

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ
по лабораторной работе №1
по дисциплине «Операционные системы»
Тема: Исследование структур загрузочных модулей

Студент гр. 0382

Кондратов Ю.А.

Преподаватель

Ефремов М.А.

Санкт-Петербург

2022

Цель работы.

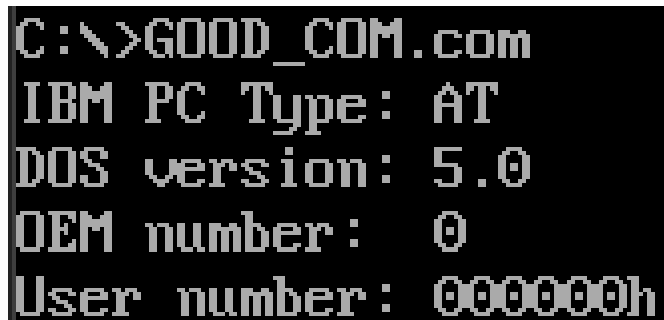
Исследование различие в структурах исходных текстов модулей типов .COM и .EXE, структур файлов загрузочных модулей и способов их загрузки в основную память.

Задание.

1. Написать текст исходного .COM модуля, который определяет тип РС и версию системы. Результатом выполнения этого шага будет «хороший» .COM модуль, а также необходимо построить «плохой» .EXE, полученный из исходного текста для .COM.
2. Написать текст исходного .EXE модуля, который выполняет те же функции, что и модуль в шаге 1 и отладить его. Таким образом, будет получен «хороший» .EXE.
3. Сравнить исходные тексты для .COM и .EXE модулей. Ответить на вопросы «Отличия исходных текстов COM и EXE программ».
4. Запустить FAR и открыть файл загрузочного модуля .COM и файл «плохого» .EXE в шестнадцатеричном виде. Затем открыть файл загрузочного модуля «хорошего» .EXE и сравнить его с предыдущими файлами. Ответить на контрольные вопросы «Отличия форматов файлов COM и EXE модулей».
5. Открыть отладчик TD.EXE и загрузить СО. Ответить на контрольные вопросы «Загрузка COM модуля в основную память». Представить в отчете план загрузки модуля .COM в основную память.
6. Открыть отладчик TD.EXE и загрузить «хороший» .EXE. Ответить на контрольные вопросы «Загрузка «хорошего» EXE в основную память».
7. Оформить отчет в соответствии с требованиями. Привести скриншоты. Для файлов их вид в шестнадцатеричном виде, для загрузочных модулей — в отладчике.

Выполнение работы.

Для построения good_com.com («хорошего» .com модуля) был написан файл исходного кода good_com.asm, в котором был использован шаблон из методических указаний и реализованные в нём функции преобразования значений байтов в строковые выражения. «Плохой» .exe модуль был построен из good_com.asm путём компиляции без флага /t. Выводы при запуске good_com.com и good_com.exe представлены на рисунках 1 и 2 соответственно.



```
C:\>GOOD_COM.com
IBM PC Type: AT
DOS version: 5.0
OEM number: 0
User number: 0000000h
```

Рисунок 1 – Вывод при запуске good_com.com

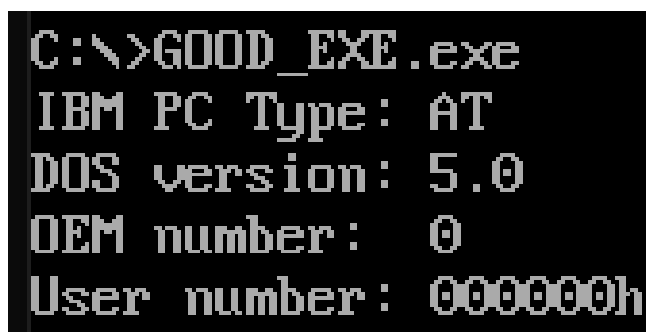


```
C:\>GOOD_COM.exe

5 0
0
0000000h
IBM PC Type: PC
IBM PC Type: PC
IBM PC Type: PC
IBM PC Type: PC
```

Рисунок 2 – Вывод при запуске good_com.exe

Для построения good_exe.exe («хорошего» .exe модуля) был написан файл исходного кода good_exe.asm, в котором был использован файл good_com.asm с добавленной разметкой сегментов. Вывод при запуске «хорошего» .exe модуля представлен на рисунке 3.



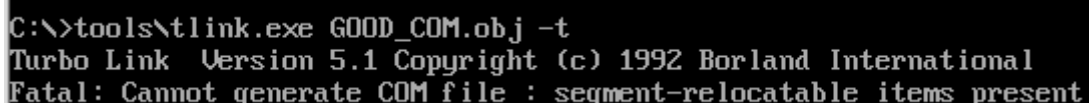
```
C:\>GOOD_EXE.exe
IBM PC Type: AT
DOS version: 5.0
OEM number: 0
User number: 0000000h
```

Рисунок 3 – Вывод при запуске good_exe.exe

Контрольные вопросы.

Отличия исходных текстов COM и EXE программ:

1. Сколько сегментов должна содержать COM-программа?
Один.
2. EXE-программа?
Может содержать несколько сегментов.
3. Какие директивы должны быть обязательно в тексте COM-программы?
ORG 100h (для смещение относительно нулевого адреса на 256 байт – место для PSP). ASSUME CS:SEG, DS:SEG, SS:NOTHING, ES:NOTHING.
4. Все ли форматы команд можно использовать в COM-программе?
Нет. Так как в .com файле отсутствует relocation table, команды вида mov *register*, *segment_name* не поддерживаются. На рисунке 4 показано, какую ошибку в случае применения таких команд выдаёт turbo link.



```
C:\>tools\tlink.exe GOOD_COM.obj -t
Turbo Link Version 5.1 Copyright (c) 1992 Borland International
Fatal: Cannot generate COM file : segment-relocatable items present
```

Рисунок 4 – Ошибка при выполнении неподдерживаемой форматом .COM команды

Отличия форматов файлов .com и .exe модулей:

1. Какова структура файла .COM? С какого адреса располагается код?
.COM файл (см. рисунок 5) состоит из единственного сегмента, началом которого инициализируются все сегментные регистры. Код располагается с адреса 0h, однако при загрузке программы в память в начало этого сегмента будет добавлен PSP размером 100h байт, поэтому выставляется смещение 100h.
2. Какова структура файла «плохого» EXE? С какого адреса располагается код?
Что располагается с адреса 0?
«Плохой» .exe (см. рисунок 6) файл состоит из единственного сегмента. Код располагается с адреса 300h (512 байт с адреса 0 – заголовок и relocation table, 256 байт – смещение).

Рисунок 5 – «Хороший» .com модуль в бинарном виде

Рисунок 6 – «Плохо» .exe модуль в бинарном виде

3. Какова структура «хорошего» EXE? Чем он отличается от файла «плохого» EXE?

В хорошем .exe модуле (см. рисунок 7) программа имеет несколько сегментов. В начале модуля расположен заголовок и relocation table (512 байт), далее расположены сегменты в том порядке, в котором они определены в коде. Сначала сегмент стека – 512 байт, далее сегмент данных, после него – сегмент кода.

C:\LETI\OC\lab_works\kondratov_lab1\GOOD_EXE.EXE										h	1252	1568
00000002B0:	00	00	00	00	00	00	00	00	00	00	00	00
00000002C0:	00	00	00	00	00	00	00	00	00	00	00	00
00000002D0:	00	00	00	00	00	00	00	00	00	00	00	00
00000002E0:	00	00	00	00	00	00	00	00	00	00	00	00
00000002F0:	00	00	00	00	00	00	00	00	00	00	00	00
0000000300:	00	00	00	00	00	00	00	00	00	00	00	00
0000000310:	00	00	00	00	00	00	00	00	00	00	00	00
0000000320:	00	00	00	00	00	00	00	00	00	00	00	00
0000000330:	00	00	00	00	00	00	00	00	00	00	00	00
0000000340:	00	00	00	00	00	00	00	00	00	00	00	00
0000000350:	00	00	00	00	00	00	00	00	00	00	00	00
0000000360:	00	00	00	00	00	00	00	00	00	00	00	00
0000000370:	00	00	00	00	00	00	00	00	00	00	00	00
0000000380:	00	00	00	00	00	00	00	00	00	00	00	00
0000000390:	00	00	00	00	00	00	00	00	00	00	00	00
00000003A0:	00	00	00	00	00	00	00	00	00	00	00	00
00000003B0:	00	00	00	00	00	00	00	00	00	00	00	00
00000003C0:	00	00	00	00	00	00	00	00	00	00	00	00
00000003D0:	00	00	00	00	00	00	00	00	00	00	00	00
00000003E0:	00	00	00	00	00	00	00	00	00	00	00	00
00000003F0:	00	00	00	00	00	00	00	00	00	00	00	00
0000000400:	49	42	4D	20	50	43	20	54	79	70	65	3A
0000000410:	0A	24	49	42	4D	20	50	43	20	54	79	70
0000000420:	43	2F	58	54	0D	0A	24	49	42	4D	20	50
0000000430:	70	65	3A	20	41	54	0D	0A	24	49	42	4D
0000000440:	54	79	70	65	3A	20	50	53	32	20	6D	6F
0000000450:	33	30	0D	0A	24	49	42	4D	20	50	43	20
0000000460:	3A	20	50	53	32	20	6D	6F	64	65	6C	20

Рисунок 7 – «Плохой» .exe модуль

Загрузка .com модуля в основную память:

1. Какой формат загрузки модуля COM? С какого адреса располагается код?

Загрузка .com модуля в память происходит следующим образом: в основной памяти выделяется свободный сегмент, в первых 256 байтах этого сегмента генерируется PSP, далее записывается сама программа. Код располагается с адреса CS:0100 = 48DD:0100 (см. рисунок 8).

2. Что располагается с адреса 0?

PSP.

3. Какие значения имеют сегментные регистры? На какие области памяти они указывают?

Все сегментные регистры указывают на начало PSP. Их значения равны 48DD (см. рисунок 8).

4. Как определяется стек? Какую область памяти он занимает? Какие адреса? Под стек отведен весь сегмент, в который загружена программа. SS=48DD указывает на начало сегмента SP=FFFE указывает на последний адрес сегмента кратный двум. Адреса: 48DD:0000 – 48DD:FFFE (см. рисунок 8).

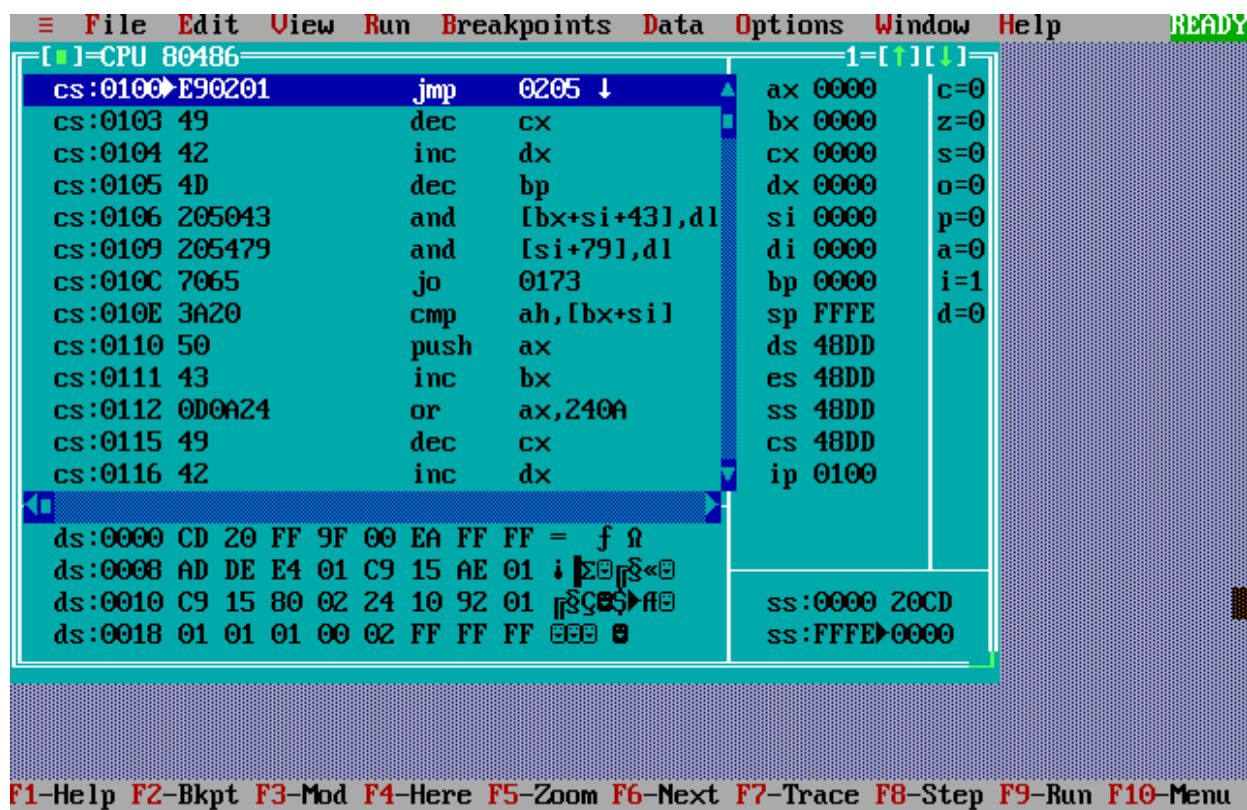


Рисунок 8 – Начальное состояние загруженного .com модуля

Загрузка «хорошего» .exe модуля в основную память:

1. Как загружается «хороший» .exe? Какие значения имеют сегментные регистры?

Сначала в память загружается PSP, после которого загружается .exe модуль в соответствии с информацией в заголовке. Значение регистров см. на рисунке 9.

2. На что указывают DS и ES?

На начало PSP.

3. Как определяется стек?

Стек определяется при помощи описание стэкового сегмента в коде и директивы ASSUME. SS указывает на начало сегмента стэка, а SP – на конец стэка (на рисунке 8 видно, что SP = 0100h так как размер стэка – 256 байт).

4. Как определяется точка входа?

Точка входа определяется при помощи директивы END.

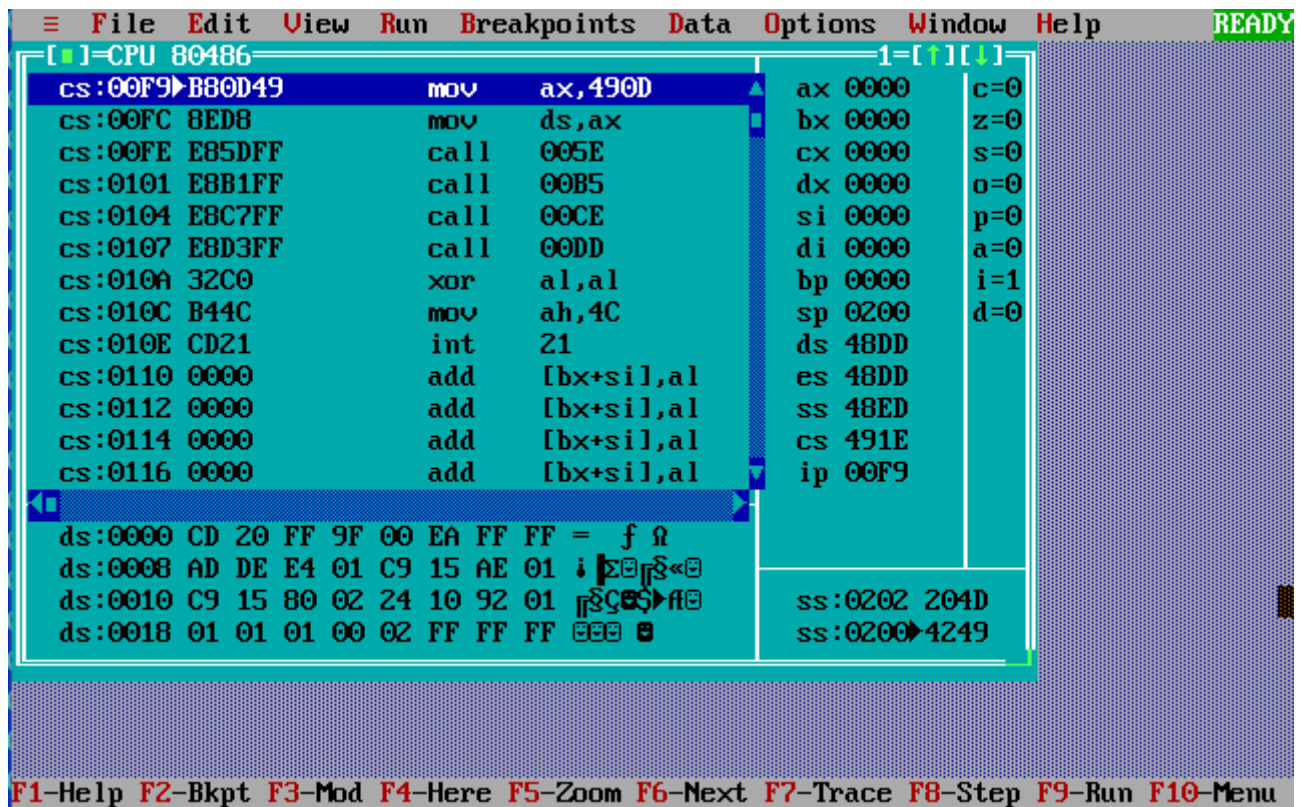


Рисунок 8 – Начальное состояние загруженного .exe модуля

Выводы.

В ходе работы были изучены основные принципы устройства .com и .exe исполняемых модулей, их сходства и различия, алгоритм загрузки в память и инициализации начального состояния. Написана программа, выводящая строковую информацию о версии операционной системы DOS.

ПРИЛОЖЕНИЕ А.

Исходный код модулей

good_com.asm:

```
main_seg SEGMENT
    ASSUME CS:main_seg, DS:main_seg, ES:NOTHING, SS:NOTHING
    ORG 100h

start:
    jmp begin

data:
    PC db  'IBM PC Type: PC',0DH,0AH,'$'
    PCXT db 'IBM PC Type: PC/XT',0DH,0AH,'$'
    AT db  'IBM PC Type: AT',0DH,0AH,'$'
    PS230 db 'IBM PC Type: PS2 model 30',0DH,0AH,'$'
    PS25060 db 'IBM PC Type: PS2 model 50 or 60',0DH,0AH,'$'
    PS280 db 'IBM PC Type: PS2 model 80',0DH,0AH,'$'
    PCjr db 'IBM PC Type: PCjr',0DH,0AH,'$'
    PCC db 'IBM PC Type: PC Convertible',0DH,0AH,'$'
    DOSV db 'DOS version:  . ',0DH,0AH,'$'
    OEM db 'OEM number:    ',0DH, 0AH, '$'
    USERN db 'User number:      h',0DH, 0AH, '$'

begin:
    call main
    xor al, al
    mov ah, 4Ch
    int 21h

print PROC NEAR
    mov ah, 09h
    int 21h
    ret
print ENDP

tetr_to_hex PROC near
    and AL,0Fh
    cmp AL,09
    jbe next
    add AL,07
next:
    add AL,30h
    ret
tetr_to_hex ENDP

byte_to_hex PROC near
    push CX
    mov AH,AL
    call tetr_to_hex
    xchg AL,AH
    mov CL,4
    shr AL,CL
    call tetr_to_hex
    pop CX
    ret
```

```

byte_to_hex ENDP

wrd_to_hex PROC near
    push BX
    mov BH,AH
    call byte_to_hex
    mov [DI],AH
    dec DI
    mov [DI],AL
    dec DI
    mov AL,BH
    call byte_to_hex
    mov [DI],AH
    dec DI
    mov [DI],AL
    pop BX
    ret
wrd_to_hex ENDP

byte_to_dec PROC near
    push CX
    push DX
    push ax
    xor AH,AH
    xor DX,DX
    mov CX,10
loop_bd:
    div CX
    or DL,30h
    mov [SI],DL
    dec SI
    xor DX,DX
    cmp AX,10
    jae loop_bd
    cmp AL,00h
    je end_l
    or AL,30h
    mov [SI],AL
end_l:
    pop ax
    pop DX
    pop CX
    ret
byte_to_dec ENDP

print_pc_type PROC NEAR
    mov ax, 0F000h
    mov es, ax
    mov al, es:[0FFFEh]
    cmp ah, 0FFh

    cmp al, 0ffh
    je _pc
    cmp al, 0feh
    je pc_xt
    cmp al, 0fbh
    je pc_xt
    cmp al, 0fch

```

```

        je _at
        cmp al, 0fah
        je ps2_30
        cmp al, 0f8h
        je ps2_80
        cmp al, 0fdh
        je pc_jr
        cmp al, 0f9h
        je pc_conv

_pc:
    mov dx, offset PC
    jmp _out
pc_xt:
    mov dx, offset PC_XT
    jmp _out
_at:
    mov dx, offset AT
    jmp _out
ps2_30:
    mov dx, offset PS230
    jmp _out
ps2_80:
    mov dx, offset PS280
    jmp _out
pc_jr:
    mov dx, offset PCjr
    jmp _out
pc_conv:
    mov dx, offset PCC
_out:
    call print
    ret
print_pc_type ENDP

print_dos_vers PROC NEAR
    mov ah, 30h
    int 21h
    mov si, offset DOSV + 13
    call byte_to_dec
    mov al, ah
    add si, 3
    call byte_to_dec
    mov dx, offset DOSV
    call print
    ret
print_dos_vers ENDP

print_oem_num PROC NEAR
    mov si, offset OEM + 13
    mov al, bh
    call byte_to_dec
    mov dx, offset OEM
    call print
    ret
print_oem_num ENDP

print_user_num PROC NEAR

```

```

        mov di, offset USERN
        add di, 18
        mov ax, cx
        call wrd_to_hex
        mov al, bl
        call byte_to_hex
        mov di, offset USERN + 13
        mov [di], ax
        mov dx, offset USERN
        call print
        ret
print_user_num ENDP

main PROC NEAR
        call print_pc_type
        call print_dos_vers
        call print_oem_num
        call print_user_num
        ret
main ENDP

main_seg ENDS
END start

```

good_exe.asm:

```

AStack SEGMENT STACK
        DW 256 DUP (?)
AStack ENDS

data SEGMENT
        PC db  'IBM PC Type: PC',0DH,0AH,'$'
        PCXT db 'IBM PC Type: PC/XT',0DH,0AH,'$'
        AT db  'IBM PC Type: AT',0DH,0AH,'$'
        PS230 db 'IBM PC Type: PS2 model 30',0DH,0AH,'$'
        PS25060 db 'IBM PC Type: PS2 model 50 or 60',0DH,0AH,'$'
        PS280 db 'IBM PC Type: PS2 model 80',0DH,0AH,'$'
        PCjr db 'IBM PC Type: PCjr',0DH,0AH,'$'
        PCC db 'IBM PC Type: PC Convertible',0DH,0AH,'$'
        DOSV db 'DOS version:  . ',0DH,0AH,'$'
        OEM db 'OEM number:    ',0DH, 0AH, '$'
        USERN db 'User number:      h',0DH, 0AH, '$'
data ENDS

main_seg SEGMENT
        ASSUME CS:main_seg, DS:data, SS:AStack

print PROC NEAR
        mov ah, 09h
        int 21h
        ret
print ENDP

tetr_to_hex PROC near
        and AL,0Fh
        cmp AL,09
        jbe next

```

```

        add AL,07
next:
        add AL,30h
        ret
tetr_to_hex ENDP

byte_to_hex PROC near
        push CX
        mov AH,AL
        call tetr_to_hex
        xchg AL,AH
        mov CL,4
        shr AL,CL
        call tetr_to_hex
        pop CX
        ret
byte_to_hex ENDP

wrd_to_hex PROC near
        push BX
        mov BH,AH
        call byte_to_hex
        mov [DI],AH
        dec DI
        mov [DI],AL
        dec DI
        mov AL,BH
        call byte_to_hex
        mov [DI],AH
        dec DI
        mov [DI],AL
        pop BX
        ret
wrd_to_hex ENDP

byte_to_dec PROC near
        push CX
        push DX
        push ax
        xor AH,AH
        xor DX,DX
        mov CX,10
loop_bd:
        div CX
        or DL,30h
        mov [SI],DL
        dec SI
        xor DX,DX
        cmp AX,10
        jae loop_bd
        cmp AL,00h
        je end_l
        or AL,30h
        mov [SI],AL
end_l:
        pop ax
        pop DX
        pop CX

```

```

        ret
byte_to_dec ENDP

print_pc_type PROC NEAR
    mov ax, 0F000h
    mov es, ax
    mov al, es:[0FFFEh]
    cmp ah, 0FFh

    cmp al, 0ffh
    je _pc
    cmp al, 0feh
    je pc_xt
    cmp al, 0fbh
    je pc_xt
    cmp al, 0fch
    je _at
    cmp al, 0fah
    je ps2_30
    cmp al, 0f8h
    je ps2_80
    cmp al, 0fdh
    je pc_jr
    cmp al, 0f9h
    je pc_conv

_pc:
    mov dx, offset PC
    jmp _out
pc_xt:
    mov dx, offset PC_XT
    jmp _out
_at:
    mov dx, offset AT
    jmp _out
ps2_30:
    mov dx, offset PS230
    jmp _out
ps2_80:
    mov dx, offset PS280
    jmp _out
pc_jr:
    mov dx, offset PCjr
    jmp _out
pc_conv:
    mov dx, offset PCC
_out:
    call print
    ret
print_pc_type ENDP

print_dos_vers PROC NEAR
    mov ah, 30h
    int 21h
    mov si, offset DOSV + 13
    call byte_to_dec
    mov al, ah
    add si, 3

```

```

        call byte_to_dec
        mov dx, offset DOSV
        call print
        ret
print_dos_vers ENDP

print_oem_num PROC NEAR
        mov si, offset OEM + 13
        mov al, bh
        call byte_to_dec
        mov dx, offset OEM
        call print
        ret
print_oem_num ENDP

print_user_num PROC NEAR
        mov di, offset USERN
        add di, 18
        mov ax, cx
        call wrd_to_hex
        mov al, bl
        call byte_to_hex
        mov di, offset USERN + 13
        mov [di], ax
        mov dx, offset USERN
        call print
        ret
print_user_num ENDP

main PROC FAR
        mov ax, data
        mov ds, ax
        call print_pc_type
        call print_dos_vers
        call print_oem_num
        call print_user_num
        xor al, al
        mov ah, 4Ch
        int 21h
main ENDP

main_seg ENDS
END main

```