МИНОБРНАУКИ РОССИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САРАТОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ИМЕНИ ГАГАРИНА Ю.А.»
(СГТУ имени Гагарина Ю.А.)
Саратовский колледж машиностроения и энергетики

«СОГЛАСОВАНО»	«УТВЕРЖДАЮ»
Решением П(Ц)МК	Заместитель директора по УР
Протокол № 8	_____ С.В. Клюквина
от «28» июня 2021 г.	«___» _____ 20__ г.

Методические рекомендации для студентов по проведению лабораторных работ по дисциплине «МДК 01.04. Системное программирование»

для специальности 09.02.07Информационные системы и программирование

Саратов 2019

Разработал преподаватель СКМ и Э ФГБОУ ВО «СГТУ имени Гагарина Ю. А.» Нечаева Н.М. _____

Данные методические рекомендации предназначены для студентов 3 курса, специальности 09.02.07 Информационные системы и программирование

СОДЕРЖАНИЕ

Введение

Лабораторные работы, как виды учебных занятий, направлены на экспериментальное подтверждение теоретических положений и формирование учебных и профессиональных практических умений и составляют важную часть теоретической и профессиональной практической подготовки. Семинар является видом лабораторных работ.

В процессе лабораторной работы обучающиеся выполняют одно или несколько лабораторных работ под руководством преподавателя в соответствии с изучаемым содержанием учебного материала.

Выполнение обучающимися лабораторных работ проводится с целью:

- формирования практических умений в соответствии с требованиями к уровню подготовки обучающихся, установленными рабочей программой дисциплины по конкретным разделам/ темам дисциплины;
- обобщения, систематизации, углубления, закрепления полученных теоретических знаний;
- совершенствования умений применять полученные знания на практике, реализации единства интеллектуальной и практической деятельности;
- развития интеллектуальных умений у будущих специалистов: аналитических, проектировочных, конструктивных и др.;
- выработки таких профессионально значимых качеств, как самостоятельность, ответственность, точность, творческая инициатива при решении поставленных задач при освоении общих компетенций.

Цель практических занятий

В результате освоения учебной дисциплины «МДК 01.04. Системное программирование» обучающийся должен **уметь:**

- Создавать программу по разработанному алгоритму как отдельный модуль;
- Оформлять документацию на программные средства.
- Осуществлять разработку кода программного модуля на языках низкого уровня и высокого уровней в том числе для мобильных платформ.
- Выполнять отладку и тестирование программы на уровне модуля.
- Оформлять документацию на программные средства.
- Применять инструментальные средства отладки программного обеспечения.

В результате освоения учебной дисциплины обучающийся должен **знать:**

- Основные этапы разработки программного обеспечения.
- Основные принципы технологии структурного и объектно-ориентированного программирования.
- Знание API современных мобильных операционных систем.
- Основные принципы отладки и тестирования программных продуктов.
- Инструментарий отладки программных продуктов.

иметь практический опыт в:

- Разрабатывать код программного продукта на основе готовой спецификации на уровне модуля.
- Разрабатывать мобильные приложения.
- Использовать инструментальные средства на этапе отладки программного продукта.
- Проводить тестирование программного модуля по определенному сценарию

ПК 1.2 Разрабатывать программные модули в соответствии с техническим заданием

ПК 1.3Выполнять отладку программных модулей с использованием специализированных программных средств

Организация лабораторных работ

Лабораторные работы должны проводиться в учебных кабинетах или специально оборудованных помещениях (площадках, полигонах и т.п.)- лаборатория Системного и прикладного программирования, и кабинета Стандартизации и сертификации, а также полигона вычислительной техники и учебных баз практик. В соответствии с требованиями ФГОС СПО 09.02.07Информационные системы и программирование реализация ППССЗ должна обеспечивать выполнение обучающимися и практических занятий, включая как обязательный компонент *лабораторные работы с использованием персональных компьютеров и лицензионного программного обеспечения: ОС Windows XP; прикладное ПО.*

Выполнению лабораторных работ предшествует проверка знаний обучающихся - их теоретической готовности к выполнению задания.

Лабораторные работы могут носить репродуктивный, частично-поисковый и поисковый характер.

Работы, носящие *репродуктивный характер*, отличаются тем, что при их проведении обучающиеся пользуются подробными инструкциями, в которых указаны: цель работы, пояснения (теория, основные характеристики), оборудование, аппаратура, материалы и их характеристики, порядок выполнения работы, таблицы, выводы (без формулировки), контрольные вопросы, учебная и специальная литература.

Работы, носящие *частично-поисковый характер*, отличаются тем, что при их проведении обучающиеся не пользуются подробными инструкциями, им не дан порядок выполнения необходимых действий, и они требуют от обучающихся самостоятельного подбора оборудования, выбора способов выполнения работы в инструктивной и справочной литературе и др.

Работы, носящие *поисковый характер*, характеризуются тем, что обучающиеся, опираясь на имеющиеся у них теоретические знания, должны решить новую для них проблему.

При планировании практических занятий необходимо находить оптимальное соотношение репродуктивных, частично-поисковых и поисковых работ, чтобы обеспечить высокий уровень интеллектуальной деятельности.

Формы организации обучающихся при проведении практических занятий - фронтальная, групповая и индивидуальная.

При *фронтальной форме* организации занятий все обучающиеся выполняют одновременно одну и ту же работу.

При *групповой форме* организации занятий одна и та же работа выполняется бригадами по 2 - 5 человек.

При *индивидуальной форме* организации занятий каждый обучающийся выполняет индивидуальное задание.

Для повышения эффективности проведения практических занятий рекомендуется:

– разработка сборников задач, заданий и упражнений;

- разработка контрольно-диагностических материалов для контроля за подготовленностью обучающихся к лабораторным работам, в том числе в форме педагогических тестовых материалов для автоматизированного контроля;
- подчинение методики проведения лабораторных работ ведущим дидактическим целям с соответствующими установками обучающимся;
- применение коллективных и групповых форм работы, максимальное использование индивидуальных форм с целью повышения ответственности каждого обучающегося за самостоятельное выполнение полного объема работ;
- проведение лабораторных работ на повышенном уровне трудности с включением в них заданий, связанных с выбором обучающимися условий выполнения работы, конкретизацией целей, самостоятельным отбором необходимого оборудования;
- подбор дополнительных задач и заданий для обучающихся, работающих в более быстром темпе, для эффективного использования времени, отводимого на лабораторные работы.

Структура лабораторной работы

Лабораторная работа № 1
Тема: Использование программы DOS DEBUG

Цели и задачи урока:

ознакомиться с программой разработки и отладки программ на языке Ассемблера, а также изучить основные команды отладчика

Тип урока – практическое занятие.

Методы обучения – репродуктивный

Форма: фронтальная

Ход урока:

1. Изучить теоретическую часть
2. Выполнить задание в соответствии с указаниями
3. Ответить на контрольные вопросы
4. Предъявить преподавателю результаты работы: проект и исходный код
5. Оформить отчет в соответствии с ходом работы

Литература

1. Разработка модулей программного обеспечения для компьютерных систем: учебник для студ. учреждений сред. проф. образования / Г. Н. Федорова. - 4-е изд., перераб. - М.: Издательский центр «Академия», 2020. - 384 с. ISBN 978-5-4468-9443-7
2. http://www.uhlib.ru/kompyutery_i_internet/sistemnoe_programmirovanie_v_srede_windows/index.php (дата обращения: 24. 05. 2021г.)

Теоретическая часть:

DEBUG – это системная программа, позволяющая выполнять просмотр и изменение состояний процессора и памяти компьютера, побайтное тестирование и побайтную обработку дисковых файлов, что обеспечивает возможность выполнения отлаживаемых программ небольшими порциями. При этом программа выполняется под "наблюдением" отладчика.

Таким образом, основное назначение этой программы – отладка программ на уровне машинных кодов и языка ассемблера. Однако возможности, предоставляемые этой программой, делают ее удобным инструментом для изучения организации персональных компьютеров.

Запуск отладчика

Чтобы запустить отладчик, в командной строке Windows (Меню Пуск – Выполнить) вводится команда debug (см. рис. 1).

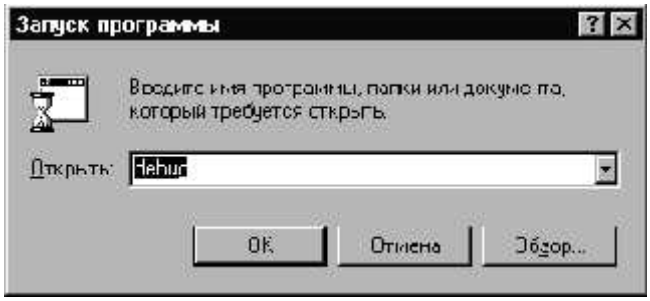


Рис. 1. Диалог "Запуск программы debug"

После нажатия кнопки "*OK*" диалога запуска отладчик загружается в память машины и переходит в режим ожидания ввода команды.

Команды программы DEBUG

DEBUG – это программа, работающая по принципу "команда – действие", т.е., чтобы произвести некоторую операцию отладчик должен получить соответствующую команду. В качестве сигнала о готовности принять команду, отладчик посылает на экран стандартный запрос – дефис (-).

Для управления процессом отладки в DEBUG применяется набор команд, список которых можно получить, введя команду помощи (символ "Вопрос"). Окно отладчика, в котором выведен список поддерживаемых команд, приведено на рис. 2.

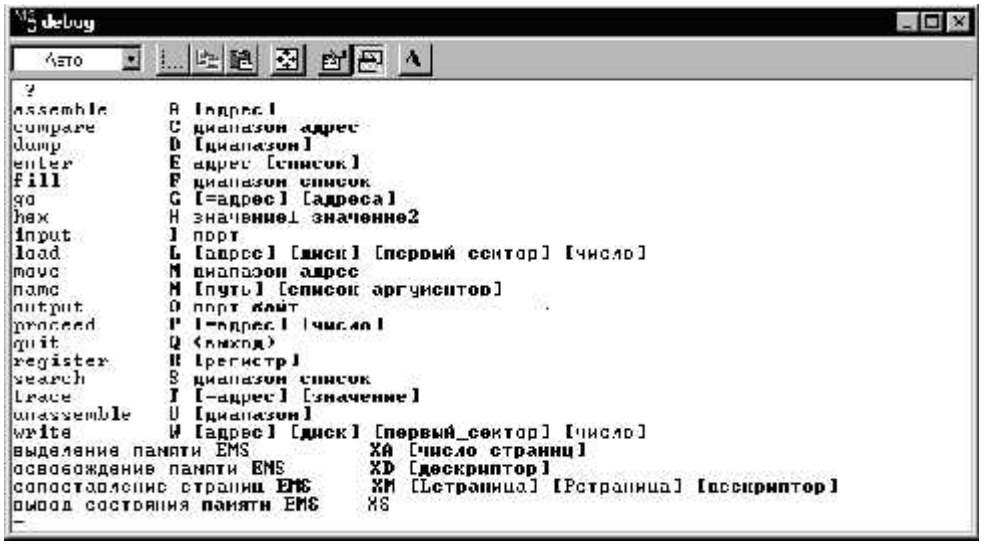


Рис. 2. Окно программы DEBUG

Общими замечаниями по вводу команд отладчика является следующее:

– все команды начинаются с буквы, заглавной или строчной;

– большая часть команд требует введения дополнительных параметров, часть из которых является необязательными;

– если два подряд расположенных параметра являются числами, то они разделяются пробелом или запятой (в противном случае параметры можно не отделять один от другого);

– все числа должны вводиться в шестнадцатеричном представлении;

– некоторые команды принимают в качестве параметра адрес, который может вводиться в двух формах: полный логический адрес – два шестнадцатеричных числа, записанные через двоеточие (первое число – сегментная компонента логического адреса, второе число – смещение) и короткий адрес – одно шестнадцатеричное число (смещение);

– при указании полного логического адреса допускается в качестве сегментной компоненты приводить имя сегментного регистра, из которого данная компонента будет выбираться.

С точки зрения изучения организации ЭВМ, набор команд, предлагаемый отладчиком, является избыточным. Минимально необходимый набор включает команды:

– (A)SSEMBLE – ассемблирование;
– (U)NASSEMBLE – дизассемблирование;
– (E)NTER – ввод данных в память;
– (D)UMP – вывод содержимого участка памяти на экран;
– (R)EGISTER – просмотр и изменение содержимого регистров;
– (T)RACE – пошаговое выполнение программы;

– (N)AME – задание имени файла программы;
– (L)OAD – загрузки файла в память;
– (W)RITE – запись области памяти в файл;
– (Q)UIT – выход из отладчика.

Перечисленные команды представляют для нас наибольший интерес. Рассмотрим их подробнее.

Команда ассемблирования (перевод мнемокода ассемблера в машинный код)

Отладчик DEBUG можно использовать для введения операторов ассемблера непосредственно в память машины. *Команду ASSEMBLE* можно использовать при составлении коротких программ на ассемблере, а также при внесении изменений в существующие программы. Эта команда позволяет вводить мнемокод ассемблера непосредственно в память, избавляя от необходимости транслировать (ассемблировать) программу. Вводимый текст не может включать метки перехода в чистом виде.

При введении команды, необходимо набрать "*a*" или "*A*" и, через пробел, необязательный параметр – адрес первой команды загружаемой программы. Если указан короткий адрес, то адрес сегмента выбирается из регистра *CS*. Если адрес не задан вообще, то машинный код будет помещаться в память, начиная с того места, где закончилась обработка предыдущей командой ASSEMBLE. Если после старта отладчика команда вводится в первый раз и в командной строке отсутствует начальный адрес, то размещение машинного кода производится с адреса *CS*:0100.

После введения команды ассемблирования на экране появляется начальный адрес. Это сигнал на введение первой команды программы. Если команда введена без ошибок, на экран выдается адрес следующей команды и отладчик опять переходит в режим ожидания. В случае ошибки отладчик обозначает ее месторасположение. Если введены все команды программы, то нажимается *Enter* – команда ASSEMBLE заканчивает работу и возвращает управление отладчику.

Пример ассемблирования небольшой программы:

```
-a 0976:0100
0976:0100  MOV AL,2A
0976:0102  MOV DI,0200
0976:0105  MOV CX,001D
0976:0108  CLD
0976:0109  REPNZ STOSB
0976:010B  MOV AL,24
0976:010D  STOSB
0976:010E  PUSH ES
0976:010F  POP DS
0976:0110  MOV DX,0200
0976:0113  MOV AH,09
0976:0115  INT 21
0976:0117  INT 20
0976:0119        <---- Нажимается Enter
-
```

Команда дизассемблирования

(перевод машинного кода в мнемокод ассемблера)

Команда UNASSEMBLE служит для перевода машинного кода на язык ассемблера. При введении команды необходимо набрать "u" или "U" и, через пробел, необязательные параметры – начальный адрес обрабатываемого кода, конечный адрес обрабатываемого кода или его размер.

В командной строке UNASSEMBLE можно не указывать начальный адрес обрабатываемого кода. Если указан короткий адрес, то адрес сегмента выбирается из регистра CS. Если адрес не задан вообще, то машинный код обрабатывается с того места, где закончилась обработка предыдущей командой UNASSEMBLE. Если после старта отладчика команда вводится в первый раз и в командной строке отсутствует начальный адрес, то обработка машинного кода производится с адреса CS:0100.

Обрабатываемый участок памяти можно определить начальным и конечным адресами. При этом, в не зависимости от формы начального адреса, конечный адрес должен быть коротким.

Другой вариант задания обрабатываемого участка памяти – задание его начального адреса и размера. Чтобы отличить размер от короткого конечного адреса перед ним вводится символ "L".

Если размер участка памяти, обрабатываемой командой UNASSEMBLE, не определен, то по умолчанию длина обрабатываемого участка равна 32 байтам.

Результатом выполнения команды дизассемблирования является листинг программы, сгруппированный в три колонки. В листинге слева (первая колонка) указывается полный логический адрес команды. Затем (вторая колонка) – значение составляющих команду байтов в машинном коде. В третьей колонке находится соответствующая этому коду инструкция ассемблера.

Пример дизассемблирования небольшой программы, введенной в предыдущем примере:

```
-u CS:0100 L19
0976:0100      B02A        MOV AL,2A
0976:0102      BF0002          MOV DI,0200
0976:0105      B91D00      MOV CX,001D
0976:0108      FC          CLD
0976:0109      F2          REPNZ
0976:010A      AA          STOSB
0976:010B      B024        MOV AL,24
0976:010D      AA          STOSB
0976:010E      06          PUSH ES
0976:010F      1F          POP DS
0976:0110      BA0002          MOV DX,0200
0976:0113      B409        MOV AH,09
0976:0115      CD21        INT 21
0976:0117      CD20        INT 20
```

Команда ввода данных в память

Ввод данных осуществляется с помощью команды ENTER. Эта команда позволяет побайтно корректировать содержимое памяти. Команда состоит из буквы e (или E) и адреса первого байта корректируемого блока. Если указан короткий адрес, то адрес сегмента выбирается в регистре DS.

Вводимые данные также включаются в командную строку. Они представляют собой последовательность чисел в шестнадцатеричном представлении и/или символьных значений, разделенных пробелом или запятой. Символьные значения заключаются в апострофы.
Проиллюстрируем работу ENTER на следующем примере:
-e DS:0000 20 2A 44 41 54 41 20 'IS' 20 48 45 52 45 2A 20
Команда вводит 16 значений. Данные последовательно заполняют память (побайтно), начиная с адреса DS:0000. Четырнадцать байтов занимают числа в шестнадцатеричном формате, два байта отводятся под символьную константу 'IS'.
Команда ENTER может использоваться для отображения и, в случае необходимости, корректировки значения конкретного байта. В этом случае команда состоит из буквы e (или E) и следующего за ней адреса. При введении команды на экране появляется адрес байта и его значение:
-e DS:0000
0958:0000 20.
При нажатии на клавишу пробела на экране появляется значение следующего байта:
-e DS:0000
0958:0000 20. 2A.
Значение байта можно изменить. Для этого вводится новое шестнадцатеричное число. Однако символьные переменные в этом случае вводить нельзя.
Чтобы завершить выполнение команды, нажимается Enter. Появление дефиса (-) – стандартного запроса отладчика, свидетельствует о его готовности принять следующую команду.
Команда вывода содержимого участка памяти на экран
Команда DUMP (d или D) служит для отображения на экране содержимого участка памяти. Полученный кусочек памяти – дамп, представляет собой последовательность значений байтов в шестнадцатеричном представлении, а также – в коде ASCII:
-d
0958:0100 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ....
0958:0110 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ....
0958:0120 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ....
0958:0130 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ....
0958:0140 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ....
0958:0150 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ....
0958:0160 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ....
0958:0170 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ....
└────────┘ └──┘ └────────┘
Адрес Значения в шестнадцатеричном виде, в виде символов
Числа в первом столбце, разделенные двоеточием – это полный логический адрес памяти, следующие 16 столбцов – значения, последовательно содержащиеся в памяти, начиная с этого адреса (в шестнадцатеричном виде), и, наконец, строка из 16 символов – символьное представление этих значений.
Значения, не имеющие символьного представления в коде ASCII, обозначаются символом "точка". В рассмотренном примере изображены только точки, поскольку в коде ASCII не существует печатных символов со значением 00. Дамп отображает содержимое 128

последовательно расположенных байтов. В приведенном выше примере начальный адрес дампа – 0958:0100, конечный – 0958:017F.

Если вводить команду "d" не указывая параметров, DEBUG будет последовательно выводить по 128 байтов памяти. То есть начальный адрес дампа будет на единицу превышать конечный адрес дампа, полученного при введении предыдущей команды "d". Если команда "d" вводится первоначально, то дамп выводится, начиная с адреса, по которому был загружен обрабатываемый файл.

Команда DUMP может использоваться с параметрами, которые задают начальный адрес дампа и его размер. Использование параметров аналогично использованию параметров в команде UNASSEMBLE. За тем исключением, что, если начальный адрес дампа задан в коротком формате, то сегментная компонента адреса выбирается из регистра DS.

Команда просмотра и изменения содержимого регистров

Команда REGISTER (r или R) выводит на экран и корректирует значения регистров и флагов состояния процессора. Эта команда также выдает информацию о следующей выполняемой команде:

```
-r
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0958 ES=0958 SS=0958 CS=0958 IP=0100    NV UP DI PL NZ NA PO NC
0958:0100  0000        ADD           [BX+SI],AL           DS:0000=CD
-
```

С помощью "r" можно изменить значение регистра. В этом случае в командной строке указывается его имя. Значение регистра выводится на экран. Теперь можно вводить новое число. Чтобы сохранить старое значение регистра, нажмите Enter.

```
-r CX
CX 0000
:245D
-r
AX=0000 BX=0000 CX=245D DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0958 ES=0958 SS=0958 CS=0958 IP=0100    NV UP DI PL NZ NA PO NC
0958:0100  0000        ADD           [BX+SI],AL           DS:0000=CD
-
```

Команда "rf" выводит на экран флаги состояния процессора. Получив значения флагов, их можно изменить. Для этого вводится одно или несколько новых значений.

Мнемонические обозначения состояний флагов приведены в табл. 1.

Таблица 1

Мнемонические обозначения состояний флагов

Флаг	Установлен	Сброшен
Переполнение (есть/нет)	OV	NV
Направления (увеличение/уменьшение)	DN	UP
Прерывания (разрешение/запрещение)	EI	DI
Знака (минус/плюс)	NG	PL
Нуля (да/нет)	ZR	NZ
Дополнительного переноса (да/нет)	AC	NA
Паритета (чет/нечет)	PE	PO

Переноса (да/нет)	CY	NC

Символьные значения вводятся в любом порядке через пробел или вообще без разделителя. Установим, например, значения флагов переполнения, знака и переноса:

```
-rf
      NV UP DI PL NZ NA PO NC   -OV  NG  CY <- Подчеркнутое вводит пользователь
-r
AX=0000 BX=0000 CX=245D DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0958 ES=0958 SS=0958 CS=0958 IP=0100    OV UP DI NG NZ NA PO CY
0958:0100  CD20        INT       20
-
```

Команда пошагового выполнения программы

Команда TRACE (*t* или *T*) – трассировка осуществляет пошаговое выполнение программы в машинном коде. При трассировке после выполнения каждой команды производится останов работы программы и на экран выводятся регистры и флаги состояния процессора. Полученная картинка аналогична картинке, получаемой с помощью команды REGISTER. Разница заключается только в том, что при введении TRACE перед появлением картинки, выполняется одна команда отлаживаемой программы.

Проиллюстрируем работу TRACE на примере нашей программы. Если она не загружена в память, то запустим DEBUG и введем:

```
-e CS:0100 B0 2A BF 00 02 B9 1D 00 FC F2 AA B0 24
-e CS:010D AA 06 1F BA 00 02 B4 09 CD 21 CD 20
-
```

Чтобы узнать адрес программы, введем команду REGISTER:

```
-r
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0976 ES=0976 SS=0976 CS=0976 IP=0100    NV UP DI PL NZ NA PO NC
0976:0100  B001        MOV           AL,2A
-
```

При введении "*t*" выполняется команда по адресу CS:IP. После этого на экран выводятся регистры и флаги состояния:

```
-t
AX=002A BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0976 ES=0976 SS=0976 CS=0976 IP=0102    NV UP DI PL NZ NA PO NC
0976:0102  BF0002      MOV           DI,0200
-t
AX=002A BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0200
DS=0976 ES=0976 SS=0976 CS=0976 IP=0105    NV UP DI PL NZ NA PO NC
0976:0105  B91D00      MOV           CX,001D
-
```

В командной строке TRACE можно указать адрес выполняемой команды. В этом случае после *t* набирается знак равенства (=) и нужный адрес. Если указан короткий адрес, то адрес сегмента выбирается из регистра *CS*:

```
-t=0100
AX=002A BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0200
```

```
DS=0976 ES=0976 SS=0976 CS=0976 IP=0102    NV UP DI PL NZ NA PO NC
0976:0102  BF0002      MOV           DI,0200
-
```

В этом случае выполнена команда по адресу CS:0100. Адрес следующей команды находится в регистрах *CS*:*IP*. Он равен 0976:0102.

Одной командой TRACE можно одновременно трассировать несколько команд отлаживаемой программы. Для этого при введении "*t*" просто указывается их количество. После выполнения каждой команды на экране появляется картинка с содержимым регистров и флагов состояния. При заполнении экрана новые данные выводятся в нижней его части, сдвигая данные в верхней части за пределы экрана. Чтобы остановить движение данных вдоль экрана, нажимаются клавиши *Ctrl-NumLock*. Чтобы возобновить движение, нажимается любая клавиша.

При нажатии *Ctrl-C* трассирование прекращается и на экране появляется стандартный запрос отладчика.

Пример:

```
-t6
AX=002A BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0200
DS=0976 ES=0976 SS=0976 CS=0958 IP=0105    NV UP DI PL NZ NA PO NC
0976:0105  B91D00      MOV       CX,001D

AX=002A BX=0000 CX=001D DX=0000 SP=FFEE BP=0000 SI=0000 DI=0200
DS=0976  ES=0976 SS=0976 CS=0958 IP=0108    NV UP DI PL NZ NA PO NC
0976:0108  FC          CLD

AX=002A BX=0000 CX=001D DX=0000 SP=FFEE BP=0000 SI=0000 DI=0200
DS=0976 ES=0976 SS=0976 CS=0958 IP=0109    NV UP DI PL NZ NA PO NC
0976:0109  F2          REPNZ
0976:010A  AA          STOSB

AX=002A BX=0000 CX=001C DX=0000 SP=FFEE BP=0000 SI=0000 DI=0201
DS=0976 ES=0976 SS=0976 CS=0958 IP=0109    NV UP DI PL NZ NA PO NC
0976:0109  F2          REPNZ
0976:010A  AA          STOSB

AX=002A BX=0000 CX=001B DX=0000 SP=FFEE BP=0000 SI=0000 DI=0202
DS=0976 ES=0976 SS=0976 CS=0958 IP=0109    NV UP DI PL NZ NA PO NC
0976:0109  F2          REPNZ
0976:010A  AA          STOSB

AX=002A BX=0000 CX=001A DX=0000 SP=FFEE BP=0000 SI=0000 DI=0203
DS=0976 ES=0976 SS=0976 CS=0958 IP=0109    NV UP DI PL NZ NA PO NC
0976:0109  F2          REPNZ
0976:010A  AA          STOSB
-
```

Команда задания имени файла программы

Команда NAME (*n* или *N*) присваивает имя обрабатываемому файлу. Затем этот файл загружается в память командой LOAD или записывается на диск командой WRITE. (LOAD и WRITE рассматриваются ниже.)

Чтобы идентифицировать файл, наберите "*n*" и, через пробел – имя файла. Воспользуемся NAME, чтобы присвоить нашей программе имя "mytest.pro":

```
-n mytest.pro
```

Команда загрузки файла в память

Загрузка файла в память осуществляется, если в командной строке DEBUG указать имя файла. Другой способ – использование *команды LOAD* (*l* или *L*).

При использовании команды LOAD необходимо специфицировать файл с помощью команды NAME.

В командной строке LOAD можно указать начальный адрес, по которому загружается файл. Если указан короткий адрес, то адрес сегмента выбирается из регистра *CS*. При отсутствии начального адреса, загрузка производится по адресу *CS*:0100.

После загрузки отладчик запоминает количество занятой файлом памяти (в байтах) в регистрах *BX* (старшее слово) и *CX* (младшее слово).

К примеру, загрузим в память файл "mytest.pro" по адресу *CS*:0100:

```
-n mytest.pro
-L
-r
AX=0000 BX=0000 CX=00CF DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0958 ES=0958 SS=0958 CS=0958 IP=0100    NV UP DI PL NZ NA PO NC
0958:0100  2A2A        SUB           CH,[BP+SI]           SS:0000=CD
-
```

В регистрах *BX* и *CX* находится значение 207 (000000CF). Это значит, что файл занял 207 байт. Тот же результат можно получить при введении спецификации файла в командной строке команды старта отладчика ("debug mytest.pro").

Команда записи области памяти в файл

Команда WRITE (*w* или *W*) переписывает на диск данные, выбирая их из памяти. При этом спецификация создаваемого файла должна задаваться с помощью команды NAME.

Перед введением команды WRITE в регистры *BX* и *CX* записывается размер занимаемой файлом памяти в байтах (шестнадцатеричное число, занимающее 4 байта). Поэтому перед записью необходимо проверить содержимое этих регистров (с помощью REGISTER).

В командной строке WRITE можно указать начальный адрес памяти, по которому производится чтение данных с последующей записью их на диск. Если указан короткий адрес, то адрес сегмента выбирается из регистра *CS*.

Если начальный адрес не указан, то запись производится, начиная с адреса *CS*:0100.

Команда выхода из отладчика

Чтобы выйти из отладчика и передать управление операционной системе, на его стандартный запрос вводится команда *q*:

```
-q
```

Задание 1. Изучить основные команды, выполнив все примеры (синего цвета).

Задание 2. В соответствии со своим вариантом выполнить задание в debug и проверить результат в ячейках.

1. *MOV BP,200*
2. *MOV BP,30*
3. *MOV DI,AX*
4. *MOV BP,AX*
5. *MOV SI,–20*
6. *MOV BX,AX*
7. *MOV DI,18*
8. *MOV SI,AX*
9. *MOV SI,40*
10. *MOV AX,120*

Контрольные вопросы:

1. Назначение программы DEBUG?
2. Что означает дизассемблирование?
3. Какие режимы работы процессора вы знаете?
4. Как записывается команда ассемблирования?
5. Какая команда служит для отображения на экране содержимого участка памяти?
6. Какая команда переписывает на диск данные, выбирая их из памяти?

Лабораторная работа № 2

Тема: Использование TASM для трансляции и компоновки программ

Цели и задачи урока:
познакомиться с приёмами работы с отладчиком ассемблера на примере простых тестовых программ типа. exe и. com
Тип урока – практическое занятие.
Методы обучения – репродуктивный
Форма: фронтальная
Обеспечение занятия- персональный компьютер с отладчиком ассемблера help DOS.
Ход урока
Задание 1. Изучить среды разработки
Задание 2. Написание первой программы
Задание 3. Изучение сегментных регистров: CS, DS, SS и ES и регистры общего назначения: AX, BX, CX и DX
Литература

1. Разработка модулей программного обеспечения для компьютерных систем: учебник для студ. учреждений сред. проф. образования / Г. Н. Федорова. - 4-е изд., перераб. - М.: Издательский центр «Академия», 2020. - 384 с. ISBN 978-5-4468-9443-7
2. http://www.uhlib.ru/kompyutery_i_internet/sistemnoe_programmirovanie_v_srede_windows/index.php (дата обращения: 24. 05. 2021г.)

Задание

I. Изучение среды работы

Создадим простую программу. У TASM нет среды разработки, поэтому придется писать bat файлы (рисунок 1):

Рисунок 1.

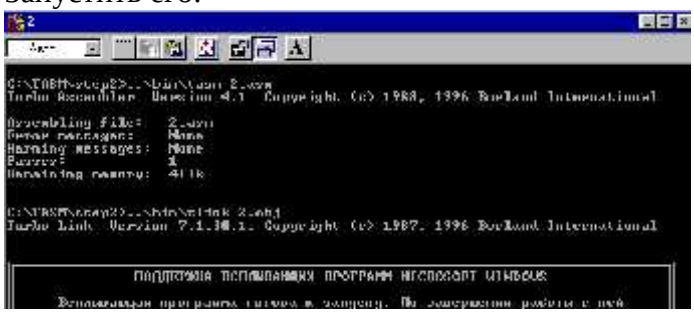
Создаем папки в каталоге TASM и исходя из этого настраиваем пути в BAT файлах. Программировать мы будем в NotePad. Исходный файл- это обычный текстовый файл с командами ассемблера. Расширение у него должно быть ASM. Итак, создаем программу с именем 2.asm. Внутри код:

```
MODEL       TINY
STACK 256
CODESEG
start:
        mov ah, 04Ch
        int 21h
end     start
```

Нужно научиться этот код собирать. Код собирается в два этапа. Первый этап компиляция и это делает TASM.EXE. В результате компиляции мы получаем OBJ файл. Второй этап — это сборка это выполняет файл TLINK.EXE который собирает исполняемый файл. Нам нужно написать BAT файл с именем 2.bat:

```
..\bin\tasm 2.asm
..\bin\tlink 2.obj
```

Запустить его:



II. Написание первой программы

Пишем код.

```
MODEL    TINY
STACK 256
DATASEG
  Hellostr DB 'Hello First Step Site '
CODESEG
start:
  mov ax,@data
  mov ds,ax
  mov bx,1
```

```
    mov cx,21
    mov dx,offset Hellostr
    mov ah,40h
    int 21h
    mov ah, 04Ch
    int 21h
end start
```

Собираем, все должно быть без ошибок, и запускаем:

Обратите внимание на **DATASEG** и **CODESEG**. В программе есть место, где хранятся данные и где хранится код. И эти места нужно разделять. Директива **DATASEG** указывает на то что далее будут идти данные а директива **CODESEG** что теперь начнутся команды процессора. Когда программа загружается в память, то операционной системе нужно знать, куда поставить указатель для выполнения команды. Именно директива **CODESEG** и указывает при сборке, где это место будет находиться. То же самое и для **DATASEG**. Если Вы откроете программу в блокноте, то увидите, что данные как раз находятся в самом конце программы.

Раньше мы использовали команду **INT**:

```
mov ah,40h
int 21h
mov ah, 04Ch
int 21h
```

Int — это сокращение слова **Interrupt,** что на русский переводиться как прерывание. Выглядит это команда так:

```
Int номер
```

То есть, у каждого прерывания есть номер. Итак, прерывания бывают двух типов:

- Программные
- Аппаратные

Устройств всяких много - клавиатура, монитор, дисковод и так далее. Если не пользоваться прерываниями, то постоянно операционная система должна опрашивать устройства, нажата ли клавиша, хотите ли вывести данные на монитор и так далее. Намного проще оговорить некоторый механизм, который и будет обращать внимание операционной системы и процессора на необходимость проведения некоторых действий. Итак, давайте посмотрим на все это в динамике. Ваша программа выполняется. В этот момент нажимается клавиша. Программа должны быть прервана. И это будет сделано, управление будет передано специальному коду (процедура обработки прерывания) а потом Ваша программа будет выполняться дальше. То же самое, когда мы вызываем прерывание для вывода символов на монитор (**int 21h 04Ch**) то сами генерируем прерывание. Зачем? В этот момент может происходить считывание с дисковода или вывод других символов на экран. А тут мы не с того не с сего со своими символами. Так как прерывания могут наступать

одновременно, то есть приоритет их обработки. Есть прерывание, которые выполняться в любом случае, даже если идет обработка другого прерывания. Процедура обработки прерывания — это программа. Вопрос в том только где она храниться. Базовая обработка прерывания храниться в **BIOS** и в самих микросхемах оборудования. Но использовать их довольно тяжело.

III. Изучение регистров

Регистры — это специальные ячейки памяти. Обращение к регистрам производиться значительно быстрее, чем к оперативной памяти **ПК**. Именно по этой причине регистры используются для команд процессора. Они бывают следующие по типам.

регистры общего назначения	AX, BX, CX, DX, BP, SI, DI, SP
сегментные регистры	CS, DS, SS, ES
счетчик команд	IP
регистр флагов	Flags

Каждое имя регистра несет некоторый смысл

A	accumulator	аккумулятор
B	base	база
C	counter	счетчик
D	data	данные
BP	base pointer	указатель базы
SI	source index	индекс источника
DI	destination index	индекс приемника
SP	stack pointer	указатель стека
CS	code segment	сегмент команд
DS	data segment	сегмент данных
SS	stack segment	сегмент стека
ES	extra segment	дополнительный сегмент
IP	instruction pointer	счетчик команд

Регистры **AX, BX, CX** и **DX** позволяют нам обращаться не к регистру а к старшему и младшему байту

AX	AH,AL
BX	BH,DL
DX	DH,DL
CX	CH,CL

На данный момент мы использовали регистр **AX**. Для задания функции прерывания.

```
mov ah,40h
int 21h
mov ah, 04Ch
int 21h
```

При этом использовали только часть регистра, а точнее старший байт:

H high старший
L low младший

Да потому что есть правила, где что должно храниться при вызове прерывания. Точнее, что и в каком регистре должно находиться. Правила, эти описаны в документации. Ну, например наша последняя функция описана так

```
Int 21H Функция 4CH
AH=4CH
AL=код возврата
Возврата нет.
```

Прекращает процесс и передает операционной системе код возврата.

На данный момент Вы должны понимать, что для большинства операций используются регистры, а для прерываний конкретно есть спецификации, в которых написано, что и в каком регистре должно находится.

Лабораторная работа № 3

Тема: Использование арифметических операций на языке ассемблера

Цели и задачи урока:
изучить арифметические операции языка ассемблера; научиться их использовать при составлении программ; научиться применять битовые команды
Тип урока – практическое занятие.
Методы обучения – репродуктивный
Форма: фронтальная
Обеспечение занятия- персональный компьютер с отладчиком ассемблера help DOS.
Ход урока

1. Изучить теоретическую часть
2. Выполнить задание в соответствии с указаниями
3. Ответить на контрольные вопросы
4. Предъявить преподавателю результаты работы: проект и исходный код
5. Оформить отчет в соответствии с ходом работы

Литература

1. Разработка модулей программного обеспечения для компьютерных систем: учебник для студ. учреждений сред. проф. образования / Г. Н. Федорова. - 4-е изд., перераб. - М.: Издательский центр «Академия», 2020. - 384 с. ISBN 978-5-4468-9443-7
2. http://www.uhlib.ru/kompyutery_i_internet/sistemnoe_programmirovanie_v_srede_windows/index.php (дата обращения: 24. 05. 2021г.)

Теоретическая часть:

Арифметические операции на языке ассемблера выполняются над целыми числами четырех типов:

Беззнаковыми двоичными, знаковыми двоичными, упакованными десятичными и неупакованными десятичными.

В данной лабораторной работе рассматриваются арифметические операции над беззнаковыми числами.

Используются следующие команды:

ADD – сложить, SUB – вычесть.

Описание команд: работают с 8 и 16 битовыми операндами, инструкция ADD выполняет сложение операнда источника (правого операнда) с содержимым операнда приемника (левый операнд), результат помещается в операнд приемник. Инструкция SUB делает тоже самое, только она вычитает операнд источник из операнда приемника, результат помещается в операнд приемник. Операндами могут быть регистры, константы, ячейки памяти в различных комбинациях, но нельзя добавить (вычесть) значение одной ячейки памяти к другой, а также в качестве операнда источника использовать константу (непосредственное значение). Команда воздействует на шесть флагов: AF, CF, OF, PF, SF, ZF.

Например, флаг переноса CF=1 если результат действия не помещается в операнде приемнике, в противном случае CF=0.

MUL - умножить.

Описание команд: инструкция MUL перемножает 8 и 16 битовые беззнаковые множители, создавая 16 или 32 битовое произведение. При 8 битовом произведении один из операндов в регистре AL другой может быть 8 битовым регистром общего назначения или переменной памяти соответствующего назначения. Результат помещается в регистр AX (16 битовый). При 16 битовых множителях один из сомножителей в 16 битовом регистре общего назначения

другой в переменной памяти, 32 битовый результат в регистрах DX:AX. При этом младшие 16 бит в AX старшие в DX.

Команда воздействует на два флага: CF, OF.

DIV – разделить.

Описание команд: позволяет разделить 32 битовое значение на 16 битовое значение или 16 битовое на 8 битовое. При делении 16 битового значения делимое помещается в AX, 8 битовый делитель помещается в регистр или в переменную соответствующего размера. Результат (8 битовый) помещается в AL, остаток в AH.

Состояние флагов не определено, но если частное не помещается в регистре AL (AX) процессор генерирует прерывание типа 0 (деление на 0). В заданиях используются директивы и команды, изученные в предыдущих лабораторных работах.

БИТОВЫЕ КОМАНДЫ

Битовые команды рассматривают свои операнды не в виде привычных уже байтов, слов и двойных слов, в виде последовательности битов. Эти команды реализуют логические операции и команды сдвигов.

Логические операции (или булевы команды) – как, следует из названия, выполняют логические операции – отрицание, конъюнкцию, дизъюнкцию и им присуще ряд черт.

Инвертировать	`not opr`
"И" (конъюнкция)	`and dst,src`
Логическое сравнение	`test opr1,opr2`
"ИЛИ" (дизъюнкция)	`or dst,src`
"Исключающее ИЛИ"	`xor dst,src`

Команда **not** на флаги не действует и работает по следующему принципу. Например:

```
mov   al, 1100b     ;al=00001100b
not   al            ;al=11110011b
```

Все остальные команды сбрасывают CF и OF, а флаги SF, ZF, PF изменяют по обычным правилам.

Команда **and** производит поразрядное логическое умножение операндов и записывает результат на место первого операнда. Например:

```
mov   al, 1100b     ;al=00001100b
and   al, 1010b     ;al=00001000b
```

Команда **test**, аналог предыдущей команды, но результат логического умножения никуда не записывается, основное назначение – установка флагов, особенно флаг нуля ZF.

```
mov   bh, 1100b
test  bh, 0011b     ;al=00000000b → ZF=1
test  bh, 1010b     ;al=00001100b → ZF=0
```

Команда **or** производит поразрядное логическое сложение операндов и записывает результат на место первого операнда. Например:

```
mov   al, 1100b     ;al=00001100b
or    al, 1010b     ;al=00001110b
```

Команда **xor** производит поразрядное логическое сложение операндов и записывает результат на место первого операнда. Данная операция соответствует фразе «или то, или другое, но не то и не другое», т.е. если биты совпадают, записывается 0, иначе 1.

```
mov   cl, 1100b
xor   cl, 1010b     ;al=00000110b
xor   cl, cl        ;cl=00000000b
```

Команды сдвига – эти команды перемещают содержимое ячейки влево или вправо. Одним из операндов этих команд является количество сдвигов cnt. Оно либо равно 1, либо определяется содержимым регистра CL (при этом CL сохраняет своё содержимое после операции).

Логические сдвиги – команды сдвига, где участвуют все биты первого операнда, при этом бит, уходящий за пределы ячейки. Заносится в флаг CA, а с другого конца в операнд добавляется ноль.

```
Логический сдвиг влево (shift left):    SHL

Логический сдвиг вправо (shift right):  SHR
   Например:
   mov  al, 01000111b
   shl  al,1              ;CF=0, al=10001110b
   mov  al, 01000111b
   shr  al,1              ;CF=1, al=00100011b
   mov  bh, 0011100b
   mov  cl,3
   shl  bh,cl             ;CF=1, al=11000000b
```

Арифметические сдвиги – предназначены для реализации быстрого умножения и деления знаковых чисел на степени двойки.

```
Арифметический сдвиг влево (shift arithmetic left):   SAL
Арифметический сдвиг вправо (shift arithmetic right): SAR
   Например:
   mov  ah,10001110b
   sar  ah,1              ;CF=0, al=11000111b
   mov  ah,10001110b
   sal  ah,1              ;CF=1, al=0011100b
```

Примечание: команда sal при трансляции будет воспринята как shl, так как это разные мнемонические названия одной и той машинной команды.

Циклические сдвиги. Особенность циклических сдвигов в том, что «уходящий» бит не теряется, а возвращается в операнд, но с другого конца.

```
Циклический сдвиг влево (shift arithmetic left):      ROL
Циклический сдвиг вправо (shift arithmetic right):    ROR
   Например:
   mov  ah,11000011b
   rol  ah,1              ;CF=1, al=10000111b
   mov  ah,11100010b
   ror  ah,1              ;CF=0, al=01110001b
```

Задание 1. Задача заключается в вычислении результата выполнения арифметического выражения, в котором некоторые числа постоянны, а другие переменные.

Формула вычислений: X=(A*2+B*C)/(D-3)

Приведенная программа сначала резервирует ячейки памяти под переменные, затем выполняет умножение однобайтовых чисел (A*2), результат умножения – двухбайтовое число в регистре AX, сохраняется в регистре CX, далее выполняется умножение однобайтовых чисел (B*C), результат – двухбайтовое число храниться в аккумуляторе AX. После сложения двух сомножителей и вычисления знаменателя (B-3) выполняется деление. Результат присваивается переменной X.

1. Наберите приведенную программу 1, запишите исходный файл с расширением *.asm, получите файл с расширением *.exe.
2. Выполните программу с 5 вариантами различных начальных значений переменных A, B, C, D по шагам (см. таблицу 1) и запишите результат выполнения в таблицу (в регистре AL – частное, AH – остаток). Переведите результат в десятичную систему.

Таблица 1

Вариант		1	2	3	4	5
Значения	A	3	0AH	20	60	20H
	B	4	5H	4	16	9H
	C	2	8H	15	5	4H
	D	5	9H	6	18	1CH
Частное AL						
Остаток AH						

```
;программа 1
;x =(a*2+b*c)/(d-3)
.model small
.stack 100h
.data
a  db   ?
b  db   ?
c  db   ?            ;Резервируем память для переменных
d  db   ?            ; A,B,C,D,X
x  dw   ?
.code
start:
mov  ax,@data
mov  ds,ax
mov  a,3
mov  b,4
mov  c,2
mov  d,5
mov  al,2
mul  a
mov  cx,ax
mov  al,b
```

```
mul   c
add   ax,cx
mov   cl,d
sub   cl,3
div   cl
mov   x,ax
mov   ah,4ch
int   21h
end   start
```

3. Наберите приведенную программу 2, запишите исходный файл с расширением *.asm, получите файл с расширением *.exe.
4. Выполните программу с 5 вариантами различных начальных значений регистра al. Переведите результат и запишите таблицу 2 в двоичной и десятичной системе.

```
;программа 2
.model tiny
.stack 100h
.code
start:
mov ah, 01001101b
shr ah,1
mov ah, 01001101b
shl ah,1
mov ah, 01001101b
sar ah,1
mov ah, 01001101b
ror ah,1
mov ah, 01001101b
rol ah,1
mov ax,4c00h
int 21h
end start
```

Команды		SHL	SHR	SAR	ROR	ROL
Значение	01001101b					
	01101010b					
	10101101b					
	11011011b					
	10101100b					

Контрольные вопросы:

1. Чем отличается выполнение арифметических операций на языке ассемблера от языков высокого уровня?
2. По какому биту регистра флагов можно установить, что предшествующее вычитание привело к отрицательному результату?

3. По какому биту регистра флагов можно установить, что результат арифметической операции превысил разрядную сетку?
4. В чем особенность выполнения арифметических операций с знаковыми и беззнаковыми числами?
5. В каком регистре необходимо указывать величину сдвига в команде сдвига?

Лабораторная работа № 4

Тема: Проверка оборудования

Цели и задачи урока:

Получение практических навыков в определении конфигурации и основных характеристик компьютера

Тип урока – практическое занятие.

Методы обучения – репродуктивный

Форма: фронтальная

Обеспечение занятия- ПК, тетради.

Ход урока

Постановка задачи

Для компьютера на своем рабочем месте определить:
тип компьютера;
конфигурацию оборудования;
объем оперативной памяти;
наличие и объем расширенной памяти;
наличие дополнительных ПЗУ;
версию операционной системы.

Литература

1. Разработка модулей программного обеспечения для компьютерных систем: учебник для студ. учреждений сред. проф. образования / Г. Н. Федорова. - 4-е изд., перераб. - М.: Издательский центр «Академия», 2020. - 384 с. ISBN 978-5-4468-9443-7
2. http://www.uhlib.ru/kompyutery_i_internet/sistemnoe_programmirovanie_v_srede_windows/index.php (дата обращения: 24. 05. 2021г.)

Пример решения задачи

Структура данных программы

Программа использует, так называемый, список оборудования — 2-байтное слово в области данных BIOS по адресу 0040:0010. Назначение разрядов списка оборудования такое:

0 установлен в 1, если есть НГМД

1 установлен в 1, если есть сопроцессор

2,3 число 16-Кбайтных блоков ОЗУ на системной плате

4,5 код видеоадаптера: 11 — MDA, 10 — CGA, 80 колонок, 01 — CGA, 40 колонок, 00 — другой

6,7 число НГМД-1 (если в разряде 0 единица)

8 9, если есть канал ПДП

9,10,11 число последовательных портов RS-232

12 1, если есть джойстик

13 1, если есть последовательный принтер

14,15 число параллельных принтеров

Структура программы

Программа состоит только из основной функции **main()**. Выделения фрагментов программы в отдельные процедуры не требуется, потому что нет таких операций, которые во время работы программы выполняются многократно.

Описание переменных

Переменные, применяемые в программе:

type_PC — байт типа компьютера, записанный в ПЗУ BIOS по адресу FF00:0FFE;

a, **b** — переменные для определения объема extended-памяти ПЭВМ, **a** — младший байт, **b** — старший байт;

konf_b — 2-байтное слово из области данных BIOS, которое содержит список оборудования;

type — массив символьных строк, представляющих типы компьютера;

typ1A — массив байт, содержащий коды типов дисплеев;

types1A[] — массив строк, содержащий названия типов дисплеев;

j — вспомогательная переменная, которая используется для идентификации типа дисплея;

seg — сегмент, в котором размещено дополнительное ПЗУ;

mark — маркер ПЗУ;

bufVGA[64] — буфер данных VGA, из которого (при наличии VGA) мы выбираем объем видеопамяти;

rr и **sr** — переменные, которые используются для задания значения регистров общего назначения и сегментных регистров, соответственно, при вызове прерывания.

Описание алгоритма программы

Алгоритм основной программы может быть разбито на 5 частей.

Часть 1 предназначена для определения типа компьютера. Для этого прочитаем байт, записанный в ПЗУ BIOS по адресу FF00:0FFE. В зависимости от значения этого байта сделаем вывод о типе ПЭВМ. Так, например, компьютеру типа AT соответствует код 0xFC.

Часть 2 предназначена для определения конфигурации ПЭВМ. Для этого прочитаем из области данных BIOS список оборудования. Для определения количества дисководов (если бит 0 установлен в 1) необходимо выделить биты 6 и 7 (маска 00C0h) и сместить их вправо на 6 разрядов, а потом добавить 1.

Для определения количества 16-Кбайтных блоков ОЗУ на системной плате необходимо выделить биты 2 и 3 с помощью маски 000Ch, сместить вправо на 2 разряды и добавить 1.

Для определения количества последовательных портов RS-232 выделить с помощью маски 0Eh биты 9-11 и сместить вправо на 9 разрядов.

Для определения наличия математического сопроцессора — проверить установку бита 1 маской 0002h.

Для определения наличия джойстика — бита 12 с помощью маски 1000h.

Определить количество параллельных принтеров можно, выделив биты 14 и 15 маской C000h и сместив их вправо на 14 разрядов.

Поскольку список оборудования содержит недостаточно информации про дисплейный адаптер, то для уточнения типа адаптера выполним дополнительные действия.

Видеоадаптер обслуживается прерыванием BIOS 10h. Для новых типов адаптеров список его функций расширяется. Эти новые функции и используются для определения типу адаптера.

Функция 1Ah доступна только при наличии расширения BIOS, ориентированного на обслуживание VGA. В этом случае функция возвращает в регистре AL код 1Ah — свою «визитную карточку», а в BL — код активного видеоадаптера. В случае, если функция 1Ah поддерживается, обратимся еще к функции 1Bh — последняя заполняет 70-байтный блок информации про состояние, из которого мы выбираемо объем видеопамяти.

Если 1Ah не поддерживается, это означает, что VGA у нас нет, в этом случае можно обратиться к функции 12h — получение информации про EGA. При наличии расширения, ориентированного на EGA, эта функция изменяет содержимое BL (перед обращением он должен быть 10h) на 0 (цветной режим) или на 1 (монохромный режим) а в BH возвращает объем видеопамяти.

Если же ни 1Ah, ни 12 не поддерживаются, то список оборудования BIOS содержит достаточную информацию про видеоадаптер и, выделивши биты 4, 5 мы можем сделать окончательный вывод про тип адаптера, который у нас есть.

В третьей части программы определим объем оперативной памяти, наличие и объем extended-памяти. Объем оперативной памяти для AT может быть прочитан из регистров 15h (младший байт) и 16h (старший байт) CMOS-памяти или из области памяти BIOS по адресу 0040:0013 (2-байтное слово). Кроме того, в ПЭВМ может быть еще и дополнительная (expanded) память свыше 1 Мбайту. Ее объем можно получит из регистров 17h (младший байт) и 18h (старший байт) CMOS-памяти. Для чтения регистра CMOS-памяти необходимо видать в порт 70h байт номера регистра, а потом из порта 71h прочитать байт содержимого этого регистра.

В следующей части программы определим наличие и объем дополнительных ПЗУ. В адресному пространстве от C000:0000 по F600:0000 размещаются расширения ПЗУ (эта память не обязательно присутствует в ПЭВМ). Для определения наличия дополнительного ПЗУ будем читать первое слово из каждых 2 Кбайт, начиная с адреса C000:0000 в поисках маркера расширения ПЗУ: 55AAh. Если такой маркер найден, то следующий байт содержит длину модуля ПЗУ.

В заключительной части программы определим версию DOS, установленную на ПЭВМ. Для этого воспользуемся функцией DOS 30h, которая возвращает в регистре AL старшее число номера версии, а в регистре AH — младшее число.

Текст программы

```
/*----Лабораторная работа N4-----------------*/
/*----"Проверка состава оборудования"--------*/

/* Подключение стандартных заголовков */
#include <dos.h>
#include <conio.h>
#include <stdio.h>

/*-----------------------------------------------*/
void main()
{
unsigned char type_PC, /* Тип компьютера  */
    a,b;   /* Переменные для определения */
        /* характеристик памяти ПЭВМ  */
unsigned int konf_b; /* Байт конфигурации из BIOS  */
 char *type[]={"AT","PCjr","XT","IBM PC","unknown"};
 unsigned char typ1A[]={0,1,2,4,5,6,7,8,10,11,12,0xff};
 char *types1A[]={"нема дисплею","MDA, моно","CGA, цв.",
  "EGA, цв.","EGA, моно","PGA, цв.",
  "VGA, моно, анал.","VGA, кол., анал.",
```

```
 "MCGA, кол., цифр.","MCGA, моно, анал."
 "MCGA, кол., анал.","неизвестный тип",
       "непредусмотренный код"};
unsigned int j;    /* Вспомогательная переменная */
unsigned int seg;   /* Сегмент ПЗУ    */
unsigned int mark=0xAA55;  /* Маркер ПЗУ  */
unsigned char bufVGA[64]; /* Буфер данных VGA */
union REGS rr;
struct SREGS sr;

textbackground(0);
clrscr();
textattr(0x0a);
cprintf("Лабораторная работа N5");
cprintf("\nПроверка состава оборудования");

/* Определения типа компьютера */
type_PC=peekb(0xF000,0xFFFE);
if( (type_PC-=0xFC)>4)
   type_PC=4;
textattr(0x0b);
cprintf("\nТип компьютера: ");
textattr(0x0f);
cprintf("%s\n\r",type[type_PC]);

/* Конфигурация*/
konf_b=peek(0x40,0x10); /* Чтение байта оборудования */
      /* из памяти BIOS      */
textattr(0x0b);
cprintf("Конфигурация:\n\r");

 /* Количество дисководов */

textattr(0x0e);
cprintf("  Дисководов ГМД:      ");
textattr(0x0f);
if(konf_b&0x0001)
   cprintf("%d\n\r",((konf_b&0x00C0)>>6)+1);
else
 cprintf("нет\n\r");
textattr(0x0e);
cprintf("  Математич. сопроцессор:  ");
textattr(0x0f);
if(konf_b&0x0002)
 cprintf("есть\n\r");
```

```
else
 cprintf("нет\n\r");
textattr(0x0e);
cprintf("  Тип дисплейного адаптера: ");
textattr(0x0f);

/* Определение активного адаптера */
/* Предположим наличие VGA */
rr.h.ah=0x1a;
rr.h.al=0;
int86(0x10,&rr,&rr);
if(rr.h.al==0x1a) /* Поддерживается функция 1Ah */
{       /* прерывания 10h       */
for(j=0;j<12;j++)
 if(rr.h.bl==typ1A[j])
 break;
cprintf("%s",types1A[j]);

if(j>0 && j<12)
{
 rr.h.ah=0x1b;
 rr.x.bx=0;
 sr.es=FP_SEG(bufVGA);
 rr.x.di=FP_OFF(bufVGA);
 int86x(0x10,&rr,&rr,&sr);
 cprintf(", %d Кбайт\n\r",((int)bufVGA[49]+1)*64);
}
else
   cprintf("\n\r");
}
else
{
/* Предположим наличие EGA */
rr.h.ah=0x12;
rr.h.bl=0x10;
int86(0x10,&rr,&rr);
if(rr.h.bl!=0x10) /* Поддерживается функция 12h */
{       /* прерывания 10h */
 cprintf("EGA");
 if(rr.h.bh)
 cprintf(" моно");
 else
 cprintf(" кол.");
 cprintf(", %d Кбайт\n\r",((int)rr.h.bl+1)*64);
}
```

```
else
{
 /* CGA или MDA */
 switch(konf_b&0x0030)
 {
 case 0:  cprintf("EGA/VGA\n\r");break;
 case 0x10: cprintf("CGA,40\n\r");break;
 case 0x20: cprintf("CGA,80\n\r");break;
 case 0x30: cprintf("MDA");break;
 }
}
}

/* Блоки ОЗУ на системной плате */
textattr(0x0e);
cprintf("\n\r  Первичный блок памяти:  ");
textattr(0x0f);
switch (konf_b&0x000C)
{
case 0:cprintf("16 Кбайт\n\r");break;
case 4:cprintf("32 Кбайт\n\r");break;
case 8:cprintf("48 Кбайт\n\r");break;
case 12:cprintf("64 Кбайт или больше\n\r");break;
}

/* Количество последовательных портов RS-232 */
textattr(0x0e);

cprintf("  Портов RS232:       ");
textattr(0x0f);
cprintf("%d\n\r",(konf_b&0x0E00)>>9);

/* Наличие джойстика */
textattr(0x0e);
cprintf("  Джойстик:         ");
textattr(0x0f);
if(konf_b&0x1000 )
cprintf("есть\n\r");
else
cprintf("нет\n\r");

/* Количество параллельних принтеров */
textattr(0x0e);
cprintf("  Принтеров:        ");
textattr(0x0f);
```

```
cprintf("%d\n\n\r",(konf_b&0xC000)>>14);

/* Объем оперативной памяти */

textattr(0x0e);
cprintf("Объем оперативной памяти: ");
textattr(0x0f);
cprintf("%d Кбайт\n\r",peek(0x40,0x13));
textattr(0x0e);

/* Наличие и объем extended-памяти */
outportb(0x70,0x17);
a=inport(0x71);
outportb(0x70,0x18);
b=inport(0x71);
cprintf("Объем extended-памяти:  ");
textattr(0x0f);
cprintf("%d Кбайт\n\n\r",(b<<8)|a);

/* Наличие дополнительных ПЗУ */
for( seg=0xC000;seg<0xFFB0;seg+=0x40)
/* Просмотр памяти от C000:0 с шагом 2 К */
if(peek(seg,0)==mark)  /* Маркер найден */
{
 textattr(0x0a);
 cprintf("Адрес ПЗУ =");
 textattr(0x0f);
 cprintf(" %04x",seg);
 textattr(0x0a);
 cprintf(". Длина модуля = ");
 textattr(0x0f);
 cprintf("%d",512*peekb(seg,2));
 textattr(0x0a);

 cprintf(" байт\n\r",peekb(seg,2));
}

/* Определение версии операцийной системы */
rr.h.ah=0x30;

intdos(&rr,&rr);
textattr(0x0c);
cprintf("\n\rВерсия MS-DOS ");
textattr(0x0f);
cprintf("%d.%d\n\r",rr.h.al,rr.h.ah);
```

```
 textattr(0x0a);
 gotoxy(30,24);
 cprintf("Нажмите любую клавишу");
 textattr(0x07);
 getch();
 clrscr();
}
```

Результаты работы программы

В процессе работы программы на экран была выведена такая информация:

Лабораторная работа N4

Проверка состава оборудования

Тип компьютера: AT

Конфигурация:

Дисководов ГМД: 2

Математич. сопроцессор: есть

Тип дисплейного адаптера: VGA, кол., анал., 256 Кбайт

Первичный блок памяти: 16 Кбайт

Портов RS232: 2

Джойстик: нет

Принтеров: 1

Объем оперативной памяти: 639 Кбайт

Объем extended-памяти: 384 Кбайт

Адрес ПЗУ = c000. Длина модуля = 24576 байт

Версия MS-DOS 6.20

Лабораторная работа № 5

Тема: Управление клавиатурой

Цели и задачи урока:

Изучение организации и принципов работы клавиатуры и закрепление практических навыков управления ею, а также практических навыков создания собственных программ обработки прерываний

Тип урока – практическое занятие.

Методы обучения – репродуктивный

Форма: фронтальная

Обеспечение занятия- ПК, тетради.

Ход урока

Постановка задачи

Разработать программу обработки прерывания от клавиатуры, которая должна:

распознавать нажатие «горячей» комбинации клавиш и реагировать на него звуковым сигналом;

при первом нажатии «горячей» комбинации переходить в режим блокировки ввода заданной клавиши, при втором — отменять этот режим;

системная обработка всех других клавиш нарушаться не должна.

Литература

1. Разработка модулей программного обеспечения для компьютерных систем: учебник для студ. учреждений сред. проф. образования / Г. Н. Федорова. - 4-е изд., перераб. - М.: Издательский центр «Академия», 2020. - 384 с. ISBN 978-5-4468-9443-7
2. http://www.uhlib.ru/kompyutery_i_internet/sistemnoe_programmirovanie_v_srede_windows/index.php (дата обращения: 25. 05. 2021г.)

Индивидуальные задания

Пример решения задачи

Индивидуальное задание:

комбинация клавиш **LeftCtrl+RightShift+F3**;

блокирование ввода клавиши 3.

Разработка алгоритма

Структура программы

Программа состоит из основной программы и трех функций.

void *readvect(int in) — функция читает вектор прерывания с номером **in** и возвращает его значение.

void writevect (int in, void *h) — функция устанавливает новый вектор прерывания in на новый обработчик этого прерывания по адресу **h**.

void interrupt new9() — процедура нового обработчика прерывания 9h.

Описание переменных

Глобальные переменные программы: **old9** — адрес старого обработчика прерывания 9h; **F3_code** — скан-код клавиши «F3», которая входит в комбинацию «горячих» клавиш;**key3_code** — скан-код клавиши «3», которая будет блокироваться/разблокироваться при каждом нажатии «горячей» комбинации клавиш; **f** — флаг, который при каждом нажатии «горячей» комбинации клавиш переключается из состояния 0 в 1 или из 1 в 0 (состояние 1 означает, что клавиша «3» заблокирована); **rr** и **sr** — переменные, которые используются для задания значений регистров общего назначения и сегментных регистров соответственно при вызове прерывания.

В главной программе использует символьный массив **string** для проверки работы программы.

Переменные процедуры обработки прерывания 9h:

c — переменная, которая используется для подтверждения приема из клавиатуры, в случае, если была нажата клавиша «3», а флаг **f** показывал, что эта клавиша заблокирована;

x, **y** — переменные, которые используются для сохранения координат курсора на экране в момент вызова процедуры обработки прерывания;

byte17 — байт флага состояния клавиатуры в области данных BIOS по адресу 0040:0017;

byte18 — байт флага состояния клавиатуры в области данных BIOS по адресу 0040:0018;

mask — маска, которая используется для определения нажатия клавиши левый **Shift** (в этом случае бит 1 в byte17 установлен в 1);

mask17 — маска, которая используется для определения нажатия клавиши **Ctrl** (в этом случае бит 2 в byte17 установлен в 1);

mask18 — маска, которая используется для определения нажатия клавиши левый **Ctrl** (в этом случае бит 0 в byte18 установлен в 1);

Описание алгоритма программы

Главная программа выполняет такие действия:

Запоминает адрес старого обработчика прерывания 9h, вызывая функцию readvect(in) с параметром in=9.
Записывает в таблицу векторов прерываний адрес нового обработчика прерывания с помощью функции writevect().
Вводом строки символов дает возможность проверить работу программы и ее реакцию на нажатие «горячей» комбинации клавиш и блокирование/разблокирование ввода клавиши «3».
В конце работы восстанавливает в таблице векторов прерываний адрес старого обработчика.
Для решения задачи процедура обработки прерывания от клавиатуры new9() должна действовать по такому алгоритму:
Прочитать флаги состояния клавиатуры (статус клавиш-переключателей), которые находятся в области данных BIOS (два байта по адресам 0040:0017 и 0040:0018).
Выделить бит 1 в флаге по адресу 0040:0017 (если он равен 1, то нажата клавиша левый Shift).
Выделить бит 2 в этом же флаге (если он равен 1, то нажата левый или правый Ctrl).
Выделить бит 0 в флаге состояния клавиатуры по адресу 0040:0018 (если он равен 1, то нажата клавиша левый Ctrl).
Из порта 60h прочитать скан-код нажатой клавиши.
Если нажата комбинация клавиш левый Shift, правый Ctrl (нажата клавиша Ctrl, но это не правый Ctrl)и клавиша F3, то выполнить п.7. Иначе — перейти к п.8.
Сигнализировать о нажатии «горячей» комбинации клавиш звуковым сигналом, переключить значение флага блокирования ввода клавиши «3» на обратное и вызвать старый обработчик прерывания от клавиатуры.
Прочитав байт из порта 60h, определить, нажата ли клавиша «3» и если, кроме этого, еще и флаг блокирования указывает на то, что она заблокирована (f=1), то выполнить п.п. 9 и 10, иначе — вызвать старый обработчик прерывания.
Послать подтверждение приема в клавиатуру. Для этого в порт 61h на короткое время выставить «1» по шине старшего разряда.
Сбросить контроллер прерываний, посылая код 20h в порт 20h.
Функция readvect() читает вектор заданного прерывания. Для чтения вектора используется функция 35h DOS (прерывания 21h):
Вход: AH = 35h;
AL = номер вектора прерывания.
Выход: ES:BX = адрес программы обработки прерывания
Функция **writevect()** устанавливает новый вектор прерывания на заданный адрес. Для записи вектора используется функция 25h DOS:
Вход: AL = номер вектора прерывания;
DS:BX = 4-байтный адрес нового обработчика прерывания.

Текст программы

```
/*--------Лабораторная работа N5---------------*/
/*---------Управление клавиатурой---------------*/
/* Подключение стандартных заголовков */
#include <dos.h>

void interrupt (*old9)(); /* Старый обработчик прерывания 9h */
void interrupt new9();  /* Новый обработчик прерывания 9h */
void *readvect (int in); /* Чтение вектора         */
```

```
void writevect (int in,void *h); /* Запись вектора      */

unsigned char F3_code=61; /* scan-code "F3"  */
unsigned char key3_code=4;/* scan-code "3"   */
char f=0;         /* Флаг           */
union REGS rr;
struct SREGS sr;

/*-----------------------------------------------*/
void main()
{
 char string[80]; /* Буфер для ввода текста */
 textbackground(0);
 clrscr();
 textattr(0x0a);
 cprintf("---------------");
 cprintf(" Лабораторная работа N5     ");
 cprintf("---------------");
 cprintf("---------------");
 cprintf("  Управление клавиатурой     ");
 cprintf("---------------");

 old9=readvect(9);
 writevect(9,new9);
 textattr(0x0c);
 cprintf("\n\n\r\"горячая\" комбинация: ");
 textattr(0x0a);
 cprintf("Left Shift, Right Ctrl, F3\n\r");
 textattr(0x0b);
 cprintf("Клавиша, которая блокируется: ");
 textattr(0x0f);
 cprintf("3");
 textattr(0x07);
 cprintf("\r\nВводите строку символов>");
 scanf("%s",string);
 writevect(9,old9);
}
/*-----------------------------------------------*/
/* Чтение вектора */
void *readvect(int in)
{
 rr.h.ah=0x35;
 rr.h.al=in;
 intdosx(&rr,&rr,&sr);
 return(MK_FP(sr.es,rr.x.bx));
```

```
}
/*---------------------------------------------*/
/* Запись вектора */
void writevect(int in,void *h)
{
 rr.h.ah=0x25;
 rr.h.al=in;
 sr.ds=FP_SEG(h);
 rr.x.dx=FP_OFF(h);
 intdosx(&rr,&rr,&sr);
}
/*---------------------------------------------*/
/* Новый обработчик 9-го прерыванмя */
void interrupt new9()
{
 unsigned char c,x,y;
 unsigned char byte17,byte18;
 unsigned char mask=0x02;
 unsigned char mask17=0x04;
 unsigned char mask18=0x01;

 byte17=peekb(0x40,0x17);
 byte18=peekb(0x40,0x18);
 if((inportb(0x60)==F3_code)&&(byte17&mask)&&
         (byte17&mask17)&&(!(byte18&mask18)))
 {
  cputs("\7");
  x=wherex();
  y=wherey();
  gotoxy(55,3);
  textattr(0x1e);
  if(f==0)
  {
  f=1;
  cprintf("Клавиша \"3\" заблокирована ");
  }
  else
  {
  f=0;
  cprintf("Клавиша \"3\" разблокирована");
  }
  gotoxy(x,y);
  textattr(0x07);
  (*old9)();
 }
```

```
 if( (f==1) && (inportb(0x60)==key3_code) )
 {
  c=inportb(0x61);
  outportb(0x61,c|0x80);
  outportb(0x61,c);
  outportb(0x20,0x20);
 }
 else
  (*old9)();
}
```

Результаты работы программы

Во время программы при первом нажатии комбинации клавиш **LeftCtrl+RightShift+F3** программа переходит в режим блокирования вводу клавиши 3, при втором — отменяет этот режим

Лабораторная работа № 6
Тема: Управление таймером

Цели и задачи урока:
Изучение функций системного таймера и закрепление практических навыков работы с ним.
Тип урока – практическое занятие.
Методы обучения – репродуктивный
Форма: фронтальная
Обеспечение занятия- ПК, тетради.
Ход урока

Постановка задачи

Построить модель аналого-цифрового преобразователя (АЦП), которая работает в реальном времени. Процесс, который дискретизуется, моделируется программой (программным блоком), который выполняет циклическое вычисление функции **y=F(x)**, где **x** — номер итерации. Преобразователь моделируется программой, которая выполняет с заданной частотой (в реальном времени) прерывание процесса, считывание и запоминание текущего значения функции. Запомнить не меньше 80 значений функции. Обеспечить наглядное представление результатов работы «АЦП».

Литература

1. Разработка модулей программного обеспечения для компьютерных систем: учебник для студ. учреждений сред. проф. образования / Г. Н. Федорова. - 4-е изд., перераб. - М.: Издательский центр «Академия», 2020. - 384 с. ISBN 978-5-4468-9443-7
2. http://www.uhlib.ru/kompyutery_i_internet/sistemnoe_programmirovanie_v_srede_windows/index.php (дата обращения: 25. 05. 2021г.)

Индивидуальные задания

Для получения более наглядного представления «процесса» допускается подбирать другие коэффициенты функции. Частоту дискретизации выдерживать с точностью до 1 гц.

Пример решения задачи

Индивидуальное задание

функция — y=50*(sin(x/10)+cos(x/8))+R+150;
R — в диапазоне 0 — 10;
частота — 36.4 Гц.

Разработка алгоритма решения

Структура программы

Программа состоит из основной программы и трех функций.

void *readvect(int in) — функция читает вектор прерывания с номером in и возвращает его значение.

void writevect (int in, void *h) — функция устанавливает новый вектор прерывания in на новый обработчик этого прерывания по адресу h.

void interrupt newtime() — процедура нового обработчика прерывания таймера.

Описание переменных и констант

В этой программе применяются две константы:

TIMEINT=8 — номер прерывания таймера;

NN=100 — максимальное число показаний АЦП.

Переменные, глобальные для всей программы:

y — массив показаний АЦП;

ny — текущий индекс в массиве показаний;

yc — текущее значение функции;

kf — счетчик вызовов **oldtime** (**oldtime** вызывается каждые второй раз);

rr и **sr** — переменные, которые используются для задания значений регистров общего назначения и сегментных регистров, соответственно при вызове прерывания.

Переменные главной программы:

oldtic — старый коэффициент деления;

newtic — новый коэффициент деления (применяется для увеличения частоты вызова прерываний таймера);

x — аргумент заданной функции **F(x)**;

dd — тип графического адаптера;

m — режим графики;

errorcode — код результата инициализации графики.

Описание алгоритма программы

Программу можно назвать моделью процесса АЦП. Главная программа постоянно вычисляет значения заданной функции **F(x)** при переменном аргументе, что имитирует непрерывный сигнал, а обработчик прерывания 8 имитирует преобразователь с постоянным шагом дискретизации по времени. Перед началом работы канал 0 таймера программируется на частоту в 2 раза большую обычной (записью в порт 43h управляющего байта 00110110b=36h, а потом посылкой в порт 40h нового значения коэффициента деления), таким образом, «частота дискретизации» составляет около 36.4 Гц. При поступлении следующего прерывания запоминается текущее значение функции **F(x)**, старый обработчик прерывания oldtime вызывается не при кожному прерывании, а лишь один раз из двух (переменная **kf** — счетчик по модулю 2), когда **oldtime** не вызывается, наш обработчик сам сбрасывает контроллер прерываний посылкой значения 20h в порт 20h. После набора 100 «показаний АЦП» восстанавливается старый вектор обработчика таймера, а результат аналого-цифрового преобразование выводится на терминал в графическом режиме в виде решетчатой функции.

Функция **readvect()** читает вектор заданного прерывания. Для чтения вектора применяется функция 35h DOS (прерывания 21h):

Вход: AH = 35h;

AL = номер вектора прерывания.

Выход: ES:BX = адрес программы обработки прерывания.

Функция **writevect()** устанавливает новый вектор прерывания по заданному адресу. Для записи вектора применяется функция 25h DOS:
Вход: AH = 25h;
AL = номер вектора прерывания;
DS:BX = 4-байтный адрес нового обработчика прерывания.

Текст программы

```
/*-------Лабораторная работа N6----------------*/
/*-------"Управление таймером"------------------*/

/* Подключение стандартных заголовков */
#include <dos.h>
#include <math.h>
#include <stdlib.h>
#include <graphics.h>
#include <time.h>
#include <conio.h>

#define TIMEINT 8   /* Прерывание таймера  */
#define NN 100 /* Максимальное количество показаний */

void interrupt (*oldtime)(); /* Новый обработчик  прерываний таймера */

void interrupt newtime(); /* Старый обработчик прерываний таймера */
static int y[NN]; /* Накопитель показаний  */
static int ny;   /* Индекс в массиве y  */
static int yc;   /* Текущее значение    */
static int kf; /* Счетчик вызовов oldtime  */
union REGS rr;   /* Запись нового вектора */
struct SREGS sr;
void *readvect(int in);   /* Получение старого вектора */
void writevect(int in, void *h); /* Запись нового вектора */
/*-----------------------------------------------*/
void main()
{
 unsigned oldtic=65535u; /* Старый коэфф. деления   */
 unsigned newtic=32768u; /* Новый коэфф. деления    */
 int dd,         /* Графический драйвер */

   m,          /* Графический режим   */
   errorcode;      /* Код ошибки      */
 double x;   /* Аргумент функций sin и cos */

 textbackground(0);
 clrscr();
 textattr(0x0a);
```

```
cprintf("  Лабораторная работа N6       ");
cprintf("\n   Управление таймером       ");
textattr(0x8e);
gotoxy(35,12);
cprintf("Please wait");
/* Программирование канала 0 */
outportb(0x43,0x36);   /* Управляющий байт */
outportb(0x40,newtic&0x00ff); /* Младший байт счетчика */
outportb(0x40,newtic>>8);  /* Старший байт счетчика */
ny=-1; /* Признак того, что АЦП еще не началось */
kf=15;
/* Подключение к вектору */
oldtime=readvect(TIMEINT);
writevect(TIMEINT,newtime);
/* Запуск "непрерывного процесса" */
randomize();
for (x=ny=0; ny<NN; x+=1)
 yc=(int)(50*(sin(x/10)+cos(x/8))+random(11)+150);
/* Восстановление вектора */
writevect(TIMEINT,oldtime);
/* Восстановление канала 0 */
outportb(0x43,0x36);    /* Управляющий байт */
outportb(0x40,oldtic&0x00ff); /* Младший байт счетчика */
outportb(0x40,oldtic>>8);  /* Старший байт счетчика */

/* Вывод запомненных результатов */
dd=3; /* EGA, 16 цветов */
m=1; /* Режим 640*350 */
initgraph(&dd,&m,"");
/* проверка результата инициализации */
errorcode = graphresult();
if (errorcode != grOk)  /* ошибка графического режима */
{
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1);      /* аварийное завершение */
}
setcolor(10);
settextstyle(0,0,2);
outtextxy(15,10,"Результати аналого-цифрового преобразования:");

setcolor(9);
rectangle(15,40,624,330);
setcolor(11);
```

```
 for(ny=0; ny<NN; ny++)
 {
  circle(22+ny*6,330-y[ny]*1,2);
  line(22+ny*6,330,22+ny*6,330-y[ny]*1);
 }
 setcolor(12);
 settextstyle(0,0,1);
 outtextxy(260,340,"Нажмите любую клавишу ...");
 getch();
 closegraph();
}

/* Новый обработчик прериваний таймера */
void interrupt newtime()
{
 if (--kf<0) {
  /* Виклик oldtime — на 2-й раз */
  (*oldtime)();
  kf=1;
  }
 else /* иначе — сброс контроллера */
  outportb(0x20,0x20);
 if ((ny>=0)    /* Если АЦП началось, */
  &&(ny<NN)) /* и NN показаний еще не набрано, */
   y[ny++]=yc;  /* запоминание очередного показания */
}

/* Получение старого вектора */
void *readvect(int in)
{
 rr.h.ah=0x35; rr.h.al=in;
 intdosx(&rr,&rr,&sr);
 return(MK_FP(sr.es,rr.x.bx));
}

/* Запись нового вектора */
void writevect(int in, void *h)
{
 rr.h.ah=0x25;
 rr.h.al=in;
 sr.ds=FP_SEG(h);
 rr.x.dx=FP_OFF(h);
 intdosx(&rr,&rr,&sr);
}
```

Результаты работы программы

Результат работы представляется в графическом режиме в виде решетчатой функции на экране терминала.

Лабораторная работа № 7

Тема: Управление видеоадаптером

Цели и задачи урока:

Изучение особенностей функционирования видеосистемы в текстовом режиме и получение практических навыков работы с видеомонитором в этом режиме.

Тип урока – практическое занятие.

Методы обучения – репродуктивный

Форма: фронтальная

Обеспечение занятия- ПК, тетради.

Ход урока

Постановка задачи

Применяя прямую запись в видеопамять получить на экране оригинальный, желательно динамический видеоэффект. Возможны (но не обязательны) такие варианты видеоэффектов:

«теннисный мячик» — шарик, который летает по экрану и отражается от рамок и границ экраны;

«сухой лист» — опадание букв с экрана;

«жук-пожиратель» — фигурка, которая перемещается по экрану по случайной траектории и «съедает» буквы;

«удав» — то же, что и «жук», но к тому же он увеличивается в размерах, по мере «поедания» букв;

Литература

1. Разработка модулей программного обеспечения для компьютерных систем: учебник для студ. учреждений сред. проф. образования / Г. Н. Федорова. - 4-е изд., перераб. - М.: Издательский центр «Академия», 2020. - 384 с. ISBN 978-5-4468-9443-7
2. http://www.uhlib.ru/kompyutery_i_internet/sistemnoe_programmirovanie_v_srede_windows/index.php (дата обращения: 25. 05. 2021г.)

Пример решения задачи

Индивидуальное задание:

весь экран (80x25 символов) условно делится на прямоугольники размером (10x5 символов).

текущий прямоугольник инвертирует экран под собой.

управлять положением текущего прямоугольника на экране можно с помощью клавиш управления курсором.

при нажатии клавиши «пробел» текущий прямоугольник обменивается содержимым с левым верхним прямоугольником.

при нажатии клавиши Enter содержимое прямоугольников экрана начинает перемешивается случайным образом между собой до нажатия любой клавиши.

после этого, используя клавиатуру, можно восстановить начальный экран или выйти из программы (клавиша Esc).

Разработка алгоритм решения

Структура программы

Программа состоит из основной функции **main()** и семи вспомогательных функций.

byte GetSym(x1,y1) — функция читает символ с заданной позиции экрана дисплея.

byte GetAtr(x1,y1) — функция читает атрибут символа с заданной позиции экрана дисплея.

void PutSym(x1,y1,sym) — функция выводит на экран дисплея символ в заданную позицию (x1,y1).

void PutAtr(x1,y1,atr) — функция меняет на экране дисплея атрибут символа в заданной позиции (x1,y1).
void Invert(x1,y1) — функция инвертирует участок на экране размером (10x5), координаты (x1,y1) задают один из участков на экране.
void Change(x,y) — функция обменивает содержимое текущего участка с содержимым левого верхнего участка на экране. Координаты (x,y) задают положение текущего участка.
void RandText(void) — функция псевдослучайным образом перетасовывает все участки на экране.

Описание переменных

Переменные, глобальные для всей программы:
xk — координата X текущего участка;
yk — координата Y текущего участка;
Координаты участка задаются в пределах: X — [0..7], Y — [0..4] .

Описание алгоритма программы

Основная функция **main()** проверяет, был ли в командной строке дополнительный параметр. Если нет, программа не очищает старый экран. Если какой-нибудь параметр был, то экран очищается и выводится инструкция по управлению программой. Далее в основной программе выполняется бесконечный цикл, в котором обрабатываются коды нажатых клавиш и, в зависимости от них, вызываются вспомогательные функции. Выход из цикла — по клавише **Esc**.
Функции **GetSym(x1,y1)**, **GetAtr(x1,y1)** читают непосредственно из видеопамяти дисплея символ и атрибут соответственно.
Функции **PutSym(x1,y1,sym)**, **PutAtr(x1,y1,atr)** выводят непосредственно в видеопамять дисплея символ и атрибут соответственно.
Во всех этих четырех функциях координаты задаются в квадрате 79x24 символов (нумерация начинается с нуля).
Функция **Invert(x1,y1)** использует функции **GetAtr** и **PutAtr** для инверсии прямоугольника экрана размером 10x5 по маске 0x7f, при этом независимо выполняется инверсия фона и цвета символа.
Функция **Change(x,y)** обменивает содержимое текущего участка с содержимым левого верхнего участка путем последовательного побайтного обмену атрибутов и символов. Она использует функции **GetSym**, **GetAtr**, **PutSym**, **PutAtr**.
Функция **RandText(void)** — псевдослучайным образом перетасовывает все участки на экране, при этом она в цикле увеличивает на единицу локальные в данной функции координаты текущего участка **xk**, **yk** и обращается к функции **Change**. Таким образом достигается эффект перемешивания. Функция работает, пока не будет нажата любая клавиша.

Текст программы

```
/*------Лабораторная работа N7-----------------*/
/*-----Управление видеоадаптером.---------------*/
#include <dos.h>

#include <stdio.h>
#include <stdlib.h>
```

```
#include <conio.h>
#include <time.h>
/*--------------Константы---------------------- */
#define VSEG 0xb800    /* Сегментный адрес видеопамяти */
#define byte unsigned char

#define word unsigned int
#define Esc 27
#define Spase 32
#define Enter 13
#define Up 0x48
#define Down 0x50
#define Left 0x4b
#define Right 0x4d
#define Home 0x47
int xk,yk;
/*----Чтение символа из видеопамяти-------------*/
byte GetSym(x1,y1)
int x1,y1;
{
 return(peekb(VSEG,y1*160+x1*2));
}
/*---Чтение атрибута из видеопамяти-------------*/
byte GetAtr(x1,y1)
int x1,y1;
{
 return(peekb(VSEG,y1*160+x1*2+1));
}
/*---Запись символа в видеопамять--------------*/
void PutSym(x1,y1,sym)
int x1,y1;
byte sym;
{
 pokeb(VSEG,y1*160+x1*2,sym);
}
/*---Запись атрибута в видеопамять--------------*/
void PutAtr(x1,y1,atr)
int x1,y1;
byte atr;
{
 pokeb(VSEG,y1*160+x1*2+1,atr);
}
/*--Инверсия квадрата на экране-----------------*/
void Invert(x1,y1)
int x1,y1;
```

```
{
 byte b;
 int i,j;
 for (j=0;j<5;j++)
 for (i=0;i<10;i++)
  {
  b=GetAtr(x1*10+i,y1*5+j);
  PutAtr(x1*10+i,y1*5+j,(b^0x7f));
  }
}
/*--Замена текущего квадрата на левый верхний--*/
void Change(x,y)
byte x,y;
{
 int i,j;
 byte ba,bs;
 if ((x!=0)||(y!=0))
 for (j=0;j<5;j++)
  for (i=0;i<10;i++)
  {
   bs=GetSym(x*10+i,y*5+j);
   ba=GetAtr(x*10+i,y*5+j);
   PutSym(x*10+i,y*5+j,GetSym(i,j));
   PutAtr(x*10+i,y*5+j,GetAtr(i,j));
   PutSym(i,j,bs);
   PutAtr(i,j,ba);
  }
}
/*--Перемешивание квадратов до нажатия клавиши-*/
void RandText(void)
{
 Invert(xk,yk);
 xk=5;
 yk=1;
 while(!kbhit())
 {
 Change(xk,yk);
 xk++;
 if (xk>7) xk=0;
 yk++;
 if (yk>4) yk=0;
 }
 Invert(xk,yk);
}
/*-----Начало основной программы----------*/
```

```
main(int argn,char *argc[])
{
int i;

xk=0;
yk=0;
if (argn>1){}
else      /* Если параметров нет, вывод инструкции */
{
textattr(10);
clrscr();
cprintf("---------------");
cprintf("   Лабораторная работа N7       ");
cprintf("---------------");
cprintf("---------------");
cprintf("  Управление видеоадаптером.      ");
cprintf("---------------");
textattr(15);
gotoxy(23,4);cprintf("Демонстрация работы с видеопамятью.");
textattr(12);
gotoxy(30,6);cprintf("<< М О З А И К А >>");
textattr(14);
gotoxy(30,8);cprintf("Клавиши управления:");
gotoxy(7,10);cprintf("< Left, Right, Up, Down> — ");
  cprintf("управление выделенним квадратом.");
gotoxy(7,11);cprintf("<Spase Bar> — Обмен содержимым ");
      cprintf("между выделенным квадратом");
gotoxy(7,12);cprintf("      и левым верхним");
      cprintf(" квадратом.");
gotoxy(7,13);cprintf("<Enter> — перемешивание квадратов");
      cprintf(" до нажатия любой клавиши.");
gotoxy(7,14);cprintf("<Esc> — вихід.");
textattr(11);
gotoxy(28,16);cprintf("З А Д А Ч А  И Г Р Ы :");
gotoxy(14,17);cprintf("Собрать при помощи клавиш ");
     cprintf("управления начальный экран.");
textattr(12);
gotoxy(27,19);cprintf("Ж е л а е м  у с п е х а !");
textattr(7);
gotoxy(1,21);cprintf("Примечание: При запуске с ");
      cprintf("параметром <->");
gotoxy(13,22);cprintf("начальным экраном для игры ");
      cprintf("является текущий.");
}
Invert(xk,yk);
```

```
 for(i=0;i==0;)
 switch(getch())
 { /* Обработка нажатых клавиш */
  case Esc: i++; break;
  case Enter:RandText();break;
  case Spase:Invert(xk,yk);

       Change(xk,yk);
       Invert(xk,yk);
       break;
  case 0:
   switch (getch()) {
     case Left:Invert(xk,yk);
          xk--;
          if(xk<0) xk=7;
          Invert(xk,yk);
          break;
     case Right:Invert(xk,yk);
          xk++;
          if(xk>7) xk=0;
          Invert(xk,yk);
          break;
     case  Up:Invert(xk,yk);
          yk--;
          if(yk<0) yk=4;
          Invert(xk,yk);
          break;
     case Down:Invert(xk,yk);
          yk++;
          if(yk>4) yk=0;
          Invert(xk,yk);
          break;
     }
 }
Invert(xk,yk);
}
```

Результаты работы программы

Результаты работы программы выводятся на экран терминала и меняются интерактивно

Лабораторная работа № 8

Тема: Главная загрузочная запись

Цели и задачи урока:

Получение практических навыков в работе с Главной Загрузочной Записью жесткого диска.

Тип урока – практическое занятие.

Методы обучения – репродуктивный

Форма: фронтальная

Обеспечение занятия- ПК, тетради.

Ход урока

Постановка задачи

Прочитать и выполнить форматный вывод на экран Главной Загрузочной Записи жесткого диска на своем рабочем месте.

Порядок выполнения

Порядок выполнения работы и содержание отчета определены в общих указаниях.

Литература

1. Разработка модулей программного обеспечения для компьютерных систем: учебник для студ. учреждений сред. проф. образования / Г. Н. Федорова. - 4-е изд., перераб. - М.: Издательский центр «Академия», 2020. - 384 с. ISBN 978-5-4468-9443-7
2. http://www.uhlib.ru/kompyutery_i_internet/sistemnoe_programmirovanie_v_srede_windows/index.php (дата обращения: 25. 05. 2021г.)

Пример решения задачи

Разработка алгоритма решения

Программа состоит из основной программы **main()**, которая реализует все действия для чтения Главной Загрузочной Записи.

Описание переменных

Переменные в основной программе:

x, y — экранные координаты;

head — номер головки (0);

Sect_Trk — номер дорожки и сектора (0,1);

ndrive=0 — номер логического диска;

EndList — указатель на подпись.

Кроме того, в программе есть такие структуры:

Структура элемента раздела:

```
 struct Part {
  byte ActFlag;   /* признак активного раздела */
  /* физический адрес начала раздела */
byte Begin_Hd;         /* # головки */
word Begin_SecTrk;  /* # сектора та дорожки */
  byte SysCode;     /* код системы */
   /* физический адрес конца раздела */
   byte End_Hd;     /* # головки */
   word End_SecTrk;   /* # сектора и дорожки */
  dword RelSec;  /* # сектора початку */
  dword Size;  /* количество секторов */
  };
```

Структура главной загрузочной записи.

```
struct MBR {
char LoadCode[0x1be]; /* программа загрузки */
struct Part rt[4];  /* 4 элемента разделов */
word EndFlag;          /* подпись MBR */
};
```

Описание алгоритма программы

Эта программа демонстрирует разделение логического диска.

Начальный адрес для чтения задается: 0,0,1. При помощи прерывания 0x13 программа считывает сектор по заданному адресу, далее происходит поэлементный анализ таблицы разделов — пока не встретится признак конца таблицы или раздел нулевого размера. Значения полей элемента таблицы выводятся на экран. Манипуляции, которые описываются макросами TRK и SECT, обеспечивают распаковку номера дорожки и сектора. Если в поле SysCode содержится признак расширенного раздела, то устанавливается новый дисковый адрес, считывается новый сектор и анализируется новая таблица.

Текст программы

```
/*-------Лабораторная работа N8----------------*/
/*---"Главная загрузочная запись"--------------*/
/* Стандартные заголовки */
#include <dos.h>
#include <conio.h>

/* Типы данных */

#define byte unsigned char
#define word unsigned int
#define dword unsigned long
void read_MBR(void);  /* Чтение MBR */
/* Получение из упакованного SecTrk # сектора */
#define SECT(x) x&0x3f
/* Получение из упакованного SecTrk # дорожки */
#define TRK(x) (x>>8)|((x<<2)&0x300)

 /* структура элемента раздела */
 struct Part {
  byte ActFlag; /* признак активного раздела */
     /* физический адрес начала раздела */
  byte Begin_Hd;         /* # головки */
  word Begin_SecTrk; /* # сектора и дорожки */
  byte SysCode;        /* код системы */
      /* физический адрес конца раздела */
  byte End_Hd;          /* # головки */
  word End_SecTrk;  /* # сектора и дорожки */
  dword RelSec;     /* # сектора початку */
  dword Size;      /* количество секторов */
  };
 /* структура главной загрузочной записи */
 struct MBR {
char LoadCode[0x1be];  /* программа загрузки */
struct Part rt[4];   /* 4 эл-та разделов */
word EndFlag;      /* подпись MBR */
} mbr;
/* дополнительные переменные */
```

```
int x=10,y;   /* экранные координаты */
byte head=0;   /* номер головки (0) */
word Sect_Trk=1; /* номер дорожки и сектора (0,1) */
int ndrive=0;  /* номер логического диска */
word *EndList;  /* указатель на подпись */
union REGS rr;
struct SREGS sr;
word i;
/*-----------------------------------------------*/
main()
{
 textbackground(0);
 clrscr();
 textattr(0x0a);
 cprintf("       Лабораторная работа N8");
 gotoxy(1,2);
 cprintf("      Главная загрузочная запись");
 textattr(12);
 gotoxy(30,4);
 cprintf("Разделы жесткого диска:\n");
 gotoxy(1,6);
 textattr(11);
 cprintf("Лог.диск -----> \n\r");
 cprintf("Признак ------> \n\r");
 cprintf("Код системы --> \n\r");
 cprintf("Начало: гол.--> \n\r");
 cprintf("    дор.--> \n\r");
 cprintf("    сект.-> \n\r");
 cprintf("Конец: гол.--> \n\r");
 cprintf("    дор. -> \n\r");
 cprintf("    сект.-> \n\r");
 cprintf("Нач.сектор ---> \n\r");
 cprintf("Размер -------> \n\r");
 textcolor(11);
NEXT:
 read_MBR();
 for (EndList=(word *)&mbr.rt[(i=0)];
   (*EndList!=0xaa55)&&(mbr.rt[i].Size>0L);
    EndList=(word *)&mbr.rt[++i])
 {
  /* координаты курсора */
  y=6;
  x+=7;
  gotoxy(x,y++);
  if (mbr.rt[i].SysCode==5)
```

```
   {textattr(13);
   cprintf("Ext  ");

   }
  else
   textattr(12);
  cprintf("%-7c",'C'+ndrive++);

  gotoxy(x,y++); textattr(14);
  cprintf("%02xH    ",mbr.rt[i].ActFlag);
  gotoxy(x,y++); textattr(15);
  cprintf("%-7d",mbr.rt[i].SysCode);
  gotoxy(x,y++); textattr(14);
  cprintf("%-7d",mbr.rt[i].Begin_Hd);
  gotoxy(x,y++); textattr(15);
  cprintf("%-7u",TRK(mbr.rt[i].Begin_SecTrk));
  gotoxy(x,y++); textattr(14);
  cprintf("%-7u",SECT(mbr.rt[i].Begin_SecTrk));
  gotoxy(x,y++); textattr(15);
  cprintf("%-7d",mbr.rt[i].End_Hd);
  gotoxy(x,y++); textattr(14);
  cprintf("%-7u",TRK(mbr.rt[i].End_SecTrk));
  gotoxy(x,y++); textattr(15);
  cprintf("%-7u",SECT(mbr.rt[i].End_SecTrk));
  gotoxy(x,y++); textattr(14);
  cprintf("%-7lu",mbr.rt[i].RelSec);
  gotoxy(x,y++); textattr(15);
  cprintf("%-7lu",mbr.rt[i].Size);
  if (mbr.rt[i].SysCode==5)
  {
   /* если код системы 5, раздел содержит свою таблицу  разделов; определяется ее дисковый
адрес,  и новая таблица считывается в память */
   head=mbr.rt[i].Begin_Hd;
   Sect_Trk=mbr.rt[i].Begin_SecTrk;
   goto NEXT;
  }
 }
 gotoxy(x,y++);
 textattr(10+128);
 gotoxy(29,18);

 cprintf("Нажмите любую клавишу...");
 getch();
}
```

```
/*-----------Читання MBR-----------------------*/
void read_MBR(void)
{
  rr.h.ah=2;      /* Чтение */
  rr.h.al=1;      /* Секторов 1 */
  rr.h.dl=0x80;    /* Жесткий диск */
  rr.h.dh=head;    /* Головка */
  rr.x.cx=Sect_Trk;  /* Дорожка, сектор */
  sr.es=FP_SEG(&mbr); /* Адрес буфера в ОП */
  rr.x.bx=FP_OFF(&mbr);
  int86x(0x13,&rr,&rr,&sr);
  /* Проверка ошибок чтения */
  if (rr.x.cflag)
  {
   printf("Ошибка чтения: %x. ",rr.h.ah);
   printf("Нажмите любую клавишу...\n\7");
   getch();
   exit();
  }
}
```

Результаты работы программы

В процессе работы программы на экран выводится информация такого вида:

```
      Лабораторная работа N8
     Главная загрузочная запись
Разделы жесткого диска:

Лог.диск -----> C   Ext  E   Ext  G
Признак ------> 80H  00H  00H  00H  00H
Код системы --> 1   5   4   5   0
Начало:  гол.--> 1   0   1   0   1
     дор.--> 0   121  121  724  724
     сект.-> 1   1   1   1   1
Конец:  гол.--> 4   4   4   4   4
    дор. -> 120  975  723  975  975
     сект.-> 17   17   17   17   17
Нач.сектор ---> 17   10285 17   51255 17
Размер -------> 10268 72675 51238 21420 21403

      Нажмите любую клавишу...
```

Лабораторная работа № 9

Тема: Дисковые структуры данных DOS

Цели и задачи урока:

Получение практических навыков в работе с Таблицей Размещения Файлов.

Тип урока – практическое занятие.

Методы обучения – репродуктивный

Форма: фронтальная
Обеспечение занятия- ПК, тетради.
Ход урока

Постановка задачи

Определить номера всех кластеров диска, которые занимает заданный преподавателем файл в текущем каталоге.

Литература

1. Разработка модулей программного обеспечения для компьютерных систем: учебник для студ. учреждений сред. проф. образования / Г. Н. Федорова. - 4-е изд., перераб. - М.: Издательский центр «Академия», 2020. - 384 с. ISBN 978-5-4468-9443-7
2. http://www.uhlib.ru/kompyutery_i_internet/sistemnoe_programmirovanie_v_srede_windows/index.php (дата обращения: 25. 05. 2021г.)

Пример решения задачи

Разработка алгоритма решения

Программа состоит из главной функции **main()** и одиннадцати вспомогательных функций.

void Read_Mbr(void) — функция чтения MBR и поиска требуемого раздела.

void Read_Boot(void) — функция чтения boot-сектора.

void Get_First(void) — функция определения абсолютного номера сектора начала логического диска.

void Read_Fat(void) — функция чтения FAT.

void Read_13(void *mem) — функция чтения сектора с помощью прерывания 13.

void Sect_to_Daddr(dword sect) — функция формирования физического дискового адреса из номера сектора.

dword Clust_to_Sect(word clust) — функция определения номера сектора по номеру кластера.

word Next_Clust(word clust) — функция выборки следующего кластера из FAT.

char *Get_Name(char *s, char *d) — функция выделения следующего элемента из строки-задания.

int Find_Name() — функция поиска имени в каталоге.

void End_of_Job(int n) — функция выдачи сообщений или аварийного завершения.

Описание переменных

В программе описаны структуры такого вида:

Физический дисковый адрес:

```
struct DADDR {
 byte h; /* головка         */
 word s, /* сектор          */
    t, /* дорожка          */
    ts; /* сектор, дорожка упакованные */
 };
```

Структура элемента раздела;

```
struct PART {
 byte Boot,       /* признак активного   */
 /* физический адрес начала раздела       */
 Begin_Hd;     /* # головки      */
 word Begin_SecTrk; /* # сектора и дорожки */
 byte SysCode,      /* код системы  */
  /* физический адрес конца раздела  */
```

```
 End_Hd;      /* # головки   */
 word End_SecTrk; /* # сектора и дорожки */
 dword RelSec,      /* # сектора початку  */
 Size;       /* количество секторов  */
};
```
Структура Главной Загрузочной Записи:
```
struct MBR
   {
char LoadCode[0x1be]; /* программа загрузки */
struct PART rt[4]; /* 4 элемента разделов  */
word EndFlag;     /* подпись MBR      */
   };
```
Структура загрузочной записи логического диска:
```
struct BootRec {
 byte jmp[3], ident[8];
 word SectSize;
 byte ClustSize;
 word ResSect;
 byte FatCnt;
 word RootSize, TotSecs;
 byte Media;
 word FatSize, TrkSecs, HeadCnt;
 word HidnSecL, HidnSecH;
 dword LongTotSecs;
 byte Drive, reserved1, DOS4_flag;
 dword VolNum; char VolLabel[11], FatForm[8];
 };
```
Структура элемента каталога:
```
struct Dir_Item {
  char fname[11];   /* имя файла      */
  byte attr;      /* атрибут       */
  byte reserved[10];
  word time;      /* время        */
  word date;      /* дата        */
  word cl;       /* номер 1-го кластера */
  dword size;      /* размер файла    */
  };
```
Переменные, глобальные для всей программы:

part — текущий элемент раздела;

buff1[512] — буфер MBR и boot;

*mbr — указатель на таблицу разделов;

*boot — указатель на корневую запись;

buff2[512] — буфер каталога и текста;

*dir — указатель на часть каталога;

*text — указатель на текстовый буфер;

*fat — указатель на FAT;
job[81] — строка-задание;
jobptr — текущий указатель в job;
cname[12] — текущее имя для поиска;
Fdisk — физический номер диска;
caddr — текущий дисковый адрес;
sect — текущий номер сектора;
clust — текущий номер кластера;
fat16 — признак формата FAT;
fsize — размер файла;
dirnum — номер элемента в каталоге;
FirstSect — абсолютный номер сектора начала;
rootdir=1 — признак корневого каталога или подкаталога (1/0);
lastsect — последний сектор при чтении;
fatalloc=0 — признак выделения памяти.

Описание алгоритм программы

Функция **main** запрашивает имя файла, потом обрабатывает его и, если все нормально, то запускает вспомогательные функции необходимые для просмотра FAT заданного файла.

Функция **Read_Mbr** выполняет выборку элемента таблицы разделов для заданного диска.

Функция **Read_Boot** считывает boot-сектор логического диска, причем для гибкого диска адрес этого сектора назначается — 0, 0, 1, а для жесткого — выбирается из **part**.

Функция **Get_First** определяет абсолютный номер начального сектора логического диска и сохраняет его переменной **First_Sect**. Это значение вычисляется из физического адреса начала, который берется из полей **Begin_Hd**, **Begin_SecTrk** элемента таблицы разделов.

Функция **Read_Fat** считывает в память FAT целиком, адрес начала FAT на диске и ее размер определяются из ранее прочитанного boot-сектора.

Функция **Read_13** читает один сектор с помощью прерывания BIOS.

Функция **Sect_to_Daddr** преобразует номер логического сектора в физический адрес.

Функция **Clust_to_Sect** преобразует номер кластера в номер сектора.

Функция **Next_Clust** определяет номер следующего кластера, анализируя FAT. Для последнего кластера (и для корневого каталога) эта функция возвращает нулевое значение.

Функция **Get_Name** предназначена для лексического разбора задания, она выделяет из задания очередное слово и переназначает **jobptr**. Пустое (NULL) значение **jobptr** — свидетельство об исчерпании задания.

Функция **Find_Name** выполняет поиск имени в каталоге. Здесь cname — требуемое имя, функция возвращает индекс найденного элемента в массиве **dir** или (-1).

Функция **End_of_Job** выполняет выдачу на экран различных сообщений при ошибках или при завершении программы.

Текст программы

```
/*-------Лабораторная работа N9----------------*/
/*-----"Дисковые структуры данных DOS."--------*/
/* Подключение стандартных заголовков */
#include <dos.h>
```

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
/*-----------------------------------------------*/
/* Типи и структуры данных */
#define byte unsigned char
#define word unsigned int
#define dword unsigned long
#define daddr struct DADDR
struct DADDR { /* физический дисковый адрес */
   byte h;
   word s, t, ts;
   };
struct PART { /* структура элемента раздела */
   byte Boot, Begin_Hd;
   word Begin_SecTrk;
   byte SysCode, End_Hd;
   word End_SecTrk;
   dword RelSec, Size;
   };
struct MBR
 { /* структура Главной Загрузочной Записи */
   char LoadCode[0x1be];
   struct PART rt[4];
   word EndFlag;
   };
struct BootRec
   { /* структура корневой записи */
   byte jmp[3], ident[8];
   word SectSize;
   byte ClustSize;
   word ResSect;
   byte FatCnt;
   word RootSize, TotSecs;
   byte Media;
   word FatSize, TrkSecs, HeadCnt;
   word HidnSecL, HidnSecH;
   dword LongTotSecs;
   byte Drive, reserved1, DOS4_flag;
   dword VolNum;
   char VolLabel[11], FatForm[8];
  };
struct Dir_Item
```

```
 { /* структура элемента директории */
  char fname[11];
  byte attr;
  char reserved[10];
  word time, date, cl;
 dword size;
 };
/*-----------------------------------------------*/

/* Описания функций */
void Read_Mbr(void);
/* Чтение MBR и поиск требуемого раздела       */
void Read_Boot(void); /* Чтение boot-сектора */
void Get_First(void); /* Определение абсолютного номера сектора начала логического диска    */
void Read_Fat(void);  /* Чтение FAT     */
void Read_13(void *mem);
/* Чтение сектора с омогощью  прерывания 13       */
void Sect_to_Daddr(dword sect);
/* Формирование физического дискового адреса из # сектора */
dword Clust_to_Sect(word clust);
/* Вычисление номера сектора из номера кластера    */
word Next_Clust(word clust);
/* Выборка следующего кластера  из FAT          */
char *Get_Name(char *s, char *d);
/* Выделение следующего элемента из строки-задания   */
int Find_Name(); /* Поиск имени в каталоге */
void End_of_Job(int n);  /* Завершение (при n=0-5 — аварийное)        */
/*-----------------------------------------------*/
/* Переменні */
struct PART part; /* текущий элемент раздела */
byte buff1[512];    /* буфер MBR и boot   */
struct MBR *mbr; /* указатель на таблицу разделов  */
struct BootRec *boot; /* указатель на корневую запись   */
byte buff2[512]; /* буфер каталога и текста  */
struct Dir_Item *dir; /* указатель на часть каталога  */
char *text;  /* указатель на текстовий буфер */
byte *fat;       /* указатель на FAT  */
char job[81];     /* строка-задание    */
char *jobptr;  /* текущий указатель в job */
char cname[12]; /* текущее имя для поиска  */
byte Fdisk;   /* физический номер диска   */
daddr caddr;  /* текущий дисковый адрес   */
dword sect;      /* текущий номер сектора */
word clust;      /* текущий номер кластера */
```

```
byte fat16;      /* признак формату FAT  */
dword fsize;      /* размер файла      */
int dirnum;   /* номер элемента в каталоге  */
dword FirstSect;    /* абс.сектор начала   */
byte rootdir=1; /* признак корневого каталога
           или подкаталога (1/0) */
word lastsect; /* последний сектор при чтении /
byte fatalloc=0; /* признак выделения памяти */
/*---------------------------------------------*/
main() {
 int n,i;
 textattr(14);
 clrscr();
 /* ввод имени файла */
 cprintf(" Просмотр таблицы FAT. ");
 cprintf("Укажите полное имя файла -->");
 scanf("%s",job);
 /* перевод в верхний регистр */
 strupr(job);
/* проверка правильности идентификатора диска */
 if ((!isalpha(job[0]))||(job[1]!=':')||(job[2]!='\\')) {
  printf("%c%c%c -",job[0],job[1],job[2]);
  End_of_Job(0);
 }
 textattr(10);
 clrscr();
 printf("     Лабораторная работа N9");
 printf("    Дисковые структуры данных DOS.");
 textattr(14);
 cprintf("Файл %s в FAT занимает такие кластеры :\n",job);
 jobptr=job+3;
 if (job[0]>'A') {
  /* для жесткого диска — физический номер и чтение MBR */
  Fdisk=0x80;
  Read_Mbr();
 }
else /* для гибкого диска — физический номер */
  Fdisk=job[0]-'A';
 Read_Boot(); /* чтение boot-сектора */
 Read_Fat(); /* чтение FAT */
 dir=(struct Dir_Item *)buff2;
 do { /* рух по каталогам */
  if (!rootdir) clust=dir[dirnum].cl; /* начальный кластер */
/* выделение следующего элемента из строки-задания */
  jobptr=Get_Name(jobptr,cname);
```

```
  do { /* пока не дойдем до последнего кластера */
   if (rootdir) { /* корневой каталог */
/* нач.сектор корневого кат. и количество секторов */
 sect=boot->ResSect+boot->FatSize*boot->FatCnt;
 lastsect=boot->RootSize*32/boot->SectSize+sect;
   }
   else { /* подкаталог */
    sect=Clust_to_Sect(clust);
    lastsect=boot->ClustSize+sect;
   }
 /* посекторное чтение всего корневого каталога
   или одного кластера подкаталога */
   for (; sect<lastsect; sect++) {
    Sect_to_Daddr(sect);
    Read_13(dir);
    /* поиск имени в прочитанном секторе */
    if ((dirnum=Find_Name())>=0) goto FIND;
   }
   /* до последнего кластера подкаталога */
  }
  while (clust=Next_Clust(clust));
  /* весь каталог просмотрен, а имя не найдено — ошибка */
  printf("%s -",cname);
  if (jobptr==NULL) End_of_Job(4);
  else End_of_Job(5);

FIND: /* имя найдено */
  rootdir=0;
 }
 while (jobptr!=NULL);
 /* найдено имя файла */
 /* из каталога получеем 1-й кластер */
 clust=dir[dirnum].cl;
 textattr(7);
 gotoxy(10,4);
 cprintf("Нажимайте любую клавишу ");
 cprintf(" пока не появится <КОНЕЦ ФАЙЛА>.");
 textattr(12);
 gotoxy(1,5);
 cprintf("-<НАЧАЛО ФАЙЛА>");
 gotoxy(1,6);
 cprintf("L->");
 i=0;
 do {
  i++;
```

```
  if((i%10)==0) getch();
  textattr(14+16);
  cprintf("%4x",clust);
  textattr(2);
  cprintf("--->");
 }
 while (clust=Next_Clust(clust));
 textattr(12);
 cprintf("<КОНЕЦ ФАЙЛА>\n");
 gotoxy(1,wherey());
 textattr(15+3*16);
 cprintf("Количество кластеров в файле: %u ",i);
 End_of_Job(7);
}
/*-----------------------------------------------*/
/* Чтение MBR и поиск нужного раздела */
void Read_Mbr(void) {
 int i;
 char ndrive;
 word *EndList;
 caddr.h=0;
 caddr.ts=1;
 ndrive='C';
 mbr=(struct MBR *)buff1;

NEXT: Read_13(buff1);
 for (EndList=(word *)&mbr->rt[(i=0)];
   (*EndList!=0xaa55)&&(mbr->rt[i].Size>0L);
   EndList=(word *)&mbr->rt[++i]) {
  if (mbr->rt[i].SysCode==5) {
   caddr.h=mbr->rt[i].Begin_Hd;
   caddr.ts=mbr->rt[i].Begin_SecTrk;
   goto NEXT;
  }
  if (ndrive==job[0]) {
   movmem(&mbr->rt[i],&part,sizeof(struct PART));
   return;
  }
  else ndrive++;
 }
 /* требуемый раздел не найден */
 printf("%c: -",job[0]);
 End_of_Job(1);
}
/*-----------------------------------------------*/
```

```
/* Чтение boot-сектора */
void Read_Boot(void) {
 if (Fdisk<0x80) {
  caddr.h=0;
  caddr.ts=1;
 }
 else {
  caddr.h=part.Begin_Hd;
  caddr.ts=part.Begin_SecTrk;
 }
 Read_13(buff1);
 boot=(struct BootRec *)buff1;
 Get_First();
}
/*-----------------------------------------------*/
/* Чтение FAT */
void Read_Fat(void) {
dword s, ls;
byte *f;
 fat=(byte *)malloc(boot->FatSize*boot->SectSize);
 if (fat==NULL) {
  printf("Размещение FAT -");
  End_of_Job(3);
 }
 fatalloc=1;
 s=boot->ResSect;
 ls=s+boot->FatSize;
 for (f=fat; s<ls; s++) {
  Sect_to_Daddr(s);
  Read_13(f);
  f+=boot->SectSize;
 }
 /* установление формата FAT */
 if (Fdisk>=0x80)
  if (part.SysCode==1) fat16=0;
  else fat16=1;
 else fat16=0;
}
/*-----------------------------------------------*/
/* Чтение сектора при помощи прерывания 13 */
void Read_13(void *mem) {
/* mem — адреса в ОП */
union REGS rr;
struct SREGS sr;
 rr.h.ah=2;
```

```
 rr.h.al=1;
 rr.h.dl=Fdisk;
 rr.h.dh=caddr.h;
 rr.x.cx=caddr.ts;
 sr.es=FP_SEG(mem);
 rr.x.bx=FP_OFF(mem);
 int86x(0x13,&rr,&rr,&sr);
 /* Проверка ошибок чтения */
 if (rr.x.cflag&1) {
  printf("%u -",rr.h.ah);
  End_of_Job(2);
 }
}
/*-----------------------------------------------*/
/* Определение абс.номера сектора начала лог.диска */
void Get_First(void) {
 word s, t;
 if (Fdisk<0x80) FirstSect=0;
 else {
  /* формирование # сектора из физич. дискового адреса */
  t=(part.Begin_SecTrk>>8)|((part.Begin_SecTrk<<2)&0x300);
  s=part.Begin_SecTrk&0x3f;
  FirstSect=(((dword)t*boot->HeadCnt)+part.Begin_Hd)*
  boot->TrkSecs+s-1;
 }
}
/*-----------------------------------------------*/
/* Формирование физического дискового адреса из # сектора */
void Sect_to_Daddr(dword sect) {
/* sect — номер сектора, caddr — адрес на диске */
dword s;
 if (Fdisk>=0x80) sect+=FirstSect;
 caddr.s=sect%boot->TrkSecs+1;
 s=sect/boot->TrkSecs;
 caddr.h=s%boot->HeadCnt;
 caddr.t=s/boot->HeadCnt;
 caddr.ts=(caddr.t<<8)|caddr.s|((caddr.t&0x300)>>2);
}
/*-----------------------------------------------*/
/* Вычисление номера сектора из номера кластера */
dword Clust_to_Sect(word clust) {
/* clust — номер кластера, возвращает номер сектора */
 dword ds, s;
 ds=boot->ResSect+boot->FatSize*boot->FatCnt+
 boot->RootSize*32/boot->SectSize;
```

```
 s=ds+(clust-2)*boot->ClustSize;
 return(s);
}
/*-------------------------------------------------*/
/* Выборка следующего кластера из FAT */
word Next_Clust(word clust) {
/* clust — номер кластера, возвращает номер следующего кластера
  или 0 — если следующего нет */
 word m, s;
 if (rootdir) return(0);
 if (!fat16) {
  m=(clust*3)/2;
  s=*(word *)(fat+m);
  if(clust%2)   /* нечетный элемент */
   s>>=4;
  else  /* четный элемент */
   s=s&0x0fff;
  if (s>0x0fef) return(0);
  else return(s);
 }
 else {
  m=clust*2;
  s=*(word *)(fat+m);
  if (s>0xffef) return(0);
  else return(s);
 }
}
/*-------------------------------------------------*/
/* Выделение следующего элемента из строки-задания */
char *Get_Name(char *s, char *d) {
/*  s  —  строка  задания,  d  —  выделенный  элемент,  возвращает
указатель на новое начало строки задания. */
char *p,*r;
int i;
 for(i=0;i<11;d[i++]=' ');
 d[11]='\0';
 if ((p=strchr(s,'\\'))==NULL) {
  /* последний элемент строки — имя файла */
  /* перезапись имени */
  for(r=s,i=0; (i<8)&&*r&&(*r!='.'); i++,r++) *(d+i)=*r;
  /* перезапись расширения */
  if (*r) for(i=0,r++; (i<3)&&*r; i++,r++) *(d+8+i)=*r;
  return(NULL);
 }
 else {
```

```
  /* следующий элемент — имя подкаталога */
  *p='\0';
  for(r=s,i=0; (i<11)&&*r; i++,r++) *(d+i)=*r;
  return(p+1);
 }
}
/*-----------------------------------------------*/
/* Поиск имени в каталоге */
int Find_Name() {
 int j;

/* cname — найденное имя; возвращает индекс найденного
  элемента в массиве dir или (-1) */
  for (j=0; j<boot->SectSize/sizeof(struct Dir_Item); j++) {
   if (dir[j].fname[0]=='\0') {
  /* конец использованных элементов каталога */
    printf("%s -",cname);
    if (jobptr==NULL) End_of_Job(4);
    else End_of_Job(5);
   }
   if ((byte)dir[j].fname[0]!=0xe5) {
    if (memcmp(dir[j].fname,cname,11)==0) {
     /* если ім‘я збігатся, то:
- при поиске файла элемент не должен иметь атрибутов "подкаталог" или "метка тома",
- при поиске подкаталога элемент должен иметь атрибут "подкаталог" */
      if (jobptr==NULL)
       if ( !(dir[j].attr&0x18) ) return(j);
      else
       if (dir[j].attr&0x10) return(j);
    }
   }
  }
 return(-1);
}
/*-----------------------------------------------*/
/* Завершение (при n=0-5 — аварийное) */
void End_of_Job(int n) {
/* n — номер сообщения */
static char *msg[] = {
    "неправильный идентификатор диска",
    "логический диск отсутствует",
    "ошибка чтения",
    "нехватка памяти",
```

```
   "подкаталог не найден",
   "файл не найден",
   "непредусмотренный конец файла",
   "" };
 /* освобождение памяти */
 if (fatalloc) free(fat);
 /* выдача сообщения */
 textattr(12+128);
 cprintf(" %s\n",msg[n]);
 gotoxy(28,wherey());
 cprintf(" Нажмите любую клавишу...\n");
 textattr(7);
 getch();
 /* завершение программы */
 exit(0);
}
```

Результаты работы программы

В процессе работы программы на экран выводится информация наподобие следующей:

```
Лабораторная работа N9
Дисковые структуры данных DOS.
Файл D:\TC\TC.EXE в FAT занимает такие кластеры:
Нажимайте любую клавишу пока не появится <КОНЕЦ ФАЙЛА>.
-<НАЧАЛО ФАЙЛА>
8L->2410--->2411--->2412--->2413--->2414--->2415--->2416--->2417-
-->2418--->2419--->241a--->241b--->241c--->241d--->241e--->241f-
-->2420--->2421--->2422--->2423--->2424--->2425--->2426--->2427-
-->2428--->2429--->242a--->242b--->242c--->242d--->242e--->242f-
-->2430--->2431--->2432--->2433--->2434--->2435--->2436--->2437-
-->2438--->2439--->243a--->243b--->243c--->243d--->243e--->243f-
-->2440--->2441--->2442--->2443--->2444--->2445--->2446--->2447-
-->2448--->2449--->244a--->244b--->244c--->244d--->244e--->244f-
-->2450--->2451--->2452--->2453--->2454--->2455--->2456--->2457-
-->2458--->2459--->245a--->245b--->245c--->245d--->245e--->245f-
-->2460--->2461--->2462--->2463--->2464--->2465--->2466--->2467-
-->2468--->2469--->246a--->246b--->246c--->246d--->246e--->246f-
-->2470--->2471--->2472--->2473--->2474--->2475--->2476--->2477-
-->2478--->2479--->247a--->247b--->247c--->247d--->247e--->247f-
-->2480--->2481--->2482--->2483--->2484--->2485--->2486--->2487-
-->2488--->2489--->248a--->248b--->248c--->248d--->248e--->248f-
-->2490--->2491--->2492--->2493--->2494--->2495--->2496--->2497-
-->2498--->2499--->249a--->249b--->249c--->249d---> <КОНЕЦ ФАЙЛА>
Количество кластеров в файле: 142
Нажмите любую клавишу...
```

Лабораторная работа № 10
Тема: Управление программами

Цели и задачи урока:
Изучение принципов управления программами в MS DOS и приобретение практических навыков работы с префиксом программного сегмента и его полями.
Тип урока – практическое занятие.
Методы обучения – репродуктивный
Форма: фронтальная
Обеспечение занятия- ПК, тетради.
Ход урока

Постановка задачи

Разработать программу, производящую форматный вывод на печать своего Префикса Программного Сегмента.

Литература

1. Разработка модулей программного обеспечения для компьютерных систем: учебник для студ. учреждений сред. проф. образования / Г. Н. Федорова. - 4-е изд., перераб. - М.: Издательский центр «Академия», 2020. - 384 с. ISBN 978-5-4468-9443-7
2. http://www.uhlib.ru/kompyutery_i_internet/sistemnoe_programmirovanie_v_srede_windows/index.php (дата обращения: 25. 05. 2021г.)

Пример решения задачи

Структура программы

Программа состоит из основной программы и двух функций:

void get_DOS_version_h(void) — функция, возвращающая в глобальной переменной dos_ver старшее число номера версии DOS.

void addr_PSP (void) — функция, получающая сегментный адрес префикса программного сегмента программы и возвращающая его в глобальной переменной pid.

Описание переменных

Переменные, глобальные для всей программы:

p_psp — указатель на структуру struct PSP,

pid — сегментный адрес PSP;

dos_ver — старшее число номера версии DOS;

i — вспомогательная переменная, используемая для просмотра таблицы файлов задачи (JFT), которая представляет собой массив из 20 элементов (хотя возможно, что их число отлично от 20, поэтому размер массива определим из поля JFT_size);

l — переменная, используемая для вывода содержимого сегмента окружения DOS и определения числа строк вызова (для версии DOS 3.0 и выше);

s — переменная, которая вначале используется как указатель на таблицу файлов задачи, затем на строки сегмента окружения и строки вызова;

rr — переменная, которая используется для задания значений регистров общего назначения при вызове прерывания.

Описание алгоритма программы

Данная программа производит распечатку основных полей своего PSP. Для этого префикс программного сегмента представим в виде следующей структуры:

```
struct psp
{    /* ФОРМАТ PSP */
byte ret_op[2];    /* команда INT 20h   */
```

```
word end_of_mem;  /* вершина доступной памяти*/
byte reserved1;
byte old_call_dos[5]; /* старый вызов DOS  */
void *term_ptr; /* адрес завершения     */
void *ctrlbrk_ptr;  /* адрес обработчика Ctrl+Break  */
void *criterr_ptr;  /* адрес обработчика крит.ошибок */
word father_psp;   /* PID родителя     */
byte JFT[20];  /* таблица файлов программы */
word env_seg;     /* адрес окружения    */
void *stack_ptr;   /* адрес стека       */
word JFT_size;    /* размер таблицы файлов */
byte *JFT_ptr;    /* адрес таблицы файлов  */
byte reserved2[24];
byte new_call_dos[3]; /* новый вызов DOS   */
} *p_psp;
```

Поле ret_op используется для возможного завершения программы по команде RET 0, поле old_call_dos, содержит команду вызова диспетчера функций DOS. Обращение к этому полю в программе может использоваться вместо команды INT 21h, но в современных версиях DOS для этих целей лучше обращаться к полю new_call _dos.

Поле end_of_mem содержит сегментный адрес конца доступной памяти в системе. В три поля: term_ptr, ctrlbrk_ptr, criterr_ptr DOS при загрузке программы копирует содержимое векторов прерываний: 22h, 23h, 24, представляющее собой адреса обработчиков: завершения программы, комбинации клавиш Ctrl+Break, критической ошибки — соответственно. Предполагается, что программа может свободно перенаправить эти векторы на собственные обработчики соответствующих ситуаций, но от забот по восстановлению векторов программа избавляется, так как при ее завершении DOS сама восстанавливает векторы из соответствующих полей PSP завершаемой программы. Для аналогичных целей предназначено и поле stack_ptr — в нем сохраняется (а при завершении — из него восстанавливается) адрес стека, использовавшегося до вызова программы. Поле, именуемое father_psp, содержит сегментный адрес PSP родителя — программы, запустившей данную программу, обычно родителем является COMMAND.COM.

При загрузке программы DOS, кроме программного сегмента, создает для нее еще и сегмент окружения. Сегмент окружения содержит ASCIIZ-строки, задающие значения некоторых глобальных переменных, эти значения могут устанавливаться командой DOS SET, они доступны командным файлам и — через PSP — программам. Набор строк окружения заканчивается пустой ASCIIZ-строкой (нулем). В DOS 3.0 и выше за ним следует еще 2-байтное число строк вызова (обычно 1) и далее — строка (или строки) вызова программы. Обычно в первую (до строк вызова) часть порождаемой программы копируется содержимое окружения программы-родителя. Программа имеет доступ к своему сегменту окружения через поле env_seg PSP, содержащее сегментный адрес окружения.

Поле JFT (Job File Table — Таблица Файлов Задачи) представляет собой массив из 20 однобайтных элементов. При открытии программой файла DOS формирует для него блок-описатль в системной таблице файлов и помещает ссылку на него (его номер) в свободный элемент JFT. Дескриптор файла, возвращаемый программе DOS при открытии файла, является номером элемента в JFT. При запуске программы первые пять элементов создаваемой для нее

JFT содержат ссылки на системные файлы, остальные свободны. При обработке JFT DOS использует не прямое обращение к полю JFT PSP, а косвенное — через поле JFT_ptr, а в качестве ограничителя размера JFT — не константу 20, а значение поля JFT_size PSP.
После всего, сказанного выше, не составляет труда написать программу, осуществляющую форматный вывод своего префикса программного сегмента. Для в начале необходимо определить версию DOS (с помощью функции get_DOS_version_h() и получить адрес PSP (с помощью функции addr_PSP()).
Функция get_DOS_version_h() определяет старшее число номера версии DOS, используя для этого функцию DOS 30h (прерывание 21h), которая возвращает в регистре AL старшее число номера версии, а в регистре AH — младшее число. Нас интересует только значение регистра AL.
Функция addr_PSP() возвращает сегментный адрес PSP путем использования функции DOS 62h:
Вход: AH = 62h
Выход: BX = сегментный адрес PSP текущего процесса

Текст программы

```
/*-----Лабораторная работа N10----------------*/
/*------"Управление программами"----------------*/
/* Подключение стандартных заголовков */
#include <dos.h>
#include <conio.h>
/* Типы данных */
#define byte unsigned char
#define word unsigned int

/* Описание функций */
void get_DOS_version_h(void); /* Определение версии DOS */
void addr_PSP (void);     /* Получение адреса PSP  */

struct psp
{    /* ФОРМАТ PSP */
byte ret_op[2];    /* команда INT 20h   */
word end_of_mem; /* вершина доступной памяти */
 byte reserved1;
 byte old_call_dos[5]; /* старый вызов DOS  */
 void *term_ptr;    /* адрес завершения   */
 void *ctrlbrk_ptr;  /* адрес обработчика Ctrl+Break  */
 void *criterr_ptr;  /* адрес обработчика крит.ошибок */
 word father_psp;   /* PID родителя      */
 byte JFT[20]; /* таблица файлов программы */
 word env_seg;     /* адрес окружения   */
 void *stack_ptr;   /* адрес стека       */
 word JFT_size; /* размер таблицы файлов  */
 byte *JFT_ptr;   /* адрес таблицы файлов  */
 byte reserved2[24];
```

```
 byte new_call_dos[3]; /* новый вызов DOS   */
 } *p_psp;

word pid;       /* сегм.адрес PSP    */
int dos_ver,      /* версия DOS       */
  i, l, j;
char *s;
union REGS rr;

main()
{
 textbackground(0);
 clrscr();
 textattr(0x0a);
 cprintf("---------------");
 cprintf("   Лабораторная работа N10    ");
 cprintf("---------------");
 cprintf("---------------");
 cprintf("       Управление программами  ");
 cprintf("---------------");
 textcolor(11);
 get_DOS_version_h();
 addr_PSP();
 /* распечатка PSP */
 cprintf("\n\n      Адрес PID = %04X\n\n\r",pid);
 p_psp=(struct psp *)MK_FP(pid,0);
 textcolor(10);
 cprintf("Команды:\n\r");
 cprintf("--------\n\r");
 textcolor(14);
 cprintf("   Завершение — int 20h:");
 textcolor(12);
 cprintf(" %02X %02X\n\r",p_psp->ret_op[0],p_psp->ret_op[1]);
 textcolor(14);
 cprintf("   Старый вызов DOS:   ");
 textcolor(12);
 for (i=0;i<5;cprintf("%02X ",p_psp->old_call_dos[i++]));
 textcolor(14);
 cprintf("\n\r   Новый вызов DOS:    ");
 textcolor(12);
 for(i=0;i<3;cprintf("%02X ",p_psp->new_call_dos[i++]));
 textcolor(10);
 cprintf("\n\n\rАдреса:\n\r");
 cprintf("-------\n\r");
 textcolor(14);
```

```
cprintf("  Конец памяти:       ");
textcolor(12);
cprintf("%04X:0000\n\r",p_psp->end_of_mem);
textcolor(14);
cprintf("  Обработчик завершения:  ");
textcolor(12);
cprintf("%Fp\n\r",p_psp->term_ptr);
textcolor(14);
cprintf("  Обработчик Ctrl+Break:  ");
textcolor(12);
cprintf("%Fp\n\r",p_psp->ctrlbrk_ptr);
textcolor(14);
cprintf("  Обработчик критич.ошибки: ");
textcolor(12);
cprintf("%Fp\n\r",p_psp->criterr_ptr);
textcolor(14);
cprintf("  Стек:           ");
textcolor(12);
cprintf("%Fp\n\n\r",p_psp->stack_ptr);
textcolor(14);
cprintf("\n\rРодитель: ");
textcolor(12);
cprintf("%04X ",p_psp->father_psp);
textcolor(0x8b);
cprintf("\n\n\rНажмите любую клавишу ...\n\r\7");
getch();
clrscr();
textattr(0x0a);
cprintf("---------------");
cprintf("       Лабораторная работа N10       ");
cprintf("---------------");
cprintf("---------------");
cprintf("       Управление программами       ");
cprintf("---------------");
/* Распечатка таблицы файлов */
s=p_psp->JFT_ptr;
textcolor(10);
cprintf("\n\n\rТаблица файлов: ");
textcolor(12);
cprintf("%Fp (%d) ",s,p_psp->JFT_size);
textcolor(11);
if (s==(byte *)p_psp+0x18)
 cprintf(" — в этом же PSP");
cprintf("\n\r");
for (i=0; ++i<=p_psp->JFT_size; cprintf("%d ",*(s++)));
```

```
 textcolor(10);
 cprintf("\n\n\rОкружение DOS: ");
 textcolor(12);
 cprintf("%04X\n\r",p_psp->env_seg);
 s=(char *)MK_FP(p_psp->env_seg,0);
 textcolor(11);
 while(l=strlen(s))
 {
  cprintf("   %s\n\r",s);
  s+=l+1;

 }
 if (dos_ver>2)
 {
  /* для DOS 3.0 и дальше можно получить строку вызова */
  s++;
  l=*((int *)s);
  textcolor(10);
  cprintf("\n\rЧисло строк вызова: ");
  textcolor(12);
  cprintf("%d\n\r",l);
  s+=2;
  textcolor(11);
  for(i=0; i<l; i++)
  {
   cprintf("%s\n\r",s);
   s+=strlen(s)+1;
  }
 }
 textattr(0x8b);
 cprintf("\n\n\n\n\rНажмите любую клавишу ...\7");
 textattr(0x07);
 cprintf("\n\r");
 getch();
 clrscr();
}

/* Определение версии DOS */
void get_DOS_version_h(void)
{
 rr.h.ah=0x30;
 intdos(&rr,&rr);
 dos_ver=rr.h.al;
}
```

```
/* Получение адреса PSP */
void addr_PSP (void)
{
 rr.h.ah=0x62;
 intdos(&rr,&rr);
 pid=rr.x.bx;
}
```

Результаты работы программы

В процессе работы программы на экран была выведена следующая информация:

```
--------------------------------------------------
--       Лабораторная работа N10       ------
Управление программами       ----
     Адрес PID = 0BA0

Команды:
--------
   Завершение — int 20h: CD 20
   Старый вызов DOS:   9A F0 FE 1D F0
   Новый вызов DOS:    CD 21 CB

Адреса:
-------
  Конец памяти:       9FC0:0000
  Обработчик завершения:  0AFA:02B1
  Обработчик Ctrl+Break:  0AFA:014A
  Обработчик критич.ошибки: 0AFA:0155
  Стек:          0E04:0F94

Родитель: 0AFA

Таблица файлов: 0BA0:0018 (20) — в этом же PSP
1 1 1 0 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

Окружение DOS: 0A1E
   CONFIG=STD
   COMSPEC=C:\DOS\COMMAND.COM
   PROMPT=$p$g
   PATH=D:\WIN;C:\;C:\DOS;C:\ARH;C:\NC;C:\BAT;D:\TP; D:\TP7;D:\BC\BIN
   TEMP=d:\~TMP

Число строк вызова: 1
D:\TC\TC_LAB10.EXE
```