in the 8-th line, which returns control back to debugger DEBUG.EXE.

DEBUG.EXE continues its job from command "N" (6.05-12) in 15-th line. It prepares name (MBR.DAT) for the file, which is to be created. You may specify other name or add a preceding path, if file is to be created elsewhere beyond the current directory. Commands in lines 16 – 19 specify length (00000200h) of the file, which is to be created. Command "W" (6.05-19) in 20-th line reads data from memory, starting at address CS:0200h, and writes these data into a file, which has its length and name (MBR.DAT) specified beforehand. Command "D" (6.05-04) in 21-st line displays reading operation exit code, stored in memory cell 01F6h.

The last command "G =130" in 22-nd line initiates execution of the second prepared succession of machine codes. Exit code of reading operation is copied from memory cell 01F6h into AL register, and then DOS's INT 21\AH=4Ch function (8.02-55) terminates debugger's session. Simultaneously the exit code from AL register becomes the errorlevel value, left behind by terminated debugger's session, so that later it may be checked with conditional "if errorlevel" command (3.15-03).

Before command file READ_MBR.SCR will be interpreted, you have to assure yourself, that it exists in the current directory on a writable media. As far as MBR will be saved in a file named MBR.DAT, any synonymous file should be removed from current directory, otherwise it will be overwritten without prompt. Having finished preparatory checks, you may send READ_MBR.SCR to debugger for interpretation:

```
DEBUG.EXE < READ_MBR.SCR
```

During interpretation a message "Program terminated normally" appears, but it means nothing more than DEBUG.EXE has successfully got control back after execution of the first succession of machine codes. Outcome of MBR reading attempt is represented by exit code – a hexadecimal number, displayed in penultimate row on the screen. Non-zero exit code informs about failure of MBR reading attempt. In this case a MBR.DAT file will be created containing nothing but garbage. On condition "if errorlevel 1" (3.15-03) it should be automatically deleted. Interpretation of non-zero exit codes according to table A.06-1 may help to reveal the cause of failure.

Exit code value 00h signifies success of MBR reading attempt. In this case file MBR.DAT contains a copy of MBR together with disk's partition table. An example of MBR.DAT file dump is shown in fig. 12 (in article A.13-5).

9.02-03 Restoration of Master Boot Record (MBR).

MBR failure is a relatively rare event, but no one has a guarantee to avoid it. Once something similar may happen to you. Then you have to test and exclude other suppositions: wrong BIOS settings, CMOS memory failure, disk's surface damage, boot sector overwriting, etc. Most convincing experiment is to save an image of current MBR into a file, as it is shown in preceding article 9.02-02, and to compare it with original MBR image, which you have saved beforehand in another file. Comparison may be performed by FC.EXE utility (6.12). If files differ, MBR must be rewritten or restored.

MS-DOS7 doesn't provide special means for writing MBR image from a file onto a hard disk, but you may prepare a command file, which will force DEBUG.EXE to do this job. Let's assume, that command file is named WriteMBR.SCR, that file with a copy original MBR image is named MBR.DAT, and that both files exist in current directory. Then contents of WriteMBR.SCR may look like this:

```
A 100
mov         DX,0080      ;prepare to access head 00h, hard disk 80h
mov         CX,0001      ;specify start at cylinder 00h, sector 01h
mov         BX,0200      ;load BX with offset 0200 of data buffer
mov         AX,0301      ;specify INT13 function 03h, write 1 sector
int         13           ;call INT13 handler, write data into sector
mov         [01F6],AH    ;save exit code in a memory cell
ret                      ;quit execution of DEBUG's "G" instruction

N MBR.DAT
L CS:0200
G =0100
d 01F6,L1
q
```

The "A 100" command in first line of this command file switches DEBUG.EXE into assembler mode, so that commands in following lines 2 – 8 are not executed at once, but rather are translated into machine codes written into memory cells from CS:0100h and on. While DEBUG.EXE stays in assembler mode, it allows to supply each line with commentaries. Therefore mission of commands in lines 2 – 8 is clear from command file itself. Empty line 9 forces DEBUG.EXE to exit assembler mode. Then "N" command (6.05-12) in 10-th line specifies name of the file to be loaded, and "L" command (6.05-10) in 11-th line loads MBR image from MBR.DAT file into memory starting at address CS:0200h.

The "G" command (6.05-07) in 12-th line initiates execution of prepared machine codes from address CS:0100h. Execution proceeds until in 8-th line the RET command (7.03-73) is encountered, which returns control back to DEBUG.EXE. Then execution of

commands in WriteMBR.SCR file is continued from 13-th line with "D 01F6,L1" command, which displays exit code, left after execution of INT13\AH=03h writing function. The last 14-th line with "Q" command terminates debugger's session.

Naturally, if your bootable disk is not the first physical HDD, you must change code of physical disk in the 2-nd line of WriteMBR.SCR file just as it is described in preceding article 9.02-02. When you are sure, that code of physical disk is specified properly, that MBR indeed needs to be restored, that all necessary files are prepared yet in current directory, then you may send WriteMBR.SCR file for execution with command

```
DEBUG.EXE < WriteMBR.SCR
```

After execution a hexadecimal exit code will be displayed on the screen. Interpretation of any exit code can be found in table A.06-1. Exit code 00h signifies, that MBR has been restored successfully.

Note 1: experiments with WriteMBR.SCR in order to ensure yourself shouldn't be performed upon HDDs which are currently in use! This may lead to unrepairable data loss! You may subject to experiments only those HDDs (both new and not new), which contain nothing worth saving.

Note 2: WriteMBR.SCR file needs direct access to disk. Therefore MBR restoration shouldn't be performed inside "DOS box" under WINDOWS OS, it should be performed under genuine MS-DOS7.

9.03 Examples of batch files

COMMAND.COM interpreter accepts ordinary command files via redirection, just as DEBUG.EXE does (9.02). But batch files represent a special class of command files, recognized and accepted by COMMAND.COM just from command line, without redirection. Moreover, in batch files COMMAND.COM can execute several important commands (3.02, 3.14, 3.21, 3.27), which can't be executed in ordinary command files or from command line. Because of these reasons a variety of command files for COMMAND.COM interpreter eventually is confined to class of batch files.

The most simple batch file in this book is AUTOEXEC.BAT file, presented in article 9.01-02. The articles below present somewhat less simple examples of batch files. These examples demonstrate techniques of batch programming, which may be useful far beyond the purposes of the shown batch files.

9.03-01 Batch file ARC.BAT for archiving

Term archiving according to computer terminology implies compression of several files into a combined file-archive. This meaning survived from those times long ago, when computers were not reliable, and personnel had to save multiple files as a combined data

stream written onto magnetic tape media. Since those times almost everything has changed, but interest to archiving hasn't vanished. Archiving decreases loss of free disk's space in clusters, makes faster both copying and defragmentation. File's size compression is essential because of limited transmission speed in communication networks. Packing of multiple files into one archive is convenient and is widely used for delivering large program products.

Known archiving algorithms are numerous, but only two of them – ZIP and RAR – have got wide-spread practical appreciation. ZIP algorithm, developed by Phil Katz in early 1990-ties, doesn't provide utmost compression, but has two other advantages: speed and compatibility. Archives, packed by original ZIP algorithm, can be unpacked by a lot of other archiving programs. Programs for packing and unpacking ZIP archives can be downloaded, for example, from site http://comp.site3k.net/?/comp/pkzip.html .

The RAR algorithm, developed by Eugene Roshal in middle 1990-ties, provides better compression and is able to emend partially damaged archives having internal recovery record. But new versions of WINRAR program (for WINDOWS OS) create archives, which can't be unpacked by earlier versions of RAR archivers. This incompatibility may let you down against your addressees. Therefore for forming RAR archives a non-newest, but sufficiently good free version 2.50 of RAR.EXE program (dated 1999) should be preferred. This version of RAR archiver can be downloaded from internet, for example, from site http://dosprogram.narod.ru/arc/index.html .

In order to make archive usage convenient, Volcov Commander file manager enables to enter inside archives with a mouse button click and to treat their contents almost as contents of ordinary directories (details – in article 6.25-04). Archive creation from file manager's menu is also much more convenient, than from command line. But file manager can't prevent user's mistakes in data preparation for archiving; moreover, finding a cause of a failure sometimes becomes even more difficult. Therefore an idea has emerged to write a command file, which will check the data transferred from Volcov Commander file manager to archiver program. This idea was implemented in a batch file, i.e. a command file for COMMAND.COM interpreter. Batch file ARC.BAT prevented archiver's failures, and error messages, sent by ARC.BAT to display, helped to elicit the cause of each error.

As years passed, number of checks in ARC.BAT file has grown up to seven. When a time has come to decide, which batch file should be presented as a useful and relatively simple example, then the ARC.BAT file was considered the most suitable.

Principle of ARC.BAT file is based on presumption, that in file manager's active panel the user chooses with right mouse's button a group of files, which are to be included into a new archive, and then with the left mouse's button highlights a filename, which will be assigned to the new archive. An example of such selection of files is shown in fig.5 (in article 6.25-01). After that a mouse button's click on corresponding menu item is enough, and new archive is created yet. If only one file manager's panel is opened at that moment,

then archive will be created in current directory. If both file manager's panels are opened, then destination directory will be that shown in opposite (inactive) panel.

In course of described procedure the states of both panels and names of selected files are sent via file manager's macrocommands, invoked by certain character combinations shown in article 6.25-02. These character combinations should be specified in a line of menu file VC.MNU (6.25-02) so that each data item, returned by macrocommand, becomes a value of a separate dummy parameter for ARC.BAT file. In particular, for creation of RAR archives the menu line invoking ARC.BAT may look like

```
@Arc.bat  RAR  !:  !~\  !~  !~@  %:  %~\
```

When this line in menu file is interpreted by Volcov Commander file manager, then each character combination, invoking a macrocommand, will be replaced by data item, returned by that macrocommand. Then command line with all performed replacements will be sent to command interpreter COMMAND.COM. The latter sets ordinal correspondence between data items in command line and dummy parameters of batch file ARC.BAT. Inside batch file the dummy parameters are denoted by ordinal numbers of corresponding data items in original command line – digits from 0 to 9, preceded by a percent sign (2.03-03). Lines of ARC.BAT file will be interpreted by COMMAND.COM, and then designators of dummy parameters will be replaced by corresponding data items. According to the shown order of data items the dummy parameters will get the following substitutions:

%0	– name of currently processed file (ARC.BAT)
%1	– RAR (or ZIP): requested archive type
%2	– letter-name of a disk, shown in active panel
%3	– path to a directory, opened in active panel
%4	– filename, highlighted by left mouse's button
%5	– file-list of files, selected by right mouse's button
%6	– letter-name of a disk, shown in inactive panel
%7	– path to a directory, opened in inactive panel

The data, received from file manager, will be complemented by other data, requested by ARC.BAT file from operating system. All these data will be taken into account in order to reveal mistakes, which may inhibit successful execution of archiving procedures. According to results of checks either an archiver program will be called for, or a comprehensive error message will be displayed. Full text of ARC.BAT file, comprising command lines with all checks, is shown below.

```
@echo off
set V1=02
if %1"==ZIP" set V1=Pkzip.exe
if %1"==RAR" set V1=Rar.exe
if %6"==" set V1=02
```

```
if %V1%==02 echo Parameters are invalid or not defined!
if %V1%==02 goto END
set V2=08
for %%Z in (%path%) do if exist %%Z\%V1% set V2=%%Z\%V1%
rem  ================== Line 10 ==================
if %V2%==08 echo The %V1% archiver hasn't been found!
if %V2%==08 goto END
set V3=13
for %%Z in (%path%) do if exist %%Z\Find.exe set V3=%%Z\Find.exe
if %V3%==13 echo The Find.exe utility hasn't been found!
if %V3%==13 goto END
if %7"==" echo Archive in inactive panel must be closed!
if %7"==" goto END
if %4"==.." echo Name for the archive isn't chosen!
rem  ================== Line 20 ==================
if %4"==.." goto END
%V3% /C /I /V ".%1" %5 | %V3% ": 0" > nul
if not errorlevel 1 echo Chosen file(s) - already %1-archive(s)!
if not errorlevel 1 goto END
%V3% /I "%4.%1" %5 > nul
if not errorlevel 1 if %2%3"==%6%7" echo Conflicting filenames!
if not errorlevel 1 if %2%3"==%6%7" goto END
ctty nul
%comspec% /f /c copy %V3% %6%7%4.%1 /Y | %V3% "1 f"
rem  ================== Line 30 ==================
ctty con
if errorlevel 1 echo Non-writable target disk or overwrite denied
if errorlevel 1 goto END
del %6%7%4.%1
if %1==RAR %V2% a -s- -rr -ems- -w%5\.. %6%7%4.%1 @%5
if %1==RAR if errorlevel 1 goto END
for %%Z in (1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1) do echo.
if %1==ZIP %V2% %6%7%4.%1 -ex -P -wHS -jhsr @%5
if %1==ZIP if errorlevel 1 goto END
rem  ================== Line 40 ==================
echo Archive %4.%1 is written into the %6%7 directory.
:END
```

Structure of ARC.BAT file is primitive: it is just a succession of checks. No cycles, no subroutines. Jumps are performed, when check conditions are not met. Destination of all jumps is a single label END in the last line. In order to make ordinal search for lines easier, each tenth line is a comment announcing line's number.

The first line of ARC.BAT file switches off ECHO flag, because otherwise it will be difficult to notice the displayed messages. Commands in lines 2 – 7 check value of the first dummy parameter and presence of 6-th dummy parameter's value. If either of these conditions is not met, then a message is displayed, that parameters are invalid or not defined. But if both conditions are met, then name of archiver program, which is to be called for, is assigned as a value to variable V1.

Command in the 9-th line checks whether the selected archiver program can be found in either of those directories, which are specified by paths in value of PATH variable (2.02-02). Proper value of PATH variable should be prepared beforehand: it is one of required conditions, stipulated in introduction article to chapter 9. Examples of value assignments to PATH variable are shown in all sets of configuration files, presented in this book. But whether the selected archiver program will actually be found – this is your own responsibility, though. If archiver program wouldn't be found, then an error message will be displayed, and jump from 12-th line to the END label will terminate execution. If archiver program will be found, then its name with preceding path will be assigned as a value to variable V2.

Similarly a command in 14-th line arranges a search for FIND.EXE utility (6.14), which is used in several further checks. If FIND.EXE utility will be found, its name with preceding path will be assigned as a value to variable V3.

Commands in lines 17 and 18 of ARC.BAT file check whether a value of the 7-th dummy parameter is empty. This value is empty, when inactive file manager's panel shows not a directory, but some other archive. In this case the error message, displayed by ARC.BAT, will prompt, that archive in inactive panel should be closed.

The 19-th line of ARC.BAT file checks that name, which the user had to highlight with mouse's left button click. This name later will be assigned to the created archive. But sometimes highlighted line in file manager's panel is left pointing at parent directory, represented by double dot alias. In this case error message will inform about invalid name choice.

Command in the 22-nd line of ARC.BAT checks contents of file-list with names of those files, which should be included into new archive. It's just the moment to remind, that value of the V3 variable, substituted twice in the 22-nd line, is FIND.EXE utility name with preceding path. Being called for the first time, the FIND.EXE utility counts in file-list the filenames not belonging to that type of archives, which is to be created. Presence of such archive in prepared group of files is allowed, but repetitive compression of archives only is inept: reliability of data storage will be decreased. Count result is transferred via intermediate redirection, and then the same FIND.EXE utility, being called for the second time, checks whether the count result is zero. If files of other types wouldn't be found in prepared group, then an error message will be displayed, that selected files already are archives of the requested type.

Command in 25-th line of ARC.BAT files searches inside the selected group for a file with the same name and the same suffix, which are specified for the new archive. If synonymous file exists yet, then new archive must be created outside the current directory, otherwise this synonymous file will be overwritten before it is included into new archive. If such situation is detected, then an error message will inform about conflict of filenames in current directory.

In the 29-th line of ARC.BAT file an attempt is undertaken to write a file, synonymous to new archive, into destination directory, where new archive is to be written. Destination directory is addressed with all those precautions, which are needed for addressing to an inaccessible media: the CTTY NUL command in 28-th line blocks up panic error messages, and a call for COMMAND.COM interpreter with /f parameter guarantees non-stop execution. It should be reminded, that value of COMSPEC variable, substituted in the 29-th line, is just the name of COMMAND.COM interpreter with preceding path. This value is assigned to COMSPEC variable automatically, when command interpreter is launched for the first time (6.04).

An attempt, undertaken in the 29-th line, may fail, if destination directory contains yet a synonymous file, protected by HRS attributes (6.01). Another cause of failure may be non-writable or write-protected media. In any case the outcome of writing attempt is reflected by a message, sent by COPY command into STDOUT channel. But in 29-th line STDOUT channel is redirected to FIND.EXE utility. If the latter registers writing attempt failure, then execution is terminated because of impossibility to create a file with prescribed name in destination directory. But in case of success the written file will be deleted by DEL command (3.09) in the 34-th line. Thus the last obstacle to a success of archiver's mission will be removed.

Before any program is called for, the way this program displays its messages should be taken into account. The RAR.EXE archiver, called in 35-th line, forms its screen field bypassing STDOUT channel, so that screen needs to be cleared later. When screen is cleared by CLS command, then the following concluding message is shown in the upper screen's row and becomes hidden under file manager's panels. Therefore in 37-th line screen is cleared by FOR command, shifting cursor 20 rows down. Besides that, file manager's panels must not be unfolded to their full length. Length of Volcov Commander panels may be trimmed by mouse or by arrow keys while ALT-F11 or ALT-F12 key combinations are kept pressed. After that selected panel's size should be stored in VC.INI file by SHIFT-F9 keystroke.

The PKZIP archiver sends its messages via STDOUT channel, therefore it is called in 38-th line after the screen is cleared. Due to preceding checks probability of archiving procedures failure is infinitesimal, nevertheless it is not neglected. Therefore in 36-th and in 39-th lines provisions are made for jumps to final END label, so that error message, sent by archiver program, will remain visible after interpretation of ARC.BAT terminates. When archiver mission is finished successfully, then ECHO command in the 41-st line

displays concluding message, informing about how the created archive is named and in which directory it is written.

Prepared batch file ARC.BAT should be stored in a directory, which may be accessed via paths, specified in value of PATH variable (2.02-02). Common directory with other file manager's files should be preferred. An example of Volcov Commander's menu file VC.MNU with lines launching ARC.BAT is shown in article 6.25-02.

9.03-02 Batch file for media status testing

When you have to repair an unknown computer, the first problem is to explore whether there are disk drives and whether these are accessible. Powerful utilities MSD.EXE (in MS-DOS6.22) and NDIAGS.EXE (from Norton Utilities software release) are aimed at solving this problem; both are compatible with MS-DOS7, but both don't report media status – whether the media is present, formatted, writable, etc.

Presented batch file – let it be named DISK.BAT – provides a short, but comprehensive survey of disk media status. The main idea is to test disk's accessibility by reading volume's label and then to test disk's writeability by writing the same label back. This is safe, because both success and failure of writing procedure wouldn't inflict changes on disk under test.

On the other hand, the DISK.BAT file may be regarded as a source of several non-obvious solution examples for some urgent tasks:

– incorporation of subroutines into a batch file;
– non-stop testing of disks, including inaccessible ones;
– avoiding undesirable error messages;
– generation of temporary command files;
– catching STDOUT output into an environmental variable.

Full text of DISK.BAT file is shown below. In order to make ordinal search for lines easier, each tenth line is a comment announcing line's number. Besides that, labels in DISK.BAT are named after numbers of corresponding lines; for example, label L28 denotes line 28, which marks end of the main part and start of a subroutine.

```
@echo off
if %1"==&" if not %2"==" goto L%2
if %Path%"==" %0 & 79 4 PATH
if %Temp%"==" %0 & 79 4 TEMP
Call %0 & 28 A Attrib D Debug F Find L Label
if VA"==" goto L87
set V1=%Path%
set Path=%1
set V2=%Path%
rem  ================== Line 10 ==================
```

```
set Path=%V1%
set V1=
Call %0 & 39 A B C D E F G H I J K L M N O P Q R S
if %V1%"==" if not %1"==" %0 & 79 5
if %1"==" set V1=A C D E F G O R
ctty nul
%VA% -H -R -S %Temp%\tmp.*
echo e 100 'call %%1 & 46' 20 > %Temp%\tmp.scr
echo w >> %Temp%\tmp.scr
rem  ================== Line 20 ==================
echo q >> %Temp%\tmp.scr
set Dircmd=
for %%Z in (%V1%) do call %0 & 53 %%Z: %1
del %Temp%\tmp.*
ctty con
for %%Z in (A D F L 1 2) do set V%%Z=
goto L87
:L28
shift
rem  ================== Line 30 ==================
shift
set VL=
for %%Z in (%path%) do if exist %%Z\%2.exe set VL=%%Z\%2.exe
if %VL%"==" echo Error: the %2.exe utility hasn't been found!
if %VL%"==" set VA=
if not %VL%"==" set V%1=%VL%
if not %4"==" goto L28
goto L87
:L39
rem  ================== Line 40 ==================
shift
if %V2%"==%2" set V1=%2
if %V2%"==%2:" set V1=%2
if not %3"==" goto L39
goto L87
:L46
set V1=%6
if not %7"==" set V1=%6 %7
if not %8"==" set V1=%6 %7 %8
rem  ================== Line 50 ==================
if %5"==has" set V1=NO NAME
goto L87
:L53
```

```
%comspec% /f /c Dir /-p %3\nul > %Temp%\tmp.txt
%VF% "Volume in d" < %Temp%\tmp.txt > %Temp%\tmp.bat
if errorlevel 1 %0 & 79 6 %3 %4
%VF% "Directory of " < %Temp%\tmp.txt
if errorlevel 1 %0 & 79 7 %3
%VF% "0 bytes free" < %Temp%\tmp.txt
rem  ================== Line 60 ==================
if not errorlevel 1 %0 & 79 8 %3
%VD% %Temp%\tmp.bat < %Temp%\tmp.scr
call %Temp%\tmp.bat %0
set V2=if errorlevel 20 del %Temp%\tmp.bat
%comspec% /f /c for %%Z in ("%VL% %3%V1%" "%V2%") do %%Z
%VF% "Volume Seria" < %Temp%\tmp.txt
if errorlevel 1 if not exist %Temp%\tmp.bat %0 & 79 9 %3
if errorlevel 1 if exist %Temp%\tmp.bat %0 & 73 1 %3
if not errorlevel 1 if not exist %Temp%\tmp.bat %0 & 73 2 %3
rem  ================== Line 70 ==================
if not errorlevel 1 if exist %Temp%\tmp.bat %0 & 73 3 %3
goto L87
:L73
if %3"==1" if %3"==A:" echo Disk %4 (%V1%) is writable > con
if %3"==1" if not %3"==A:" echo Disk %4 (%V1%) is a RAM-disk > con
if %3"==2" echo Disk %4 (%V1%) is write-protected > con
if %3"==3" echo Disk %4 (%V1%) is writable > con
Dir /a:ARD /-p /v %4\ | %Dsk%\DOS\MS7\Find.exe "otal d" > con
:L79
rem  ================== Line 80 ==================
if %3"==4" echo The %4 variable is not defined
if %3"==5" echo Wrong parameter, it must be a diskletter or none
if %3"==6" if not %5"==" echo Letter %4 doesn't refer to a disk > con
if %3"==7" echo Disk %4 has no media inside > con
if %3"==8" echo Disk %4 is probably a CD-ROM (no free space) > con
if %3"==9" echo Disk %4 is not formatted > con
:L87
```

The second line in DISK.BAT file is a check for the 1-st dummy parameter's value. If it is ampersand, a jump to subroutine is performed; if don't, execution of the main program's part is continued. Note, that address for the jump ("goto L%2") is not fixed, but rather is represented by value of the second dummy parameter. This enables to arrange subroutines as parts of the main batch file, otherwise subroutines have to be separate files.

Lines 3 – 22 specify preparation operations. First values of PATH and TEMP variables are checked. The PATH variable (2.02-02) must specify paths to MS-DOS7

files, the TEMP variable must specify a path to directory for temporary files. If either of these variables is not defined, then a jump to label L79 is performed, an error message is displayed, and interpretation of DISK.BAT file terminates. Examples of value assignments to PATH and TEMP variables are shown in all sets of configuration files, presented in this book.

The next check in the 5-th line calls for a subroutine, starting from label L28. This subroutine writes into environmental variables the names and paths of all utilities, which will be used for performing further tests. Names of these utilities, listed just in the 5-th line, are transferred to subroutine via its dummy parameters. Main operation of particular path extraction from PATH variable's value is performed in 33-rd line. If a path isn't found, then ECHO command in 34-th line displays error message. Until list of dummy parameters isn't empty, a jump from 37-th line to L28 label will repeat the same cycle for finding next path to the next utility. Finally paths to Attrib.exe, Debug.exe, Find.exe, Label.exe utilities will become values of variables VA, VD, VF, VL correspondingly.

After return from L28 subroutine the command interpreter continues execution of commands in lines 7 – 11. These operations convert disk's letter-name, specified by user in command line, to upper case. But user may specify some other sign instead of disk's letter-name. Therefore another subroutine, located in lines 39 – 45 of DISK.BAT, is called from 13-th line in order to determine, whether the specified sign belongs to list of disk's letter-names. If specified sign is rejected, then a jump is performed from 14-th line to label L79, an error message is displayed, and execution terminates. If specified sign is indeed a letter-name, it will be assigned as a value to variable V1, and then this disk only will be examined. But DISK.BAT may be launched without parameters, and then a predetermined group of disks should be examined. A list of corresponding letter-names is assigned as a value to variable V1 in 15-th line.

Disks examination procedures need three auxiliary files to be created in directory for temporary files: TMP.SCR, TMP.TXT, TMP.BAT. In order to guarantee creation of those auxiliary files, operation in 17-th line takes off attributes from those synonymous files, which may exist yet in that directory. Value of variable VA, substituted in 17-th line, is name of ATTRIB.EXE utility with preceding path. The first of auxiliary files – TMP.SCR – is created by output redirections in lines 18 – 21. Contents and purpose of this file will be explained later.

The FOR command in the 23-rd line launches the main exploration cycle, sequentially for each of the disks under test. Disks to be tested are represented by value of the V1 variable. Disk's examination is performed by a subroutine, located in lines 53 – 72 of DISK.BAT file. For examination of one disk a call for this subroutine looks like

```
call %0 & 53 %%Z: %1
```

where the word "call" means a command to return back into the same FOR cycle each time when execution of L53 subroutine terminates. Dummy parameter "%0" causes substitution

of batch file name: DISK.BAT or some other, if this file will be named otherwise. Parameters "& 53" specify label L53 as destination for the jump, which is to be performed from the second line. Substitution of disk's letter-name for "%%Z:" parameter specifies the disk to be examined.

It's important to notice, that FOR cycle in 24-th line is preceded by a CTTY NUL command (3.07) in 16-th line, which disrupts default communications with DOS's IN/OUT functions except those specified explicitly. This enables to avoid numerous error messages, which otherwise inevitably will be caused by access attempts to invalid or nonexistent disks. But error messages may be useful for debugging the DISK.BAT file itself; therefore during its first execution it is better to have the CTTY NUL command disabled by preceding it with REM command (3.24). If everything goes well, the preceding REM command in 16-th line should be removed.

Disk's examination subroutine starts at label :L53. The first test in 54-th line is performed by command

```
%Comspec% /f /c Dir /-p %3\nul
```

Just before execution the name of variable %COMSPEC% will be replaced by its value, that is by name of COMMAND.COM interpreter with preceding path. Hence, a separate resident module of command interpreter will be loaded. The "/f" parameter forces this resident module to work non-stop, automatically answering "FAIL" to all queries about errors. The "/c" parameter means that resident module must execute only one following command (DIR) and unload itself just afterwards. The message, sent by DIR command into STDOUT channel, is redirected in 54-th line into auxiliary file TMP.TXT.

Contents of TMP.TXT file are examined by FIND.EXE utility four times: in 55-th, 57-th, 59-th and 66-th lines. Name FIND.EXE is substituted in these lines for name of VF variable. The first examination enables to reject nonexistent drives; the second – drives for removable media having no media inside, the third examination reveals CD/DVD-ROM drives. If examination conditions are not met, then a jump to label L79 is performed, and a corresponding message is displayed on the screen. Through the first three examinations those disks only will pass, which exist, have a media inside and are not CD/DVD-ROMs.

A string, selected by FIND.EXE utility during the first examination in 55-th line, is sent via output redirection and is written into auxiliary file TMP.BAT; its contents, for example, may look like

```
Volume in drive D is EXTENDED1
```

After the first three examinations, in 62-nd line, contents of TMP.BAT file are transferred as data to DEBUG.EXE, which accepts commands from command file TMP.SCR. The latter is created beforehand by redirections in lines 18 – 21; it contains the following lines:

```
e 100 'call %1 & 46' 20
w
q
```

In the first line command "e 100" (6.05-05) forces DEBUG.EXE to overwrite a part of the loaded string, which thus is transformed to

```
call %1 & 46 ve D is EXTENDED1
```

The second command "w" (6.05-19) makes DEBUG.EXE to write the transformed string back into file TMP.SCR, and the third command "q" closes debugger's session.

After inflicted transformation the TMP.BAT file contains a CALL command (3.02) of COMMAND.COM interpreter. Execution of this CALL command will invoke that program, which will be defined by first dummy parameter %1 of TMP.BAT file. Just that happens, when TMP.BAT file is executed in 63-rd line of DISK.BAT file. But the first parameter of TMP.BAT file in 63-rd line is %0 dummy parameter of DISK.BAT file, i.e. the DISK.BAT file itself. Hence, command in 63-rd line performs a recursive call for DISK.BAT file. While this recursive call is executed, the "& 46" parameters define target label L46 for a jump from the second line, and the sixth parameter ("EXTENDED1" in the shown example) is volume's label of the examined disk. Therefore control is transferred to a subroutine in lines 46 – 51 of DISK.BAT file. This subroutine replaces former value of V1 variable with a value of 6-th dummy parameter – volume's label of the examined disk. Thus mission of L46 subroutine is fulfilled, and command interpreter returns to 64-th line of DISK.BAT file, to subroutine L53.

Command in 64-th line of DISK.BAT file prepares value of the V2 variable with a single purpose: to avoid wrap-around of command string in 65-th line, which otherwise were too long. Cycle FOR in 65-th line performs two operations, defined by values of variables VL, V1 and V2. The first operation is an attempt to write volume label back onto the same disk. The second operation is a conditional deletion of TMP.BAT file, if errorlevel value, returned by preceding writing attempt, informs about failure of this attempt. Since this moment existence of TMP.BAT file is an evidence of label writing attempt success and, consequently, signifies writeability of the examined disk.

A check in 66-th line reveals whether disk's serial number is present in file TMP.TXT. Result of check is represented by errorlevel value. This errolevel value together with existence of TMP.BAT file are those two arguments, which are enough for media status identification by conditional control transfers in lines 67 – 71 of DISK.BAT file. Control is transferred either to subroutine L73 or to subroutine L79, which both send to CON device (to display) messages, informing about status of the examined media.

Subroutine L73 is executed when examined media is found accessible, so that a command in 78-th line will be able to show size of disk under test and percent of occupied disk's space. Subroutine L79 is executed when status of examined media leaves no hope to get more information about it. Subroutines L73 or L79 terminate examination of one disk.

Then control is returned to continuation of cycle FOR in 23-rd line, where disk examination procedure was originally called for.

Cycle FOR in 23-th line will continue to call for disk's examination subroutine L53, sequentially specifying next disk's letter-names from list of disks to be examined. When all disks are examined, then DEL command in 24-th line deletes remaining auxiliary files, CTTY CON command in 25-th line restores default communications of DOS's I/O functions, and cycle FOR in 26-th line deletes all local environmental variables. A jump from 27-th line to final label L87 terminates execution of DISK.BAT file.

While using the DISK.BAT file it should be taken into account that it needs direct access to disks under test. Therefore DISK.BAT file shouldn't be launched inside "DOS box" under WINDOWS OS, it may be launched only under MS-DOS7 or MS-DOS8. Besides that, all 5 necessary conditions of successful execution, stipulated in introduction article to chapter 9, certainly must be met.

9.04 Configuration files with relocation to RAM-disk

Presented here versions of configuration files (CONFIG.SYS and AUTOEXEC.BAT) implement two alternatives: either ordinary booting mode or booting with arrangement of a virtual RAM-disk, followed by DOS relocation onto this RAM-disk. Virtual RAM-disk is much faster, than any real disk, its usage decreases load and wear of physical drives. But for computer's reparatory purposes a RAM-disk is essential because of other reason. When a computer boots itself from a removable media – a diskette or a CD-ROM – the drive, containing the media, is permanently busy with its mission. You can't insert other media in the drive unless operating system is relocated elsewhere. In these circumstances RAM-disk is the most suitable target to relocate DOS.

The RAMDRIVE.SYS driver (5.05-01), supplied in WINDOWS-95/98 release, enables to arrange virtual RAM-disks under MS-DOS7. Common problem for most RAM-disk drivers, including RAMDRIVE.SYS, is that the letter-name, assigned to virtual RAM-disk, isn't preset in advance. RAM-disk is given the first available letter-name after those assigned to fixed logical disk(s). But number of fixed logical disks in various computers may differ. Therefore the letter-name, assigned to RAM-disk, can't be known beforehand, it has to be determined. For this purpose MS-DOS8 employs two special files: batch file FINDRAMD.BAT and an executable file FINDRAMD.EXE. The proposed pair of configuration files (CONFIG.SYS and AUTOEXEC.BAT) solves letter-name determination problem otherwise, exclusively by MS-DOS7's means and without auxiliary files.

9.04-01 CONFIG.SYS file loading RAMDRIVE.SYS driver

This version of CONFIG.SYS file presents an example of a relatively simple block structure. The first block with reserved name [menu] offers two alternatives. When it is executed, two headers of menu items are shown, and you are invited to select one with UP and DOWN arrow keys. After your confirmation of your choice with ENTER keystroke, the loader IO.SYS assigns a name of the chosen alternative ("relocation" or "ordinary") as a value to the CONFIG environmental variable, and then proceeds to interpretation of commands in synonymous block of CONFIG.SYS file.

```
[menu]
numlock off
menuitem=relocation, Relocate DOS onto a 5.6 Mb RAM-disk
menuitem=ordinary, MS-DOS 7.10, ordinary loading
menudefault=relocation,20

[relocation]
include=ordinary
devicehigh=\DOS\DRV\Ramdrive.sys 5600 /E

[ordinary]
accdate c- d- e- r-
device=\DOS\DRV\Himem.sys /v
device=\DOS\DRV\Emm386.exe ram v
dos=high,umb,noauto
buffershigh=30,0
fileshigh=30
lastdrivehigh=Z
fcbshigh=1,0
stackshigh=8,256
country=007,866,\DOS\DRV\Country.sys
devicehigh=\DOS\DRV\Dblbuff.sys
devicehigh=\DOS\DRV\Ifshlp.sys
devicehigh=\DOS\DRV\Setver.exe
devicehigh=\DOS\DRV\Atapimgr.sys /W:6 /NDR /T:5 /LUN
devicehigh=\DOS\DRV\Oakcdrom.sys /D:CD001

[common]
installhigh=\DOS\DRV\Ctmouse.exe
installhigh=\DOS\DRV\Keyrus.com
shell=\Command.com \ /E:2016 /L:511 /U:255 /p
```

Block [ordinary] is similar to version of CONFIG.SYS file, presented in article 9.08-01, but there are two differences. First, the ATAPIMGR.SYS driver (5.07-01) is loaded in order to enable access to DVD-ROM disks. Second difference is that MSCDEX.EXE TSR program (5.08-03) is not loaded in this CONFIG.SYS file, because its loading here may affect allotment of disk's letter-names. An equivalent TSR program SHSUCDX.COM (5.08-04) will be loaded later from the next configuration file AUTOEXEC.BAT (9.04-02).

Naturally, all paths, specified in CONFIG.SYS file, must be correlated with actual directory structure in your computer. If you choose the [ordinary] alternative, MS-DOS7 will be loaded in an ordinary way, without relocation.

Block [relocation] consists of two lines. Command "include=" in the first line forces IO.SYS loader to execute all the lines of block [ordinary]. Then line 2 in block [relocation] loads Microsoft's RAM-disk driver RAMDRIVE.SYS (5.05-01). The latter creates a 5600 kb virtual RAM-disk in extended memory according to specified options. This size of RAM-disk can be afforded in all computers having 8 Mb of RAM memory or more, that is in all modern and even in somewhat obsolete computers.

The last block in this CONFIG.SYS file has a reserved name [common]. A block with this name is executed always, irrespective of the chosen alternative. Commands in [common] block load "mouse's" driver CTMOUSE.EXE (5.03-03) and combined driver KEYRUS.COM (5.02-05) for switching codepages and keyboard's layouts. The last line in [common] block transfers control to command interpreter COMMAND.COM: the latter has to complete configuration procedure with execution of the last configuration file AUTOEXEC.BAT (9.04-02).

9.04-02 AUTOEXEC.BAT file with relocation to RAM-disk

This version of AUTOEXEC.BAT file is devised for loading MS-DOS7 from logical disk A: with at least one directory \DOS. Logical disk A: may be represented by a real diskette or may be emulated by BIOS from an image stored in a CD-ROM disc. If you intend to use other disk to boot your computer, you have to

– change disk's letter-name assignment in the 5-th line;
– exclude this disk from list of disks to be tested (line 3 in "_relocation" section)
– correct conditions of "IF" commands in lines 4 and 5 in "_relocation" section.

Of course, the required corrections may be made automatically, if letter-name of current disk is determined, for example, by REASSIGN.COM utility (9.06), as it is shown in article 9.01-03. But here it's better to concentrate your attention on other problem – on a search for that letter-name, which is assigned to RAM-disk. Most often similar loading scenarios are initiated just from logical disk A:.

```
@echo off
if %1"==J" if not %2"==" goto _%2
prompt $p$g
set dircmd= /A /O:GNE /P
set dsk=A:
set comspec=%dsk%\Command.com
goto _%config%

:_relocation
echo.
echo Seeking RAM-disk as the last valid disk...
for %%Z in (C D E F G H I J K L) do call \Autoexec.bat J test %%Z:
if A:==%dsk% echo RAM-disk is not found!
if A:==%dsk% goto _ordinary
echo RAM-disk is assumed to be %dsk%
echo.
set comspec=%dsk%\Command.com
\DOS\MS7\Xcopy.exe \*.* %dsk%\ /S /E
%dsk%\Autoexec.bat J ordinary

:_ordinary
%dsk%
cd \
if not exist TEMP\nul %comspec% nul /f /c md TEMP
if exist TEMP\nul set Temp=%dsk%\TEMP
if %Temp%"==" echo Note: the TEMP variable is left not defined!
Lh \DOS\DRV\Shsucdx.com /D:?CD001 /L:N /~+ /R /Q
set VC=%dsk%\DOS\VC4
path ;
path=%VC%;%dsk%\DOS\OTH;%dsk%\DOS\MS7
Vc.com /TSR /no2E /noswap
goto _end

:_test
echo Testing disk %3 ...
%comspec% nul /f /c if exist %3\nul cd DOS
if exist ..\nul set dsk=%3
if exist ..\nul echo              valid
if not exist ..\nul echo              inaccessible
cd \
:_end
```

This version of AUTOEXEC.BAT begins with a conditional jump in 2-nd line, which enables recursive subroutine calls. When AUTOEXEC.BAT is interpreted for the first time, this jump is not performed. After that ordinary value assignments follow. The 7-th line performs a jump to a label, which is stored as a value of CONFIG variable since execution of CONFIG.SYS file (9.04-01). This value may be either "relocation" (if you have chosen DOS relocation) or "ordinary" (if you prefer to leave DOS "as it is").

If "ordinary" alternative is chosen, then commands in section "_ordinary" are interpreted. These include value assignments for variables TEMP and PATH, loading of SHSUCDX.COM TSR program for access to CD/DVD-ROMs and launching VC.COM file manager. Finally MS-DOS7 stays active on that disk, which was used to boot the computer.

If "relocation" alternative is chosen, then commands in section "_relocation" are interpreted. In the 3-rd line after "_relocation" label a cycle FOR is executed, which recursively calls for a test subroutine in the last section "_test" of the same AUTOEXEC.BAT file. Test procedure is applied to a number of disks, specified by their letter-names inside brackets of the FOR command. This is done in order to find the last letter-name, corresponding to a valid, accessible disk. Just this letter-name corresponds to RAM-disk, if IFS and network drivers are not loaded yet. The latter condition was the reason to postpone loading of SHSUCDX.COM TSR program until test cycle comes to end.

Disk's letter-names are sequentially sent to "_test" subroutine via the third dummy parameter %3. Second line after the "_test" label executes a presence test for the root directory of any given disk. This test can be applied even to inaccessible and nonexistent disks. If root directory exists in the disk under test, then on the current disk the current directory is changed to \DOS. The following lines check existence of the parent directory for the current one: the \DOS directory has a parent (the root), but the root itself has no parent. If current directory has been changed, then letter-name of the disk under test is assigned as a value to environmental variable DSK. The last line in "_test" section returns current directory to the root.

In a FOR cycle, being given a sequence of disk's letter-names, the "_test" procedure sequentially overwrites former values of DSK variable with next values, each representing a letter-name of next accessible disk. But letter-name of the last valid disk will not be overwritten, it will be the value of DSK variable after the FOR cycle comes to end. This value will be just the letter-name of RAM-disk.

When FOR cycle terminates, interpretation of commands in "_relocation" section continues. The COMSPEC variable's value is reassigned once more, this time pointing to the root directory of RAM-disk. Then XCOPY.EXE copies all non-hidden files together with whole directory structure from original (current) disk to RAM-disk. Hidden files (IO.SYS and MSDOS.SYS) at that moment have finished their mission and are no more

needed. Note, that utility XCOPY.EXE, which performs copying, must be able to find its library file XCOPY32.EXE in the same directory (i.e. in \DOS\MS7).

The last line in section "_relocation" recursively transfers control not to original AUTOEXEC.BAT file, but to its copy in the root directory of RAM-disk, because value of DSK variable has been changed and now points just at RAM-disk. Because of the same reason all further addressing, defined by the DSK variable, will also refer to RAM-disk.

Interpretation of AUTOEXEC.BAT file's copy continues from first line in section "_ordinary". Operation in its first line acquires great importance: it changes current disk to RAM-disk. Since that time original bootable disk becomes abandoned, all its files are closed yet, and it may be ejected off the drive. Now active operating system is a copy of MS-DOS7 on RAM-disk. All following operations in section "_ordinary" will be performed as if MS-DOS7 were normally loaded from RAM-disk.

9.05 Examples of simple utilities

Though debugger DEBUG.EXE (6.05) is not the most convenient instrument for making executable programs, nevertheless it enables to write simple programs of COM format. Remarkable feature of COM format is that program is laid out in PC's memory for execution just as it is presented in file. Therefore programmer's experience should start from writing the most simple programs of COM format. Simplest programs are not structured, don't appeal to disks, to files, to user. Simplicity enables to ignore some restrictions, including restrictions on allocated memory space (note 2 to 8.02-50).

Further in this part two examples of very simple programs are presented. Both were written spontaneously for solving unexpected problem. Later in internet more perfect programs have been found for solving the same problems. Nevertheless here the simplest original versions are presented, because perfection shouldn't be the main purpose for a beginner. At this stage our main purpose – your ability to understand the concept and to implement it.

9.05-01 Blue color brightness correction

An impetus for writing this simplest program was a display changeover from former CRT to a newer LCD. Due to different modulation characteristics of LCD screen the customary dark blue color of file manager's panels became too bright, causing visual irritation. In order to return former color to file manager's panels the BLUE.COM program has been written. It has effect on videocard's digital-to-analog converter (DAC) so that blue color brightness is decreased. BLUE.COM program is designed for videomode 03h only and has no adaptation capabilities. Nevertheless it has proved itself useful. May be, you'll find it useful too.

BLUE.COM file is produced by debugger DEBUG.EXE as a result of command sequence execution. This command sequence should be written into command file BLUE.SCR by means of editor program (as described in introduction article to chapter 9) and then sent to debugger via input redirection:

```
Debug.exe < Blue.scr
```

Command file BLUE.SCR has to contain the following lines:

```
A 100
MOV           AX,1010       ;100 Specify brightness function (8.01-24)
MOV           BX,0001       ;103 Prepare DAC's register number
MOV           CX,0015       ;106 CL - blue color brightness,
MOV           DX,0000       ;109               CH - green, DH - red
INT           10            ;10C Call for BIOS's INT10 handler
MOV           AX,4C00       ;10E Specify DOS's exit function (8.02-55)
INT           21            ;111 Call for DOS's INT21 handler

N Blue.com
R BX
0000
R CX
0013
W
Q
```

The first command "A 100" switches debugger DEBUG.EXE into assembler mode and specifies start address CS:0100h for writing machine codes. The next 7 lines define all actions of BLUE.COM program. Meaning of each action is explained by a comment after semicolon in each corresponding line. The 9-th line is left empty (7.01-04), it will force DEBUG.EXE to exit assembler mode. Then "N" command (6.05-12) announces a name for the file, which is to be created, and its length (00000013h = 19 bytes) is written into registers BX and CX. The "W" command in 15-th line creates file BLUE.COM and copies there the prepared machine codes. The "Q" command in the last line terminates debugger's session.

While command file BLUE.SCR is processed, the debugger displays a listing on the screen. Listing enables to monitor correctness of BLUE.SCR file's typesetting by comparison of actual offsets in listing with proper offsets, presented just after semicolon as the first item of comment in each line. Comparison technique is described in article 9.07-01. If listing doesn't reveal errors, then BLUE.COM utility is ready for execution. Further testing for such simple programs is not needed. If you are not satisfied with that blue color brightness level, which is set by BLUE.COM utility, then brightness value, written into CX register in the 4-th line of BLUE.SCR file, may be corrected as you want.

The corrected BLUE.SCR file should be sent once more to DEBUG.EXE via redirection, so that new version of BLUE.COM utility will be created.

The BLUE.COM utility should be launched from that line of AUTOEXEC.BAT file, which precedes launching of Volcov Commander file manager. Besides that, default brightness levels are reset anew, for example, by SCANDISK.EXE utility (6.21) and by LXPIC.EXE viewer, mentioned in relation to VC.EXT file (6.25-03). After such programs the BLUE.COM utility should be executed once more. This may be done automatically, if BLUE.COM is launched from the next line of common batch file or of common configuration file (VC.EXT, for example). Such examples are not presented here, because necessity of color correction depends on personal visual perception. Nevertheless blue color correction, performed by BLUE.COM utility, has been implemented for making screenshots fig.3 (in article 6.09) and fig.5 (in article 6.25-01).

9.05-02 How an ATX computer can be switched off

One more impetus for writing a simple program has emerged in 1999, when computers of ATX form-factor have ousted former AT computers. Newer ATX computers were designed for being switched off by operating system, and adjustable role of power button in ATX computers was difficult to foresee. DOS has never had a utility for switching power off. Since such utility was needed, it has been written, and it has got name TURN_OFF.COM.

TURN_OFF.COM file is produced by debugger DEBUG.EXE as a result of command sequence execution. This command sequence should be written into command file TURN_OFF.SCR by means of editor program (as described in introduction article to chapter 9) and then sent to debugger via input redirection:

```
Debug.exe < Turn_off.scr
```

Command file TURN_OFF.SCR has to contain the following lines:

```
a 100
mov         AX,5301      ;100 Specify APM activation function
mov         BX,0000      ;103 Prepare identifier of APM BIOS
int         15           ;106 Call INT15 handler for activation
mov         AX,530E      ;108 Specify request for APM emulation
mov         BX,0000      ;10B Prepare identifier of APM BIOS
mov         CX,0102      ;10E Request emulation of version 1.2
int         15           ;111 Call INT15 handler for emulation
mov         AX,5307      ;113 Specify power supply mode function
mov         BX,0001      ;116 Prepare all device's identifier
mov         CX,0003      ;119 Request power OFF operation
int         15           ;11C Call INT15 handler for power OFF
```

```
mov           AX,4C00       ;11E Specify DOS's exit function code
int           21            ;121 Call DOS's INT21 handler for exit

n turn_off.com
r BX
0000
r CX
0023
w
q
```

Proposed command file TURN_OFF.SCR is indeed very simple, it doesn't specify conditional jumps, it uses two types only of machine codes, and produced executable file is as short as 35 (23h) bytes.

The first command "A 100" switches debugger DEBUG.EXE into assembler mode and specifies start address CS:0100h for writing machine codes. The next 13 lines define all actions of TURN_OFF.COM program. Meaning of each action is explained by a comment after semicolon in each corresponding line. More detailed information about INT 15\AH=53h functions can be found in articles 8.01-70 – 8.01-72.

The 15-th line is left empty (7.01-04), it will force DEBUG.EXE to exit assembler mode. Then "N" command (6.05-12) announces a name for the file, which is to be created, and its length (00000023h = 35 bytes) is written into registers BX and CX. The "W" command in 21-st line creates file TURN_OFF.COM and copies there the prepared machine codes. The "Q" command in the last line terminates debugger's session.

In order to monitor correctness of TURN_OFF.SCR file's typesetting a comment in each line starts with proper offset of corresponding machine command. This offset should be compared with actual offset, shown by debugger in displayed listing. Comparison technique is described in article 9.07-01. It may be expedient to check the length of created file either with DIR command (3.10) or by length indication in file manager's panels. If both listing and length check don't reveal errors, then TURN_OFF.COM utility is ready for execution. Further testing for such simple programs usually isn't necessary.

While using the TURN_OFF.COM utility it should be taken into account that most part of computers produced in previous millennium have no APM system and therefore will ignore calls for INT 15\AH=53h functions. Besides that, TURN_OFF.COM utility shouldn't be run inside "DOS box" under WINDOWS OS, it may be run under genuine DOS only. It is convenient to launch TURN_OFF.COM via file manager's menu (example – in article 6.25-02).

It were desirable, of course, to make provisions against probable errors: against execution inside "DOS box" and against absence of APM system in a particular computer. However, then TURN_OFF.COM utility were not so simple. On the other hand, relevant

supplementary checks are also not too complex. For example, recognition of WINDOWS OS operating environment is implemented in lines 13Ah – 141h of a file in article 9.10-02. Multiple examples of error indication are shown in articles 9.06, 9.08 and 9.10. Having acquainted yourself with these examples, you will be able to upgrade TURN_OFF.COM utility as you want.

Note 1: unprepared loss of power supply causes loss of these data, which may be not written yet at that moment by SMARTDRV.EXE driver (5.06-01) from cache buffer to disk. This is equally relevant for both unexpected mains power loss and for switching power OFF with TURN_OFF.COM utility. If you intend to employ SMARTDRV.EXE driver, you may force data writing to disk just before TURN_OFF.COM utility is launched, but it's safer to disable write-behind caching at all.

9.06 Value assignment to an environmental variable

Long time ago DOS has been developed as general purpose operating system, and its set of commands has been formed according to this purpose. But now DOS's role has specialized, so that former set of commands now seems partially redundant and partially deficient. This article presents a utility performing three operations, which replenish the most urgent deficiencies of DOS's commands set. In particular, just these three operations enable to implement adaptive loading scenario, described in part 9.09.

As far as proposed utility replaces current value of some environmental variable with requested other data, it has been named REASSIGN.COM. File REASSIGN.COM is not large – in all 992 bytes long – because it relies on cooperation with DOS's internal SET command (3.26) and doesn't duplicate its functions. The SET command prepares an environmental variable, and the first character of prepared variable's value defines operation for REASSIGN.COM utility:

1 – report size of largest free XMS memory block;
2 – accept input from keyboard;
3 – report letter-name of the current disk.

The rest characters in prepared value are ignored, but their presence reserves space for the new value. If this space is not enough to contain the returned result, then an error message is displayed. If the result doesn't occupy the whole reserved space, its rest part is filled with space characters (20h).

When computer boots from any removable media, letter-name of the current disk is assigned by BIOS. Function 3 of REASSIGN.COM utility enables to determine the assigned letter-name:

```
set disk=33
Reassign disk
echo Current disk is %disk%
```

Prepared value 33 will be replaced by a new one, for example, D:. Since assigned letter-name becomes known, all further addressing in configuration files can be adapted automatically.

The next problem of adaptive loading procedures is to determine feasible size of RAM-disk for DOS relocation, whereas the amount of available memory isn't known beforehand. But function 1 of REASSIGN.COM utility gives the answer:

```
set xms=11111
Reassign xms
echo Largest free XMS memory block is %xms% kb
```

The last line in the example above displays the returned result. Now you are ready to specify size of RAM-disk, and function 2 of REASSIGN.COM utility will accept the desired value:

```
echo Specify the size of RAM-disk in kb:
set ramdisk=22222
Reassign ramdisk
Tdsk.exe R: %ramdisk% 512 /M /F:2
```

During execution of function 2 the REASSIGN.COM utility invites you to input the desired value via keyboard. Wrong digits may be erased with BackSpace keystroke. When data input is completed, further execution should be resumed by ENTER keystroke. Command in the last line of the shown example substitutes desired size value into a set of parameters for TDSK.EXE driver (5.05-02), which arranges RAM-disk of desired size. Of course, REASSIGN.COM utility can find a lot of other applications, besides the mentioned ones.

It is important to note, that all shown examples can't be executed from command lines, presented by Norton Commander, Volcov Commander or similar file managers. The reason is that file managers execute each command line inside a separate environment. Hence a value, set by SET command in one command line, becomes unavailable in the next command line. But ordinary command lines as well as lines inside batch files are executed in common environment, so that there all shown examples will be executed properly.

REASSIGN.COM utility is produced by debugger DEBUG.EXE as a result of command sequence execution. This command sequence should be written into command file REASSIGN.SCR by means of editor program (as described in introduction article to chapter 9). Verbal commentaries may be omitted. Special attention should be paid to empty line – 8-th from the end. It must be there, because empty line forces DEBUG.EXE

to exit assembler mode (7.01-04). Then command file REASSIGN.SCR should be sent to debugger via input redirection:

```
Debug.exe < Reassign.scr
```

Command file REASSIGN.SCR has to contain the following lines:

```
a 100
      ;************* Reassign.com **************
      ;********* Section 1: initial preparations
         ; 110 - target offset for jump from line 104
cmp           SP,2010       ; 100 Allotted less than 8 kb?
jbe           0110          ;*104 If yes, leave it intact
mov           SP,1FFE       ; 106 Set stack's top at 8 kb
mov           BX,0200       ; 109 Request for 8 kb space
mov           AH,4A         ; 10C Call for free MCB
int           21            ; 10E        creation function
mov           DX,03A8       ;=110 Message 4 offset (help)
cmp byte ptr  [005D],20     ; 113 Is 1-st byte in FCB free?
jz            0151          ;*118 If yes, go to display help
cmp byte ptr  [005D],3F     ; 11A 1-st byte - question mark?
jz            0151          ;*11F If yes, go to display help
cld                         ; 121 Set count upwards (DF=UP)
      ;********** Section 2: search results analysis
call          0173          ;*122 Call for PSP test subroutine
cmp byte ptr  [0165],F7     ; 125 1-st or 2-nd cycle errors?
ja            0151          ;*12A Exit, if yes
mov           DX,0345       ;*12C Prepare message 2 offset
les           DI,[00F8]     ; 12F ES:DI - address of variable
ES:                         ; 133 Read 1-st character in
mov           BL,[DI]       ; 134           variable's value
cmp           BL,31         ; 136 Lower limit: function 1
jb            0151          ;*139 Exit, if value is less
cmp           BL,33         ; 13B Upper limit: function 3
ja            0151          ;*13E Exit, if value is greater
shl           BL,1          ; 140 multiply by 2
mov           BH,00         ; 142 Prepare BX for calculation
call          [BX+0105]     ;*144 Call functional subroutines
jz            0151          ;*148 ZR state - message display
jb            015F          ;*14A CY - fail, errorlevel in DH
      ;************* Section 3: execution conclusion
         ; 151 - target for 118, 11F, 12A, 139, 13E, 148
         ; 15F - target for jump from lines 14A, 14F
call          01F9          ;*14C Call for copying subroutine
jmp           015F          ;*14F Jump to termination
```

```
mov           BX,DX         ;=151 DS:DX - message address
mov           CX,[BX-02]    ; 153 CX = number of characters
mov           BX,0001       ; 156 BX = handle to STDOUT
mov           AH,40         ; 159 Call for DOS's message
int           21            ; 15B            display function
mov           DH,F0         ; 15D Errorlevel = F0h
mov           AL,DH         ;=15F Restore errorlevel
mov           AH,4C         ; 161 Call for DOS's program
int           21            ; 163        termination function
      ;************** Section 4: data block.
         ; 165 - used in lines 125, 1D1, 1D6, 1F4, 205
dw            00FC
         ; 167 = (2*31 + 105) used in lines 144, 221, 22D
         ; 169 - used in lines 225, 236, 240, 247, 252, 280
dw            0213,029A,02DC,0000,0000,0000
      ;************** Section 5: PSP test subroutine
         ; 173 - target for calls from lines 122, 1A3
         ; 19F - target for a jump from line 197
         ; 1A6 - target for jumps from 17A, 183, 18F, 19D
xor           DI,DI         ;=173 Write zero in DI register
ES:                         ; 175 Does segment start
cmp word ptr  [DI],20CD     ; 176          from CD20 command?
jnz           01A6          ;*17A Exit, if not
ES:                         ; 17C Is there CD21 command
cmp word ptr  [0050],21CD   ; 17D            at offset 0050h?
jnz           01A6          ;*183 Exit, if not
push          ES            ; 185 Save PSP segment address
ES:                         ; 186 Load environment's segment
mov           ES,[002C]     ; 187            into ES register
call          01B3          ; 18B Call for name search
pop           ES            ; 18E Restore PSP address in ES
jz            01A6          ;*18F Exit, if name isn't found
mov           AX,ES         ; 191 Load PSP segment into AX
mov           DI,0016       ; 193 Is word in ES:[0016] cell
scasw                       ; 196     equal to segment in AX?
jnz           019F          ;*197 If no, word is parent's PSP
mov           DI,0010       ; 199 Is word in ES:[0010] cell
scasw                       ; 19C     equal to segment in AX?
jz            01A6          ;*19D If yes, exit, nothing found
ES:                         ;=19F Load parent's PSP segment
mov           ES,[DI-02]    ; 1A0            into ES register
call          0173          ; 1A3 Call for PSP test subroutine
ret                         ;=1A6 Return from subroutine
```

```
      ;************* Section 6: name search subroutine
         ; 1A7 - target for jumps from lines 1C8, 1CF
         ; 1B3 - target for call from line 18B
         ; 1DF - target for jumps from lines 1B1, 1BA
mov           CX,0100       ;=1A7 Set number of comparisons
mov           AL,00         ; 1AA Compare with zero value
repnz                       ; 1AC Repeat till 00 is found
scasb                       ; 1AD Compare [ES:DI] with AL=00
cmp           CX,0000       ; 1AE If number of repetitions
jz            01DF          ;*1B1        expires, exit search
mov           DX,0318       ;=1B3 Entrance point to search
ES:                         ; 1B6 If [DI]==00h, hence
cmp byte ptr  [DI],00       ; 1B7     environment is searched
jz            01DF          ;*1BA     up to end, exit search
mov           SI,005D       ; 1BC Name address - in DS:SI
mov           CX,000A       ; 1BF Set number of comparisons
repz                        ; 1C2 Repeat until difference
cmpsb                       ; 1C3 Compare [DS:SI] and [ES:DI]
cmp byte ptr  [SI-01],20    ; 1C4 Does name end with space?
jnz           01A7          ;*1C8 If not, compare next name
ES:                         ; 1CA Is there equality sign
cmp byte ptr  [DI-01],3D    ; 1CB            after the name?
jnz           01A7          ;*1CF If not, compare next name
sub byte ptr  [0165],04     ;*1D1 Shift record offset by 04
mov           SI,[0165]     ;*1D6 Load record offset into SI
mov           [SI],DI       ; 1DA Save name's address
mov           [SI+02],ES    ; 1DC Save environment's segment
ret                         ;=1DF Return from subroutine
      ;******** Section 7: value copying subroutine
         ; 1E0 - target for jump from line 210
         ; 1E5 - target for jump from line 1F1
         ; 1F3 - target for jumps from lines 1E9, 1EE
         ; 1F9 - target for a call from line 14C
         ; 1FB - target for jump from line 202
         ; 204 - target for jump from line 1FF
CS:                         ;=1E0 Load source address
lds           SI,[00F8]     ; 1E1               into DS:SI
ES:                         ;=1E5 Is there free space at
cmp byte ptr  [DI],00       ; 1E6        destination address?
jz            01F3          ;*1E9 If not, start next cycle
cmp byte ptr  [SI],00       ; 1EB Is the source empty?
jz            01F3          ;*1EE If yes, start next cycle
movsb                       ; 1F0 Let's copy one byte
```

```
jmp           01E5          ;*1F1 Jump to check next byte
CS:                         ;=1F3 Calculate cell offset with
add word ptr  [0165],0004   ; 1F4    next destination address
mov           AL,20         ;=1F9 Subroutine entrance point
ES:                         ;=1FB Is there a space to fill
cmp byte ptr  [DI],00       ; 1FC     at destination address?
jz            0204          ;*1FF If not, finish filling
stosb                       ; 201 Send space to destination
jmp           01FB          ;*202 Go to check next byte
CS:                         ;=204 Load offset of cell with
mov           SI,[0165]     ; 205   destination address in SI
CS:                         ; 209 Load destination address
les           DI,[SI]       ; 20A                   into ES:DI
cmp           SI,00F8       ; 20C Is cell offset the last?
jnz           01E0          ;*210 If not, start next cycle
ret                         ; 212 Return from subroutine
      ;****** Section 8: function 1, XMS-memory space
         ; 213 - target address, stored in cell 167
         ; 256 - target for jumps from lines 23E, 247, 250
         ; 25B - target for jump from line 263
         ; 269 - target for jump from line 277
         ; 275 - target for jump from line 272
         ; 284 - target for jump from line 27E
         ; 285 - target for jumps from lines 21A, 234
mov           AX,4300       ;=213 Function 1 entrance point
int           2F            ; 216 Check whether HIMEM.SYS
cmp           AL,80         ; 218         driver is installed
jnz           0285          ;*21A Exit, if not installed
mov           AX,4310       ; 21C Driver's entrance request
int           2F            ; 21F Entrance address - in ES:BX
mov           [0167],BX     ; 221 Store entrance offset
mov           [0169],ES     ; 225 Store entrance segment
mov           BL,00         ; 229 Prepare 00h in BL register
mov           AH,08         ; 22B Code of XMS space request
call far      [0167]        ; 22D Send request to HIMEM.SYS
cmp           BL,00         ; 231 Is request satisfied?
jnz           0285          ;*234 Exit, if not
mov byte ptr  [0169],00     ; 236 Let errorlevel be 00 if
cmp           AX,1900       ; 23B   free XMS area is not
jb            0256          ;*23E   enough for 6 Mb disk
inc byte ptr  [0169]        ; 240 Let errorlevel be 01 if
cmp           AX,4900       ; 244   5600 Kb XMS-disk can
jb            0256          ;*247   be arranged
```

```
inc byte ptr [0169]       ; 249 Let errorlevel be 02 if
cmp          AX,8C00      ; 24D   XMS-disk must be limited
jb           0256         ;*250   to 16 Mb, if not, then
inc byte ptr [0169]       ; 252   let errorlevel be 03
mov          BP,SP        ;=256 Store current SP state
mov          BX,000A      ; 258 Set decimal divisor
xor          DX,DX        ;=25B Write zero into DX register
div          BX           ; 25D Divide by decimal divisor
push         DX           ; 25F Push remainder into stack
cmp          AX,0000      ; 260 Is dividing finished?
jnz          025B         ;*263 If not, go to next cycle
les          DI,[00F8]    ; 265 Target address - into ES:DI
pop          AX           ;=269 Pop a digit out of stack
add          AL,30        ; 26A Translate digit to ASCII
mov          SI,DI        ; 26C Store DI state in SI
ES:                       ; 26E Is there free space
cmp byte ptr [DI],00      ; 26F          at target address?
jz           0275         ;*272 Skip writing, if not
stosb                     ; 274 Write digit, increment DI
cmp          SP,BP        ;=275 All digits are popped?
jb           0269         ;*277 If not, go pop next digit
mov          DX,036F      ;*279 Prepare 3-rd message offset
cmp          DI,SI        ; 27C Has writing been skipped?
jz           0284         ;*27E If skipped, skip errorlevel
mov          DH,[0169]    ; 280 Read errorlevel into DH
ret                       ;=284 Return from subroutine
mov          DH,FF        ;=285 Prepare errorlevel = FF
stc                       ; 287 Indicate failure by CF
ret                       ; 288 Return from subroutine
      ;******* Section 9: function 2, keyboard input
         ; 289 - target for jump from line 2AD
         ; 29A - target from 169, 28D, 291, 295, 2B4, 2BD
         ; 2BF - target for jump from line 2A8
         ; 2C7 - target for jumps from lines 2A3, 2C3
ES:                       ;=289 Let's check whether
cmp byte ptr [DI],00      ; 28A              buffer is full
jz           029A         ;*28D If yes, wait 1Bh, 0Dh, 08h
cmp          AL,20        ; 28F Characters below 20h
jb           029A         ;*291       shouldn't be accepted
cmp          AL,3D        ; 293 Equality sign
jz           029A         ;*295       shouldn't be accepted
int          29           ; 297 Display inputted character
stosb                     ; 299 Copy it into ES:DI buffer
```

```
mov          AH,10        ;=29A Function 2 entrance point
int          16           ; 29C Read inputted character
mov          DH,FF        ; 29E Errorlevel = FFh
cmp          AH,01        ; 2A0 Is the ESC key pressed?
jz           02C7         ;*2A3 If yes, no copying, exit
cmp          AH,1C        ; 2A5 Is the ENTER key pressed?
jz           02BF         ;*2A8 If yes, copy and then exit
cmp          AH,0E        ; 2AA Is BackSpace key pressed?
jnz          0289         ;*2AD If not, start next cycle
ES:                       ; 2AF Check, whether offset
cmp byte ptr [DI-01],3D   ; 2B0     in DI would point ahead
jz           029A         ;*2B4     to buffer's first cell
mov          AX,2008      ; 2B6 Output backspace sign and
call         02D2         ;*2B9          a space via INT 29
dec          DI           ; 2BC Decrement offset in DI by 1
jmp          029A         ;*2BD Return to start of cycle
cmp          DI,[00F8]    ;=2BF Has offset in DI changed?
jz           02C7         ;*2C3 If not, let's leave DH = FF
mov          DH,00        ; 2C5 If yes, let's set DH = 00
mov          AX,0A0D      ;=2C7 Output of line feed and
call         02D2         ;*2CA         CR signs via INT 29
cmp          DH,20        ; 2CD Set flags by comparison
cmc                       ; 2D0 Reverse state of CF flag
ret                       ; 2D1 Return from subroutine
      ;***** Section 10: function 2, output subroutine
         ; 2D2 - target for calls from lines 2B9, 2CA
         ; 2D5 - cycle return offset from line 2D9
mov          CX,0003      ;=2D2 Preset repetitions limit
int          29           ;=2D5 Send character to display
xchg         AL,AH        ; 2D7 Exchange characters
loop         02D5         ;*2D9 Cycle iteration check
ret                       ; 2DB Return from subroutine
      ;***** Section 11: function 3, disk determination
         ; 2DC - target address, stored in cell 16B
         ; 2EE - target for call from line 2E9
         ; 2F6 - target for call from line 30C
         ; 305 - target for call from line 2FE
         ; 30E - target for calls from lines 2F9, 308
mov          AH,19        ;=2DC Function 3 entrance point
int          21           ; 2DE Determine current disk
mov          DL,AL        ; 2E0 Copy disk number into DL
add          AL,41        ; 2E2 Translate number into ASCII
stosb                     ; 2E4 Copy ASCII code into ES:DI
```

```
ES:                         ; 2E5       and increment DI by 1
cmp byte ptr [DI],00        ; 2E6 Is buffer full?
jz           02EE           ;*2E9 If yes, don't append colon
mov          AL,3A          ; 2EB ASCII code of colon - in AL
stosb                       ; 2ED Copy colon's code to ES:DI
mov          DH,00          ;=2EE Prepare zero errorlevel
push         DI             ; 2F0 Save DI pointer in stack
mov          AX,0803        ; 2F1 Query for 1-st DDT address
int          2F             ; 2F4 1-st DDT address - in DS:DI
cmp          [DI+04],AL     ;=2F6 Is it a floppy disk?
ja           030E           ;*2F9 If no, don't search further
cmp          [DI+04],AH     ; 2FB If disk drive is the same,
jz           0305           ;*2FE        it should be skipped
mov          AH,[DI+04]     ; 300 Read disk drive number
inc          DH             ; 303 Increment number of drives
cmp word ptr [DI],FFFF      ;=305 Is this DDT the last?
jz           030E           ;*308 If yes, no further search
lds          DI,[DI]        ; 30A Next DDT address - in DS:DI
jmp          02F6           ;*30C Go to investigate next DDT
pop          DI             ;=30E Restore DI state from stack
push         CS             ; 30F Restore original
pop          DS             ; 310               segment in DS
cmp          DH,FF          ; 311 Clear ZF flag to NZ state
clc                         ; 314 Clear CF flag to NC state
ret                         ; 315 Return from subroutine
     ;********************* Section 12: message texts
dw           002B
        ; 318 - 1-st message, mentioned at 1B3
db           0D 0A 'ERROR: specified name hasn'
db           27 't been found' 0D 0A
dw           0028
        ; 345 - 2-nd message, mentioned at 12C
db           0D 0A 'ERROR: invalid value of the'
db           20 'variable' 0D 0A
dw           0037
        ; 36F - 3-rd message, mentioned at 279
db           0D 0A 'ERROR: no space for new value,'
db           20 'old one is too short' 0D 0A
dw           0138
        ; 3A8 - 4-th message (help), mentioned at 110
db           0D 0A 09 'Reassign.com overwrites value'
db           20 'of existing variable' 0D 0A 'Usage:'
db           0D 0A 09 'Reassign Anyname' 0D 0A 'Anyn'
```

```
db            'ame - name example (up to 8 letters) o'
db            'f a variable' 0D 0A 'The first in its'
db            20 'value must be a digit 1, 2 or 3 - i'
db            't defines function:' 0D 0A 09 '1 - get'
db            20 'size of largest free XMS block' 0D 0A
db            09 '2 - accept keyboard' 27 's input' 0D
db            0A 09 '3 - get current disk letter' 0D 0A
         ; 4E0

n Reassign.com
rbx
0000
rcx
03E0
w
q
```

Text of REASSIGN.COM utility begins with procedure releasing that memory, which certainly wouldn't be needed (details – in note 5 to A.12-7). Then in lines 110 – 11F a check is performed for that name of environmental variable, which should be read by command interpreter from command line, translated to upper case and written into first FCB block (note 4 to A.07-1) starting at offset 005Dh. If a cell at offset 005Dh is empty or contains an interrogation sign, then REASSIGN.COM displays help message and returns control to command interpreter, leaving errorlevel 240. Otherwise the cell at offset 005Dh is considered containing a name of that environmental variable, which is to have its value reassigned.

Section 2 begins in line 122 with a call for variable's name search in current and in underlying environments. Search procedure includes PSP test subroutine, presented in section 5, and name search subroutine, presented in section 6. Commands in lines 175 – 183 check specific PSP signatures in cells at offsets 0000h and 0050h. If PSP validity is confirmed, then address of corresponding environment is read from a cell at offset 002Ch and is sent to name search subroutine, called for from line 18B.

Name search subroutine inspects the whole environment space and exits with ZF flag set, if the requested name isn't found. But when the search ends successfully, then full address (segment: offset) of that variable's value is written into a memory cell. Offset of this memory cell is calculated by subtraction of 4 from a pointer, stored at offset 0165. Therefore addresses, found in different environments, don't overwrite each other, but rather are stored side-by-side at the end of the PSP, belonging to REASSIGN.COM utility.

If name search subroutine returns with ZF flag cleared, then PSP test subroutine has to "descend" into underlying ("parent") PSP and to continue it's tests there. Pointers in current PSP cells at offsets ES:[0016] and ES:[0010] are regarded as "parent" PSP candidate addresses. With respect to selected "parent" PSP candidate the PSP test subroutine in line 1A3 recursively calls for itself. All checks are repeated in that underlying PSP and in its environment. Recursive descent into underlying PSP may be repeated several times. It comes to end, when it reaches the "bottom" PSP, or when the last PSP is found invalid, or when the requested variable's name isn't found in the last environment.

After termination of PSP test subroutine execution of operations in section 2 continues. There in line 12F the address of variable's value, related to the nearest ("upper") environment, is written into registers ES:DI. This address points just at the first character of variable's value, which defines requested function for REASSIGN.COM utility. This character must be a digit, corresponding to either of ASCII codes 31h, 32h, 33h. If there is some other code, then REASSIGN.COM displays error message and returns control back to command interpreter. But if all checks are successfully passed, then in line 144 a subroutine is called for, which is to execute the requested function. Address of subroutine's entrance point is taken from a list at offset 0167h in section 4.

Execution of function 1 begins from entrance point 0213h in section 8. First a call for INT 2F\AX=4300h (8.03-22) detects whether HIMEM.SYS driver (5.04-01) is loaded or not. If it isn't loaded, then REASSIGN.COM displays no error message, but leaves errorlevel 255 and returns control back to command interpreter. When HIMEM.SYS driver is loaded, then a call for INT 2F\AX=4310h (8.03-23) follows in order to get driver's entrance point address. Returned full address is used in line 22D for a far call to driver's 08h function (A.12-3). It returns several parameters, including size of the largest free XMS-memory block. In lines 236 – 252 a value of errorlevel is set in order to prompt a feasible size of RAM-disk. Commands in lines 256 – 263 convert hexadecimal XMS block size into decimal number. Then in lines 265 – 277 the digits of decimal number are transformed into ASCII codes and are written in place of variable's value in the nearest ("upper") environment. If there is no place enough, REASSIGN.COM displays error message, leaves errorlevel 240 and returns control to command interpreter. If place is sufficient, execution returns from subroutine to final section 3, where a copying subroutine is called for from line 14C. It appends written value with spaces (if needed) and copies it into underlying environments. Thus execution of function 1 terminates.

Execution of function 2 begins from its entrance point 029Ah in section 9. Inputted characters are sequentially accepted and written in place of variable's value in the nearest ("upper") environment. Input by means of INT 16\AH=10h (in line 29C) and display via INT 29 (in line 297) are not subjected to DOS's I/O redirections. Commands in lines 28D – 29B patch the place of preceding character with a space, when user presses the BackSpace key. REASSIGN.COM terminates input cycle after ENTER or ESC keystroke.

After ESC keystroke commands in lines 2CD – 2D0 set flags so that new value is not copied into underlying environments. REASSIGN.COM exits, leaving errorlevel 255 and preserving variable's former value. After ENTER keystroke a return from subroutine and all final operations are performed just as it was described above for function 1. The variable gets a new value, and execution comes to end leaving errorlevel 000.

Execution of function 3 starts from its entrance point 02DCh in section 11. Number of current disk is requested with INT 21\AH=19h, converted into ASCII code of disk's letter-name and written in place of variable's former value. If there is a place for at least one more character, then letter-name is appended with a colon.

As far as current disk certainly exists, errorlevel for function 3 is charged with other mission – number of floppy drives determination. Number of floppy drives must be known to adaptive loading procedures, because in computers having one floppy drive all requests to other floppies are automatically readdressed by MS-DOS to single existing floppy drive A:. Floppy problem is even more complicated because of possibility of floppy drive emulation, which is not reflected in CMOS memory data.

In order to determine actual number of floppy drives the commands in lines 2F1 – 30C appeal to DDT tables (A.03-2) for information about correspondence between logical disks and physical drives. This information enables to count actual disks, ignoring repeated references to the same physical drive. After that execution returns from subroutine to final section 3. Having accomplished function 3, REASSIGN.COM leaves errorlevel value, which helps to define proper disks testing order (example – in 9.09-02).

More detailed explanation of each command is given in commentaries, complementing each line after a semicolon.

It's worth to pay attention to modular structure of REASSIGN.COM utility. Each function is performed by a separate subroutine, independent from other functions. Moreover, there are 3 unfilled positions (16D, 16F, 171) in list of entrance point addresses, so that you may add entrance points for your own subroutines. But before updating REASSIGN.COM, you have to bother about proper typesetting of the current assembler text. It is not as simple as texts in preceding part 9.05. You shouldn't dare to execute at once that file, which will be created by DEBUG.EXE in response to the text you have typed before this text is thoroughly verified.

Some practical recommendations, given in following part 9.07, will help you to detect errors in typesetting, to test executable files and to avoid undesirable consequent complications.

Note 1: if there is any synonymous variable in some underlying environment, then REASSIGN.COM will assign new value to that variable too, and this new value

may be truncated according to length of former one. Repeated usage of one name in different contexts should be avoided.

9.07 Some advices on checking and testing

Ordinary practice of verifying assembler texts includes listing analysis, correction of revealed errors and following step-by-step tracing. Error correction is necessary anyway, but user can't expect considerable help from DEBUG.EXE debugger: it doesn't reveal non-syntactic errors and doesn't provide automatic linking, as more perfect assemblers do.

Despite DEBUG.EXE debugger's obvious drawbacks, in some circumstances it has no suitable alternatives. This happens in uncertainty conditions and also when debugging affects essential functions or data structures of either DOS or BIOS. Though debugger's capabilities are limited, nevertheless these capabilities may be used effectively. Examples of debugger's capabilities usage are shown below in part 9.07.

9.07-01 Listing analysis

Presence of errors should be assumed in any assembler text one has typed. When DEBUG.EXE performs commands, constituting assembler text, it displays listing. Errors, which are found by DEBUG.EXE, are marked in that listing. Since listing scrolls over the screen rapidly, error marks can easily be missed. In order to examine the shown part of listing more attentively, you may suspend execution, pressing the PAUSE key. After that any other keystroke resumes execution, which then may be suspended at desired moments as many times as you will. Manipulations with PAUSE key were sufficient in obsolete computers, but modern computers shift listing too rapidly. Therefore listing may be examined more conveniently later, after it is redirected and written into a file, for example:

```
Debug.exe < Reassign.scr > Listing.txt
```

Stored file LISTING.TXT may be printed or looked through with any viewer or editor program. A part of REASSIGN.COM program's (9.06) listing is shown in fig. 6.

In LISTING.TXT all the errors, detected by DEBUG.EXE, are pointed at from the following line with a word ^Error (preceded by a caret pointer). The caret points at that character in preceding line, which can't be interpreted by DEBUG.EXE. Most often this is sufficient for understanding how the error should be corrected. A caret in fig. 6 points at an error in preceding line: indeed, there is "imp" typed instead of "jmp". Sometimes the caret points at the end of preceding line or at the semicolon sign, where a comment starts. This happens, when some required element in command specification is absent. Which particular element is absent – this can be cleared up from chapter 7, presenting all forms of machine command specifications, acceptable for DEBUG.EXE.

```
0C5B:0134       mov         BL,[DI]     ; 134          variable's value
0C5B:0136       cmp         BL,31       ; 136 Lower limit: function 1
0C5B:0139       jb          0151        ;*139 Exit, if value is less
0C5B:013B       cmp         BL,33       ; 13B Upper limit: function 3
0C5B:013E       ja          0151        ;*13E Exit, if value is greater
0C5B:0140       shl         BL,1        ; 140 multiply by 2
0C5B:0142       mov         BH,00       ; 142 Prepare BX for calculation
0C5B:0144       call        [BX+005]    ;*144 Call functional subroutines
0C5B:0147       jz          0151        ;*148 ZR state - message display
0C5B:0149       jb          015F        ;*14A CY - fail, errorlevel in DH
0C5B:014B            ;************* Section 3: execution conclusion
0C5B:014B               ; 151 - target for 118, 11F, 12A, 139, 13E, 148
0C5B:014B               ; 15F - target for jump from lines 14A, 14F
0C5B:014B       call        01F9        ;*14C Call for copying subroutine
0C5B:014E       jmp         015F        ;*14F Jump to termination
                ^ Error
0C5B:014E       mov         BX,DX       ;=151 DS:DX - message address
0C5B:0150       mov         CX,[BX-02]  ; 153 CX = number of characters
0C5B:0153       mov         BX,0001     ; 156 BX = handle to STDOUT
0C5B:0156       mov         AH,40       ; 159 Call for DOS's data
0C5B:0158       int         21          ; 15B          output function
0C5B:015A       mov         DH,F0       ; 15D Errorlevel = F0h
0C5B:015C       mov         AL,DH       ;=15F Restore errorlevel
0C5B:015E       mov         AH,4C       ; 161 Call for DOS's program
0C5B:0160       int         21          ; 163       termination function
```

Fig. 6

In any case a line with an error is not assembled, all offsets below become shifted, and after a single error all following addressing goes wrong. Registered errors must be corrected before any further actions are taken.

Informative listing enables to reveal even those errors, which are not recognized by DEBUG.EXE. Therefore you have to prepare a sufficiently informative assembler text: correct offset values should be specified in comments, preferably at every line.

Almost each line of assembler texts, presented in parts 9.06, 9.08 and 9.10, includes a comment, where correct offset of corresponding machine command is specified just after semicolon sign. Correct offset should be compared with actual offset, displayed in listing at start of each line. Mismatch of these offsets signifies presence of an error in some preceding line. In fig. 6 compared offsets are different from line 147 and on, therefore an error should be expected in preceding line at offset 144. Indeed, comparison of an assembler command at offset 144 in fig.6 with the same line in original text reveals the error: command "call [BX+005]" has been typed instead of "call [BX+0105]". DEBUG.EXE hasn't registered this error. Similar errors emerge during typing of data and textual messages in assembler texts. In any case offsets mismatch should induce a search for its reason and finally must be corrected.

Addresses inside machine commands also may contain errors, which are not detected by DEBUG.EXE. If a comment in assembler line begins with an asterisk, hence this assembler command includes a target address. It must be equal to actual address of first byte in that machine command or in that data block, which have the same correct address specified in a comment. In order to make visual search for target lines easier, comments to these lines are preceded with equality sign. Besides that, separate lines with comments just

under a header of each section specify positions of commands with target addresses pointing to this section.

The last important offset to be checked is the one marking that empty line, which forces DEBUG.EXE to exit its assembler mode of operation. In assembler text, presented in part 9.06, this empty line is the 8-th from the end. As far as machine codes of ordinary programs are written starting at offset 100h, offset of empty line must be exactly 100h greater, than the whole length of assembled program. If you want to save assembled code in a file , you have to preset the CX register with file length value just 100h bytes less than the actual offset returned in response to the empty line. If you persecute another aim – to detect possible offset shift errors, then you have to compare the returned offset of empty line with file length value, preset in register CX. In particular, in the third line from the end of assembler text, presented in part 9.06, the CX register is preset with hexadecimal number 03E0h. Hence, in listing the offset value 04E0h, returned in response to the empty line, will signify absence of offset shifts up to the last processed assembler command.

9.07-02 Interactive debugging opportunities

When effect of some command or of an interrupt handler needs to be clarified, it is not difficult to type 5 – 7 lines and to force DEBUG.EXE to execute these command lines at once. But long successions of command lines can't be composed so easily. You'll be compelled to prepare long successions of commands as textual command files, which then are to be sent to interpreter. For DEBUG.EXE the only way to accept command files is via input redirection. When input is redirected, DEBUG.EXE doesn't accept commands from keyboard, and all advantages of interactive debugging become lost. This is usually taken for granted as having no alternatives. But this opinion is wrong.

Via input redirection the DEBUG.EXE debugger is ready to accept and to execute those commands, which will cancel input redirection and restore debugger's interactive capabilities. Required succession of commands may look as follows:

```
CS:                         ; Restore JFT contents for
mov word ptr [0018],0101    ;          the program under test
mov          AH,62          ; A query for segment address
int          21             ;           of debugger's own PSP
mov          DS,BX          ; Restore JFT contents
mov word ptr [0018],0101    ;                for the debugger
int          20             ; Return control to debugger
```

The first two command lines restore references to the first SFT table entry in cells of JFT table (note 3 to A.07-1) at offsets CS:0018 and CS:0019. These cells correspond to STDIN and STDOUT channels, and appeals to the first SFT table entry (A.01-4) activate the CON device driver. Hence the performed substitution restores normal interaction with

display and keyboard for commands of the program under test. The same operation may be done by INT 21\AH=46h function (8.02-48), but it isn't applied here, because in this particular case the performed substitution is not enough.

DEBUG.EXE doesn't obey to references in that copy of its JFT, which is formed for the program under test. Changes also must be applied to original JFT – to that one, which is formed by COMMAND.COM interpreter inside debugger's own PSP. Therefore in the 4-th line of presented example a call for INT 21\AH=62h (8.02-73) function returns segment address of that original PSP in BX register. In the 5-th and 6-th lines of the presented example this segment address is used in order to make the same substitution in original debugger's JFT. Thus all preparations are made for restoration of normal interaction between debugger and the user. The last peculiarity is that control is returned to DEBUG.EXE not with RET command (not as in examples in part 9.02), but with a call for INT 20 handler (8.02-01). Unlike the RET command, the INT 20 handler leaves stack pointer in its original state, normal for interactive operation mode.

In order to avoid confusion with commands of the program under test, the proposed commands may be appended to redirected command file in a form of machine codes. These codes can be arbitrary located in any free memory space. Let's assume that memory after offset F00h is available and is certainly free. Then the lines, which should replace ordinary last commands "w" and "q" in command file, will look like this:

```
e F00 2E C7 06 18 00 01 01 B4 62 CD 21 8E DB C7 06 18 00 01 01 CD 20
g=F00
```

If a command file with these two final lines is sent to DEBUG.EXE via input redirection, then all preceding command lines will be executed as usual, but after that interactive operation mode of DEBUG.EXE will be restored (instead of control transfer to COMMAND.COM interpreter). An obvious advantage of a return to interactive operation mode is that there is no more need to determine beforehand the length of a file, which is to be saved. Actual length of assembled code is shown in listing, and required length value may be then (post factum) inputted by the user into CX register from command line. One more advantage is that there are no more restrictions (note 1 to 6.05) on testing of commands, which appeal to STDIN and STDOUT channels. At last, there is no more need to save the processed program in a file after each correction: now it's enough to insert corrections just into original assembler text.

The mentioned advantages are especially important, when a program is composed of several sequentially complemented parts. Sequential debugging promotes early detection of non-syntactic errors and makes emending of their consequences easier. Advantages of sequential modular composition have evinced themselves to full extent during preparation of those programs, which are presented completed in articles 9.06, 9.08, 9.10.

9.07-03 Batch-file control over tests

When after errors correction the last listing reveals no more errors, then it's time to cope with all the rest mass of errors in course of testing.

That executable file, which is produced at the last assembling iteration, is usually too raw to be launched at once. First it should be conveyed to debugger. DEBUG.EXE can accept a file just from command line, as it is shown in introduction article to part 6.05. Nevertheless there are some reasons to prefer debugging arrangement by means of a specially prepared batch file. Batch file enables you to get rid of tedious retyping command lines anew each time you will need to repeat the test. Next, batch file always provides common environment. Besides that, batch file can specify auxiliary functions, making test results more informative.

General composition of test batch file includes preparation part, main test execution part and a final part, where test results are analyzed and displayed. Particular implementation of each part, of course, must reflect specific features of the program under test. Let's consider an example of a test batch file, written especially for testing the 2-nd function of REASSIGN.COM utility (9.06):

```
@echo off
set input=22222
echo Original value of input=%input%
Debug.exe Reassign.com input < Reassign.scr
set E=
set Z=00
set N=
:ErrCycle
for %%Y in (0 1 2 %N%) do if errorlevel %E%%%Y%Z% set E=%E%%%Y
if %Z%"==" goto OUT
if %Z%"==0" set Z=
if %Z%"==00" set Z=0
set N=3 4 5
if not %E%"==2" if not %E%"==25" set N=%N% 6 7 8 9
goto ErrCycle
:OUT
echo Errorlevel is %E%
echo New value of input=%input%
pause
```

The first three lines in proposed batch file represent preparation part, the fourth line – main test execution part. It should be taken into account, that during debugging you may easily lose orientation in successions of processed commands. Therefore a source program listing with comments should be prepared beforehand.

In the 4-th line DEBUG.EXE gets a group of parameters from command line and, besides that, accepts a command file via input redirection. Here the main purpose of command line parameters is their participation in filling PSP for the program under test. In the simplest case file REASSIGN.COM, specified in command line, may be empty at all, but owing to presence of its name in command line this name together with following parameters, if there are any, will be written into dedicated PSP fields (A.07-1). Of course, the file REASSIGN.COM may be not empty, and then its code will be written by debugger into memory starting from address CS:0100h and on. This feature is convenient for preparation to debugging of relatively large executable files.

Input redirection in the 4-th line will force DEBUG.EXE to assemble commands from file REASSIGN.SCR. Translated machine commands will be written into memory starting from arbitrary specified address, either overwriting or complementing that code, which has been initially copied from REASSIGN.COM file. Therefore file REASSIGN.SCR may contain only those parts of assembler text, which need further debugging. The latter doesn't relate to assembler texts, presented in this book: there is no sense to divide complete texts. In any case the assembler text, sent via redirection, must end with those two command lines, proposed in article 9.07-02. These commands will force DEBUG.EXE to escape from redirection into interactive operation mode.

Waiting for commands from keyboard, DEBUG.EXE presents its prompt – blinking underscore – and thus invites the user to act further. It's better to begin from testing program parts (or subroutines) with commands "G" (=Go, 6.05-07) and "P" (=Proceed, 6.05-14). Testing program parts is less difficult, than testing step-by-step. Special attention should be paid to states of flags and registers in critical points – for example, in points of return from subroutines. If some part of program returns wrong results, it should be subjected to step-by-step debugging. Even if some part of program causes hanging, it wouldn't take much time to reset computer and to launch the same test batch file anew. Testing step-by-step enables to cope with the most stubborn bugs.

When debugger's session is to be closed in order to make a correction, then you may enter the "Q" (= Quit) command and then press ENTER. After such termination DEBUG.EXE always leaves zero errorlevel. But sometimes it is important to check errorlevel, returned by the program under test. In the latter case debugger's session should be terminated by a call for DOS's INT 21\AH=4Ch function (8.02-55), just as the program under test must be terminated, when it is executed beyond the debugger's shell.

After termination of debugger's session the commands in final part of test batch file will be executed. Results of REASSIGN.COM program execution are expressed via errorlevel code and via returned value of some environmental variable. In order to catch the returned errorlevel the program under test may be executed inside a shell of command interpreter COMMAND.COM, launched with undocumented /Z parameter (6.04). However, this shell doesn't convey values of environmental variables. Therefore lines 5 – 16 of the proposed test file contain a procedure of errorlevel determination. Then

commands in lines 17 – 18 display returned values of both errorlevel and of that variable, which was to be changed by REASSIGN.COM utility. The last line with PAUSE command prevents the displayed results from being hidden under file manager's panels.

In order to check some other function of the program under test, you have to change parameters in preparation part of test batch file. Sometimes command line parameters in main test execution part need to be changed too. Replacement of parameter's values may be performed via dummy parameters of test batch file. But in presented examples dummy parameters are not employed for simplicity reasons. It is assumed, that all necessary changes can be made just in batch file text by means of an editor program. When batch file is saved with all necessary changes, it is again ready to be used for testing the next function.

Generally, every program must be subjected to three types of test series. Test series of the first type check all functions of the program in normal conditions. Test series of the second type check program's response to probable abnormal conditions: invalid specifications, absence of required parameters, inadmissible data values, etc. Test series of the third type are for those programs, which perform direct manipulations with hardware: such programs must prove their functionality in all relevant hardware configurations. Naturally, particular composition of tests depends on program's mission and on scope of challenges, but principle is the same for all programs, even as small as proposed REASSIGN.COM utility. When your program passes all tests successfully, then your mission concerning this program may be considered completed.

9.08 Let's try to assemble a driver

When DOS is used to explore some computer for the first time, it is desirable to have a fixed letter-name assigned to RAM-disk, prepared for DOS relocation. This can be achieved by means of a driver, which declares a necessary number of nonexistent (dummy) disks so that DOS is forced to present the desired letter-name to RAM-disk.

As far as I know, three attempts to create such driver have been a success. However, freeware versions of such drivers don't hide dummy disks, and the one version which does hide is not a freeware. Therefore one more (4-th) attempt of my own is presented below. It doesn't follow known solutions. It is undertaken especially for making driver's peculiarities more clear and easy to understand. The proposed driver is tiny: 503 bytes total. Let it be named SKIPDSK.SYS.

SKIPDSK.SYS driver is produced by debugger DEBUG.EXE as a result of command sequence execution. This command sequence should be written into command file SKIPDSK.SCR and should be sent to debugger via input redirection:

```
DEBUG.EXE < SKIPDSK.SCR
```

The SKIPDSK.SCR file should be prepared by means of editor program (as described in introduction article to chapter 9) according to the text presented below. Verbal comments may be omitted, but proper offset values in comments should be preserved in order to make further debugging and testing easier.

```
a 0000
      ;************* Section 1: driver's header
         ; 000  Place for next driver's address
dw            FFFF,FFFF
         ; 004  Driver's attributes (A.05-2)
dw            2202
         ; 006  Strategy routine entrance point
dw            0031
         ; 008  Interrupt routine entrance point
dw            006E
         ; 00A  Number of disks, accessed from 058, 113
db            00
         ; 00B  An identifier (7 following bytes)
db            53,6B,69,70,44,73,6B
      ;************* Section 2: data
         ; 012  Request's offset, accessed from 032, 073
         ; 014  Request's segment, accessed from 037
dw            0000,0000
         ; 016  BPB data (A.03-4), referred at 093 - 0C1
db            00,02,FF,01,00,01,40,00,00,22,F0,01,00
db            12,00,01,00,01,00,00,00,00,00,00,00
         ; 02F  CDS, from lines 03D, 04D, 052, 087, 0D8
dw            FEFE
      ;******* Section 3: TSR part of strategy routine
         ; 031 - specified at offset 006
CS:                        ;=031 Strategy routine entrance
mov           [0012],BX    ;*032 Store offset of the request
CS:                        ; 036         data block (A.05-3)
mov           [0014],ES    ;*037 Store its segment
retf                       ; 03B Return far to DOS
      ;*********** Section 4: response to media check
         ; 03C - target for jump from line 084
         ; 05C - target for jump from line 06C
CS:                        ;=03C Is the shift for dummy CDS
cmp byte ptr  [002F],FE    ; 03D      record ready in CS:002Fh?
jnb           008E         ;*042 Return back, if no
mov           AH,52        ; 044 Call for List-of-Lists address
```

```
int          21           ; 046 Now the address is in ES:BX
ES:                       ; 048 Load address of the 1-st CDS
les          BX,[BX+16]   ; 049     record into the ES:BX pair
CS:                       ; 04C Calculate offset of
add          BX,[002F]    ;*04D     the first dummy CDS record
CS:                       ; 051 Mark FFh at 002Fh means that
mov byte ptr [002F],FF    ;*052     media check has been done
CS:                       ; 057 Read the number of
mov          AH,[000A]    ; 058     dummy CDS records (units)
cmp          AH,01        ;=05C Compare the number with 01h
jb           008E         ;*05F Exit, if there are no units
ES:                       ; 061 Else write "disabled disk"
mov word ptr [BX+43],0000 ; 062      attributes into the CDS
add          BX,0058      ; 067 Calculate offset for next CDS
dec          AH           ; 06A Get number of remaining cycles
jmp          005C         ;*06C Repeat the cycle for next CDS
      ;********** Section 5: TSR part of interrupt routine
         ; 06E - specified at offset 008
         ; 08E - target for jumps from 042, 05F, 082, 13C
pushf                     ;=06E Interrupt routine entrance
push         ES           ; 06F Save states
push         BX           ; 070      of registers and flags
push         AX           ; 071
CS:                       ; 072 Load request block address
les          BX,[0012]    ;*073       into ES:BX pair
ES:                       ; 077 Set "unknown media" status
mov word ptr [BX+03],8007 ; 078        to be returned
ES:                       ; 07D Check operation code
cmp byte ptr [BX+02],01   ; 07E       in request data block
ja           008E         ;*082 Jump to exit, if above 01h
jz           003C         ;*084 Go to media check, if 01h
CS:                       ; 086 If below 01h, then is it
cmp byte ptr [002F],FE    ;*087     the 1-st initialization?
jz           00C3         ;*08C Go to initialize, if yes
pop          AX           ;=08E Else restore states
pop          BX           ; 08F        of registers
pop          ES           ; 090
popf                      ; 091 Restore flags
retf                      ; 092 Return far to DOS
      ;********** Section 6: array of BPB offsets for disks
         ; 093 - mentioned in lines 11E, 12A
dw           0016,0016,0016,0016,0016,0016,0016,0016
dw           0016,0016,0016,0016,0016,0016,0016,0016
```

```
dw          0016,0016,0016,0016,0016,0016,0016,0016
      ;********* Section 7: interrupt routine, non-TSR part
         ; 0C3 - target for jump from line 08C
push        DX          ;=0C3 Save states of
push        DS          ; 0C4       registers in stack
push        SI          ; 0C5
cld                     ; 0C6 Clear direction flag
ES:                     ; 0C7 Get in AH the disk number,
mov         AH,[BX+16]  ; 0C8        suggested by DOS
mov         AL,41       ; 0CB Convert the disk number into
add         AL,AH       ; 0CD       disk's letter-name in AL
CS:                     ; 0CF Replace with this letter-name
mov         [01E0],AL   ;*0D0       the former one in message
mov         AL,58       ; 0D3 Calculate in AX the shift for
mul         AH          ; 0D5    the first dummy CDS record
CS:                     ; 0D7 Now shift for the first dummy
mov         [002F],AX   ;*0D8    CDS record is in CS:[002F]
      ;********* Section 8: reading of disk's letter-name
         ; 0E2 - target for jump from line 0E7
         ; 0E9 - target for jump from line 0EE
ES:                     ; 0DB Load in DS:SI a pointer to
lds         SI,[BX+12]  ; 0DC       command-line arguments
mov         DX,013F     ;*0DF Offset of 1-st error message
lodsb                   ;=0E2 Load a character into AL and
cmp         AL,20       ; 0E3    arrange a cycle searching
jb          00F8        ;*0E5        for first space after
ja          00E2        ;*0E7                driver's name
lodsb                   ;=0E9 Load a character into AL and
cmp         AL,20       ; 0EA    arrange a cycle searching
jb          00F8        ;*0EC       for valid letter after
jz          00E9        ;*0EE             the first space
cmp         AL,43       ; 0F0 Is the letter less than C: ?
jb          00F8        ;*0F2 If yes, jump to section 9
cmp         AL,59       ; 0F4 Is the letter less than Y: ?
jbe         0102        ;*0F6 If yes, jump to section 10
      ;********** Section 9: message display part
         ; 0F8 - target for jumps from 0E5, 0EC, 0F2, 10C
push        CS          ;=0F8 Prepare segment address in DS
pop         DS          ; 0F9          for display function
mov         AH,09       ; 0FA Call for a string
int         21          ; 0FC          display function
mov         AL,00       ; 0FE 0 disks to be set after error
jmp         010E        ;*100 Go to prepare return to DOS
```

```
      ;*********** Section 10: calculation of TSR part size
         ; 102 - target for jump from line 0F6
         ; 10E - target for jump from line 100
inc           AL            ;=102 Get RAM-disk's letter-name
mov           DX,01C3       ; 104 Offset of 2-nd error message
CS:                         ; 107 Calculate number of dummy
sub           AL,[01E0]     ;*108     disks to be established
jb            00F8          ;*10C If below 0, display a message
ES:                         ;=10E Write number of disks into
mov           [BX+0D],AL    ; 10F       the request data block
CS:                         ; 112 Duplicate the number
mov           [000A],AL     ; 113        into driver's header
mov           AH,00         ; 116 Calculate offset for a
cmp           AL,00         ; 118    pointer to 1-st byte past
jz            0121          ;*11A       driver's TSR part
shl           AX,1          ; 11C         according to formula
add           AX,0093       ;*11E                (2*AX + 093h)
      ;********* Section 11: filling the request data block
         ; 121 - target for jump from line 11A
ES:                         ;=121 Write the pointer's offset
mov           [BX+0E],AX    ; 122      into request data block
ES:                         ; 125 Write segment address for
mov           [BX+10],CS    ; 126        pointer offset at 0Eh
ES:                         ; 129 Offset of array of BPB offsets
mov word ptr  [BX+12],0093  ;*12A       for each installed disk
ES:                         ; 12F Write segment address for
mov           [BX+14],CS    ; 130       the array of BPB offsets
ES:                         ; 133 Write "happy end"
mov word ptr  [BX+03],0100  ; 134        status word for return
pop           SI            ; 139 Restore states
pop           DS            ; 13A          of registers
pop           DX            ; 13B
jmp           008E          ;*13C Go to return to DOS
      ;*********** Section 12: error messages
         ; 13F - 1-st error message, referred at 0DF
db            0D 0A "SkipDsk: diskletter isn" 27 "t found"
db            20 "or out of range" 0D 0A
db            "Example:" 0A "device=A:\SkipDsk.sys Q:"
db            0D 0A 09 09 09 "Q: - diskletter (C: - Y:)"
db            20 "to be skipped" 0D 0A 0A 24
         ; 1C3 - 2-nd error message, referred at 104
db            0D 0A "SkipDsk: diskletters below" 20
         ; 1E0 - letter-name, accessed from 0D0, 108
```

```
db            "A: are assigned yet" 0D 0A 0A 24
         ; 1F7 - checkpoint, end of driver's code

m 0000 L01F7 0100
n SkipDsk.sys
rBX
0000
rCX
01F7
w
q
```

Unlike ordinary executable files, drivers are loaded starting from offset 0000h, i.e. without reserved space for PSP (A.07-1). Because of this reason start command for assembling drivers must be not "A 100", as for ordinary programs, but "A 0000". If start address is specified otherwise, a muddle occurs in displayed target offsets.

One more specific feature is that DOS drivers have two entrance points, and neither of these coincides with driver's code start address. The first entrance point, related to strategy routine, is used to accept a request and to initiate its execution by the corresponding device (a printer, disk drive, etc.). The second entrance point, related to interrupt routine, is used to gather the result and to form a response to the accepted request. Time in between is spent in parallel by DOS and by physical devices, each performing its own job. Entrance points are announced in driver's header: for strategy routine at offset 0006h, for interrupt routine at offset 0008h. These and other elements of driver's header are shown in table A.05-1.

Resident part of strategy routine in SKIPDSK.SYS driver is represented by section 3 of assembler text. It is simple and just writes segment address and offset of request data block (A.05-3) into a prepared memory cell. Strategy routine of SKIPDSK.SYS driver has no non-resident part.

All request-specific operations are performed by interrupt routine, which starts in section 5 of assembler text. Section 5 begins with saving states of flags and registers – this is also a specific responsibility of any driver. Then a code of requested operation is checked. As far as SKIPDSK.SYS driver "serves" dummy disks only, requests with operation codes greater than 01h must always cause a return to DOS with "invalid media" status byte. But requests for operations 00h and 01h invoke execution of commands, presented in sections 7 – 11 or in section 4.0.

The first request to driver is always for initialization with operation code 00h. Since initialization is requested only once, the corresponding sections 7 – 11 are beyond driver's resident part. Commands in section 7 prepare data for further use, commands in section 8 read letter-name of the disk to be skipped from command line, commands in section 9 are

always ready to display an error message. Templates for error messages are prepared yet in section 12. Commands in section 10 calculate number of dummy disks and actual length of driver's resident part. Commands in section 11 fill request block with data, which should be returned to DOS. According to these data DOS corrects its DPB (A.03-1) and CDS (A.03-3) tables so that disk's letter-names up to the one specified appear busy. When later a RAM-disk driver will be initiated, it will be given the next free letter-name, which is thus predetermined by SKIPDSK.SYS driver.

While assigning a letter-name to RAM-disk, DOS must consider the declared dummy disks valid. But afterwards their valid status confines further letter-names assignment. A mixture of actual disks and dummies causes confusion. Therefore the next task for SKIPDSK.SYS driver is to disable dummy disks. For this purpose SKIPDSK.SYS must be activated once more.

For the second time SKIPDSK.SYS is activated by a request for "media check" operation (code 01h), because it precedes all other requests, addressed to removable disks. Besides that, DOS automatically requests media check operation, when interpretation of CONFIG.SYS file completes. In response to media check request the SKIPDSK.SYS driver executes commands in its resident section 4.0. These commands determine address of the first dummy CDS record, and then perform a cycle of writing an invalid attribute word 0000h into all dummy CDS records. After that all "dummy" disks become hidden and are considered invalid. Their former letter-names may be assigned to CD-ROMs and to network drives.

When former letter-name of a dummy disk is given yet to another driver, then writing of attribute word 0000h into the same CDS record must be prohibited. Therefore one of commands in 4-th section writes a mark FFh into a memory cell at offset 02Fh. Presence of this mark is checked each time when interrupt routine of SKIPDSK.SYS driver is called for. Owing to this check repeated requests to SKIPDSK.SYS driver can't impair proper access to other disks, which have got former letter-names of dummy disks.

More detailed explanation of particular operations can be found in comments to lines of assembler text file SKIPDSK.SCR.

As in most assembler texts, it is important to note an empty line, the 9-th from the end. It forces DEBUG.EXE to exit assembler mode of operation and therefore must be present. The next line with "M" command (6.05-11) moves the whole assembled code 100h bytes further, thus making PSP area free. This is necessary in order to specify driver's name, which is to be written just into PSP area and is announced in the following line by "N" command (6.05-12). Concluding lines of SKIPDSK.SCR file prepare length of driver's file in BX:CX registers and write the SKIPDSK.SYS driver to current disk.

Produced file SKIPDSK.SYS shouldn't be considered valid until its listing is thoroughly checked, as it is described in article 9.07-01. The last actual offset in listing is given in response to the mentioned empty line – the 9-th from the end of text. As far as

assembling started from address CS:0000h, this last offset must be 01F7h, exactly the same as total driver's length, specified in the 8-th and 3-rd lines from end of text. If listing reveals no errors, there is a chance for SKIPDSK.SYS to pass tests successfully.

While arranging tests it should be taken into account, that for reasons of simplicity and compact size the capabilities of SKIPDSK.SYS driver are limited. It can't affect allotment of disk's letter-names, which are already assigned. It can't accept letter-name specification, preceded with a slash. It can't tolerate tabulation code (09h) instead of a space (20h) in its command line. Finally, it can't work under MS-DOS versions earlier than 4.0.

In practice a necessity may arise to reassign "dummy" disk's letter-names before interpretation of CONFIG.SYS file is finished. For this purpose a media check request to SKIPDSK.SYS driver should be provoked earlier, for example, by the following commands:

```
devicehigh=\DOS\DRV\SkipDsk.sys Q:
devicehigh=\DOS\DRV\Ramdrive.sys 2400 /E
install=\Command.com nul /low /f /c vol Q:
```

In this example the letter-name Q: will be assigned to the last dummy disk, and RAM-disk will be given the next letter-name R:. In the last line an explicit appeal to "dummy" disk Q: provokes media check request, and since that moment all "dummy" disk's letter-names become free for reassignment by software, which may be loaded afterwards by INSTALL= or INSTALLHIGH= commands. One more similar example of SKIPDSK.SYS driver usage is presented in article 9.09-01.

9.09 Exploratory configuration files

Versions of configuration files CONFIG.SYS and AUTOEXEC.BAT, suggested in this article, implement several loading modes. Besides MS-DOS7 loading in an ordinary way, it can be relocated to RAM-disk or to a HDD, and also can be loaded without drivers, as it should be done for RAM testing and for reprogramming BIOS memory chips.

Main advantage of proposed configuration files is that a choice of the most suitable loading mode is not necessarily "blind", it can be based on results of preliminary tests. If you prefer MS-DOS7 relocation onto a RAM-disk, it's size choice will be based on results of XMS-memory investigation. If you prefer MS-DOS7 relocation onto a HDD, you will be informed in advance about which disks are available, about total size and usage percent for all writable disks.

When operating system loading procedures are not completed, then hardware tests can't be performed just as though operating system were loaded yet. Therefore here disk's tests can't be arranged exactly as in file DISK.BAT (9.03-02). Because of the same reason standard DOS's means are not enough, some other drivers and utilities have to be employed. On the other hand, during OS loading some simplifying assumptions can be

adopted. It is assumed, that current disk is the one used to load MS-DOS7, that directory structure on this disk is known, and it is known also, which loading operations are not performed yet at each stage. Due to the last assumption, in particular, there is no need to identify RAM-disks and CD/DVD-ROM drives.

Suggested configuration files are most suitable for loading MS-DOS7 on AT-compatible computers with unknown hardware configuration. Loading adaptation capabilities are especially essential for repairing services.

9.09-01 Multi-alternative CONFIG.SYS file

This version of CONFIG.SYS file presents 5 loading alternatives listed in [menu] section. Several of these alternatives include operations, which can't be performed by standard DOS's means. In particular, an opportunity to adapt RAM-disk's size is implemented by a freeware version 2.42 of TDSK.EXE driver (5.05-02). Suitable replacements for this driver are not known. Besides that, the SKIPDSK.SYS driver (9.08) is used, which you may make yourself. Potentially JDRIVE.SYS driver from JAMSOFT can be used instead, but it is not free.

Data and parameters for other employed drivers can be found in chapter 5. These drivers either are included in WINDOWS-95/98 release or have close equivalents, therefore their choice is not critical. Naturally, each replacement implies command line parameters correction according to driver's requirements. Text of proposed CONFIG.SYS file is presented below.

```
[menu]
numlock off
menuitem=L047, User-configurable real mode
menuitem=L048, User-configurable V86 mode
menuitem=L029, Quick boot in real mode
menuitem=L031, Quick boot in V86 mode
menuitem=L104, Real mode without drivers
menudefault=L029,20

[L047]
device=\DOS\DRV\Himem.sys /v
device=\DOS\DRV\Umbpci.sys
include=L104
country=007,866,\DOS\DRV\Country.sys
devicehigh=\DOS\DRV\Dblbuff.sys
devicehigh=\DOS\DRV\Ifshlp.sys
devicehigh=\DOS\DRV\Setver.exe
devicehigh=\DOS\DRV\Dvs.sys /D:CD001
devicehigh=\DOS\DRV\Skipdsk.sys Q:
```

```
device=\DOS\DRV\Tdsk.exe 0
install=\Command.com nul /low /f /c vol Q:
installhigh=\DOS\DRV\Shsucdx.com /D:?CD001 /L:N /~+ /R /Q
installhigh=\DOS\DRV\Ctmouse.exe
installhigh=\DOS\DRV\Keyrus.exe

[L048]
device=\DOS\DRV\Himem.sys /v
device=\DOS\DRV\Jemm386.exe X=TEST noems verbose
include=L104
country=007,866,\DOS\DRV\Country.sys
devicehigh=\DOS\DRV\Dblbuff.sys
devicehigh=\DOS\DRV\Ifshlp.sys
devicehigh=\DOS\DRV\Setver.exe
devicehigh=\DOS\DRV\Dvs.sys /D:CD001
devicehigh=\DOS\DRV\Skipdsk.sys Q:
device=\DOS\DRV\Tdsk.exe 0
install=\Command.com nul /low /f /c vol Q:
installhigh=\DOS\DRV\Shsucdx.com /D:?CD001 /L:N /~+ /R /Q
installhigh=\DOS\DRV\Ctmouse.exe
installhigh=\DOS\DRV\Keyrus.exe

[L029]
include=L047

[L031]
include=L048

[L104]
accdate C- D- E- F-
dos=high,umb,noauto
buffershigh=30,0
fileshigh=30
lastdrivehigh=Z
fcbshigh=1,0
stackshigh=9,256

[common]
shell=\Command.com \ /E:2016 /L:511 /U:255 /p
```

Text of CONFIG.SYS file starts with [menu] section. There are 5 items, each having an explicit header and a name code (L029 – L104). Choice of an item with some name code initiates interpretation of commands in synonymous section of CONFIG.SYS file.

Besides this, name code of selected item is automatically assigned as a value to CONFIG environmental variable; this value is used later in order to select corresponding part of other configuration file – AUTOEXEC.BAT. For this purpose items in menu are named after label marks in AUTOEXEC.BAT file (9.09-02).

Choice of menu items L029, L031, L047, L048 causes loading of almost the same sets of drivers, defined in sections [L047] and [L048]. The only difference between these sections is in their line 2: driver UMBPCI.SYS (5.04-04) gives access to upper memory in CPU's real mode, whereas driver JEMM386.SYS (note 4 to 5.04-02) gives similar access in V86 mode. Both sections L047 and L048 include loading of XMS-memory driver, driver for access to CD/DVD-ROMs, "mouse" driver and some others. Note that TDSK.EXE driver for arranging a RAM-disk, unlike most others, is loaded into conventional memory, and that RAM-disk's size is not specified here. If arranging of a RAM-disk will be considered expedient, its size should be specified later, during interpretation of AUTOEXEC.BAT file (9.09-02).

Sufficient differences between configurations L029 – L048 also evince themselves later, during interpretation of AUTOEXEC.BAT file, except one configuration, defined by menu item L104. Corresponding section [L104] includes DOS settings only, without drivers and TSRs. The last section in CONFIG.SYS file is section [common], which is executed in any case. It loads command interpreter COMMAND.COM.

All paths in lines of CONFIG.SYS file don't include disk's letter-name and therefore are suitable for loading from any disk. A care should be taken about presence of all mentioned drivers and other files in prescribed directories. Path changes are not prohibited, but affect conditions of AUTOEXEC.BAT file execution (9.09-02) and therefore must be made consistent.

Proposed CONFIG.SYS file should be typed as non-formatted text and saved in the root directory of bootable removable media – either a diskette or a flash card, dedicated for loading MS-DOS7 onto AT-compatible computers with unknown hardware configuration.

9.09-02 AUTOEXEC.BAT file for adaptive loading

This version of AUTOEXEC.BAT file implements exploratory and adaptive capabilities of MS-DOS7. For this purpose the REASSIGN.COM utility is used, which may be assembled by you yourself (9.06). Full-functional replacement for REASSIGN.COM utility is not known.

In order to make orientation in AUTOEXEC.BAT file more easy, label marks include corresponding line numbers; for example, label L029 denotes 29-th line. Besides that, each tenth line is a comment announcing line's number.

Local variables in AUTOEXEC.BAT are named V0 – V8. One of them – V8 – has no permanent mission. Dedicated missions of the rest local variables are:

V0 – list of disks, selected for being tested
V1 – size of the largest free XMS-memory block
V2 – recommended size of RAM-disk
V3 – candidate target disk for relocation
V4 – current disk, used to boot the PC
V5 – writeability status of the current disk
V6 – list of inaccessible or non-writable disks
V7 – list of writable disks

Text of AUTOEXEC.BAT file version for adaptive loading is presented below.

```
@echo off
if %1"==J" if not %2"==" goto L%2
prompt $p$g
path ;
path=\DOS\MS7;\DOS\OTH
set V0=C D E F
set V4=33
Reassign.com V4
if errorlevel 2 set V0=%V0% B
       rem  ================== Line 10 ==================
if errorlevel 1 set V0=%V0% A
set comspec=%V4%\Command.com
set V1=11111111
set V2=300
Reassign.com V1
if errorlevel 1 if not errorlevel 2 set V2=5600
if errorlevel 2 if not errorlevel 3 set V2=16000
if errorlevel 3 if not errorlevel 100 set V2=32000
if errorlevel 100 for %%Z in (1 2) do set V%%Z=
       rem  ================== Line 20 ==================
if errorlevel 100 if not %config%"==L104" set config=L047
if not %config%"==L104" goto %config%
for %%Z in (3 5 6 7) do set V%%Z=
for %%Z in (%V0%) do call \Autoexec.bat J 117 %%Z: Q
if %V5%"==F" echo Warning: current disk %V4% is not writable!
if not %V7%"==" echo         Writable disk(s): %V7%
if %V7%"==" echo Warning: no writable disks have been found!
goto L104
:L029
       rem  ================== Line 30 ==================
:L031
set V3=
set ramdrive=?:
```

```
%V4%\DOS\DRV\Tdsk.exe %V2% /E /M /F:2
if not %ramdrive%"==?:" if not %ramdrive%"==::" set V3=%ramdrive%
if not %V3%"==" if not %V2%==300 goto L086
if not %V3%"==" if %V2%"==300" goto L104
set V1=
echo RAM-disk arranging procedure has failed!
       rem  ================== Line 40 ==================
goto L047
:L042
for %%Z in (1 1 1 1 1 1 1) do echo=
ctty nul
call %V4%\Autoexec.bat J 117 %V3% S
ctty con
:L047
:L048
for %%Z in (1 1 1 1 1 1 1) do echo=
       rem  ================== Line 50 ==================
for %%Z in (5 6 7) do set V%%Z=
ctty nul
for %%Z in (%V0%) do call %V4%\Autoexec.bat J 117 %%Z: V
ctty con
if %ramdrive%"==" set ramdrive=R:
if not %V7%"==" echo Writable disk(s): %V7%
if %V7%"==" echo Writable disks have not been found!
if not %V1%"==" echo Available XMS-memory is %V1% kb
if %V1%"==" echo XMS-memory is unavailable
       rem  ================== Line 60 ==================
echo  Select one of the shown keys and then press ENTER:
if not %V7%"==" echo      %V7% - set DOS onto the chosen disk
if not %V6%"==" echo      %V6% - retry inaccessible disk(s)
if not %V1%"==" echo      %ramdrive% - set a RAM-disk for DOS
echo      ESC - leave DOS on %V4%, no relocation
echo=
:L067
set V3=2
Reassign.com V3
       rem  ================== Line 70 ==================
if not errorlevel 128 if %V3%"==" goto L067
if errorlevel 128 set V3=
if errorlevel 128 goto L104
set V8=%path%
path %V3%
set V3=%path%:
```

```
path=%V8%
if %V3%"==%ramdrive%" if not %V1%"==" goto L031
set V8=N
       rem  ================== Line 80 ==================
for %%Z in (%V6%) do if %V3%==%%Z set V8=F
if %V8%==F goto L042
for %%Z in (%V7%) do if %V3%==%%Z set V8=T
if %V8%==T if %V3%"==%V4%" goto L104
if not %V8%==T goto L067
:L086
ctty nul
for %%Z in (. OTH MS7 VC4) do Attrib -h -r -s %V3%\DOS\%%Z\*.*
for %%Z in (. ..\TEMP OTH MS7 VC4) do md %V3%\DOS\%%Z
       rem  ================== Line 90 ==================
for %%Z in (Autoexec.bat Command.com) do copy /B \%%Z %V3%\DOS /Y
ctty con
if not exist %V3%\DOS\Command.com set V3=
if %V3%"==" echo Relocation attempt has failed!
if %V3%"==" goto L104
echo Copying the following files to disk %V3%
for %%Z in (OTH MS7 VC4) do copy /B \DOS\%%Z\*.* %V3%\DOS\%%Z /Y
set comspec=%V3%\DOS\Command.com
%V3%
       rem  ================== Line 100 ==================
for %%Z in (OTH MS7 VC4) do \DOS\MS7\Attrib +r \DOS\%%Z\*.ini > nul
set V4=%V3%
%V3%\DOS\Autoexec.bat J 104
:L104
if %V3%"==" for %%Z in (%V7%) do set V3=%%Z
if not %V3%"==" if not exist %V3%\Temp\nul md %V3%\Temp
if not %V3%"==" if exist %V3%\Temp\nul set Temp=%V3%\TEMP
if %Temp%"==" echo Warning: the TEMP variable is not defined!
set dircmd= /A /O:GNE /P
       rem  ================== Line 110 ==================
path=%V4%\DOS\OTH;%V4%\DOS\MS7;%V4%\DOS\VC4;%V4%\;%V4%\DOS
%V4%\DOS\OTH\Blue.com
if not %config%==L104 set VC=%V4%\DOS\VC4
for %%Z in (0 1 2 3 4 5 6 7 8) do set V%%Z=
if not %config%==L104 %VC%\Vc.com /TSR /no2E /noswap
goto END
:L117
%comspec% nul /f /c if exist %3\nul cd \DOS
if not exist ..\NUL if %4"==Q" goto END
```

```
       rem  ================= Line 120 ==================
if not exist ..\NUL goto L152
%comspec% nul /f /c call %0 J 141 %3 %4
%V4%
if not exist ..\NUL if %3"==%V4%" set V5=
if not exist ..\NUL if %4"==S" goto END
if not exist ..\NUL if not %V7%"==" set V7=%3 %V7%
if not exist ..\NUL if %V7%"==" set V7=%3
if not exist ..\NUL goto END
cd \
       rem  ================= Line 130 ==================
if not %4"==Q" echo Disk %3 is not writable! > con
if %3"==%V4%" set V5=F
if not %4"==S" if not %V6%"==" set V6=%V6% %3
if not %4"==S" if %V6%"==" set V6=%3
if not %4"==S" goto END
echo If disk %3 is write-protected, close protection > con
echo hole in its cartridge. Press any key to continue > con
pause < con > nul
goto END
       rem  ================= Line 140 ==================
:L141
set TEMP=
%3
ver | shift
if not %4"==" goto END
cd %V4%\
if not %3"==V" goto END
echo Disk %2 is writable: > con
dir /a:ARD /-p /v %2\ | %V4%\DOS\MS7\Find.exe "otal d" > con
       rem  ================= Line 150 ==================
goto END
:L152
cd \DOS
set V8=if errorlevel 15 if not errorlevel 16 cd \
if %4"==S" set V8=if errorlevel 20 cd \
%comspec% /f /c for %%Z in ("Label.exe %3trial" "%V8%") do %%Z
if not exist ..\nul if not %4"==S" goto END
if not %4"==S" if not %V6%"==" set V6=%V6% %3
if not %4"==S" if %V6%"==" set V6=%3
       rem  ================= Line 160 ==================
set V8=Disk %3 either is unformatted or has an improper format
if exist ..\nul set V8=Drive %3 probably has no media inside
```

```
if %4"==V" set V8=Disk %3 is either unformatted or not inserted
echo %V8% > con
cd \
if not %4"==S" goto END
echo Insert proper media and press any key to continue > con
pause < con > nul
:END
```

Specific features of this AUTOEXEC.BAT file version begin with a conditional jump in 2-nd line, which enables to execute recursive calls for internal subroutines. But jump conditions are not met, when AUTOEXEC.BAT is interpreted for the first time. Then in lines 3 – 5 values are assigned to environmental variables PROMPT and PATH. Value of PATH variable is not final; it will be replaced by final value after DOS relocation.

The REASSIGN.COM utility is called for the first time in line 8 in order to determine letter-name of the current disk. Errorlevel value, returned by REASSIGN.COM utility, helps to compile a list of disks, selected for being tested. The next time REASSIGN.COM utility is called for in line 15 in order to ascertain availability of XMS-memory and determine size of the largest XMS-block. This time the returned errorlevel value prompts recommended size of RAM-disk, which may be arranged later. If XMS-memory is found unavailable, then in 21-st line a value of CONFIG variable will be altered so that certainly unsuccessful attempts to arrange a RAM-disk will be prevented.

An important jump is performed by GOTO command in 22-nd line: it makes further processing consistent with menu item user's choice, defined by value of CONFIG environmental variable. If quick loading has been chosen, then jump leads to labels L029 or L031. There in 34-th line the TDSK.EXE driver arranges a RAM-disk of prescribed size and assigns the letter-name of RAM-disk to environmental variable RAMDRIVE. It's just the time to remind that RAMDRIVE variable is defined by the only version 2.42 of TDSK.EXE driver. If earlier versions of that driver or other similar RAM-disk drivers are used (BITDISK.EXE, SRDISK.EXE, XMSDSK.EXE), then fixed value R: should be assigned to RAMDRIVE variable by SET command (3.26). Naturally, opportunities of RAM-disk letter-name adaptation will be lost.

When the prescribed size of RAM-disk is not large enough, then a jump in 37-th line leads to termination part without DOS relocation, and RAM-disk will be used exclusively as a place for temporary files. But normally size of RAM-disk is enough to relocate DOS, and then a jump in 36-th line leads to label L086 – to DOS relocation part of AUTOEXEC.BAT file.

Relocation part of AUTOEXEC.BAT file is composed so that it can be used for DOS relocation to RAM-disk and to any non-empty physical disk as well. Therefore relocation

procedure begins with removing file's attributes in target directories, because otherwise standard DOS' utilities can't overwrite synonymous files there and even can't determine their presence. Operation in line 89 creates a typical structure of target directories, if it didn't exist before. Then files AUTOEXEC.BAT and COMMAND.COM are copied into directory %V3%\DOS. If copying is a success, then in line 97 all the rest files are copied to their target directories, and new values are assigned to COMSPEC and to V4 environmental variables. Command in line 103 transfers control to a copy of AUTOEXEC.BAT file in target directory %V3%\DOS so that a return to original AUTOEXEC.BAT file wouldn't happen, and execution of the copy starts from its 105-th line.

If the user originally has chosen a user-configurable loading alternative, then a jump from 22-nd line leads to labels L047 or L048, where disks exploration procedure starts. Disks exploration is performed by subroutine L117, which is a part of the same AUTOEXEC.BAT file. It is called recursively from cycle FOR in line 53, separately for each disk under test. Subroutine L117 undertakes first attempt to access the submitted disk in line 118 in order to check whether it is readable. If disk is not readable, then a jump to label L152 follows, where the next test in 156-th line is to discriminate between nonexistent drives and those having no formatted media inside. Nonexistent drives are ignored, but all others are added to list of inaccessible disks, represented by value of V6 environmental variable. As far as status of these disks may be changed, their testing may be repeated.

If disk proves to be readable, then the next examination is writeability test, performed by subroutine L141. It is called from line 122 by a separate module of command interpreter COMMAND.COM. The decisive operation is in line 144: there intermediate redirection implies creation of a temporary file on the submitted disk. If this temporary file can't be created, then the SHIFT command in the same line wouldn't be executed, dummy parameters wouldn't be shifted, subroutine L141 will exit in line 145, and commands in lines 133 – 134 will add letter-name of this non-writable disk to list of inaccessible disks. If redirection in line 144 can create a temporary file, it will be deleted automatically just afterwards, but the SHIFT command will be executed, the ECHO command in line 148 will display a confirming message, and the DIR command in line 149 will display usage of disk's space. Then in line 151 subroutine L141 terminates, and letter-name of writable disk in lines 126 – 127 is added to list of accessible disks, represented by value of environmental variable V7.

Disk exploration subroutine L117 terminates either at line 128 or at line 135 – it depends on test result. In both cases control is returned to cycle FOR in line 53. Then subroutine L117 will be called again and again until a list of disk's letter-names, represented by value of variable V0, comes to end. When all disks are tested, the results together with suggested alternatives are displayed to the user by commands in lines 56 –

65. The user is offered to choose the most suitable alternative. User's answer is accepted by REASSIGN.COM utility in line 69. As far as user's response may be expressed by both upper-case and lower-case letters, commands in lines 74 – 77 turn any accepted letter to upper case, and then conditional commands in lines 78 – 85 fulfill user's will.

A check in line 78 is to clear up whether the returned letter is the letter-name of RAM-disk. If yes, then a jump to label L031 is performed, and all the following events repeat the scenario of quick loading onto a RAM-disk just as it was described above. In case of any other choice cycle FOR in line 81 searches for the returned letter through list of inaccessible disks. If the letter is found, then a jump to L042 follows. There in line 45 the L117 exploratory subroutine is called in order to test the selected disk more thoroughly. Thorough mode of investigation, defined by the fourth parameter "S", implies other test criteria and more detailed prompts about the ways to regain disk's accessibility. Subroutine makes a pause, allowing to change removable disk or to close write-protection hole in its cartridge. After that in line 53 a normal call follows for the same subroutine L117 in order to update lists of inaccessible and writable disks, and then the user is offered to make his choice once more.

User's choice of DOS relocation onto a writable disk is registered by cycle FOR in 83-rd line. In this case execution proceeds through label L086 to DOS relocation part of AUTOEXEC.BAT file. Relocation procedure is performed just as it is was described above for RAM-disk. It is important to note, that DOS relocation procedure doesn't make the selected disk bootable, it only allows to continue current DOS session after removing bootable media from that storage device, which was used to boot the PC. As far as DOS relocation procedure doesn't write files in the root directory, original bootability of the selected disk also can't be affected.

The disk used to boot the PC also may be writable, and it's letter-name can be included in the list, represented by value of V7 variable. But DOS relocation onto this disk is senseless and should be avoided. This is why a check in line 84 intercepts such user's choice and directs further execution to label L104. Thus DOS relocation is bypassed. As far as bootable disk is writable, it also will be used as a place for temporary files. However, ordinary 1.44 Mb diskettes have no place enough for temporary files; any other writable disk should be preferred. Therefore the user is given one more chance to bypass DOS relocation: the ESC keystroke forces a jump from line 73 to the same target label L104, but in this case an attempt will be undertaken to find a place for temporary files on some other suitable disk.

The last item in boot menu (9.09-01) is loading without drivers and TSRs. User's choice of this alternative also leads to the same label L104 via a jump from line 28. But before that in line 24 the exploratory subroutine L117 is called in quiet mode, defined by its fourth parameter Q. Detailed exploration is skipped, intermediate messages are not

displayed. In this case the only purpose of exploration is data preparation for warning messages, displayed in lines 25 – 27.

Label L104 denotes final part of AUTOEXEC.BAT file, where all loading alternatives converge. Cycle FOR in line 105 appoints a writable disk for temporary files, if this appointment was not made yet. Operations in lines 106 – 114 prepare final values for TEMP, DIRCMD and PATH environmental variables, and delete local variables V0 – V8. Unless special loading alternative L104 is chosen, the Volcov Commander file manager is launched. The last operation in AUTOEXEC.BAT file is an unconditional jump to final label END.

This version of AUTOEXEC.BAT is to be stored in the root directory of a removable bootable media. Those files, which are called from lines of AUTOEXEC.BAT file, must be present in specified directories of the same media. It is implied, that command interpreter COMMAND.COM is present in the root directory, files ATTRIB.EXE, FIND.EXE and LABEL.EXE are present in \DOS\MS7 directory, utilities REASSIGN.COM and BLUE.COM – in \DOS\OTH directory, driver TDSK.EXE (version 2.42) – in \DOS\DRV directory, files VC.COM and VC.OVL – in \DOS\VC4 directory. Naturally, presence of other files in these directories is allowed. If you intend to implement other file's allocation or other directory structure, then all affected path specifications should be corrected accordingly.

9.10 Experiments with linear addressing

It is often thought, that modern 32-bit CPUs in real mode can't address to memory space beyond 1088 kb. Though this opinion is not denied explicitly in official data, nevertheless it is wrong. Since early 1990-ties several reports have proclaimed usage of undocumented CPUs features for access to extended memory. Tomas Roden is believed to be author of the idea.

Analysis of HIMEM.SYS (5.04-01) driver's code evinces presence of all necessary elements for providing access to extended memory in real mode. May be, this is just what HIMEM.SYS does, but no official confirmation for that has been published. Despite considerable age of this topic, idea of linear 32-bit addressing in real mode is still a matter of rumors.

Now there is no need to prove the possibility of linear addressing in real mode, but there is a need to demonstrate effective implementation of the idea. Therefore part 9.10 of this book presents texts of two tiny utilities: GS_LIMIT.COM utility takes off segment boundary protection from GS register, and GS_DUMP.COM displays a dump of memory area, pointed at by a given 32-bit linear address. Effect of presented utilities is shown in fig.7: that memory area, which has just been inaccessible beyond segment limits, becomes accessible after a call for GS_LIMIT.COM utility.

```
D:\MSDOSTXT\TRIAL>GS_dump.com 10FE0

 GS= 0010
 00010FE0   F1 32 1E 33 F3 32 F4 32 F5 32 F6 32 F7 32 F8 32  ±2.3≤2⌠2J2÷2≈2°2
 00010FF0   F9 32 FA 32 FB 32 FC 32 FD 32 FE 32 FF 32 00 33  ·2·2√2ⁿ2²2■2 2.3
 00011000       - above GS limit

D:\MSDOSTXT\TRIAL>GS_limit.com off

       GS segment limit protection is turned OFF

D:\MSDOSTXT\TRIAL>GS_dump.com 10FE0

 GS= 0008
 00010FE0   F1 32 1E 33 F3 32 F4 32 F5 32 F6 32 F7 32 F8 32  ±2.3≤2⌠2J2÷2≈2°2
 00010FF0   F9 32 FA 32 FB 32 FC 32 FD 32 FE 32 FF 32 00 33  ·2·2√2ⁿ2²2■2 2.3
 00011000   30 04 90 0C 11 04 32 00 00 00 01 00 00 00 00 00  0.P...2.........
 00011010   00 EB 0E 69 6F 6E 61 6C 56 43 56 49 45 57 20 20  .ж.ionalVCVIEW..
 00011020   45 58 54 20 00 00 00 00 00 00 00 00 00 00 9C 79  EXT...........by
 00011030   62 33 82 03 F7 02 00 00 4D 41 43 38 36 36 20 20  b3B.≈...MAC866..
 00011040   54 42 4C 20 00 00 00 00 00 00 00 00 00 00 01 00  TBL.............
 00011050   C2 28 83 03 10 01 00 00 00 00 00 00 00 00 00 00  ┬(Г.............

D:\MSDOSTXT\TRIAL>
```

Fig. 7

Those who dare to implement the presented examples will get a unique chance to look into all the corners and foldovers of 32-bit address space.

9.10-01 Segment protection switching on/off

As calculated linear address goes beyond segment boundaries, CPU's segment protection becomes actuated and ruins all hopes to get any benefit from address size override prefix (7.02-07) in real mode. However, the problem is not so hopeless as it seems at a first glance. All modern processors, starting from 80386 model and on, are able to change segment size. When segment size is made equal to highest address space limit, then any calculated segment address will be found allowable, and segment protection in fact becomes turned off.

Is it good or bad to turn off segment protection? On one hand, this will cause total loss of stability in multi-tasking operating systems. On the other hand, this will open new opportunities for service and diagnostic programs in DOS operating environment. As any effective instrument, turning off segment protection may be both beneficial and dangerous. Therefore CPU designers have been very cautious about control over segment protection. An opportunity to set arbitrary segment size has been given to protected mode software at the highest (zero) privilege level only.

When in protected mode at the highest privilege level the operating system core is active yet, then all other programs have no chances to get access to critical CPU's settings. But chances are, while CPU operates in real mode: any real mode program may switch

CPU into protected mode, set maximum segment size in a "shadow" register, and then switch CPU back into real mode. After that linear 32-bit addressing with respect to affected segment register will be available without a risk to cause actuation of segment protection. This is, in short, the main idea, implemented by the proposed utility.

For the role of affected segment register the GS segment register is chosen, because its usage by real mode programs is not common. Besides that, properly composed programs don't violate segment limits intentionally. The author has tested many ordinary DOS programs, not aimed at GS segment status determination. All such programs have behaved identically both with and without GS segment limit protection.

Because of GS segment register choice the suggested utility has been named GS_LIMIT.COM. It differs from its known counterparts in that it is able to perform its mission not under "bare" DOS only, but also in cooperation with HIMEM.SYS driver (5.04-01). Second difference is that GS_LIMIT.COM is able to turn segment protection both OFF and ON. In order to turn protection OFF you have to specify a command

```
GS_limit  off
```

Restoration of standard 64-kb limit for GS segment is initiated by command

```
GS_limit  on
```

If neither of the shown parameters (OFF or ON) is specified, the GS_LIMIT.COM utility displays a short help.

GS_LIMIT.COM utility (662 bytes long) is produced by debugger DEBUG.EXE as a result of command sequence execution. This command sequence should be written into command file GS_LIMIT.SCR by means of editor program (as described in introduction article to chapter 9). Verbal comments may be omitted. Special attention should be paid to empty line – 8-th from the end. It must be there, because empty line forces DEBUG.EXE to exit assembler mode (7.01-04). Then command file GS_LIMIT.SCR should be sent to debugger via input redirection:

```
Debug.exe < GS_limit.scr
```

Command file GS_LIMIT.SCR has to contain the following lines:

```
a 100
      ;***************** GS_limit.com ******************
      ;********** Section 1: memory and parameters check
         ; 110 - target for jump from line 104
cmp           SP,2010       ; 100 Less than 8 kb allocated?
jbe           0110          ;*104 If yes, leave it as it is
mov           SP,1FFE       ; 106 Set stack's top at 8 kb
mov           BX,0200       ; 109 Request for 8 kb space
mov           AH,4A         ; 10C A call for free MCB
int           21            ; 10E         creation function
```

```
cmp byte ptr [005E],46    ;=110 Is there the "F" parameter?
jz           0148         ;*115 If yes, let's go ahead
cmp byte ptr [005E],4E    ; 117 Is there the "N" parameter?
jz           0148         ;*11C If yes, let's go ahead
mov          DX,0317      ;*11E As required parameters
mov          AL,01        ; 121      are not present, go to
jmp          020A         ;*123      display help and exit
      ;********** Section 2: data, pointers, descriptors
         ; 126 - filled from 17D, called from 187, 1F7
         ; 128 - filled from 181, accessed at 190, 1E5
         ; 12A - GDT pseudo descriptor, accessed at 1CA
         ; 12C - GDT linear address, calculated at 1B2
                           ; 126 HIMEM.SYS entrance point
dw           0000,0000
                           ; 12A GDT size (3 descriptors)
db           18 00
                           ; 12C GDT address (offset = 130)
db           30 01 00 00
                           ; 130 "Empty" descriptor 0000
db           00 00 00 00 00 00 00 00
                           ; 138 4-Gb size descriptor 0008
db           FF FF 00 00 00 93 8F 00
                           ; 140 64-kb size descriptor 0010
db           FF FF 00 00 00 93 00 00
      ;******** Section 3: CPU and protection mode checks
         ; 148 - target for jumps from lines 115, 11C
         ; 162 - target for jump from line 158
pushf                      ;=148 Push 2 copies of original
pushf                      ; 149    flag's states into stack
pop          CX            ; 14A Load a copy into CX
xor          CH,70         ; 14B Invert bits 0C, 0D, 0E
push         CX            ; 14E Send inverted states via
popf                       ; 14F         stack into flags
pushf                      ; 150         register, and then
pop          AX            ; 151         via stack into AX
popf                       ; 152 Restore initial flag states
xor          AH,CH         ; 153 Set non-coincident bits
test         AH,40         ; 155 Is it a 16-bit CPU?
jz           0162          ;*158 If no, check protection
mov          AL,04         ; 15A If yes, go to display
mov          DX,024F       ;*15C         error message
jmp          020A          ;*15F         024F and exit
test         AH,30         ;=162 Is protected mode set?
```

```
jz            016F          ;*165 If no, go ahead
mov           AL,08         ; 167 If yes, go to display
mov           DX,0271       ;*169         error message
jmp           020A          ;*16C         0271 and exit
      ;********** Section 4: address bus A20 preparation
         ; 16F - target for jump from line 165
         ; 18B - target for jump from line 176
mov           AX,4300       ;=16F Check whether HIMEM.SYS
int           2F            ; 172         driver is installed
cmp           AL,80         ; 174 If no, go for direct
jnz           018B          ;*176             access to A20
mov           AX,4310       ; 178 If yes, query for entrance
int           2F            ; 17B              point address
mov           [0126],BX     ;*17D Store entrance point
mov           [0128],ES     ;*181     address in memory cells
mov           AH,05         ; 185 Call for A20 bus
call far      [0126]        ;*187      activation function
call          021C          ;=18B Check A20 state and jump
jnz           01A7          ;*18E      if A20 bus is active
mov word ptr  [0128],0001   ;*190 Mark: bus A20 is not active
mov           AL,FF         ; 196 Attempt to activate A20 bus
call          022F          ;*198          by subroutine 022F
call          021C          ;*19B Check A20 state and jump
jnz           01A7          ;*19E        if A20 bus is active
mov           AL,02         ; 1A0 If A20 is not active, go
mov           DX,029C       ;*1A2        to display error
jmp           020A          ;*1A5        message and exit
      ;*********** Section 5: preparation of GDT table
         ; 1A7 - target for jumps from lines 18E, 19E
                            ;=1A7 Data size override prefix
db            66
xor           AX,AX         ; 1A8 Write zero into EAX
mov           AX,CS         ; 1AA Copy CS segment into AX
mov           CL,04         ; 1AC Shift 4 bits leftwards
db            66
shl           AX,CL         ; 1AF Obtaining linear address
db            66
add           [012C],AX     ;*1B2 GDT's linear address
      ;******** Section 6: selectors, ban on interrupts
         ; 1C3 - target for jump from line 1BE
mov           CX,0008       ; 1B6 0008 - 4-Gb size selector
cmp byte ptr  [005E],46     ; 1B9 Is there "F" parameter ?
jz            01C3          ;*1BE If yes, leave CX=0008
```

```
mov          CX,0010      ; 1C0 If no, let it be CX=0010
cli                       ;=1C3 Prohibit interrupts
mov          AL,80        ; 1C4 Prohibit NMI by
out          70,AL        ; 1C6       sending byte 70h
in           AL,71        ; 1C8       into port 70h
     ;********* Section 7: transitions to PM and back
                          ; 1CA = Lgdt fword ptr [012A]
db           0F 01 16 2A 01
                          ; 1CF = mov EAX,CR0
db           0F 20 C0
or           AL,01        ; 1D2 Set protected mode
                          ; 1D4      bit (= mov CR0,EAX)
db           0F 22 C0
                          ; 1D7 Load GS (= mov GS,CX)
db           8E E9
and          AL,FE        ; 1D9 Clear protected mode
                          ; 1DB      bit (= mov CR0,EAX)
db           0F 22 C0
     ;********** Section 8: return to former state
        ; 1F5 - target for jump from line 1EC
mov          AL,7F        ; 1DE Enable NMI by
out          70,AL        ; 1E0      sending byte 7Fh
in           AL,71        ; 1E2      into port 70h
sti                       ; 1E4 Enable interrupts
cmp word ptr [0128],0001  ;*1E5 Check A20 state mark
jb           01FB         ;*1EA If mark=0000, do nothing
ja           01F5         ;*1EC If mark>0001 - to HIMEM.SYS
mov          AL,FD        ; 1EE If mark=0001, restore
call         022F         ;*1F0      A20 state by means
jmp          01FB         ;*1F3      of subroutine 022F
mov          AH,06        ;=1F5 Call for HIMEM.SYS function
call far     [0126]       ;*1F7      switching A20 bus OFF
     ;********** Section 9: message display and exit
        ; 1FB - target for jumps from lines 1EA, 1F3
        ; 208 - target for jump from line 203
        ; 20A - jump target from lines 123, 15F, 16C, 1A5
mov          DX,02E9      ;=1FB "Limit set" message offset
cmp byte ptr [005E],46    ; 1FE Is there "F" parameter?
jnz          0208         ;*203 If yes, specify offset for
mov          DX,02B9      ; 205      message "limit is OFF"
mov          AL,00        ;=208 Specify zero errorlevel
push         AX           ;=20A Jumps target point
mov          AH,40        ; 20B      to display messages
```

```
mov          BX,DX         ; 20D Read into CX a number
mov          CX,[BX-02]    ; 20F      characters to display
mov          BX,0001       ; 212 0001 = STDOUT's handle
int          21            ; 215 Send message to STDOUT
pop          AX            ; 217 Return errorlevel into AL
mov          AH,4C         ; 218 Call for DOS's program
int          21            ; 21A       termination function
      ;******** Section 10: A20 state check subroutine
         ; 21C - target for calls from lines 18B, 19B
push         DS            ;=21C Save DS segment in stack
xor          SI,SI         ; 21D Write zero into SI
mov          DS,SI         ; 21F now DS:SI=0000:0000
mov          DI,F0F1       ; 221 Set ES:DI=F0F1:F0F0
mov          ES,DI         ; 224    in order to obtain
dec          DI            ; 226    F0F10h + F0F0h = 100000h
cld                        ; 227 Set count UP
mov          CX,0010       ; 228 Set repeat limit 16 bytes
repz                       ; 22B      and repeat while equal
cmpsb                      ; 22C Is there a foldover?
pop          DS            ; 22D Restore DS, return result
ret                        ; 22E       via state of ZF flag
      ;******** Section 11: A20 state change subroutine
         ; 22F - target for calls from lines 198, 1F0
push         AX            ;=22F Save command code in stack
call         0241          ;*230 Wait controller's readiness
mov          AL,D1         ; 233 D1 - first byte of command,
out          64,AL         ; 235       sent into port 64h
call         0241          ;*237 Wait controller's readiness
pop          AX            ; 23A Restore command code in AX
out          60,AL         ; 23B     and send it to port 60h
call         0241          ;*23D Wait controller's actuation
ret                        ; 240            and then return
      ;******** Section 12: controller waiting subroutine
         ; 241 - target for calls from lines 230, 237, 23D
         ; 245 - cycle return point from line 249
push         CX            ;=241 Save CX state in stack
mov          CX,FFFF       ; 242 Write cycle's limit into CX
in           AL,64         ;=245 Read a byte from port 64h
test         AL,02         ; 247 Check state of the 2-nd bit
loopnz       0245          ;*249 Repeat, if port is busy
pop          CX            ; 24B Restore CX state from stack
ret                        ; 24C Return from subroutine
      ;********** Section 13: messages
```

```
db            20 00
        ; 24F - 1-st message, mentioned at 15C
db            0D 0A 09 "16-bit processor"
db            20 "can" 27 "t suit" 0D 0A
db            29 00
        ; 271 - 2-nd message, mentioned at 169
db            0D 0A 09 "GS_limit can" 27 "t run"
db            20 "in protected mode" 0D 0A
db            1B 00
        ; 29C - 3-rd message, mentioned at 1A2
db            0D 0A 09 "Line A20 control error" 0D 0A
db            2E 00
        ; 2B9 - 4-th message, mentioned at 205
db            0D 0A 09 "GS segment limit"
db            20 "protection is turned OFF" 0D 0A
db            2C 00
        ; 2E9 - 5-th message, mentioned at 1FB
db            0D 0A 09 "GS segment limit"
db            20 "protection is restored" 0D 0A
db            7F 00
        ; 317 - 6-th message (help), mentioned at 11E
db            0D 0A "GS_limit.com removes the GS segment"
db            20 "limit protection or can restore it back"
db            0D 0A "Usage examples:" 0A 0D 09 09 "GS_lim"
db            "it off" 0A 0D 09 09 "GS_limit on" 0D 0A
        ; 396 End of assembler text

n GS_limit.com
rBX
0000
rCX
0296
w
q
```

Assembler text starts in section 1 with ordinary release of allocated memory excess (note 5 to A.12-7). Then parameter's presence check is performed. Form of parameter's presentation is chosen so that parameters are automatically uppercased and written into first FCB (note 4 to A.07-1) inside program's PSP. If required parameters are not specified, then in line 123 a jump occurs to final section, where help message is displayed. After that utility comes to end, leaving errorlevel code 01.

When required command line parameters are specified, execution continues from line 148 in section 3, where two conditions are checked: CPU's suitability and its operation in real mode. The essence of these checks is described in notes 2 and 3 to appendix A.11-4. If either of conditions is not met, then jumps occur to final section, where corresponding error message is displayed, and execution comes to end, leaving errorlevel codes either 04 or 08.

When CPU checks are passed successfully, execution continues from line 16F in section 4, where state of address bus line A20 is prepared. Necessity of such preparation is not obvious, but has been confirmed. Address bus line A20 must be switched ON. If HIMEM.SYS driver is installed yet, its function AH=05 (A.12-3) should be called for in order to switch ON the address bus line A20. If HIMEM.SYS driver is not installed, then a subroutine 022F from section 11 should be called, which attempts to switch ON the address bus line A20 by means of keyboard controller (details – in note 1 to A.11-3).

Initial and final states of address bus line A20 are checked in lines 18B and 19B by calls for subroutine 021C in section 10. If attempts to switch ON address bus line A20 fail, then in line 1A5 a jump will be performed to final part of program. There an error message is displayed, and program's execution comes to end, leaving errorlevel code 02. Normally at least one attempt to switch ON address bus line A20 is a success, and then execution continues from line 1A7 in section 5.

Commands in section 5 prepare a pseudo descriptor for GDT table. Address of GDT table is copied from this pseudo descriptor by CPU into its GDTR register. In line 1AC segment address CS is transformed into linear address by shift 4 bits leftwards. Summation in line 1B2 forms linear address of GDT table inside pseudo descriptor template (in lines 12A – 12F of section 2).

Detailed structure of GDT table's descriptors is shown in appendix A.12-2. Second and third descriptors are prepared for data segments. In line 138 the second descriptor (selector 0008) defines 4 Gb segment size. It should be used to switch off GS segment protection. In line 140 the third descriptor (selector 0010) defines 64 kb segment size. This descriptor should be used in order to restore back normal protection for GS segment. Before CPU is switched to protected mode, in register CX that selector should be prepared, which corresponds to requested task. A choice between those two selectors – 0008 and 0010 – is made by commands in lines 1B6 – 1C0 of section 6. Following commands in section 6 prohibit interrupts, including NMI (details – in note 1 to article 8.01-03).

The first command in 7-th section is LGDT (= Load Global Descriptor Table), which loads data from GDT pseudo descriptor into CPU's GDTR register. DEBUG.EXE doesn't "know" the LGDT command, therefore its code (0F 01 16) is introduced as data by DB instruction. The last two bytes in LGDT command – 2A 01 – represent pseudo descriptor's

offset counted from segment address in DS register, i.e. just number 012A of a line, where pseudo descriptor's template is prepared in section 2 of assembler text.

CPU transition to protected mode can be performed by INT 15\AH=89h function (8.01-78), which requires a larger GDT table and reprograms both interrupt controllers. As far as proclaimed task implies a return back to real mode, later a backward reprogramming will be needed for interrupt controllers. In order to avoid excess complexity, transition to protected mode is performed here by setting the PE bit (PE = Protection Enable) in CPU's CR0 register (A.11-4). Therefore a command in line 1CF copies contents of CR0 register into AX, in this copy the PE bit is set, and then in line 1D4 the altered copy is written back into CR0 register. Commands concerning CR0 register (note 1 to 7.03-58) are not "known" to DEBUG.EXE and hence are introduced in lines 1CF and 1D4 as data following DB instruction.

Of course, PE bit setting is not sufficient for obtaining a full-functional protected mode, but in this case full functionality isn't required. In protected mode the GS_LIMIT.COM utility must perform a single operation: write into GS segment register that selector (0008 or 0010), which is prepared yet in CX register. Command copying CX contents into GS (note 2 to 7.03-58) is not "known" to DEBUG.EXE, and therefore its code is introduced in line 1D7 as data after DB instruction. Copying of a prepared selector into GS register brings about that hidden event, which is the main goal of GS_LIMIT.COM utility: new contents, including new segment limit, is written into CPU's GS "shadow" register from the GDT descriptor, pointed at by the copied selector.

When summit is reached yet, it's just the time to begin descent back. First of all, the PE bit must be cleared in that code, which is still preserved in EAX register since it has been copied from CPU's control register CR0. Command in line 1DB writes this twice altered code back into CR0 register. Thus CPU is returned into real mode. Then commands in 8-th section cancel interrupt prohibition and restore initial state of address bus line A20 by just those facilities, which previously have altered this state. Commands in lines 1FB – 215 of 9-th section select appropriate final message and display it. In lines 218 – 21A a call for DOS's INT 21\AH=4Ch function terminates execution of GS_LIMIT.COM utility.

After its termination the GS_LIMIT.COM utility leaves errorlevel code, which may present a separate interest in order to determine CPU's operating mode:

```
GS_limit  on > nul
if errorlevel 7 echo Processor runs in V86 mode
if not errorlevel 7 echo Processor runs in real mode
```

The last peculiarity of GS_LIMIT.COM utility is that the value, left behind in GS segment register, retains status of a selector. This enables to experiment with improper

linear addressing in CPU. After any operation, writing a new value into GS register, its contents will acquire proper status of a segment address.

9.10-02 Display of a linearly addressed dump

The GS_DUMP.COM utility, presented in this article, shows a 128-byte memory dump on the screen, almost as DEBUG.EXE does in response to its "D" command (6.05-04). The main difference is that GS_DUMP.COM utility instead of ordinary addresses (segment: offset) accepts 32-bit linear addresses, which enable to define just any access point within 4 Gb address space. While segment protection is active, GS_DUMP.COM utility responds with an error message to all requests for access beyond GS segment. But when GS segment protection is switched OFF by GS_LIMIT.COM utility (9.10-01), then dump of any memory region within 4 Gb address space can be displayed.

Besides its obvious demonstration effect, GS_DUMP.COM utility provides answers to many questions, concerning practical implementation of linear addressing in real mode.

In command line the name GS_DUMP.COM must be followed by linear address, composed of up to 8 hexadecimal digits, for example:

```
GS_dump.com  FFFE0
```

Specified linear address is interpreted as absolute address, counted from the start of address space irrespective to actual contents of GS register. After linear address there may be one of two optional parameters:

A – don't alter initial state of address bus line A20;
R – write zero into GS segment register.

Command line with an optional parameter may look, for example, as

```
GS_dump.com  FFFE0  A
```

When address bus line A20 initially is not active, GS_DUMP.COM utility with "A" optional parameter shows address space foldover at address 100000h. By default GS_DUMP.COM utility activates address bus line A20 and always shows address space opened irrespective to its actual initial state.

GS_DUMP.COM utility (1291 bytes long) is produced by debugger DEBUG.EXE as a result of command sequence execution. This sequence should be written into command file GS_DUMP.SCR by means of editor program (as described in introduction to chapter 9). Comments may be omitted. Special attention should be paid to empty line – 8-th from the end. It must be there, because it forces DEBUG.EXE to exit assembler mode (7.01-04). Then command file GS_DUMP.SCR should be sent to debugger via input redirection:

```
DEBUG.EXE < GS_DUMP.SCR
```

Command file GS_DUMP.SCR must contain the following lines:

```
a 100
      ;*************** GS_dump.com *************
      ;********* Section 1: preliminary preparations
         ; 110 - target for jump from line 104
cmp           SP,2010       ; 100 Less than 8 kb allocated?
jbe           0110          ;*104 If yes, leave it as it is
mov           SP,1FFE       ; 106 Set stack's top at 8 kb
mov           BX,0200       ; 109 Request for 8 kb space
mov           AH,4A         ; 10C A call for free MCB
int           21            ; 10E          creation function
push          CS            ;=110 Prepare
pop           DS            ; 111           DS = CS
db            66
xor           BX,BX         ; 113 Write zero in EBX register
      ;********* Section 2: CPU checks
         ; 12F - target for jump from line 125
pushf                       ; 115 Push 2 copies of original
pushf                       ; 116    flag's states into stack
pop           CX            ; 117 Load a copy into CX
xor           CH,70         ; 118 Invert bits 0C, 0D, 0E
push          CX            ; 11B Send altered bits via
popf                        ; 11C    stack to flags register,
pushf                       ; 11D        and then again
pop           AX            ; 11E        via stack into AX
popf                        ; 11F Restore initial flag states
xor           AX,CX         ; 120 Set non-coincident bits
test          AH,40         ; 122 Is it a 16-bit CPU?
jz            012F          ;*125 If no, go to next check
mov           AL,02         ; 127 If yes, go to display
mov           DX,03D6       ;*129        error message
jmp           02BF          ;*12C            03D6 and exit
test          AH,30         ;=12F Is protected mode set?
jz            0163          ;*132 If no, bypass section 3
      ;********* Section 3: protected mode checks
         ; 14B - target for jump from line 141
mov word ptr  [03D4],0483   ;*134 Prepare message address
mov           AX,1687       ; 13A Send trial request
int           2F            ; 13D           for DPMI server
or            AX,AX         ; 13F Is DPMI server installed?
jnz           014B          ;*141 If not, go further
mov           AL,08         ; 143 If yes, go to display
mov           DX,03FB       ;*145       error message
```

```
jmp           02BF          ;*148            03FB and exit
push          DS            ;=14B Save DS segment in stack
mov           DS,BX         ; 14C 0 = interrupt table segment
mov           DS,[019E]     ; 14E Load EMM's segment into DS
cmp word ptr  [0014],4249   ; 152 Is it IBM's driver?
pop           DS            ; 158 Restore DS from stack
jz            0163          ;*159 Continue, if IBM's driver
mov           AL,02         ; 15B If not, go to display
mov           DX,041E       ;*15D      error message
jmp           02BF          ;*160            041E and exit
      ;********* Section 4: linear address reading
         ; 163 - target for jumps from lines 132, 159
         ; 16A - cycle return target from line 19C
         ; 17B - target for jump from line 175
         ; 188 - target for jump from line 17E
         ; 194 - target for jumps from lines 16F, 179
mov           CX,0004       ;=163 Preset 4 bit shift
cld                         ; 166 Set SI count upwards
mov           SI,005D       ; 167 Set start address
lodsb                       ;=16A Load one ASCII code into AL
sub           AL,30         ; 16B Translate ASCII code
cmp           AL,09         ; 16D If it is decimal digit,
jbe           0194          ;*16F        append it to EBX
sub           AL,07         ; 171 Translate A-F letter codes
cmp           AL,0A         ; 173 Is it character below "A"?
jb            017B          ;*175 If yes, check the reason
cmp           AL,0F         ; 177 Is it one of letters A-F?
jbe           0194          ;*179 If yes, go append it to EBX
cmp           SI,005E       ;=17B Is it the first iteration?
jnz           0188          ;*17E If not, let's check further
mov           AL,01         ; 180 If yes, go to display
mov           DX,04F1       ;*182        help message 04F1
jmp           02BF          ;*185          and then exit
cmp           AL,E9         ;=188 Is last character a space?
jz            019E          ;*18A If yes, no more characters
mov           AL,01         ; 18C If no, go to display
mov           DX,04CB       ;*18E         error message
jmp           02BF          ;*191             04CB and exit
                            ;=194 Data size override prefix
db            66
shl           BX,CL         ; 195 Shift EBX 4 bits leftwards
or            BL,AL         ; 197 Insert next character in BL
cmp           SI,0064       ; 199 Is 8-th iteration reached?
```

```
jbe           016A          ;*19C If not, repeat the cycle
      ;******** Section 5: address bus line A20 activation
         ; 19E - target for jump from line 18A
         ; 1C3 - target for jump from line 1AC
         ; 1D3 - target for jump from line 1A3
         ; 1D8 - target for jump from line 1C6
cmp byte ptr [006D],41     ;=19E Is "A" parameter present?
jz            01D3         ;*1A3 If yes, bypass section 5
mov           AX,4300      ; 1A5 Is the HIMEM.SYS
int           2F           ; 1A8         driver installed?
cmp           AL,80        ; 1AA If no, bypass appeals
jnz           01C3         ;*1AC         to HIMEM.SYS driver
push          BX           ; 1AE Save BX contents in stack
mov           AX,4310      ; 1AF Call for HIMEM.SYS
int           2F           ; 1B2      entrance point address
mov           [03C0],BX    ;*1B4 Store entrance point
mov           [03C2],ES    ;*1B8     address in memory cells
mov           AH,05        ; 1BC Call for bus line A20
call far      [03C0]       ;*1BE         activation function
pop           BX           ; 1C2 Restore former BX contents
call          02D3         ;=1C3 Is bus line A20 active?
jnz           01D8         ;*1C6 If yes, bypass next attempt
mov word ptr [03C2],0001   ;*1C8 If no, set a mark
mov           AL,FF        ; 1CE Activate bus line A20
call          02EB         ;*1D0          by subroutine 02EB
call          02D3         ;=1D3 Is bus line A20 active?
jz            01DD         ;*1D6 If yes, message 0445
mov byte ptr [0445],24     ;=1D8        must be made invalid
      ;******* Section 6: indication of GS segment address
         ; 1DD - target for jump from line 1D6
         ; 1E6 - target for jump from line 1E2
cmp byte ptr [006D],52     ;=1DD Is "R" parameter present?
jz            01E6         ;*1E2 If yes, don't read GS
                           ; 1E4  = MOV SI,GS
db            8C EE
                           ;=1E6  = MOV GS,SI
db            8E EE
                           ; 1E8  = PUSH GS
db            0F A8
pop           [03D0]       ;*1EA Store GS contents in 3D0
call          0339         ;*1EE Send 0Dh 20h to STDOUT
call          0349         ;*1F1 Display word stored in 3D0
mov           DX,0445      ;*1F4 Display
```

```
call          0330          ;*1F7         message 0445
mov           DX,045B       ;*1FA Display
call          0330          ;*1FD         message 045B
      ;********* Section 7: datum point calculation
db            66
xchg          SI,BX         ; 201 Copy address into ESI
mov           BX,SI         ; 203 Return in BX 2 bytes only
and           BX,000F       ; 205 Select the rightmost digit
neg           BL            ; 209 Get datum point
and           SI,FFF0       ; 20B Now ESI - base address
db            66
mov           DI,SI         ; 210 Copy ESI into EDI
db            66
xchg          [03D0],DI     ;*213 Exchange [03D0] with EDI
db            66
shl           DI,CL         ; 218 In EDI - linear address GS
mov           DX,[03D4]     ;*21A Prepare message number
db            66
sub           SI,DI         ; 21F Calculate offset in ESI
jnb           0226          ;*221 If below zero, change
mov           DX,0460       ;*223      message number to 0460
      ;********* Section 8: flags and pointers replacement
         ; 226 - target for jump from line 221
                            ;=226 Data size override prefix
db            66
pushf                       ; 227 Push EFLAGS into stack
mov           BP,SP         ; 228 Copy stack pointer into BP
mov           AL,[BP+02]    ; 22A Read and store the
mov           [03C4],AL     ;*22D          third flag's byte
and byte ptr  [BP+02],FE    ; 230 Clear alignment flag
db            66
popf                        ; 235 Write EFLAGS back
in            AL,21         ; 236 Read port 21h mask
mov           [03C5],AL     ; 238          byte and store it
or            AL,60         ; 23B Set bits 05 and 06
out           21,AL         ; 23D Disable IRQ5 and IRQ6
mov           [03C8],DS     ;*23F Fill segment cells in
mov           [03CC],DS     ;*243         handler's addresses
push          BX            ; 247 Save BX contents in stack
mov           AL,0D         ; 248 Exchange handlers
call          03A0          ;*24A    for interrupt INT 0D and
call          03A0          ;*24D        for interrupt INT 0E
pop           BX            ; 250 Restore BX contents
```

```
      ;********* Section 9: trial reading attempt
         ; 251 - cycle return target from line 289
mov           [03CE],SI     ;=251 Save SI for comparison
db            65 67
lodsb                       ; 257 Attempt to read a byte
cmp           SI,[03CE]     ;*258 Has SI value changed?
jnz           0266          ;*25C If yes, go to main cycle
call          0342          ;*25E Display linear address
call          0321          ;*261 Calculate next address
jmp           0286          ;*264 Jump to check new address
      ;********* Section 10: main line display cycle
         ; 266 - target for jump from line 25C
         ; 26E - cycle return target from line 273
         ; 278 - cycle return target from line 27D
         ; 286 - target for jump from line 264
                            ;=266 Data size override prefix
db            66
dec           SI            ; 267 Restore index in ESI
call          0349          ;*268 Display linear address
call          0318          ;*26B Intermediate correction
call          0362          ;=26E Display a byte in AL
inc           BL            ; 271 Increment bytes count by 1
loop          026E          ;*273 1-st line display cycle
call          0309          ;*275 Intermediate correction
call          0379          ;=278 Display a byte as ASCII
inc           BL            ; 27B Increment bytes count by 1
loop          0278          ;*27D 2-nd line display cycle
call          0339          ;*27F Send 0Dh 20h to STDOUT
mov           DX,[03D4]     ;*282 Update message number
cmp           BL,80         ;=286 Is display line the last?
jb            0251          ;*289 If not, display next line
      ;********* Section 11: return to initial states
         ; 2B4 - target for jump from line 2AB
mov           AL,0D         ; 28B Restore handlers for
call          03A0          ;*28D        interrupt INT 0D and
call          03A0          ;*290        for interrupt INT 0E
mov           AL,[03C5]     ;*293 Read former mask byte and
out           21,AL         ; 296         send it to port 21h
db            66
pushf                       ; 299 Read EFLAGS into stack
mov           BP,SP         ; 29A Copy stack pointer into BP
mov           AL,[03C4]     ;*29C Read and restore former
mov           [BP+02],AL    ; 29F            3-rd flag's byte
```

```
db            66
popf                        ; 2A3 Restore EFLAGS states
cmp word ptr  [03C2],0001   ;*2A4 Check mark of A20 state
jb            02BA          ;*2A9 If 0000, don't touch A20
ja            02B4          ;*2AB IF >0001 - go use HIMEM.SYS
mov           AL,FD         ; 2AD Restore A20 states with
call          02EB          ;*2AF             subroutine 02EB
jmp           02BA          ;*2B2 Jump to final operations
mov           AL,06         ;=2B4 Call HIMEM.SYS function
call far      [03C0]        ;*2B6       in order to close A20
      ;******** Section 12: program's termination
         ; 2BA - target for jumps from lines 2A9, 2B2
         ; 2BF - jumps from lines 12C, 148, 160, 185, 191
mov           DX,0442       ;=2BA Offset for line's end bytes
mov           AL,00         ; 2BD Happy end errorlevel
push          AX            ;=2BF Save errorlevel in stack
mov           AH,09         ; 2C0 Display final
int           21            ; 2C2               message
pop           AX            ; 2C4 Restore errorlevel
mov           AH,4C         ; 2C5 Call for termination
int           21            ; 2C7     function, return to DOS
      ;******* Section 13: interrupt's 0D and 0E handlers
         ; 2C9  - must be specified in cell 3CA
         ; 2CC  - must be specified in cell 3C6
mov           DX,04A5       ;=2C9 Call target for Int 0E
mov           BP,SP         ;=2CC Call target for Int 0D
add word ptr  [BP+00],+03   ; 2CE Correct return address
iret                        ; 2D2         in stack and return
      ;******** Section 14: state check for bus line A20
         ; 2D3 - target for calls from lines 1C3, 1D3
push          DS            ;=2D3 Save states of
push          CX            ; 2D4       DS and CX registers
db            66
xor           SI,SI         ; 2D6 Write zero into ESI
push          SI            ; 2D8 Save SI=0 in stack
mov           DS,SI         ; 2D9 Now DS:SI=0000:0000
mov           DI,F0F1       ; 2DB Prepare ES:DI=F0F1:F0F0
mov           ES,DI         ; 2DE    in order to get address
dec           DI            ; 2E0    F0F10h + F0F0h = 100000h
cld                         ; 2E1 Set count upwards
mov           CX,0010       ; 2E2 Set count limit 16 bytes
repz                        ; 2E5 Repeat while equal
cmpsb                       ; 2E6 Is there a foldover?
```

```
pop          SI            ; 2E7 Restore states of
pop          CX            ; 2E8    registers and return,
pop          DS            ; 2E9        keeping result as
ret                        ; 2EA            state of ZF flag
      ;***** Section 15: bus line A20 control subroutine
         ; 2EB - target for calls from lines 1D0, 2AF
push         AX            ;=2EB Save command in stack
call         02FD          ;*2EC Wait controller readiness
mov          AL,D1         ; 2EF Send 1-st command's
out          64,AL         ; 2F1         byte into port 64h
call         02FD          ;*2F3 Wait controller readiness
pop          AX            ; 2F6 Send 2-nd command's
out          60,AL         ; 2F7         byte into port 60h
call         02FD          ;*2F9 Wait for controller's
ret                        ; 2FC   actuation and then return
      ;********* Section 16: wait for controller readiness
         ; 2FD - target for calls from lines 2EC, 2F3, 2F9
         ; 301 - cycle return target from line 305
push         CX            ;=2FD Save CX contents in stack
mov          CX,FFFF       ; 2FE Store cycles limit in CX
in           AL,64         ;=301 Read state of port 64h
test         AL,02         ; 303 Check readiness bit
loopnz       0301          ;*305 Repeat, if not ready
pop          CX            ; 307 Restore CX and return
ret                        ; 308      to the caller program
      ;********* Section 17: intermediate correction
         ; 309 - target for call from line 275
         ; 318 - target for call from line 26B
sub          BL,10         ;=309 Return count one line back
push         BX            ; 30C Save BX contents in stack
mov          BL,10         ; 30D 10h = increment value
db           66
add          [03D0],BX     ;*310 Linear address correction
db           66
sub          SI,BX         ; 315 Offset correction
pop          BX            ; 317 Restore former BX contents
call         0393          ;=318 Twice send a space
call         0393          ;*31B        character to STDOUT
mov          CL,10         ; 31E Preset bytes count
ret                        ; 320 Return from subroutine
      ;********* Section 18: transition to next dump line
         ; 321 - target for call from line 261
         ; 330 - target for calls from lines 1F7, 1FD
```

```
        ; 339 - target for calls from lines 1EE, 27F
add           BL,10         ;=321 Increment bytes count by 16
push          BX            ; 324 Save BX contents in stack
mov           BL,10         ; 325 Set increment by 16
db            66
add           [03D0],BX     ;*328 Linear address correction
db            66
add           SI,BX         ; 32D Offset correction
pop           BX            ; 32F Restore former BX contents
mov           DI,DX         ;=330 Message address - into DI
mov           AH,09         ; 332 Call for DOS's STDOUT
int           21            ; 334         output function
mov byte ptr  [DI],24       ; 336 Disable message
mov           AL,0D         ;=339 Send carriage return
call          0395          ;*33B          command to STDOUT
call          0393          ;*33E Send a space to STDOUT
ret                         ; 341 Return from subroutine
      ;********* Section 19: cell 3D0 contents display
        ; 342 - target for a call from line 25E
        ; 349 - target for calls from lines 1F1, 268
        ; 354 - target for jump from line 35E
        ; 361 - target for jump from line 347
mov           DI,DX         ;=342 Is the requested
cmp byte ptr  [DI],24       ; 344          message disabled?
jz            0361          ;*347 If yes, don't display it
mov           AL,0A         ;=349 Send a line feed
call          0395          ;*34B          command to STDOUT
push          BX            ; 34E Save BX contents in stack
xor           BL,BL         ; 34F Turn off limit check
mov           DI,03D3       ; 351 Load 1-st byte offset in DI
mov           AL,[DI]       ;=354 Copy that byte into AL
call          0368          ;*356 Call AL byte translation
dec           DI            ; 359 Turn to next byte
cmp           DI,03D0       ;*35A Is it the last byte?
jnb           0354          ;*35E If not, repeat the cycle
pop           BX            ; 360 Restore former BX contents
ret                         ;=361 Return from subroutine
      ;********* Section 20: AL translation subroutine
        ; 362 - target for a call from line 26E
        ; 368 - target for a call from line 356
call          0393          ;=362 Call to display a space
db            67 65
lodsb                       ; 367 Read GS:[linear address]
```

```
push          CX            ;=368 Save CX contents in stack
mov           CL,04         ; 369 Preset 4 bits shift in CL
shl           AH,CL         ; 36B Clear 4 bits in AH
shl           AX,CL         ; 36D Separate half-bytes from AL
shr           AL,CL         ; 36F Clear 4 bits in AL
pop           CX            ; 371 Restore former CX contents
call          0384          ;*372 Call for AH display
call          0384          ;*375 Call for AL display
ret                         ; 378 Return from subroutine
      ;********* Section 21: one character display
         ; 379 - target for a call from line 278
         ; 384 - target for calls from lines 372, 375
         ; 38E - target for calls from lines 37E, 382, 38A
         ; 393 - called from lines 318, 31B, 33E, 362
         ; 395 - called from lines 33B, 34B, jump from 391
                            ;=379 Copy one character into AL
db            67 65
lodsb                       ; 37B    from GS:[linear address]
cmp           AL,20         ; 37C Is it 20h value or more?
ja            038E          ;*37E Values AL < 20h replace
mov           AL,2E         ; 380      with dots and jump
jmp           038E          ;*382         to check boundary
xchg          AH,AL         ;=384 Exchange contents AH - AL
add           AL,30         ; 386 Translate 0 - 9 into ASCII
cmp           AL,39         ; 388 Is it 0 - 9 or A - F ?
jbe           038E          ;*38A Leave digits 0 - 9 intact
add           AL,07         ; 38C Translate A - F into ASCII
cmp           BL,80         ;=38E Check boundary
jb            0395          ;*391 If beyond boundary,
mov           AL,20         ;=393     replace it with space
push          AX            ;=395 Save states of AX and
push          DX            ; 396      DX registers in stack
mov           AH,02         ; 397 Call for DOS's function
mov           DL,AL         ; 399      of character output
int           21            ; 39B             into STDOUT
pop           DX            ; 39D Restore former states
pop           AX            ; 39E      of AX and DX registers
ret                         ; 39F Return from subroutine
      ;********* Section 22: handler's exchange subroutine
         ; 3A0 - called from lines 24A, 24D, 28D, 290
mov           AH,35         ;=3A0 Call for handler's address
int           21            ; 3A2   reading function in ES:BX
mov           DI,AX         ; 3A4 Prepare in DI a memory cell
```

```
shl          DI,1          ; 3A6        offset: DI = AX * 2
shl          DI,1          ; 3A8 350D*4=D434    350E*4=D438
sub          DI,D06E       ;*3AA D434-D06E=3C6  D438-D06E=3CA
push         DS            ; 3AE Store contents of DS and
push         DX            ; 3AF       DX registers in stack
lds          DX,[DI]       ; 3B0 DS:DX - pointer to a cell
mov          AH,25         ; 3B2 Call for handler's address
int          21            ; 3B4           writing function
pop          DX            ; 3B6 Restore former contents
pop          DS            ; 3B7      of DS and DX registers
mov          [DI],BX       ; 3B8 Store address of the former
mov          [DI+02],ES    ; 3BA     handler in memory cells
inc          AL            ; 3BD Prepare next number in AL
ret                        ; 3BF Return from subroutine
     ;******** Section 23: data and addresses
        ; 3C0 - HIMEM.SYS offset, in lines 1B4, 1BE, 2B6
        ; 3C2 - HIMEM.SYS segment, in lines 1B8, 1C8, 2A4
db           00 00 00 00
        ; 3C4 - 3-rd byte of EFLAGS, in lines 22D, 29C
db           00
        ; 3C5 - mask for port 21h, in lines 238, 293
db           00
        ; 3C6 - offset of this cell is calculated in 3AA
        ; 3C8 - segment of 0Dh handler, written from 23F
db           CC 02 00 00
        ; 3CA - offset of this cell is calculated in 3AA
        ; 3CC - segment of 0Eh handler, written from 243
db           C9 02 00 00
        ; 3CE - place to store SI, accessed from 251, 258
db           00 00
        ; 3D0 - linear address - 1EA, 213, 310, 328, 35A
        ; 3D3 - most significant address byte, from 351
db           00 00 00 00
        ; 3D4 - place for message address - 134, 21A, 282
db           71 04
     ;******** Section 24: messages
        ; 3D6  1-st message, mentioned at 129
db           0D 0A 'Error: 16-bit machine can' 27
db           't suit' 0D 0A 24
        ; 3FB  2-nd message, mentioned at 145
db           0D 0A 'Error: can' 27 't run under WINDOWS'
db           0D 0A 24
        ; 41E  3-rd message, mentioned at 15D
```

```
db            0D 0A 'Error: incompatible EMM386 version'
        ; 442  4-th virtual message, mentioned at 2BA
db            0D 0A 24
        ; 445  5-th message, mentioned at 1D8, 1F4
db            09 'Line A20 is disabled' 24
        ; 45B  6-th message, mentioned at 1FA
db            'GS=' 20 24
        ; 460  7-th message, mentioned at 223
db            09 '- below GS base' 24
        ; 471  8-th message, addressed in line 3D4
db            09 '- above GS limit' 24
        ; 483  9-th message, mentioned at 134
db            09 'GS range or privilege violation' 20 24
        ; 4A5  10-th message, mentioned at 2C9
db            09 'Page isn' 27 't initialized or is'
db            20 'swapped' 24
        ; 4CB  11-th message, mentioned at 18E
db            0D 0A 'Address error: invalid character(s)' 0A
        ; 4F1  12-th message (help), mentioned at 182
db            0D 0A 09 'GS_dump.com - linear GS address'
db            20 'dump utility' 0D 0A 'Usage examples:'
db            0D 0A 09 09 'GS_dump  002FABCD' 0D 0A 09 09
db            'GS_dump  002FABCD  A' 0D 0A 09 09
db            'GS_dump  002FABCD  R' 0D 0A 20
db            '002FABCD  - linear address example, up to'
db            20 '8 hexadecimal digits long' 0D 0A 09
db            'A  - option: don' 27 't try to enable' 20
db            'line A20' 0D 0A 09 'R  - option: reset GS'
db            20 'to zero' 0D 0A 24
        ; 60B   End of assembler text

n GS_dump.com
rBX
0000
rCX
050B
w
q
```

Program starts in section 1 with ordinary release of allocated memory excess (note 5 to A.12-7). CPU checks and protected mode checks are performed exactly as in GS_LIMIT.COM program (9.10-01). But in case of protected mode the GS_DUMP.COM utility is not terminated at once; investigation continues in section 3. If protected mode is controlled not by WINDOWS OS, but by compatible IBM's version 4.50 of EMM386.EXE driver (5.04-02), then execution proceeds further.

In the 4-th section linear address, specified in command line, is read into EBX register. In course of reading the ASCII code characters are translated into "raw" code and are checked for correctness. If something else is found there except correct hexadecimal digits, then GS_DUMP.COM utility displays error message and terminates.

In the 5-th section of GS_DUMP.COM the address bus line A20 is activated just as it is activated in section 4 of GS_LIMIT.COM program (9.10-01).

Important specific operations in GS_DUMP.COM program begin with preparation of GS register in section 6. Preparation is needed, because contents of GS register may retain status of a selector. In order to get rid of uncertainty, command in line 1E6 performs writing into GS register. Even when contents of GS register should be preserved, command in line 1E6 writes in GS register just that value, which has been read from GS register by a command in preceding line 1E4. After writing operation the contents of GS register acquire normal status of a segment address.

Commands in final lines of 6-th section display GS segment address. After that commands in lines 201 – 20B of 7-th section calculate datum point byte and base address, i.e. linear address of the first byte in a line of dump. In line 218 a shift 4 bits leftwards transforms GS segment address into linear address. In line 21F a difference between base address and GS linear address is calculated. This difference represents just that offset, which is necessary for reading data at absolute address, specified by the user in command line.

As far as absolute address reading operation is not necessarily directed within allowed segment limits, it may invoke generation of exceptions INT 0D, INT 0E (notes 6 and 7 to 8.01-09) and INT 11 (note 1 to 8.01-42). Each of these exceptions will cause computer's hanging, unless some special precautions are taken in advance. Exception INT 11 can be prevented by clearing misalignment control bit (12h) in EFLAGS register (A.11-4). Therefore commands in lines 226 – 235 copy EFLAGS register contents into stack, initial state of the third read byte is stored in a 03C4 memory cell, misalignment control bit is cleared just in stack, and the altered result is written back from stack into EFLAGS register.

Exceptions INT 0D and INT 0E can't be prevented, processor inevitably will generate these exceptions at attempts of access to protected memory regions. Calls for exception handlers can be intercepted, but in real mode it is difficult to discriminate between CPU's calls for exceptions and external interrupt calls, received via IRQ 5 and IRQ 6 lines of interrupt controller (8.01-09). In order to avoid confusion the GS_DUMP.COM program prohibits reception of external interrupts via lines IRQ 5 and IRQ 6 for the time needed to display the dump. For this purpose commands in lines 236 – 23D read mask of 21h port, save it's initial state in memory cell 03C5, set bits 05 and 06 in this mask, and write the altered mask back to port 21h.

Suitable replacements for INT 0D and INT 0E exception handlers are prepared beforehand in section 13 of GS_DUMP.COM program. Commands in 13-th section make return address correction in stack and then return control back to interrupted program. Return address correction prevents hanging in a cycle of single command execution and enables resumption of interrupted program execution from the next command. Prepared handler's replacements are made active by commands in lines 23F – 24D of section 8. These commands fill memory cells 03C8 and 03CC with actual segment address, and then twice call for subroutine 03A0, which exchanges interrupt handler's addresses in interrupt table with those prepared in memory cells 03C8 and 03CC. Since that moment the GS_DUMP.SYS program eliminates threat of computer's hanging caused by CPU's exceptions INT 0D and INT 0E.

As far as access rights in address space can be changed with discreteness not less than 16 bytes, readability of all bytes inside a 16-byte paragraph can be determined by one trial access attempt, undertaken in line 257 of section 9. When trial byte is accessible, then CPU while performing the LODSB command (7.03-53) in line 257 increments offset in register SI by 1. But when reading request causes segment protection actuation, then LODSB command is not carried out, and SI register doesn't get its increment. Command in line 258 compares current value in SI register with the former one, saved in 03CE memory cell. Equality of the compared values is an evidence of segment protection actuation.

If CPU responds to trial access with segment protection actuation, then further attempts of access to any byte in the same paragraph are useless. For such lines of dump the following commands in section 9 display linear address, display a message about protection actuation, and calculate linear address for the next line of a dump. Then a jump follows to line 286, where number of currently displayed line in a dump is checked. If it is not the last line in dump, then a return is performed to start of trial access procedure, which is to be repeated for the next line of dump.

If SI register contents values, compared in line 258, happen to be different, then a jump from line 25C leads to the main dump display procedures in section 10. These procedures include a call for 0349 subroutine in order to display base linear address, a cycle 26E –

273, displaying 16 bytes of dump, and then cycle 278 – 27D, displaying the same 16 bytes as codes ASCII. If current line of dump is not the last one, then a return is performed to start of section 9, where all the same operations will be repeated for the next line of dump.

When dump processing is finished, commands in section 11 restore initial states of all affected elements. Subroutine 03A0 is called twice in order to restore former handler's addresses for interrupts INT 0D and INT 0E. Commands in lines 293 – 2A3 restore former mask in port 21h and former state of EFLAGS register. Commands in lines 2A4 – 2B6 restore former state of address bus line A20.

Having restored initial states, GS_DUMP.COM utility proceeds to concluding section 12. If dump has been displayed successfully, then final message is not displayed, and errorlevel is set to zero value. But if attempt to display a dump has failed, then the same operations in lines 2BF – 2C4 display an error message. In line 2C7 a call for DOS's INT 21\AH=4Ch function terminates execution of GS_DUMP.COM utility.

Note 1: errorlevel code, left behind by GS_DUMP.COM utility, may present a separate interest. In particular, the largest errorlevel code 8 is left by GS_DUMP.COM utility after attempt of its execution inside the "DOS box" of WINDOWS OS. Registration of termination with errorlevel code 8 (3.15-03) enables to prevent execution of those programs, which shouldn't be launched inside "DOS box", for example, of the TURN_OFF.COM utility (9.05-02).

9.11 MS-DOS7 loading alternatives

With respect to preservation of data and of main computer's operating system, any experiments with DOS are most safe when DOS is loaded from an external drive or from a removable media – a diskette, or flash card, or compact disc. This way of loading DOS, described in article 9.11-01, is common for emergency service. But it isn't sufficiently reliable for regular DOS usage, because in this role a lifetime of either removable media is not long enough.

For regular DOS usage a multi-alternative loading is arranged either by special boot managers or by loaders of some operating systems. Articles 9.11-02 and 9.11-03 describe usage for this purpose of WINDOWS-2000 and WINDOWS-XP loaders, and also of MS-DOS7 itself, as far as initially it was devised for launching WINDOWS-95/98 operating systems. Of course, capabilities of MS-DOS7 as a boot manager are poor, but it has a useful feature: compatibility with loadable BIOS extensions. May be, for alternative loading you'll have to choose just MS-DOS7.

9.11-01 MS-DOS7 loading from removable media

Ordinary 1.44 Mb diskette becomes bootable when a valid boot sector and DOS's system files are written onto that diskette. These operations under MS-DOS7 are performed by SYS.COM utility (6.24). Under Windows you may download a self-extracting file-image of a bootable diskette via Internet, for example, from host http://1gighost.com/ed/jamiephiladelphia/ or from site http://anbcomp.com/files/bootdisk/ . Versions without DOS relocation to RAM-disk (files boot95b.exe, boot98c.exe, boot98sc.exe) are preferable, since standard bootable diskettes employ relocation method which may fail in modern computers. Because of the same reason formation of a bootable diskette shouldn't be ordered to standard formatting procedure. However, the mentioned drawback isn't inherent to bootable diskettes with MS-DOS8, prepared by formatting procedure under Windows-XP.

Having got a standard bootable diskette for DOS, you most probably wouldn't be satisfied with its poor contents. Minimal suitable set of utilities you'll have to prepare yourself. Many utilities for MS-DOS7 can be got from Windows-95/98 release. Bootable diskette images with better sets of software can be found, for example, in http://www.netbootdisk.com/ and in http://www.multiboot.ru/ . Drivers and utilities, mentioned in configuration files in parts 6.25, 9.01, 9.04, 9.09 of this book, may be regarded as author's recommendations on compiling sets of software for bootable diskettes. Of course, in any case all specifications and paths in configuration files must correspond exactly to actual placement of drivers and utilities in the chosen bootable media.

Your attempt to load MS-DOS7 from diskette may fail because of inadequate parameter's specifications in BIOS Setup (about entering BIOS Setup – in article 1.01). Type of your floppy drive must be specified properly in page "Main" of BIOS Setup. Besides that, page "Boot" of BIOS Setup must assign to floppy drive a higher boot device priority, than to fixed drive(s). In obsolete computers boot device priority was defined by order of disk's letter-names; therefore in page "Boot" a list of letter-names must start from letter "A", which denotes first floppy drive. In modern computers first floppy drive must be the first in a list of removable drives, and loading from removable drives must be given higher priority, than loading from fixed drives.

Unfortunately, active operating system on a diskette is too slow and causes intensive wear-out. It diminishes diskette's lifetime to 10 – 20 hours. Functioning becomes much more fast and reliable, if DOS is relocated from bootable diskette to a RAM-disk (examples – in 9.04, 9.09). However, even in the latter case diskette can't be considered a fortunate bootable media. Slow and noisy DOS relocation procedure causes irritation. Besides that, total capacity of a diskette now seems insufficient.

Optical discs seem much more attractive because of their greater reading speed, much larger capacity and longer lifetime. However, a long lifetime is inherent to those optical discs only, which are produced with fixed contents by matrix replication technology.

Writable discs implement quite different storage principle: decay of organic dye. Optically written track loses contrast each time it is read. Therefore active lifetime of writable optical discs is similar to that of diskettes. Rumors about reading speed are also half-true: track seek time in optical drives is about 100 mS, while in HDDs it is about 2 mS. Because of short active lifetime and slow access, bootable optical disks need DOS relocation no less that floppy disks.

Some complexity is due to difference between optical disc's file systems and those "known" to DOS's core. Therefore DOS can't be loaded immediately from optical disc. An image of other disk – a bootable disk with a "known" file system FAT12 or FAT-16 – must be written onto optical disc. BIOS system of your computer must be able to emulate a logical disk from this image, and only then MS-DOS7 can be loaded from that emulated logical disk. If prototype for this image was a bootable diskette, then letter-name "A" is assigned to emulated logical disk, and real floppy drive is given the next letter-name "B".

Almost all programs for writing optical disks are able to copy a bootable diskette into an image and to write this image on optical disc, thus making it bootable. For this role no special preparation of bootable diskettes is needed. Configurations with MS-DOS7 relocation onto a RAM-disk are quite suitable. Examples of such configurations are shown in parts 9.04 and 9.09. Of course, configurations should include those drivers and TSRs (5.08-04, 5.09-04), which enable access to optical disc's space beyond emulated logical disk and thus eliminate problem of its insufficient capacity.

DOS can't be loaded from optical disc, unless optical disc drive is registered by computer's BIOS system as bootable device. A list of registered bootable devices is shown on page "BOOT" of BIOS Setup program. In this list different BIOS versions specify either a class of device (for example, CD-ROM) or a particular type of device. A place of device in this list defines its priority. If optical disc drive is found there, then you have to specify that priority order, which will force BIOS to address optical disc drive before any of fixed disk drives is addressed. After that all you have to do is to insert a bootable optical disc into the drive in time before BIOS begins its start tests.

In modern computers registration of a drive by BIOS system may fail because of wrong specification of interface parameters. In order to check parameters of IDE interface you have to open page "Main" of BIOS Setup program and press button "IDE Configuration". A new page will be opened, where parameter "Onboard IDE operate Mode" should be given value "Compatible Mode", and parameter "Combined Mode Option" should be given that value, which corresponds to actual type of IDE connection. Nowadays most internal optical disc drives use parallel P-ATA connection. If there is no devices with serial S-ATA connection in your computer, then parameter "Combined Mode option" should be given the "P-ATA only" value, otherwise "P-ATA+S-ATA" value should be preferred. Any changes of interface parameters will have effect on a list of registered devices not at once, but after reboot.

In order to set parameters of USB interface, page "Advanced" of BIOS Setup program should be opened, and there button "USB Configuration" should be pressed. A new page will be opened, where parameter "USB Function" (or else "USB Controller") must have the "Enabled" value. This is enough for access to USB devices by means of drivers (5.07-05). But those BIOS systems, which enable to connect bootable devices via USB bus, provide more optional parameters. In latest versions of AMI BIOS there is parameter "Legacy USB support"; it must be enabled, if you intend to connect bootable devices via USB bus. It should be reminded, that enabled state of "Legacy USB support" parameter may cause conflicts between BIOS and USB bus drivers (5.07-05), therefore USB bus drivers shouldn't be loaded in this case. If there are separate parameters for USB 2.0 controller in "USB Configuration" page, these parameters also should be given the "Enabled" value, and data transmission speed should be set to "HiSpeed".

As far as USB bus allows complex connection configurations, registration of bootable devices with USB interface may fail because of restrictions, imposed on BIOS startup tests. In order to remove these restrictions you have to open page "Boot" of BIOS Setup program, and there button "Boot setting configuration" should be pressed. A new page will be opened, where "Quick Boot" option should be disabled. Naturally, BIOS start tests will become about 3 seconds longer.

When BIOS system registers at least one storage device on USB bus, then in modern computers from page "USB Configuration" of BIOS Setup program you'll be able to open the next page "USB Mass Storage Device Configuration". In the latter page there is a list of all registered USB storage devices, and for each device the applied emulation method is shown. Inadequate emulation method specification can be one more reason of booting failure. By default emulation method is determined automatically, but errors may be caused by absence of media in the drive at the moment of test, by presence of unformatted media and even by media insertion just in course of tests.

Naturally, emulation method must correspond to class of device: for magnetic hard disk drives it should be defined as "Hard Disk", for external optical disc drives – as "CD-ROM", for external floppy drives – as "Floppy". But sometimes it isn't clear, which class of devices should correspond, for example, to a flash card. In such cases decision should be based on that flash cards with capacity 512 Mb and higher are always formatted as hard disks. For flash cards of smaller capacity a special emulation method may be applied – "Forced FDD", which means that the card will be presented by BIOS as "Big Floppy" irrespective to its actual format.

If you intend to subject a storage media to formatting procedure, then emulation method must correspond to the desired format type. Inaccurate emulation method specifications as "Forced FDD" and "Auto" may lead to unpredictable results. New unformatted storage devices are registered by BIOS, but are not given letter-names as logical disks. For their initial formatting modern programs should be preferred, for example, "Partition Magic" version 8.01 or higher. One of primary partitions must be

made active. After a reboot the new partitions will be given letter-names, and then the active partition may be made bootable, for example, by SYS.COM utility (6.24). Configurations with relocation of MS-DOS7 to a RAM-disk are not necessary, if the formatted media is indeed a hard disk.

Most BIOS systems wouldn't take control over those storage devices, which have media not emulated, but really formatted as "Big Floppy", i.e. without MBR. Normally such media are accessed by means of a driver, for example, ASPIDISK.SYS (5.07-03, 5.07-05). MBR can be written onto such media by utilities, mentioned in note 5 to article 6.13. After a reboot, BIOS will accept media with MBR as a HDD. Having been taken under BIOS control, such media can be treated as hard disk, can be formatted and made bootable.

When at least one storage device is recognized by BIOS system as a hard disk, then BIOS Setup program from its page "Boot" enables to open another page "Hard Disk Drives". There a list of all registered hard disk drives is shown. But only the first of these hard disk drives is regarded by BIOS as bootable device. If you want to boot the computer from an external storage device, registered as a hard disk drive, you should set this device in the first place in a list of drives in page "Hard Disk Drives" of BIOS Setup program. Just at once specification of the chosen storage device will appear in list of bootable devices on page "Boot Device Priority". There the chosen storage device must not necessarily be set in the first place, if devices with higher priority at that moment have no removable storage media inside.

Emulation method "Hard Disk" is suitable for storage devices with removable media, but with one required condition: the same removable media must be permanently present in this storage device since computer is switched on. Another removable disk in the same disk drive can't be read. If a storage device is emulated as hard disk, it will be assigned a letter-name of hard disk. But if a removable storage media isn't present in this device at the moment when computer is switched on, then this storage device will get no letter-name at all. This may be beneficial, if your USB adapter has several slots for different types of flash cards, and you don't want to spend letter-names for those types of cards, which certainly wouldn't be used.

Storage devices, emulated as "Floppy" or as "Forced FDD", get a letter-name irrespective of presence or absence of their removable media. Those modern BIOS systems, which were tested by the author, allotted to these storage devices letter-names A: and B: only, and didn't allow to exchange initially inserted removable media. If computer has several such devices, then letter-names are assigned to the first two only; the rest such devices are inaccessible. If you intend to boot your computer from a storage device, registered by BIOS as a floppy, then you have to set it the first in a list on page "Removable Drives" of BIOS Setup program.

Now flash card adapters and solid-state storage devices with USB interface have became widespread. Such devices can be used for loading MS-DOS7 and other operating

systems. For this purpose storage devices with USB interface must be taken under BIOS control, just as external hard disk drives. Emulation method "Hard disk" should be attempted first. If storage device becomes accessible, hence it is formatted as hard disk. Otherwise emulation method "Forced FDD" should be preferred. In the latter case you have to check a place of corresponding storage device in a list on page "Removable Drives" of BIOS Setup program. It must be set there either in the first or in the second place, otherwise BIOS wouldn't allot letter-name to this storage device, and then it can't be addressed to.

If storage device is recognized as a hard disk, you have to check whether its primary partition has status of active (bootable) partition. As far as "Partition Magic" program deals with real HDDs only, partition's status in "fake" HDDs, physically represented by solid-state storage devices, should be checked and changed, if necessary, by FDISK.EXE utility, being launched with parameters /fprmt and /actok (6.13). Many BIOS versions don't provide bootability from HDD partitions with FAT-12 file system, but FDISK.EXE inevitably marks with FAT-12 all 16 Mb and smaller partitions. If necessary, partition table may be read (9.02-02) and written back (9.02-03) with file system identifier changed from 01h to 06h (A.13-6). After that the small partition concerned should be formatted by FORMAT.COM utility with /z:1 parameter (6.15), and thus will get the desired FAT-16 file system.

Second stage of solid-state bootable media preparation includes boot sector writing and DOS system files copying by SYS.COM utility (6.24). After that a DOS loading configuration should be prepared. Preferable configurations are those with DOS relocation onto a RAM-disk (9.04, 9.09), because many types of solid-state storage devices are slow, and all types of such devices withstand a limited number of overwriting cycles. According to the desired configuration on solid-state media a directory structure should be formed, and directories should be filled with required files.

When bootable solid-state media is prepared, then corresponding storage device should be set in the first place either in a list on page "Hard Disk Drives" or in a list on page "Removable Drives" – it depends on which emulation method is applied. If necessary, at this stage emulation method "Hard Disk" may be changed to "Forced FDD". When name of corresponding storage device appears on page "Boot Device Priority" of BIOS Setup program, it should be set there in a place with highest priority relative to all other devices, which are ready for booting at that moment. Then you have to close BIOS Setup program, saving the latest settings. After computer's reboot a DOS loading process starts from the prepared solid-state media.

9.11-02 Windows-2000/XP as boot manager for MS-DOS7

When MS-DOS7 starts from a hard disk, it requires a primary partition with either FAT-16 or FAT-32 file system. Unlike MS-DOS7, operating systems Windows-2000/XP can be installed in both primary and non-primary partitions, formatted with either FAT-32 or NTFS file system. Neither combination should be regarded as hopeless. Even if the whole hard disk in your computer is formatted as one NTFS partition, you may release some disk's space for DOS partition with Partition Magic program, and then any appropriate boot manager (for example, System Commander) can arrange alternative loading of either selected operating system. The way described below is a more simple opportunity, using nothing but proprietary loader of Windows-2000/XP operating systems.

Partition structures, comprising partitions with FAT-32 (or FAT-16) file system, may be inherited from former operating system(s). Windows-2000/XP can be installed over Windows-95/98, either losing or preserving an opportunity to load previous operating system. If your computer after being switched on shows a boot menu with item "Previous operating system", and if this previous operating system is just Windows-95/98, hence partition structure on your disk is at least partially inherited. In this case for arranging alternative loading of MS-DOS7 you have to correct not the loading specifications of Windows-2000/XP, but preserved configuration files of previous Windows-95/98 operating system. Examples of such correction are shown in article 9.11-03.

If boot menu doesn't appear or doesn't contain a line "Previous operating system", then type of file system should be determined on that disk, which is claimed bootable in BIOS specifications. Most often it is disk C:. Having launched the Explorer program, you have to highlight the bootable disk, open context menu with right mouse's button, and choose in context menu the "Properties" item. A window will appear, where type of file system is shown. If this type is "NTFS", then MS-DOS7 can't be installed in that partition. But if file system type is "FAT", then MS-DOS7 can be installed. Installation will require BOOT.INI file records correction and copying of MS-DOS7 files onto that disk.

Officially prescribed path to correction of records in BOOT.INI file starts from menu "Start" and goes via item "Settings", via "Control Panel", via icon "Performance and Maintenance", via item "System", via button "Advanced", via button "Startup and Recovery Settings" to final button "Edit". In the same window a non-zero menu indication time should be set. Corrected version of BOOT.INI should be saved and after that the opened windows must be twice closed with OK button click. Besides that, contents of BOOT.INI file may be corrected by MSCONFIG.EXE utility. It can be launched from command line in a window, opened via item "Run" in menu "Start".

Syntax in BOOT.INI file is the same as that in files MSDOS.SYS (5.01-01) and CONFIG.SYS (9.04-01, 9.09-01). Each line presents a separate specification, which begins with a name, separated from its value with equality sign. Headers of file's sections are enclosed in square brackets. There are two sections in BOOT.INI file: the first

specifies loading parameters, the second presents a list of operating systems, which may be loaded. Here is an example of BOOT.INI file, enabling to load any of 3 operating systems:

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(3)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(3)\WINDOWS="Microsoft Windows XP...
multi(0)disk(0)rdisk(0)partition(2)\WINNT="Microsoft Windows 2000...
C:\bootsect.dos="Microsoft DOS 7.10"
```

In presented example the 5-th and the 6-th lines are truncated to size of page, in real file these lines are longer. But it doesn't matter: in any case the tails of truncated lines shouldn't be altered. From line's texts it becomes clear, that the shown BOOT.INI file enables to load Windows-XP and Windows-2000 operating systems, installed in the 3-rd and in the 2-nd partitions of a single hard disk, so that the first partition remains free. As far as the first partition is left free, it may be used for a separate installation of MS-DOS7. The last line in the shown example is just that line, which you have to add yourself in order to load MS-DOS7, in particular, from disk C:. The words, enclosed in double quotes, represent just a name of menu item and have no effect on loading process.

Lines of BOOT.INI file are interpreted by NTLDR loader. The latter will "understand" the line you have wrote as command to find a file-image of boot sector BOOTSECT.DOS in the root directory of disk C:. If BOOTSECT.DOS file isn't present there, it should be created anew under MS-DOS7, which may be loaded from diskette or other removable media. First, current boot sector should be saved into a file, as it is described in article 9.02-01. Then boot sector should be overwritten with SYS.COM utility (6.24). New boot sector similarly must be copied in a file, this time in a file named BOOTSECT.DOS. After that the former contents of boot sector must be restored from the previously saved file, also as it is recommended in article 9.02-01. The SYS.COM utility, used to overwrite boot sector, at the same time copies into the root directory system files COMMAND.COM and IO.SYS. Just the IO.SYS loader will be given control after execution of boot sector's code.

The next your task is to ensure, that all necessary files for MS-DOS7 can be found along their proper paths. In the root directory of bootable disk the following files must be present:

IO.SYS	– hidden system file: loader and the core of MS-DOS7;
MSDOS.SYS	– hidden system file: parameters of loading (5.01-01);
CONFIG.SYS	– configuration file (9.01-01, 9.04-01, 9.11-03);
AUTOEXEC.BAT	– configuration file (9.01-02, 9.04-02, 9.11-03);
COMMAND.COM	– a read-only file: command interpreter (6.04).

Three files from this list – MSDOS.SYS, CONFIG.SYS, AUTOEXEC.BAT – you have to compose yourself. Those articles, where composition examples can be found, are specified inside parentheses for each corresponding line in the list.

Besides root directory files, MS-DOS7 needs drivers, specified in lines of configuration files, and also various programs. The latter may be selected from those described in chapter 6. Drivers and programs should be stored inside a directories structure, which is to be arranged on the same disk. All examples of configuration files in this book are designed for the same directories structure: drivers are stored in \DOS\DRV directory, original MS-DOS7 files – in \DOS\MS7 directory, file manager – in \DOS\VC4 directory, all other files – in \DOS\OTH directory. You may arrange other directories structure, but in any case it must be exactly consistent with references in your configuration files.

Which particular configuration example should be followed – it is a matter of your choice. Sometimes the simplest version, shown in article 9.01, is quite sufficient. More often some version with DOS relocation to a RAM-disk should be preferred (9.04, 9.09). For experiments with other operating systems one more version is shown in article 9.11-03. You are free to choose and to compose those MS-DOS7 loading configurations, which are the most suitable for your own tasks.

Note 1: when over MS-DOS7 or over Windows-95/98 the Windows-2000/XP operating system is installed, the latter assigns attributes H (hidden) and S (system) to COMMAND.COM interpreter in the root directory. Therefore calls to command interpreter from batch files are not executed. The mentioned attributes should be taken off with ATTRIB.EXE utility (6.01).

9.11-03 MS-DOS7 as boot manager

Operating systems Windows-95/98 use MS-DOS7 as primary loader, but don't reveal opportunities of its separate configuration. Meanwhile it is possible and may be beneficial not only for MS-DOS7 itself, but also for alternative loading of those other operating systems, which are able to start under DOS.

Here a version of CONFIG.SYS file is suggested, which enables to load Windows-95/98, two configurations of MS-DOS7 and yet two other operating systems: QNX and Linux. The author has tested versions of QNX and Linux, requiring bootable disk with file system FAT-16. If you don't need anything more than a choice between Windows-95/98 and MS-DOS7, then all the lines related to QNX and Linux may be omitted, and file system on the bootable disk may be FAT-32 as well.

File CONFIG.SYS starts from section [menu]. A choice of an item in menu directs interpretation further to sections [L08] – [L25]: each section corresponds to one of

alternatives. Sections are named after corresponding labels in AUTOEXEC.BAT file. Empty lines between sections are inserted for convenience of perception only and may be omitted.

```
[menu]
numlock off
menuitem=L08, Real mode MS-DOS7
menuitem=L09, Protected mode MS-DOS7
menuitem=L16, Microsoft's Windows-98
menuitem=L24, QNX v.6.0
menuitem=L25, Linux Slackware v.3.5
menudefault=L16,20

[L08]
device=\DOS\DRV\Himem.sys /v
device=\DOS\DRV\Umbpci.sys
include=S08
include=S09

[S08]
accdate c- d- e-
dos=high,umb,noauto
buffershigh=30,0
fileshigh=30,0
lastdrivehigh=Z
multitrack=On
fcbshigh=1,0
stackshigh=9,256

[L09]
device=\DOS\DRV\Himem.sys /v
device=\DOS\DRV\Emm386.exe ram v
include=S08
include=S09

[S09]
country=007,866,C:\DOS\DRV\Country.sys
devicehigh=\DOS\DRV\Dblbuff.sys
devicehigh=\DOS\DRV\Ifshlp.sys
devicehigh=\DOS\DRV\Setver.exe
devicehigh=\DOS\DRV\Atapimgr.sys /W:6 /T:5 /LUN
devicehigh=\DOS\DRV\Oakcdrom.sys /D:CD001
installhigh=\DOS\DRV\Shsucdx.com /D:CD001 /L:N /~+ /R /Q
```

```
[L16]
device=\WINDOWS\Himem.sys
include=S08
Country=007,866,C:\WINDOWS\COMMAND\Country.sys
devicehigh=\WINDOWS\Dblbuff.sys
devicehigh=\WINDOWS\Ifshlp.sys
devicehigh=\WINDOWS\Setver.exe
devicehigh=\WINDOWS\COMMAND\Display.sys con=(ega,,1)

[L24]
device=\QNX\boot\bin\loadqnx.sys C:\QNX\boot\fs\qnxbas.ifs

[L25]
device=\DOS\DRV\Himem.sys
include=S08
install=\linux\loadlin.exe @\linux\linparam.scr

[common]
installhigh=\DOS\DRV\Mouse.com
shell=C:\COMMAND.COM C:\ /E:2016 /L:511 /U:255 /p
```

Sections [L08] and [L09] in the shown version of CONFIG.SYS file load MS-DOS7 as a separate operating system. Section [L09] provides ordinary type of access to UMB memory region – by means of EMM386.EXE driver (5.04-02), but section [L08] provides another type of access – by means of UMBPCI.SYS driver (5.04-04) without switching CPU to protected mode. The latter alternative is necessary for experiments with real mode programs, for example, with DUSE.EXE (5.07-05) and with GS_LIMIT.COM (9.10-01)

Each operating system should have its own directory tree. Presence of separate directory trees is not a required condition, but nevertheless is desirable: independence of directories contents makes multialternative loading more reliable.

Section [L16] for loading WINDOWS-95/98 includes a number of specifications, which usually are taken by default, but here are shown explicitly for the sake of consistency with separate loading of MS-DOS7. Paths in [L16] section correspond to ordinary directories structure, created automatically during installation of WINDOWS-95/98 onto a bootable disk. But if in your computer directories structure is different, all references in configuration files must be made corresponding to actual structure.

Choice of menu items [L24] and [L25] transfers control to loaders of UNIX-like operating systems, which don't return control back to MS-DOS7 loader. Therefore commands in section [common] will not be performed, so that the only way back is via SHUTDOWN command with following reboot. Paths in sections [L24] and [L25] reflect

directory structures, created for each UNIX-like operating system in course of unpacking their release packages.

Choice of menu items [L08], [L09] or [L16] enables to proceed to execution of commands in [common] section. In its last line the SHELL command transfers control to COMMAND.COM interpreter. After that the last stage of MS-DOS7 loading begins – interpretation of commands in configuration file AUTOEXEC.BAT.

As far as further processing of alternatives [L08] and [L09] is the same, the whole variety of alternatives shrinks to two, and AUTOEXEC.BAT file becomes relatively simple. Particular contents of AUTOEXEC.BAT file may look, for example, as

```
@echo off
prompt $p$g
set dsk=C:
if not exist %dsk%\Temp\nul md %dsk%\Temp
set Temp=%dsk%\Temp
set dircmd= /A /O:GNE /P
goto %CONFIG%
:L08
:L09
Lh %dsk%\DOS\DRV\Keyrus.com
path ;
set VC=%dsk%\DOS\VC4
path=%VC%;%dsk%\DOS\OTH;%dsk%\;%dsk%\DOS\MS7
Vc.com /TSR /no2E /noswap
goto L25
:L16
path=%dsk%\WINDOWS;%dsk%\WINDOWS\COMMAND
Mode.com con codepage prepare=((866) %dsk%\WINDOWS\COMMAND\Ega3.cpi)
Mode.com con codepage select=866
Lh Keyb.com ru,866,%dsk%\WINDOWS\COMMAND\Keybrd3.sys
echo.
echo Loading Windows-98. Wait...
Win.com
:L24
:L25
```

In this version of AUTOEXEC.BAT file the lines 2 – 6 represent common part, assigning values to ordinary environmental variables. It's important not to leave spare spaces at the ends of 3-rd and 5-th lines, where values are assigned to variables DSK and TEMP. In the 7-th line a jump occurs to the label, defined by value of CONFIG environmental variable. This value is just the code of selected menu item, assigned implicitly by IO.SYS loader during interpretation of [menu] section of CONFIG.SYS file.

As far as execution of AUTOEXEC.BAT file is reached after alternatives L08, L09, L16 only, a jump in the 7-th line can be directed to labels :L08, :L09 and :L16.

Labels :L08 and :L09 are followed by a group of final commands for loading MS-DOS7. Commands of this group define paths, specific for MS-DOS7, and launch the Volcov Comander file manager.

Label :L16 is followed by another group of commands, presenting final operations for loading Windows-95/98 system. Here the PATH variable gets another paths, specific for Windows-95/98. Attention should be paid to usage of trivial national adaptation method, which enables proper switching of national codepages inside "DOS box" of Windows OS. In final lines the Windows GUI loader – file WIN.COM – is called for. Windows logotype will not be displayed, a textual message is displayed instead. When GUI loading is completed, this message becomes hidden under customary Windows desktop.

Note 1: though Windows-XP OS is not devised for being started under MS-DOS7, the required initial start conditions can be prepared by free utility Dostowxp.com. Its recent version, modified by V.Ashumov, can initiate loading of Windows Vista and Windows-7. Both original and modified versions of this utility are available at http://www.multiboot.ru/files.htm . Thus MS-DOS7 is enabled to initiate launching of pre-installed modern Windows OS versions similarly to other OS launching examples, shown in article 9.11-03.

Note 2: installation of Linux OS is performed by recompiling its core with a specific set of drivers, required by hardware of a particular computer. Internet server ftp://ftp.wolfmountaingroup.org/pub/linuxware/ presents software package linuxware-09072008.tar.gz enabling to recompile the core 2.4 of Linux OS into a form of DOS's ordinary application. This recompiled core starts Linux under DOS, preserves DOS's structure intact and returns to DOS after shutdown. Core 2.4 is used in Mandrake Linux versions 8 – 10 and in many other modern clones of Linux OS.

Appendixes

A.01 PC's main data structures

Both BIOS and DOS store their important data in especially devoted areas of computer's memory. Data placement inside these areas is not fixed for ever and may depend on version of BIOS and DOS. Therefore data in system structures should not be addressed directly, but rather should be accessed via special functions, described in chapter 8 of this book. One more reason is that data can't be updated properly unless the corresponding service function is called for.

Nevertheless direct access to system data structures may be necessary. It enables to get more information than you are allowed to know via service functions. For debugging purposes you may need to see data "as they were", without being updated. You may need to intervene, to change certain settings in order to provoke desirable consequences. Of course, each such action is done exclusively at your own risk, but it may give you a chance, which otherwise would be lost.

A.01-1. BIOS data area

Just when computer is switched on its BIOS system begins to gather data and arrange its data area. In AT-compatible computers the BIOS data area occupies 100h bytes at 0040:0000h – 0040:00FFh. The table below gives general disposition of selected data items with references to separate data tables for floppy drives (A.08-1), video system (A.10-6), keyboard (A.02-3) and other hardware (A.11-1).

Offset	Size	Description
00h	2	Port COM-1 base I/O address
02h	2	Port COM-2 base I/O address
08h	2	Port LPT-1 base I/O address
0Eh	2	Auxiliary BIOS data segment (0000h if absent)
10h	2	Installed hardware word (A.11-1)
12h	1	Status of POST self-test
13h	2	Base memory size in kilobytes
17h	39	Keyboard's buffer and flags (A.02-3)
3Eh	7	Floppy drive status registers (A.08-1)
49h	22	Current video mode data (A.10-6)

Continuation of table A.01-1

67h	4	Restart address after CPU reset (note 4 to A.12-1)
6Ch	4	Timer ticks, counted since midnight
70h	1	Count of days, reset after INT 1A\AH=00h call
71h	1	Bit 7 set after Ctrl-Break keystroke
72h	2	Prescribed action of POST test (note 1)
74h	1	HDD's last operation error code (A.06-01)
75h	1	Number of hard disk drives
77h	1	Hard disk drive I/O port address
78h	1	Port LPT-1 timeout counter
7Ch	1	Port COM-1 timeout counter
7Dh	1	Port COM 2 timeout counter
80h	4	Keyboard buffer's start and end offsets (A.02-3)
84h	8	Video control registers (A.10-6)
8Ch	3	HDD controller's status registers
8Fh	7	Floppy drive controller's information (A.08-1)
96h	2	Keyboard's status bytes (A.02-3)
98h	4	Pointer to wait-complete flag (INT 15\AX=8300h)
9Ch	4	Timer's wait count in microseconds
A0h	1	System timer's flags: bit 0: a call for INT 15\AH=86h has occurred bit 7: waiting time has elapsed
CEh	2	Count of days since last boot
F0h	16	Intra-application communication area

Note 1: after a reboot, initiated by a jump far to F000:FFF0h address (note 4 to A.12-1), POST test performance depends on contents, preserved in 0040:0072h memory cell:

0000h – "cold" boot (full POST with memory test)

1234h – "warm" boot (bypass memory test)

Note 2: since data disposition in BIOS data area may depend on BIOS version, it is for you to decide, whether the data item at any particular offset is indeed the one you expect to find there.

A.01-2 Selected entries in DOS' list-of-lists

List-of-lists is the basic data structure, created by IO.SYS loader at start of DOS loading procedure. A pointer to first byte of DOS's list-of-lists is returned by INT 21\AH=52h function (8.02-59). Fig.10 (in appendix A.03-3) illustrates process of access to list-of-lists. Selected information about data items in DOS's list-of-lists is given in the following table.

Offset	Size	Description
– 02h	2	Segment address of the first MSB (A.12-7)
00h	4	Pointer to the first DPB block (note 1)
04h	4	Pointer to SFT (System File Table, A.01-4)
0Ch	4	Pointer to active CON (Console) device driver
10h	2	Maximum bytes per sector on any drive
16h	4	Pointer to first record in CDS structure (A.03-3)
20h	1	Number of registered drives (block devices)
21h	1	Number of records in CDS structure (A.03-3)
22h	18	Actual NUL device driver header (note 2)
34h	1	Number of virtual drives created by JOIN.EXE
37h	4	Pointer to SETVER's table (0000:0000h if none)
3Dh	2	PSP segment of the last executed program
43h	1	Boot drive (01h = A:, 03h = C:, and so on)
45h	2	Extended memory size (in kilobytes).

Note 1: DPB blocks (Drive Parameter Blocks, A.03-1) are arranged in a chain, so that a pointer to the next DPB is at offset 19h in the previous DPB.

Note 2: NUL device driver header at offset 22h is the first in a chain of driver headers. First dword in each header is a pointer to the next header. Another way to obtain a pointer to the second driver header is via INT 2F\AX=122Ch. Last header in the chain begins with word FFFFh.

A.01-3. Selected items in DOS's swappable data area

Current address and size of DOS's swappable data area (SDA) are returned by INT 21\AX=5D06h function (8.02-70). SDA stores continuously updated data, including main DOS's system stacks. These data reflect ever changing DOS's status at each current moment. Total size of swappable area may be as large as several kilobytes, it depends on size of the stacks.

The SDA area is named "swappable", because its swapping is the main mechanism of obtaining DOS's re-enterability, i.e. opportunity to call DOS's functions by TSRs and handlers, which themselves may be called while execution of previous DOS's function hasn't been finished. If interrupt handler reveals, that either critical error flag (at offset 00h in SDA) or InDOS flag (at offset 01h in SDA) have a non-zero value, hence a call for this handler has interrupted execution of a DOS's function, and then the next call for any DOS's function can change data in SDA area so that a return back to proper resumption of the interrupted program will become impossible. Though sometimes resumption of interrupted program is possible (8.02-28, 8.02-87), nevertheless the most radical solution is to save contents of SDA before any DOS's function is called, and to restore SDA state

afterwards on basis of the saved data. It also should be taken into account, that INT 21\AX=5D06h function itself (8.02-70) is non-reentrant; therefore a call for SDA's address should be performed beforehand, during handler's initialization, so that at the moment of interrupt the handler may read a prepared SDA address without a new call for INT 21\AX=5D06h.

The table below shows selected data items inside swappable data area

Offset	Size	Description
00h	1	Critical error flag ("ErrorMode")
01h	1	InDOS flag (INT 21\AH=34h, 8.02-28)
02h	1	Drive, caused critical error, or FFh if no error
03h	1	Locus of the last error (A.06-4)
04h	2	Extended code of the last error (A.06-1)
06h	1	Suggested action after the last error (A.06-3)
07h	1	Class of the last error (A.06-2)
08h	4	ES:DI pointer at the moment of last error
0Ch	4	Current address of DTA (8.02-16)
10h	2	Current program's PSP segment (process identifier)
14h	2	Errorlevel value from last program's termination
16h	1	Current logical disk number
17h	1	The BREAK flag (3.01, 4.02, 8.02-25)
2Ah	1	Flag of FAIL response to INT 24 (8.02-84) call
2Bh	1	Allowable INT 24 actions (8.02-84)
30h	1	Day of month
31h	1	Month
32h	2	Year, counted from 1980
34h	2	Number of days since 01-01-1980
36h	1	Day of week (0 = Sunday)

A.01-4 SFT entry structure

Associations between access handles and the corresponding "opened" objects are stored by DOS in a chain-like queue of tables, named SFT (System File Tables). This name is not quite correct, because "opened" objects may be not files only, but also allotted XMS areas, driver's access channels and other objects, known as "character devices".

A pointer to the first SFT is at offset 04h in the list-of-lists (A.01-2). Each SFT begins at offset 00h with a dword pointer to the next SFT, except the last one, which is marked with word FFFFh at offset 00h. Such structure can easily be traced through from the first

SFT to the last. A word at offset 04h in each SFT declares number of entries in this particular SFT. Each entry corresponds to one "opened" object. Total number of SFT entries is limited by FILES specification in CONFIG.SYS file (4.12).

Numbers of SFT entries, which are "opened" to a given program, are stored in JFT at offset 18h inside PSP (A.07-1), belonging to this program. The corresponding "opened" objects are addressed to with handles (INT 21\AH=3Dh, 8.02-33), which identify SFT entries according to disposition order of their numbers in JFT. The number of SFT entry, corresponding to a given handle, can be found also with INT 2F\AX=1220h (8.03-11). Then INT 2F\AX=1216h function (8.03-09), being given the SFT entry number, returns a pointer to this entry. Some information about active SFT entries and corresponding objects can be obtained via INT 21\AX=4400h function (8.02-40).

The first three SFT entry numbers have fixed default associations: 00h – AUX channel (COM1 port), 01h – the CON device (console), 02h – the PRN device (LPT1 port). These 3 SFT entry numbers are automatically written into JFT table of each program. Order of their disposition in JFT (01h, 01h, 01h, 00h, 02h) defines associated handles: 0000h – for STDIN channel, 0001h – for STDOUT channel, 0002h – for STDERR channel, 0003h – for COM1 port, 0004h – for LPT1 port. SFT entries for handles 0005h and on are created on requests by INT 21\AH=3Dh (8.02-33) and by INT 21\AX=6C00h (8.02-78) functions.

The first entry in each SFT table starts at offset 06h. As far as each entry has fixed length 3Bh, start points of the following entries can be easily calculated. The table below specifies selected data items inside each entry with offsets counted from start point of that entry. The first column "CDE" of the table corresponds to "character device" entries, the second column "OFE" – to entries, associated with ordinary files.

CDE	OFE	Size	Description
00h	00h	2	Number of handles to the object (FFFFh if none)
02h	02h	1	Access and sharing conditions (A.09-4)
	03h	1	Flags (note 1)
	04h	1	File's attributes (A.09-2)
05h	05h	2	Object's info word (A.04-2 for files, A.05-2 for others)
07h	07h	4	Pointer to DPB (A.03-1) for files or pointer to driver's header (A.05-1) for other objects
	0Bh	2	Starting cluster number (for local files only)
	0Dh	2	File's packed time (INT 21\AX=5700h, 8.02-63)
	0Fh	2	File's packed date (INT 21\AX=5700h, 8.02-63)
	11h	4	File's size
	15h	4	File's pointer position (INT 21\AH=42h, 8.02-38)
	19h	2	Relative number of the last cluster accessed
19h		4	Pointer to IFS redirector records

Continuation of table A.01-4

	1Bh	4	Number of directory sector containing file's entry
	1Fh	1	Number of file's entry in directory sector
20h	20h	11	Name in FCB-format (A.09-5) of file or other object
	31h	2	Identifier (PSP segment) of file's owner process
	35h	2	Absolute number of the last accessed cluster
37h		4	Pointer to IFS-driver or 0000:0000h for ordinary files

Note 1: flags byte at 03h includes all BH register settings, specified for INT 21\AX=6C00h function (8.02-78). Besides this, set state of bit 7 in flags byte marks files opened by means of FCB.

A.02 Keyboard codes and national adaptation

A.02-1 Key codes

User's control over PC via keyboard must be enabled always, even when operating system isn't loaded yet. Therefore motherboard's hardware and BIOS system are responsible for compatibility with different types of keyboards. Just when PC is switched on, its BIOS system loads INT 09 and INT 16 handlers, providing various keyboard functions.

Keyboard signals are received and transformed by keyboard controller. It responds to each change of key's state (press or release) with preparing code for port 60h and sending a request via IRQ 01 interrupt request line. Request invokes the INT 09 handler. Some keys induce keyboard controller to send a succession of codes, so that INT 09 handler is invoked several times. The code, read by INT 09 handler from port 60h, is most often the scan code of some key, but it may be a service code. Codes from E0h to FFh, and also 00h and AAh are interpreted as service codes. Service codes, charged with a special keystroke identification mission, are known as prefixes:

E0h – prefix for discrimination between those keys, which for the sake of compatibility with obsolete 84-key keyboards have been given identical scan codes.

E1h – prefix for keys having a two byte scan code. In ordinary keyboards there is only one such key: Pause/Break (note 6 to table A.02-1).

Reading of a prefix from port 60h makes INT 09 handler prepared to a specific interpretation of that scan code, which will be received by the next call for INT 09. In the second column (INT 09) of the table below hexadecimal scan codes of keystrokes are shown, read by INT 09 handler from port 60h. Key release codes are not shown in the table, because release codes can be easily derived from keystroke codes by making its 7-th bit set. For example, key "A" sends keystroke (press) code 1Eh, hence its release code is 9Eh. But those scan codes, which are sent preceded by a prefix either E0h or E1h, are

shown in the second table's column together with this prefix. Naturally, release codes of the same keys are preceded by the same prefix.

"Raw" scan code, read from port 60h, is translated by INT 09 and INT 16 handlers into a new pair – a unified scan code and ASCII value, corresponding to the pressed (or released) key. Just this data pair will be presented to program, sending a request for keyboard input via INT 16. Unified scan-code most often is equal to press scan code, but may be altered, if at the same time a "functional" key is kept pressed: SHIFT, CTRL, or ALT. Each "functional" key has its own scan code (in second column of the table below). Scan codes of "functional" keys are taken into account by INT 09 handler, but are not stored in keyboard buffer. States of "functional" keys are expressed otherwise: via a status word, returned by INT 16\AH=12 function (8.01-85).The following table represents keys of the most widely used 104-key "enhanced" keyboard. The word "num" before a key name in the first table's column denotes keys of a numerical keypad at the right side of keyboard. The shown codes for such keys correspond to turned OFF state of NUMLOCK switch (note 6). Data order is defined by key's scan codes values in the second table's column.

Hexadecimal numbers in columns 3 – 6 of the table below represent the data, returned in AX register by INT 16\AH=10h function (8.01-83). Left two digits in each number define unified scan-code, returned in AH register, and the right two digits – ASCII code of corresponding character, returned in AL register. Data in the 3-rd column (AX) correspond to keystrokes, not accompanied by keeping pressed any "functional" key. Data in the 4-th column (SHIFT) correspond to the case when SHIFT key is kept pressed, data in the 5-th column (CTRL) – to the case of keeping pressed the CTRL key, data in the 6-th column (ALT) – to the case of keeping pressed the ALT key. Blank space in place of any particular value means that corresponding key or key combination is sensed by BIOS, but isn't reported via INT 16 handler.

Keys		INT09	AX	SHIFT	CTRL	ALT	Comments
Esc		01	011B	011B	011B	0100	note 1
1	!	02	0231	0221		7800	
2	@	03	0332	0340	0300	7900	
3	#	04	0433	0423		7A00	
4	$	05	0534	0524		7B00	
5	%	06	0635	0625		7C00	
6	^	07	0736	075E	071E	7D00	
7	&	08	0837	0826		7E00	
8	*	09	0938	092A		7F00	
9	(	0A	0A39	0A28		8000	
0	)	0B	0B30	0B29		8100	
–	_	0C	0C2D	0C5F	0C1F	8200	

Continuation of table A.02-1

= +	0D	0D3D	0D2B		8300	
Backspace	0E	0E08	0E08	0E7F	0E00	note 1
Tab	0F	0F09	0F00	9400	A500	notes 1, 2
Q	10	1071	1051	1011	1000	
W	11	1177	1157	1117	1100	
E	12	1265	1245	1205	1200	
R	13	1372	1352	1312	1300	
T	14	1474	1454	1414	1400	
Y	15	1579	1559	1519	1500	
U	16	1675	1655	1615	1600	
I	17	1769	1749	1709	1700	
O	18	186F	184F	180F	1800	
P	19	1970	1950	1910	1900	
[{	1A	1A5B	1A7B	1A1B	1A00	note 1
] }	1B	1B5D	1B7D	1B1D	1B00	note 1
Enter	1C	1C0D	1C0D	1C0A	1C00	note 1
num Enter	E0 1C	E00D	E00D	E00A	A600	notes 1, 3
Left Ctrl	1D					note 4
Right Ctrl	E0 1D					note 4
A	1E	1E61	1E41	1E01	1E00	
S	1F	1F73	1F53	1F13	1F00	
D	20	2064	2044	2004	2000	
F	21	2166	2146	2106	2100	
G	22	2267	2247	2207	2200	
H	23	2368	2348	2308	2300	
J	24	246A	244A	240A	2400	
K	25	256B	254B	250B	2500	
L	26	266C	264C	260C	2600	
; :	27	273B	273A		2700	note 1
' "	28	2827	2822		2800	note 1
` ~	29	2960	297E		2900	note 1
Left Shift	2A					note 4
SysRq	E0 2A			7200		note 5
\ |	2B	2B5C	2B7C	2B1C	2B00	note 1
Z	2C	2C7A	2C5A	2C1A	2C00	
X	2D	2D78	2D58	2D18	2D00	
C	2E	2E63	2E43	2E03	2E00	
V	2F	2F76	2F56	2F16	2F00	
B	30	3062	3042	3002	3000	
N	31	316E	314E	310E	3100	
M	32	326D	324D	320D	3200	
, <	33	332C	333C		3300	note 1

Continuation of table A.02-1

. >	34	342E	343E		3400	note 1
/ ?	35	352F	353F		3500	note 1
num /	E0 35	E02F	E02F	9500	A400	notes 1, 2, 3
Right Shift	36					note 4
num *	37	372A	372A	9600	3700	note 1, 2
Left Alt	38					note 4
Right Alt	E0 38					note 4
Spacebar	39	3920	3920	3920	3920	
Caps Lock	3A					note 4
F1	3B	3B00	5400	5E00	6800	
F2	3C	3C00	5500	5F00	6900	
F3	3D	3D00	5600	6000	6A00	
F4	3E	3E00	5700	6100	6B00	
F5	3F	3F00	5800	6200	6C00	
F6	40	4000	5900	6300	6D00	
F7	41	4100	5A00	6400	6E00	
F8	42	4200	5B00	6500	6F00	
F9	43	4300	5C00	6600	7000	
F10	44	4400	5D00	6700	7100	
NumLock	45					note 4
Pause	E1 1D 45					notes 4, 6
ScrollLock	46					note 4
num 7	47	4700	4737	7700	0007	note 7
Home	E0 47	47E0	47E0	77E0	9700	note 1, 3
num 8	48	4800	4838	8D00	0008	notes 2, 7
Arrow Up	E0 48	48E0	48E0	8DE0	9800	notes 1, 2, 3
num 9	49	4900	4939	8400	0009	note 7
PgUp	E0 49	49E0	49E0	84E0	9900	notes 1, 3
num –	4A	4A2D	4A2D	8E00	4A00	notes 1, 2
num 4	4B	4B00	4B34	7300	0004	note 7
LeftArrow	E0 4B	4BE0	4BE0	73E0	9B00	notes 1, 3
num 5	4C	4C00	4C35	8F00	0005	notes 2, 7
num 6	4D	4D00	4D36	7400	0006	note 7
RightArrow	E0 4D	4DE0	4DE0	74E0	9D00	notes 1, 3
num +	4E	4E2B	4E2B	9000	4E00	notes 1, 2
num 1	4F	4F00	4F31	7500	0001	note 7
End	E0 4F	4FE0	4FE0	75E0	9F00	notes 1, 3
num 2	50	5000	5032	9100	0002	notes 2, 7
ArrowDown	E0 50	50E0	50E0	91E0	A000	notes 1, 2, 3
num 3	51	5100	5133	7600	0003	note 7
PgDn	E0 51	51E0	51E0	76E0	A100	notes 1, 3
num 0	52	5200	5230	9200		notes 2, 7

Continuation of table A.02-1

Ins	E0 52	52E0	52E0	92E0	A200	notes 1, 2, 3
num .	53	5300	532E	9300		notes 2, 7
Del	E0 53	53E0	53E0	93E0	A300	notes 1, 2, 3
F11	57	8500	8700	8900	8B00	note 8
F12	58	8600	8800	8A00	8C00	note 8
LeftWin	E0 5B	B6E0	C2E0	CEE0	DAE0	note 8
RightWin	E0 5C	B7E0	C3E0	CFE0	DBE0	note 8
Menu	E0 5D	B8E0	C4E0	D0E0	DCE0	note 8

Note 1: the INT16\AH=00h function doesn't respond to this keystroke when "functional" key ALT is kept pressed.

Note 2: the INT16\AH=00h function doesn't respond to this keystroke when "functional" key CTRL is kept pressed.

Note 3: the INT16\AH=00h function returns 00h instead of ASCII code E0h, except for two keystrokes: after "num /" it returns ASCII code 35h, after "num Enter" keystroke it returns ASCII code 1Ch.

Note 4: code of this key is not written into keyboard buffer, but it does affect translation of other key's codes by INT 09 handler.

Note 5: keyboard controller responds to SysRq keystroke with "E0 2A E0 37" codes succession, and to SysRq key release – with inverse succession "E0 B7 E0 AA". Some INT 16 handlers may return other response to CTRL-SysRq key combination.

Note 6: release of Pause/Break key is not registered separately. After each Pause/Break keystroke it's press code is immediately followed by release code, forming a succession "E1 1D 45 E1 9D C5". Having received such succession of codes, the INT 09 handler resets keyboard buffer and calls for INT 1B (8.01-95).

Note 7: the shown codes for this key correspond to turned OFF state of Numlock switch. When NumLock switch is turned ON, codes shown in 3-rd and in 4-th columns of the table get exchanged.

Note 8: the INT 16\AH=00h function gives no response to these keys.

Note 9: several models of keyboards have three auxiliary keys for power control: "Power", "Sleep" and "Wake Up". Corresponding scan codes for these keys are E0 5E, E0 5F, E0 63.

A.02-2 Keyboard layouts and national codepages

The following table comprises data for MS-DOS7 national adaptation by those means, which are supplied in Microsoft's Windows-95/98 release. These means include data file COUNTRY.SYS, three files with keyboard layouts (KEYBOARD.SYS, KEYBRD2.SYS and KEYBRD3.SYS), and four files with fonts for different codepages (EGA.CPI, EGA2.CPI, EGA3.CPI and ISO.CPI).

The first column (Abbr) in this table contains literal country codes, the third column (ID) – keyboard layout identifiers. Both these items are needed for KEYB.COM driver's (5.02-04) command line composition. Layout identifier is necessary for those countries only, where more than one keyboard's layout is used, for other countries it may be omitted. The 4-th column (Keyb) of the table specifies which file with keyboard layout should be loaded: digit 1 corresponds to KEYBOARD.SYS, digit 2 – to KEYBRD2.SYS, digit 3 – to KEYBRD3.SYS, word "Any" – to either of these three files.

The 5-th column (Code) of the table shows numerical country code, used for loading COUNTRY.SYS data file (5.02-01) with COUNTRY command (4.05).

Last 7-th column of the table shows codepages used in various countries. Number of a codepage is needed for MORE.COM utility (6.18), which has to select one font from a group of fonts in each *.CPI file (example – in 9.01-02). The ISO.CPI file supplies fonts, recommended by International Standards Organization. Proprietary Microsoft's fonts are in EGA*.CPI files, about 5 fonts in each. Therefore 6-th table's column (Ega*) specifies which one of EGA*.CPI files should be loaded: digit 1 corresponds to file EGA.SYS, digit 2 – to EGA2.CPI, digit 3 – to EGA3.CPI, word "Any" – to either of these files.

Abbr	Country	ID	Keyb	Code	Ega*	Codepage
GR	Austria		Any	043	Any	CP850
BE	Belgium		1,2	032	Any	CP850
BG	Bulgaria	442	2	359	3	CP855
BR	Brazil	274, 275	1,2	055	Any	CP850
CF	Canada French	058	1,2	002	1	CP863
CZ	Czech Republic	243	Any	042	Any	CP852
DK	Denmark		1,3	045	1	CP865
SU	Finland		Any	358	Any	CP850
FR	France	120, 189	1,3	033	Any	CP850
GR	Germany		Any	049	Any	CP850
GK	Greece	319	2	030	2	CP737, 869
HU	Hungary		Any	036	Any	CP852
IS	Iceland	161	2	354	2	CP861
IT	Italy	141,142	Any	039	Any	CP850
LA	Latin America		1	003	Any	CP850
NL	Netherlands		1,3	031	Any	CP850
NO	Norway		1,2	047	1	CP865
PL	Poland		Any	048	Any	CP852
PO	Portugal		1	351	1	CP860
RO	Romania	333	2	040	Any	CP852
RU	Russia	441	2,3	007	3	CP866
SL	Slovakia	245	Any	421	Any	CP852

Continuation of table A.02-2

SP	Spain		1,3	034	Any	CP850
SV	Sweden		Any	046	Any	CP850
SF	Switzerland French		1,3	041	Any	CP850
TR	Turkey	440, 179	2	090	2	CP857
UK	Britain + Ireland	166, 168	Any	044	Any	CP850
US	USA + Australia		Any	001	1,3	CP437
YC	Yugoslavia Cyrillic	118	2	038	3	CP855
YU	Yugoslavia Latin	234	Any	038	Any	CP852

Note 1: KEYBOARD.SYS is the only file, which supports typewriter mode of keyboard layout.

Note 2: Microsoft's files for national adaptation are not compatible with KEYRUS.COM driver (5.02-05). The latter uses internal code tables and keyboard layouts.

Note 3: fonts for some other countries (China, Israel, Japan, etc.) are supplied exclusively with special national versions of Microsoft's operating systems.

A.02-3 Keyboard data fields in BIOS data area

The following table shows disposition of keyboard data in BIOS data area. All offsets are given relative to segment address 0040h, where BIOS data area starts.

Offset	Size	Description
17h	2	Flags, returned in AX by INT 16\AH=12h (8.01-85)
19h	1	Character input via ALT followed by ASCII code
1Ah	2	Pointer to the next character in keyboard buffer
1Ch	2	Pointer to the first free cell in keyboard buffer
1Eh	32	Keyboard's circular buffer
71h	1	Flag: bit 7 set if Ctrl-Break has been pressed
80h	2	Keyboard buffer's start offset (normally 1Eh)
82h	2	Keyboard buffer's END+1 offset (normally 3Eh)
96h	1	Flags: bit 0 set: last code read was E1h prefix bit 1 set: last code read was E0h prefix bit 2 set: right CTRL key has been pressed bit 3 set: right ALT key has been pressed bit 4 set: "enhanced" keyboard is installed bit 6 set: 1-st byte is received of 2-byte scan-code
97h	1	Status: bit 0 set: the Scroll Lock LED is switched ON bit 1 set: the Num Lock LED is switched ON bit 2 set: the Caps Lock LED is switched ON bit 7 set: keyboard has sent error flag

Note 1: presented data disposition may depend on BIOS version (A.01-1).

A.02-4 National adaptation parameters block

Data block with currently active national adaptation parameters is returned by INT 21\AX=6501h function (8.02-74). Data block of the same structure is accepted by INT 21\AX=7002h function (note 3 to 8.02-74), defining national adaptation for MS-DOS7.

Offset	Size	Description
00h	1	= 01h on return (note 1)
01h	2	Table size on return (note 1)
03h	2	Country code in hexadecimal form (A.02-2)
05h	2	Hexadecimal codepage number (A.02-2)
07h	2	Date format: = 0000h – American (mm dd yy) = 0001h – European (dd mm yy) = 0002h – Japanese (yy mm dd)
09h	5	ASCII currency name, ending with 00h byte
0Eh	2	Thousands separator for numbers
10h	2	Integer and fractional parts separator
12h	2	ASCII date separator character
14h	2	ASCII time separator character
16h	1	bit 0 set: currency symbol follows value (else precedes) bit 1 set: space between value and currency symbol bit 2 set: currency symbol replaces decimal point
17h	1	Number of digits after decimal point in currency
18h	1	bit 1 set: 24-hour clock, else 12-hour clock
19h	4	Entrance address of case map routine (note 2)
2Dh	2	ASCII data-list separator character

Note 1: when data block is sent to INT 21\AX=7002h function, then this item is ignored.
Note 2: case map routine translates national characters (with ASCII codes larger than 80h) into upper case and back. The case map routine should be called for with CALL FAR command (7.03-08). AL register is used for both sending ASCII code of the character to be translated and for returning the result back.

A.02-5 Country-dependent restrictions for filenames

A pointer to this table is returned by INT 21\AX=6505h function (note 1 to 8.02-74).

Offset	Size	Description
00h	2	Table's size (this word shouldn't be counted)
03h	1	Lowest permissible character value for filenames
04h	1	Highest permissible character value for filenames
06h	1	First character's value of prohibited range
07h	1	Last character's value of prohibited range
09h	1	Number ("N") of filenames terminating codes
0Ah	N	ASCII codes used to terminate filenames

A.02-6 Information about available code pages

The following table shows structure of DISPLAY.SYS (5.02-02) driver's data block; a pointer to this block is returned by INT 2F\AX=AD03h function (8.03-27).

Offset	Size	Description
00h	2	number M of codepages, specified by configuration
04h	2	number N of codepages, loaded by default
06h	2N	identifiers of codepages, loaded by default
06h+2N	2M	identifiers of codepages, specified by configuration (or = FFFFh if configuration isn't prepared yet)

A.02-7 Definition of "hot" keys in AMIS specification

Common practice for TSR programs is assignment of functions to predetermined "hot" keys irrespective to which keys have been charged yet with other missions by previously loaded TSR programs. The least harmful outcome of this practice is a loss of opportunities to invoke functions of previously loaded TSR programs and drivers. A real chance to prevent interception of "hot" key functions is suggested by AMIS specification (A.07-6). According to AMIS specification, resident modules must respond to calls for multiplex interrupt INT 2D with operation code AL = 05h, returning in DX:BX registers a pointer to a list of their active "hot" keys. Any program, which intends to arrange its own "hot" keys, should be given access to data about previous "hot" key assignments.

The first byte, at offset 00h from the start of "hot" keys list, informs about "hot" calls interception method (note 1). The second byte at offset 01h is a number of "hot" keys, kept active by responding resident module. This number also defines total length of the returned list, because after the second byte, starting at offset 02h, a group of "hot" key descriptors follows, each 6 bytes long. Structure of these descriptors is shown in table below. Offsets in the table are counted from start of each descriptor.

Offset	Size	Description	Comments
00h	1	Scan-code of the "hot" key	Note 2
01h	2	Required shift states	Note 3
03h	2	Disallowed shift states	Note 4
05h	1	Auxiliary key's flags	Note 5

Note 1: byte at offset 00h in "hot" keys list informs about "hot" calls interception method. Bit 7 in this byte must be cleared; other bits have the following meaning:

bit 0 set: – interception before INT 09 handler
bit 1 set: – interception after INT 09 handler
bit 2 set: – interception before INT 15\AH=4Fh
bit 3 set: – interception after INT 15\AH=4Fh
bit 4 set: – interception of INT 16\AH=00h,01h,02h calls
bit 5 set: – interception of INT 16\AH=10h,11h,12h calls
bit 6 set: – interception of INT 16\AH=20h,21h,22h calls.

Note 2: if the most significant bit of scan-code is clear, hence actuation is registered when the key is pressed; if the most significant bit of scan code is set, hence actuation is registered when the key is released. If actuation is caused exclusively by a specific states combination of "functional" keys, then 00h or 80h values should be specified instead of scan code.

Note 3: a word of required shift states is almost identical to the word of keyboard flags, returned by INT 16\AH=12h function (8.01-85). The only difference is the meaning of bit 7: in a word of required shift states it corresponds to keeping pressed either (left or right) SHIFT key. Set state of any bit in a word of required shift states specifies a necessary condition for "hot" key actuation.

Note 4: bits in a word of disallowed shift states have the same meaning, as in a word of required shift states (note 3), but their set state expresses the opposite condition: prevention of "hot" key actuation. Combination of required and disallowed conditions helps to decrease probability of false actuations.

Note 5: the last byte in each "hot" key descriptor is a byte of auxiliary flags. Bits 6 and 7 in this byte must be cleared; states of other bits have the following meaning:

bit 0 set: – actuation before module's execution
bit 1 set: – actuation after module's execution
bit 2 set: – monitoring interception is allowed
bit 3 set: – actuation is blocked by other keys
bit 4 set: – role of this "hot" key is redefined
bit 5 set: – actuation depends on execution conditions.

A.02-8. ASCII service marks and commands

Positions 0 – 31 in American Standard Code for Information Interchange (ASCII) are devoted to service marks and commands. All DOS's codepages have inherited these 32 service codes. Under MS-DOS7 most part of these service codes is ignored, but some are executed as commands.

Some service codes can be entered by key combinations, described in article 1.05. Another way to input service codes is by their decimal ASCII number (0 – 31) with keys in numerical keypad while the ALT key is kept pressed.

First response to entered service code may be got from input module of the CON (console) device driver. Further response may be given by command interpreter. When a service code is sent to output, BIOS system enables to execute it as a command (8.01-21, 8.01-33) or to avoid its execution (8.01-17) as well. By default the output module of the CON (console) device driver doesn't attempt to avoid execution of some service codes. Such behavior may be altered by sending a parameters string (8.02-41) to the CON device driver, but there is no reason to do this because DOS programs are allowed to output data directly via desirable BIOS function(s).

Sometimes ASCII service codes may be useful, but it must be known beforehand, where and how each particular service code will be interpreted. Therefore the following table shows a list of those service codes, which are active under MS-DOS7, with description of actions, associated with these codes.

Code	Number	Description
00h	0	End marker of interpreted lines, including lines with names and with environmental variable's values.
03h	3	"End of Text" marker, terminates execution of command files (example – in article 3.21).
07h	7	"Beep" code. Being sent to output, it causes a short sound signal.
08h	8	Shifts cursor one character cell leftwards. Being sent via CON device driver, erases the last character.
09h	9	Horizontal tabulation code. Being sent to display, it is automatically expanded into 8 spaces.
0Ah	10	"Line feed" code, causes transition to the next line without cursor's return to start of line (note 1).
0Ch	12	Command "Eject Sheet" for printers. Both BIOS and CON device driver ignore this command.
0Dh	13	"Carriage Return" code, returns cursor to start of a line. Also marks end of line in DTA region (8.02-16).
1Ah	26	Optional end mark for textual files; at this mark copying of a

Continuation of table A.02-8

		file may be disrupted (3.06).
1Bh	27	"Escape" code is used as marker for commands, addressed to ANSI.SYS driver (if it is installed).

Note 1: service codes 0Dh 0Ah together are used as end-of-line marker in all textual files typed under DOS.

Note 2: there is a non-zero probability to encounter resident modules, responding to some service codes, which are not shown here and normally are ignored under DOS.

A.03 Disks access databases

A.03-1. Structure of Drive Parameters Blocks (DPB)

DOS stores disk access parameters in DPB blocks – one per each available logical disk of any kind, and in one more copy of such block for default (current) drive. The INT 21\AX=7302h function (8.02-79) enables to copy any DPB block into a prepared buffer. Pointers to DPB blocks are returned by INT 21\AH=1Fh and by INT 21\AH=32h functions (8.02-24), documented in previous versions of DOS. In fig.8 below the whole access path to disk's C: DPB block is shown, including a call for INT 21\AH=32h function, reading block's address 00C9:13C0h from DS:BX registers and display of DPB block's dump. In the displayed dump at offset 19h there is address 00C9:13FDh of the next DPB block, related to next logical disk D:. A dump of DPB block for disk D: is also shown in fig.8.

```
D:\MSDOSTXT>debug
-a100
195B:0100 mov AH,32
195B:0102 mov DL,03
195B:0104 int 21
195B:0106 nop
195B:0107
-g=100 106

AX=3200  BX=13C0  CX=0000  DX=0003  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=00C9  ES=195B  SS=195B  CS=195B  IP=0106   NV UP EI PL NZ NA PO NC
195B:0106 90            NOP
-d 00C9:13C0 L3D
00C9:13C0  02 02 00 02 0F 04 01 00-02 00 02 15 02 E3 F9 FA   ................
00C9:13D0  00 F5 01 5E 00 70 00 F8-00 FD 13 C9 00 00 00 FF   ...^.p..........
00C9:13E0  FF FF FF 00 1A 72 0D 06-2E 15 02 00 00 E3 F9 00   .....r..........
00C9:13F0  00 FA 00 00 00 56 2E 8B-36 00 00 00 00            .....V..6....
-d 00C9:13FD L3D
00C9:13F0                                          03 03 00                ...
00C9:1400  02 0F 04 01 00 02 00 02-F9 01 21 EB EC 00 D9 01   ..........!.....
00C9:1410  5E 00 70 00 F8 00 3A 14-C9 00 FC 0B FF FF FF FF   ^.p...:.........
00C9:1420  96 00 5E C3 1E 56 F9 01-00 00 21 EB 00 00 EC 00   ..^..V....!.....
00C9:1430  00 00 04 1A 00 B4 00 00-00 00                     ..........
-
```

Fig. 8

The mentioned legal functions of access to DPB blocks automatically attempt to read the requested disk in order to update data in DPB block. This makes program's execution slower and sometimes can't be applied to removable media, which may be absent in the drive at that moment. Alternative is to read DPB block's address from a cell at offset 45h in CDS entry (A.03-3) of the same disk.

All DPB blocks have the same structure as that shown below. Bytes up to offset 20h are the same as in previous DOS versions, but bytes beyond offset 20h are specific for extended DPB blocks in MS-DOS7.

Offset	Size	Description
00h	1	Logical disk number (00h = A:, 02h = C:, and so on)
01h	1	Disk's number in driver's list of disks
02h	2	Sector size (in bytes)
04h	1	Highest sector number in a cluster
05h	1	Shift count to convert clusters into sectors
06h	2	Number of reserved sectors (preceding FAT)
08h	1	Number of FAT tables
09h	2	Maximum number of root directory entries
0Bh	2	Number of first sector containing user data
0Dh	2	Highest cluster number (number of clusters + 1)
0Fh	2	Number of sectors per FAT table
11h	2	Sector number of root directory first sector
13h	4	Pointer to disk's driver header (A.05-1)
17h	1	Media ID byte (INT 21\AH=1Ch, 8.02-17)
18h	1	Flags (= 00h if disk was accessed, or = FFh if not)
19h	4	Pointer to DPB block for the next disk
1Dh	2	Cluster at which to start search for free space
1Fh	2	Number of free clusters on disk (FFFFh if unknown)
21h	2	Most significant word of free cluster count
23h	2	– bits 0 – 3: zero-based FAT number of active FAT – bit 7: don't copy active FAT to inactive FATs
25h	2	Number of FAT information sector (note 2)
27h	2	Sector number of backup boot-sector
29h	4	First sector number of the first disk's cluster
2Dh	4	Number of the last disk's cluster
31h	4	Number of sectors occupied by FAT
35h	4	Cluster number where root directory starts

Note 1: data in DPB block are translated from BPB block (A.03-4) of the same disk by means of INT 21\AH=53h function.

Note 2: a word FFFFh at offset 25h means that there is no FAT information sector on a requested disk. If FAT information sector is present, it contains at offset 00h a fixed double-word signature 61417272h, the second double word at offset 04h is number of free clusters (or FFFFFFFFh if unknown), the third double word at offset 08h is a number of the most recently allocated cluster.

A.03-2 Disk data tables (DDT)

DDT tables represent a special database for block device drivers, integrated into DOS's core. DDT tables correspond to local disks, which are properly detected and supported by BIOS, including those emulated by BIOS from disk's images on bootable CD/DVD-ROMs. There are no DDT tables for IFS drives, dummy disks, RAM-disks and all other disks, opened for access by drivers, specified in configuration files.

DDT tables are arranged as a chain-like queue of tables, each 96h bytes long. A pointer to the first table is returned by INT 2F\AX=0803h function (8.03-04). This is enough to trace the whole queue, since the first double word in each table is a pointer to DDT table for the next logical disk. The table having the first word FFFFh is the last one in the chain.

```
D:\MSDOSTXT>debug
-a100
195B:0100 mov AX,0803
195B:0103 int 2F
195B:0105 nop
195B:0106
-g=100 105

AX=0803  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B52  ES=195B  SS=195B  CS=195B  IP=0105   NV UP EI PL NZ NA PO NC
195B:0105 90            NOP
-d 0B52:0000 L20
0B52:0000  96 00 52 0B 00 00 00 02-FF 01 00 02 40 00 68 01   ..R.........@.h.
0B52:0010  00 02 00 09 00 01 00 00-00 00 00 00 00 00 00 00   ................
-d 0B52:0096 L20
0B52:0090                  2C 01-52 0B 01 01 00 02 01 01         ,.R.........
0B52:00A0  00 02 E0 00 60 09 F9 07-00 0F 00 02 00 00 00 00   ....`...........
0B52:00B0  00 60 09 00 00 00                                 .`....
-d 0B52:012C L20
0B52:0120                                    C2 01 52 0B                ..R.
0B52:0130  80 02 00 02 10 01 00 02-00 02 00 00 F8 FA 00 3F   ...............?
0B52:0140  00 40 00 BF 1F 00 00 41-A0 0F 00 00               .@.....A....
-
```

Fig. 9

Fig.9 illustrates access to DDT tables for disks A:, B: and C:, including a call for INT 2F\AX=0803h function, reading the returned address (0B52:0000h) of the first DDT table from DS:DI registers, display of a partial dump of DDT table for disk A:, reading

from the first 4 bytes of that dump the address (0B52:0096h) of DDT table for the next disk B:, display of a partial dump of DDT table for disk B:, and repetition of the last two operations relative to DDT table for disk C:.

The table below shows data structure in one DDT table. The same data structure is accepted by INT 2F\AX=0801 function (8.03-02), appending a chain of DDT tables with a one more table for a new logical disk.

Offset	Size	Description
00h	4	Pointer to next DDT table (or FFFFh if the last table)
04h	1	Corresponding physical drive number: from 00h and on for floppy disk drives from 80h and on for hard (fixed) disk drives
05h	1	Logical disk number in a list of disks, accessed by DOS's core drivers. If disk's letter-names were not reassigned, then 00h = A:, 02h = C:, and so on.
06h	25	Current disk's BPB block (up to offset 19h, A.03-4)
3Bh	1	– bit 6: – file system FAT-16, – bit 7: – disk must return "Not Ready" to all appeals
3Ch	2	Counter of opened files belonging to this disk
3Eh	1	Device type (as byte at offset 01h in table A.04-3)
3Fh	2	– bit 0: – fixed hard disk – bit 1: – door lock ("changeline") supported – bit 2: – changes of current BPB are not allowed – bit 3: – all sectors in a track have the same size – bit 4: – LUN number must be specified (note 1) – bit 5: – several logical disks on that physical drive – bit 6: – disk change detected – bit 7: – disk's parameters were changed (note 2) – bit 8: – disk reformatted, media's BPB was changed – bit 9: – access ban flag (note 3)
43h	25	Default disk's BPB (A.03-4) block (note 4)
7Dh	12	11-byte long volume label, terminated with 00h
89h	4	Disk's serial number
8Dh	9	File system type name, terminated with 00h.

Note 1: LUN (Logical Unit Number) is used for discrimination between devices, sharing the same number on a bus. In particular, this is necessary for optical DVD-RAM drives, which present themselves with different LUN numbers either as removable HDD or as CD/DVD-ROM disk. Flash card adapters also represent flash cards in different slots as disks with different LUN numbers.

Note 2: if disk's parameters were changed, data in DDT table must be reset by INT 21\AX=440Dh\CX=4840h function (8.02-46).

Note 3: access ban flag disables both reads and writes. It is applied to HDDs only, in particular, to other primary partitions beyond the first primary partition. Inverse state of access ban flag is reported by INT 21\AX=440Dh\CX=4867h function and may be set anew by INT 21\AX=440Dh\CX=4847h function (8.02-46).

Note 4: the BPB data block (A.03-4) at offset 43h corresponds not to the current media, but to default type of removable media for this drive. Normally default media is the highest capacity media.

A.03-3 Current Directory Structure (CDS)

CDS table is an array of data blocks (entries). Each CDS entry corresponds to one logical disk and specifies several parameters of that disk, including a path to the current (default) directory.

A pointer to the first CDS entry is stored at offset 16h in DOS's list of lists (A.01-2). Just there a byte at offset 21h stores total number of entries in CDS table, defined by LASTDRIVE specification in CONFIG.SYS file (4.17). Each CDS entry is 58h bytes long.

```
D:\MSDOSTXT>debug
-a 100
195B:0100 mov AH,52
195B:0102 int 21
195B:0104 nop
195B:0105
-g =100 104

AX=5200  BX=0026  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=195B  ES=00C9  SS=195B  CS=195B  IP=0104   NV UP EI PL NZ NA PO NC
195B:0104 90            NOP
-d 00C9:0026 L1A
00C9:0020                   46 13-C9 00 CC 00 C9 00 4C 00         F.......L.
00C9:0030  70 00 16 00 70 00 00 02-6D 00 C9 00 00 00 03 D2   p...p...m.......
-d D203:0000 L58
D203:0000  41 3A 5C 44 4F 53 5C 4D-53 37 00 00 00 00 00 00   A:\DOS\MS7......
D203:0010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
D203:0020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
D203:0030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
D203:0040  00 00 00 00 40 46 13 C9-00 96 02 00 00 FF FF 02   ....@F..........
D203:0050  00 00 00 00 00 00 00 00                           ........
-
```

Fig. 10

Access to CDS table is illustrated by fig.10. The first step is a call for INT 21\AH=52h function (8.02-59), returning address of DOS's list-of-lists (00C9:0026h) in registers ES:BX. The next step is display of memory contents dump, starting at returned address. Length of the displayed dump is chosen so that the last 4 bytes at offsets 16h – 19h show the address of CDS table (D203:0000h). Memory contents dump, starting at that

address, presents CDS entry for disk A:. 58h bytes behind each previous CDS entry a next CDS entry follows, representing data for the next logical disk.

Among CDS table entries there are dummy entries, having no corresponding logical disks. Dummy entries reserve letter-names for those logical disks, which may be created or become accessible later: RAM-disks, IFS disks, network disks, etc. CDS table is created by IO.SYS loader during commands interpretation in CONFIG.SYS file. Later CDS table can't be appended with new entries. Therefore a sufficient number of dummy entries must be ordered beforehand by means of LASTDRIVE command (4.17).

Data structure of one CDS entry is shown in the table below.

Offset	Size	Description
00h	67	Path to the current directory (notes 1 and 2).
43h	2	Attribute word of this logical disk: bit 7: – hide disk's letter-name from assign-list bit 12: – virtual disk created by SUBST.EXE (note 2) bit 13: – virtual disk created by JOIN.EXE (note 2) bit 14: – disk is on a physical drive (notes 3 and 4) bit 15: – disk is accessed via network redirector (note 4)
45h	4	Pointer to DPB block for this disk (A.03-1)
49h	2	Starting cluster of current directory (note 5)
4Fh	2	Number of characters to hide (note 6)
52h	4	pointer to redirector or to IFS driver (or zero if none)

Note 1: for local disks a path to the current directory includes disk's letter-name, colon, backslash, and the rest part of path. End of path must be marked with byte 00h.

Note 2: if either bit 12 or bit 13 in attribute word is set, then the path at offset 00h isn't a real path.

Note 3: a zero value of both attribute bits 14 and 15 means that this entry is a dummy or is disabled. Such disks are kept hidden.

Note 4: if both bits 14 and 15 in attribute word are set, hence this disk represents an IFS file system.

Note 5: this cluster number is counted from the start of logical disk. Therefore for the root directory cluster number is 0000h. If disk hasn't been accessed yet, cluster field is filled with FFFFh.

Note 6: DOS may report only the final part of the path to current directory, if a word at offset 4Fh defines a non-zero number of characters, which are to be hidden.

Note 7: in early DOS versions CDS entry was 51h byte long; bytes 51h – 57h have been added since MS-DOS4 for IFS and network drivers.

A.03-4 BIOS Parameter Block (BPB) for a disk

BPB is a part of boot sector on disk media. When DOS starts, BPB data are used to fill internal DOS's tables: DPB (A.03-1) and DDT (A.03-2). This procedure is repeated each time a media change is detected, so that data are always kept updated.

In disk's partitions with FAT-16 file system the BPB is 39h bytes long; its structure is shown in the first column ("F16") of the table below. BPB data inside DDT table have a slightly different "standard" structure, shown in third column ("STD"). Standard BPB blocks are accepted by INT 21\AX=440Dh\CX=0840h function and are returned by INT 21\AX=440Dh\CX=0060h function (8.02-46). Both these functions can be applied to disks with FAT-12 or FAT-16 file systems only.

In disk's partitions with FAT-32 file system the BPB is 5Ah bytes long; its structure is shown in the second column ("F32") of the table below. The same data inside DDT table are arranged according to new "extended" structure, shown in the fourth column ("EXT") of the table. For these data structures MS-DOS7 provides other functions (8.02-46), which were not available in previous DOS versions: INT 21\AX=440Dh\CX=4840h to refresh BPB data in DDT and INT 21\AX=440Dh\CX=4860h to read BPB data in DDT. Both these new functions should be applied to disks formatted with FAT-32 file system.

F16	F32	Std	Ext	Size	Description
00h	00h			3	Jump command EBh 3Ch 90h for FAT-16 or EBh 5Ah 90h for FAT-32
03h	03h			8	ID of program, which formed BPB block
0Bh	0Bh	00h	00h	2	Sector's size (in bytes)
0Dh	0Dh	02h	02h	1	Sectors per cluster (FFh if unknown)
0Eh	0Eh	03h	03h	2	Reserved sectors preceding first FAT
10h	10h	05h	05h	1	Number of FATs (normally 2)
11h	11h	06h	06h	2	Number of entries in the root directory
13h		08h	08h	2	= 0000h (note 1)
15h	15h	0Ah	0Ah	1	Media ID byte (note 2)
16h	16h	0Bh	0Bh	2	Number of sectors per FAT (note 3)
18h	18h	0Dh	0Dh	2	Number of sectors per track
1Ah	1Ah	0Fh	0Fh	2	Number of heads
1Ch	1Ch	11h	11h	4	Starting sector number (note 4)
20h	20h	15h	15h	4	Total number of sectors (note 1)
	24h		19h	4	Number of sectors per FAT (note 3)
	28h		1Dh	2	Same as word 23h in DPB (A.03-1)
		1Fh		2	Number of cylinders (note 5)
	2Ah		1Fh	2	Version of file system
		21h		1	Device type (as byte 01h in A.04-3)
	2Ch		21h	4	Root directory's first cluster number

Continuation of table A.03-4

		22h		2	Attributes (as word 02h in A.04-3)
24h	40h			1	Physical drive number
			25h	2	Information sector (note 2 to A.03-1)
26h	42h			1	Extended boot-sector signature (= 29h)
27h	43h		27h	4	Disk's serial number (in binary form)
2Bh	47h		2Bh	11	Volume's label (or "NO NAME ")
36h	52h		36h	8	File system type name

Note 1: for partitions smaller than 32 Mb a double word at offset 15h must be zero, the number of sectors in such partitions must be specified in a word at offset 08h.

Note 2: media ID byte corresponds to specification of INT 21\AH=1Ch function (8.02-17). If type of removable media is not identified, then the 00h value is assigned to media ID byte.

Note 3: in "extended" BPB blocks a cell at offset 0Bh has 0000h value, and number of sectors, occupied by FAT table, is expressed by a double word value in a cell at offset 19h.

Note 4: in BPBs of HDD's partitions the starting sector number is the same as that specified in corresponding partition's descriptor (A.13-5) at offset 08h.

Note 5: a word at 1Fh and following bytes 21h, 22h of the standard BPB block are not included in extended BPB block and in BPB blocks inside DDT table (A.03-2).

A.04 I/O control data tables

A.04-1 Data block for IOCTL serial number functions

This data block is returned by functions INT 21\AX=440Dh\CX=4866h (8.02-46) and INT 21\AX=6900h (8.02-77), reading serial number from storage media. Data block of the same structure is accepted by functions INT 21\AX=440Dh\CX=4846h (8.02-46) and INT 21\AX=6901h (8.02-77) in order to assign new serial number to a disk.

Offset	Size	Description
00h	2	= 0000h
02h	4	Disk's serial number (in binary form)
06h	11	Disk's volume label (or "NO NAME ", if none)
11h	8	On return only: file system type name (note 1)

Note 1: name "CDROM " corresponds to High-Sierra CD-ROM file system, name "CD001 " corresponds to ISO 9660 CD-ROM file system.

A.04-2 File handle's information word

Handle's information word, read at offset 05h in SFT entry (A.01-4), is returned by INT 21\AX=4400h function (8.02-40). If specified handle refers to SFT entry, related to a non-file object, then returned information word should be interpreted according to table A.05-2. If specified handle refers to SFT entry, related to an opened file, then returned information word should be interpreted according to the table below. Distinctive feature of file handle's information word is clear state of its 7-th bit.

Bits	Description
15	File is not local, it is accessed via a redirector (network)
14	Don't set file date/time when file is closed
11	File is stored on a fixed (non-removable) media
7	Clear state of bit 7 is a distinctive feature of file's handle
6	Writing operation has not been performed yet
5-0	disk number (000000b = A:, 000001b = B:, 000010b = C:, and so on)

A.04-3 Data block for disk parameters specification

A pointer to this data block should be prepared in DS:DX registers before subfunction INT 21\AX=440Dh\CX=4840h (8.02-46) is called for in order to update BPB data in tables DPB (A.03-1) and DDT (A.03-2). Data block of the same structure is returned by INT 21\AX=440Dh\CX=4860h subfunction (8.02-46). Buffer's address for the returned data block should be prepared in advance in DS:DX registers. A byte of flags at offset 00h is not returned, it defines request conditions and should be specified before the call.

Offset	Size	Description
00h	1	Flags (bits 3 – 7 must be zero): bit 0: – apply operation to current BPB (note 1) bit 1: – use track layout fields only (note 2) bit 2: – sectors are of the same size (note 2)
01h	1	Device type: = 00h – 320 kb or 360 kb floppy drive = 01h – 1.2 Mb floppy drive = 02h – 720 kb floppy drive = 05h – fixed (hard) disk drive = 06h – tape drive storage device = 07h – other devices (including 1.44 Mb floppy) = 08h – optical disc drive = 09h – 2.88 Mb floppy drive
02h	2	Storage device attributes (bits 2 – 15 must be zero):

Continuation of table A.04-3

		bit 0: – device with non-removable medium bit 1: – media change registration supported
04h	2	Number of cylinders (or number of tracks)
06h	1	Media flags: = 01h – 320kb/360kb diskette = F8h – compressed logical disk = 00h – all other types of media
07h	31	BPB data block (note 3)

Note 1: if bit 0 in flag's byte is set, then updating or copying of BPB block (A.03-4) causes an attempt of access to physical storage media. But when bit 0 in flag's byte is clear, then access to physical storage media wouldn't be attempted: subject of operation will be a copy of BPB block inside DDT table (A.03-2) at offset 43h, which specifies the default type of storage media for this particular device.

Note 2: bits 1 and 2 in flag's byte specify interpretation of optional sub-table, defining sector allocation in a track. This sub-table up to 256 bytes long may start at offset 26h for subfunction CX=0840h and at offset 5Ch for subfunction CX=4840h. Media with non-equal sector sizes are not considered in this book, though. For subfunction CX=4860h bit 1 in flag's byte must be clear.

Note 3: for subfunction CX=0840h the BPB table at offset 07h must have standard BPB structure (A.03-4). Final 6 bytes of BPB block are accepted by subfunction CX=0840h if flag's byte at offset 00h has its bit 0 set, otherwise bytes after offset 1Eh are ignored. For subfunction CX=4840h the BPB table at offset 07h must be 53 bytes long according to extended BPB structure (A.03-4).

A.04-4 Structure of data block for read/write functions

A pointer to this data block is accepted by INT 21\AX=440Dh\CX=4861h reading function and by INT 21\AX=440Dh\CX=4841h writing function (8.02-46). These functions can't be executed inside "DOS box" under WINDOWS OS unless the addressed logical disk is locked in advance (8.01-58).

Offset	Size	Description
00h	1	= 00h
01h	2	Requested number of drive's head
03h	2	Requested number of drive's cylinder
05h	2	Number of the sector to start reading or writing
07h	2	Number of sectors to be read or written
09h	4	Pointer to a buffer with data or for data

A.04-5 Structure of data block for format/verify functions

A pointer to this data block is accepted by INT 21\AX=440Dh\CX=4842h function for formatting and by INT 21\AX=440Dh\CX=4862h verifying function (8.02-46). These functions can't be executed inside "DOS box" under WINDOWS OS unless the addressed logical disk is locked in advance (8.01-58).

Offset	Size	Description
00h	1	bit 0: query for status code, don't actually format bit 1: format multiple tracks (for HDDs only)
01h	2	Drive's head to be activated
03h	2	The cylinder where the heads should be driven
05h	2	Number of tracks to be formatted or verified

Note 1: for format function a word at 05h is ignored, if byte at offset 00h has its bit 1 clear: only one track would be formatted in this case.

Note 2: for verification function a number of tracks in word 05h should correspond to no more than 255 sectors, bit 0 set in byte at offset 00h specifies verification of multiple tracks, bit 1 must be zero.

Note 3: on return the byte at offset 00h is replaced with status code:

00h – this function is supported by BIOS,
01h – this function is not supported by BIOS,
02h – given specifications don't suit for this logical disk,
03h – there is no media in the drive.

Returned status code 00h doesn't confirm successful outcome: success should be confirmed by returned clear state of CF flag.

A.05 Driver's data structures

A.05-1 Driver's header structure

The table below shows data offsets for 3 types of DOS driver's headers:
column "B" – for "block" devices, i.e. disk and tape storage drives;
column "C" – for 'character" devices, i.e. communication channels;
column "D" – for CD-ROM drivers, cooperating with MSCDEX.EXE.

A pointer to disk driver header is given in a double word at offset 13h in Drive Parameter Block for the corresponding disk (A.03-1). "Character" device drivers may be identified by a signature at offset 0Ah in their header while tracing a chain of header's addresses. Tracing start address may be got at offset 22h in DOS's List-of-Lists (A.01-2), or else may be obtained by means of INT 2F\AX=122Ch function (8.03-12).

B	C	D	Size	Description
00h	00h	00h	4	Next driver's address field (note 1)
04h	04h	04h	2	Driver's attributes (A.05-2)
06h	06h	06h	2	Offset of strategy routine entry point
08h	08h	08h	2	Offset of interrupt routine entry point
	0Ah	0Ah	8	Driver's signature field
		14h	1	First disk, controlled by the driver (note 2)
0Ah		15h	1	Number of disks, controlled by driver

Note 1: the next driver's address field must be initialized as FFFF:FFFFh; later DOS will fill this field with address of the next driver. If no next driver would be loaded, then remaining FFFFh value at start of current driver's header will signify the end of driver's addresses reference chain.

Note 2: byte at offset 14h must be initialized with 00h value. Later MSCDEX.EXE program (5.08-03) or SHSUCDX.EXE program (5.08-04) will replace initial zero value with number (note 1 to 8.02-17) of the first disk, controlled by this driver.

A.05-2 Driver's attributes

Driver's attribute word is at offset 04h in driver's header (A.05-1). But meaning of most bits in attribute word is different for "character" device drivers (in second column of the table below) and for "block" devices drivers (in the third column). CD/DVD-ROM device drivers, cooperating with programs MSCDEX.EXE (5.08-03) or SHSUCDX.EXE (5.08-04), and also drivers of virtual disks, created by SUBST.EXE program (6.23), formally belong to "character" device drivers, as far as bit 15 in their attribute word is set.

Character device driver's attribute word is used as a basis of channel handle's information word, which is stored at offset 05h in corresponding SFT entry (A.01-4) and is returned by INT 21\AX=4400h function (8.02-40) in response to channel handle requests (about file's handle requests – in A.04-2). Differences between "character" device driver's attribute word and channel's handle information word evince themselves in bits 4 – 7 in the second column of the table below: these bits in "character" device driver's attribute word normally are clear. In channel handle's information words bit 7 is set: it is their main distinctive feature from file handle's information words.

Bit	"Character" devices (channels)	"Block" devices (disks, tapes)
0	STDIN channel (note 1)	= 0 (reserved)
1	STDOUT channel (note 1)	32-bit addresses support
2	NUL channel (note 1)	= 0 (reserved)
3	CLOCK channel (note 1)	= 0 (reserved)

Continuation of table A.05-2

4	Output via INT 29 supported	= 0 (reserved)
5	Raw output (note 2)	= 0 (reserved)
6	Channel adds EOF on input	IOCTL support (note 3)
7	= 1 (as a non-file handle)	IOCTL support (note 4)
9	= 0 (reserved)	No direct I/O (note 5)
11	Lock support (note 6)	Lock support (note 6)
12	= 0 (reserved)	CD-ROM or remote device
13	Output until busy supported	Non-IBM's format
14	IOCTL support (note 7)	IOCTL support (note 7)
15	= 1 – "character" device symptom	= 0 – "block" device symptom

Note 1: among attribute bits 0 – 3 for character device drivers one bit only may be set (or none).

Note 2: raw (binary) output means that neither of output characters is interpreted by driver as a command (as it is shown in A.02-8).

Note 3: set state of bit 6 signifies support for functions INT 21\AX=440Ch, 440Dh, 440Eh, 440Fh.

Note 4: set state of bit 7 signifies support for functions INT 21\AX=4410h, 4411h.

Note 5: set state of bit 9 means that disks, controlled by this driver, are inaccessible to functions of BIOS's INT 13 handler. Set state of bit 9 is typical for drivers, providing access to remote disks, to IFS disks and to disks with parameters, substituted by DRIVPARM command (4.09).

Note 6: set state of bit 11 means that driver is able to transfer slot lid lock signals for removable disk drives.

Note 7: set state of bit 14 means that driver is able to cope with control parameters, sent via INT 21\AX=4403h and INT 21\AX=4405h functions (8.02-41).

Note 8: driver's attribute bits, not mentioned in this table, are considered reserved and normally must be clear.

A.05-3 Selected requests to device drivers

Interaction between DOS and any device driver is performed by sending an address of request data block in ES:BX registers with a CALL FAR command to driver's strategy routine. The driver receives code of operation and initiates its execution. After some time DOS sends another call for driver's interrupt routine, which fills the same request data block with requested results of performed operation. DOS accepts the result, if successful termination of operation is confirmed by status byte in returned data block (A.05-4).

The same forms of request data blocks are accepted by INT 2F\AX=0802h function (8.03-03), which implicitly sends requests to block device drivers, integrated into DOS's core. These drivers control logical disks, having parameters registered in corresponding

DDT tables (A.03-2). Only these logical disks can be addressed by INT 2F\AX=0802h function.

The first column of the table below specifies size of request data block, the second column – code of the requested operation, the fourth column shows which driver type this operation can be applied to. The fifth column shows whether the operation can be requested via INT 2F\AX=0802h function.

Size	Code	Operation	Device type	802	Comments
19h	00h	Initialization	both types	N	A.05-5
0Fh	01h	Media change check	"block" type	Y	note 2
14h	03h	Store IOCTL string	note 1	N	A.05-7
1Eh	04h	Read data	both types	Y	A.05-6
0Eh	05h	Nondestructive read	character type	N	note 3
0Dh	06h	Input status request	character type	N	A.05-4
0Dh	07h	Flush input buffer	character type	N	A.05-4
1Eh	08h	Write (send) data	both types	Y	A.05-6
1Eh	09h	Write to disk & verify	"block" type	Y	A.05-6
0Dh	0Ah	Output status request	character type	N	A.05-4
0Dh	0Bh	Flush output buffer	character type	N	A.05-4
14h	0Ch	Receive IOCTL string	note 1	N	A.05-7
0Dh	0Dh	Device open	both types	N	A.05-4
0Dh	0Eh	Device close	both types	N	A.05-4
0Dh	0Fh	Detect removable disk	"block" type	Y	A.05-4
14h	10h	Send data until busy	character type	N	A.05-7
0Dh	17h	Get disk's number	"block" type	Y	A.05-4

Note 1: requests for sending or receiving IOCTL string can be addressed to those drives only (of either type), which have bit 14 set in their attribute word (A.05-2).

Note 2: command 01h (media check) accepts media identifier at offset 0Dh in request data block and returns in the same data block a status byte at offset 0Eh. Value of status byte should be interpreted as follows:

FFh – media has not been changed;
01h – media has been changed;
00h – media change state can't be determined.

Note 3: command 05h (nondestructive read) returns one data byte at offset 0Dh in request data block, if BUSY bit in status byte at offset 04h (A.05-4) isn't set on return.

A.05-4 Format of a request header

Presented header format is used in request data blocks, sent to drivers either with CALL FAR command (A.05-3) or via a call for INT 2F\AX=0802h function (8.03-03). In both cases a pointer to request data block must be in ES:BX registers. The header occupies bytes at offsets 00h – 0Ch in request data block. For a number of operations (06h, 07h, 0Ah, 0Bh, 0Dh, 0Eh, 0Fh, 17h) the request data block is nothing else but a header. Structure of a header, common for all requests to drivers, is shown in the table below.

Offset	Size	Description
00h	1	Length of request block (table A.05-3, column 1)
01h	1	Addressed logical disk's number (note 2)
02h	1	Code of operation (table A.05-3, column 2)
03h	1	Error code (notes 3 and 4)
04h	1	Returned status byte: 01h – operation is done successfully 02h – addressed device is busy 80h – error, operation has failed

Note 1: if operation 0Fh (detect removable drive) returns status 02h (busy), this means the addressed drive is a fixed drive.

Note 2: here a logical disk is defined by its number in a list of disks, controlled by the addressed driver. When a request is sent via INT 2F\AX=0802h function (8.03-03), then these logical disk numbers are identical to absolute logical disk numbers: 00h = A:, 02h = C:, and so on, but for those logical disks only, which have their parameters specified in DDT tables (A.03-2).

Note 3: error code is returned only when status byte at offset 04h has the 80h value, i.e. confirms erroneous outcome. Then error code should be interpreted according to records for INT 2F in table A.06-1.

Note 4: in case of success the 17h operation (Get disk number) returns absolute number of the requested logical disk at offset 03h. If requested number is beyond the list of logical disks, controlled by the addressed driver, then the 00h value is returned. In any case disk's type and media presence are not checked.

A.05-5 Initialization request data block

Only once, just when driver is installed by IO.SYS loader, DOS sends to this driver a request for initialization procedure. Request is sent by CALL FAR command with a pointer to request data block in registers ES:BX. Code 00h of initialization procedure is specified in a header of request data block (A.05-03). Having accepted initialization request, the driver explores the available hardware it is responsible for. Some initial data as well as data, returned by the driver, are transferred in the rest part of request data block,

beyond its header. Data disposition in this part of request data block (offsets 0Dh – 18h) is shown in the table below.

Offset	Size	Description
0Dh	1	On return: number of logical disks controlled by this driver.
0Eh	4	On call: pointer to byte past the end of that memory space which may be occupied by this driver. On return: pointer to the first free byte past the memory space actually occupied by TSR part of this driver.
12h	4	On call: pointer to command line parameters. On return for "block' device drivers only: pointer to BPB data array (A.03-4).
16h	1	On call: zero based number of the first logical disk controlled by this driver (i.e. A: = 00h, C: = 02h and so on).
17h	2	On return: error message flag (note 2)

Note 1: "character" device drivers must return zero in a double word at offset 12h.
Note 2: error message flag value 0000h at offset 17h doesn't cause error message display. But if driver returns error message flag value 0001h, then DOS displays message: "There is an error in your CONFIG.SYS file in line..."

A.05-6 Structure of I/O request data block

I/O request data block is used for driver's data transfer operations called either directly with CALL FAR command (A.05-3) or via INT 2F\AX=0802h function (8.03-03). Address of request data block is presented in ES:BX registers, and header of this request data block specifies code of the requested operation: 04h, 08h or 09h. Data reading operation (code 04h) transfers data from a media into a prepared buffer in memory. Data writing operations (codes 08h and 09h) send data from buffer to disk or to output channel. All mentioned data transfer operations use request data block of the same structure, including a header (A.05-4) and the rest part with access parameters. Disposition of these parameters beyond the header is shown in the table below.

Offset	Size	Description
0Dh	1	Media identifier (for block devices only)
0Eh	4	Address of buffer with data or for data
12h	2	Length of data packet (note 1)
14h	2	Starting sector number (note 2)
16h	4	Pointer to volume identifier (note 3)
1Ah	4	32-bit starting sector number (note 2)

Note 1: length of data packet for channel drivers is counted in bytes. Length of data packet for disks ("block" device) drivers is expressed in number of sectors.

Note 2: some other DOS versions use another data format with a 4-byte starting sector number at offset 14h; a distinctive feature of this data format is length 18h of request data block, specified in the first byte of header (A.05-4). MS-DOS7 sends a 4-byte starting sector number to those drivers only, which declare their support for 32-bit addressing by setting bit 1 in driver's attribute word (A.05-2). Starting sector number for these drivers is specified at offset 1Ah, and then a cell at offset 14h is filled with FFFFh.

Note 3: a pointer to volume identifier is returned by driver when error 0Fh occurs (improper change of media).

A.05-7 Request data block for string operations

This data block is used for driver's byte string transfer operations called by CALL FAR command (A.05-3). Address of data block is specified in ES:BX registers, and header (A.05-4) of that data block specifies code of the requested operation: 03h, 0Ch or 10h. Operation with code 10h sends data string to a channel. Requests with operation codes 03h, 0Ch can be addressed to those drivers only, which declare IOCTL support by having bit 14 set in their attribute word (note 7 to A.05-2). Request with operation code 03h suggests to take into account new values of control parameters, sent via INT 21\AX=4403h or INT 21\AX=4405h function (8.02-41). Operation 0Ch is an offer to the driver to report its actual control parameters, requested either via INT 21\AX=4402h or via INT 21\AX=4404h function (8.02-41).

Request data block for the mentioned operations must have the same structure, including a header (A.05-4) and the rest part; data disposition in this rest part of request data block is shown in the table below.

Offset	Size	Description
0Dh	1	Media identifier (for "block" device drivers only)
0Eh	4	Address of buffer area (with data or for data)
12h	2	On call: number of bytes to read or to write On return: actual number of bytes read or written

A.06 Error codes

A.06-1 Summary table of error codes

After any fault both BIOS and DOS functions return error code. Functions of MS-DOS7 usually leave error code in AL. After BIOS's operations error code may be returned in AH. Interpretation of many error codes depends on which handler has left this error code. For convenience reasons the presented summary table comprises almost all error code interpretations, which may be encountered under MS-DOS7. Being given the whole variety of alternatives, you'll easily choose the appropriate one according to the handler, which has returned the error code.

Code	Handler	Description
00h	INT 24-2F	write-protection violation attempt
	other	no error, successful completion of operation
01h	INT 13	invalid parameter or requested disk doesn't exist
	INT 15	parity error
	INT 16	keyboard buffer is full yet
	INT 24-2F	disk number unknown to the driver
	other	invalid function number or operation number
02h	INT 13	address mark not found
	INT 15	interrupt error
	INT 24-2F	drive is not ready
	other	file not found
03h	INT 13	disk is write-protected
	INT 15	address line A20 gating failed
	INT 24-2F	command is unknown to the driver
	other	path error or path not found
04h	INT 13	sector not found or read error
	INT 24-2F	data error (bad CRC)
	other	too many opened files (no place for more handles)
05h	INT 13	reset failed
	INT 24-2F	bad length of request data block
	other	access denied
06h	INT 13	no media in the drive or media has been changed
	INT 24-2F	seek error
	other	invalid handle
07h	INT 13	drive parameter activity failed
	INT 24-2F	unknown media type
	other	memory control block (MCB) destroyed
08h	INT 13	DMA overrun

Continuation of table A.06-1

	INT 24-2F	sector not found
	other	insufficient memory
09h	INT 13	DMA attempt across 64K or more than 80h sectors
	INT 15	invalid identifier of APM device
	INT 24, 2F	printer is out of paper
	other	invalid memory block address
0Ah	INT 13	bad sector flag detected
	INT 24-2F	write attempt failure
	other	invalid environment
0Bh	INT 13	bad track detected
	INT 15	specified device is not under APM control
	INT 24-2F	read fault
	other	invalid format
0Ch	INT 13	unsupported track format or invalid media
	INT 24-2F	general failure
	other	invalid access mode
0Dh	INT 13	invalid number of sectors on format
	INT 24-26	sharing violation
	other	invalid data
0Eh	INT 13	control data address mark detected
	INT 24-2F	lock violation or media unavailable
0Fh	INT 13	DMA arbitration level out of range
	INT 24-2F	invalid disk change
	other	invalid drive
10h	INT 13	uncorrectable CRC or ECC error on read
	INT 24	FCB is unavailable
	other	attempt to remove the current directory
11h	INT 13	data have been ECC-corrected
	INT 24-26	sharing buffer overflow
	other	it is not the same device
12h	INT 24	code page mismatch
	other	no more files, file index is out of range
13h	INT 24-26	out of input
	other	disk is write-protected
14h	INT 24, 26	insufficient disk space
	other	unknown unit
15h		drive not ready
16h		unknown command
17h		data CRC error
18h		bad length of request data block
19h		seek error
1Ah		unknown media type (non-DOS disk)

Continuation of table A.06-1

1Bh		sector not found
1Ch		printer is out of paper
1Dh		write fault
1Eh		read fault
1Fh		general failure
20h	INT 13	controller failure
	other	sharing violation
21h		lock violation
22h		disk change invalid (see note 2 to table A.06-1)
23h		FCB (File Control Block) unavailable
24h		sharing buffer overflow
25h		code page mismatch
26h		cannot complete file operation (out of input)
27h		insufficient disk space
30h	INT 13	drive has no media sensor
31h	INT 13	no media in the drive
32h	INT 13	non-default media
	other	network request not supported
33h		remote computer not listening
34h		duplicate name on network
35h		network name not found
36h		network is busy
37h		network device no longer exists
38h		network BIOS command limit exceeded
39h		network adapter hardware error
3Ah		incorrect response from network
3Bh		unexpected network error
3Ch		incompatible remote adapter
3Dh		print queue full
3Eh		queue not full
3Fh		not enough space to print file
40h	INT 13	seek failed
	other	network name was deleted
41h		access to network is denied
42h		network device type incorrect
43h		network name not found
44h		network name limit exceeded
45h		network BIOS session limit exceeded
46h		temporarily pause
47h		network request not accepted
48h		network print/disk redirection paused
50h		file exists

Continuation of table A.06-1

52h		cannot make directory
53h		fail on INT 24h
54h		too many redirections
55h		duplicate redirection
56h		invalid password
57h		invalid parameter
58h		network write fault
59h		this function is not supported on network
5Ah		required system component not installed
60h	INT 15	requested APM mode is unavailable (blocked)
64h	Mscdex.exe	unknown error
65h	Mscdex.exe	not ready
66h	Mscdex.exe	EMS memory no longer valid
67h	Mscdex.exe	not High Sierra or ISO-9660 format
68h	Mscdex.exe	slot door is opened
80h	INT 13	timeout, no response (drive may be not present)
	INT 67	internal error
	other	invalid command or function not implemented
81h	INT 67	hardware malfunction
	Himem.sys	VDISK driver was detected
82h	Himem.sys	an A20 line error has occurred
83h	INT 67	invalid handle
84h	INT 67	undefined function requested by application
85h	INT 67	no more handles available
86h	INT 67	error in save or restore of mapping context
	other	requested function is not supported
87h	INT 67	insufficient number of memory pages is present
88h	INT 67	insufficient number of memory pages is available
89h	INT 67	zero number of pages requested
8Ah	INT 67	invalid logical page number encountered
8Bh	INT 67	invalid physical page number encountered
8Ch	INT 67	page-mapping hardware state save area is full
8Dh	INT 67	save of mapping context failed
8Eh	INT 67	restore of mapping context failed
	Himem.sys	a general XMS driver error
8Fh	INT 67	undefined subfunction
	Himem.sys	an unrecoverable XMS driver error
90h	INT 67	undefined attribute type
	Himem.sys	HMA does not exist or is not managed by XMS provider
91h	INT 67	this feature is not supported
	Himem.sys	HMA is already in use
92h	INT 67	success, but a portion of source region is overwritten

Continuation of table A.06-1

	Himem.sys	DX is less than the /HMAMIN parameter (5.04-01)
93h	INT 67	length of data exceeds space allocated to the handle
	Himem.sys	HMA is not allocated
94h	INT 67	conventional and expanded memory regions overlap
	Himem.sys	A20 line is still enabled
95h	INT 67	offset within logical page exceeds size of logical page
96h	INT 67	region length exceeds 1 Mb
97h	INT 67	source and destination have same handle and overlap
98h	INT 67	memory source or destination type undefined
9Ah	INT 67	specified map register or DMA register set not supported
9Bh	INT 67	all map register or DMA register sets are allocated
9Ch	INT 67	map register or DMA register sets not supported
9Dh	INT 67	undefined or unallocated map or DMA register sets
9Eh	INT 67	dedicated DMA channels not supported
9Fh	INT 67	specified dedicated DMA channel not supported
A0h	INT 67	no such handle name
	Himem.sys	all extended memory is allocated
A1h	INT 67	a handle found had no name, or duplicate handle name
	Himem.sys	all available extended memory handles are allocated
A2h	INT 67	attempt to wrap around 1 M conventional address space
	Himem.sys	invalid handle
A3h	INT 67	source array corrupted
	Himem.sys	source handle is invalid
A4h	INT 67	operating system denied access
	Himem.sys	source offset is invalid
A5h	Himem.sys	destination handle is invalid
A6h	Himem.sys	destination offset is invalid
A7h	Himem.sys	length is invalid
A8h	Himem.sys	copy operation has an invalid overlap
A9h	Himem.sys	parity error occurred
AAh	INT 13	drive not ready
	Himem.sys	block is not locked
ABh	Himem.sys	block is locked
ACh	Himem.sys	block lock count overflowed
ADh	Himem.sys	lock failed
B0h	Himem.sys	only a smaller UMB is available
	INT 13	volume is not locked in drive
B1h	Himem.sys	no UMBs are available
	INT 13	volume is locked in drive
B2h	Himem.sys	UMB segment number is invalid
	INT 13	volume is not removable
B3h	INT 13	volume is in use, write cache isn't empty

Continuation of table A.06-1

B4h	INT 13	lock count has been exceeded
B5h	INT 13	a valid eject request failed
B6h		media is write-protected
BBh	INT 13	undefined hard disk error
CCh	INT 13	write fault on hard disk
E0h	INT 13	status register error on hard disk
FFh	INT 13	sense operation failed on hard disk
	INT 15	error enabling address line A20
	other	matching file not found, or no more files, or bad FCB.

Note 1: if error code is returned in AX register, its most significant byte (in AH register) is zero.

Note 2: together with error code 22h a pointer to media identifier is returned in ES:DI registers. This media identifier includes:
at offset 00h – 12 bytes: disk's volume label, ending with 00h;
at offset 0Ch – 1 double word: disk's serial number (in binary form).

Note 3: the HIMEM.SYS driver (5.04-01) returns error codes in BL register.

Note 4: error code, returned by the latest executed DOS's function, is stored in DOS's swappable data area SDA (A.01-3) at offset 04h. BIOS' functions write their error code into BIOS data area (A.01-1), most probably in cell 0040:0074h.

A.06-2 Error class codes

Error class code, returned in BH register by INT 21\AH=59h function (8.02-65), is stored at offset 07h in DOS's swappable area SDA (A.01-3). Interpretation of error class codes is shown in the table below.

Code	Description
01h	out of resource (storage space or I/O channels)
02h	temporary situation (file or record lock)
03h	authorization (denied access)
04h	internal (system software bug)
05h	hardware failure
06h	system failure (configuration file missing or incorrect)
07h	application program error
08h	object not found
09h	bad format
0Ah	object is locked
0Bh	media error
0Ch	object already exists
0Dh	unknown error class

A.06-3 Codes of suggested action

Code of suggested action is returned in BL register by INT 21\AH=59h function (8.02-5) and is stored at offset 06h in DOS's swappable area SDA (A.01-3).

Code	Recommended action
01h	retry
02h	retry after some time
03h	prompt user to reenter input
04h	close opened files, delete temporary files and abort
05h	immediate abort
06h	ignore this error
07h	retry after user intervention

A.06-4 Error locus codes

Error locus code is returned in CH register by INT 21\AH=59h function (8.02-65) and is stored at offset 03h in DOS's swappable area SDA (A.01-3).

Code	Probable locus of the error
01h	unknown or not appropriate
02h	block device (disk error)
03h	network related
04h	device connected to serial port (channel timeout)
05h	memory related

A.06-5 I/O error status codes

Error status code is returned in AH register by INT 25 and INT 26 handlers (8.02-85).

Code	Description
01h	invalid command
02h	invalid address mark
03h	disk is write-protected (for INT 26 only)
04h	requested sector not found
08h	DMA failure
10h	data error (bad CRC)
20h	controller failure
40h	seek operation failure
80h	device failed to respond (timeout)

A.07 Execution service structures

A.07-1 Program Segment Prefix

When a program is loaded for execution into an allotted memory segment, executable code of the program is placed at offset 100h and on. Preceding part of the segment (offsets 00h – FFh) is known as PSP, i.e. Program Segment Prefix. It is filled with important service data, which are used by DOS functions and may be used by the program itself.

By means of DEBUG.EXE that PSP can be peeped most easily, which is formed by COMMAND.COM interpreter for DEBUG.EXE itself. A procedure of displaying a part of that PSP is shown in fig.11. The rest part of that PSP is filled with zeros.

```
E:\DOS\MSDOSDOC\PICT_RAW>debug.exe cjpeg.exe -baseline vc01.bmp
-a100
22D4:0100 mov AH,62                   ; This call returns DEBUG's
22D4:0102 int 21                      ;    segment address in BX register
22D4:0104 nop
22D4:0105
-g=100 104

AX=6200  BX=0D18  CX=6074  DX=0000  SP=0080  BP=0000  SI=0000  DI=0000
DS=0D18  ES=0D18  SS=236D  CS=22D4  IP=0104   NV UP EI PL NZ NA PO NC
22D4:0104 90            NOP
-d 0D18:0000 LC0
0D18:0000  CD 20 FF 9F 00 9A F0 FE-1D F0 4F 03 7C 06 8A 03   . ........O.|...
0D18:0010  7C 06 17 03 7C 06 6B 06-01 01 01 00 02 FF FF FF   |...|.k.........
0D18:0020  FF FF FF FF FF FF FF FF-FF FF FF FF 08 0D 64 3E   ..............d>
0D18:0030  7C 06 14 00 18 00 18 0D-FF FF FF FF 00 00 00 00   |...............
0D18:0040  07 0A 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0D18:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 00 2D 42 41   .!...........-BA
0D18:0060  53 45 4C 49 4E 20 20 20-00 00 00 00 56 43 30   SELIN    .....VC0
0D18:0070  31 20 20 20 20 42 4D 50-00 00 00 00 00 00 00 00   1    BMP........
0D18:0080  14 20 2D 62 61 73 65 6C-69 6E 65 20 76 63 30 31   . -baseline vc01
0D18:0090  2E 62 6D 70 20 0D 76 63-30 31 2E 62 6D 70 20 0D   .bmp .vc01.bmp .
0D18:00A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0D18:00B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-
```

Fig. 11

The program under test in fig.11 is file CJPEG.EXE. Parameters "-baseline" and "VC01.BMP" are transferred to program under test in order to present an example of filling the first and the second FCB blocks, starting at offsets 5Ch and 6Ch correspondingly. Contents of these and some other PSP data fields are explained in the table below.

Offset	Size	Description
00h	2	INT 20 command (for CP/M compatibility)
02h	2	First segment beyond memory allotted to the program
06h	2	Size of executable code (for *.COM files)
0Ah	4	Stored INT 22 termination address

Continuation of table A.07-1

0Eh	4	Stored INT 23 Control-Break handler address
12h	4	Stored INT 24 critical error handler address
16h	2	Segment of parent's PSP (notes 1 and 2)
18h	20	Job File Table (JFT, note 3)
2Ch	2	Segment of environment for the current process
2Eh	4	Caller's SS:SP on entry to last INT 21 call
32h	2	Number of entries in JFT (default is 20)
34h	4	Pointer to JFT (default is PSP:0018h)
3Ch	1	= 00h (= 01h for hieroglyphical keyboards)
40h	2	DOS version to be reported by INT 21/AH=30h
50h	2	A call for the INT 21 functions dispatcher
5Ch	16	First FCB area (note 4)
6Ch	16	Second FCB area (note 4)
80h	1	Length of command line tail (notes 5 and 6)
81h	127	Command line tail or DTA area (notes 5 and 6)

Note 1: if segment address of the parent's PSP at offset 16h points at current PSP, then program is regarded as having no parent or, in other terms, as being its own parent. This is a distinctive feature of permanently loaded program, for example, of command interpreter. Permanently loaded programs can't be terminated by a call for INT 20 or for INT 21\AH=4Ch function.

Note 2: in protected mode some PSP fields, including the "parent" segment field at offset 16h, may be overwritten. Therefore tracing a chain of PSP references may get confused, unless each candidate "parent" PSP segment confirms presence of typical signatures, for example, code CD20h (INT 20) at offset 00h or code CD21h at offset 50h.

Note 3: when current program starts, its JFT (Job File Table, offset 18h) contains SFT (A.01-4) entry numbers – one byte each – for "opened" objects, which are inherited from the parent process. Free spaces in JFT are filled with FFh byte. Values 80h – FEh in JFT correspond to remote files, opened by network redirectors. Default size of JFT – 20 bytes – imposes a restriction on number of opened objects. The INT 21\AH=67h function (8.02-76) enables to overcome this restriction: it arranges a larger JFT outside PSP, replaces a pointer to JFT in PSP at offset 34h and a count of JFT entries at offset 32h. However, child processes in any case can't inherit from their parent process more than 20 "opened" objects.

Note 4: areas at offsets 5Ch and 6Ch are filled as unopened FCBs (A.09-5) with parsed data from first and second command line parameters. Parameters are parsed with INT 21\AX=2901h function (8.02-19). Count of parameters includes those that can't be parsed.

Note 5: the "command tail" area 81h – FFh is filled with a copy of command line with all parameters, which follow command name. The filled part of command tail area is terminated by byte 0Dh. Length of the filled part is written at offset 80h. If the

length is set to 7Fh, and byte at offset FFh is 0Dh, hence real length of command tail exceeds 126 bytes, and its non-truncated version should be found in value of CMDLINE environmental variable.

Note 6: "command tail" area 80h – FFh is used as default DTA (data transfer area) by "find file" functions INT 21\AH=11h,12h,4Eh,4Fh. You may prevent overwriting of "command tail" by changing DTA address with INT 21\AH=1Ah function (8.02-16).

A.07-2 Data block for loading a program

The table below shows structure of a data block, used by INT 21\AX=4B00h and INT 21\AX=4B01h functions (8.02-53) in order to load a program into memory for its further execution.

Offset	Size	Description
00h	2	Environment segment for child process (note 1)
02h	4	Pointer to command line (note 2)
06h	4	Pointer to data for FCB at offset 5Ch (note 3)
0Ah	4	Pointer to data for FCB at offset 6Ch (note 3)
0Eh	4	On return: stack top SS:SP for loaded program (note 4)
12h	4	On return: loaded program entrance point CS:IP (note 4)

Note 1: whole environment of the parent process will be copied into this segment. If child process should be given access not to a copy, but to parent's environment itself, then the 0000h value should be assigned to a word at offset 00h in this data block.

Note 2: command line must include all what is to be written into PSP of the child process starting at offset 80h (note 5 to A.07-1). Command line string must begin with a byte, specifying its length, and must end with byte 0Dh.

Note 3: this data string will be copied into corresponding FCB block (note 4 to A.07-1) inside PSP for the child process. Required structure of this data string is shown in the "N" column of table A.09-5. First 12 bytes should be filled, then 4 bytes 00h must follow. If FCB should be left empty, then its first byte must be 00h, and then 11 bytes 20h must follow.

Note 4: double words at offsets 0Eh and 12h are returned by INT 21\AX=4B01h function only. This function loads a program, but doesn't initiate it's execution. Returned stack top and entrance point enable to start execution of the loaded program later.

A.07-3 Execution state descriptor

The table below shows data structure in execution state descriptor, used by INT 21\AX=4B05h function (8.02-54).

Offset	Size	Description
00h	2	= 0000h (reserved)
02h	2	Flags: bit 0 set: – program is of *.EXE format bit 1 set: – loaded code is an overlay
04h	4	Pointer to program's name, ending with 00h byte
08h	2	PSP segment address of the loaded program
0Ah	4	Entrance point CS:IP of the loaded program
0Eh	4	Size of the loaded program (including PSP)

A.07-4 Data block for server function

As far as server function INT 21\AX=5D00h (8.02-68) enables to execute any INT 21 function as a separate process, this data block defines the states of all registers as required for execution of the selected function. Before this selected function is called for, all specified states will be copied from data block into registers automatically.

Offset	Size	Description
00h	2	Required contents of AX register
02h	2	Required contents of BX register
04h	2	Required contents of CX register
06h	2	Required contents of DX register
08h	2	Required contents of SI register
0Ah	2	Required contents of DI register
0Ch	2	Required contents of DS register
0Eh	2	Required contents of ES register
10h	2	= 0000h (reserved)
12h	2	Virtual machine identifier (note 1)
14h	2	Process identifier (i.e. PSP segment address)

Note 1: if selected function is to be executed under current MS-DOS7, then the 0000h value should be specified as virtual machine identifier.

Note 2: when this data block is used in order to close a process by means of INT 21\AX=5D01h function (8.02-69), then words at offsets 12h and 14h only are taken into account, all other words are ignored.

A.07-5 Interrupt sharing protocol

Many drivers and TSR programs load their interrupt handlers and have to write address of this handler into a certain cell of interrupt table. However, this certain cell may be occupied yet by an address of another handler, which has been loaded beforehand. If new handler has to replace the former one, then a problem arises how to release memory, occupied by the former handler. If new handler complements functions of the former one, then a problem arises how to arrange their interaction. In both cases a solution is that each resident module must provide data, necessary for other resident modules, which may be loaded later.

The first step in arranging resident modules interaction was IBM's Interrupt Sharing Protocol (ISP), stipulating presence of 16-byte data block with fixed placement relative to call address for corresponding resident module. The ISP protocol enables to form a traceable chain of references to all the modules, sharing a common interrupt number. ISP protocol gives an opportunity to alter the order of references in the chain and to remove certain references from the chain. The latter is a necessary condition for unloading resident modules.

According to ISP protocol the call address, written into interrupt table, must point at a command of a short jump 16 bytes ahead, where executable code of resident module starts. The jumped over 16 bytes is just a place for data block. Structure of this data block is shown in the table below. All offsets in the table are counted from resident module's call address.

Offset	Size	Description
00h	2	Short jump command (EBh 10h) to executable code
02h	4	Address of previous handler in handler's chain
06h	2	Signature 4Bh 42h (= "KB")
08h	1	= 00h – this handler is not the first = 80h – this handler is the first in the chain
09h	2	Pointer to module unloading subroutine with RETF command at the end
0Bh	7	Reserved (must be zeros)

Note 1: many resident modules don't conform to ISP protocol. It may be intentionally ignored in order to prevent references chain tracing or resident module unloading.

A.07-6 Alternative Multiplex Interrupt Specification (AMIS)

Multiplex interrupt INT 2F, described in part 8.03, has a serious drawback: coincident identifiers sometimes are appointed to different TSR modules due to inconsistent decisions of their developers. In order to avoid such conflicts an idea has been suggested to assign

identifiers not beforehand by the will of module's developers, but automatically just in course of module's loading. Authorship of this idea is known to belong to Ralf Brown. Idea has been institutionalized by Alternative Multiplex Interrupt Specification (AMIS) and has been implemented by multiplex interrupt INT 2D. Contents of this article are based on version 3.6 of AMIS specification. Besides that, each resident module using multiplex interrupt INT 2D must conform to IBM's Interrupt sharing protocol ISP (A.07-5).

According to AMIS specification an identifier for resident module should be searched in a cycle of calls for multiplex interrupt INT 2D with operation code AL = 00h and with successive incrementation of candidate identifiers in AH register, starting from AH = 00h. If anyone of loaded resident modules considers a particular candidate identifier in AH register as its own, it must set AL = FFh, must return in CH:CL registers its version number and must return in DX:DI registers a pointer to a signature up to 80 bytes long, ending with byte 00h. Search cycle should terminate on condition of returned zero value in AL register: it means that neither loaded resident module has appropriated the last specified candidate identifier. Hence, the latter is free, and current resident module can assign this identifier to itself.

A similar cycle, but with other termination condition, should be arranged in order to find out whether a particular resident module is loaded or not. For this purpose the main role belongs to signature, pointed at by address returned in DX:DI registers. 16 bytes of this signature must be sufficient for identification of resident module. First 8 bytes must specify company or developer's name, the following 8 bytes – a name of program or a driver, which has loaded the responding resident module. Abridged names are allowed. If a name is shorter than 8 bytes, it should be appended with spaces (bytes 20h). The rest part of signature beyond 16 bytes is optional, but it may specify version and other useful data. Signature check enables to prevent repetitive loading of the same resident modules. Search cycle for a signature check also reveals identifier, assigned to the requested resident module. When the identifier is known, then other functions of this resident module can be called for.

As far as calls for multiplex interrupt imply a search through a chain of references and are performed slowly, hence repetitive calls for module's specific functions via INT 2D are not expedient (though are allowed with operation codes above 10h). Direct addressing to module's functions with a CALL FAR command (7.03-08) is preferable. In order to obtain direct address, the INT 2D interrupt should be called once with operation code AL = 01h and with particular module's identifier in AH register. Address for direct calls, returned in DX:BX registers, should be saved, and then later more calls for multiplex interrupt wouldn't be needed.

Beside the mentioned operations with operation codes AL = 00h and AL = 01h, AMIS specification stipulates several other operations, listed in the table below. Unified codes of these operations, shown in the first column of the table, enable to apply identical services for all resident modules. Some operations are optional. If addressed resident module

returns in AL register the status code value 00h, hence it doesn't support the requested operation. On the contrary, returned status code value FFh confirms, that the requested operation is supported and is done successfully. Several operations may return other status code values, informing about specific features of resident module. These and some other peculiarities of AMIS operation execution are explained in notes after the following table.

Code	Description	Comments
00h	Installation check	
01h	Request for direct address	Note 1
02h	Uninstall resident module	Note 2
03h	Request for TSR program activation	Note 3
04h	Report about chained interrupts	Note 4
05h	Request for a list of "hot" keys	Note 5
06h	Request for device driver information	Note 6

Note 1: returned status value AL = 00h means that requested resident module can't be addressed with CALL FAR command. Validity of the address returned in DX:BX registers must be confirmed by returned status code value AL = FFh.

Note 2: a request to uninstall resident module must supply in DX:BX registers an address for return after completion of uninstall operation, though resident module may ignore this address. Returned in AL register status code values (except 00h and FFh) have the following meaning:

- 01 – uninstall attempt failure;
- 02 – uninstall operation will be completed later;
- 03 – module has no uninstaller and stays active;
- 04 – the same, as 03, but module is deactivated;
- 05 – uninstall attempt should be repeated later;
- 06 – module is deactivated, but can't be uninstalled;
- 07 – the same, as 03, plus driver unloading required.

Return of status code values 03, 04 or 07 means a necessity to launch a special uninstaller program. For this program the addressed resident module must return in BX register that segment address, where its executable code is loaded.

Note 3: returned in AL register status code values (except 00h and FFh) have the following meaning:

- 01 – activation attempt should be repeated later;
- 02 – program will be activated later;
- 03 – program is active yet;
- 04 – activation attempt has failed.

After successful activation, confirmed by status code FFh, some programs may report extra information in BX register. After a failure, confirmed by status code 04h, extra information may be returned in BX and CX registers. If the cause of failure remains unknown, BX and CX registers should return zeros.

Note 4: requests for operation 04h must specify in BL register a number of that interrupt (except INT 2D), which is to be checked. Returned in AL register status code values (except 00h) have the following meaning:

01 – check result can't be determined;
02 – specified interrupt has been intercepted;
03 – the same as 02 plus handler's address – in DX:BX;
04 – in DX:BX – pointer to a list of interceptions;
FF – specified interrupt is not intercepted.

Status code value 04 means that interrupt number, specified in BL register, is ignored. Returned list is composed of 3-byte groups per each interrupt: the first byte – interrupt number, the following 2 bytes – offset (inside DX segment) of that interrupt handler's entrance address. End of list is marked with code 2Dh in a place for interrupt number.

Note 5: after successful outcome, confirmed by status code FFh, in DX:BX registers a pointer to a list of "hot" keys is returned. Structure of this list is described in appendix A.02-7.

Note 6: operation 06h returns in AL register number of drivers, installed by addressed resident module, and in DX:BX registers – a pointer to header of the first of these installed drivers (A.05-1). In AH register a byte of flags is returned; bits 3 – 7 in this byte are reserved and must be clear. Set state of the rest flags should be interpreted as follows:

bit 0 – drivers can't be unloaded from memory;
bit 1 – drivers are not included in DOS's drivers chain;
bit 2 – installed drivers are not reenterable.

If addressed TSR program didn't install drivers, it has to return AL = 00h value. On return the contents of AH, BX and DX registers may be arbitrary altered.

A.08 Floppy drive's data structures

A.08-1 Floppy drive's data in BIOS data area

The table presents that information in BIOS data area, which relates to floppy drive(s). All offsets are counted from segment address 0040h, i.e. from the start of BIOS data area.

Offset	Size	Description
10h	2	Flags: bit 0: floppy is able to boot the PC bits 6 – 7: number of floppy drives minus one
3Eh	1	Bit 7 set by IRQ6 handler marks completion of FDD's job
3Fh	1	Motor's status in floppy drive(s)
40h	1	Floppy motor OFF timeout count

Continuation of table A.08-1

41h	1	Status: bits 0 – 4: last operation error (note 2) bit 5: general controller failure bit 6: seek error bit 7: drive isn't ready
42h	3	Registers of floppy drive controller
8Bh	1	Floppy drive data rate control
8Fh	1	Floppy drive registration: bit 0: drive 0 supports 80 tracks bit 2: presence of drive 0 is confirmed bit 4: drive 1 supports 80 tracks bit 6: presence of drive 1 is confirmed
90h	1	Floppy drive 0 media status: bits 0 – 2: =111b for 3.5" disks bit 3: diskette 2.88 Mb bit 4: media type has been determined bits 6 – 7: current data transfer rate
91h	1	Floppy drive 1 media status (just as for drive 0)
94h	1	Floppy drive 0 current track number
95h	1	Floppy drive 1 current track number

Note 1: data placement in BIOS data area may depend on BIOS version (A.01-1).
Note 2: particular values of last error byte at offset 41h should be interpreted as it is shown in table A.06-1 for INT 13.

A.08-2 Access and formatting parameters

Computer's BIOS system stores access and formatting parameters for each floppy disk drive in separate 11-byte tables. A pointer to such table for any particular floppy drive can be obtained with INT 13\AH=00h function (8.01-49). Besides that, one more similar 11-byte table is created for the default ("current") floppy drive; a pointer to the latter table is stored in a cell 0000:0078h (also known as INT 1E) inside interrupt table.

Access and formatting parameters can be changed by INT 13\AH=18h function (8.01-54), but the changes wouldn't come into effect until floppy controller is reset by a call for INT 13\AH=00h (8.01-44).

Offset	Size	Description
00h	1	Parameters, first byte: bits 7 – 4: step rate of head shift bits 3 – 0: head unload time (0Fh = 0.24 s)
01h	1	Parameters, second byte: bits 7 – 1: head load time (01h = 0.004 s)

Continuation of table A.08-2

		bit 0: set state means data transfer via DMA
02h	1	Delay until motor turned off (in clock ticks 1/18 s)
03h	1	Bytes per sector: = 00h – 128 bytes, = 01h – 256 bytes, = 02h – 512 bytes, = 03h – 1024 bytes.
04h	1	Number of sectors per track
05h	1	Gap length between sectors: = 2Ah – for diskettes 5.25", = 1Bh – for diskettes 3.5".
07h	1	Gap length between sectors for formatting: = 50h – for diskettes 5.25", = 6Ch – for diskettes 3.5".
08h	1	Format filler byte (default is F6h)
09h	1	Head settle time in milliseconds
0Ah	1	Motor start time in ticks (1 tick = 1/18 second)

A.08-3 Floppy drive types registered by BIOS

BIOS Setup program stores data about registered floppy drives in a cell 10h of CMOS memory. In order to read these data the cell address 10h has to be sent with OUT command (7.03-66) into port 70h, and after that the required data byte can be read by IN command (7.03-26) from port 71h (some more about that – in note 1 to A.14-1). The required data byte occurs in AL register; bits 4 – 7 in that byte specify features of the first floppy drive, bits 0 – 3 specify features of the second floppy drive, if it exists. Hexadecimal values, expressed by each of these 4-bit groups, should be interpreted independently according to the table below.

Value	Type of floppy disk drive
0	Floppy disk drive isn't present
1	Drive for 360 kb 5.25" diskettes
2	Drive for 1.2 Mb 5.25" diskettes
3	Drive for 720 kb 3.5" diskettes
4	Drive for 1.44 Mb 3.5" diskettes
5	Drive for 2.88 Mb 3.5" diskettes

A.09 Directories and file's data tables

A.09-1 Directory records and file's data

Data concerning files, volume labels and subdirectories are stored in corresponding directory records. Data structure in ordinary 32-byte directory record for an object with a "short" name is shown in the first column "D" of the table below. DOS's "find file" functions INT 21\AH=4E00h (8.02-57) and INT 21\AH=4Fh (8.02-58) read directory records and return the found data in DTA area (8.02-16); format of data, returned by these functions, is shown in second column "F4E" of the table below. Other "find file" functions INT 21\AH=11h (8.02-11) and INT 21\AH=12h (8.02-12) also return found data in DTA area, but in other formats. Third column "F1N" of the table below shows format of data, returned after a search request with normal FCB block (column "N" in appendix A.09-5). But when search request is presented in a form of extended FCB block (column "E" in appendix A.09-5), then the same functions return other data structure, which is shown in the fourth column "F1E" of the table below.

D	F4E	F1N	F1E	Size	Description
			00h	1	= FFh – signature of extended FCB
			06h	1	Attributes for a search (A.09-2)
	00h	00h	07h	1	Disk: 01h = A:, 03h = C: . ., (note 1)
00h	01h	01h	08h	8	Name, appended with spaces to 8 bytes
08h	09h	09h	10h	3	Suffix, appended with spaces to 3 bytes
	0Ch			1	Attributes for a search (A.09-2)
	0Dh			2	Ordinal number of directory record
	0Fh			2	Number of the first directory cluster
0Bh	15h	0Ch	13h	1	Actual attributes (A.09-2) of the object
0Ch				1	Auxiliary attribute byte (notes 2 and 3)
0Dh		0Eh	15h	1	Time in 0.01-second units (note 2)
0Eh		0Fh	16h	2	Object creation time (note 2)
10h		11h	18h	2	Object creation date (note 2)
12h		13h	1Ah	2	Date of the last access
14h		15h	1Ch	2	Starting cluster number (note 4)
16h	16h	17h	1Eh	2	Time of the last update
18h	18h	19h	20h	2	Date of the last update
1Ah		1Bh	22h	2	Starting cluster number (note 4)
1Ch	1Ah	1Dh	24h	4	Object's size in bytes (binary form)
	1Eh			13	Object's name and suffix (note 5)

Note 1: search functions INT 21\AX=4E00h (8.02-57) and INT 21\AH=4Fh (8.02-58) return this byte with its 7-th bit set, if disk is accessed via a network.

Note 2: if object is created under DOS, then this data field is not filled. Copying procedures under DOS don't copy those data, which may be present in this field.
Note 3: this data field is used by operating systems Windows-2000/XP, but the author has no information about role of these data.
Note 4: object's starting cluster number in FAT-16 volumes is a word at offset 1Ah; field at offset 14h is not used. But cluster number in FAT-32 volumes is a double word. The most significant two bytes of this double word are stored at offset 14h.
Note 5: search functions INT 21\AX=4E00h (8.02-57) and INT 21\AH=4Fh (8.02-58) don't overwrite name search template at offset 01h; actual name of the found object, ending with 00h byte, is returned at offset 1Eh.

A.09-2 Structure of attribute byte

Attribute byte at offset 0Bh in a directory record (A.09-1) defines class of the object, associated with this record. Bitfields of attribute byte are explained in the table below.

Bit	Description
0	Read-only file
1	Hidden file
2	System file
3	Volume label (must be zero for files and directories)
4	Directory (must be zero for files and volume labels)
5	File, which is to be stored in archive
6,7	Not used under MS-DOS, must be zero

Note 1: the 0Fh value of attribute byte is regarded as a signature of LFN directory records, associated with files having "long" names (A.09-3). Such records are formed by Windows-95/98/ME operating systems.
Note 2: states of bits 3 and 4 in attribute byte can't be changed by INT 21\AX=4301h function (8.02-39) or by ATTRIB.EXE utility (6.01).
Note 3: file search functions INT 21\AX=4E00h (8.02-57) and INT 21\AH=4Fh (8.02-58) ignore states of bits 0 and 5 in attribute byte.
Note 4: extended "file open" function INT 21\AX=6C00h (8.02-78) accepts in CX register an attribute word with clear bits 4 and 6-15. Role of other bits corresponds to that shown in table A.09-2.

A.09-3 Format of LFN directory records

Each "long" filename, accepted by Windows-95/98/ME operating systems, occupies at least several directory records of standard 32-byte size. Truncated version of "long" filename is stored in the last of these records; its structure corresponds to that shown in

table A.09-1. But the rest records, associated with the same file, store unicode characters of "long" filename. These rest LFN records have other structure, shown in the table below.

Offset	Size	Description
00h	1	Ordinal number of LFN record (note 1)
01h	10	First portion of "long" filename's characters
0Bh	1	= 0Fh – signature of LFN record
0Ch	1	= 00h (reserved)
0Dh	1	Checksum for short filename (note 2)
10h	12	Second portion of "long" filename's characters
1Ah	2	= 0000h for all LFN records
1Ch	4	Third portion of "long" filename's characters

Note 1: the last LFN record, associated with the same "long" filename, is marked by set state of bit 6 in the first byte.

Note 2: the short filename checksum byte is calculated by adding up the eleven bytes of the short filename, with rotating the intermediate sum right one bit before adding each next character byte.

A.09-4 Bitfields of access and sharing byte

While preparing an object for access, the INT 21\AH=3Dh (8.02-33) and INT 21\AX=6C00h (8.02-78) functions accept a byte of access conditions. This byte is written into a cell at offset 02h in corresponding SFT entry (A.01-4). Role of bitfields in access and sharing conditions byte is shown in the table below.

Bits	Description
1-0	Access conditions: 00b – for reading only 01b – for writing only 10b – for reading and writing 11b – for execution and transfer
2	If set, prohibits updating file's last-access time
3	= 0b (reserved)
6-4	Sharing conditions: 000b – compatibility mode 001b – prohibit access for others 010b – prohibit write access for others 011b – prohibit read access for others 100b – allow full access for others
7	If set, file's handle will not be inherited by child processes.

Note 1: sharing conditions are ignored unless SHARE.EXE utility is loaded.
Note 2: previous versions of MS-DOS require clear state of bit 2.

A.09-5 Unopened file control blocks

File control block (FCB) is an obsolete form of object's properties specification. It gives no access to objects beyond current directory and to disks with FAT-32 file system. Nevertheless some functions employ partially filled (unopened) FCBs just as a template of specification for object's search, renaming and deletion. Unlike operations with completely filled (opened) FCB blocks, operations with unopened FCBs (INT 21\AH=11h, 12h, 13h, 17h) are still used and can be applied to objects in the current directory on disks with FAT-32 file system. Unopened FCB's data structure is shown in the table below.

In MS-DOS7 two forms of FCB blocks are allowed: normal FCB blocks up to 36 bytes long and extended FCBs up to 43 bytes long. A distinctive feature of extended FCB is FFh value of its first byte. Normal FCB blocks define files only, except those having "Hidden" and "System" attributes. Extended FSBs include search attributes specification and therefore may be applied to different objects: files, volume labels and subdirectories. Both normal and extended FCBs may be completely filled (opened) and partially filled (unopened). Column "N" of the table below shows data structure in unopened normal FCB blocks, column "E" shows the same for unopened extended FCB blocks. Those FCB bytes, which are not shown in the table below, must have the 00h value.

N	E	Size	Description
	00h	1	= FFh – signature of extended FCB
	06h	1	Attributes specification for search (A.09-2)
00h	07h	1	Logical disk number: 00h = "current" disk, 01h = A:, 03h = C:, and so on (except the FFh value).
01h	08h	8	Object's name or its search mask (note 1)
09h	10h	3	Object's suffix or its search mask (note 1)
0Ch	13h	1	On return: search attributes (from offset 06h)
0Dh	14h	2	On return: object's record number in directory
0Fh	16h	2	On return: current directory's cluster number
11h	18h	8	On call for INT 21\AH=17h: new name for file
15h	1Ch	1	On return: disk number (01h=A:, 03h=C:, and so on)
19h	20h	3	On call for INT 21\AH=17h: new suffix for file

Note 1: in FCBs all characters of name and suffix must be in upper case. Name is appended with spaces (20h) to its nominal length 8 bytes, suffix is appended with spaces to its nominal length 3 bytes. If empty, both name and suffix fields must

be filled with spaces. The mentioned and some other requirements to filling FCB fields can be met by means of INT 21\AH=29h function (8.02-19).

Note 2: being called for the first time, functions INT 21\AH=11h (8.02-11) and INT 21\AH=13h (8.02-13) require 00h values in all fields after offset 0Ch in normal FCB and after offset 13h in extended FCB. On return these fields contain data, which must be preserved intact from each previous search call to each next search call. In the same FCB fields the INT 21\AH=17h function (8.02-14) accepts new name for the renamed file, requiring buffer 28 bytes long for normal FCB and 35 bytes long for extended FCB.

Note 3: unopened FCB blocks are not subjected to restriction, imposed by FCBS command specification (4.10) in CONFIG.SYS file.

A.09-6 Canonical structure of a CD directory record

High Sierra and ISO 9660 file systems implement slightly different data structures in CD directory records. Both these data structures can be translated by INT 2F\AX=150Fh function (8.03-19) to a common canonical form, which is shown below.

Offset	Size	Description
00h	1	Length of attribute record in logical blocks
01h	4	File's first logical block number
05h	2	Size of file in logical blocks
07h	4	File's length in bytes
0Bh	7	Date and time
12h	1	Bit flags
13h	1	Interleave size (for AVI files only)
14h	1	Interleave skip factor (for AVI files only)
15h	2	Volume set sequence number
17h	1	Length of file name
18h	38	Name of file, ending with 00h byte
3Eh	2	File version number
40h	1	Number of bytes in system data block
41h	220	System data block

A.10 Video data tables

A.10-1 Selected videomodes

Videomodes define screen appearance. Both BIOS and DOS use textual videomodes: color videomode 03h or monochrome videomode 07h. Each program is allowed to set the most appropriate video mode, either textual or graphic.

Available videomodes depend on PC's hardware. During hardware evolution some videomodes have become common for the sake of compatibility. Later a subset of videomodes has acquired the status of a standard. The table below lists only those videomodes, which are almost certainly supported by any modern video card. Video cards with insufficient internal memory probably will not be able to support graphic videomodes with high resolution. Obsolete PC's produced before 1991 don't support SVGA videomodes at all.

EGA and VGA videomodes are defined by one-byte code, specified in the first column of the table below. These videomodes may be set by INT 10\AH=00h function (8.01-10).

SVGA video modes are defined by 2-byte hexadecimal code, which should be specified in BX register for INT 10\AX=4F02h function (8.01-37). The table below doesn't specify the most significant half-byte of SVGA videomode code, because this half-byte (bits 15 – 12) is charged with another mission. Its 12-th and 13-th bits must be cleared, the 14-th bit enables linear frame buffer access, and the 15-th bit forces to retain video memory contents. For example, you may specify BX=0102h, when you want video memory to be cleared, or BX=8102h if you want video memory contents to be preserved: in both cases you'll get the same videomode, which is specified as 102h in the first column of the table below. Codes of other SVGA videomodes are shown in the same way – without specification of the most significant half-byte. Codes of non-SVGA video modes may be specified for INT 10\AX=4F02h function in the least significant byte of BX register while its bits 15 and 14 are charged with the described missions, and bits 13 – 8 are made clear.

Monochrome videomodes, both textual and graphic, are marked in the second column of the table as "b/w" (instead of colors number).

Textual video modes are characterized in column 3 of the table below by number of characters in a row and by number of rows per screen height. For example, definition 80x25 means that you may address rows 0 – 24 and character cells 0 – 79 in each row. All listed textual modes accept 8x16 fonts.

Graphic modes are characterized by their resolution in pixels, shown in 4-th column. For example, resolution 640x480 means that you are allowed to address screen lines 0 – 479 and pixels 0 – 639 in each line.

The 5-th column in the table shows video buffer starting address for those videomodes, which use fixed video buffer in UMB address space.

Videomode	Colors	Text	Graphics	Buffer	Class
01h	16	40x25		B800	VGA
03h	16	80x25		B800	VGA
06h	b/w		640x200	B800	EGA,VGA
07h	b/w	80x25		B000	VGA
0Eh	16		640x200	A000	EGA,VGA
0Fh	b/w		640x350	A000	EGA,VGA
10h	16		640x350	A000	VGA
11h	b/w		640x480	A000	VGA
12h	16		640x480	A000	VGA
13h	256		320x200	A000	VGA
100h	256		640x400	*1	SVGA
101h	256		640x480	*1	SVGA
102h	16		800x600	*1	SVGA
103h	256		800x600	*1	SVGA
104h	16		1024x768	*1	SVGA
105h	256		1024x768	*1	SVGA
108h	16	80x60		*1	SVGA
109h	16	132x25		*1	SVGA
10Ah	16	132x43		*1	SVGA
10Bh	16	132x50		*1	SVGA
10Ch	16	132x60		*1	SVGA
110h	32k		640x480	*1	SVGA
111h	64k		640x480	*1	SVGA
112h	16M		640x480	*1	SVGA
113h	32k		800x600	*1	SVGA
114h	64k		800x600	*1	SVGA
115h	16M		800x600	*1	SVGA
116h	32k		1024x768	*1	SVGA
117h	64k		1024x768	*1	SVGA
118h	16M		1024x768	*1	SVGA
119h	32k		1280x1024	*1	SVGA
11Ah	64k		1280x1024	*1	SVGA
11Bh	16M		1280x1024	*1	SVGA
120h	256		1600x1200	*1	SVGA

Note 1: position and size of video memory access "windows" in address space for SVGA videomodes may depend on PC's hardware. Video memory access parameters should be determined by call for INT 10\AX=4F01h function (8.01-36, A.10-7).

Note 2: when SVGA standard hasn't been adopted yet, then equivalent to SVGA's 102h videomode was 6Ah videomode (800x600x16). The 6Ah videomode still can be set by INT 10\AH=00h function (8.01-10).
Note 3: SVGA standard reserves BX=81FFh code for special video mode, enabling unlimited direct access to video memory.

A.10-2 Information about video adapter status

The table below shows structure of 64-byte data block, returned by INT 10\AH=1Bh function (8.01-34). This block presents information about current status of video adapter.

Offset	Size	Description
00h	4	Address of static functionality table (A.10-3)
04h	1	Current video mode
05h	2	Number of columns or of pixels along a line
07h	2	Size of regeneration buffer in bytes
09h	2	Starting address of regeneration buffer
0Bh	16	Cursor positions (2 bytes each) for pages 0 – 7
1Bh	2	Cursor's start and stop scan lines
1Dh	1	Active screen page
1Eh	2	CRT controller's port address
20h	2	Last values sent to ports 03x8h and 03x9h
22h	1	Number of rows (or screen lines) minus one
23h	2	Number of bytes per font's character
25h	1	Active video adapter code
26h	1	Code of alternate video adapter (if it exists)
27h	2	Number of videomode's colors (0000h = monochrome)
29h	1	Number of screen pages supported by videomode
2Ah	1	Active scan lines (note 1)
2Bh	1	Character generator's primary font block
2Ch	1	Character generator's secondary font block
2Dh	1	Current status flags (note 2)
31h	1	Video memory, 00h – 03h correspond to 64,128,192,256k
32h	1	Flags, just as at offset 0Eh in table A.10-3.

Note 1: number of active scan lines is defined by set state of one bit in a byte at offset 2Ah. Set state of bit 0, 1, 2, 3, 4, 5, 6 corresponds to numbers of lines 200, 350, 400, 480, 512, 600, 768.
Note 2: bits in flag's byte at offset 2Dh have the following meaning:
bit 0 – no restrictions on videomode choice
bit 1 – gray scale summing is on
bit 2 – monochrome display attached

bit 3 – default palette loading disabled
bit 4 – cursor emulation enabled
bit 5 – role of 7-th bit in color byte (A.10-5)
bit 6 – 9-dot wide fonts are not supported

If 5-th bit in flag's byte is cleared, then 7-th bit in color byte defines brightness of background, otherwise it defines blinking.

A.10-3 Format of static functionality table

Static functionality table informs about variety of capabilities, potentially supported by PC's video adapter. A pointer to static functionality table is returned by INT 10\AH=1Bh function (8.01-34) in current video adapter status table (A.10-2) at offset 00h.

Offset	Size	Description
00h	7	Bits 0 – 13h correspond to video modes 00h – 13h; if a bit is set, the corresponding video mode is supported. The rest bits are reserved for OEM videomodes.
07h	1	Bits 0, 1, 2, 3, 4, 5, 6 signify support to scan lines numbers 200, 350, 400, 480, 512, 600, 768.
08h	1	Maximum number of fonts in textual videomodes
09h	1	Maximum number of active fonts in textual videomodes
0Ah	2	Supported operations: bit 0 – all modes on all displays supported bit 1 – gray summing function supported bit 2 – character font loading function supported bit 3 – default palette loading enable/disable supported bit 4 – cursor emulation function supported bit 5 – internal EGA palette present bit 6 – color palette present bit 7 – color-register paging function supported bit 8 – light pen supported (INT 10\AH=04h) bit 9 – save/restore state function 1Ch supported bit 10 – intensity/blinking switching supported (A.10-5) bit 11 – more than one video adapter supported
0Eh	1	Fonts and palettes support: bit 0 – 512-character sets supported bit 1 – dynamic determination of save area supported bit 2 – textual font override supported bit 3 – graphics font override supported bit 4 – palette override supported bit 5 – video adapter code extensions supported

A.10-4 BIOS information about SVGA extensions

The table below shows selected data from a data block 512 bytes long returned by INT 10\AX=4F00 function (8.01-35). These data characterize software supplied in fixed storage chip(s) of video adapter.

Offset	Size	Description
00h	4	Signature "VESA" or "VBE2"
04h	2	Version number of SVGA BIOS extensions
06h	4	Pointer to manufacturer's (OEM) name
0Eh	4	Pointer to list of supported videomodes (end mark FFFFh)
12h	2	Amount of video memory in 64 kb blocks

A.10-5 16-color codes

Though AT-compatible computer's hardware suggests a large variety of videomodes, the default videomode for both BIOS and MS-DOS7 is a 16-color 80x25 textual videomode 03h. For this videomode and for all other 16-color videomodes (A.10-1) colors are defined by a 4-bit code, shown in the table below.

0000	0	black	1000	8	gray
0001	1	blue	1001	9	bright blue
0010	2	green	1010	10	bright green
0011	3	cyan	1011	11	bright cyan
0100	4	red	1100	12	bright red
0101	5	magenta	1101	13	bright magenta
0110	6	brown	1110	14	yellow
0111	7	white	1111	15	bright white

Color codes are used to compose color bytes, also known as display attribute bytes. Video memory in textual videomodes is filled with alternating color bytes and character bytes. In each color byte bits 3 – 0 define foreground (character's) color, and bits 6 – 4 define background color. By default the 7-th bit defines character's blinking instead of background's brightness, but role of the 7-th bit may be reprogrammed by INT 10\AX=1003h function (8.01-23), and then the most significant bits in both 4-bit groups will have the same mission. By default the 3-rd bit defines foreground brightness, but this role also may be reprogrammed by INT 10\AX=1103h function (8.01-28), and then bit 3 will redirect character generator to another font block, thus enabling to display characters from two fonts at the same time.

A.10-6 Video data fields in BIOS data area

The table below shows those selected items in BIOS data area, which have relation to computer's video subsystem. All offsets in the table are counted from the start of BIOS data area (A.01-1) at segment address 0040h.

Offset	Size	Description
10h	2	Bits 5 –4 define initial video mode: 00b – according to video adapter settings 01b – 40x25 textual CGA color mode 10b – 80x25 textual CGA color mode 11b – 80x25 monochrome textual mode
49h	1	Current video mode (A.10-1)
4Ah	2	Number of columns (or pixels) per screen width
4Ch	2	Video buffer's screen page size (in bytes)
4Eh	2	Current page start address in video buffer
50h	16	Cursor XY positions on each of 8 video pages
60h	2	Cursor start and end scan lines (INT 10/AH=01h)
62h	1	Active screen page number
63h	2	CRT controller base I/O port address (usually 03D4h)
65h	1	Last control byte value sent to port 03B8h/03D8h: bit 5 – blinking control (INT 10\AX=1003h)
66h	1	Last control byte value sent to I/O port 03D9h: bit 4 – background brightness
84h	1	Number of rows (or lines) on screen minus one
85h	2	Font height in scan lines
87h	5	Video adapter control flags: bit 0: – cursor emulation disabled bit 1: – monochrome display attached bit 2: – wait for CRT display enable bit 7: – don't clear RAM on videomode set
A8h	4	Pointer to VGA video pointers table

Note 1: presented data placement may depend on BIOS version (A.01-1).

A.10-7 Features of requested SVGA videomode

This table presents selected data from a 256-byte data block, returned by INT 10\AX=4F01 function (8.01-36) in response to a request about any SVGA videomode, supported by computer's hardware.

Offset	Size	Description
00h	2	Flags: bit 0 – requested videomode is supported bit 2 – functions 8.01-21, 8.01-33 are supported bit 3 – set for color videomode bit 4 – set for graphic videomode bit 5 – videomode differs from VGA standard bit 6 – memory banks switching isn't supported bit 7 – linear frame buffer is supported
02h	1	Window "A": bit 0 – sliding window "A" is active bit 1 – window "A" is readable bit 2 – window "A" is writable
03h	1	Window "B": the same, as at offset 02h for window "A"
04h	2	Shift step (in kb) of windows "A", "B" in video memory
06h	2	Size of sliding windows "A" and "B" (in kb)
08h	2	Segment address of window "A" in CPU's address space
0Ah	2	Segment address of window "B" in CPU's address space
0Ch	4	Direct call address for sliding windows positioning program, similar to INT 10\AX=4F05h (8.01-39)
10h	2	Number of video memory bytes per one screen line
12h	2	Screen line length in pixels for graphic videomodes or in character cells for textual videomodes
14h	2	Screen height in pixels for graphic videomodes or in character cells for textual videomodes
16h	1	Character cell width (in pixels)
17h	1	Character cell height (in pixels)
18h	1	Number of video memory planes
19h	1	Number of video memory bits per one pixel
1Ah	1	Number of video adapter's memory banks
1Bh	1	Video memory filling model: 00h – textual, alternate character and color bytes 03h – 16-color graphic EGA model 04h – graphic model with "packed" pixels 06h – 3 color bytes per pixel (HiColor) 07h – luminance-chrominance model (YUV/YIQ)
1Ch	1	Video memory bank size (in kilobytes)
1Dh	1	Number of screen pages
28h	4	Physical address of linear video buffer (VBE v2.0)

A.11 PC's hardware specifications

A.11-1 Hardware configuration word

Hardware configuration word is returned by INT 11 handler (8.01-42); it reads this word in BIOS data area (A.01-1) at address 0040:0010h (exact address may depend on BIOS version). Bifields in hardware configuration word should be interpreted according to the following table.

Bits	Description
0	PC can be booted from existing floppy drive
1	Math coprocessor is present
2	BIOS controlled pointing device (mouse) is attached
4-5	Code of initial video mode (A.10-1)
6-7	Number of floppy drives minus one (if bit 0 is set)
9-11	Number of available serial ports (COM-ports)
12	Game port is present (for joystick)
13	Internal modem is present
14-15	Number of available parallel ports (LPT-ports)

A.11-2 PC model identifiers for HIMEM.SYS driver

In order to enable access to computer's extended memory, HIMEM.SYS driver (5.04-01) has to determine CPU model. However, in some computers HIMEM.SYS can't determine CPU model properly, and then computer's identifier or its numerical code should be specified explicitly in driver's command line.

The table below presents identifiers and corresponding numerical codes for computers, which don't ensure CPU determination for at least some versions of HIMEM.SYS driver. The first place in this table (code 1) is an exception: IBM AT is a determinable model, it represents the default choice. Latest versions of HIMEM.SYS driver are able to detect properly most part of PC types, specified in this table, except Acer 1100, Wyse, and IBM 7552.

Identifier	Code	PC model
at	1	IBM PC AT and compatible models
ps2	2	IBM PS/2
ptlcascade	3	Phoenix Cascade BIOS

Continuation of table A.11-2

hpvectra	4	HP Vectra (A & A+)
att6300plus	5	AT&T 6300 Plus
acer1100	6	Acer 1100
toshiba	7	Toshiba 1600 & 1200XE
wyse	8	Wyse 12.5 Mhz 286
tulip	9	Tulip SX
zenith	10	Zenith ZBIOS
at1	11	reserved by IBM
at2	12	reserved by IBM
css	12	CSS Labs
at3	13	reserved by IBM
philips	13	Philips
fasthp	14	HP Vectra
ibm7552	15	IBM 7552 Industrial Computer
bullmicral	16	Bull Micral 60
dell	17	Dell XBIOS

A.11-3 Keyboard controller

Keyboard controller is a chip on computer's motherboard. Though types of keyboard controllers may be different, their mission and their interface in all AT-compatible computers are unified. Main interaction between CPU and keyboard controller occurs via ports 60h and 64h.

Port 64h is always opened for reading keyboard controller's current status by IN command (7.03-26). Set state of bits in status byte, read from port 64h, should be interpreted as follows:

bit 7 – an error has occurred in data sent from keyboard
bit 6 – keyboard doesn't respond to controller
bit 4 – keyboard is blocked with ADh command
bit 2 – keyboard's self-test has been successful
bit 1 – previous operation isn't completed yet
bit 0 – a key code is prepared for reading in port 60h

At each keystroke and at each key release keyboard controller exhibits transformed key code in port 60h and just after that announces its readiness for reading via bit 0 in port 64h and via bit 4 of controller's output line (note 1 below). The latter signal invokes INT 09 handler (8.01-09), which reads the prepared byte from port 60h. Each reading access to port 60h clears bit 0 in port 64h.

Besides that, port 64h receives operation codes, sent to keyboard controller with OUT command (7.03-66). As far as keyboard controller is much slower, than CPU, before sending an operation code the CPU must wait until bit 1 will be cleared in status byte, read

from the same port 64h: it will signify, that keyboard controller has finished its previous operation and is ready to receive the next operation code for execution. Codes of some important operations, which may be sent to keyboard controller's port 64h, are shown in the table below.

Code	Operation
ADh	Block the keyboard (switch it OFF)
AEh	Activate the keyboard (switch it ON)
D1h	Open port 60h for data reception (note 1)
EDh	Open port 60h for data reception (note 2)
FEh	Send reset signal to CPU (note 3)

Note 1: having received operation code D1h via port 64h, keyboard controller begins to wait for reception of data byte via port 60h and then transfers the received data byte to its output bus. Bits of data byte are distributed among output bus lines in the following way:

bit 7 – command output to keyboard via data line;
bit 6 – clock output to keyboard via clock line;
bit 4 – a call for INT 09 (8.01-09) via line IRQ 1;
bit 1 – to gate of CPU's address line A20;
bit 0 – to CPU's reset pin.

Active state of controller's output lines corresponds to cleared states of bits in data byte; hence, sending a data byte with clear bit 0 is not allowed – CPU will get blocked. Because of the same reason for opening A20 line gate a data byte FFh should be sent to port 60h, and for closing access to HMA – data byte FDh.

Note 2: having received operation code EDh via port 64h, keyboard controller begins to wait for reception of data byte via port 60h and then transfers bit states of the received data byte to control lines of keyboard's LED indicators, in particular:

bit 2 – to Caps Lock indicator;
bit 1 – to Num Lock indicator;
bit 0 – to Scroll Lock indicator.

Indicator will be lit, if corresponding bit in data byte is set. Not mentioned bits in this data byte must be cleared.

Note 3: having received operation codes F0h – FFh via port 64h, keyboard controller sends four least significant bits of this operation code to lines 3 – 0 of its output bus. Unlike to response on reception of operation code D1h, after reception of operation codes F0h – FFh the imposed states of output lines are not fixed for ever, but are kept for about 6 milliseconds as a solitary pulse. In particular, operation code FEh causes a pulse sent to CPU's reset pin, just as after a press on RESET button on a face panel of computer's system block. Some ways to affect further events after CPU's reset are described in note 4 to A.12-1.

A.11-4. CPU's flags register

Former 16-bit flags register in modern processors, starting from model 80386, has been expanded to 32 bits; besides that, control registers CR0, CR2 and CR3 have been introduced. Later, starting from Pentium CPU, one more control register CR4 has been added. Flags are present in all of the mentioned control registers, except CR2: it stores linear address of that last command, which requested access to a forbidden memory page. Missions of flags in flags register, and also of some flags in control registers are shown in the table below.

Register	Bit	Description	Comments
FLAGS	00h	CF – carry flag	note 1
FLAGS	02h	PF – bit's parity in the least significant byte	note 1
FLAGS	04h	AF – intermediate carry flag	note 1
FLAGS	06h	ZF – zero (or equality) flag	note 1
FLAGS	07h	SF – sign flag	note 1
FLAGS	08h	TF – trace (step-by-step) flag	
FLAGS	09h	IF – interrupt enable flag	note 1
FLAGS	0Ah	DF – index count direction flag	note 1
FLAGS	0Bh	OF – overflow flag	note 1
FLAGS	0Ch	I/O privilege level 2-bit field	note 2
FLAGS	0Eh	Nested task flag	note 3
FLAGS	0Fh	Distinctive feature of CPU 8086	note 3
EFLAGS	10h	Ignore debugging flag (A.11-5)	note 4
EFLAGS	11h	VM – V86 mode flag	notes 4, 5
EFLAGS	12h	AC – alignment check enable flag	notes 4, 6
EFLAGS	13h	VIF – virtual interrupt flag	note 4
EFLAGS	14h	VIP – virtual interrupt pending	note 4
EFLAGS	15h	ID – CPU identification flag	notes 3, 4
CR0	00h	PE – protection enable flag	note 7
CR0	01h	Coprocessor synchronization (7.02-05)	note 7
CR0	02h	Coprocessor emulation via INT 07	note 7
CR0	03h	TS – task switched flag	note 7
CR0	04h	Coprocessor's commands support	note 7
CR0	05h	Exception enable on coprocessor's errors	note 7
CR0	10h	WP – write protection	notes 7, 8
CR0	12h	AM – alignment mask	notes 7, 6
CR0	1Dh	NW – not write-through cache	note 7
CR0	1Eh	CD – cache disable	note 7
CR0	1Fh	PG – paging enabled	note 7
CR3	03h	Cache write-through	notes 7, 9

Continuation of table A.11-4

CR3	04h	Page caching disabled	notes 7, 9
CR4	00h	VME – V86 mode extensions enabled	note 7
CR4	01h	PVI – virtual interrupts enabled	note 7
CR4	02h	TSD – time stamp disabled	note 7
CR4	03h	Debugging: allow INT 01 on port calls	note 7

Note 1: mission and states of this flag are described in article 6.05-15.

Note 2: bits 0Ch and 0Dh in flags register express required privilege level for performing I/O operations. By default under DOS both bits 0Ch and 0Dh are set: hence, direct I/O operations are allowed for all processes. But only processes with the highest (zero) privilege level are allowed to alter states of 0Ch and 0Dh bits with POPF command (7.03-68). The latter feature enables to determine whether the current process indeed is performed at the highest privilege level.

Note 3: flags register enables to perform a coarse identification of CPU type. Inability to clear the 0Fh flag is a distinctive feature of obsolete 8086 CPU. Inability to set the 0Eh flag is specific for 16-bit processors. If CPU is able to set the 0Eh flag, hence it is a 32-bit CPU and is equipped with extended EFLAGS register. In the latter case bit 15 in EFLAGS register will show, whether this CPU is able to respond properly to CPU identification command (CPUID, machine code 0Fh A2h)

Note 4: EFLAGS is a 32-bit extension of 16-bit flags register. In real mode access to EFLAGS's bits 31 – 16 can be provided by PUSHF and POPF commands, preceded by prefix 66h, as it is described in article 7.02-06.

Note 5: an opportunity to set V86 mode with POPF command is blocked by hardware. Nevertheless V86 mode can be set from stack with IRET command, if CPU is in protected mode and if at the same time the 06 bit in segment descriptor allows 32-bit addressing (A.12-2).

Note 6: here alignment implies that address of each operand in memory must be a multiple of this operand's size (in bytes). Alignment check can be performed at the lowest (the third) privilege level only, when CPU is in protected mode (note 1 to 8.01-42). Alignment mask bit in CR0 enables alignment exceptions even if these are not enabled by AC flag in EFLAGS register.

Note 7: control registers can be accessed with MOV command (note 1 to 7.03-58). Besides that, contents of CR0 register can be read by INT 67\AX=DE07h function (8.03-72).

Note 8: bit 10h is used in order to protect application programs segments from being accessed for writing to operating system or to other processes, which may have higher privilege level.

Note 9: 20 most significant bits of CR3 register store base address of page directory. This base address must be a multiple of page size (normally 4 kb). Each writing operation to CR3 register causes updating of TLB buffer contents; this should be done after every change in page address translation table(s).

A.11-5 CPU's debugging registers

Debugging registers enable to call for INT 01 handler (8.01-02) each time a particular target is addressed: a port or a prescribed memory region, including non-writable regions of address space, where breakpoints can't be stored. All modern processors of x86 platform, starting from model 80386, are equipped with debugging registers DR0 – DR7. Access to debugging registers is provided by INT 67\AX=DE08h-DE09h functions (8.03-73), and also by MOV command (note 1 to 7.03-58).

Registers DR0 – DR3 store four 32-bit absolute linear addresses of prescribed target points. The DR7 register defines conditions of access event enrolment. The DR6 register stores some circumstances of happened access event (programmable and external interrupts don't affect DR6 contents). Missions of selected bitfields in DR6 and DR7 registers are shown in the following table.

Register	Bit	Description	Comments
DR6	00h	Event has occurred at DR0 address	note 1
DR6	01h	Event has occurred at DR1 address	note 1
DR6	02h	Event has occurred at DR2 address	note 1
DR6	03h	Event has occurred at DR3 address	note 1
DR6	0Dh	Breakpoint debug access detected	note 1
DR6	0Eh	BS – single step state detected	note 1
DR6	0Fh	TS – task switch state detected	note 1
DR7	00h	2-bit permission field for DR0	note 2
DR7	02h	2-bit permission field for DR1	note 2
DR7	04h	2-bit permission field for DR2	note 2
DR7	06h	2-bit permission field for DR3	note 2
DR7	0Dh	GD – general detect enabled	note 3
DR7	10h	4-bit control field for DR0	note 4
DR7	14h	4-bit control field for DR1	note 4
DR7	18h	4-bit control field for DR2	note 4
DR7	1Ch	4-bit control field for DR3	note 4

Note 1: access event is fixed in bits 01h – 03h of DR6 register even when exception generation is not permitted by bit 0Dh in DR7 register or by bit 10h in EFLAGS register (A.11-4). Bit 0Eh in DR6 register fixes state of TF flag at the moment of access event, bit 0Fh – state of task switch. Set state of bit 0Dh in DR6 register reminds that exception hasn't been generated yet, though access event has happened (bit 0Dh is cleared by exception).

Note 2: permission field defines either local or global permission to enroll access events. The first of bits in permission field acts locally within current task only and is turned off at each task change. The second bit in permission field is imparted with global property and enables to enroll access events beyond the current task.

Note 3: bit 0Dh in DR7 register doesn't affect access event enrolment, but rather allows exception generation, caused by access event. State of 0Dh bit in DR7 can't be changed unless the process either has the highest privilege level or is executed in real mode.

Note 4: the first pair of bits in each control field defines purpose of those access events, which should be intercepted:

00 – attempt to execute machine code
01 – attempt of writing into memory
10 – I/O address to a port (for CPUs Pentium+)
11 – both reading and writing attempts.

The second pair of bits in each control field defines size of the monitored address space: a byte, a word or a double word. Attempt of access to either byte within monitored address space is equally enrolled as access event.

A.12 Memory allocation and management

A.12-1 General memory map

This table shows general allocation of memory space below 1 Mb, typical for AT-compatible computers, controlled by DOS operating system. However, memory allocation depends on BIOS version, on BIOS Setup settings, on particular computer's configuration. Therefore some features of memory allocation in your computer may differ from those shown below.

Address	Size	Description
0000:0000	400h	Interrupt table for real mode
0000:0074	4	Pointer to video register's default settings
0000:0078	4	Pointer to default floppy data table (A.08-2)
0000:007C	4	Pointer to 8x8 graphic font characters 80 – FFh
0000:0104	4	Pointer to 1-st HDD parameters table (A.13-1)
0000:010C	4	Pointer to current graphic font (8.01-30)
0000:0118	4	Pointer to 2-nd HDD parameters table (A.13-1)
0040:0000	100h	BIOS data area (A.01-1)
0050:0000	1	Printer's status for INT 05 (8.01-06)
0050:0004	1	Floppy drive choice (A: or B:)
0050:0040	BCh	Selected pointers to original interrupt handlers (note 6)
0000:7C00	200h	Default area to load and execute boot record
9000:FFFF	–	Upper boundary of "conventional" memory (note 1)
A000:0000	10000h	Video memory access "window" (note 2)
B000:0000	10000h	Video memory access "window" (note 2)

Continuation of table A.12-1

B800:0000	8000h	Video buffer for textual videomodes EGA+
C000:0000	8000h	Video adapter's BIOS area (note 3)
C000:0070	7	"EXTMODE" signature: SVGA videomodes support
C800:0000	4000h	Hard disk's BIOS area
D000:0000	10000h	Default area for UMBs, arranged by EMM386.EXE
E000:0000	10000h	Default area for expanded memory pages
F000:0000	FFFFh	Relocated copy of PC's ROM BIOS (note 3)
F000:FFF0	–	Reboot program's entrance point (note 4)
F000:FFF5	8	BIOS date
F000:FFFD	1	BIOS code checksum
F000:FFFE	1	Computer's model code
FFFF:0010	FFEFh	High memory area (note 5)

Note 1: the 640 kb boundary of conventional memory is hardware defined by dynamic memory controller chip in motherboard's chipset. Above this boundary the next 384 kb of address space are reserved for video memory and for BIOS ROM chips. Free space in this area normally is made accessible in protected mode due to address translation mechanism in CPU.

Note 2: address space area A000:0000 – B000:FFFF provides access to video memory. Particular usage of this area depends on videomode (A.10-1). SVGA BIOS of modern video adapters arranges in this area one or two "sliding" windows, providing "sliding" access to selected part(s) of large video memory (some details – in article 8.01-39).

Note 3: access via the same areas of address space may be arranged either directly to BIOS and video BIOS codes in ROM chips or to copies of these codes in a more fast RAM. A choice of a particular alternative depends on "shadowing" parameters settings for corresponding memory areas, set by BIOS Setup program.

Note 4: reset program's entrance point address F000:FFF0h is hardware defined by CPU: at power-on the initial state of its address bus is just FFFF0h. Further booting process depends on value written in byte 0Fh in BIOS's CMOS RAM (note 1 to A.14-1):

- 00h – ordinary booting with POST test
- 04h – reboot with a call for INT 19 (8.01-90)
- 05h – reset and jump to address in 0040:0067 cell (A.01-1)
- 0Ah – jump to address prepared in 0040:0067 cell (A.01-1)

Unlike booting after power-on, POST test after reboot depends on a word at address 0040:0072h (note 1 to A.01-1). Alternatives 05h and 0Ah are used after reboot only and differ in that whether interrupt controller will be reset or not.

Note 5: high memory area is accessed, when segment address summation with offset produces a carry bit, directed into A20 line of address bus. High memory area is

accessible in real mode, but needs HIMEM.SYS driver (5.04-01) to be installed, which provides control over A20 line gate.

Note 6: the 0050:0040 – 0050:00FB area stores copies of selected interrupt handler's addresses (INT 00 – INT 1F, INT 40 – INT 43, INT 46, INT 70 – INT 77), prepared by BIOS for subsequent loading of operating system. In main interrupt table these addresses may be overwritten by addresses of other handlers, installed later either by MS-DOS7 itself, or by TSR programs, or by drivers.

A.12-2 Segment descriptors

Segment boundaries and access rights in protected mode are defined by segment descriptors. A number of most important segment descriptors constitute global descriptor table (GDT). CPU transition to protected mode implies presence of a GDT, at least partially prepared beforehand, while CPU is in real mode. Order and selection of descriptors in the prepared GDT depend on requirements of that procedure, which should control CPU transition to protected mode.

Examples of GDT tables for different procedures are shown in articles 8.01-76, 8.01-78 and 9.10-01. In all GDT tables the first descriptor must be filled with zeros: it is a template for non-requested segments and memory pages. All descriptors have the same internal structure, shown in the following table. Least

Offset	Size	Description
00h	2	Less significant 2 bytes of segment size, the least first
02h	3	Less significant 3 bytes of base address, the least first
05h	1	Access rights byte (note 2): bit 0: = 0 – segment hasn't been accessed yet = 1 – segment has been accessed bit 1: = 0 – data reading or code execution only = 1 – data writing and code reading allowed bit 2: – direction of expansion (note 3) bit 3: = 0 – segment contains data = 1 – segment contains executable code bit 4: = 0 – system descriptor's marker = 1 – application descriptor's marker bits 5 – 6: – privilege level: = 00 – highest = 11 – lowest bit 7: = 0 – segment must be read from disk = 1 – segment is present in RAM
06h	1	bits 0 – 3: – most significant 4 bits of segment size bit 4: – free bit (note 4):

Continuation of table A.12-2

		bit 5: = 0 (reserved) bit 6: – address and operand's size (note 5): = 0 – 16-bit addressing and operands = 1 – 32-bit addressing and operands bit 7: – granularity bit: = 0 – segment size count in bytes = 1 – segment size count in 4 kb units
07h	1	Most significant byte of base address

Note 1: bytes 06h and 07h of segment descriptor are taken into account by CPU models 80386 and higher. If program is expected to be executed by CPU 80286, then bytes 06h and 07h must be cleared. Zero values of bytes 06h and 07h are a distinctive feature of protected mode 16-bit programs, enabling their proper execution by 32-bit CPU models.

Note 2: interpretation of bits 0 – 3 in access rights byte depends on bit 4. The shown interpretation of bits 0 – 3 relates to application programs, including their data segments and executable code segments. In system descriptors bits 0 – 3 define 16 types of different descriptor's subtypes.

Note 3: interpretation of bit 2 in access rights byte depends on bit 3. In code segments cleared state of bit 2 means that code can be executed by programs of the same privilege level (otherwise code also can be executed by programs having higher privilege level). For data segments cleared state of bit 2 means normal expansion direction upwards, whereas set state of bit 2 means reverse expansion direction – downwards, as used in stack segments.

Note 4: bit 4 in byte 06h is available to programmer's regulation. In memory page descriptors this bit is used as a redefinition ban mark, for example, for mapping I/O address space into memory.

Note 5: size bit 6 in byte 06h of code segment descriptors defines sizes of both addresses and operands. In system segments bits 4 – 6 of byte 06h must be cleared.

A.12-3 Selected subfunctions of XMS-driver

XMS subfunctions are performed by extended memory driver HIMEM.SYS (5.04-01). Before these subfunctions can be used, two preliminary operations have to be done. First is to check with INT 2F\AX=4300h function (8.03-22) whether HIMEM.SYS driver is installed. The second operation is to find out with INT 2F\AX=4310h function (8.03-23) an address of XMS driver's entrance point. Returned double-word address should be specified for a CALL FAR command (7.03-08). On call the subfunction to be executed is defined by value in AX register, shown in the first column of the table below. DX register's mission is shown in second column, returned AX contents – in the fourth column. If AX returns status, AX=0001h means success, AX=0000h means failure. In case of failure

almost all subfunctions (except AH=00h) return error code (A.06-1) in BL register. Query subfunctions AH=08h and AH=88h return error code in BL register in any case.

AH	DX	Subfunction	AX on return	Comments
00h		Report XMS version	XMS version	note 1
05h		Turn ON the A20 line gate	status	
06h		Turn OFF the A20 line gate	status	
08h		Report free XMS-memory	largest block	note 2
09h	size	Allot XMS-memory block	status	note 3
0Ah	handle	Release XMS-memory block	status	
0Bh		Copying in XMS-memory	status	A.12-4
0Ch	handle	Lock XMS-memory block	status	note 4
0Dh	handle	Unlock XMS-memory block	status	
0Eh	handle	Get XMS handle information	status	note 5
0Fh	handle	Resize XMS-memory block	status	note 6
10h	size	Allot UMB memory block	status	notes 7, 8
11h	segment	Release UMB memory block	status	note 7
12h	segment	Resize UMB memory block	status	notes 7, 8
88h		Report free XMS memory	largest block	notes 2, 9
89h	size	Allot XMS-memory block	status	notes 3, 9
8Eh	handle	Get XMS handle information	status	notes 5, 9
8Fh	handle	Resize XMS-memory block	status	notes 6, 9

Note 1: this subfunction returns status of HMA area in DX register: DX=0001h signifies that HMA area is in use, DX=0000h – that HMA area is not used.

Note 2: on call subfunction 08h needs BL=00h. On return size of free XMS memory (in kilobytes) is reported in DX register, and size of the largest available XMS-memory block is reported in AX register. The 88h subfunction does the same, but returns similar results in 32-bit registers EDX and EAX. Besides that, subfunction 88h in ECX register returns maximum physical address, corresponding to furthermost available byte of XMS-memory.

Note 3: both 09h and 89h subfunctions accept requested size of XMS-memory block in kilobytes, but 09h subfunction accepts requested size from DX register, whereas 89h subfunction accepts requested size from 32-bit EDX register. Both 09h and 89h subfunctions return a handle for allotted XMS-memory block in DX register.

Note 4: in case of success registers DX:BX return 32-bit physical address of that memory block, which has been locked.

Note 5: the 0Eh subfunction returns in BH register a number of lock counts for the requested XMS block, in BL register – number of free handles, in DX register – size in kilobytes of XMS block, opened for access by the specified handle. The 8Eh subfunction does the same, but returns number of free handles in CX register, and size of XMS block – in EDX register.

Note 6: resizing subfunction 0Fh accepts new size (in kilobytes) for the requested memory block from BX register. The 8Fh subfunction does the same, but accepts new size from EBX register. Requested memory block must not be locked.

Note 7: for CPU models 80386 and higher the 10h – 12h subfunctions usually are implemented by means of address translation in CPU's TLB buffer. Therefore execution of these subfunctions is relegated to EMM386.EXE driver (5.04-02), which arranges address translation and intercepts address of direct calls for HIMEM.SYS driver. However, subfunctions 10h – 12h don't necessarily require switching CPU to protected mode and can be implemented in real mode by UMBPCI.SYS driver (5.04-04).

Note 8: both 10h and 12h subfunctions operate with UMB block size specifications in 16-byte units (paragraphs). Resizing subfunction 12h accepts requested new size from BX register. Allocation subfunction 10h in case of successful termination returns segment address of UMB block in BX register, and actual size of UMB block – in DX register. In case of a failure, marked by AX=0000h value, both 10h and 12h subfunctions return size of the largest available UMB block in DX register.

Note 9: unlike 0xh subfunctions, the 8xh subfunctions can't be implemented by obsolete 16-bit processors and require HIMEM.SYS driver's version not less than 3.07.

A.12-4 Format of XMS copy request

A pointer to this request data block is accepted from DS:SI registers by subfunction AH = 0Bh (A.12-3) of XMS driver HIMEM.SYS (5.04-01). Subfunction AH = 0Bh copies a group of bytes from one XMS block, addressed via source handle, to another XMS block, addressed via destination handle.

Offset	Size	Description
00h	4	Number of bytes to copy (must be even)
04h	2	Source handle
06h	4	Offset in source block
0Ah	2	Destination handle
0Ch	4	Offset in destination block

Note 1: if source and destination overlap, only forward copying (source base less than destination base) is guaranteed to work properly.

Note 2: if either handle in the request is 0000h, then the corresponding offset double-word is interpreted as ordinary address (segment: offset) inside directly addressable conventional memory.

A.12-5 Format of EMS copy descriptor

EMS copy descriptor specifies source and destination for copying and exchange functions INT 67\AX=5700h-5701h (8.03-69), performed by EMM386.EXE driver (5.04-02). Both source and destination may belong either to EMS memory page(s) or to conventional memory. In the latter case 0000h value should be written instead of corresponding handle number, and location should be specified by segment address in place of EMS logical page number.

Offset	Size	Description
00h	4	Length in bytes of the data block to be copied/exchanged
04h	1	= 00h: source block is in conventional memory = 01h: source block is in EMS-memory page
05h	2	Source handle (0000h if source in conventional memory)
07h	2	Source offset in page or in conventional memory segment
09h	2	Source logical page or segment in conventional memory
0Bh	1	= 00h: destination block is in conventional memory = 01h: destination block is in EMS-memory page
0Ch	2	Destination handle (0000h if destination in conventional memory)
0Eh	2	Destination offset in page or in conventional memory segment
10h	2	Destination logical page or segment in conventional memory

Note 1: for move operation the source and destination may overlap, but then only one direction of copying provides proper result.

A.12-6 Data block for jumps inside EMS memory

The shown data block specifies parameters of a call for subroutine inside EMS memory, performed by INT 67\AH=56h function (8.03-68) of EMM386.EXE driver (5.04-02). A far jump operation inside EMS memory, performed by INT 67\AH=55h function (8.03-68), uses a part of the shown data block up to offset 09h.

Offset	Size	Description
00h	4	Target address (segment: offset)
04h	1	Length of new page mapping list
05h	4	Pointer to new page mapping list
09h	1	Length of current page mapping list
0Ah	4	Pointer to current page mapping list
0Eh	8	(reserved for EMM386.EXE driver's data)

Note 1: internal structure of page mapping lists is described in note 3 to article 8.03-66. Data about current page mapping list, which is to be replaced, are needed for return to current program execution when subroutine's execution terminates.

A.12-7 Memory control descriptors

Computer's memory allocation is a prerogative of operating system. Each memory block, allotted by DOS, is preceded by a 16 bytes long memory control descriptor. These descriptors are also known as MCB (Memory Control Blocks). MCBs are easy to find: segment address of MCB descriptor is always a unity less, than segment address of allotted memory block, associated with this MCB descriptor.

DOS traces the whole available memory via a chain of MCB descriptors (more about that – in note 3 below). Free memory space beyond the allocated memory areas is considered by DOS as a separate memory block: it also must be preceded by MCB descriptor. A distinctive feature of MCB descriptor(s), associated with free memory space(s), is code 0000h instead of segment address of the owner program. All data about available free memory and about disposition of particular free memory areas DOS acquires from a traceable chain of MCB descriptors.

Data structure inside a MCB descriptor is shown in the following table.

Offset	Size	Description	Comments
00h	1	= 4Dh (= M) – not-the-last MCB = 5Ah (= Z) – the last MCB in a chain	note 1
01h	2	Segment address of owner program	note 2
03h	2	Size of associated memory block	note 3
05h	3	Not used	
08h	8	Name of program's file	note 4

Note 1: memory block with main DOS's system data is divided into subblocks, each having its own MCB descriptor, but with other identifiers in a byte at offset 00h:

- 42h (= B) – subblock with buffers (4.03)
- 44h (= D) – subblock for DOS's drivers
- 45h (= E) – data subblock for DOS's drivers
- 46h (= F) – subbblock with SFT table (4.12)
- 49h (= I) – subblock for IFS data
- 4Ch (= L) – subblock with CDS table (4.17)
- 53h (= S) – subblock for DOS's stacks (4.27)
- 54h (= T) – subblock for transition code(s)
- 58h (= X) – subblock for FCBS (4.10)

Note 2: if associated memory space is free, then in MCB descriptor a word at offset 01h is filled with zeros. If associated memory block is allocated by DOS to itself, then

code 0008h is written in a word at offset 01h instead of segment address of the owner program.

Note 3: size of associated memory block is specified in 16-byte units (paragraphs). Segment address of each next MCB descriptor is a unity greater than a sum of a number, specified at offset 03h in current MCB descriptor, with segment address of current MCB descriptor. On basis of this formula DOS traces a chain of MCB descriptors. Tracing starts from the first MCB descriptor; segment address of this first MCB descriptor is stored in a word just preceding DOS's List-of-Lists. This word is marked in table A.01-2 as having offset –02h.

Note 4: name of program's file is specified in those MCB descriptors, which are associated with PSP memory blocks, with driver's subblocks and with IFS subblocks. At offset 08h in several other MCB descriptors there are signatures with the following meaning:

SC – associated block contains DOS's executable code;
SD – associated block contains DOS's data;
SM – associated block is the last in UMB area;
UMB – associated block is the first in UMB area.

Bytes 08h – 0Fh in MCB descriptors, associated with other memory blocks, are not used and may contain "garbage".

Note 5: program files with *.COM suffix have no headers, which specify the required memory space for other types of executable files. This is why for programs with *.COM suffix DOS allocates the whole free memory space, following previously allocated memory areas. When DOS has no more free memory space, then computer may get hanged at each next request for memory space, which may come from unexpectedly activated resident program of from a handler, invoked by external interrupt. In order to avoid a threat of hanging the programs with *.COM suffix must call for INT 21\AH=4Ah function (8.02-52), forming a separate MCB descriptor for unused part of memory and thus announcing it free. Examples of such calls are shown in the first 6 lines of assembler texts in articles 9.06, 9.10-01 and 9.10-02.

A.13 Hard disk data structures

A.13-1 BIOS tables of physical HDD's parameters

BIOS systems in obsolete computers, produced before 1996, addressed HDDs with parameters CHS (Cylinder-Head-Sector). HDD's storage space, available for CHS addressing, is limited to 528 Mb. BIOS system stored CHS parameters of first and second physical HDDs in data tables; addresses of these tables were written in memory cells 0000:0104h and 0000:0118h correspondingly. Both these cells are inside interrupt table (A.12-1) and sometimes are referred to as INT 41 and INT 46. If computer was equipped

with more than two HDDs, then the only way to obtain CHS parameters of the rest HDDs was a call for INT 13\AH=08h function (8.01-49).

In 1995 capacities of HDDs have reached 1 Gb. The 528 Mb limit, inherent to CHS addressing, had to be overcame. For newer BIOS systems LBA addressing (note 4 to A.13-6) and extended INT 13 functions (8.01-55 – 8.01-60) have been developed. But compatibility with former programs had to be preserved. Therefore in newer BIOS systems the INT 13\AH=08h function (8.01-49) has been "taught" to yield not the real, but transformed CHS parameters. When programs call for "old" INT 13 functions (8.01-46 – 8.01-54) and specify transformed CHS parameters, then BIOS system automatically performs reverse transformation so that HDD's storage space, available for CHS addressing, is expanded to 8.4 Gb (some details – in note 2 to A.13-6).

Data blocks, addressed by pointers stored in cells 0000:0104h and 0000:0118h, may have different data structures: it depends on whether there are real or transformed CHS parameters. Both kinds of data structures are shown in the table below. Real data offsets are shown in the first column (Std). Offsets in the second column (Trs) correspond to transformed CHS data, conforming to Phoenix-1995 specification for HDDs with more than 1024 cylinders. Though described data blocks are supported by modern BIOS systems, nevertheless for programs being developed now usage of parameters from these data blocks is not recommended.

Std	Trs	Size	Description
00h	00h	2	Number of HDD's cylinders (note 1)
02h	02h	1	Number of HDD's heads (note 1)
	03h	1	Signature A0h – distinctive feature of blocks, conforming to Phoenix-1995 specification.
	04h	1	Number of sectors per track (note 2)
05h	05h	2	Cylinder number, where write precompensation should start (note 3)
08h	08h	1	Flags: bit 2 – no recalibration; bit 3 – more than 8 heads; bit 5 – defect map is present (note 4); bit 6 – disable ECC reading retries; bit 7 – disable access retries.
	09h	2	Number of tracks (up to 65536, note 2)
	0Bh	1	Number of heads (up to 16, note 2)
0Ch	0Ch	2	Cylinder number of landing zone (note 3)
0Eh	0Eh	1	Number of sectors per track (note 1)
	0Fh	1	Data checksum

Note 1: CHS parameters in these positions are for "old" functions of INT 13 handler (8.01-46 – 8.01-54). Nevertheless there are some BIOS versions, which specify in

these positions a number of cylinders exceeding 1024, inadmissible for "old" functions.

Note 2: these positions are filled with real HDD's parameters for programs, appealing not to INT 13 handler, but directly to HDD controller's port. These positions are not filled in those data blocks, which supply real parameters for "old" functions of INT 13 handler.

Note 3: modern disk drives perform landing and write precompensation independently. Attempts to affect these operations are ignored.

Note 4: number of the track, where surface defect map is written, is usually a unity greater than number of cylinders, specified at offset 00h in BIOS tables of physical HDD's parameters.

A.13-2 Extended table of HDD's parameters

Extended table of HDD's parameters is written into a prepared buffer by INT 13\AH=48h function (8.01-60).

Offset	Size	Description
00h	2	On call: length of prepared buffer (8.01-60) On return: actual length of returned data
02h	2	Flags: bit 0: DMA boundary errors handled transparently bit 1: CHS data (A.13-1) are valid bit 2: this is a removable drive, bits 4 – 6 are valid bit 3: writing with verification is supported bit 4: drive provides change-line support bit 5: drive provides locking and unlocking bit 6: media isn't present, default CHS data reported
04h	4	Number of physical cylinders on the drive (note 1)
08h	4	Number of physical heads on the drive (note 1)
0Ch	4	Number of physical sectors per track (counted from 1)
10h	8	Total number of sectors (number of the last sector plus 1)
18h	2	Bytes per sector
1Ah	4	Pointer to auxiliary DPTE table (note 2)
1Eh	2	= BEDDh: signature confirming path data presence
20h	1	= 2Ch: size of path data, including the signature
24h	4	Bus type (ISA or PCI) appended with a space (20h)
28h	8	Interface type (note 3)
30h	8	Interface path field (note 4)
38h	16	Device path field (note 5)
49h	1	Complement checksum for bytes at offsets 1Eh – 48h

Note 1: valid numbers of tracks, cylinders and heads are counted from zero, therefore the last valid number is a unity less than the value specified in this field. A value in this field is a real value. Therefore this value shouldn't be specified in calls for "old" INT 13 functions: these functions require transformed parameters, returned by INT 13\AH=08h function (8.01-49).

Note 2: BIOS INT 13 extensions below version 2.x don't return DPTE table, and fill DPTE pointer field with FFFFh:FFFFh value. Structure of DPTE table is shown in appendix A.13-3. DPTE table is presented in a temporary buffer; its contents are not preserved after next calls for BIOS functions.

Note 3: interface type field may be filled with the following words: 1394, ATA, ATAPI, SCSI, USB. Words are appended with spaces up to nominal field length 8 bytes.

Note 4: for ISA bus the path field contains a 2-byte base I/O port address, bytes at offsets 32h – 37h are zeros. For PCI bus a byte at offset 30h presents bus number, byte at offset 31h – slot number, byte at offset 32h – function number, byte at offset 33h – controller number, bytes at offsets 34h – 37h are zeros.

Note 5: for 1394 (Firewire) interface a 8-byte extended unique identifier (EUI-64) is specified at offset 38h. For ATA (IDE) interface the 00h value at offset 38h signifies master device, the 01h value at offset 38h signifies slave device. Byte at offset 38h has the same meaning for ATAPI interface, but byte at offset 39h represents logical unit number (LUN). For SCSI interface a 2-byte device identifier (SCSI ID) is specified starting at offset 38h, and a 8-byte logical unit number (LUN) is specified starting at offset 3Ah. For USB interface a 8-byte serial number is written starting at offset 38h. Not mentioned bytes at offsets 3Ah – 48h are reserved and must be cleared.

A.13-3 Auxiliary device parameters table DPTE

BIOS INT 13 version 2.0 and higher supplement extended HDD's parameters table (A.13-2) with auxiliary device parameters table DPTE. A pointer to DPTE table is returned by INT 13\AH=48h function (8.01-60) at offset 1Ah inside extended HDD's parameters table (A.13-2). DPTE address points at a temporary buffer, which doesn't preserve its contents after following BIOS functions calls. Data in DPTE table are consigned for those programs, which intend to appeal directly to ports of HDD controller. DPTE data structure is shown in the table below.

Offset	Size	Description
00h	2	Physical I/O port base address for the device
02h	2	Disk drive control registers port address
04h	1	Flags: bits 0 – 3 are cleared, bits 5 and 7 are set bit 4: cleared if master device, set if slave bit 6: set if LBA addressing is enabled

Continuation of table A.13-3

06h	1	Bits 0 – 3: IRQ number, bits 4 – 7 are cleared
07h	1	Blocks (sectors) count for multi-block transfers
08h	1	Bits 0 – 3: DMA channel number, bits 4 – 7: DMA type according to ATA-2 specification.
09h	1	Bits 0 – 3: PIO type, if in a word at offset 0Ah bit 0 is set.
0Ah	2	Flags: bit 0: fast PIO enabled, byte 09h is valid bit 1: fast DMA access enabled bit 2: multi-sector transfers enabled bit 3: CHS parameters translation enabled bit 4: LBA translation enabled bit 5: drive uses removable media bit 6: ATAPI interface device (probably CD-ROM) bit 7: 32-bit transfer mode enabled bit 8: ATAPI signal readiness for packet transfers bits 9 – 10: CHS parameters translation type: = 00 – bit-shifting translation = 01 – LBA-assisted translation = 10, 11 – proprietary translation bit 11: ultra DMA access enabled.
0Eh	1	INT 13 extension version number
0Fh	1	Complement checksum of bytes 00h – 0Eh

A.13-4 Disk address packet

This form of data packet is used by extended reading function INT 13\AH=42h (8.01-56) and by extended writing function INT 13\AH=43h (8.01-57). Before applying these functions a check for their BIOS support should be made with a call for INT 13\AH=41h (8.01-55).

Offset	Size	Description
00h	1	Size of address packet (note 1)
02h	1	Number (up to 7Fh) of data blocks to transfer (note 2)
04h	4	Pointer to data transfer buffer (note 3)
08h	8	LBA absolute number of the first data block (note 4)
10h	8	64-bit pointer to data transfer buffer (note 3)
18h	8	Number of data blocks to transfer (note 2)

Note 1: if extended disk address packet is supported, then size is 20h, otherwise size is 10h. Set state of bit 3 in CX register, returned by INT 13\AH=41h function (8.01-55), signifies that extended disk address packet is supported.

Note 2: if extended disk address packet is supported, and if the FFh value is written into a byte at offset 02h, then number of blocks to transfer will be read from a quad

word at offset 18h. On return this number at offset 18h is replaced with number of blocks, which actually have been transferred.

Note 3: if extended disk address packet is supported, and if FFFF:FFFFh value is written into a double word at offset 04h, then pointer to data buffer will be read from a quad word at offset 10h.

Note 4: for disk drives, not supporting LBA addressing (note 4 to A.13-6), absolute number of starting block is calculated according to formula

$$(C \bullet N + H) \bullet T + S - 1$$

where: C – number of the selected cylinder, N – number of heads (by 1 greater than maximum head number), H – number of the selected head, T – number of sectors per track, S – number of the selected sector on a track.

A.13-5 HDD's partition descriptors

In sector 01h of bootable physical HDD's 00-th head 00-th cylinder there is master boot record (MBR), containing up to 446 bytes of executable code, 4-byte identifier (at offsets 1B8h – 1BBh) and partition table. Identifier is written by operating systems Windows-NT/2000/XP only and may be absent. In order to view MBR sector it should be copied into a file, as it is explained in article 9.02-02. Non-textual file can be opened by a viewer of Volcov Commander file manager (6.25) or else by debugger DEBUG.EXE (6.05). Fig.12 shows fragments of MBR sector, copied from a real physical disk.

Fig. 12

```
C:\DOS\SRV\BIOS>debug mbr.dat
-d 100 L20
195B:0100  EB 02 7C 01 FA 33 C0 8E-D0 8E C0 8E D8 BC 00 7C   ..|..3.........|
195B:0110  8B F4 FB BF 00 06 B9 00-01 F3 A5 BB 20 06 FF E3   ............ ...
-d 260 L20
195B:0260  FF 01 B9 00 20 E2 FE 59-C3 0D 0A 45 72 72 6F 72   .... ..Y...Error
195B:0270  20 4C 6F 61 64 69 6E 67-20 4F 53 00 AA 55 01 00    Loading OS..U..
-d 2BE L42
195B:02B0                                                  00 00                ..
195B:02C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 80 01   ................
195B:02D0  01 02 06 3F 3F FF BF 1F-00 00 41 A0 0F 00 00 00   ...??.....A.....
195B:02E0  41 00 05 3F BF D2 00 C0-0F 00 40 BB 1C 00 00 00   A..?......@.....
195B:02F0  C1 B8 82 3F FF C7 00 92-3A 00 00 FC 00 00 55 AA   ...?....:.....U.
-
```

The first fragment presents starting part of executable code (proprietary OnTrack's MBR version). The second fragment presents final part of that executable code with a prepared error message. The third fragment presents partition table, defining division of HDD's writable surface into partitions in a particular computer.

Partition table consists of 4 partition descriptors, each 16 bytes long. If offsets are counted from start of MBR sector, then partition descriptor's offsets are 1BEh, 1CEh, 1DEh, 1EEh correspondingly. However, MBR sector's copy, shown in fig.12, is loaded from offset 100h and on; therefore in fig.12 partition descriptor's offsets are 2BEh, 2CEh,

2DEh, 2EEh. The last word in MBR sector is a signature AA55h, marking the end of bootable disk's MBR.

Four partition descriptors enable to create up to four primary partitions in any physical HDD drive. If there is less than 4 partitions, the rest descriptors are filled with zeros. In fig.12 the first descriptor in partition table is filled with zeros. Hence, this particular HDD is divided into three valid primary partitions.

On a bootable HDD one primary partition must be marked as active (potentially bootable) with a 80h mark in the first descriptor's byte. In fig.12 the 80h mark is present at offset 02CEh, which is the first byte of the second partition descriptor. Hence, active partition in this particular HDD is the second partition.

Missions of the mentioned and of other bytes in partition descriptors are shown in the table below. Offsets in the first column of the table are counted from the start of each partition descriptor.

Offset	Size	Description
00h	1	Status indicator (80h – active partition)
01h	1	Partition's start HDD's head
02h	1	Partition's first sector (note 1)
03h	1	Partition's first track (note 1)
04h	1	Partition's file system identifier (A.13-6)
05h	1	Partition's final HDD's head
06h	1	Partition's last sector (note 2)
07h	1	Partition's last track (note 2)
08h	4	Number of sectors preceding the partition
0Ch	4	Length of the partition (in sectors)

Note 1: in a byte at offset 02h bits 5 – 0 express number of partition's first sector on a track, but bits 6 and 7 represent most significant bits of partition's 10-bit first track number. 8 least significant bits of partition's 10-bit first track number are stored in a byte at offset 03h.

Note 2: in a byte at offset 06h bits 5 – 0 express number of partition's last sector on a track, but bits 6 and 7 represent most significant bits of partition's 10-bit last track number. 8 least significant bits of partition's 10-bit last track number are stored in a byte at offset 07h.

Note 3: descriptors to partitions with LBA addressing (note 4 to A.13-6) may contain invalid CHS parameters (numbers of tracks, heads and sectors). Nevertheless data in bytes at offsets 08h – 0Fh must be valid.

A.13-6 Selected file system identifiers

In each partition descriptor (A.13-5) a byte at offset 04h is file system identifier. Operating system reads file system identifier and "decides" whether it can ensure access to this partition. If file system identifier is not "known" to operating system, then access to this partition will not be attempted. Most probably such partition even wouldn't be shown to user. Some file system identifiers denote hidden partitions, which can be accessed for system purposes, but stay hidden for the user. The table below presents interpretation of selected file system identifiers.

ID	Description
00h	Free disk's space
01h	File system FAT-12 for volumes 16 Mb and less
04h	Obsolete FAT-16 up to 32 Mb without cluster structure
05h	Extended partition with CHS addressing (notes 1 and 2)
06h	FAT-16 up to 2 Gb with CHS addressing (note 2)
07h	NTFS file system (note 3)
0Bh	FAT-32 with CHS addressing (note 2)
0Ch	FAT-32 with LBA addressing (note 4)
0Eh	FAT-16 up to 2 Gb with LBA addressing (note 4)
0Fh	extended partition with LBA addressing (notes 1 and 4)
11h	Hidden FAT-12 partition (for OS/2 boot manager)
14h	Hidden FAT-16 partition (for OS/2 boot manager)
1Bh	Hidden FAT-32 with CHS addressing (note 2)
1Ch	Hidden FAT-32 with LBA addressing (note 4)
3Ch	PowerQuest's Partition Magic recovery partition
42h	Dynamic partition of Windows Vista OS
43h	PTS DOS boot manager's (BootWizard's) partition
4Dh-4Fh	QNX OS partitions
54h	OnTrack Disk Manager's DDO (Dynamic Drive Overlay)
64h-65h	Novell Netware OS partitions
82h	Linux OS swap partition
83h	Ext2fs file system of Linux OS
84h	Partition for power supply state recovery
85h	Linux OS extended partition (note 1)
A0h	Partition for portable PC's state recovery
A5h	FreeBSD OS partition
A6h	OpenBSD OS partition
A8h	UFS file system of MacOS
A9h	Net BSD OS partition
ABh	Bootable partition of MacOS

Continuation of table A.13-6

BEh	Bootable partition of Solaris OS
D8h, DBh	CP/M OS partitions
EBh	BFS1 file system of BeOS
EEh	GPT partition of 64-bit Windows OS versions (note 5)
FDh	RAID partition of Linux OS

Note 1: extended partition is a formal specification of disk's space for placement of several non-primary partitions (logical disks). Descriptors of non-primary partitions are written not in MBR, but in separate dedicated sectors, traced via a chain of references. MS-DOS doesn't allow this chain to be closed in a loop, otherwise MS-DOS hangs in infinite cycle of finding the end of this loop.

Note 2: as far as parameters CHS (Cylinder-Head-Sector) occupy 3 bytes in partition descriptors (A.13-5), hence CHS parameters enable to address not more than 2^24 sectors 512 bytes each, equivalent to 2^23 kilobytes, or else 8 Gb. Therefore partitions with CHS addressing can't be arranged beyond first 8 Gb, counted from the start of disk's space (LBA addressing should be applied further). In table A.13-6 those only identifiers are marked with CHS, which are used as distinctive features of CHS addressing.

Note 3: the 07h file system identifier is interpreted by Microsoft as belonging to installable file systems (IFS), that is file systems with transformed presentation to the user. But in fact, besides NTFS, only one rarely used IBM's HPFS file system is marked with 07h identifier.

Note 4: Linear Block Addressing (LBA) is based on sectors count from start of disk's space according to data in bytes 08h – 0Fh in partition descriptors (A.13-5). LBA enables to overcome the 8 Gb boundary, inherent to CHS addressing. LBA requires support for extended functions of INT 13 handler (8.01-55) from both disk drive and BIOS system. All modern computers provide such support.

Note 5: data about partitions GPT (= GUID Partition Table) constitute extended MBR, occupying not a single sector, but a considerable part of disk's first track. Only 64-bit versions of operating systems Windows server 2003, Windows-XP and Windows Vista provide support for GPT partitions.

Note 6: as far as it is known, partition identifiers 21, 23, 26, 31, 33, 34, 36, 71, 73, 74, 76, 86, A1, A3, A4, A6, B1, B3, B4, B6, E5, E6, F3, F6 are reserved and are not used yet.

A.13-7 Disk's free space table

This table with data about logical disk's free space is returned by INT 21\AX=7303h function (8.02-80), which may be applied to logical disks, formatted with FAT-12, FAT-16 and FAT-32 file systems.

Offset	Size	Description
00h	2	Size of this table (in bytes)
02h	2	Must be = 0000h on call
08h	4	Number of bytes per sector
0Ch	4	Number of free clusters
10h	4	Total number of clusters in logical disk
14h	4	Number of free physical sectors in logical disk
18h	4	Total number of physical sectors in logical disk
1Ch	4	Number of available allocation units
20h	4	Total number of allocation units in logical disk

A.14 Ports

A.14-1 Selected port addresses

Ports represent computer's hardware and therefore should be addressed either via BIOS functions, which are adapted to a particular computer's motherboard, or via device drivers for expansion boards. Direct access to ports can't be recommended for application programs, though there are some exceptions. But knowledge of port addresses is beneficial, at least in order to avoid address conflicts with expansion boards.

The table below shows relatively steady features of general port addresses allocation in AT-compatible computers. Of course, port addresses allocation in your particular computer may somewhat differ from the one shown below.

Address ranges	Target devices
0000h – 001Fh	1-st direct memory access controller (DMA1)
0020h – 0021h	1-st interrupt controller (IRQ 1 – IRQ 7, 8.01-09)
0022h – 0023h	Dynamic RAM controller
0060h – 0064h	Keyboard controller (A.11-3)
0070h	CMOS RAM requests reception port (note 1)
0071h	CMOS RAM data I/O port (note 1)
0080h	Manufacturing diagnostics port
00A0h – 00A1h	2-nd interrupt controller (IRQ 8 – IRQ 15, 8.03-75)
00B2h – 00B3h	Advanced power management ports
00C0h – 00DFh	2-nd direct memory access controller (DMA2)
00F0h – 00FFh	Arithmetical coprocessor
0168h – 016Fh	IFS devices or expansion boards
0170h – 0177h	2-nd IDE HDD controller (default IRQ 15)

Continuation of table A.14-1

01E8h – 01Efh	PS/2 mouse or other devices (IRQ 12)
01F0h – 01F7h	1-st IDE HDD controller (default IRQ 14)
01F8h	A20 line gate control
0200h – 020Fh	Game port (joystick)
0279h	Plug-and-play configuration register port
02E8h – 02EFh	Serial port COM4
02F8h – 02FFh	Serial port COM2 (default IRQ 3)
0300h – 031Fh	NE2000-compatible Ethernet adapters
0330h – 0331h	Musical instrument's MIDI interface
0378h – 037Ah	Parallel port LPT1 (default IRQ 7)
03C0h – 03CFh	Ports of EGA-compatible video adapters (note 2)
03C4h	EGA sequencer's selector port (note 3)
03C5h	EGA sequencer's data port (note 3)
03CEh	Graphic register's selector port (note 4)
03CFh	Graphic register's data port (note 4)
03DAh	CGA/EGA/VGA video adapter's status port (note 5)
03E0h – 03E7h	PCMCIA i82365 controller's ports
03E8h – 03EFh	Serial port COM3
03F0h – 03F7h	Floppy disk controller (default IRQ 6)
03F8h – 03FFh	Serial port COM1 (default IRQ 4)
0A79h	Plug-and-play system data port
0CF8h – 0CFFh	PCI bus configuration ports

Note 1: some data in CMOS RAM are accessible via BIOS Setup program (1.01). Besides that, some hardware data and memory data are read by INT 11 (8.01-42) and by INT 12 (8.01-43) handlers. Direct appeals to CMOS RAM may be necessary for masking NMI (note 1 to 8.01-03), for obtaining data about floppy drives (offset 10h in A.08-03) and for defining CPU's actions after reset, dependent on a byte at offset 0Fh in CMOS RAM (note 4 to A.12-1). For access to data in CMOS RAM the requested byte's offset (up to 7Fh) should be sent by OUT command (7.03-66) to port 70h; then via port 71h the requested byte's value can be read by IN command (7.03-26) or altered with OUT command.

Note 2: as far as drawing graphics with INT 10\AH=0Ch function (8.01-19) is slow, operating systems appeal directly to video memory and to ports of EGA-compatible video adapters. Therefore EGA port addresses are preserved as a standard, though EGA adapter itself has come out of use long ago.

Note 3: port 03C5h transfers a byte, sent by OUT command (7.03-66), to that internal register in video adapter, which is defined beforehand via port 03C4h. If a byte AL = 02h is sent in advance to port 03C4h, then the next byte, sent to port 03C5h, will be accepted as color mask (its normal value is 0Fh).

Note 4: port 03CFh transfers a byte, sent by OUT command (7.03-66), to that internal register in video adapter, which is defined beforehand via port 03CEh. If a byte

AL = 08h is sent in advance to port 03CEh, then the next byte, sent to port 03CFh, will be accepted as a bit-mask for 8 consecutive pixels. If a byte AL = 05h is sent in advance to port 03CEh, then the next byte, sent to port 03CFh, will specify mode (00h – 02h) of writing into video memory (note 3 to 8.01-39).

Note 5: port 03DAh is intended for data reading only. Bit 3 in a byte, read from port 03DAh, is kept set during field retrace intervals and is kept cleared outside these intervals. Therefore field retrace intervals can be registered by IN command (7.03-26) in order to avoid image disruptions, which are noticeable, if changes of the displayed image occur outside retrace intervals.

A.14-2 Status of serial port

This table presents interpretation of bitfields in status byte, returned in AH register by INT 14\AH=00h – INT 14\AH=03h functions (8.01-65 – 8.01-68).

Bit	Description
0	Received data are ready
1	Overrun error
2	Parity error
3	Framing error
4	Break detected
5	Transmit holding register is empty
6	Transmit shift register is empty
7	Timeout. no reply

A.14-3 Status of printer, connected to parallel port

This table presents interpretation of bitfields in status byte, returned in AH register by INT 17\AH=00h – INT 17\AH=02h functions (8.01-86 – 8.01-88).

Bit	Description
0	Timeout, no reply
1	EPP BIOS only: requested port is not supported
2	Unused
3	I/O error
4	Port is busy
5	Printer is out of paper
6	Acknowledge
7	Printer is ready (not busy)

Note 1: returned set state of CF flag with status byte AH = 03h means that EPP BIOS is present, but it doesn't support the requested port.
Note 2: status byte AH = 00h means that EPP data are returned in registers (A.14-4).

A.14-4 Selected functions of EPP BIOS

EPP is a BIOS supplement, enabling enhanced modes of data transfer through LPT ports according to IEEE 1284 specification. Presence of EPP BIOS in your computer should be proved by INT 17\AX=0200h function (8.01-88), which also reports LPT port base address, EPP BIOS version, and address of its entrance point. The latter is used as a target address for CALL FAR command (7.03-08); being called in this way, EPP BIOS performs the operation, specified by a value in AH register at that moment. For the latest revision 7 of EPP BIOS the choice of a particular LPT port is defined by its base address in DX register; earlier EPP BIOS versions define LPT port by its number 00h – 03h in DL register. Besides this, some EPP BIOS functions need other data, shown in the second column of the table below or in notes, marked by note number in fifth column. Unless specified otherwise, almost all EPP BIOS functions return status byte (A.14-7) in AH register, mark failure with set state of CF flag and don't preserve contents of BX register.

AH	On call	EPP BIOS function	On return	Comments
00h		Report configuration	AL = IRQ	A.14-5
01h	A.14-6	Set transfer mode	AX altered	
02h		Report transfer mode	AH altered	A.14-6
03h	AL=00h	Enable LPT interrupts		
03h	AL=01h	Disable LPT interrupts		
04h		Reset EPP	AL altered	
05h	AL=address	Address-write I/O cycle	AL altered	
06h		Address-read I/O cycle	AL=address	
07h	AL=byte	Send a byte		
08h		Send a block of data		note 1
09h		Receive a byte	AL=byte	
0Ah		Receive a block of data		note 2
0Bh	AL=address	Addressed byte reading	AL=byte	
0Ch	AL=address	Addressed byte sending		note 3
0Dh	AL=address	Addressed block reading		note 2
0Eh	AL=address	Addressed block sending		note 1
0Fh	AL=port	Lock LPT port		note 4
10h	AL=port	Unlock LPT port		note 4
11h	CH=00h	Disable device interrupts		note 5
11h	CH=01h	Enable device interrupts		note 5
12h	AL=00h	Real-time device present?	AL=01h if yes	

Continuation of table A.14-4

12h	AL=01h	Add real-time device		
12h	AL=02h	Remove real-time device		
40h		Query multiplexer		notes 6, 7
41h	AL=port	Query multiplexer port		notes 6, 8
50h	AL=port	Query daisy chain		notes 6, 9
51h	AL=port	Reassign chain numbers		note 6

Note 1: on call a pointer to data block must be specified in DS:SI registers, length of data block (in bytes) – in CX register. Earlier versions of EPP BIOS, preceding the 7-th revision, accept a pointer to data block from ES:DI registers. Function returns in CX register a number of bytes not sent yet.

Note 2: on call a pointer to buffer for data must be specified in ES:DI registers, length of this buffer – in CX register. After successful termination function returns buffer filled, and in CX register – a number of bytes left not filled in the buffer.

Note 3: on call the data byte to be sent must be specified in CL register. Earlier versions of EPP BIOS, preceding the 7-th revision, accept data byte from DH register.

Note 4: if external devices are connected via multiplexer, then multiplexer port number (01h – 08h) should be specified in bits 3 – 0 of a byte in AL register. If external devices are connected as a daisy chain, then daisy chain port number (10h – 80h) should be specified in bits 7 – 4 of a byte in AL register.

Note 5: multiplexer port number (01h – 08h) should be specified in AL register. If multiplexer is not used, then AL register must be cleared. For interrupt enable function (CH = 01h) a pointer to interrupt event handler should be specified in ES:DI registers.

Note 6: this function is not implemented in earlier versions, preceding EPP BIOS revision 7. This function shouldn't be applied, if external devices are not connected to a certain LPT port via multiplexer or as a daisy chain. The requested LPT port will be identified not by its number, but by its I/O base address, which should be specified in DX register.

Note 7: the number of currently active multiplexer port is returned by this function in AL register, and a flags byte is returned in CH register. Set state of bit 0 in flags byte signifies locked state of multiplexer port, set state of bit 1 in flags byte signifies that there is an interrupt pending from this multiplexer port.

Note 8: this function returns in CH register a byte of flags. Set state of bits in this flags byte have the following meaning:

- bit 0 – this port is selected (active)
- bit 1 – port is locked
- bit 2 – interrupts from this port are enabled
- bit 3 – interrupt is pending from this port.

Note 9: this function returns in BH register the EPP BIOS version, in BL register – number of currently active daisy chain device, in CH register – a flags byte (as described in note 7), in CL register – number of devices constituting the daisy

chain (or 00h, if there is no daisy chain), in ES:DI registers – a pointer to driver's vendor identification string.

A.14-5 EPP BIOS configuration byte

The "Report configuration" function, defined on call by 00h value in AH register (A.14-4), returns in AL register a LPT port's IRQ line number, in BH register – the EPP BIOS version, in ES:DI registers – a pointer to driver's information, in CX register – LPT port's I/O base address (for versions 1.0 – 3.0 of EPP BIOS only), and in BL register returns EPP BIOS configuration byte. Interpretation of configuration byte's bitfields is given in the table below.

Bit	Description
0	Multiplexer is present
1	Bi-directional data transfer is supported
2	Daisy chain is present
3	ECP specification is supported
4	EPP software emulation is supported
5	EPP BIOS functions are supported
6	"Fast Centronics" data transfer is supported
7	Standard EPP I/O map

A.14-6 Code of EPP BIOS data transfer mode

The "Report transfer mode" function, defined on call by 02h value in AH register (A.14-4), returns code of current transfer mode in AL register. The "Set transfer mode" function, defined on call by 01h value in AH register (A.14-4), enables to change LPT data transfer mode according to mode's code, specified in AL register. Interpretation of bitfields in this mode's code is given in the table below.

Bit	Description
0	Compatibility mode enabled
1	Bi-directional data transfer enabled
2	Data transfer according to EPP specification
3	Data transfer according to ECP specification (note 1)
4	EPP software emulation enabled (note 1)
5	"Fast Centronics" data transfer enabled (note 1)
6	= 0b (reserved)
7	EPP port interrupts enabled (note 2)

Note 1: modes, defined by set state of bits 3 – 5 in mode's code, can't be implemented by earlier versions of EPP BIOS, preceding its latest 7-th revision.
Note 2: as far as device interrupts are enabled by AH = 11h function (A.11-4), set state of bit 7 is not accepted by "Set transfer mode" function (AH = 01h), but nevertheless may be reported by "Report transfer mode" EPP BIOS function (AH = 02h).

A.14-7 Status code of EPP BIOS functions

Almost all EPP BIOS functions, except 01h and 02h (A.14-4), return in AH register a status byte. Interpretation of EPP BIOS status byte codes is given in the following table.

Code	Description
00h	Successful termination
02h	Requested command or feature is not supported
03h	Requested parallel port is not supported
05h	Request is not supported in current mode
06h	Invalid subfunction requested
07h	Request is already done or set
20h	LPT multiplexer isn't present (AMI BIOS versions only)
40h	LPT multiplexer isn't present (other EPP BIOS versions)
41h	Currently multiplexer is locked
80h	I/O timeout, no response
FFh	Requested function either is invalid or isn't supported

A.15 CD/DVD service tables

A.15-1 Bootable CD/DVD specification packet

BIOS functions INT 13\AH=4A00h and INT 13\AX=4C00h (8.01-61) emulate a logical disk on basis of its copy, read from an optical CD/DVD disc. Emulation control parameters must be prepared in a form of bootable CD/DVD specification packet. Internal data structure of this packet is shown in the following table.

Offset	Size	Description
00h	1	Size of this packet in bytes (=13h normally)
01h	1	Disk copy type (same as at offset 21h in A.15-3)
02h	1	Drive number to emulate (note 1 to 8.01-44)
03h	1	Drive controller number to emulate

Continuation of table A.15-1

04h	4	CD Logical Block Address where disk copy begins (the same as at offset 28h in table A.15-3)
08h	2	Bit 0: emulate a slave IDE drive Bits 7 – 0: LUN and SCSI ID number (for SCSI) Bits 15 – 8: bus number (for SCSI)
0Ah	2	Segment address of prepared 3 kb read buffer (or 0000h value if no caching)
0Ch	2	Segment address to load boot sector of disk's copy (the same as at offset 22h in table A.15-3)
0Eh	2	Number of 512-byte virtual sectors in disk's copy (the same as at offset 26h in table A.15-3)
10h	1	Least significant 8 bits in number of disk's copy cylinders (as is returned by INT 13\AH=08h in CH)
11h	1	Bits 5 – 0: number of last sector on a track Bits 7 – 6: most significant bits of copy cylinders number (as is returned by INT 13\AH=08h in CL)
12h	1	Number of heads in emulated disk drive

A.15-2 Format of command packet

This command packet specifies parameters for INT 13\AH=4Dh function (8.01-63), which reads sectors from optical CD/DVD discs. During boot procedure this command packet specifies reading of boot catalog from CD/DVD disc.

Offset	Size	Description
00h	1	Size of packet in bytes (= 08h normally)
01h	1	Number of sectors to be read
02h	4	Pointer to buffer for read data
06h	2	Number of the first sector to be read

A.15-3 Boot catalog of optical CD/DVD discs

Optical discs potentially are able to implement several computer booting scenarios. Initial data, defining each booting scenario, must be present on bootable optical disc(s) in form of a hidden directory, also known as boot catalog. This directory can be read by INT 13\AH=4Dh function (8.01-63). As in ordinary directory, records in boot catalog have standard length 20h bytes. Minimal contents of boot catalog are composed of two required records: the first record is known as validation entry, the second record is a descriptor of default bootable disk's copy. The table below presents structures of both

mentioned required records. Offsets 00h – 1Fh correspond to validation entry record, offsets 20h – 3Fh – to descriptor of default bootable disk's copy.

Offset	Size	Description
00h	1	= 01h: signature of validation entry start
01h	1	Platform type: = 00h – AT-compatible = 01h – Power PC = 02h – Apple Macintosh
04h	24	CD/DVD drive manufacturer (ASCII string)
1Ch	2	Complement checksum of bytes 00h – 1Fh
1Eh	2	= AA55h: validation entry termination signature
20h	1	= 88h: signature of bootable disk's descriptor (note 2)
21h	1	Bits 3-0: = 0000b – invalid descriptor = 0001b – copy of 1.2 Mb diskette = 0010b – copy of 1.44 Mb diskette = 0011b – copy of 2.88 Mb diskette = 0100b – copy of a hard disk Bit 6: – copy of a disk with ATAPI interface Bit 7: – copy of a disk with SCSI interface
22h	2	Segment address for loading boot sector (if = 0000h, then segment address is 07C0h by default)
24h	1	Disk's copy file system identifier (A.13-6)
26h	2	Number of 512-byte virtual sectors in disk's copy
28h	4	CD/DVD Logical Block Address where disk's copy starts

Note 1: besides two required records, presented in table A.15-3, CD boot catalog may contain other 32-byte records, grouped in several sections. Each section represents a separate booting option and consists of not less than two records: a header record and a bootable disk's copy descriptor for that booting option. Header record begins with signature byte 90h, except header record in the last section, which begins with signature byte 91h. In every header record a word at offset 02h announces number of 32-byte records in this section. In each section descriptor of booting option may be followed by auxiliary records. Data structure in each descriptor of booting option is the same as that of default bootable disk's descriptor, shown in table A.13-3 at offsets 20h – 3Fh.

Note 2: descriptors of non-bootable disks are allowed too, their distinctive feature is starting signature 00h.

A.15-4 Commands, performed by CD/DVD drivers

In order to send a command to a driver, you have to obtain a handle, associated with this driver. The first step is to find a pointer to CD/DVD driver header by means of INT 2F\AX=1501h function (8.03-14), performed by TSR program either MSCDEX.EXE (5.08-03) or SHSUCDX.COM (5.08-04). Second step is to read a 8-byte name (signature) of driver's access channel inside driver's header at offset 0Ah. Driver access channel usually is named after the identifier, which follows the /D: parameter in command line, loading the driver (for example, the /D:MSCD001 identifier for CD/DVD drivers, shown in articles 5.10-01 – 5.10-03). The next third step is to use the name for obtaining a handle with INT 21\AH=3Dh function (8.02-33). The name must be uppercased and appended to 8 bytes with spaces (20h), if it is shorter. Returned handle should be placed in BX register before the desired command will be sent to the driver by INT 21\AX=4403h or by INT 21\AX=4402h function (8.02-41). Besides that, these functions need a request data block to be prepared. A pointer to request data block must be specified in DS:DX registers, and length of request data block – in CX register. Length of request data block for different commands is shown in the second column of the table below. The third column shows codes of CD/DVD driver's commands, which are to be specified at offset 00h in request data block. If command implies return of some data, driver will write these data into cells of the same request data block.

AX	CX	Code	Operation	Comments
4402h	05h	00h	Report driver's header address	note 2
4402h	06h	01h	Report drive's head location	note 3
4402h	09h	04h	Report audio control status	A.15-5
4402h	05h	06h	Report CD/DVD drive status	A.15-6
4402h	04h	07h	Read mode, sector size	note 4
4402h	05h	08h	Get number of sectors	note 2
4402h	02h	09h	Report disc change status	note 5
4402h	07h	0Ah	Get number of tracks	note 6
4402h	08h	0Bh	Get start of track	note 7
4403h	01h	00h	Eject the tray	
4403h	02h	0100h	Unlock the door	
4403h	02h	0101h	Lock the door	
4403h	01h	02h	Reset the drive	note 1
4403h	09h	03h	Audio control	A.15-5
4403h	01h	05h	Pull the track in	

Note 1: after any request to CD/DVD driver sent by INT 21\AX=4402h function and before the driver is activated for any other purpose it must be reset by sending command 02h via INT 21\AX=4403h function.

Note 2: after execution of operations 00h and 08h the requested result is written into request data block starting at offset 01h. This result is either a 4-byte number or a 4-byte address according to the requested operation.

Note 3: after a request for drive's head location a returned byte at offset 01h in request data block represents CD/DVD addressing format:

00h – HSG format

01h – Red Book format (frames/seconds/minutes).

Besides that, a double word at offset 02h in request data block is drive's head location in units according to addressing format.

Note 4: after a request for read mode it is returned in request data block at offset 01h:

00h – reading with error correction (cooked)

01h – reading with ECC code, but without error correction (raw).

Besides that, returned word at offset 02h presents sector's size.

Note 5: after a request for disc change status driver returns status byte at offset 01h in request data block:

00h – change status isn't determined,

01h – disc has not been changed,

FFh – disc has been changed.

Note 6: after a request for number of tracks driver returns number of the first track in a byte at offset 01h, number of the last track – in a byte at offset 02h, and start address of the first track (in Red Book format) is returned as a double word at offset 04h.

Note 7: on call for start of a track the number of requested track must be specified in byte at offset 01h in request data block. In response to this call driver writes into request data block at offset 02h a double word address of requested track's starting point (in Red Book format). Besides that, in a word at offset 06h driver returns flags, where set state of bits means the following:

bit 12 – audio track, written with preemphasis,

bit 13 – digital copying is permitted,

bit 14 – this track contains data (not audio),

bit 15 – this is a 4-channel audio track.

A.15-5 CD/DVD audio control

If in CD/DVD drive status word (A.15-6) bit 8 is set, then this CD/DVD drive doesn't need audio card in order to control audio playback: this CD/DVD drive itself is able to control audio playback.

For sending a request to CD/DVD drive via INT 21\AX=4403h function (A.15-4) a pointer to data block with requested parameters must be specified in DS:DX registers. The table below presents structure of a data block with audio control parameters, which should be prepared for audio control operation 03h in order to alter audio playback. Data block of

the same structure is written into a prepared buffer by INT 21\AX=4402h function in response to a request for audio control status operation (A.15-4). On call for INT 21\AX=4402h function a byte at offset 00h in prepared buffer must be filled yet: it must specify code 04h of audio control status operation.

Offset	Size	Description
00h	1	Function: 03h for AX=4403h or 04h for AX=4402h
01h	1	Input channel (0 – 3) for output channel 0
02h	1	Volume for output channel 0
03h	1	Input channel (0 – 3) for output channel 1
04h	1	Volume for output channel 1
05h	1	Input channel (0 – 3) for output channel 2
06h	1	Volume for output channel 2
07h	1	Input channel (0 – 3) for output channel 3
08h	1	Volume for output channel 3

Note 1: output channels 0 and 1 correspond to left and right; output channels 2 and 3 correspond to rear left and rear right. Each channel may be switched off by sending its volume value 00h.

Note 2: by default each input channel is connected to output channel with the same number, and volume is set to maximum value FFh.

A.15-6 Bitfields in optical disc drive status word

This table shows meaning of bitfields in status word, returned by INT 21\AX=4402h function inside data block at offset 01h in response to CD/DVD drive status request 06h (A.15-4).

Bit	Description
0	Drive's tray is ejected
1	Tray lid is unlocked
2	"Raw" reading mode is supported (note 1)
3	Drive enables writing onto CD/DVD discs
4	CD/DVD drive is able to play audio/video tracks
5	CD/DVD drive supports interleaving (note 2)
7	CD/DVD drive supports prefetch requests (note 3)
8	CD/DVD drive supports audio channel control
9	Red Book addressing is supported (in addition to HSG)
10	CD/DVD drive drive is busy with playing audio
11	There is no disc in CD/DVD drive
12	CD/DVD drive has separate read and write channels

Note 1: "raw" reading mode implies that ECC code together with data is read and sent to output, but error correction is not performed. Normal "cooked" reading mode implies that ECC code is used for error correction, but is not sent to output together with corrected data.

Note 2: interleaving here is related to video files, composed of alternating groups of image frames and audio frames.

Note 3: prefetch requests cause reading into drive's memory buffer, so that later the requested data can be obtained without waiting for access to the requested track.

A.16 Some relevant abbreviations

ACPI – Advanced Configuration and Power Interface specification stipulates presentation of motherboard's parameters to operating system in a form of data tables in dedicated memory areas.

AGP – Accelerated Graphic Port: a slot for inserting video adapters, and also specification of video adapter's interaction with motherboard.

AH – CPU's 8-bit register, representing bits 15 – 8 of 16-bit AX register.

AL – CPU's 8-bit register, representing bits 7 – 0 of 16-bit AX register.

AMIS – Alternate Multiplex Interrupt Specification (A.07-6)

ANSI – American National Standards Institute (USA)

API – Application Program Interface, i.e. OS services for programs.

APM – Advanced Power Management: extension of computer's BIOS system, providing control over power supply (8.01-70 – 8.01-72)

ASCII – American Standard Code for Information Interchange.

ASCIIZ – a string in ASCII code terminated with at least one 00h byte.

ASPI – Advanced SCSI Programming Interface: enhanced set of commands for SCSI interface (5.07-03). Nowadays many ASPI commands are implemented in ATAPI (5.07-01) and USB (5.07-05) controllers.

AT – Advanced Technology: a name of IBM's PC model produced in 1984. Most modern computers inherit some features of AT model.

ATA – AT Attachment: disk storage device's interface, implemented for the first time in IBM's PC AT model.

ATAPI – ATA Packet Interface: packet enhancement of ATA (5.07-01).

ATX – AT extension: enhanced specification of PC blocks construction features, implemented since 1998.

AUX – reserved word, used to address serial port COM1.

AVI – suffix of video files, composed of interleaved audio and video frames.

AX – 16-bit general purpose register, associated with CPU's arithmetic unit. In 32-bit CPUs AX represents a part (bits 15 – 0) of 32-bit register EAX.

b – binary: distinctive mark of binary numbers.

BAT – suffix of batch files, interpreted by COMMAND.COM (6.04). Unlike ordinary command files, batch files are accepted by COMMAND.COM

interpreter from command line without input redirection.
BH – CPU's 8-bit register, representing bits 15 – 8 of 16-bit BX register.
BIOS – Basic Input-Output System, the one supplied with PC's motherboard.
BL – CPU's 8-bit register, representing bits 7 – 0 of 16-bit BX register.
BP – Base Pointer: 16-bit register, used as a base for addressing data arrays. In 32-bit CPUs BP represents bits 15 – 0 of 32-bit register EBP.
BPB – BIOS Parameters Block (A.03-4).
BSD – Berkley Software Distribution, known for a freeware OS.
BX – 16-bit general purpose register, also used as a base for addressing data. In 32-bit CPUs BX represents bits 15 – 0 of 32-bit register EBX.
CD – Compact Disc: a one-sided 650 – 800 Mb optical disc.
CD-ROM – a read-only compact disc or a drive for such discs..
CDS – Current Directory Structure (A.03-3).
CF – (1): CPU's Carry Flag, used to indicate carry and errors.
CF – (2): Compact Flash – a type of removable storage cards.
CGA – Color Graphic Adapter: the first IBM's model of color video adapter.
CH – CPU's 8-bit register, representing bits 15 – 8 of 16-bit register CX.
CHS – Cylinder-Head-Sector: mode of HDDs addressing (note 2 to A.13-6).
CL – CPU's 8-bit register, representing bits 7 – 0 of 16-bit register CX.
CMOS – Complementary Metal-Oxide Semiconductor: BIOS' memory block on CMOS chips, which doesn't lose stored data when PC is switched off.
COM – (1): suffix of executable files, which have no header.
COM – (2): reserved word for access to serial ports.
COM – (3): Common Object Model – a programming technique.
CON – console, i.e. keyboard for input and display for output.
CP – Codepage (more about CP – in articles 1.06 and A.02-2).
CP/M – Control Program for Microcomputers: a prototype of DR-DOS.
CPU – Central Processing Unit – the main processor chip in a PC.
CR – Control Registers: 32-bit registers CR0, CR2, CR3 (A.11-4), introduced in 80386 CPU model. CR4 has been introduced later in Pentium CPU.
CRC – Cyclic Redundancy Check: code for error detection only (no correction)
CRT – Cathode Ray Tube.
CS – Code Segment: 16-bit segment register, defining segment address of the code, executed by CPU.
CSM – Compatibility Support Module supplements UEFI BIOS with features of ordinary BIOS systems, thus enabling to use DOS' software, to launch Windows-XP and many other OSes, which can't start under UEFI.
CWR – Control Word Register in arithmetical coprocessor.
CX – 16-bit general purpose register, often used as a counter. In 32-bit CPUs CX represents a part (bits 15 – 0) of 32-bit ECX register.
DAC – Digital-to-Analog Converter.
dd – two-digit decimal day number in a month.

DDO – Dynamic Drive Overlay: OnTrack's BIOS extension for access to HDDs over 512 Mb in obsolete PCs.
DH – CPU's 8-bit register, representing bits 15 – 8 of 16-bit register DX.
DI – Destination Index: 16-bit register normally used to store target offset. In 32-bit CPUs DI represents a part (bits 15 – 0) of 32-bit register EDI.
DL – CPU's 8-bit register, representing bits 7 – 0 of 16-bit register DX.
DMA – Direct Memory Access.
DOS – Disk-based Operating System.
DPB – Drive Parameters Block (A.03-1).
DPMI – DOS Protected Mode Interface: API functions for programs, designed to be executed in CPU's V86 mode. DPMI is implemented, in particular, by "DOS box" of Windows OS (more about that – in 8.03-21).
DPR – Data Pointer Register in arithmetical coprocessor.
DPTE – Drive Parameter Table Extension (A.13-3).
DR – (1): Digital Research – company developer of CP/M and DR-DOS.
DR – (2): Debug Registers – CPU's registers DR0 – DR7 (A.11-5).
DS – Data Segment: 16-bit segment register, defining segment address of current program's data block.
DTA – Data Transfer Area (8.02-16, A.09-1)
DVD – Digital Versatile Disk – optical disc with 4.7 Gb of data per side.
DX – 16-bit general purpose register. In 32-bit CPUs DX represents a part (bits 15 – 0) of 32-bit register EDX.
EAX – 32-bit general purpose register in 32-bit CPUs. Least significant bits of EAX register (bits 15 – 0) constitute AX register.
EBIOS – BIOS extension providing LBA mode of disk access in obsolete PCs.
EBP – 32-bit base address register in 32-bit CPUs. Least significant bits of EBP register(bits 15 – 0) constitute BP register.
EBX – 32-bit general purpose register in 32-bit CPUs. Least significant bits of EBX register (bits 15 – 0) constitute BX register.
ECC – Error Correcting Code.
ECP – Extended Capabilities Port: data transfer specification for LPT ports.
ECX – 32-bit general purpose register in 32-bit CPUs. Least significant bits of ECX register (bits 15 – 0) constitute CX register.
EDI – 32-bit destination offset register in 32-bit CPUs. Least significant bits of EDI register (bits 15 – 0) constitute DI register.
EDX – 32-bit general purpose register in 32-bit CPUs. Least significant bits of EDX register (bits 15 – 0) constitute DX register.
EFI – Extensible Firmware Interface: Intel's specification of 32-bit BIOSes, originally intended for Itanium 64-bit single-core CPU (2002). Revision of EFI for newer 32-bit multi-core CPUs is known as UEFI (2007).
EGA – Enhanced Graphics Adapter: obsolete IBM's color video adapter. Modern video adapters inherit many important features from EGA.

EHCI – Enhanced Host Controller Interface: controller specification for USB bus versions 2.x (more about that – in 5.07-05).
EMM – Expanded Memory Manager: EMM386.EXE driver (5.04-02).
EMS – Expanded Memory Specification, implemented by EMM (5.04-02).
EOF – End Of File: EOF mark in ASCII code is byte 1Ah.
EPP – Enhanced Parallel Port: BIOS extension of LPT functions (A.14-4)
ES – 16-bit segment register in CPU, defining target segment address.
ESI – 32-bit source offset register in 32-bit CPUs. Least significant bits of ESI (bits 15 – 0) constitute SI register.
ESP – 32-bit stack pointer register in 32-bit CPUs. Least significant bits of ESP (bits 15 – 0) constitute SP register.
EXE – Executable: suffix for executable files having a header.
FASM – Flat ASseMbler: modern freeware assembler for DOS, Windows, Linux and Unix. Can be downloaded from http://www.flatassembler.net/ .
FAT – File Allocation Table.
FCB – File Control Block (A.09-5).
FCBS – command (4.10), reserving memory for FCBs.
FDD – Floppy Disk Drive.
FS – auxiliary 16-bit segment register, introduced since CPU 80386.
GDT – Global Descriptor Table – a table of 8-byte segment descriptors, defining main system segments in protected mode. (A.12-2).
GDTR – CPU's system register. Stores GDT's linear address and size.
GS – auxiliary 16-bit segment register, introduced since CPU 80386.
GUI – Graphical User Interface – alternative to textual command line.
GUID – Globally Unique IDentifier – 32-bytes long universal identifier.
h – hexadecimal: distinctive mark of hexadecimal numbers.
HDD – Hard Disk Drive.
HMA – High Memory Area: memory area 1024 – 1088 kb.
HRS – Hidden, Read-only, System: a set of attributes for system files.
HSG – High Sierra Group specification – prototype of ISO 9660 standard.
IBM – International Business Machines company.
ID – identifier.
IDE – Integrated Drive Electronics: HDD interface, equivalent to ATA.
IDT – Interrupt Descriptor Table: table of interrupts for protected mode. Its segment descriptor is sometimes also denoted as IDT.
IDTR – CPU's system register. Stores IDT's linear address and size.
IEEE – Institute of Electrical and Electronics Engineers.
IFS – Installable File System: file system (5.08-01), accessed by means of installable driver.
IML – Initial Machine Load system.
INT – Interrupt: event or command (7.03-28), invoking interrupt handler.
I/O – Input-Output, i.e. data transfer operations.

IOCTL – Input-Output Control system (8.02-41).
IP – Instruction Pointer: 16-bit register, defining offset of the next command. In 32-bit CPUs IP represents a part (bits 15 – 0) of 32-bit EIP register.
IPR – Instruction Pointer Register in arithmetical coprocessor.
IRQ – Interrupt ReQuest line(s).
ISA – Industrial Standard Architecture: obsolete bus for expansion cards.
ISO – International Standards Organization.
ISP – Interrupt Sharing Protocol (A.07-5).
JFT – Job File Table: table of opened handles (note 3 to A.07-1).
LAN – Local Area Network.
LBA – Linear Block Addressing – HDDs addressing mode (note 4 to A.13-6).
LCD – Liquid Crystal Display.
LFN – Long File Name (A.09-3).
LIM – Lotus-Intel-Microsoft: the founders of EMS specification.
LPT – Line PrinTer: port, also known as parallel port.
LUN – Logical Unit Number: identifier for devices, sharing one bus address (note 1 to A.03-2).
MASM – Macro ASseMbler: Microsoft's low-level code assembler.
MBR – Master Boot Record (A.13-5).
MCB – Memory Control Block: 16 bytes long descriptor (A.12-7).
MDA – Monochrome Display Adapter, used in IBM's obsolete PCs.
mm – two-digit decimal month number in a year.
MO – Magneto-Optical disks or disk drives.
MS – mark for objects, owned or developed by Microsoft.
MSWR – Machine Status Word Register: control register in CPU 80286. In modern CPUs MSWR is a part of control register CR0.
NTFS – New Technology File System for HDDs under Windows NT/2000/XP.
NUL – (1): channel "to nowhere", as alternative to real channels.
NUL – (2): the 00h byte value.
OEM – Original Equipment Manufacturer: direct delivery of components (as antonym of retail sale).
OHCI – Open Host Controller Interface: controller specification for USB bus versions 1.x (5.07-05).
OS – Operating System.
PC – Personal Computer.
PCI – Peripheral Components Interconnect: type of bus for expansion cards.
PCMCIA – PC Memory Card International Association: interface standard, originally designed for memory expansion cards (5.07-02).
PD – Powerful Disk: a 650 Mb rewritable optical discs of CD-RAM type, prototype of DVD-RAM discs.
PIO – Programmed I/O control for devices with ATAPI interface.
PM – Protected Mode of CPU operation.

POST – Power-On Self Test: performed by BIOS when PC is switched on.
PRN – reserved word used to address printer port LPT1.
PSP – Program Segment Prefix (A.07-1).
PS/2 – Personal System/2: IBM's PC model developed in 1987.
PS2 – mouse port and connector type, first introduced in PS/2 PCs.
PTS – PhysTechSoft – russian software vendor, known for its PTS-DOS
RAID – Redundant Array of Inexpensive Disks: distributed storage technique, enhancing transfer speed. Under permanent qualified maintenance some RAID versions also may reduce risk of data loss.
RAM – Random Access Memory – ordinary writable memory, as alternative to sequential access to tape and disk media.
ROM – Read-Only Memory, i.e. non-rewritable storage media.
SCSI – Small Computer System Interface (5.07-03).
SFT – System File Table of associations for active handles (A.01-4)
SFX – SelF eXtracting packed archive or module.
SI – Source Index: 16-bit register, used to store source address offset. In 32-bit CPUs SI represents a part (bits 15 – 0) of 32-bit ESI register.
SIMD – Single Instruction Multiple Data: class of commands, performing the same operation over a group of data items.
SP – Stack Pointer: 16-bit register, defining offset of stack's top. In 32-bit CPUs SP register represents a part (bits 15 – 0) of 32-bit ESP register.
SS – Stack Segment: 16-bit register, defining stack's segment address.
SSE – Streaming SIMD Extensions: extensions of SIMD command set, implemented in modern CPUs.
STDIN – input channel, corresponding to handle 0000h and having default association with keyboard as data source.
STDOUT – output channel, corresponding to handle 0001h and having default association with display as target device.
STDERR – channel to display error messages, associated with handle 0002h.
SVGA – SuperVGA videomodes (A.10-1), suggested by VESA.
SWR – Status Word Register in arithmetical coprocessor (7.04-08, 7.04-64).
TASM – TurboASseMbler: a low-level code assembler from Borland Co.
TLB – Translation Lookaside Buffer: cache buffer in CPU, performing translation of linear addresses into physical addresses.
TSR – Terminate and Stay Resident: resident modules or programs (8.02-23)
TWR – Tags Word Register in arithmetical coprocessor.
UEFI – Unified EFI - revision of EFI, adopted in 2007 for modern 32-bit multi-core CPUs. UEFI stipulates for graphic shell, for network support and for retention of compatibility with OSes by means of CSM module.
UHCI – Universal Host Controller Interface: controller specification for USB bus versions 1.x (5.07-05).

UMB – Upper Memory Blocks: address space pieces, allotted for loading drivers inside 640 – 1024 kb area.
USB – Universal Serial Bus (5.07-05).
V86 – virtual 8086 mode: emulation of obsolete CPU 8086 by modern CPUs, operating in protected mode. V86 mode enables to execute DOS programs at the lowest (third) privilege level.
VBE – Video BIOS Extensions, developed by VESA (8.01-35) in order to enable implementation of SVGA videomodes.
VC – Volcov Commander shell (6.25).
VCPI – Virtual Control Program Interface: protocol of interaction enabling control transfer from one control program to another (5.04-02).
VESA – Video Electronics Standards Association.
VGA – Video Graphics Array: video adapter for IBMs PS/2 computers.
XMS – Extended Memory Specification implemented by Himem.sys (5.04-01)
YIQ – model of pixel representation in luminance and two chrominance axes, corresponding to highest and lowest visual color resolution.
YUV – model of pixel representation in luminance and two chrominance axes, conforming to specifications of CIE (Comite Internationale d'Eclairage).
yy – year (in MS-DOS7 year is represented by a four-digit number).
ZIP – (1): suffix of archive files, compressed by PKZIP utility.
ZIP – (2): trade mark of removable disk drives, produced by Iomega Co.
ZF – CPU's zero flag, used to indicate equality or zero result.