ФЕДЕРАЛЬНОЕ АГЕНТСТВО ВОЗДУШНОГО ТРАНСПОРТА

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ГРАЖДАНСКОЙ АВИАЦИИ» (МГТУ ГА)

Кафедра вычислительных машин, комплексов, систем и сетей

Н.И. Черкасова

# ПРОГРАММИРОВАНИЕ НА МАШИННО-ОРИЕНТИРОВАННЫХ ЯЗЫКАХ

**Учебно-методическое пособие**
по выполнению лабораторной работы № 4

*для студентов II курса*
*направления 09.03.01*
*очной формы обучения*

Москва
ИД Академии Жуковского
2020

УДК 004.43
ББК 6Ф7.3
Ч-48

Рецензент:
*Феоктистова О.Г.* – д-р техн. наук, доцент

**Черкасова Н.И.**

Ч-48 Программирование на машинно-ориентированных языках [Текст] : учебно-методическое пособие по выполнению лабораторной работы № 4 / Н.И. Черкасова. – М.: ИД Академии Жуковского, 2020. – 36 с.

Данное учебно-методическое пособие издается в соответствии с рабочей программой учебной дисциплины «Программирование на машинно-ориентированных языках» по учебному плану для студентов II курса направления 09.03.01 очной формы обучения

Рассмотрено и одобрено на заседаниях кафедры 29.08.2020 г. и методического совета 29.08.2020 г.

**УДК 004.43**
**ББК 6Ф7.3**

*В авторской редакции*

Подписано в печать 16.12.2020 г.
Формат 60x84/16 Печ. л. 2,25 Усл. печ. л. 2,09
Заказ № 687/1008-УМП05 Тираж 90 экз.

Московский государственный технический университет ГА
125993, Москва, Кронштадтский бульвар, д. 20

Издательский дом Академии имени Н. Е. Жуковского
125167, Москва, 8-го Марта 4-я ул., д. 6А
Тел.: (495) 973-45-68
E-mail: zakaz@itsbook.ru

© Московский государственный технический университет гражданской авиации, 2020

Содержание

**ЛАБОРАТОРНАЯ РАБОТА№4**
**Программирование приложений для ОС Windows**

## 1. Цель лабораторной работы

Целью лабораторной работы является освоение:

1. Программирование приложений для ОС Windows на Ассемблере.
2. Структуры и состав программы
3. Приемов работы с использованием средств Ассемблера для программирования приложений для ОС Windows
4. Приемов работы с использованием подпрограмм – процедур. Вызов и возврат.
5. Составление приложений для ОС Windows на Ассемблере.

## 2.Содержание отчёта

Отчет по лабораторной работе должен включать:
1) цель лабораторной работы;
2) конкретный вариант задания на выполнение;
3) тексты программ;
4) схемы алгоритмов;
5) результаты выполнения программ.

**3. Краткие теоретические сведения**.

Особенности приложений для ОС Windows в ассемблере.

Программный модуль на ассемблере обладает, как правило, более высоким быстродействием, надежностью и минимальным объёмом используемых ресурсов, чем на языке высокого уровня. Это связано с меньшим числом команд, требуемых для реализации фрагмента кода, ведь чем меньше выполняется центральным процессором команд, тем производительней программа.

Отметим, что совместное использование ассемблера с языками высокого уровня представляется тремя механизмами:

1. на основе объединения объектных модулей (раннее связывание);
2. на основе динамических библиотек (позднее связывание);
3. на основе встроенного языка ассемблера.

Интеграция ассемблера с языками высокого уровня в свою очередь характеризуется:

1. согласованием имен
2. согласованием параметров
3. согласованием вызовов.

В ОС Windows для обеспечения взаимодействия различных процессов и потоков в приложении используется механизм обработки сообщений. Для того чтобы иметь возможность работать с каким-либо устройством, например, с клавиатурой или мышью, программам ОС DOS приходилось отслеживать состояние этих устройств и ожидать их реакции на посланные им сообщения. ОС Windows управляется сообщениями, и уже не программа ожидает реакции от устройства, а сообщение о реакции устройства запускает ту или иную программу. Та часть программы, которая запускается в ответ на конкретное сообщение, называется функцией его обработки. Большую часть работы по передаче сообщений и вызову соответствующих функций обработки берут на себя внутренние процедуры Windows.

Отметим некоторые особенности приложения для Windows.

Windows-приложения выполняются в собственных окнах. Каждое приложение располагает, по крайней мере, одним собственным окном. Через окна приложения выполняется ввод/вывод информации пользователя. Главное окно – это и есть само приложение, но окно – это также и визуальный интерфейс.

Работа в ОС Windows ориентирована на события. В Windows приложения выполняются пошагово. После решение одной подзадачи, управление возвращаетсясистеме, которая может вызывать другие программы. Windows переключается между различными приложениями. Программист инициирует событие (вызов команды меню, щелчок мыши на окне), событие обрабатывается, и программное управление передается в соответствующее приложение. Приложение вызывается для обработки события. Таким образом, разработка приложения – это создание окна

приложения (создать окно, зарегистрировать его класс, сделать его видимым) и организация обработки сообщений пользователя.
Написать простое приложение с фамилией и номером Группы.

### 3.1. Программирование приложений для ОС Windows

#### 3.1.1. Окно приложения.

ОС Windows поддерживает 32/64-битный интерфейс программирования Win32 API- (Application Programming Interface – интерфейс прикладного программирования). API- это набор, похожих на подпрограммы процедур - функций, которые программы вызывают для решения всех задач, связанных с работой ОС. Реализованы они в виде библиотек динамической компоновки .dll, основными из которых являются gdi, user, kernel. Эти библиотеки отображаются в адресное пространство каждого процесса.

Windows-приложения выполняются в собственных окнах или в окнах, представляемых операционной системой. Каждое приложение располагает, по крайней мере, одним собственным окном.

#### 3.1.2. Два типа приложений Windows

Операционная система Windows поддерживает два типа приложений: оконные и консольные.

Оконное приложение строится на базе набора функций API, составляющих графический интерфейс пользователя (Graphic User Interface, GUI). Оконное приложение представляет собой программу, которая весь вывод на экран производит в графическом виде. Первым результатом работы оконного приложения является отображение на экране специального объекта - окна. После того как окно появилось на экране, вся работа приложения направлена на то, чтобы поддерживать его в актуальном состоянии.

Main.asm

```
.386
.model flat, stdcall
option casemap :none
include MASM32\INCLUDE\windows.inc
include MASM32\INCLUDE\user32.inc
include MASM32\INCLUDE\kernel32.inc

includelib MASM32\LIB\user32.lib
includelib MASM32\LIB\kernel32.lib

.data
msg  db "[redacted] ЭВМ 2-1",0
ttl     db "MessageBox",0

.code
start:

    invoke MessageBox,NULL,ADDR msg,ADDR ttl,MB_OK
    invoke ExitProcess,0

end start
```

Рис 1 - Простое Windows приложение


Рис 2 – Результат выполнения программы

Так называемое консольным, приложение представляет собой программу, работающую в текстовом режиме. Работа консольного приложения напоминает работу программы MS-DOS. Но это лишь внешнее впечатление. Поддержка работы консольного приложения обеспечивается специальными функциями Windows.

Вся разница между двумя типами Windows-приложений состоит в том, с каким типом информации они работают. Отметим, что безусловно, основным типом приложений в ОС Windows являются оконные.

### 3.1.3. Сообщения в ОС Windows

В Windows программа пассивна. После запуска она ждет, когда ей уделит внимание операционная система. Операционная система делает это посылкой специально оформленных групп данных, называемых сообщениями. Сообщения могут быть разного типа, они функционируют в системе довольно хаотично, и приложение не знает, какого типа сообщение придет следующим. Отсюда следует, что логика построения Windows-приложения должна обеспечивать корректную и предсказуемую работу при поступлении сообщений любого типа.

Итак, работа в Windows ориентирована на события. В Windows приложения выполняются пошагово. После решение одной подзадачи, управление возвращается Windows, которая может вызывать другие программы. Windows переключается между различными приложениями. Программист инициирует событие (вызов команды меню, щелчок мыши на окне), событие обрабатывается, и программное управление передается в соответствующее приложение. Приложение вызывается для обработки события.

В ОС Windows для обеспечения взаимодействия различных процессов и потоков в приложении используется механизм обработки сообщений. Для того чтобы иметь возможность работать с каким-либо устройством, например, с клавиатурой или мышью, программам DOS приходилось отслеживать состояние этих устройств и ожидать их реакции на посланные им сообщения.

ОС Windows управляется сообщениями, и уже не программа ожидает реакции от устройства, а сообщение о реакции устройства запускает ту или иную программу. Та часть программы, которая запускается в ответ на конкретное сообщение, называется функцией его обработки. Большую часть

работы по передаче сообщений и вызову соответствующих функций обработки берут на себя внутренние процедуры Windows.

Таким образом, разработка приложения – это создание окна приложения (создать окно, зарегистрировать его класс, сделать его видимым) и организация обработки сообщений пользователя.

### 3.2. Разработка Windows-приложений

. Операционные системы MS-DOS и Windows поддерживают две совершенно разные идеологии программирования. В чем разница? Программа DOS после своего запуска должна быть постоянно активной. Если ей, к примеру, требуется получить очередную порцию данных с устройства ввода-вывода, то она сама должна выполнять соответствующие запросы к операционной системе. При этом программа ОС DOS работает по определенному алгоритму, она всегда знает, что и когда ей следует делать.

В Windows все наоборот. В Windows программа пассивна. После запуска она ждет, когда ей уделит внимание операционная система. Операционная система делает это посылкой специально оформленных групп данных, называемых сообщениями. Сообщения могут быть разного типа, они функционируют в системе довольно хаотично, и приложение не знает, какого типа сообщение придет следующим. Отсюда следует, что логика построения Windows-приложения должна обеспечивать корректную и предсказуемую работу при поступлении сообщений любого типа. Тут можно провести определенную аналогию между механизмом сообщений Windows и механизмом прерываний в архи-Программирование оконных Windows-приложений в текстуре IBM PC. Для нормального функционирования своей программы программист должен уметь эффективно использовать функции интерфейса прикладного программирования (Application Program Interface, API) операционной системы.

### 3.3. Программирование различных оконных Windows-приложений

Любое оконное Windows-приложение имеет типовую структуру, основу которой составляет так называемое каркасное приложение, содержащее минимально необходимый для функционирования полноценного Windows-приложения программный код. Не случайно во всех источниках в качестве первого Windows-приложения рекомендуется изучать и исследовать работу некоторого каркасного приложения, так как именно оно отражает основные особенности взаимодействия программы с операционной системой Windows. Более того, написанное и однажды отлаженное каркасное Windows-приложение используется и в дальнейшем в качестве основы для написания любого другого значительно более сложного приложения.

#### 3.3.1. Каркасное Windows-приложение

Минимальное приложение Windows состоит из трех частей:

- главной функции;
- цикла обработки сообщений;
- оконной функции.

Выполнение любого оконного Windows-приложения начинается с главной функции. Она содержит код, осуществляющий настройку (инициализацию) приложения в среде Windows. Видимым для пользователя результатом работы главной функции является появление на экране графического объекта в виде окна. Последним действием кода главной функции является создание цикла обработки сообщений. После его создания приложение становится пассивным и начинает взаимодействовать с внешним миром посредством специальным образом оформленных данных - сообщений. Обработка поступающих приложению сообщений осуществляется специальной функцией, называемой оконной.

Оконная функция уникальна тем, что может быть вызвана только из операционной системы, а не из приложения, которое ее содержит (функция обратного вызова). Тело оконной функции имеет определенную структуру. Таким образом, Windows-приложение, как минимум, должно состоять из трех перечисленных элементов.

Одним из главных критериев выбора языка разработки Windows-приложения является наличие в нем средств, способных поддержать строго определенную последовательность шагов. Язык ассемблера является универсальным языком и пригоден для реализации любых задач, поэтому можно смело предположить, что на нем можно написать также любое Windows-приложение. Но мало написать сам текст Windows-приложения, необходимо знать средства пакета транслятора, специально предназначенные для разработки таких приложений, и уметь пользоваться этими средствами

Далее представлен пример программы, создающей одно главное окно и два окна — дочернее и собственное с использованием API-функциий, то есть полноценное Windows-приложения( листинг представлен в приложении1)

Следует обратить внимание на один нетривиальный момент. В программе имеются три процедуры окна, и в каждой присутствуют метки, имеющие одинаковые названия. Транслятор MASM32 автоматически считает все метки процедуры локальными, т. е. при трансляции расширяет их имена до уникальных. Таким образом, внутри процедуры мы можем свободно пользоваться переходами, не боясь, что переход будет осуществлен в другую процедуру.

Для каждого из трех появляющихся окон определена своя процедура обработки сообщений. Эти процедуры обработки сообщений не содержат команды PostQuitMessage. Это и понятно: закрытие дочернего или собственного окна не должно вызывать выход из программы.

Во всем остальном содержимое процедуры этих окон может быть таким же, как и для главного окна (окна верхнего уровня). Они могут содержать элементы управления, иметь заголовки, обрабатывать любые сообщения и т. д.

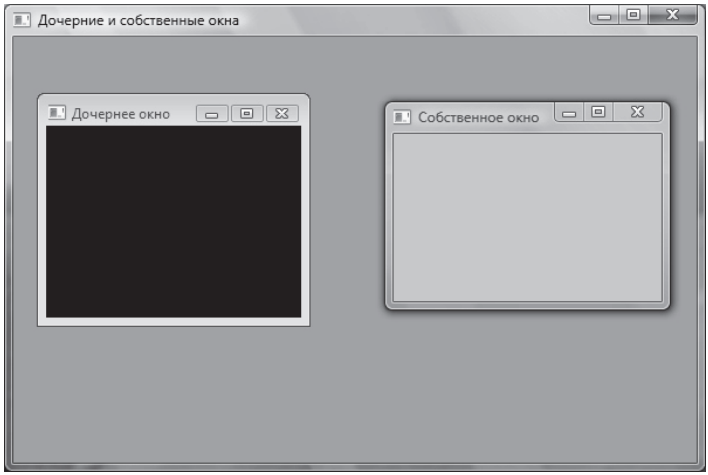


Рис3 - Программа, создающей одно главное окно и два окна — дочернее и собственное с использованием API-функциий

**3.3.2. Структура приложения**

В отличие от ОС DOS, каждую программу ОС Windows запускает в отдельном виртуальном адресном пространстве. Это означает, что теперь память - это большое последовательное 4х гигабайтовое пространство или более для 64 разрядных ОС. При этом, безусловно ,содержимое сегментных регистров нельзя изменять по своему усмотрению.

При программировании под Win32 следует помнить, что Windows использует регистры esi, edi, ebp и ebx для определенных целей и, что значение этих регистров нельзя изменить произвольно. Если же какой-либо из этих четырех регистров используется в вызываемой функции, то их необходимо восстановить перед возвращением управления Windows.

Рассмотрим один из шаблонов программы:

Листинг1.

```
.386
.MODEL Flat, STDCALL
.DATA
  < инициализированные данные>
.DATA?
  < неинициализированные данные>
.CONST
  < константы>
.CODE
  <метка>
  < код>
end <метка>
```

Или более сокращенный вариант:

Листинг 2.

```
.686P
.model flat
.code
star:
;тело программы
ret
.data
;данные программы
end star
```

Рассмотрим каркас программы Листинга1.

**.386**

Это ассемблерная директива, указующая ассемблеру использовать набор операций для процессора 80386. Можно использовать и .486, .586, но на начальном этапе самый безопасный выбор - это указывать .386. Также есть два практически идентичных выбора для каждого варианта CPU. .386/.386p, .486/.486p. Эти "p"-версии необходимы только в тех случаях, когда программа использует привилегированные инструкции, то есть инструкции, зарезервированные процессором/операционной системой для защищенного режима. Они могут быть использованны **только** в защищенном коде, например, драйверами. Разрабатываемые в данном курсе программы будут работать в непривилигированном режиме, так что лучше использовать не-"p" версии. Подробное описание директив см. Табл 1

.MODEL FLAT, STDCALL

.MODEL - это ассемблерная директива, определяющая модель памяти программы см. Табл 2. Под Win32 есть только одна модель - плоская. STDCALL говорит Компилятору о порядке передачи параметров, слева направо или справа налево, а также о том, кто очищает стек после того как функция вызвана.

Под Win16 существует два типа передачи параметров, C и PASCAL. По C-договоренности, параметры передаются справа налево, то есть самый правый параметр кладется в стек первым. PASCAL-передача параметров - это C-передача наоборот. Согласно ей, параметры передаются слева направо и вызываемый должен очищать стек. Win16 использует этот порядок передачи данных, потому что тогда код программы становится меньше. C-порядок полезен, когда вы не знаете, как много параметров будут переданы функции, как например, в случае wsprintf(), когда функция не может знать заранее, сколько параметров будут положены в стек, так что она не может его очистить. STDCALL - это гибрид C и PASCAL.

Согласно ему, данные передаются справа налево, но вызываемый ответственнен за очистку стека. Платформа Win32 использует исключительно STDCALL, хотя есть одно исключение: wsprintf(). В последнем случае вы должны следовать сишному порядку вызова.

.DATA .DATA?

.CONST
.CODE
Все четыре директивы – предназначены для создания того, что называется **секциями**. В Win32 нет сегментов, но адресное пространство делится на логические секции. Начало одной секции отмечает конец предыдущей. Есть две группы секций: **данных и кода**.
.DATA - Эта секция содержит инициализированные данные программы.
.DATA? - Эта секция содержит неинициализированные данные программы. Иногда нужно только «предварительно» выделить некоторое количество памяти, но не требуется инициализировать ее. Эта секция для этого и предназначается. Преимущество неинициализированных данных следующее: они не занимают места в исполняемом файле. Например, если выделить 10.000 байт в вашей .DATA? секции, exe-файл не увеличится на 10kb. Его размер останется таким же. Вы всего лишь говорите компилятору, сколько места вам нужно, когда программа загрузится в память.
.CONST - Эта секция содержит объявления констант, используемых программой. Константы не могут быть изменены ей.
Не обязательно задействовать все три секции. Объявляйте только те, которые хотите использовать.
Есть только одна секция для кода: .CODE, там где содержится весь код.

Таблица 1-Директива описания типа микропроцессора

| Директива | Назначение |
|---|---|
| 8086 | Разрешены инструкции базового процессора i8086 (и идентичные им инструкции процессора i8088). Запрещены инструкции более поздних процессоров. |
| .186 .286 .386 .486 .586 .686 | Разрешены инструкции соответствующего процессора x86 (x=1,…,6). Запрещены инструкции более поздних процессоров. |
| .187 .287 .387 .487 .587 | Разрешены инструкции соответствующего сопроцессора x87 наряду с инструкциями процессора x86. Запрещены инструкции более поздних процессоров и сопроцессоров. |
| .286c .386c .486c .586c .686c | Разрешены НЕПРИЛЕГИРОВАННЫЕ инструкции соответствующего процессора x86 и сопроцессора x87. Запрещены инструкции более поздних процессоров и сопроцессоров. |
| .286p .386p .486p .586p .686p | Разрешены ВСЕ инструкции соответствующего процессора x86, включая привилегированные команды и инструкции сопроцессора x87. Запрещены инструкции более поздних процессоров и сопроцессоров. |
| .mmx | Разрешены инструкции MMX-расширения. |
| .xmm | Разрешены инструкции XMM-расширения. |
| .K3D | Разрешены инструкции AMD 3D. |

Таблица2 -Директива .model

| Модель памяти | Количество и размер сегментов | | Тип указателя | | Описание |
|---|---|---|---|---|---|
| | кода | данных | для кода | для данных | |
| 16-разрядные приложения MS-DOS и Windows 3.x | | | | | |

| | | | | | |
|---|---|---|---|---|---|
| Tiny | один, <=64Kb | | near | near | Код, данные и стек находятся в одном сегменте. Эта модель памяти используется для написания программ типа .COM |
| Small | один,<=64Kb | один,<=64Kb | near | near | Код программы находится в одном сегменте. Данные и стек находятся в другом сегменте |
| Medium | несколько, <=64Kb | один,<=64Kb | far | near | Код находится в нескольких сегментах, по одному на каждый программный модуль. Данные объединены в один сегмент |
| Compact | один,<=64Kb | несколько, <=64Kb | near | far | Код находится в одном сегменте. Данные могут находится в нескольких сегментах. Для ссылки на данные из кода применяются указатели дальнего типа |
| Large | несколько, <=64Kb | несколько, <=64Kb | far | far | Код может размещаться в нескольких сегментах, на каждый модуль новый сегмент. Данные также размещаются в нескольких сегментах. Для ссылки на данные из кода применяются указатели дальнего типа. |
| Huge | несколько, >64Kb | несколько, >64Kb | huge | huge | Код может размещаться в нескольких сегментах, на каждый модуль новый сегмент. Данные и код имеют тип huge |
| Tchuge | несколько, =<64Kb | несколько, =<64Kb | far | far | Также как для модели large, но с иным использованием сегментных регистров |
| 32-разрядная Windows | | | | | |
| Flat | не ограничено | не ограничено | flat | flat | Соответствует варианту модели small, но с использованием 32-разрядной адресации |

### 3.3.3.Регистры центрального процессора

Рассмотрим основные регистры процессора. Регистры общего назначения (**РОН**) использоваться без ограничений: для любых целей.

Названия этих регистров происходят из того, что некоторые команды применяют их специальным образом:

***Аккумулятор*** часто используется для хранения результата действий, выполняемых над двумя операндами (результата бинарных операций).
Если результат такой операции не помещается в аккумуляторе, то старшая часть результата помещается в ***регистр данных***.
***Регистр-база*** используется при адресации по базе.
***Регистр-счетчик*** используется как счетчик в циклах и строковых операциях.
Младшие 16 бит этих регистров могут использоваться как самостоятельные регистры с именами **AX, BX, CX, DX**.
Байты с именами **AH, AL, …, DH, DL** могут использоваться как самостоятельные регистры. Буква **H** в названиях байт обозначает **Hi**, т.е. старший (байт слова).
Буква **L** - обозначает **Low** – младший байт слова.
Последние 4 РОН могут применяться для хранения временных переменных только, когда они не используются по назначению.
**ESI**– индекс источника (source).
**EDI** – индекс приемника (Destination).
**EBP** – ***указатель базы.***
**ESP** – ***указатель стека.***
**ESI** и **EDI** используются в строковых операциях. **EBP** и **ESP** используются при работе со стеком.
У процессора Intel 6 сегментных регистров длиной по 2 байта.

В зависимости от режима работы процессора по содержимому сегментных регистров определяются адреса памяти, с которых начинаются соответствующие сегменты.
**CS** (**Code Segment**) – ***сегмент кода*** – содержит команды программы. Для доступа к этому сегменту используется регистр **CS**.
**DS** (**Data Segment**) – ***сегмент данных*** – содержит обрабатываемые программой данные. Для доступа к этому сегменту используется регистр **DS**.
**SS** (**Stack Segment**) – ***сегмент стека*** – область памяти с ограниченным доступом к ее содержимому. Доступ к данным в стеке организуется по правилу FILO (LIFO).

Сегмент кода содержит программу, использующуюся в данный момент.

Запись нового содержимого в регистр **CS** приводит к тому, что далее будет исполнена не следующая по тексту программы команда, а команда из кода, находящегося в другом сегменте, но с тем же смещением.
Если программе недостаточно одного сегмента данных, то она может задействовать еще 3 дополнительных сегмента данных. Адреса дополнительных сегментов данных должны содержаться в регистрах дополнительного сегмента данных (**Extantion Data Segment Registers): ES, GS, FS**.
Смещение следующей выполняемой команды всегда хранится в специальном регистре – **EIP** – ***регистр указатель команд*** (**Instruction Pointer Register**).
.Системные регистры
Для работы в защищенном режиме введены дополнительные регистры:

1. CR0-CR4 – 32-разрядные управляющие регистры;
2. LDTR – регистр локальной таблицы дескрипторов;
3. GDTR – регистр глобальной таблицы дескрипторов;
4. IDTR – регистр дескрипторной таблицы прерываний;
5. TR – регистр задачи;
6. DR0-DR7 – отладочные регистры;
7. TR3-TR7 – тестовые регистры.

Регистры CR0-CR3 предназначены для общего управления системой. Регистры управления доступны только программам с уровнем привилегий 0. Регистр CR0 содержит набор бит, которые управляют режимами работы микропроцессора и отражают его состояние глобально, независимо от конкретных выполняющихся задач (таблица3).

Таблица3 -Назначение битов регистра CR0

| Номер бита в CR0 | Название и назначение | Мнемоника |
|---|---|---|
| 0 | Protection Enable – разрешение защищенного режима. Если PE=PG=0, то процессор работает в реальном режиме. | PE |
| 1 | Monitor Soprocessor – слежение за сопроцессором. Этот бит используется совместно с битом TS. Если MP=TS=1, то при выполнении команды WAIT генерируется особая ситуация 7 – «сопроцессор отсутствует». | MP |
| 2 | Emulate Soprocessor – эмуляция сопроцессора. Если EM=1, то генерируется особая ситуация 7 – «сопроцессор отсутствует» и все команды сопроцессора эмулируются. При выполнении команд MMX этот флаг должен быть сброшен (EM=0). | EM |
| 3 | Task Switcher – переключатель задач. Этот бит устанавливается при каждом переключении задач и вызывает особую ситуацию 7 при поступлении команд сопроцессора, MMX/3DNow и SIMD. Это нужно, чтобы предотвратить выполнение этих команд над данными, которые остались в оперативной памяти от другой задачи. | TS |
| 4 | Processor Extension Type – тип сопроцессора. Если ET=1, значит, используется 32-битный протокол FPU, если ET=0, используется 16-битный протокол FPU. | ET |
| 5 | Numeric Error – ошибка сопроцессора. Если NE=0, то обработка исключений сопроцессора происходит в стиле MS DOS (через вызов внешнего прерывания). Если NE=1, то действует внутренний механизм | NE |

| | | |
|---|---|---|
| | обработки исключений при поступлении команд сопроцессора, MMX/3DNow и SIMD. | |
| 16 | Write Protection – защита записи. Если WP=1, то происходит защита записи от записи супервизором операционной системы страниц пользовательского уровня. В противном случае, такая защита снимается. WP | WP |
| 18 | Alignment Mask – маска выравнивания. Контроль выравнивания операндов в оперативной памяти выполняется, если AM=AC=1 IOPL=3 i386 для i386 AM=0. | AM |
| 20 | No Write – запрет сквозной записи. Работает вместе с битом CD и предназначен для управления внутренней кэш-памятью. | NW |
| 30 | Cash Data – разрешение работы внутренней кэш-памяти (L1, L2) CD=0 | CD |
| 31 | Paging Enable – включение режима страничной адресации (PG=1) | PG |

Функции регистра CR1 не определены. Он зарезервирован для дальнейшего использования.

20 старших бит (биты 31-12) регистра CR3 содержат смещение каталога страниц в памяти (используется для преобразования логического адреса в физический адрес), биты 11-5 и 2-0 равны нулю, назначение остальных битов в таблице 4.

Таблица 4 Назначение битов регистра CR3

| Номер бита в CR3 | Название и назначение | Мнемоника |
|---|---|---|
| 3 | Page level Write Transparent – бит, управляющий сквозной записью страниц. Используется для управления кэшированием текущего каталога страниц, если установлен режим страничной адресации (бит PG=1 в регистре CR0). Если PWT=1, тогда обновление текущего каталога страниц происходит методом сквозной записи (writethrough), если PWT=0 – методом обратной записи (write-back). | PWT |
| 4 | Page level Cash Disable – бит, контролирующий кэширование | PCD |

| | | |
|---|---|---|
| | каталога страниц. Если PCD=1, то кэширование не происходит.<br>Если PCD=0 – каталог страниц может кэшироваться. Этот флаг влияет только на L1 и L2 (внутренние кэши процессора).<br>Процессор игнорирует этот флаг, если страничная адресация<br>отключена (флаг PG в CR0 сброшен) или если вообще отключен | |

Регистр глобальной таблицы дескрипторов содержит 32-битовый базовый адрес глобальной дескрипторной таблицы (начальный адрес системной области памяти, в которой хранятся общедоступные селекторы, описывающие глобальные сегменты памяти, обычно такие сегменты относятся к операционной системе) и 16-битовое значение предела, представляющее собой размер в байтах таблицы GDT.

Регистр дескрипторной таблицы прерываний хранит 32-битовый базовый адрес дескрипторной таблицы прерываний микропроцессора IDT (системной области памяти, в которой размещается таблица прерываний) и 16-битовое значение предела, представляющее собой размер в байтах таблицы IDT.

16-разрядный регистр локальной таблицы дескрипторов содержит селектор, описывающий для данной задачи таблицы дескрипторов. Этот селектор является указателем в таблице GDT, который и описывает сегмент, содержащий локальную дескрипторную таблицу LDT.

16-разрядный регистр задачи хранит селектор сегмента, то есть указатель на дескриптор в таблице GDT, в котором записывается информация о текущей задаче, выполняемой микропроцессором. Этот дескриптор описывает текущий сегмент состояния задачи (TSS – Task Segment Status). Этот сегмент создается для каждой задачи в системе, имеет жестко регламентированную структуру и содержит контекст (текущее состояние) задачи. Основное назначение сегментов TSS – сохранять текущее состояние в момент переключения на другую задачу.

Регистры GDTR, IDTR 48-разрядные и хранят физические адреса этих таблиц, которые непосредственно выдаются на шину адреса микропроцессора.

32-разрядные отладочные регистры (Debug Registers) DR0-DR7: DR0-DR3 (Linear Breakpoint Address 0..3) содержат 32-битные линейные адреса точек прерывания, а DR4-DR7 устанавливают, что должно произойти при достижении точки прерывания. Регистр DR6 (Breakpoint Status) – регистр состояния отладки. Биты этого регистра устанавливаются в соответствии с причинами, которые вызывали возникновение последнего исключения с номером 1. Регистр DR7 (Breakpoint Control) – регистр управления отладкой. В этом регистре для каждого из четырех регистров контрольных точек отладки

DR0-DR3 имеются поля, с помощью которых можно уточнить условия, при которых следует сгенерировать прерывание.
Тестовые регистры TR3-TR7 (Test Registers) используются для контроля постраничной системы распределения памяти операционной системы:
TR3 – регистр данных внутреннего кэша,
TR4 – тестовый регистр состояния кэша,
TR5 – управляющий регистр тестирования кэша,
TR6 (Test Control) – управляющий регистр для теста кэширования страниц,
TR7 (Test Status) – регистр данных для теста кэширования страниц.

. Регистр флагов (RFLAGS/EFLAGS)

Регистр RFLAGS/EFLAGS используется для получения информации о результатах выполнения команд и влияет на состояние самого микропроцессора. 64/32-битовый регистр RFLAGS/ FLAGS содержит информацию, которая используется, скорее побитно, нежели как 64/32-битное число. Старшие 32 бита RFLAGS равны нулю. Младшие 16 разрядов регистра RFLAGS/EFLAGS, для совместимости со старыми программами, полностьюаналогичны регистру FLAGS i8086, 8 младших разрядов которого, в свою очередь, аналогичны регистру FLAGS i8080.
Регистр RFLAGS/EFLAGS представляет собой набор флагов, устанавливаемых или сбрасываемых по результатам выполняемых команд. Флаг – это переменная длиной 1 бит, используемая в командах условного перехода. Если значение этой переменной равно 1, то считается, что флаг установлен, если 0 – сброшен.

Флаги делятся на:
8 флагов состояния (NT, IOPL, SF, ZF, AF, PF, CF). Эти флаги могут изменяться после выполнения машинных команд. Флаги состояния регистра RFLAGS/EFLAGS отражают особенности результата исполнения арифметических или логических операций. Это дает возможность анализировать состояние вычислительного процесса и реагировать на него с помощью команд условных переходов и вызовов подпрограмм.

Для работы с флагом CF существуют специальные команды CLC (очистить флаг CF), STC (установить флаг CF), CMC (инвертировать флаг CF);
флаг направления (DF) используется цепочечными командами. Значение этого флага определяет направление поэлементной обработки в этих операциях: от начала строки к концу (DF=0), либо наоборот, от конца строки к ее началу (DF=1).

Для работы с флагом DF существуют специальные команды CLD (очистить флаг DF), STD (установить флаг DF). Применение этих команд позволяет привести флаг DF в соответствие с алгоритмом и обеспечить автоматическое увеличение или уменьшение счетчиков при выполнении операций со строками;

5 системных флагов (FT, IF, RF, VM, AC, VIP, VIF, ID), управляющих вводом/выводом, маскируемыми прерываниями, отладкой, переключение между задачами и виртуальным режимом 8086. Прикладным программам не рекомендуется модифицировать без необходимости эти флаги, так как в большинстве случаев это приведет к прерыванию работы программы. Для работы с флагом IF существуют специальные команды CLI (очистить флаг IF), STI (установить флаг IF).

Из флагов состояния программиста, создающего программы на языке ассемблера, в первую очередь интересуют: флаг нуля, флаг переноса и флаг знака.

CF флаг переноса (Carry Flag) устанавливается при переносе или займе старшего бита в арифметических операциях, в остальных случаях сбрасывается.

PF флаг паритета (Parity Flag) устанавливается, если 8 младших разрядов результата содержат четное число единиц; в противном случае сбрасывается.

AF вспомогательный флаг переноса (Auxiliary carry Flag). Используется командами работающими с BCD числами (AAA, AAS, AAD, AAM, DAA, DAS). Устанавливается при переносе из 3-го бита в 4-й в результате сложения, или при займе из 4-го бита в 3-й в результате вычитания, в остальных случаях сбрасывается.

ZF флаг нуля (Zero Flag) устанавливается в случае получения нулевого результата при выполнении очередной команды и сбрасывается при остальных ненулевых значениях.

SF флаг знака (Sign Flag) устанавливается при единичном значении старшего бита результата – признак отрицательного числа.

TF флаг трассировки (Trace Flag) помогает отлаживать программы. Этот флаг не устанавливается в результате работы микропроцессора, а устанавливается программой с помощью специальной команды. Его также называют флагом трассировки, флагом пошаговой работы или флагом ловушки. Используется преимущественно для осуществления пошагового режима работы. Когда флаг TF установлен, микропроцессор x86 автоматически вырабатывает сигнал внутреннего прерывания INT 1h после выполнения каждой команды. Это обеспечивает удобство проверки программ,написанных на машинном коде, поскольку они при этом выполняются команда за командой. Адрес сервисной процедуры организации пошагового режима должен быть определен в абсолютных адресах от 00004h до 00007h. При генерации прерывания INT 1h микропроцессор x86 автоматически загружает в стек содержимое регистра флагов (с единичным флагом ловушки), после чего обнуляет флаги ловушки и прерываний. Таким образом, микропроцессор x86 не переходит в пошаговый режим пока выполняется сама процедура обслуживания прерывания, которая реализует различные диагностические операции. Процедура обслуживания прерывания INT 1h может включать в себя отображение на дисплее такой информации, как содержимое регистров и определенной области памяти сразу после

выполнения команды. Последняя машинная команда в процедуре обслуживания – команда IRET (возврат из прерывания). Этим восстанавливается прежнее единичное состояние флага TF, и по завершении следующей команды вновь генерируется прерывание INT 1h.

Флаг ловушки TF занимает восьмой разряд в регистре флагов. Для установки флага TF в ноль или единицу можно использовать команду POPF. Для противодействия просмотру фрагментов программ под отладчиком подменялась процедура организации пошагового режима, которая выводила на экран не тот фрагмент программы, которую исследовал хакер, а какой-либо другой.

IF флаг прерываний (Interrupt Flag) управляет внешними прерывания. Пока флаг прерываний сброшен в 0, никакие внешние прерывания не будут обрабатываться микропроцессором (за исключением немаскируемых). Когда он установлен в 1, будет производиться обработка любых возникающих прерываний.

DF флаг направления (Direct Flag) устанавливает направление обработки строк.

OF флаг переполнения (Overflow Flag) используется для фиксирования факта потери значащего бита при арифметических операциях, устанавливается при переносе или займе старшего знакового бита в арифметических операциях, в остальных случаях сбрасывается.

IOPL флаг уровня привилегий ввода /вывода (Input/Output Privilege Level). Используется в защищенном режиме работы микропроцессора для контроля доступа к командам ввода/вывода в зависимости от привилегированности задачи. Уровень привилегий ввода/вывода (IOPL) указывает максимальное значение текущего уровня привилегий (CPL – current privilege level). Для максимально допустимого значения CPL при выполнении команд ввода/вывода без генерирования прерывания по #13 исключению он также указывает максимальное значение CPL, позволяющее изменить бит IF, когда новые значения загружаются из стека в регистры FLAGS или EFLAGS. Команды POPF и IRET могут изменять поле IOPL, когда выполняются при CPL=0. Операции включения задач всегда изменяют поле IOPL, когда новый образ флага загружается из сегмента состояния задачи.

NT флаг вложенности задачи (Nested Task). Данный флаг используется в защищенном режиме. NT устанавливается, чтобы показать, что выполнение данной задачи вложено в пределах другой задачи. Если он установлен, то сегмент состояния текущей вложенной задачи имеет достоверную обратную связь с сегментом состояния предыдущей задачи. Данный бит устанавливается или сбрасывается командами, передающими управление другим задачам. Значение NT в EFLAGS проверяется командой IRET. Чтобы установить NT следует выполнять внутризадачное или внешнезадачное возвращение. Команды POPF или IRET будут оказывать воздействие на установку данного бита согласно образу EFLAGS на любом уровне привилегий.

RF флаг возобновления (Resumption Flag) используется при пошаговом

режиме или совместно с точками прерываний регистров отладки. Он проверяется на границе команды, перед обработкой точки останова. Если RF установлен, то любая ошибка отладки будет на следующей команде проигнорирована. RF автоматически сбрасывается при успешном окончании каждой команды (ошибки не сигнализируются), кроме команд IRET, POPF,JMP, CALL, ENTER, которые вызывают включение задачи. Эти команды устанавливают RF в соответствии с определенным образом памяти.

Таблица 5 Регистр флагов.

**Регистр флагов Intel x86**

| Бит, № | Обозначение | Название | Описание | Тип флага | Когда введён |
|---|---|---|---|---|---|
| | | | **FLAGS** | | |
| 0 | CF | Carry Flag | Флаг переноса | Состояние | |
| 1 | 1 | | Зарезервирован | | |
| 2 | PF | Parity Flag | Флаг чётности | Состояние | |
| 3 | 0 | | Зарезервирован | | |
| 4 | AF | Auxiliary Carry Flag | Вспомогательный флаг переноса | Состояние | |
| 5 | 0 | | Зарезервирован | | |
| 6 | ZF | Zero Flag | Флаг нуля | Состояние | |
| 7 | SF | Sign Flag | Флаг знака | Состояние | |
| 8 | TF | Trap Flag | Флаг трассировки (пошаговое выполнение) | Системный | |
| 9 | IF | Interrupt Enable Flag | Флаг разрешения прерываний | Системный | |
| 10 | DF | Direction Flag | Флаг направления | Управляющий | |
| 11 | OF | Overflow Flag | Флаг переполнения | Состояние | |
| 12<br>13 | IOPL | I/O Privilege Level | Уровень приоритета ввода-вывода | Системный | 80286 |
| 14 | NT | Nested Task | Флаг вложенности задач | Системный | 80286 |
| 15 | 0 | | Зарезервирован | | |
| | | | **EFLAGS** | | |
| 16 | RF | Resume Flag | Флаг возобновления | Системный | 80386 |
| 17 | VM | Virtual-8086 Mode | Режим виртуального процессора 8086 | Системный | 80386 |
| 18 | AC | Alignment Check | Проверка выравнивания | Системный | 80486SX |
| 19 | VIF | Virtual Interrupt Flag | Виртуальный флаг разрешения прерывания | Системный | Pentium |
| 20 | VIP | Virtual Interrupt Pending | Ожидающее виртуальное прерывание | Системный | Pentium |

| | | | | | |
|---|---|---|---|---|---|
| 21 | ID | ID Flag | Проверка на доступность инструкции CPUID | Системный | Поздние 80486[2] |
| 22 | | | | | |
| … 31 | 0 | | Зарезервированы | | |
| | | | **RFLAGS** | | |
| 32 | 0 | | Зарезервирован | | |

VM флаг виртуального режима (Virtual 8086 Mode) обеспечивает виртуальный 8086 режим в пределах защищенного. Если он установлен в то время, когда микропроцессор находится в защищенном режиме, 8086 включается в виртуальную 8086 операцию, манипулируя загрузкой сегментов, как это делает 8086, генерируя 13 прерывание привилегированных операционных кодов. Бит VM может быть установлен в защищенном режиме командой IRET, если текущий привилегированный уровень равен 0, и путемвключения задач на любом уровне привилегий. Бит VM не подчиняется действию команды POPF. PUSHF всегда посылает 0 в этот разряд, даже, если работает в виртуальном 8086 режиме. Образ EFLAGS, сохраненный в стеке во время обработки прерывания или во время включения задачи, будет содержать единицу в этом бите, если прерывание обрабатывалось как виртуальная 8086 задача.

AC контроль выравнивания (Alignment Control). Флаг контроля используется совместно с битом AM регистра CR0 для отслеживания особых ситуаций, связанных с выравниванием операндов при обращении к оперативной памяти. Особая ситуация контроля выравнивания генерируется только, если уровень привилегий IOPTL=3.

VIF виртуальное прерывание (Virtual Interruption Flag). Виртуальное подобие флага IF. Флаг VIF применяется совместно с флагом VIP для нормального функционирования устаревшего программного обеспечения, использующего команды управления маскируемыми прерываниями.

VIP флаг ожидания виртуального прерывания (Virtual INTERRUPTION).

ID флаг идентификации (IDENTIFICATION). Проверка способности программы поддерживать команду идентификации процессора CPUID.

Не все операции устанавливают флажки. За приблизительное правило можно принять, что:

а) арифметические операции действуют на все флажки,

б) логические операции (кроме операции NOT) сбрасывают флажки переноса и переполнения (OF и CF) и действуют на все другие флажки,

в) операции приращения и уменьшения на 1 (команды INC и DEC) не действуют на флажок переноса и действуют на все другие флажки (нельзя использовать флажок переноса для контроля переполнения при операции уменьшение на 1, но можно использовать флаг знака).

Не изменяют флажки:

а) все операции перемещения (MOV, MOVZX, MOVSX, LEA, LDS,

LES, LFS, LGS, LSS, XCHG, BSWAP),
б) все операции с портами (IN, OUT, INS, OUTS),
в) все команды перехода (JMP, Jcc, JCXZ, JECXZ),
г) все команды преобразования (CBW, CWD, CWDE, CDQ),
д) все строковые команды (кроме SCAS и CMPS),
е) все операции со стеком (кроме POPF).

### 3.4. Средства Ассемблера для разработки Windows-приложений

#### 3.4.1. Подготовка исходного текста программы

Обычно, разработка программы включает несколько этапов, что впрочем. относится ко многим языкам программирования:

1. подготовка (изменение) исходного текста программы,
2. ассемблирование программы (получение объектного кода),
3. компоновка программы (получение исполняемого файла программы),
4. запуск программы,
5. отладка программы.

В случае необходимости эти этапы циклически повторяются, потому что при нахождении ошибок при ассемблировании, компоновки или отладке приходится вновь возвращаться к первому этапу и изменять текст программы для устранения ошибок.

При использовании стандартных редакторов текста необходимо сохранять редактируемые файлы с текстами программ в виде обыкновенных файлов в формате ASCII, то есть без дополнительных символов форматирования, которые вставляются в текст специализированными редакторами, например. Word.

#### 3.4.2. Ассемблирование программы

Подготовленный текст является исходными данными для специальных программ, называемых ассемблерами. Задача ассемблеров – преобразовать текст программы в форму двоичных команд, которые могут быть выполнены микропроцессором. Если обнаружены синтаксические ошибки, то результирующий код создан не будет. Процесс создания исполняемого файла происходит в две стадии:
.asm -> .obj -> .exe/.dll/.com

На первой стадии (.asm -> .obj) из ассемблерного файла путем компиляции получаются файлы промежуточного объектного кода, имеющего расширение .obj (при этом могут использоваться дополнительные inc-файлы). Файл с расширением .obj содержит машинный код при условии, что не встретились синтаксические и семантические ошибки. Если в исходном файле с программой на языке ассемблера обнаруживаются ошибки, то программисту выдается список обнаруженных ошибок, в котором ошибки указываются с номером строки, в которой они обнаружены. Программист циклически выполняет действия по редактированию и компиляции до тех пор, пока не будут устранены все ошибки в исходном файле. На этом этапе уже возможно получение готовой программы, но чаще всего в ней не хватает некоторых компонентов. Если компилятор по какой-либо причине (неверно прописан

путь к такому файлу или файл отсутствует) не может найти inc-файл, то выдается предупреждение и obj-файл получен не будет.

Ассемблирование, как правило, проходит в два приема, если ассемблер (компилятор) двухпроходный. При первом проходе переводятся мнемонические команды, десятеричные числа и символы в соответствующие машинные коды, подсчитывается, сколько какая команда занимает места, обнаруженные имена, введенные пользователем (константы, метки, переменные) их тип и числовое значение записывается в таблицу. В эту же таблицу записывается, с каких адресов начинаются процедуры, адреса меток, адреса начала/конца сегментов и т. д., при втором проходе подставляются адреса начала процедур, заменяются названия меток на адреса.

В результате ассемблирования получается так называемый «объектный файл». В качестве дополнительной возможности ассемблер может создать файл листинга программы.

Обычно для получения файлов объектного кода необходимо выполнить соответствующую программу ассемблера (программы MASM.EXE и ML.EXE фирмы Microsoft и TASM.EXE или TASM32.EXE фирмы Borland), указав в командной строке имя файла с текстом программы.

Кроме имени текстового файла, необходимо указывать опции ассемблирования. Если синтаксических ошибок в файле prog.asm, нет, тогда будет создан файл prog.obj. Более подробную информацию об опциях программы ассемблирования следует искать в документации к этим программам.

### 3.4.3. Компоновка программы

Следующий этап (.obj -> .exe/.dll/.com) называется линковкой или компоновкой и служит для замещения символьных имен, используемых программистом, на реальные адреса.

При сравнении шестнадцатеричного содержимого OBJ и EXE файла, который получился, можно отметить, что в EXE-файле присутствует та же последовательность байтов, что и в OBJ-файле. Но помимо этого еще присутствует: имя ассемблированного файла, версия ассемблера, «имя собственное» сегмента и так далее. Это «служебная» информация, предназначенная для тех случаев, когда исполняемый файл собирается из нескольких. При разработке больших приложений исходный текст состоит, как правило, из нескольких модулей (файлов с исходными текстами), потому что хранить все тексты в одном файле неудобно – в них сложно ориентироваться. Каждый модуль по раздельности компилируется в отдельный файл с объектным кодом. В каждом из этих файлов прописаны свои сегменты кода/данных/стека, которые затем надо объединить в одно целое. А исполнимый файл нам нужно получить только один – с единым сегментом кода/данных/стека. Именно это LINK и делает: завершает определение адресных ссылок и объединяет, если это требуется, несколько программных модулей в один. И этот один и является исполняемым.

Кроме того, отметим, что к модулям надо добавить машинный код подпрограмм, реализующих различные стандартные функции (например, вычисляющих математические функции SIN или LN). Такие функции содержатся в библиотеках (файлах со стандартным расширением .LIB), которые либо поставляются вместе с компилятором, либо создаются самостоятельно. Поэтому процесс подготовки обязательно включает в себя этап компоновки, когда определяются все неизвестные при раздельном ассемблировании адреса совместно используемых переменных или функций.

Процесс объединения объектных модулей в один файл осуществляется специальной программой-компоновщиком или *сборщиком* (программа LINK.EXE фирмы Microsoft и TLINK.EXE фирмы Borland), которая выполняет связывание объектных модулей и машинного кода стандартных функций, находя их в библиотеках, и формирует на выходе работоспособное приложение – *исполняемый код* для конкретной платформы.

Исполняемый код – это законченная программа с расширением COM, DLL или EXE, которую можно запустить на компьютере с установленной операционной системой, для которой эта программа создавалась. Имя исполняемого файла задастся именем первого .OBJ файла:

link prog1.obj prog2.obj

Содержимое объектного файла анализируется компоновщиком. Он определяет, есть ли в программе внешние ссылки, то есть содержит ли программа команды вызовов процедур, находящихся в одной из библиотек объектных модулей (link library). Компоновщик находит эти ссылки в объектном файле, копирует необходимые процедуры из библиотек, объединяет их вместе с объектным файлом и создает исполняемый файл (executable file). В качестве дополнительных возможностей компоновщик может создать файл перекрестных ссылок, содержащих план полученного исполняемого файла.

Возможно потребуется LST-файл, особенно, если необходимо проверить сгенерированный машинный код. CRF-файл (файл перекрестных ссылок) полезен для очень больших программ, где необходимо видеть, какие команды ссылаются на какие поля данных. Кроме того, ассемблер генерирует в LST-файле номера строк, которые используются в CRF-файле

Листинг содержит не только исходный текст, но также слева транслированный машинный код в шестнадцатиричном формате. В самой левой колонке находятся шестнадцатеричные адреса команд и данных.

За листингом ассемблирования программы следует таблица идентификаторов. Первая часть таблицы содержит определенные в программе сегменты и группы вместе с их размером в байтах, выравниванием и классом.

Вторая часть содержит идентификаторы - имена полей данных в сегменте данных и метки, назначенные командам в сегменте кодов.

В MAP-файле перечисляются имена. адреса загрузки и размеры всех сегментов, входящих в программу. В нем также приводятся имена и адреса загрузки каждой группы в программе, начальный адрес программы и

сообщения о тех или иных ошибках, которые могли иметь место. Если в командной строке LINK задана опция /MAP, то в map-файле перечисляются имена и адреса загрузки всех обобщенных символов.

### 3.5.Средства TASM и MASM32 для разработки Windows-приложений

При разработке Windows-приложений на языке ассемблера с помощью Win32 API нужен один из пакетов ассемблера, так можно использовать пакет TASM версии 5.0. Современные 32-разрядные операционные системы Windows используют формат PE исполняемого файла. В состав пакета TASM 5.0 входят два компилятора ассемблера - 16- и 32-разрядный. Они имеют имена исполняемых файлов, соответственно, tasm.exe и tasm32.exe. То же касается и редакторов связей - tlink.exe и tlink32.exe. Получить файл формата EXE можно только при совместном использовании файлов tasm32.exe и tlink32.exe.

Для создания программы нужны еще два файла: файл определений компоновщика и файл описания ресурсов.

Назначение файла определений компоновщика состоит в том, чтобы предоставить редактору связей информацию о способе загрузки программы. Несмотря на то что в архитектуре Win32 нет особого смысла использовать данный файл, редактор tlink32.exe требует указания этого файла среди файлов, подаваемых ему в качестве входных.

Перечислим необходимые для разработки Windows-приложения файлы.
- Файл с исходным текстом программы (.asm). Формируется программистом.
- Включаемый файл с описаниями структур данных и констант
Win32 (.inc или.ash). Файл формируется программистом по мере расширения используемых им средств Win32. Источником информации для этого файла служат включаемые файлы (.h) из пакета компилятора C/C++, например VC++ версии 4.0 и выше.
- Файл с библиотекой импорта import32.lib. Этот файл требуется компоновщику для разрешения внешних ссылок на функции Win32 API. Этот файл можно создать самим. Такая необходимость может возникнуть, если понадобятся функции из библиотек DLL, информация о которых отсутствует в существующем варианте файла import32.lib. Для этого существует специальная утилита implib.exe, поставляемая в пакете TASM 5.O. Командная строка для ее запуска имеет вид:
implib имя_файла_li b список_dll_библиотек
- Файл с описанием ресурсов, используемых в приложении (.res)
- Другие файлы. Например, звуковые файлы (.wav).
- Файлы tasm32, tlink32.exe и, возможно, некоторые другие вспомогательные файлы из пакета TASM 5.O.

Подготовленный текст – файл name.asm - обрабатывается макроассемблером (ml) командой
ml /c /coff /Cp name.asm
результат – объектный файл name.obj, который должен быть обработан

линковщиком link командой link /SUBSYSTEM:WINDOWS /LIBPATH:c:\masm32\lib name.obj
При безошибочном выполнении последней операции будет получен выполняемый файл name.exe.

Создадим bat-файл, который позволит автоматизировать описанные выше процедуры:

```
cls
if exist %~n1.exe del %~n1.exe
if not exist %~n1.rc goto over1
\masm32\bin\rc /v %~n1.rc
\masm32\bin\cvtres /machine:ix86 %~n1.res
\masm32\bin\ml /c /Cp /Gz /I\masm32\include \
/coff /nologo %~n1.asm
if errorlevel 1 goto TheEnd
\masm32\bin\Link /SUBSYSTEM:WINDOWS /ALIGN:16 \
/MERGE:.data=.text /LIBPATH:\masm32\lib /NOLOGO \
%~n1.obj %~n1.res
if errorlevel 1 goto TheEnd
del %~n1.res
goto TheEnd
:over1
\masm32\bin\ml /c /Cp /Gz /I\masm32\include \
/coff /nologo %~n1.asm
if errorlevel 1 goto TheEnd
\masm32\bin\Link /SUBSYSTEM:WINDOWS /ALIGN:16 \
/MERGE:.data=.text /LIBPATH:\masm32\lib /NOLOGO \
%~n1.obj
:TheEnd
if exist %~n1.obj del %~n1.obj
pause
```

### 3.6.**Примеры программ**

Листинг 1

```
.686P
.model flat
include windows.inc
includelib user32.lib
extern
_imp__MessageBoxA@16:dword .code
start: push 0
push offset sztext
push offset szText
push 0
call _imp__MessageBoxA@16
```

```
ret
sztext db "Моё первое приложение",0
szText db "Привет!",0
end start
```

Листинг 2

Программа, выводящая на экран в текстовом режиме строку символов

```
.386
.model flat, stdcall option casemap: none  include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
 includelib kernel32.lib
   .data
  cFNPrompt      db "Hello ",13,10   cbFNPrompt      dd  sizeof cFNPrompt
  hStdOut         dd 0    dwBytes          dd 0    szAppName    db "input    ",0    hStdIn
dd 0
 .code
main PROC
        invoke FreeConsole ; освободить консоль          invoke    AllocConsole    ;
добавить консоль
        Invoke GetStdHandle, STD_OUTPUT_HANDLE
        Mov hStdOut,EAX
        Invoke GetStdHandle, STD_INPUT_HANDLE
        Mov hStdIn,EAX
        Invoke  WriteConsoleA,hStdOut,  Addr  cFNPrompt,  cbFNPrompt,  Addr
dwBytes, 0
;       ожидание ввода символов
 Invoke ReadConsole, hStdIn, Addr szAppName, 10, Addr dwBytes, 0     invoke
ExitProcess,0 main ENDP

end main
```

**Дополнение**

**Пример1**-. Вывод строки, содержащей символы кириллицы

Для преобразования текстовой строки в кодировку Win1251 используется функция CharToOem

```
invoke CharToOem, Addr cTransl, Addr cResult ; перевод в кодировку 1251
```

Первый аргумент функции – указатель на транслируемую строку (с 0 в конце), второй – указатель на строку результата.

**Пример 2**. Изменение цвета символов и цвета фона.

Цвет текста и цвет фона задает байт атрибутов с помощью вызова функции

```
invoke SetConsoleTextAttribute, hStdOut, atr
```

Первый аргумент типа dword определяет дескриптор терминала - окно вывода, второй – также типа dword - цвет фона и цвет символов. Младший байт должен иметь следующий формат ( R – красный; G – зеленый; B – синий)

**4.Порядок выполнения**

1. Выполнить программу, приведенную влистинге1 .
2. Вывести на экран предложение «Ассемблер ОС Windows » красным цветом на черном фоне, затем белым цветом на голубом фоне.
3. Вывести на экран предложение «Ассемблер ОС Windows » в собственном окне приложения.

**Литература**

1.В.И.Юров .- Ассемблер, 2 издание- СПб: Питер, 2003.
2.Пирогов В. Ю.- Ассемблер для Windows. Изд. 4-е перераб. и доп. — СПб.: БХВ- Петербург, 2015.
3. И.А. Калашников. Ассемблер – это просто. Программирование на Ассемблере. «Бином», 2008 г.

Приложение 1

```
.586P
.MODEL FLAT, stdcall
;сообщение приходит при закрытии окна
WM_DESTROY equ 2
;сообщение приходит при создании окна
WM_CREATE equ 1
;сообщение при щелчке левой кнопкой мыши в области окна
WM_LBUTTONDOWN equ 201h
;свойства окна
CS_VREDRAW equ 1h
CS_HREDRAW equ 2h
CS_GLOBALCLASS equ 4000h
WS_OVERLAPPEDWINDOW equ 000CF0000H
WS_POPUP equ 80000000h
WS_CHILD equ 40000000h
STYLE equ CS_HREDRAW+CS_VREDRAW+CS_GLOBALCLASS
BS_DEFPUSHBUTTON equ 1h
WS_VISIBLE equ 10000000h
WS_CHILD equ 40000000h
STYLBTN equ WS_CHILD+BS_DEFPUSHBUTTON+WS_VISIBLE
;идентификатор стандартной пиктограммы
IDI_APPLICATION equ 32512
;идентификатор курсора
IDC_ARROW equ 32512
;режим показа окна - нормальный
SW_SHOWNORMAL equ 1
;прототипы внешних процедур
EXTERN MessageBoxA@16:NEAR
EXTERN CreateWindowExA@48:NEAR
```

```
EXTERN DefWindowProcA@16:NEAR
EXTERN DispatchMessageA@4:NEAR
EXTERN ExitProcess@4:NEAR
EXTERN GetMessageA@16:NEAR
EXTERN GetModuleHandleA@4:NEAR
EXTERN LoadCursorA@8:NEAR
EXTERN LoadIconA@8:NEAR
EXTERN PostQuitMessage@4:NEAR
EXTERN RegisterClassA@4:NEAR
EXTERN ShowWindow@8:NEAR
EXTERN TranslateMessage@4:NEAR
EXTERN UpdateWindow@4:NEAR
;структуры
;структура сообщения
MSGSTRUCT STRUC
MSHWND DD ?
MSMESSAGE DD ?
MSWPARAM DD ?
MSLPARAM DD ?
MSTIME DD ?
MSPT DD ?
MSGSTRUCT ENDS
;----структура класса окон
WNDCLASS STRUC
CLSSTYLE DD ?
CLWNDPROC DD ?
CLSCBCLSEX DD ?
CLSCBWNDEX DD ?
CLSHINST DD ?
CLSHICON DD ?
CLSHCURSOR DD ?
CLBKGROUND DD ?
CLMENNAME DD ?
CLNAME DD ?
WNDCLASS ENDS
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
;сегмент данных
_DATA SEGMENT
NEWHWND DD 0
MSG MSGSTRUCT <?>
WC WNDCLASS <?>
HINST DD 0 ;дескриптор приложения
```

```
TITLENAME DB 'Дочерние и собственные окна',0
TITLENAMED DB 'Дочернее окно',0
TITLENAMEO DB 'Собственное окно',0
CLASSNAME DB 'CLASS32',0
CLASSNAMED DB 'CLASS321',0
CLASSNAMEO DB 'CLASS322',0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить дескриптор приложения
PUSH 0
CALL GetModuleHandleA@4
MOV HINST, EAX
REG_CLASS:
;фрагмент, где осуществляется регистрация главного окна
;стиль
MOV WC.CLSSTYLE,STYLE
;процедура обработки сообщений
MOV WC.CLWNDPROC, OFFSET WNDPROC
MOV WC.CLSCBCLSEX, 0
MOV WC.CLSCBWNDEX, 0
MOV EAX, HINST
MOV WC.CLSHINST, EAX
;----------пиктограмма окна
PUSH IDI_APPLICATION
PUSH 0
CALL LoadIconA@8
MOV WC.CLSHICON, EAX
;----------курсор окна
PUSH IDC_ARROW
PUSH 0
CALL LoadCursorA@8
MOV WC.CLSHCURSOR, EAX
;----------регистрация основного окна
MOV WC.CLBKGROUND, 17 ;цвет окна
MOV DWORD PTR WC.CLMENNAME,0
MOV DWORD PTR WC.CLNAME,OFFSET CLASSNAME
PUSH OFFSET WC
CALL RegisterClassA@4
;фрагмент, где осуществляется регистрация дочернего окна
;стиль
MOV WC.CLSSTYLE,STYLE
;процедура обработки сообщений
```

```
MOV WC.CLWNDPROC, OFFSET WNDPROCD
MOV WC.CLSCBCLSEX, 0
MOV WC.CLSCBWNDEX, 0
MOV EAX, HINST
MOV WC.CLSHINST, EAX
MOV WC.CLBKGROUND, 2 ;цвет окна
MOV DWORD PTR WC.CLMENNAME,0
MOV DWORD PTR WC.CLNAME,OFFSET CLASSNAMED
PUSH OFFSET WC
CALL RegisterClassA@4
;фрагмент, где осуществляется регистрация собственного окна
;стиль
MOV WC.CLSSTYLE,STYLE
;процедура обработки сообщений
MOV WC.CLWNDPROC, OFFSET WNDPROCO
MOV WC.CLSCBCLSEX, 0
MOV WC.CLSCBWNDEX, 0
MOV EAX, HINST
MOV WC.CLSHINST, EAX
MOV WC.CLBKGROUND, 1 ;цвет окна
MOV DWORD PTR WC.CLMENNAME,0
MOV DWORD PTR WC.CLNAME,OFFSET CLASSNAMEO
PUSH OFFSET WC
CALL RegisterClassA@4
;создать окно зарегистрированного класса
PUSH 0
PUSH HINST
PUSH 0
PUSH 0
PUSH 400 ; DY - высота окна
PUSH 600 ; DX - ширина окна
PUSH 100 ; Y-координата левого верхнего угла
PUSH 100 ; X-координата левого верхнего угла
PUSH WS_OVERLAPPEDWINDOW
PUSH OFFSET TITLENAME ;имя окна
PUSH OFFSET CLASSNAME ;имя класса
PUSH 0
CALL CreateWindowExA@48
;проверка на ошибку
CMP EAX,0
JZ _ERR
MOV NEWHWND, EAX ;дескриптор окна
PUSH SW_SHOWNORMAL
PUSH NEWHWND
```

```
CALL ShowWindow@8 ; показать созданное окно
PUSH NEWHWND
CALL UpdateWindow@4 ; команда перерисовать видимую
;часть окна, сообщение WM_PAINT
;цикл обработки сообщений
MSG_LOOP:
PUSH 0
PUSH 0
PUSH 0
PUSH OFFSET MSG
CALL GetMessageA@16
CMP EAX, 0
JE END_LOOP
PUSH OFFSET MSG
CALL TranslateMessage@4
PUSH OFFSET MSG
CALL DispatchMessageA@4
JMP MSG_LOOP
END_LOOP:
;выход из программы (закрыть процесс)
PUSH MSG.MSWPARAM
CALL ExitProcess@4
_ERR:
JMP END_LOOP
;процедура главного окна
;расположение параметров в стеке
; [EBP+014H] ;LPARAM
; [EBP+10H] ;WAPARAM
; [EBP+0CH] ;MES
; [EBP+8] ;HWND
WNDPROC PROC
PUSH EBP
MOV EBP,ESP
PUSH EBX
PUSH ESI
PUSH EDI
CMP DWORD PTR [EBP+0CH], WM_DESTROY
JE WMDESTROY
CMP DWORD PTR [EBP+0CH], WM_CREATE
JE WMCREATE
CMP DWORD PTR [EBP+0CH], WM_LBUTTONDOWN
JE LB
JMP DEFWNDPROC
WMCREATE:
```

```
MOV EAX,0
JMP FINISH
LB:
;создать дочернее окно
PUSH 0
PUSH HINST
PUSH 0
PUSH DWORD PTR [EBP+08H]
PUSH 200 ; DY - высота окна
PUSH 200 ; DX - ширина окна
PUSH 50 ; Y-координата левого верхнего угла
PUSH 50 ; X-координата левого верхнего угла
PUSH WS_CHILD OR WS_VISIBLE OR WS_OVERLAPPEDWINDOW
PUSH OFFSET TITLENAMED ;имя окна
PUSH OFFSET CLASSNAMED ;имя класса
PUSH 0
CALL CreateWindowExA@48
;создать собственное окно
PUSH 0
PUSH HINST
PUSH 0
PUSH DWORD PTR [EBP+08H]
PUSH 200 ; DY - высота окна
PUSH 200 ; DX - ширина окна
PUSH 150 ; Y-координата левого верхнего угла
PUSH 250 ; X-координата левого верхнего угла
PUSH WS_POPUP OR WS_VISIBLE OR WS_OVERLAPPEDWINDOW
PUSH OFFSET TITLENAMEO ;имя окна
PUSH OFFSET CLASSNAMEO ;имя класса
PUSH 0
CALL CreateWindowExA@48 92
DEFWNDPROC:
PUSH DWORD PTR [EBP+14H]
PUSH DWORD PTR [EBP+10H]
PUSH DWORD PTR [EBP+0CH]
PUSH DWORD PTR [EBP+08H]
CALL DefWindowProcA@16
JMP FINISH
WMDESTROY:
PUSH 0
CALL PostQuitMessage@4 ;сообщение WM_QUIT
MOV EAX, 0
FINISH:
POP EDI
```

```
POP ESI
POP EBX
POP EBP
RET 16
WNDPROC ENDP
;процедура дочернего окна
;расположение параметров в стеке
; [EBP+014H] ;LPARAM
; [EBP+10H] ;WAPARAM
; [EBP+0CH] ;MES
; [EBP+8] ;HWND
WNDPROCD PROC
PUSH EBP
MOV EBP,ESP
PUSH EBX
PUSH ESI
PUSH EDI
CMP DWORD PTR [EBP+0CH], WM_DESTROY
JE WMDESTROY
CMP DWORD PTR [EBP+0CH], WM_CREATE
JE WMCREATE
JMP DEFWNDPROC
WMCREATE:
JMP FINISH
DEFWNDPROC:
PUSH DWORD PTR [EBP+14H]
PUSH DWORD PTR [EBP+10H]
PUSH DWORD PTR [EBP+0CH]
PUSH DWORD PTR [EBP+08H]
CALL DefWindowProcA@16
JMP FINISH
WMDESTROY:
MOV EAX, 0 Глава 1.3. Примеры простых программ на ассемблере 93
FINISH:
POP EDI
POP ESI
POP EBX
POP EBP
RET 16
WNDPROCD ENDP
;процедура собственного окна
;расположение параметров в стеке
; [EBP+014H] ;LPARAM
; [EBP+10H] ;WAPARAM
```

```
; [EBP+0CH] ;MES
; [EBP+8] ;HWND
WNDPROCO PROC
PUSH EBP
MOV EBP,ESP
PUSH EBX
PUSH ESI
PUSH EDI
CMP DWORD PTR [EBP+0CH], WM_DESTROY
JE WMDESTROY
CMP DWORD PTR [EBP+0CH], WM_CREATE
JE WMCREATE
JMP DEFWNDPROC
WMCREATE:
JMP FINISH
DEFWNDPROC:
PUSH DWORD PTR [EBP+14H]
PUSH DWORD PTR [EBP+10H]
PUSH DWORD PTR [EBP+0CH]
PUSH DWORD PTR [EBP+08H]
CALL DefWindowProcA@16
JMP FINISH
WMDESTROY:
MOV EAX, 0
FINISH:
POP EDI
POP ESI
POP EBX
POP EBP
RET 16
WNDPROCO ENDP
_TEXT ENDS
END START
```

Трансляция программы:

```
ml /c /coff prog.asm
link /subsystem:windows prog.obj
```