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

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Операционные системы»

Тема: Исследование структур загрузочных модулей

Студентка гр. 7383 _____ Маркова А.В.

Преподаватель _____ Ефремов М.А.

Санкт-Петербург

2019

Постановка задачи.

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

Описание процедур, используемых в работе:

- TETR_TO_HEX – вспомогательная функция, которая переводит из двоичной в шестнадцатеричную систему счисления, используется для работы функции BYTE_TO_HEX;
- BYTE_TO_HEX – переводит байтовое число из регистра AL в шестнадцатеричную систему счисления;
- WRD_TO_HEX – переводит число из регистра AX в шестнадцатеричную систему;
- BYTE_TO_DEC – переводит байтовое число в десятичную систему счисления;
- LINE_OUTPUT – выводит сообщение на экран;
- GET_PC_TYPE – определение типа IBP PC из предпоследнего байта ROM BIOS;
- Write_PC_TYPE – вывод типа IBM PC на экран;
- Write_PC_VERS – определяет номер версии PC и выводит его на экран;
- Write_PC_OEM – определяет серийный номер OEM и выводит его на экран;
- Write_SER_NUM – вывод на экран серийного номера пользователя;

Таблица 1 – Описание структур данных:

Название	**Тип**	**Назначение**
PC_TYPE	db	Тип IBM PC
PC_VERS	db	Версия PC
PC_OEM	db	Серийный номер OEM
SER_NUM	db	Серийный номер пользователя

Окончание таблицы 1

TYPE_PC	db	PC (код: FF)
TYPE_PCXT	db	PC/XT (код: FE, FB)
TYPE_AT	db	AT (код: FC)
TYPE_PS2_30	db	PS2 модель 30 (код: FA)
TYPE_PS2_50_60	db	PS2 модель 50/60 (код: FC)
TYPE_PS2_80	db	PS2 модель 80 (код: F8)
TYPE_PCjr	db	PCjr (код: FD)
TYPE_PC_Con	db	PC Convertible (код: F9)

Ход работы.

1. Был написан текст исходного .COM модуля, который определяет тип IBM PC, версию системы, номер OEM и серийный номер пользователя.
2. Смонтирован виртуальный диск k с каталогом tasm с помощью команды, представленной на рис. 1.

```
Z:\>mount k d:/7383/lr1/tasm
Drive K is mounted as local directory d:/7383/lr1/tasm\

Z:\>k:\
```

Рисунок 1 – Создание виртуального диска

3. Было произведено транслирование программы с помощью строки tasm «имя файла», в последствии чего создался объектный файл, результат вызова команды показан на рис. 2.

```
K:\>tasm bad
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland International

Assembling file:   bad.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  471k
```

Рисунок 2 – Транслирование программы

4. Линковка загрузочного модуля с помощью команды tlink и tlink/t.
5. Получены «хороший» .COM модуль и «плохой» .EXE из исходного текста для .COM модуля.
6. Был написан текст исходного .EXE модуля, который выполняет те же функции, получен «хороший» .EXE.
7. Результаты работы программы показаны на рис. 3 – 5.

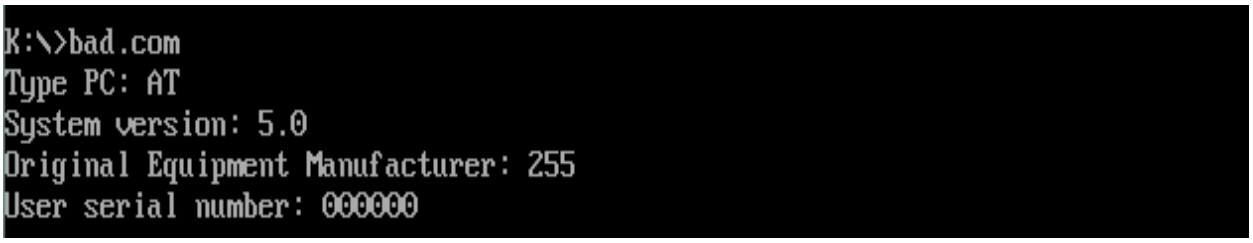

Рисунок 3 – Результат выполнения программы bad.com

Рисунок 4 – Результат выполнения программы bad.exe

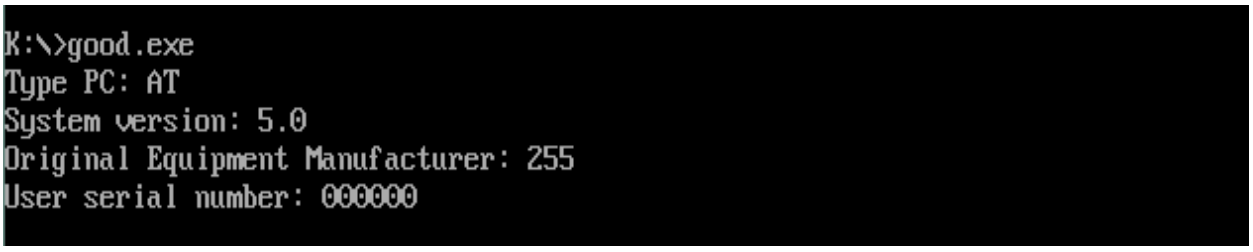

Рисунок 5 – Результат выполнения программы good.exe

Выводы.

В ходе данной лабораторной работы были исследованы различия в структурах исходных текстов модулей типов .COM и .EXE, структур файлов загрузочных модулей и способов их загрузки в основную память. Код программы bad_exe.asm представлен в приложении А, код программы good_exe.asm представлен в приложении Б. Было замечено, что структура EXE программ сложнее структуры COM программ.

Ответы на контрольные вопросы.

Отличия исходных текстов COM и EXE программ

1. Сколько сегментов должна содержать COM-программа?

   Один сегмент.

2. EXE-программа?

   Минимум один сегмент, может состоять и из нескольких.

3. Какие директивы должны обязательно быть в тексте COM-программы?

Обязательными директивами являются ORG 100h, которая сдвигает адресацию в программе на 256 байт для расположения PSP, и ASSUME, которая ставит в соответствие сегментам CS и DS начало программы. Если же не будет директивы ASSUME, то программа не скомпилируется, так как не обнаружит начало сегмента кода.

4. Все ли форматы команд можно использовать в COM-программе?

Не все, в COM-программе нельзя использовать команды с дальней адресацией, так как в этих командах используется таблица настройки, в которой содержатся адреса сегментов, однако такая таблица существует только в EXE-файлах, в COM-программах нельзя использовать сегментную адресацию.

То есть в COM-программе нельзя использовать команды вида mov регистр или сегмент.

Отличия форматов файлов COM и EXE модулей

1. Какова структура файла COM? С какого адреса располагается код?

В файлах данного типа содержатся только машинный код и данные программы. В файле код начинается с нулевого адреса. Структура COM-файла представлена на рис. 6.

Рисунок 6 – HEX представление .COM файла

HEX – представление было получено в FarManager с помощью нажатия комбинации клавиш F3 и F4.

2. Какова структура файла «плохого» EXE? С какого адреса располагается код? Что располагается с адреса 0?

По рис. 7 видно, что структура «плохого» EXE-файла больше, чем у COM-файла. Код располагается с адреса 300h, так как заголовок занимает 200h байт и команда ORG 100h сдвигает код на дополнительные 100h. Следовательно с нулевого адреса располагается заголовок. В первых двух

байтах видим символы MZ, которые указывают на то, что формат файла 16-битный и его следует запускать в соответствии со структурой EXE-файлов. Затем идёт таблица настроек, при помощи которых строится данный EXE-файл и зарезервированные директивой ORG 100h байт.

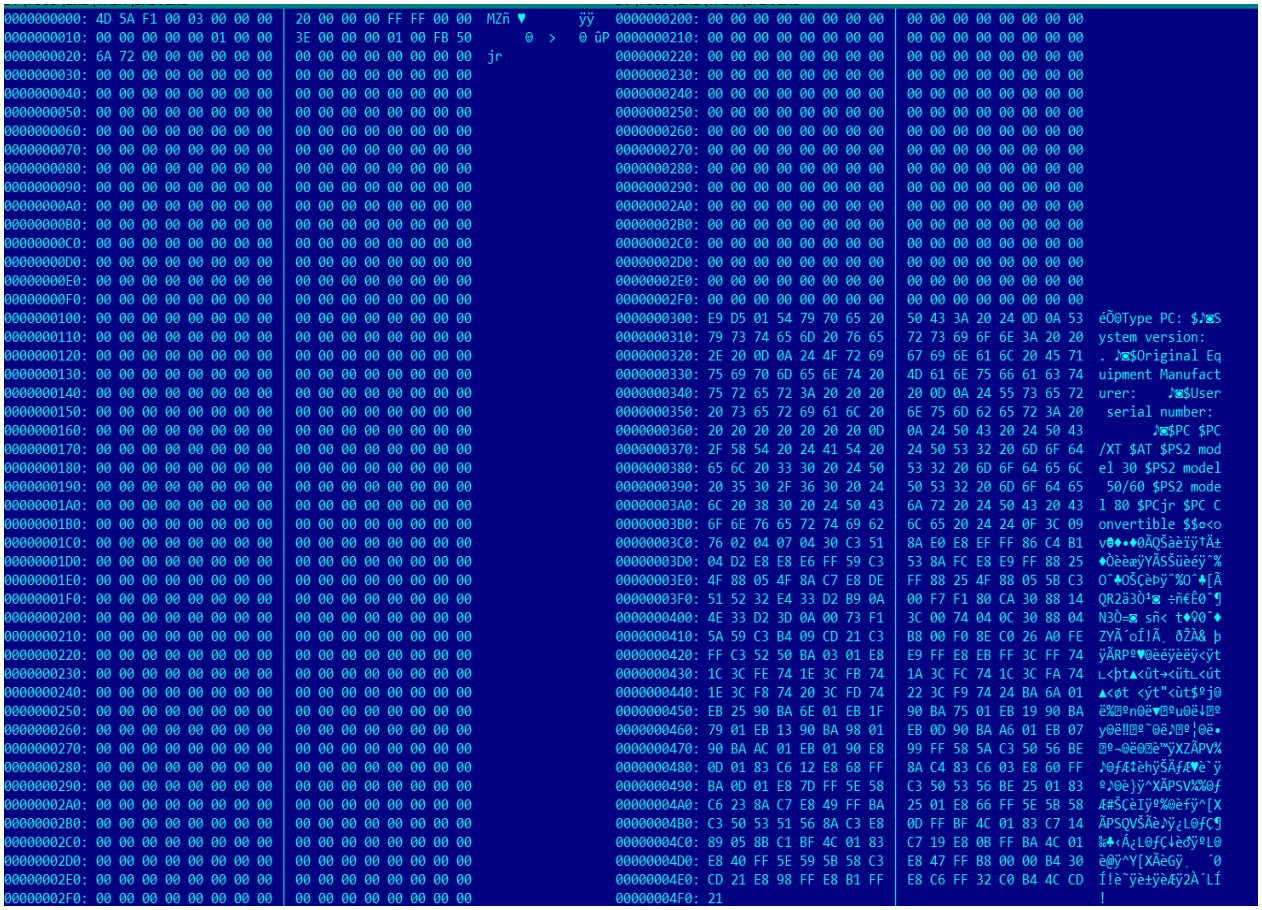

Рисунок 7 – HEX-представление «плохого» .EXE-файла

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

В отличие от «плохого» EXE, в «хорошем» код, стек и данные выделены в отдельные сегменты. В структуре «хорошего» файла отсутствует директива ORG 100h, которая резервировала пространство, так как в EXE нет необходимости её использовать, потому что загрузчик автоматически ставит программу после PSP. С нулевого адреса все так же находится заголовок с таблицей настроек. С адреса 200h в «хорошем» EXE-файле идет сегмент стека,

под который дополнительно выделено 20 байт, а с адреса 220h располагается код программы. Иллюстрация «хорошего» EXE–файла представлена на рис. 8.

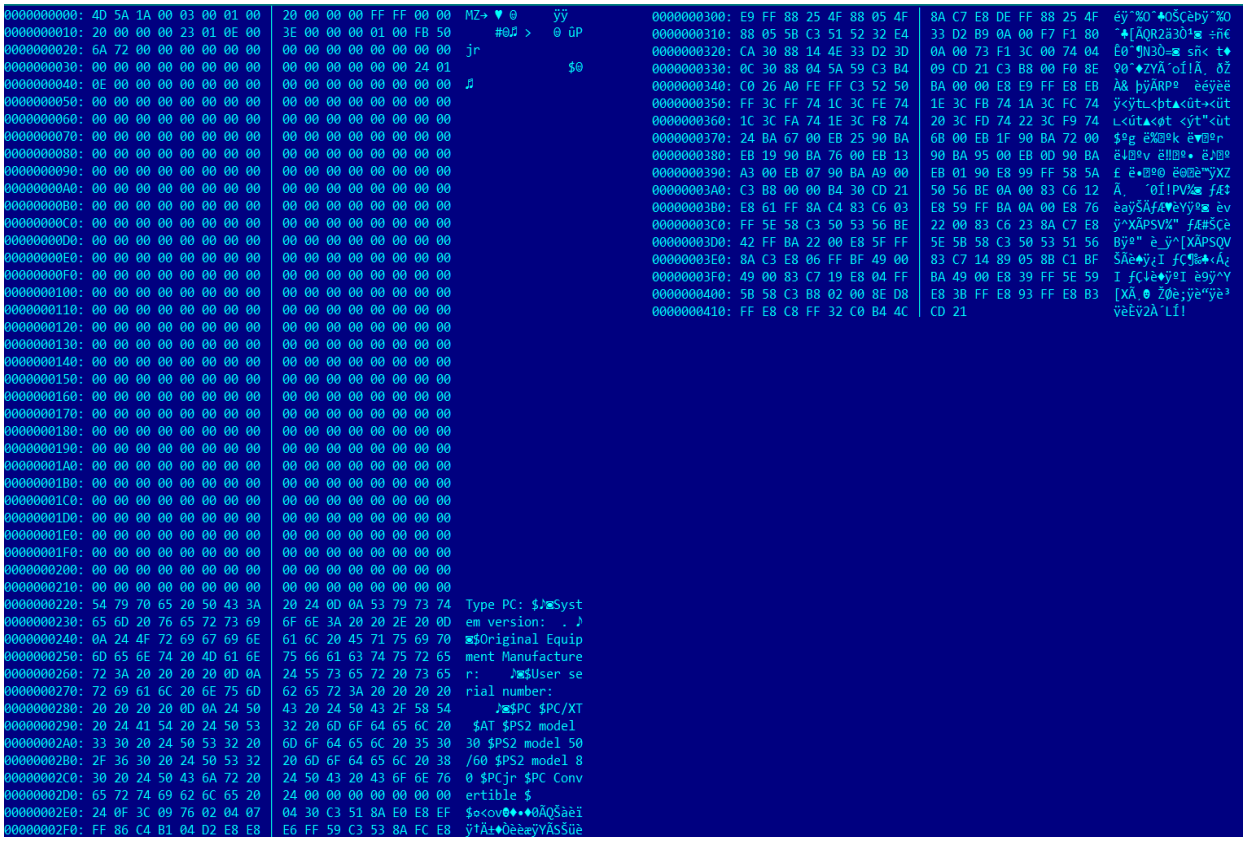

Рисунок 8 – HEX – представление «хорошего» EXE – файла

Загрузка COM модуля в основную память

Для ответа на следующие вопросы открыли отладчик с помощью команды td «имя файла», появившееся окно представлено на рис. 9.

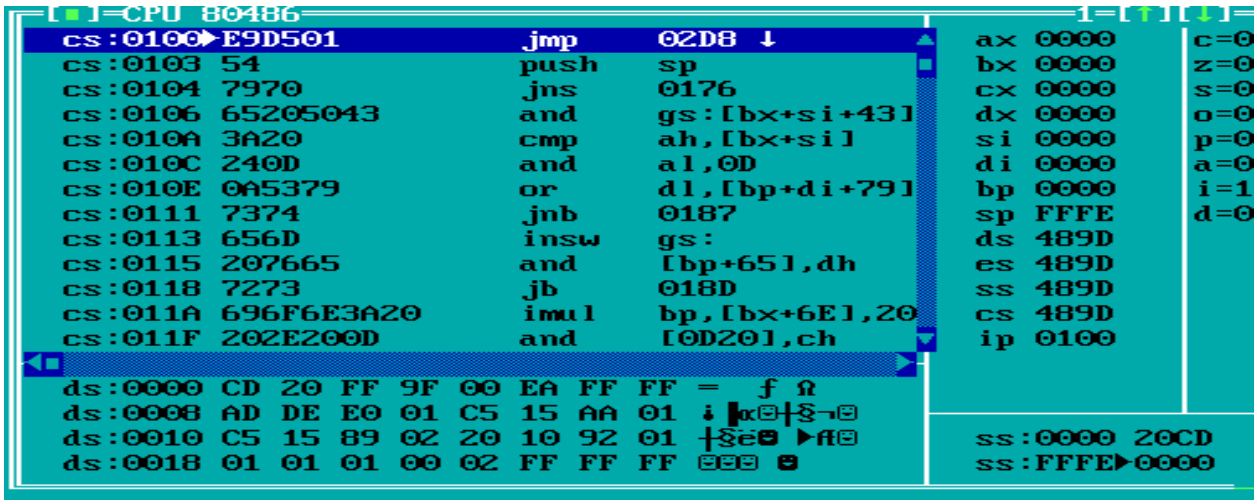

Рисунок 9 – Результат загрузки .COM в отладчик

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

Сначала система выделяет свободный сегмент памяти, затем заносит его адрес во все сегментные регистры. Потом происходит построение PSP в первых 100h байтах памяти. Загружается содержимое COM-файла и присваивается регистру IP значение 100h. Указатель на стек SP указывает на конец выделенной памяти, поэтому запись данных стека осуществляется снизу вверх.

Начало кода определяется директивой ORG 100h от начала выделенного фрагмента, в данном случае код будет располагаться с адреса 489D:0100h.

2. Что располагается с адреса 0?

PSP.

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

Все сегментные регистры (CS, SS, ES, DS) имеют одно и то же значение равное 489D (см. рис. 9). И все они указывают на начало сегмента программы, то есть на начало PSP.

4. Как определяется стек? Какую область памяти он занимает? Какие адреса?

DOS автоматически определяет стек. Если для программы размер сегмента в 64K достаточен, то DOS устанавливает SP = FFFE. Стек занимает весь сегмент COM – программы, а его начало находится в конце сегмента. SS указывает на начало, а SP – на конец. При добавлении данных в стек, адрес на который указвает SP уменьшается и данные записываются в конце выделенной области памяти, то есть заполняется снизу вверх. Стек может дойти до кода/данных программы при большом количестве элементов и тем самым переписать их. Адреса могут быть расположены в диапазоне 0000h-FFFEh.

Загрузка «хорошего» EXE модуля в основную память

Результат запуска «хорошего» EXE модуля в отладчике представлен на рис. 10.

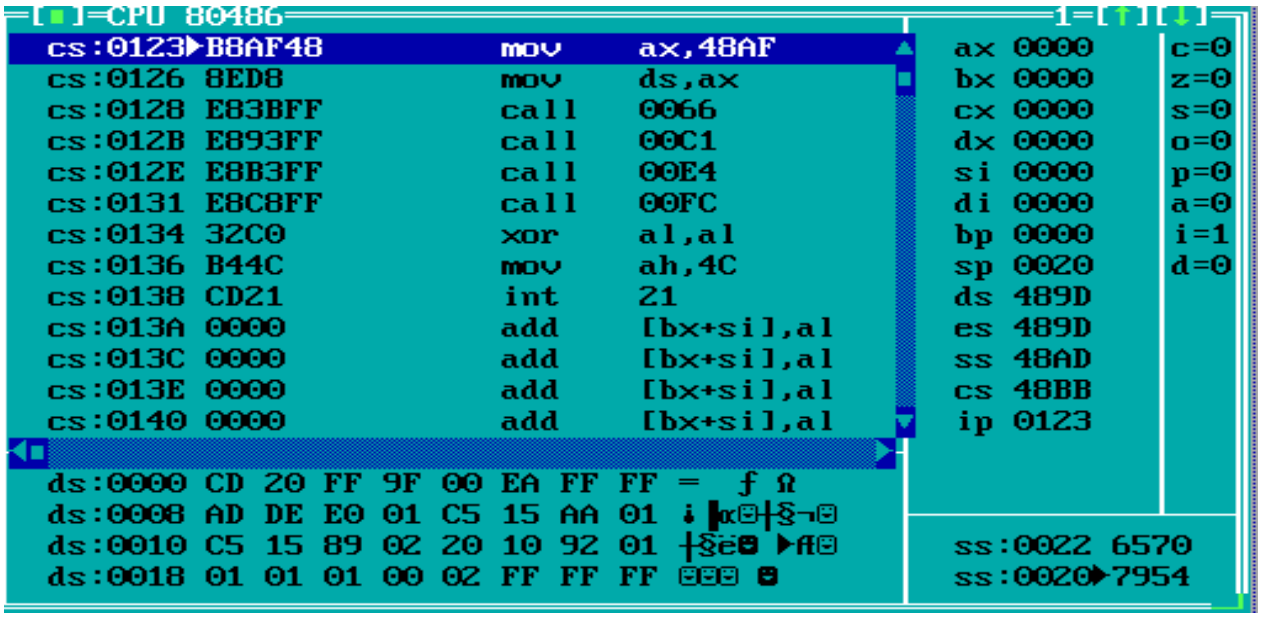

Рисунок 10 – Результат загрузки «хорошего» .EXEв отладчик

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

Сначала помещается PSP, а дальше устанавливаются сегментные регистры. DS и ES устанавливаются на начало PSP (DS = ES = 489D). SS – на начало сегмента стека (SS = 48AD), CS – на начало сегмента кода (CS = 48BB).

2. На что указывают регистры DS и ES?

На начало PSP.

3. Как определяется стек?

В исходном коде модуля стек определяется при помощи директивы STACK. При запуске программы в SS заносится адрес сегмента стека, его начало, а в SP – адрес его вершины. (см. рис. 10).

4. Как определяется точка входа?

Точка входа в программу определяется с помощью директивы END, после которой указывается метка, куда переходит программа при запуске.

ПРИЛОЖЕНИЕ А

BAD_EXE.ASM

```
TESTPC SEGMENT
      ASSUME CS:TESTPC, DS:TESTPC, ES:NOTHING, SS:NOTHING
      ORG 100H
START: JMP BEGIN
PC_TYPE    db 'Type PC: $'
PC_VERS    db 0DH,0AH,'System version:  . ',0DH,0AH,'$'
PC_OEM     db 'Original Equipment Manufacturer:    ',0DH,0AH,'$'
SER_NUM    db 'User serial number:        ',0DH,0AH,'$'
TYPE_PC          db 'PC $'
TYPE_PCXT        db 'PC/XT $'
TYPE_AT          db 'AT $'
TYPE_PS2_30      db 'PS2 model 30 $'
TYPE_PS2_50_60   db 'PS2 model 50/60 $'
TYPE_PS2_80      db 'PS2 model 80 $'
TYPE_PCjr        db 'PCjr $'
TYPE_PC_Con      db 'PC Convertible $'
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
      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 DX
      pop  CX
      ret
BYTE_TO_DEC ENDP
LINE_OUTPUT PROC near
    mov  AH,09H
      int  21H
      ret
LINE_OUTPUT ENDP

GET_PC_TYPE PROC near
      mov AX,0F000H
```

```
      mov ES,AX
      mov AL,ES:[0FFFEH]
      ret
GET_PC_TYPE ENDP

Write_PC_TYPE PROC near
    push DX
      push AX
      mov  DX, OFFSET PC_TYPE
      call LINE_OUTPUT
      call GET_PC_TYPE
      cmp  AL,0FFH
      je   PC

      cmp  AL,0FEH
      je   PCXT

      cmp  AL,0FBH
      je   PCXT

      cmp  AL,0FCH
      je   PC_AT

      cmp  AL,0FAH
      je   PS2_30

      cmp  AL,0F8H
      je   PS2_80

      cmp  AL,0FDH
      je   PCjr

      cmp  AL,0F9H
      je   PC_Convertible

      PC:
            mov  DX, OFFSET TYPE_PC
            jmp  print
      PCXT:
            mov  DX, OFFSET TYPE_PCXT
            jmp  print
      PC_AT:
            mov  DX, OFFSET TYPE_AT
```

```
            jmp  print
      PS2_30:
            mov  DX, OFFSET TYPE_PS2_30
            jmp  print
      PS2_80:
            mov  DX, OFFSET TYPE_PS2_80
            jmp  print
      PCjr:
            mov  DX, OFFSET TYPE_PCjr
            jmp  print
      PC_Convertible:
            mov  DX, OFFSET TYPE_PC_Con
            jmp  print
    print:
        call LINE_OUTPUT
        pop AX
        pop DX
        ret
Write_PC_TYPE ENDP

Write_PC_VERS PROC near
      push AX
      push SI
      mov  SI,OFFSET PC_VERS
      add  SI,18
      call BYTE_TO_DEC
      mov  AL,AH
      add  SI,3
      call BYTE_TO_DEC
      mov  DX,OFFSET PC_VERS
      call LINE_OUTPUT
      pop  SI
      pop  AX
      ret
Write_PC_VERS  ENDP

Write_PC_OEM PROC near
    push AX
      push BX
      push SI
      mov  SI,OFFSET PC_OEM
      add  SI,35
      mov  AL,BH
```

```
      call BYTE_TO_DEC
      mov  DX,OFFSET PC_OEM
      call LINE_OUTPUT
      pop  SI
      pop  BX
      pop  AX
      ret
Write_PC_OEM ENDP
Write_SER_NUM PROC near
      push AX
      push BX
      push CX
      push SI
      mov  AL,BL
      call BYTE_TO_HEX
      mov  DI,OFFSET SER_NUM
      add  DI,20
      mov  [DI],AX
      mov  AX,CX
      mov  DI,OFFSET SER_NUM
      add  DI,25
      call WRD_TO_HEX
      mov  DX,OFFSET SER_NUM
      call LINE_OUTPUT
      pop SI
      pop CX
      pop BX
      pop AX
      ret
Write_SER_NUM ENDP
BEGIN:
      call Write_PC_TYPE
      mov  AX,0
      mov  AH,30H
      int  21H
      call Write_PC_VERS
      call Write_PC_OEM
      call Write_SER_NUM
      xor AL,AL
      mov AH,4CH
      int 21H
TESTPC ENDS
       END START
```

ПРИЛОЖЕНИЕ Б

GOOD_EXE.ASM

```
ASTACK SEGMENT STACK
       DW 010H DUP(?)
ASTACK ENDS

DATA SEGMENT
PC_TYPE    db 'Type PC: $'
PC_VERS    db 0DH,0AH,'System version:  . ',0DH,0AH,'$'
PC_OEM     db 'Original Equipment Manufacturer:    ',0DH,0AH,'$'
SER_NUM    db 'User serial number:        ',0DH,0AH,'$'
TYPE_PC         db 'PC $'
TYPE_PCXT       db 'PC/XT $'
TYPE_AT         db 'AT $'
TYPE_PS2_30     db 'PS2 model 30 $'
TYPE_PS2_50_60  db 'PS2 model 50/60 $'
TYPE_PS2_80     db 'PS2 model 80 $'
TYPE_PCjr       db 'PCjr $'
TYPE_PC_Con     db 'PC Convertible $'
DATA ENDS
CODE SEGMENT
ASSUME CS:CODE, DS:DATA, ES:NOTHING, SS:ASTACK
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
      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 DX
      pop  CX
      ret
BYTE_TO_DEC ENDP
LINE_OUTPUT PROC near
    mov  AH,09H
      int  21H
```

```
      ret
LINE_OUTPUT ENDP

GET_PC_TYPE PROC near
      mov AX,0F000H
      mov ES,AX
      mov AL,ES:[0FFFEH]
      ret
GET_PC_TYPE ENDP

Write_PC_TYPE PROC near
     push DX
      push AX
      mov  DX, OFFSET PC_TYPE
      call LINE_OUTPUT
      call GET_PC_TYPE

      cmp  AL,0FFH
      je   PC

      cmp  AL,0FEH
      je   PCXT

      cmp  AL,0FBH
      je   PCXT

      cmp  AL,0FCH
      je   PC_AT

      cmp  AL,0FAH
      je   PS2_30

      cmp  AL,0F8H
      je   PS2_80

      cmp  AL,0FDH
      je   PCjr

      cmp  AL,0F9H
      je   PC_Convertible

      PC:
            mov  DX, OFFSET TYPE_PC
```

```
            jmp  print
      PCXT:
            mov  DX, OFFSET TYPE_PCXT
            jmp  print
      PC_AT:
            mov  DX, OFFSET TYPE_AT
            jmp  print
      PS2_30:
            mov  DX, OFFSET TYPE_PS2_30
            jmp  print
      PS2_80:
            mov  DX, OFFSET TYPE_PS2_80
            jmp  print
      PCjr:
            mov  DX, OFFSET TYPE_PCjr
            jmp  print
      PC_Convertible:
            mov  DX, OFFSET TYPE_PC_Con
            jmp  print
    print:
        call LINE_OUTPUT
        pop AX
        pop DX
        ret
Write_PC_TYPE ENDP

Write_PC_VERS PROC near
     mov  AX,0
      mov  AH,30H
      int  21H
      push AX
      push SI
      mov  SI,OFFSET PC_VERS
      add  SI,18
      call BYTE_TO_DEC
      mov  AL,AH
      add  SI,3
      call BYTE_TO_DEC
      mov  DX,OFFSET PC_VERS
      call LINE_OUTPUT
      pop  SI
      pop  AX
      ret
```

```
Write_PC_VERS  ENDP

Write_PC_OEM PROC near
    push AX
     push BX
     push SI
     mov  SI,OFFSET PC_OEM
     add  SI,35
     mov  AL,BH
     call BYTE_TO_DEC
     mov  DX,OFFSET PC_OEM
     call LINE_OUTPUT
     pop  SI
     pop  BX
     pop  AX
     ret
Write_PC_OEM ENDP

Write_SER_NUM PROC near
     push AX
     push BX
     push CX
     push SI
     mov  AL,BL
     call BYTE_TO_HEX
     mov  DI,OFFSET SER_NUM
     add  DI,20
     mov  [DI],AX
     mov  AX,CX
     mov  DI,OFFSET SER_NUM
     add  DI,25
     call WRD_TO_HEX
     mov  DX,OFFSET SER_NUM
     call LINE_OUTPUT
     pop SI
     pop CX
     pop BX
     pop AX
     ret
Write_SER_NUM ENDP
BEGIN:

     mov  AX,DATA
```

```
      mov  DS,AX
      call Write_PC_TYPE
      call Write_PC_VERS
      call Write_PC_OEM
      call Write_SER_NUM
      xor AL,AL
      mov AH,4CH
      int 21H
CODE ENDS
END BEGIN
```