МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ГРАЖДАНСКОЙ АВИАЦИИ

Н.И. Черкасова

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА АССЕМБЛЕРЕ ДЛЯ ОС WINDOWS

Москва
2017

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

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА АССЕМБЛЕРЕ ДЛЯ ОС WINDOWS

Утверждено Редакционно-
издательским советом МГТУ ГА
в качестве учебного пособия

Москва-2017

УДК 681.3
ББК 6ф7.3
Ч-48

Печатается по решению редакционно-издательского совета
Московского государственного технического университета ГА

Рецензенты: канд. техн. наук, доц. Л.А. Вайнейкис (МГТУ ГА);
канд. физ.-мат. наук, доц. В.Р. Соловьев (МФТИ)

Черкасова Н.И.
Ч-48 Основы программирования на Ассемблере для ОС Windows: учебное пособие. — М.: МГТУ ГА, 2017. — 83 с., лит.: 10 наим.

ISBN 978-5-903865-22-2

Данные тексты лекций содержат материалы учебно-методического характера, необходимые для освоения знаний и умений по предмету «Основы программирования на Ассемблере для ОС Windows». Содержит материал второй части учебной дисциплины «программирование на машинно-ориентированном языке», в котором рассматриваются вопросы работы с ассемблерами в современных операционных системах.

Данное учебное пособие издается в соответствии с рабочей программой учебной дисциплины «Основы программирования на Ассемблере для ОС Windows» по Учебному плану направления 09.03.01 всех форм обучения.

Рассмотрено и одобрено на заседании кафедры 25.04.17 г. и методического совета 25.04.17 г .

ББК 6ф7.3
Св. тем. план 2017 г.
поз. 34

ЧЕРКАСОВА Наталья Ивановна

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА АССЕМБЛЕРЕ ДЛЯ ОС Windows
Учебное пособие

	Подписано в печать 14.06.2017 г.	
Печать офсетная	Формат 60x84/16	3,58 уч.-изд. л.
4,88 усл.печ.л.	Заказ № 1725/196	Тираж 30 экз.

Московский государственный технический университет ГА
125993 Москва, Кронштадтский бульвар, д. 20
ООО «ИПП «ИНСОФТ»
107140, г. Москва, 3-й Красносельский переулок д.21, стр. 1

Содержание

1. Программирование приложений

Язык программирования Ассемблер – это язык программирования низкого уровня, а вернее семейство языков, максимально приближенных к аппаратному обеспечению компьютера или группа так называемых машинно-ориентированных языков программирования.

Язык программирования – это система обозначений для описания данных и алгоритмов их обработки на компьютере. Программы для первых вычислительных машин составлялись на простейшем из языков программирования – машинном коде, при помощи только двух символов: нуля и единицы. И если вначале программы писали в двоичном коде, то затем в шестнадцатеричной системе счисления.

Написанная на машинном коде программа имеет вид таблицы из цифр, каждая строчка которой соответствует одному оператору – машинной команде. Она определяет компьютеру выполнение определенных действий. Константы и команды являются конструкциями машинного кода. Команда разделяется на группы бит (или поля). Первые несколько бит это поле – код операции (также операционный код, опкод – англ operation code), который определяет действия компьютера. Остальные поля, называемые операндами, идентифицируют требуемую команде информацию, показывают, где именно в памяти компьютера находятся нужные числа (слагаемые, сомножители и тому подобное) и куда следует поместить результат операции (сумму, произведение и так далее). Операнд может содержать данные, часть адреса, косвенный указатель на данные или другую информацию, относящуюся к обрабатываемым командой данным.

опкод Операнд1 … Операнд n

Рис. 1. Общий формат команды

Пример: команда перемещения для микропроцессора 80x86 выглядит так: C605EF00400005. C605 – опкод операции перемещения.

По такой команде компьютер помещает число 05 (две последние цифры 05) в ячейку памяти с номером 004000EF (цифры EF004000).

Двоичная система счисления используется в цифровых устройствах по следующим причинам:

1. наименьшее количество возможных состояний элемента ведет к упрощению конструкции в целом, а также повышает помехоустойчивость и скорость работы;
2. двоичная система счисления позволяет упростить операции сложения и умножения – основных действий над числами.

Следует отметить, что составление программ на машинном коде – достаточно тяжелая и кропотливая работа, требующая чрезвычайного внимания и высокой

квалификации программиста, в программе на машинном коде легко ошибиться и достаточно трудно отыскать ошибку. Расширение или сокращение уже написанных программ также трудоемко и неэффективно.

1.1. Язык программирования Ассемблер

Очевидно, разные типы процессоров имеют различные системы команд. Если язык программирования ориентирован на конкретный тип процессора и учитывает его особенности, то он называется языком программирования низкого уровня. Операторы подобного языка близки к машинному коду и ориентированы на конкретные команды процессора.

Язык ассемблера, как язык низкого уровня, позволяет максимально использовать потенциал процессора и написать программу для любых, в том числе и нестандартных. ситуаций, при этом, оптимизировав ее под требуемую задачу. Он обеспечивает доступ к регистрам, указание методов адресации и описание операций в терминах команд процессора. Язык ассемблера может содержать средства более высокого уровня: встроенные и определяемые макрокоманды, соответствующие нескольким машинным командам, автоматический выбор команды в зависимости от типов операндов, средства описания структур данных.

Главное достоинство языка ассемблера – «приближенность» к процессору, а главным неудобством – слишком мелкое деление типовых операций, которое большинством пользователей воспринимается с трудом. Однако язык ассемблера в значительно большей степени отражает само функционирование компьютера, чем другие языки.

Оптимальной можно считать программу, которая работает правильно, по возможности быстро и занимает, возможно, малый объем памяти. Кроме того, ее легко читать и понимать; ее легко изменить; ее создание требует мало времени и незначительных расходов. В идеале язык ассемблера должен обладать совокупностью характеристик, которые бы позволяли получать программы, удовлетворяющие как можно большему числу перечисленных качеств.

Отметим, что язык ассемблера обладает двумя принципиальными преимуществами:

1. во-первых, написанные на нем программы требуют значительно меньшего объема памяти,
2. во-вторых, выполняются гораздо быстрее, чем программы-аналоги, написанные на языках программирования высокого уровня

1.2. Мнемоники команд

Язык ассемблера представляет каждую команду машинного кода, но не в виде двоичных чисел, а с помощью условных символьных обозначений, называемых мнемониками.

Использование машинных кодов затруднено, поскольку смысл кода команды не ясен без соответствующего справочника, однако код каждой машинной команды можно заменить коротким именем, называемым мнемоническим кодом. Например, код 01000000b или 40h для микропроцессора x86 означает «увеличить содержимое регистра EAX на единицу», выглядит как «INC EAX». Мнемонической командой служит трёх-или четырёхбуквенное сокращение английского глагола (см. табл. 1).

Таблица 1

Мнемоники команд

Мнемоника	Смысл команды	Производное английское слова
jmp	Продолжать выполнение с нового адреса в памяти	jump – прыгнуть
mov	переместить данные	move – переместить
sub	получить разность двух значений	substract – вычесть
xchg	обменять значения в регистрах/ ячейках памяти	exchange – обменять

От ассемблера к ассемблеру может меняться синтаксис аргументов, но мнемоники, обычно, остаются одинаковыми.

Рис. 2. Примеры ассемблерных команд МП IA32/64

Машинный код определяется конструкцией микропроцессора и не изменяется. Мнемоника языка ассемблера разрабатывается изготовителем микропроцессора, чтобы обеспечить удобство программирования и не зависит от конструкции микропроцессора.

Каждый микропроцессор имеет свой собственный машинный код и, следовательно, собственный язык ассемблера (разрабатываемый изготовителем микропроцессора).

Язык ассемблер для микропроцессоров IA32/64 поддерживают два синтаксиса Intel и AT&T. Под Intel-синтаксис разработаны следующие ассемблеры:

MASM (Macro Assembler – Microsoft Corporation), BASM, TASM (Build-in Assembler, Turbo Assembler – Borland Inc), ASM-86 (Intel Corporation), FASM (Flat Assembler – Tomasz Grysztar), LZASM (lazzy assebmler – Степан Половников), WASM (Open Watcom Assembler – фирма Watcom), HLASM, HLA (High Level Assembler – IBM), NASM (NetWide Assembler – Simon Tatham, Julian Hall, Peter Anvin), YASM (Yet Another Assembler – Peter Johnson, Michael Urman), RosAsm (ReactOS Assembler), GoAsm (Jeremy Gordon) и т.д.

Синтаксис AT&T используют AS (UNIX assembler) и GAS (GNU assembler).

Все вышеперечисленные ассемблеры включают стандартные мнемонические команды, в том числе команды арифметических операций для чисел с плавающей запятой для сопроцессора. Несмотря на имеющиеся различия между этими ассемблерами, их структуры и форматы команд языков в значительной мере совместимы. Краткие характеристики ассемблеров представлены в Приложении 1.

1.2.1. Разработка приложения на Ассемблере

Разработка программы на ассемблере обычно включает несколько этапов:

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.

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.

Ассемблера требует помещение определенного количества строк в качестве заголовка программ, то есть нужно записывать несколько псевдооператоров, которые сообщают языку ассемблера основную информацию. Ниже приведен абсолютный минимум, необходимый для программ:

Листинг 1:

```
1       .686P
2       .model flat
3       .code
4       star:
5       ;тело  программы
6       ret
7       .data
8       ;данные программы
9       end star
```

В первой строке .686P – это директива описания типа микропроцессора (может быть еще и .8086, .8087, .186, .286, .287, .386, .387, .486, .586, .mmx с добавлением или без добавления букв P (привилегированные команды) и C или N (непривилегированные команды)). Если не указывать тип микропроцессора, то программа будет сгенерирована в кодах i8086.

Директива описания типа микропроцессора

Представим назначение директив описания типа микропроцессора:

1. .8086 - Разрешены инструкции базового процессора i8086 (и идентичные им инструкции процессора i8088). Запрещены инструкции более поздних процессоров.
2. .186 .286 .386 .486 .586 .686 - Разрешены инструкции соответствующего процессора x86 (x=1,…,6). Запрещены инструкции более поздних процессоров.

3. .187 .287 .387 .487 .587 -Разрешены инструкции соответствующего сопроцессора x87 наряду с инструкциями процессора x86. Запрещены инструкции более поздних процессоров и сопроцессоров.
4. .286c .386c .486c .586c .686c - Разрешены непривилегированные инструкции соответствующего процессора x86 и сопроцессора x87. Запрещены инструкции более поздних процессоров и сопроцессоров.
5. .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.
6. .mmx - Разрешены инструкции MMX-расширения.
7. .xmm - Разрешены инструкции XMM-расширения.
8. .K3D- Разрешены инструкции AMD 3D.

Подробно описание остальных директив, в частности. директивы модель памяти представлено в приложении 2.

В третьей строчке написано - code. По этой команде (директиве) будет определен сегмент кода. Это необходимо, потому что реальные программный код (команды) и данные (переменные) должны быть разделены. Для добавления в каком-либо месте программы данных, определяется .data, а потом уже сами данные (но их можно и не разделять). p\Рекомендуется вставлять данные между командами ret и end start. star: – это место, где находится точка входа в программу. На языке ассемблера всегда надо помечать место, где находится начало программы (первая команда). Строка ret нужна, чтобы выйти из программы, так как если бы ее не было, то программа бы «завесила» компьютер. Последняя строчка end start завершает текст исходной программы.

2. Организация памяти

Под памятью (memory) подразумевается оперативная память компьютера. В отличие от памяти жесткого диска, которую называют внешней памятью (storage), оперативной памяти для сохранения информации требуется постоянное электропитание.

Память является важнейшим ресурсом, требующим тщательного управления со стороны мультипрограммной операционной системы. Особая роль памяти объясняется тем, что процессор может выполнять инструкции только в том случае, если они находятся в памяти. Память распределяется как между модулями прикладных программ, так и между модулями самой операционной системы.

Процессор аппаратно поддерживает структурную организацию программы в виде трех частей, расположенных в памяти в так называемых сегментах. Такая организация памяти называется сегментной. Их существование обусловлено спецификой организации и использования оперативной памяти архитектуры Intel .

В архитектуре Intel аппаратно поддерживает две модели использования оперативной памяти, где оперативная память– это физическая память, к которой процессор имеет доступ по шине адреса.

1. Сегментированная модель. В этой модели программе выделяются непрерывные области памяти (сегменты), а сама программа может обращаться только к данным, которые находятся в этих сегментах.
2. Странично-сегментная модель. Ее можно рассматривать как надстройку над сегментированной моделью памяти. В случае использования этой модели памяти, оперативная память рассматривается как совокупность блоков фиксированного размера. Основное применение этой модели связано с организацией виртуальной памяти, что позволяет операционной системе использовать для программ пространство памяти большее, чем объем физической памяти.

Сегментация – это механизм адресации, обеспечивающий существование нескольких независимых адресных пространств, как в пределах одной задачи, так и в системе в целом для защиты задач от взаимного влияния. В основе механизма сегментации лежит понятие сегмент, который представляет собой независимый поддерживаемый на аппаратном уровне блок памяти.

Каждая программа может состоять из любого количества сегментов (в зависимости от организации модели памяти), но реально имеется доступ к трем основным сегментам памяти - сегмент кода, сегмент данных, сегмент стека. Программа никогда не знает, по каким физическим адресам размещены ее сегменты. Этим занимается ОС. ОС размещает сегменты программы по определенным физическим адресам, после чего помещает значения этих адресов в определенные места. Куда именно поместит сегменты программы ОС зависит от работы процессора. Так, в реальном режиме эти адреса помещаются непосредственно в соответствующие сегментные регистры, а в защищенном режиме они размещаются в элементы специальной системной дескрипторной таблице. Внутри сегмента программа обращается к адресам относительно начала сегмента линейно, т.е. начиная с 0 и заканчивая адресом, равным размеру сегмента. Это относительный адрес или смещение, который используется для доступа к данным внутри сегмента, называется эффективным.

Будем различать три основных модели сегментированной организации памяти:

1. сегментированная модель памяти реального режима,
2. сегментированная модель памяти защищенного режима,
3. сплошная модель памяти защищенного режима.

2.1. Порядок формирования физического адреса в реальном режиме.

Физический адрес – это адрес памяти, выдаваемый на шину адреса микропроцессора или линейный адрес.

2.2. Сегментированная модель памяти реального режима.

В реальном режиме механизм адресации физической памяти имеет следующие характеристики:

1. диапазон изменения физического адреса от 0 до 1Мбайт. Эта величина определяется тем, что шина адреса i8086 имела 20 линий.
2. максимальный размер сегмента 64 Кбайта. Это объясняется 16-разрядной архитектурой I8086. Нетрудно подсчитать, что максимальное значение, которое могут содержать 16-разрядные регистры, составляют 216-1, что применительно к памяти и определяет величину 64 Кбайт.
3. для обращения к конкретному физическому адресу оперативной памяти необходимо определить адрес начала сегмента (сегментную составляющую) и смещение внутри сегмента. В сегментном регистре находятся только старшие 16 бит физического адреса начала сегмента. недостающие 4 бита 20-битового адреса получаются путем сдвига значения в сегментном регистре влево на 4 разряда.

Получение 20-битового адреса выполняется аппаратно и для программного обеспечения абсолютно прозрачно. Полученный адрес соответствует адресу начала сегмента. Вторая составляющая адреса – смещение, которое представляет собой 16-битовое значение. Это значение может присутствовать в команде в явном или косвенном виде. Получение смещения зависит от способа адресации, используемой в команде.

Недостатками такой организации памяти являются:

1. сегменты бесконтрольно размещаются с любого адреса, кратного 16 (т.к. содержимое сегментного регистра аппаратно смещается на 4 разряда). В результате чего, программа может обращаться по любым адресам, в том числе и реально не существующим.
2. сегменты имеют максимальный размер 64 Кбайт
3. сегменты могут перекрываться с другими сегментами.

Использование сегментных регистров называлось на компьютерном жаргоне словом kludge (приспособление для временного устранения проблемы). Проблема заключалась в адресации более 64 Кбайт памяти – предела, который устанавливается использованием 16-битных регистров, так как 216=65535 – это наибольшее число, которое может содержать такой регистр. Специалисты фирмы Intel для решения этой проблемы использовали сегменты и регистры сегментов и тем самым усложнили процесс адресации в микропроцессорах x86. Хотя адрес формируется из двух 16-разрядных регистров, микропроцессор i8086 (а после i80186, i80286) не формировал для адреса 32-разрядное число, а обходился 20-разрядным адресом. Отметим, что микропроцессор разбивает память на перекрывающиеся сегменты. Сегмент – это 64-Кбайтный участок памяти. Каждый новый сегмент начинается через каждые 16 байт. Первый сегмент (сегмент #0) начинается в памяти с ячейки #0(=0000.0000b); второй (сегмент #1) начинается с ячейки #16

(10h=0001.0000b); третий – с ячейки #32 (20h=0010.0000b) и так далее. Таким образом, начальный адрес сегмента памяти всегда кратен 16 и последние 4 бита у этого адреса нулевые. В сегментных регистрах хранят только первые 16 битов начального адреса сегмента памяти. Для микропроцессора i8086 полный 20-разрядный (абсолютный или физический) адрес получался сложением содержимого сегментного регистра, умноженное на 16 (начального адреса сегмента) и содержимого какого-нибудь 16-разрядного регистра общего назначения, указывающего на смещение внутри сегмента (эффективный адрес или смещение).

Физический адрес = сегмент*16 + смещение.

Например, адресная пара регистр ES=1234h и регистр DI=0053h задают следующий абсолютный адрес:

ES: DI=10h*1234h+0053h=12340h+53h=12393h.

Отметим, что при других значениях в ES=1000h и DI=2393h мы получим тот же абсолютный адрес

ES: DI=10h*1000h+2393h=10000h+2393h=12393h.

При сложении максимальных значений сегмента и смещения представленная формула даст адрес 0FFFF0h+0FFFFh=10FFEFh, но из-за 20-разрядного ограничения на шину памяти эта комбинация указывает на адрес 0FFEFh. Таким образом, получается совмещение адресов – адреса от 100000h до 10FFEFh соответствовали адресам от 0 до 0FFEFh. Начиная с i80286 шина адреса увеличена до 24 разрядов, i80386 до 32, а с Pentium Pro до 36 разрядов. Меняются и способы адресации к памяти. Микропроцессор x86 позволяет 24 способа адресации.

2.3. Защищенный режим

Для того чтобы процессы не мешали друг другу, требуются меры принудительной защиты критических ресурсов, к которым прежде всего относится оперативная память. Современные операционные системы используют защищенный режим работы процессора, в котором подобная защита реализуются на аппаратном уровне. Поскольку программа может взаимодействовать с подсистемами компьютера только через пространства памяти и портов ввода-вывода, а также аппаратные прерывания, то защищать нужно эти три типа ресурсов.

Рассмотрим самую сложную защиту - память. Операционная система выделяет каждому процессу области памяти – сегменты – различного назначения и с разными правами доступа. Из одних сегментов можно только читать данные, в другие возможна и запись. Для программного кода выделяются специальные сегменты, инструкции могут выбираться и исполняться только из них.

Процессор трактует содержимое ячейки памяти, на которую передалось управление, как код инструкции или префикс. Если ошибочно управление

передалось на область данных, то дальнейшее поведение процессора непредсказуемо. Защита не позволяет передать управление на сегмент данных – сработает исключение защиты, которое обрабатывается операционной системой, и ошибочный процесс будет принудительно завершен. Чтобы выдержать принцип сохранности программы, на время ее загрузки в память или программной модификации ту же область объявляют сегментом данных, в который разрешена запись. Система защиты может полностью контролировать распределение памяти, генерируя исключения в случаях различных нарушений. Эффективность защиты (устойчивость компьютера к ошибкам) в том числе определяется и предусмотрительностью разработчиков операционной системы.

Отметим, что чем сложнее программа и больше объем обрабатываемых ею данных, тем больше ее потребности в памяти. В первых процессорах семейства память предоставлялась в виде сегментов с размером по 64 Кбайт, а суммарный объем программно-адресуемой памяти не превышал значения в 1 Мбайт. Архитектура PC ограничивала размер оперативной памяти объемом в 640 Кбайт, начиная с нулевых адресов. Эта область называется стандартной памятью (conventional memory), и для прикладных программ из нее остается доступной область порядка 400–550 Кбайт (остальное занимала операционная система с драйверами). Потребности решаемых задач довольно быстро переросли эти ограничения, и в процессоры ввели средства организации виртуальной памяти. Впервые они появились в 80286, но удобный для употребления вид приняли только в 32-разрядных процессорах (80386 и выше). Во-первых, было снято ограничение на 64 Кбайтный размер сегмента – теперь любой сегмент может иметь почти произвольный размер до 4 Гбайт. Во-вторых, был введен механизм страничной переадресации памяти (paging). Теперь любая страница (область фиксированного размера) виртуальной логической памяти (адресуемой программой в пределах выделенных ей сегментов) может отображаться на любую область физической памяти (реально установленной оперативной). Отображение поддерживается с помощью специальных таблиц страничной адресации, в которых кроме связи адресов есть указание на присутствие страницы в физической памяти на данный момент времени. Теперь страница памяти, не нужная процессору в данный момент времени, может быть выгружена на устройство хранения (диск). На ее место можно загрузить нужную страницу. Заявку на загрузку-выгрузку нужной страницы делает сам процессор, без каких-либо усилий выполняемой программы: если программе потребовалась ячейка виртуальной памяти из страницы, образа которой сейчас нет в физической памяти, вырабатывается специальное исключение. Обработчик этого исключения (это часть ОС) найдет свободную физическую страницу (выгрузив на диск ту, которая, по его мнению, пока не нужна), «подкачает» на нее с диска требуемую информацию и вернет управление процессу, прерванному исключением. Этот процесс проходит незаметно для программы (кроме некоторой задержки выполнения инструкций). Таким образом, в распоряжение всех процессов, исполняемых на

компьютере псевдопараллельно, предоставляется виртуальная оперативная память, размер которой ограничен суммой объема физической оперативной памяти и областью дисковой памяти, выделенной для подкачки страниц. Память может быть логически организована в виде одного или множества сегментов переменной длины (в реальном режиме – фиксированный).

Регистры дескриптора сегмента

Дескрипторы сегментов – это специальные указатели, определяющие расположение сегмента в памяти.

Регистры дескриптора сегмента невидимы для программиста, однако их содержание очень полезно знать при этом, в x86 регистр дескриптора соотнесен с каждым регистром селектора.

Каждый из них содержит 32-битовый базовый адрес сегмента, его границу (предел) и другие необходимые свойства сегмента. Когда адрес сегмента загружается в сегментный регистр, ассоциативный (соотнесенный) регистр дескриптора автоматически модифицируется в соответствии с новой информацией. В режиме реальной адресации только базовый адрес модифицируется напрямую, путем сдвига его значения на четыре разряда влево, поскольку максимальная граница и признаки сегмента фиксированы. В защищенном режиме базовый адрес, граница, все признаки модифицируются содержимым регистра дескриптора сегмента, индексированного селектором. Каждый раз, когда происходит ссылка на ячейку памяти, регистр дескриптора сегмента автоматически вовлекается со ссылкой на ячейку памяти. 32-битовый базовый адрес сегмента становится компонентом вычисления линейного адреса, 32-битовое значение границы используется для операций контроля границы, а признаки проверяются на соответствие типу ссылки на ячейку памяти, которая запрашивается.

3. Регистры микропроцессора

Регистры являются составной частью процессора и используются для временного хранения информации. Интенсивное использование регистров микропроцессором при работе с программой определяется тем, что скорость доступа к ним намного больше, чем к ячейкам памяти. 32-/64-битные процессоры имеют набор регистров для хранения данных общего назначения; набор сегментных регистров; набор из 8 80-битных регистров для работы с числами с плавающей точкой (ST0-ST7); набор из 8 64-битных регистров целочисленного MMX-расширения (MMX0-MMX7, имеющих общее пространство с регистрами ST0-ST7); набор из 16 128-битных регистров SSE для работы с числами с плавающей точкой (XMM0–XMM15); программный стек – специальная информационная структура, работа с которой предусмотрена на уровне машинных команд. Помимо основных регистров из реального режима доступны также регистры управления памятью (GDTR,

IDTR, TR, LDTR) регистры управления (CR0-CR4), отладочные регистры (DR0-DR7) и машинно-специфичные регистры.

3.1. Регистры общего назначения

Различают 8 целочисленных 32-х разрядных регистров общего назначения (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP) их 64-разрядные расширения (RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP) и 8 целочисленных 64-разрядных регистров общего назначения (R8-R15), которые могут хранить следующие типы данных:

1. операнды для логических и арифметических операций;
2. операнды для расчета адресов;
3. указатели на ячейки памяти.

Хотя для хранения операндов, результатов операций и указателей можно использовать любой из вышеперечисленных регистров, требуется осторожность в работе с регистром ESP/RSP. В нем хранится указатель вершины стека, и некорректное изменение этого значения приведет к неправильной работе программы и ее аварийному завершению.

Многие команды используют конкретные регистры для хранения своих операндов. Например, команды обработки текстовых строк используют содержимое регистров ECX/RCX, ESI/RSI и EDI/RDI в качестве операндов.
Основные случаи использования регистров общего назначения:

EAX/RAX – используется для хранения операндов и результатов операций;

EBX/RBX – как указатель на данные в сегменте DS;

ECX/RCX – как счетчик для строковых операций и циклов;

EDX/RDX – указатель для ввода/вывода, по умолчанию регистры EAX/RAX и EDX/RDX используются для умножения и деления;

ESI/RSI – указатель на данные в сегменте DS, а также как указатель на источник в командах работы со строками;

EDI/RDI – указатель на данные в сегменте ES, а также как указатель на приемник в командах работы со строками;

ESP/RSP – указатель вершины стека в сегменте SS;

EBP/RBP – указатель на некоторые данные в стеке.

К регистрам, оканчивающимся на X, можно обращаться и как целиком к 64-битным (RAX, RBX, RCX, RDX), и к их младшей 32-битной части (EAX, EBX, ECX, EDX), и к младшим 16-ти битам (AX, BX, CX, DX), которые в свою очередь можно разделить на старший/младший байт (AH/AL, BH/BL, CH/CL, DH/DL) и работать с ними, как с регистрами длиной 8 бит. Регистры-указатели RSP/ESP (указатель вершины стека) и RBP/EBP (базовый регистр), а также индексные регистры RSI/ESI (индекс источника) и RDI/EDI (индекс приемника) допускают либо 64-/32-битное обращение, либо обращение только к младшим 16 битам (SP, BP, SI и DI), либо к младшим 8 битам (SPL, BPL, SIL, DIL). Для

обращения к младшим 8/16/32-битам регистров R8-R15 используют суффиксы b/w/d.

Например, R9b – младший байт 64-разрядного регистра R9 (по аналогии с AL), R9w – младшее слово регистра R9 (то же, что AX в EAX). Для обращения к старшей части регистров РОН используют сдвиги или математические операции.

3.2. Сегментные регистры

Сегментные регистры (CS, DS, SS, ES, FS и GS) хранят 16-битные базовые адреса сегментов, определяющие сегменты памяти текущей адресации.

В защищенном режиме каждый сегмент может иметь размеры от одного байта до целого линейного и физического пространства машины до 4 Гб в режиме реальной адресации, максимальный размер сегмента ограничен на 64 Кб.

Шесть сегментов, адресуемых в любой данный момент, определяются содержимым регистров CS, SS, DS, ES, FS и GS:

1. значение в CS (Code Segment Register) указывает на сегмент, содержащий команды программы;
2. содержимое DS (Data Segment Register) указывает на сегмент, содержащий обрабатываемые программой данные;
3. содержимое SS (Stack Segment Register) сегмент, содержащий стек программы;
4. значения в ES, GS указывают на дополнительные сегменты данных;
5. регистр FS содержит блок данных потока (thread information block TIB). Этот блок описывает выполняющийся в данный момент поток.

Таблица 2

Адреса регистров в двоичном формате

8-битный операнд	16-битный операнд	32-битный операнд	64-битный операнд	Адреса регистров в двоичном формате
AL	AX	EAX	RAX	000b
CL	CX	ECX	RCX	001b
DL	DX	EDX	RDX	010b
BL	BX	EBX	RBX	011b
AH/SPL	SP	ESP	RSP	100b
CH/BPL	BP	EBP	RBP	101b
DH/DIL	SI	ESI	RSI	110b
BH/SIL	DI	EDI	RDI	111b

3.3. Внутренние регистры

Общее количество внутренних регистров микропроцессора зависит от его модели. Первоначально (до появления суперскалярного микропроцессора) регистры были не виртуальными, на один программный регистр приходилось по одному внутреннему регистру, затем – по 2 внутренних регистров на один программный и так далее. Количество внутренних регистров (RF) больше, чем программных. Регистры RF не привязаны к программным регистрам. Регистры RF привязаны к операциям (точнее к микрооперациям – мопам) – каждому мопу назначается новый регистр RF для записи результата. Поэтому и количество регистров зависит от того, сколько мопов могут одновременно находиться в конвейере процессора – чем длиннее конвейер, тем больше должно быть RF и ROB (reorder buffer, в котором временно хранятся мопы). Поскольку современные процессоры суперскалярные и рассчитаны на обработку в среднем до трех мопов за такт (Core 2 Duo до 4), то количество RF и ROB должно быть не меньше утроенной задержки (в тактах) прохождения мопа по конвейеру с выхода декодера (или Т-кэша) до выхода в отставку (удаления из ROB). Например, в Pentium Pro-Pentium III количество RF и ROB – 40, а в Pentium 4E конвейер в 3 раза длиннее и соответственно RF и ROB – 128.

3.4. Регистр флагов и команды управления флагами

Регистр флагов — регистр процессора, отражающий текущее состояние процессора. Регистр флагов содержит различные биты, отражающие текущее состояние процессора и частично управляющие его работой.

Регистр флагов был уже в первом 16-разрядном микропроцессоре Intel — 8086. Он носил имя FLAGS и имел длину 16 бит. Появление микропроцессора 80286 добавило в регистр FLAGS бит NT и битовое поле IOPL, однако сам регистр оставался 16-разрядным.

С выходом микропроцессора 80386 регистр был расширен до 32 бит и стал называться EFLAGS. В нём появились ещё два дополнительных бита: VM и RF. В последующих поколениях 32-разрядных микропроцессоров были добавлены биты AC, VIF, VIP и ID. Тем не менее, сохранялась полная совместимость «снизу вверх»: любая старая программа, которая корректно работала с регистром EFLAGS, не модифицируя «неизвестные» ей биты, оставалась работоспособной и на последующих процессорах.

Расширение разрядности процессора до 64 бит удвоило и размер регистра флагов, получившего название RFLAGS. Однако его старшая половина (биты с 32-го по 63-й включительно) пока не используется и является зарезервированной, поэтому формат «значащей» части RFLAGS полностью совпадает с форматом EFLAGS.

Формат регистра EFLAGS выглядит следующим образом:

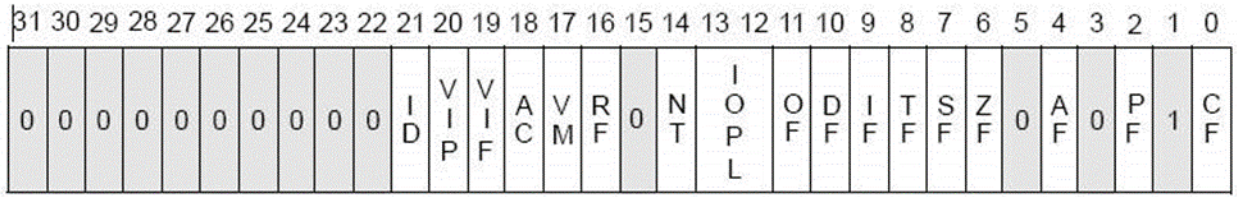

Рис.3 - Формат регистра EFLAGS.

Все неиспользуемые (зарезервированные) биты равны нулю. Исключением является бит 1, который всегда имеет единичное значение.
Флаги подразделяются на следующие виды:

- **Флаги состояния.**

Флаги отражают результат, полученный в предыдущей арифметико-логической операции. Многие из них могут использоваться в дальнейшем для выполнения условных переходов или условных пересылок.

Флаги этой группы могут изменяться любыми программами с помощью команд, заносящих в регистр флагов новое значение (SAHF, POPF/POPFD/POPFQ).

Флаги состояния отражают результат выполнения арифметических инструкций, таких как ADD, SUB, MUL, DIV:

CF Carry Flag - Флаг переноса (бит 0). Устанавливается при переносе из/заёме в (при вычитании) старший значащий бит результата и показывает наличие переполнения в беззнаковой целочисленной арифметике. Также используется в длинной арифметике.

PF Parity Flag - Флаг чётности (бит 2). Устанавливается, если младший значащий байт результата содержит чётное число единичных (ненулевых) битов. Изначально этот флаг был ориентирован на использование в коммуникационных программах: при передаче данных по линиям связи для контроля мог также передаваться бит чётности и инструкции для проверки флага чётности облегчали проверку целостности данных.

AF Auxiliary Carry Flag - Вспомогательный флаг переноса (бит 4). Устанавливается при возникновении переноса или заёма из 4-ого раззряда в 3-ий разряд. Сбрасывается при отсутствии такового. Используется командами десятичной коррекции.

ZF Zero Flag - Флаг нуля (бит 6). Устанавливается при получении нулевого результата, сбрасывается в противном случае.

SF Sign Flag - Флаг знака (бит 7). Устанавливается, если в результате операции получено отрицательное число, т.е. если старший разряд результата равен единице. В противном случае сбрасывается

OF Overflow Flag - Флаг переполнения (бит 11). Устанавливается, если в результате арифметической операции зафиксировано знаковое переполнение,

то есть если результат, рассматриваемый как число со знаком, не помещается в операнд-приёмник. Если знакового переполнения нет, этот флаг сбрасывается

- **Флаг направления.**

Может изменяться любыми программами с помощью специальных инструкций CLD и STD, а также загрузкой нового содержимого регистра флагов.

DF Direction Flag - Флаг направления (бит 10). Когда этот флаг сброшен, строковые операции (MOVS, CMPS, SCAS, LODS и STOS) обрабатывают данные в порядке возрастания адресов (увеличивая содержимое регистров SI/ESI/RSI и DI/EDI/RDI после каждой итерации), а когда установлен — в порядке убывания адресов (уменьшая содержимое указанных регистров после каждой итерации)

- **Системные флаги**.

Изменяются только системным программным обеспечением, работающим при нулевом текущем уровне привилегий. Исключением является флаг IF, состояние которого в определённых условиях может меняться менее привилегированным кодом. Загрузка регистров флагов программами, выполняющимися не на нулевом уровне привилегий, не изменяет значения системных флагов.

TF Trap Flag - Флаг трассировки (пошаговое выполнение) (бит 8). Когда установлен, вызывает прерывание #DB (вектор 1) после выполнения каждой команды процессора за исключением той, которая осуществила установку этого флага.

IF Interrupt Enable Flag - Флаг разрешения прерываний (бит 9). Когда установлен, процессор обрабатывает маскируемые прерывания, запросы на которые поступают от контроллера прерываний или улучшенного контроллера прерываний. Когда сброшен, маскируемые прерывания процессором игнорируются (находятся в ожидании обработки). Возможностью установки и сброса флага IF программами, выполняющимися не на нулевом уровне привилегий, управляют поле IOPL регистра флагов и бит VE управляющего регистра CR3(Может бит VME регистра CR4)

IOPL I/O Privilege Level - Уровень приоритета ввода-вывода (биты 12 и 13). Содержит уровень привилегий ввода-вывода для выполняемой в настоящий момент программы. Если текущий уровень привилегий (CPL) программы численно меньше либо равен значению, находящемуся в поле IOPL, программа может использовать инструкции ввода-вывода, а также менять состояние флага IF. Поле IOPL дополняется картой разрешения ввода-вывода в TSS, а также битом VME в управляющем регистре CR4

NT Nested Task - Флаг вложенности задач (бит 14). Этот флаг устанавливается процессором автоматически при переключении на новую задачу. Переключение на задачу происходит при использовании её в качестве обработчика прерывания через шлюз задачи в IDT или при её вызове командой CALL, обращённой к TSS или шлюзу задачи, находящимся в GDT или LDT RF

Resume Flag - Флаг возобновления (бит 16). Управляет реакцией процессора на точки останова. Когда установлен, запрещает генерацию отладочных прерываний (#DB). Основной функцией флага RF является обеспечение повторного выполнения инструкции после возникновения отладочного прерывания, вызванного точкой останова. Для этого отладчик перед возвратом к прерванной программе должен установить этот бит в регистре флагов прерванной программы, сохранённом в стеке, что обеспечит нормальное выполнение инструкции, к которой произойдёт возврат из отладчика. После её выполнения процессор автоматически сбросит флаг RF, что обеспечит нормальное функционирование установленных точек останова

VM Virtual-8086 Mode - Режим виртуального процессора 8086 (бит 17). Когда этот флаг установлен, процессор работает в режиме виртуального процессора 8086

AC Alignment Check - Проверка выравнивания (бит 18). Когда одновременно установлены этот флаг и бит AC управляющего регистра CR0, а программа выполняется на третьем уровне привилегий, активизируется проверка выравнивания операндов, расположенных в памяти. При попытке доступа к невыровненному операнду возникает исключение #AC (вектор 17)

VIF Virtual Interrupt Flag - Виртуальный флаг разрешения прерывания (бит 19). Этот флаг используется совместно с флагом VIP, если в управляющем регистре CR4 установлен бит VME, разрешающий использование расширений режима виртуального процессора 8086

VIP Virtual Interrupt Pending - Ожидающее виртуальное прерывание (бит 20).

Этот флаг используется совместно с флагом VIF, если в управляющем регистре CR4 установлен бит VME, разрешающий использование расширений режима виртуального процессора 8086

ID ID Flag - Проверка на доступность инструкции CPUID (бит 21). Возможность программно устанавливать и сбрасывать этот флаг служит признаком того, что процессор поддерживает инструкцию CPUID

Пример работы команд и вывод флагов на экран представлен в Приложении 2

Команды управления флагами:

Флаг переноса CF

Команда CLC сбрасывает флаг CF.

Команда STC устанавливает флаг CF в единицу.

Команда CMC инвертирует значение флага CF.

Флаг направления DF

Команда CLD сбрасывает флаг DF.

Команда STD устанавливает флаг DF в единицу.

Флаг прерывания IF
Команда CLI сбрасывает флаг IF (запрещает прерывания).
Команда STI устанавливает флаг IF в единицу (разрешает прерывания).

Команды LAHF и SAHF
LAHF загружает младший байт регистра флагов в AH
SAHF загружает содержимое AH в младший байт регистра флагов

Команды PUSHF и POPF/POPFD/POPFQ
PUSHF поместить FLAGS в стек
POPF загрузить FLAGS из стека
POPFD загрузить вершину стека в EFLAGS
POPFQ загрузить в вершину стека и с нулевой протяженностью в RFLAGS

Команда SALC
Устанавливает al в 0FFh, если CF=1, иначе устанавливает в 00h

Отметим, что не все операции устанавливают флаги. Примем за приблизительное правило, что:

1. арифметические операции действуют на все флаги,
2. логические операции (кроме операции NOT) сбрасывают флаги переноса и переполнения (OF и CF) и действуют на все другие флаги,
3. операции приращения и уменьшения на 1 (команды INC и DEC) не действуют на флаги переноса и действуют на все другие флаги (нельзя использовать флаг переноса для контроля переполнения при операции уменьшение на 1, но можно использовать флаг знака).

Не изменяют флаги:

1. все операции перемещения (MOV, MOVZX, MOVSX, LEA, LDS, LES, LFS, LGS, LSS, XCHG, BSWAP),
2. все операции с портами (IN, OUT, INS, OUTS),
3. все команды перехода (JMP, Jcc, JCXZ, JECXZ),
4. все команды преобразования (CBW, CWD, CWDE, CDQ),
5. все строковые команды (кроме SCAS и CMPS),
6. все операции со стеком (кроме POPF).

3.5. Системные регистры

Дополнительные регистры, введенные в микропроцессор, связаны с работой в защищенном режиме:
CR0-CR4 – 32-разрядные управляющие регистры;
LDTR – регистр локальной таблицы дескрипторов;
GDTR – регистр глобальной таблицы дескрипторов;
IDTR – регистр дескрипторной таблицы прерываний;

TR – регистр задачи;
DR0-DR7 – отладочные регистры;
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.

3.6. Указатель команд

В регистре IP/EIP/RIP (instruction pointer, указатель команд) всегда находится адрес команды, которая должна быть выполнена следующей. Более точно, в IP/EIP/RIP находится адрес этой команды, отсчитанный от начала сегмента команд, на начало которого указывает регистр CS. Поэтому абсолютный адрес этой команды определяется парой регистров CS и IP/EIP. За исключением случаев перехода, последовательно выполняемые команды в памяти следуют друг за другом, содержимое регистра IP/EIP/RIP увеличивается после выполнения каждой команды.

В ранних микропроцессорах (до i8086) выборка команд осуществлялась следующим образом: когда микропроцессор был готов к выполнению следующей команды, он посылал содержимое IP по адресной шине во внешнюю память, считывал в соответствующей адресу ячейки памяти 1 байт данных и по шине данных пересылал его в регистр IP. Затем микропроцессор действовал в соответствии с командой и в процессе ее выполнения мог один или несколько раз обращаться к внешней памяти. Одна машинная команда может иметь длину более одного байта. В зависимости от длины команды (для микропроцессоров младше i80386 от 1 до 6 байт, для i80386 и старше от 1 до 15 байт) содержимое увеличивается на 1, 2 или более байт, если это не команда перехода. (При выполнении команды перехода содержимое CS и IP/EIP/RIP может измениться на любую величину.)

Очередь команд. Для увеличения скорости выполнения программ в микропроцессоре i8086 регистр IP был дополнен 6-байтной очередью команд, организованной по принципу FIFO («первым пришел – первым ушел»). Очередь команд непрерывно заполняется только тогда, когда системная шина не требуется для других операций. Если системная шина занята – команды выбираются из очереди команд.

Конвейеризация вычислений. С появлением микропроцессора i80486 для еще большего увеличения скорости выполнения программ очередь команд была дополнена конвейером. Конвейер – специальное устройство, реализующее такой метод обработки команд внутри микропроцессора, при котором исполнение команды разбивается на несколько этапов. i80486 имел пятиступенчатый конвейер. Соответствующие пять этапов включали:

1. выборку команды из очереди команд;
2. декодирование команды;
3. генерацию адреса, при котором определяются адреса операндов в памяти;
4. выполнение операции с помощью АЛУ;
5. запись результата (куда будет записан результат, зависит от алгоритма работы конкретной машиной команды).

Таким образом, на стадии выполнения каждая машинная команда как бы разбивается на более элементарные операции.

4. Программирование в среде Windows

4.1. Особенности приложений для ОС Windows

Операционные системы Windows обладает рядом особенностей. Прежде всего – это графический интерфейс, обеспечивающий пользователю удобство в работе и привлекательное графическое изображение. 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. В таблице представлены основные системные библиотеки динамической компоновки.

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 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.

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

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.

API это программный интерфейс приложения. Другими словами, это те возможности, которые предоставляет операционная система Windows для использования прикладными программами. Системные функции, которые предоставляет Windows программисту, называются ещё функциями API. Программирование с использованием только этих функций называется API-программированием.

Структура API-программ

Классическая структура API-программы определяется четырьмя компонентами: инициализация; цикл ожидания, или цикл обработки сообщений; функция главного окна; другие функции. В простейшем случае последний компонент может отсутствовать. Два первых компонента располагаются в функции WinMain.

Функция WinMain:

```
int WINAPI WinMain
(
HISTANCE hInstance,
HINSTANCE hPrevInctance,
LPSTR lpCmdLine,
int nCmdShow
).
```

Функция WinMain вызывается системой, в которую передаются четыре параметра:

1. hInstance - дескриптор текушего экземпляра приложения;
2. hPrevInctance - всегда равен NUL;
3. lpCmdLine - указатель на командную строку запускаемой программы;
4. nCmdShow - способ визуализации окна.

Инициализация

Здесь производится регистрация класса окна, его создание и вывод на экран. Регистрация класса окна осуществляется функцией

RegisterClass(CONST WNDCLASS *lpwcx).

Единственный параметр функции - указатель на структуру WNDCLASS. После того как класс будет зарегистрирован, окно из данного класса может быть создано функцией CreateWindow.

```
typedef struct _WNDCLASS
{
UNIT    style;
```

```
WNDPROC lpfnWndProc;
int  cbClsExtra;
int  cbWndExtra;
HANDLE  hInstance;
HICON  hIcon;
HCURSOR hCursor;
HBRUSH  hbrBackground;
LPCTSTR lpszMenuName;
LPCTSTR lpszClassName;
} WNDClASS
```

Перечислим некоторые типичные значения членов структуры:

1. Стили класса окон. Стиль окна определяется комбинацией нескольких предопределённых констант. Довольно часто он полагается нулю, что означает "стиль по умолчанию".

2. Дескриптор иконки окна. Определяется с помощью функции LoadIcon. Первым параметром данной функции является дескриптор приложения, вторым - строка, определяющая имя иконки в ресурсах. Для того чтобы задать одну из стандартных иконок, первый параметр должен иметь значение NULL, а второй значение одной из следующих констант:

- IDI_APLICATION - стандартная иконка приложения;
- IDI_ASTERISK - иконка "информация";
- IDI_EXCLAMATION - "восклицательный знак";
- IDI_HAND - "знак Стоп";
- IDI_QUESTION - "вопросительный знак".

3. Дескриптор курсора. Для определения курсора используется API-функция LoadCursor. Функция похожа на функцию LoadIcon.

4. Имя класса. Название класса - это просто строка, которая потом используется при создании окна.

Создаётся окно функцией CreteWindow:

```
HWND CreateWindow
(
 LPCTSTR lpClassName, //указывает на имя зарегистрированного окна;
 LPCTSTR lpWindowName, //название окна;
 DWORD dwStyle, //стиль окна:
 int x, //горизонтальная координата;
 int y, //вертикальная координата;
 int nWidth, //ширина окна;
 int nHeight, //высота окна;
 HWND hWndParent, //дескриптор родителя или владельца окна;
 HMENU hMenu, //дескриптор меню окна;
 HANDLE hINSTANCE, //дескриптор приложения;
 LPVOID lpParam //указатель на дополнительную информацию;
 ).
```

Функция возвращает дескриптор созданного окна, при ошибке – 0.

Для того чтобы корректно отобразить окно на экране, следует выполнить ещё две функции:

3. BOOL ShowWindow(HWND hWnd, int nCmdShow) - эта функция отображает окно на экране. Первый параметр - дескриптор окна, второй - режим отображения.
4. BOOL UpdateWindow(HWND hWnd) - вызов данной функции приводит к немедленной перерисовке окна и посылке функции окна сообщения WM_PAINT.

Цикл обработки сообщений

Цикл обработки сообщений присутствует во всех приложениях Windows. Правда, не всегда этот цикл представлен явно в программе.

```
while (GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

В цикле сообщения присутствует три функции. Эти функции есть там всегда, но кроме них в цикле могут быть и другие. Функция GetMessage() выбирает из очереди сообщений приложения очередное приложение. вместо этой функции используют так же функции PostMessage() и PeekMessage(). Во всех трех функциях присутствует указатель на строку MSG:

```
typedef struct tagMSG
{
HWND hwnd;
UINT message;
WPARAM wParam;
LPARAM lParam;
DWORD time;
POINT pt;
}
```

MSG:

1. hwnd - дискриптор окна;
2. message - код сообщения;
3. wParam - дополнительный параметр;
4. lParam - дополнительный параметр;
5. time - время посылки сообщения;
6. pt - положение курсора мыши;
7. Прототип функции MessageBox().

```
BOOL
GetMessageBox
(
```

LPMSG lpMsg;
HWND hWnd;
UINT wMsgFilterMin,
UINT wMsgFilterMax;
).

Первый параметр функции - указатель на строку MSG, куда и будет помещена получаемая информация. Вторым параметром является дескриптор окна, которому предназначено сообщение. Если параметр равен NULL, то "отталкиваются" все сообщения, получаемые приложением. Два последних параметра определяют диапазон сообщений. Для того чтобы получать сообщения из всего диапазона, эти параметры должны быть равны 0.
Функция TransleteMessage() преобразует сообщения в единый формат. Функция DispatchMessage() перенаправляет сообщение оконной процедуре.

Оконная функция

Это еще один компонент приложения, отвечающий за обработку сообщений окна. Эта функция вызывается системой и имеет четыре параметра, совпадающих с первыми членами структуры MSG. В приложении представлен листинг программы, создания простого окна приложения (пример 4).

С 1991 года операционная система Windows, созданная компанией Microsoft сменила несколько версий. За это время она выпускалась и для 16-разрядных процессоров и для 32-разрядных. А сегодня уже появились версии Windows для работы с 64-разрядными процессорами. Менялся не только дизайн системы, но и принципы ее внутренней архитектуры. Неизменным по сей день остается одно - наличие динамических библиотек (DLL), являющихся основными поставщиками функций (Application Programming Interface - API) для работы системы и приложений. С самого начала необходимо усвоить один важный момент: API - не то же самое, что операционная система, но конкретный API можно использовать только на определенных версиях Windows. В таблице 3 представлены основные системные библиотеки API-функций и их описание.

Таблица 3

Основные библиотеки

Библиотека	**Описание**
kernel32.dll	Системные функции низкого уровня. В этой библиотеке находятся функции управления памятью, задачами, распределения ресурсов и т.д.
user32.dll	Функции, управляющие работой Windows. В этой библиотеке находятся функции для работы с сообщениями, меню, указателями мыши, курсорами, таймерами и большинство других функций, не связанных с выводом на экран.
gdi32.dll	Библиотека интерфейса графических устройств (GDI). Содержит функции, связанные с выводом на устройства. В ней

	находится большинство функций рисования, работы с контекстами устройств, метафайлами, координатами и шрифтами.
comdlg32.dll lz32.dll version32.dll	Эти библиотеки обеспечивают дополнительные возможности, в том числе поддержку стандартных диалоговых окон, сжатия файлов и контроля версий.
advapi32.dll	Библиотека содержит функции для работы с реестром Windows и файлами инициализации (INI).
wininet.dll	В данной библиотеке содержатся функции для использования Internet и FTP.
netapi32.dll	Сетевые функции Windows
crypt32.dll cryptdll.dll cryptnet.dll	В библиотеках содержатся криптографические функции и функции для работы с крипто-провайдерами.
ntdll.dll	Иногда в других библиотеках API функции всего лишь объявлены, а реализованы в данной библиотеке, но с другим именем.
comctl32.dll	В этой библиотеке реализован новый (усовершенствованный) набор управляющих элементов Windows, в числе которых - иерархические списки и улучшенные текстовые поля.
odbc32.dll	Это одна из dll, реализующих архитектуру ODBC (Open Database Connectivity). Функции предоставляют API для работы с базами данных.

За время существования 32-разрядной Windows функции API не изменились, меняется лишь их количество. Для удобства разработчиков Windows, а может это они позаботились о программистах, пишущих программы для Windows, все функции API разбиты на определенные категории и находятся в соответствующих библиотеках динамической компоновки (DLL). В таблице 3 представлены основные системные библиотеки API-функций и их описание.

4.2. Особенности приложений для ОС Windows на Ассемблере

Рассмотрим особенности написания приложения для ОС Windows на Ассемблере.

С появлением в языке ассемблера возможностей для создания приложений, работающих в операционной системе Windows, возрос интерес к нему. Конечно, доля программ, написанных на ассемблере невелика, но он

никогда не потеряет актуальность. Свое широкое применение он находит при написании драйверов различных устройств, трансляторов, антивирусов и программ для микроконтроллеров. Даже если использовать его не для написания всей программы, то хотя бы для тех её фрагментов, от которых требуется быстродействие, надежность и минимальный объем используемых ресурсов.

Программный модуль на ассемблере обладает, как правило, более высоким быстродействием, чем написанный на языке высокого уровня. Это связано с меньшим числом команд, требуемых для реализации фрагмента кода. Меньшее число команд быстрее выполняется центральным процессором, что, соответственно, повышает производительность программы.

Более легкое написание программ обусловливают следующие факторы:

1. наличие множества функций, облегчающих разработку приложений;
2. гибкая система адресации к памяти: возможность обращаться к памяти через любой регистр общего назначения;
3. отсутствие необходимости описания сегментных регистров;
4. доступность больших объёмов виртуальной памяти;
5. многообразие средств создания графического пользовательского интерфейса (различные виды окон, диалогов, меню и т.п.).

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.

Рассмотрим эти факторы подробно.

Согласование вызовов имеет особую значимость В ОС MS-DOS, работающей в реальном режиме, вызываемая процедура могла находиться либо в том же сегменте, что и команда вызова, тогда вызов назывался близким (NEAR) или внутрисегментным, либо в другом сегменте, тогда вызов назывался дальним (FAR) или межсегментным. Разница заключалась в том, что адрес в первом случае формировался из двух байтов, а во втором — из четырех байтов. Соответственно, возврат из процедуры мог быть либо близким (RETN), т. е. адрес возврата формировался на основе двух байтов, взятых из стека, либо дальним (RETF), и в этом случае адрес формировался на основе четырех байтов, взятых опять же из стека. Очевидно, что вызов и возврат должны быть согласованы друг с другом. В рамках единой программы это, как правило, не вызывало больших проблем. Но вот когда необходимо было подключить или

какую-то библиотеку, или объектный модуль, могли возникнуть трудности. Если в объектном модуле возврат осуществлялся по инструкции RETN, необходимо компоновать объектные модули так, чтобы сегмент, где находится процедура, был объединен с сегментом, откуда осуществляется вызов. Вызов в этом случае, разумеется, должен быть близким. Если же возврат из процедуры осуществлялся по команде RETF, то и вызов этой процедуры должен быть дальним. При этом вызов и сама процедура при компоновке должны были попасть в разные сегменты. Проблема согласования дальности вызовов усугублялась еще и тем, что ошибки обнаруживались не при компоновке, а при исполнении программы. С этим были связаны и так называемые модели памяти в языке С. В каталоге библиотек С для DOS) для каждой модели памяти существовала своя библиотека. Сегментация памяти приводила в С еще к одной проблеме - проблеме указателей.

Рассмотрим согласование имен. Согласование вызовов, как мы убедились, снято с повестки дня, а вот согласование имен год от года только усложнялось. Транслятор MASM, как известно, если принята модель stdcall (т. е. стандартный вызов), добавляет в конце имени @N, где N — количество передаваемых в стек параметров. То же по умолчанию делает и компилятор Visual C++. Другая проблема — символ подчеркивания перед именем. Транслятор MASM генерирует подчеркивание автоматически, если в начале программы устанавливается модель stdcall. Еще одна проблема — согласование заглавных и прописных букв. Транслятор MASM делает это автоматически. Как известно, и в стандарте языка С с самого начала предполагалось различие между заглавными и прописными буквами. В Паскале же прописные и заглавные буквы не различаются. В этом есть своя логика: Турбо Паскаль и Delphi не создают стандартных объектных модулей, зато могут подключать их. При создании же динамических библиотек туда помещается имя так, как оно указано в заголовке процедуры. Наконец, последняя проблема, связанная с согласованием имен, — это уточняющие имена в C++. Дело в том, что в C++ возможна так называемая перегрузка функций. Это значит, что одно и то же имя может относиться к разным функциям. В тексте программы эти функции отличаются по количеству и типу параметров и типу возвращаемого значения. Поэтому компилятор C++ автоматически делает в конце имени добавку — так, чтобы разные по смыслу функции различались при компоновке своими именами. Другими словами, имена в C++ искажаются. Разумеется, фирмы Borland и Microsoft и тут не пожелали согласовать свои позиции и делают в конце имени совершенно разные добавки. Обойти эту проблему не так сложно, нужно в программе на языке С для тех имен, к которым предполагается обращаться из других (внешних) модулей, использовать модификатор EXTERN "C".

Встроенный ассемблер – достаточно мощное средство. Надо только иметь в виду, что встроенные ассемблеры часто несколько отстают от обычных ассемблеров в поддержке новых команд микропроцессоров. Это вполне

объяснимо, т. к. разработка новой версии пакета, скажем Visual C++, требует гораздо больше времени, чем пакета MASM32.

Рассмотрим особенности работы с динамическими библиотеками. Обратим внимание, что никаких принципиальных отличий использования динамических библиотек, написанных на языках высокого уровня, от использования динамических библиотек, написанных на языке ассемблера, нет. И это тоже способ интеграции языка ассемблера с языками высокого уровня. В листинге содержится программа на языке ассемблера, которая загружает динамическую библиотеку и запускает процедуру setup.

Отметим еще раз, что обращение к сервису операционной системы в Windows осуществляется посредством вызова функций, а не прерываний. Здесь нет передачи параметров в регистрах при обращении к сервисным функциям и, соответственно, нет и множества результирующих значений, возвращаемых в регистрах общего назначения и регистре флагов. Следовательно, проще запомнить и использовать протоколы вызова функций системного сервиса.

Данная программа осуществляет вывод на экран окна с текстовым сообщением, содержащего кнопку OK.

Листинг 2.

```
.386
.model flat, stdcall
option casemap :none
include \MASM32\INCLUDE\windows.inc
include \MASM32\INCLUDE\masm32.inc
include \MASM32\INCLUDE\gdi32.inc
include \MASM32\INCLUDE\user32.inc
include \MASM32\INCLUDE\kernel32.inc
includelib \MASM32\LIB\masm32.lib
includelib \MASM32\LIB\gdi32.lib
includelib \MASM32\LIB\user32.lib
includelib \MASM32\LIB\kernel32.lib
.data
message db "Привет!",0
mestitle db "Мое первое приложение  ",0
.code
start:
invoke MessageBox,0,ADDR message,ADDR mestitle,MB_OK invoke ExitProcess,0
end start
```

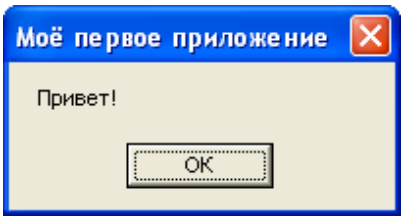

Рис.4 – выполнение программы

4.3. Средства программирования в Windows

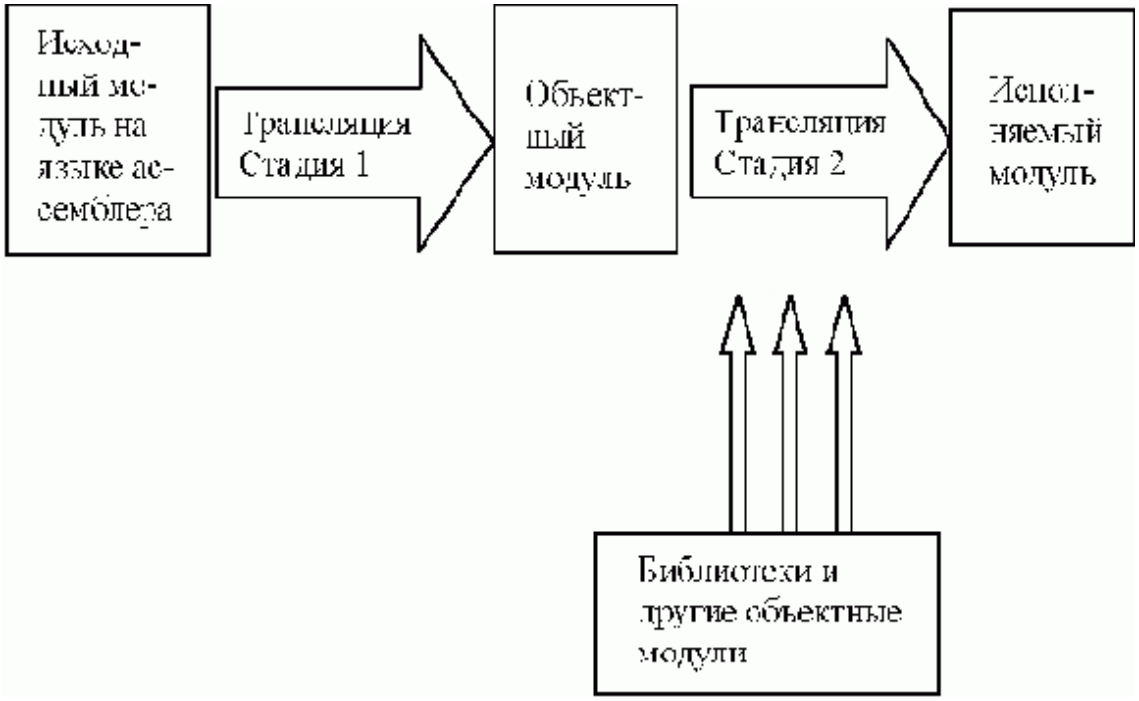

Рис.5 - Схема трансляции ассемблерного модуля.

Двум стадиям трансляции (Рис.5.) соответствуют две основные программы: ассемблер ML.EXE и редактор связей LINK.EXE (или TASM32.EXE и TLINK32.EXE в Турбо Ассемблере).

Пусть файл с текстом программы на языке ассемблера называется PROG.ASM, тогда, не вдаваясь в подробный анализ, две стадии трансляции будут выглядеть следующим образом: c:\masm32\bin\ml /c /coff PROG.ASM - в результате появляется модуль PROG.OBJ, а также c:\masm32\bin\link /SUBSYSTEM:WINDOWS PROG.OBJ - в результате появляется исполняемый модуль PROG.EXE. При этом, /c и /coff являются параметрами программы ML.EXE, а /SUBSYSTEM:WINDOWS является параметром для программы LINK.EXE.

Чтобы процесс трансляции сделать привычным, рассмотрим несколько простых, "ничего не делающих" программ. Первая из них представлена в листинге 3.

Листинг 3. "Ничего не делающая" программа

```
.586P ;плоская модель памяти
.MODEL FLAT, STDCALL ;
---------------------------------------------- ;
сегмент данных
_DATA SEGMENT
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
RET ;выход
_TEXT ENDS
END START
```

В листинге представлена "ничего не делающая" программа. Назовем ее PROG1.Чтобы получить загружаемый модуль, необходимы следующие команды:

```
ml /c /coff PROG1.ASM
link /SUBSYSTEM:WINDOWS PROG1.OBJ
```

Часто удобно разбить текст программы на несколько частей и объединить эти части еще на 1-й стадии трансляции. Это достигается посредством директивы INCLUDE. Например, один файл будет содержать код программы, а константы, данные (определение переменных) и прототипы внешних процедур помещаются в отдельные файлы. Обычно такие файлы записывают с расширением inc.

Именно такая разбивка демонстрируется в следующей программе (листинг 2). Если имя транслируемых модулей содержит пробелы, то название модулей придется заключать в кавычки, например, так: ML /c /coff "PROG 1.ASM".

Листинг 4. Пример использования директивы INCLUDE

```
;файл CONSTANTS.INC
CONS1 EQU 1000
CONS2 EQU 2000
CONS3 EQU 3000
CONS4 EQU 4000
CONS5 EQU 5000
CONS6 EQU 6000
CONS7 EQU 7000
CONS8 EQU 8000
CONS9 EQU 9000
CONS10 EQU 10000
CONS11 EQU 11000
CONS12 EQU 12000
;файл DATA.INC
DAT1 DWORD 0
```

```
DAT2 DWORD 0
DAT3 DWORD 0
DAT4 DWORD 0
DAT5 DWORD 0
DAT6 DWORD 0
DAT7 DWORD 0
DAT8 DWORD 0
DAT9 DWORD 0
DAT10 DWORD 0
DAT11 DWORD 0
DAT12 DWORD 0
;файл PROG1.ASM
.586P
;плоская модель памяти
.MODEL FLAT, STDCALL
;подключить файл констант
INCLUDE CONSTANTS.INC
;-------------------------------------------------------
;сегмент данных
_DATA SEGMENT
;подключить файл данных
INCLUDE DATA.INC
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
MOV EAX,CONS1
SHL EAX,1 ;умножение на 2
MOV DAT1,EAX
;---------------------------------
MOV EAX,CONS2
SHL EAX,2 ;умножение на 4
MOV DAT2,EAX
;---------------------------------
MOV EAX,CONS3
ADD EAX,1000 ;прибавим 1000
MOV DAT3,EAX
;---------------------------------
MOV EAX,CONS4
ADD EAX,2000 ;прибавим 2000
MOV DAT4,EAX
;---------------------------------
MOV EAX,CONS5
```

```
SUB EAX,3000 ;вычесть 3000
MOV DAT5,EAX
;---------------------------------
MOV EAX,CONS6
SUB EAX,4000 ;вычесть 4000
MOV DAT6,EAX
;---------------------------------
MOV EAX,CONS7
MOV EDX,3
IMUL EDX ;умножение на 3
MOV DAT7,EAX
;---------------------------------
MOV EAX,CONS8
MOV EDX,7 ;умножение на 7
IMUL EDX
MOV DAT8,EAX
;---------------------------------
MOV EAX,CONS9
MOV EBX,3 ;деление на 3
MOV EDX,0
IDIV EBX
MOV DAT9,EAX
;---------------------------------
MOV EAX,CONS10
MOV EBX,7 ;деление на 7
MOV EDX,0
IDIV EBX
MOV DAT10,EAX
;---------------------------------
MOV EAX,CONS11
SHR EAX,1 ;деление на 2
MOV DAT11,EAX
;---------------------------------
MOV EAX,CONS12
SHR EAX,2 ;деление на 4
MOV DAT12,EAX
;---------------------------------
RET ;выход
_TEXT ENDS
END START
```

Трансляция программы из листинга 4:

```
ml /c /coff prog1.asm
link /subsystem:windows prog1.obj
```

Программа из листинга 4 демонстрирует удобства использования директивы INCLUDE. В данном случае команда IDIV осуществляет операцию деления над операндом, находящемся в паре регистров EDX:EAX. Обнуляя EDX, мы указываем, что операнд целиком находится в регистре EAX.

Объектные модули

Перейдем к вопросу об объединении нескольких объектных модулей и подсоединении объектных библиотек на второй стадии трансляции. Прежде всего, отметим, что, сколько бы ни объединялось объектных модулей, один объектный модуль является главным. Смысл этого весьма прост: именно с этого модуля начинается исполнение программы.

Главный модуль всегда в начале сегмента кода будет содержать метку START, ее мы указываем после директивы END — транслятор должен знать точку входа программы, чтобы указать ее в заголовке загружаемого модуля

Обычно во второстепенные модули помещаются процедуры, которые будут вызываться из основного и других модулей. Рассмотрим пример такого модуля. Этот модуль вы можете видеть в листинге 1.3.

Листинг 5. Модуль PROG2.ASM, процедура которого PROC1 будет вызываться из основного модуля

```
.586P
;модуль PROG2.ASM
;плоская модель памяти
.MODEL FLAT, STDCALL
PUBLIC PROC1
_TEXT SEGMENT
PROC1 PROC
MOV EAX,1000
RET
PROC1 ENDP
_TEXT ENDS
END
```

Прежде всего, следует обратить внимание на то, что после директивы END не указана какая-либо метка. Ясно, что это не главный модуль, процедуры его будут вызываться из других модулей. Другими словами, все его точки входа вторичны и совпадают с адресами процедур, которые там расположены.

Второе, на что следует обратить внимание, — это то, что процедура, которая будет вызываться из другого объектного модуля, должна быть объявлена как PUBLIC. Тогда это имя будет сохранено в объектном модуле и далее может быть связано с вызовами из других модулей программой LINK.EXE.

Если просмотреть объектный модуль с помощью какой-нибудь простой viewer-программы, например той, что есть у программы Far.exe. Вместо имени PROC1 есть имя _PROC1@0. Во-первых, подчеркивание в начале имени отражает стандарт ANSI, предписывающий всем внешним именам (доступным нескольким модулям) автоматически добавлять символ подчеркивания.

Цифра после знака @ указывает количество байтов, которые необходимо передать в стек в виде параметров при вызове процедуры. В данном случае, что процедура параметров не требует. Сделано это для удобства использования директивы INVOKE.

Листинг 6. Модуль PROG1.ASM с вызовом процедуры из модуля PROG2.ASM

```
.586P
; плоская модель памяти
.MODEL FLAT, STDCALL
;-------------------------------------------------------
;прототип внешней процедуры
EXTERN PROC1@0:NEAR
;сегмент данных
_DATA SEGMENT
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
CALL PROC1@0
RET ;выход
_TEXT ENDS
END START
```

Процедура, которая расположена в другом модуле, но будет вызываться из данного модуля, объявляется как EXTERN. Далее, вместо имени PROC1 необходимо использовать имя PROC1@0.

Как было отмечено ранее, в операционной системе MS-DOS тип NEAR означал, что вызов процедуры (или безусловный переход) будет происходить в пределах одного сегмента. Тип FAR означал, что процедура (или переход) будет вызываться из другого сегмента. Подчеркнем, что в операционной системе Windows реализована так называемая плоская модель памяти, когда все адресное пространство процесса можно рассматривать как один большой сегмент. И здесь логично использование типа NEAR.

При объединении нескольких модулей первым должен указываться главный, а остальные — в произвольном порядке. Название исполняемого модуля тогда будет совпадать с именем главного модуля.

4.3.1.Директива INVOKE

Удобство ее заключается, во-первых, в том, что можно забыть о добавке @N. Во-вторых, эта команда сама заботится о помещении передаваемых параметров в стек. Последовательность команд

```
PUSH par1
PUSH par2
```

```
PUSH par3
PUSH par4
CALL NAME_PROC@N ; N – количество отправляемых в стек байтов заменяется на
INVOKE NAME_PROC, par4, par3, par2, par1
```

Причем параметрами могут являться регистр, непосредственно значение или адрес. Кроме того, для адреса может использоваться как оператор OFFSET, так и оператор ADDR.

Листинг 7. Пример использования директивы INVOKE

```
.586P
;плоская модель памяти
.MODEL FLAT, STDCALL
;-------------------------------------
;прототип внешней процедуры
PROC1 PROTO
;сегмент данных
_DATA SEGMENT
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
INVOKE PROC1
28 Часть I. Основы программирования в Windows
RET ;выход
_TEXT ENDS
END START
```

Отметим, что внешняя процедура объявляется теперь при помощи директивы PROTO. Данная директива позволяет при необходимости указывать и наличие параметров. Например, строка PROC1 PROTO :DWORD, :WORD будет означать, что процедура требует два параметра длиной в четыре и два байта (всего 6, т. е. @6).

На схеме (см. рис. 4.) указано, что существует возможность подсоединения не только объектных модулей, но и библиотек. Поэтому объектные модули объединяются в библиотеки. Для подсоединения библиотеки в MASM удобнее всего использовать директиву INCLUDELIB, которая сохраняется в объектном коде и используется программой LINK.EXE. Для создания библиотеки из объектных модулей у программы LINK.EXE имеется специальная опция /LIB, используемая для управления статическими библиотеками. Предположим, необходимо создать библиотеку LIB1.LIB, состоящую из одного модуля — PROG2.OBJ. для этого выполняется следующая команда:

```
LINK /lib /OUT:LIB1.LIB PROG2.OBJ
```

Если необходимо добавить в библиотеку еще один модуль (MODUL.OBJ), то достаточно выполнить команду:

LINK /LIB LIB1.LIB MODUL.OBJ

Рассмотрим два полезных примера использования библиотеки:
Вместо косой черты для выделения параметра программы LINK.EXE можно использовать черточку, например, так LINK -LIB LIB1.LIB MODUL.OBJ.

LINK /LIB /LIST LIB1.LIB — получить список модулей библиотеки;

LINK /LIB /REMOVE:MODUL.OBJ LIB1.LIB — удалить из библиотеки модуль
MODUL.OBJ.

Листинг 8. Пример использования библиотеки

```
.586P
;плоская модель памяти
.MODEL FLAT, STDCALL
;-------------------------------------------------------
;прототип внешней процедуры
EXTERN PROC1@0:NEAR
;-------------------------------------------------------
INCLUDELIB LIB1.LIB
;сегмент данных
_DATA SEGMENT
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
CALL PROC1@0
RET ;выход
_TEXT ENDS
END START
```

Данные в объектном модуле

Рассмотрим использование данных (переменных), определенных в другом объектном модуле. Термин «внешняя переменная» используется по аналогии с термином «внешняя Процедура».

Листинг 9. Модуль, содержащий переменную ALT, которая используется в другом модуле

```
.586P
;модуль PROG2.ASM
;плоская модель памяти
.MODEL FLAT, STDCALL
PUBLIC PROC1
PUBLIC ALT
;сегмент данных
_DATA SEGMENT
```

```
ALT DWORD 0
_DATA ENDS
_TEXT SEGMENT
PROC1 PROC
MOV EAX,ALT
ADD EAX,10
RET
PROC1 ENDP
_TEXT ENDS
END
```

Листинг 10. Модуль, использующий переменную ALT, определенную в другом
модуле (PROG2.ASM)

```
.586P
;модуль PROG1.ASM
;плоская модель памяти
.MODEL FLAT, STDCALL
;-------------------------------------
;прототип внешней процедуры
EXTERN PROC1@0:NEAR
;внешняя переменная
EXTERN ALT:DWORD
;сегмент данных
_DATA SEGMENT
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
MOV ALT,10
CALL PROC1@0
MOV EAX,ALT
RET ;выход
_TEXT ENDS
END START
```

Отметим, что в отличие от внешних процедур, внешняя переменная не требует добавки @N, поскольку размер переменной известен.

4.3.2. Упрощенный режим сегментации

Ассемблер MASM32 поддерживает так называемую упрощенную сегментацию. Упрощенная сегментация довольно удобна, особенно при программировании под Windows. Суть такой сегментации в следующем: начало сегмента определяется директивой .CODE, а сегмента данных — .DATA.

Причем обе директивы могут появляться в тексте программы несколько раз. Транслятор затем собирает код и данные вместе, как положено. Основной целью такого подхода, по-видимому, является возможность приблизить в тексте программы данные к тем строкам, где они используются. Такая возможность, как известно, в свое время была реализована в C++ (в классическом языке C это было невозможно).

Листинг 11. Пример программы, использующей упрощенную сегментацию

```
.586P
;плоская модель памяти
.MODEL FLAT, STDCALL
;-------------------------------------------------------
;сегмент данных
.DATA
SUM DWORD 0
;сегмент кода То есть переменная, определенная в другом модуле.
Есть директива и для стека — это .STACK.
.CODE
START:
;сегмент данных
.DATA
A DWORD 100
;сегмент кода
.CODE
MOV EAX,A
;сегмент данных
.DATA
B DWORD 200
;сегмент кода
.CODE
ADD EAX,B
MOV SUM,EAX
RET ;выход
END START
```

Трансляция программы из листинга 1.9:

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

Заметим, что макрокоманды .DATA и .CODE могут использоваться внутри кодового сегмента, определенного традиционным способом. Это удобно для создания разных полезных макроопределений

4.4. Разработка оконных и консольных приложений

Как отмечено выше, программирование в Windows основывается на использовании функций API (Application Program Interface, интерфейс программного приложения). И соответственно, программа в значительной степени будет состоять из таких вызовов. Все взаимодействие с внешними устройствами и ресурсами операционной системы будет происходить посредством таких функций. Список функций API и их описание лучше всего брать из файла WIN32.HLP.Подробнейшее описание по функциям API и вообще по программированию в Windows содержится в документации к Visual Studio .NET.Главным элементом программы в среде Windows является окно. Для каждого окна определяется своя процедура обработки сообщений (см. далее). Программирование в Windows в значительной степени заключается в программировании оконных процедур. Окно может содержать элементы управления: кнопки, списки, окна редактирования и др. Эти элементы, по сути, также являются окнами, но обладающими особыми свойствами, которые автоматически поддерживает операционная система. События, происходящие с этими элементами (и самим окном), способствуют приходу сообщений в процедуру окна.

Операционная система Windows использует линейную адресацию памяти.

Для программиста на языке ассемблера это означает, что адрес любой ячейки памяти будет определяться содержимым одного 32-битного регистра, например, EBX. Следствием предыдущего пункта является то, что программы фактически не ограничены в объеме данных, кода или стека, а вот сегменты в тексте программы играют роль определения секций исполняемого кода, которые обладают определенными свойствами: запрет на запись, общий доступ и т. д.

В рамках одной программы может быть осуществлена многозадачность.

4.4.1. Вызов функций API

Рассмотрим вызов функций на примере MessageBox().

Описание функции в нотации языка C:

int MessageBox(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType);

Данная функция выводит на экран окно с сообщением и кнопкой (или кнопками) выхода. Смысл параметров функции: hWnd — дескриптор окна, в котором будет появляться окно-сообщение, lpText — текст, который будет появляться в окне, lpCaption — текст в заголовке окна, uType — тип окна.

Также можно определить количество кнопок выхода.

Типы указанных параметров в действительности 32-битные целые числа:

HWND — 32-битное целое, LPCTSTR — 32-битный указатель на строку, UINT — 32-битное целое.

Отметим, что к имени функций иногда приходится добавлять суффикс "A". Это означает, что строковые параметры в данной функции должны иметь кодировку в стандарте ANSI. Добавление в конце функции суффикса "W" будет означать, что все строковые параметры будут иметь кодировку Unicode
Напомним, кроме этого, при использовании MASM необходимо также в конце имени добавить @16 (4 параметра по 4 байта).

Таким образом, вызов указанной функции будет выглядеть так: CALL MessageBoxA@16. Параметры следует предварительно поместить в стек командой PUSH.

Пусть дескриптор окна расположен по адресу HW, строки — по адресам STR1 и STR2, а тип окна-сообщения — это константа. Самый простой тип сообщения имеет значение 0 и называется MB_OK. Это сообщение предполагает окно с одной кнопкой. Имеем следующую схему
вызова:

```
MB_OK equ 0
STR1 DB "Неверный ввод! "
STR2 DB "Сообщение об ошибке. "
HW DWORD ?
PUSH MB_OK
PUSH OFFSET STR1
PUSH OFFSET STR2
PUSH HW
CALL MessageBoxA@16
```

Отметим, что в документации Microsoft утверждает, что при возвращении из функции API должны сохраниться регистры EBX, EBP, ESP, ESI, EDI. Значение функции, как обычно, возвращается в регистр EAX. Сохранность содержимого других регистров не гарантируется.

Аналогичным образом в ассемблере легко воспроизвести те или иные C-структуры. Рассмотрим, например, структуру, определяющую системное сообщение:

```
typedef struct tagMSG { // msg
HWND hwnd;
UINT message;
WPARAM wParam;
LPARAM lParam;
DWORD time;
POINT pt;
} MSG;
```

На MASM эта структура будет иметь вид:

```
MSGSTRUCT STRUC
```

```
MSHWND DD ?
MSMESSAGE DD ?
MSWPARAM DD ?
MSLPARAM DD ?
MSTIME DD ?
MSPT DD ?
MSGSTRUCT ENDS
```

При вызове функции API может возникнуть ошибка, связанная либо с неправильными входными параметрами, либо невозможностью получения искомых результатов из-за определенного состояния системы. В этом случае индикатором данного результата будет содержимое, возвращаемое в регистре EAX. Но, для разных функций это может быть разное значение. Это может быть 0 (чаще всего), –1 или какое-либо другое ненулевое значение. Напомним в этой связи, что 0 во многих языках программирования считается синонимом значения FALSE, а значению TRUE ставится в соответствие 1. В каждом конкретном случае следует свериться с документацией. С помощью функции GetLastError можно получить код последней происшедшей ошибки.

4.4.2. Структура программы

Обратимся к классической структуре программы.
В такой программе имеется главное окно, а следовательно и процедура главного окна. Далее, как отмечалось ранее, в коде программы можно выделить следующие секции:

1. регистрация класса окон;
2. создание главного окна;
3. цикл обработки очереди сообщений;
4. процедура главного окна.

Конечно, в программе могут быть и другие разделы, но данные разделы образуют основной каркас программы. Рассмотрим эти разделы по порядку.

4.4.3. Создание окна

Регистрация класса окон осуществляется с помощью функции RegisterClassA, единственным параметром которой является указатель на структуру WNDCLASS, содержащую информацию об окне.

На основе зарегистрированного класса с помощью функции CreateWindowExA (или CreateWindowA) можно создать экземпляр окна.

Цикл обработки очереди сообщений выглядит следующим образом :

```
while (GetMessage (&msg,NULL,0,0))
{
// Разрешить использование клавиатуры
// путем трансляции сообщений о виртуальных клавишах
```

```
// в сообщения об алфавитно-цифровых клавишах
TranslateMessage(&msg);
// Вернуть управление Windows и передать сообщение дальше
// процедуре окна
DispatchMessage(&msg);
}
```

Функция GetMessage() определяет очередное сообщение из ряда сообщений данного приложения и помещает его в структуру MSG. Если сообщений в очереди нет, то функция ждет появление сообщения. Вместо функции GetMessage часто используют функцию PeekMessage с тем же набором параметров. Отличие функции GetMessage от PeekMessage заключается в том, что последняя функция не ожидает сообщения, если его нет в очереди. Функцию

PeekMessage часто используют, чтобы оптимизировать работу программы.

Что касается функции TranslateMessage, то ее компетенция касается сообщений WM_KEYDOWN и WM_KEYUP, которые транслируются в WM_CHAR и WM_DEDCHAR, а также WM_SYSKEYDOWN и WM_SYSKEYUP, преобразующиеся в WM_SYSCHAR и WM_SYSDEADCHAR.

Смысл трансляции заключается не в замене, а в отправке дополнительных сообщений. Так, например, при нажатии и отпускании алфавитно-цифровой клавиши в окно сначала придет сообщение WM_KEYDOWN, затем WM_KEYUP, а затем уже WM_CHAR.

Как можно видеть, выход из цикла ожиданий имеет место только в том случае, если функция GetMessage возвращает 0. Это происходит только при получении сообщения о выходе (сообщение WM_QUIT). Таким образом, цикл ожидания играет двоякую роль: определенным образом преобразуются сообщения, предназначенные для какого-либо окна, и ожидается сообщение о выходе из программы.

Процедура главного окна

Вот прототип функции окна на языке C:

```
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
```

Оставив в стороне тип возвращаемого функцией значения6, обратите внима- ние на передаваемые параметры. Вот смысл этих параметров: hwnd — иден- тификатор окна, message — идентификатор сообщения, wParam и lParam — параметры, уточняющие смысл сообщения (для каждого сообщения они мо- гут играть разные роли или не играть никаких). Все четыре параметра, как вы, наверное, уже догадались, имеют тип DWORD.

А теперь рассмотрим каркас этой функции на языке ассемблера (листинг).

Листинг 12 . Каркас оконной процедуры

```
WNDPROC PROC
PUSH EBP
MOV EBP,ESP ; теперь EBP указывает на вершину стека
PUSH EBX
PUSH ESI
PUSH EDI
PUSH DWORD PTR [EBP+14H] ; LPARAM (lParam)
PUSH DWORD PTR [EBP+10H] ; WPARAM (wParam)
PUSH DWORD PTR [EBP+0CH] ; MES (message)
PUSH DWORD PTR [EBP+08H] ; HWND (hwnd)
CALL DefWindowProcA@16
POP EDI
POP ESI
POP EBX
POP EBP
RET 16
WNDPROC ENDP
```

Рассмотрим фрагмент из листинга 12 более подробно.

RET 16 — возврат из процедуры с освобождением стека от четырех параметров (16 = 4 . 4).

Доступ к параметрам осуществляется через регистр EBP:

```
DWORD PTR [EBP+14H] ; LPARAM (lParam)
DWORD PTR [EBP+10H] ; WPARAM (wParam)
DWORD PTR [EBP+0CH] ; MES (message) — код сообщения
DWORD PTR [EBP+08H] ; HWND (hwnd) — дескриптор окна
```

Так как функция DefWindowProc вызывается для тех сообщений, которые не обрабатываются в функции окна, в данном листинге, не обрабатываются все приходящие в функцию окна сообщения. Гарантирована сохранность четырех регистров: EBX, EBP, ESI, EDI. Этого требует документация, хотя обычно стараются, чтобы не изменялись все регистры.

Еще раз следует отметить, что в случае если сообщение обрабатывается процедурой окна, стандартным возвращаемым значением является значение 0 (FALSE). Однако бывают и специальные случаи, когда требуется возвращать 1 или –1.

В листинге 15 Приложениј 4 представлена простая программа.

Прежде всего, обратите внимание на директивы INCLUDELIB. В пакете MASM32 довольно много разных библиотек. В данном примере используются две: user32.lib и kernel32.lib. Вначале определяются константы и внешние библиотечные процедуры. В действительности все эти определения можно найти в include-файлах, прилагаемых к пакету MASM32. В примере не использованы стандартные include-файлы по двум причинам: во-первых, так удобнее понять технологию программирования, во-вторых, легче перейти от MASM к другим ассемблерам. Задача процедуры окна — правильная реакция

на все приходящие сообщения.. Отметим, что необработанные сообщения должны возвращаться в систему при помощи функции DefWindowProcA. Отслеживается пять сообщений:
WM_CREATE, WM_CLOSE, WM_DESTROY, WM_LBUTTONDOWN, WM_RBUTTONDOWN. Сообщения WM_CREATE и WM_DESTROY в терминах объектного программирования играют роли конструктора и деструктора: они приходят в функцию окна при создании окна и при уничтожении окна. Если щелкнуть по кнопке-крестику в правом углу окна, то в функцию окна придет сообщение WM_CLOSE, а после закрытия окна — сообщение WM_DESTROY.. Сообщение WM_CLOSE, таким образом, сообщает приложению, что пользователь намерен закрыть окно. Не всякое ли закрытие окна приводит к закрытию самого приложения. Это верно только при условии, что речь идет о главном окне приложения. И поэтому, несмотря на то, что сообщение WM_CLOSE обрабатывается в программе, далее запускается функцию DefWindowProc. Именно она обеспечивает закрытие окна, а затем приход на функцию окна сообщения WM_DESTROY. Чтобы проверить данное утверждение, достаточно проделать маленький эксперимент. Замените фрагмент, обработки сообщения WM_CLOSE на следующий:

```
WMCLOSE:
PUSH 0 ; MB_OK
PUSH OFFSET CAP
PUSH OFFSET MES3
PUSH DWORD PTR [EBP+08H] ; дескриптор окна
CALL MessageBoxA@16
MOV EAX,0
JMP FINISH
```

В этом случае, обойден вызов функции DefWindowProc. В результате перестанет закрываться окно и, естественно, само приложение: крестик в правом верхнем углу окна перестал работать.

После прихода в функцию окна сообщения WM_DESTROY (а оно приходит уже после того, как окно закрыто) будет выполнена функция PostQuitMessage, и приложению будет послано сообщение WM_QUIT, которое вызовет выход из цикла ожидания и выполнение функции ExitProcess, что в свою очередь приведет к удалению приложения из памяти. Заметим также, что переход на метку WMDESTROY при щелчке правой кнопкой мыши приводит к закрытию приложения. Разумеется, при выходе из приложения операционная система удаляет из памяти все, что было как-то связано с данным приложением, в том числе и главное окно.

Опишем метку _ERR — переход на нее происходит при возникновении ошибки, и здесь можно поместить соответствующее сообщение.

Если оконная процедура обрабатывает какое-либо сообщение, то она должна возвратить 0, то есть нулевое значение следует поместить в регистр EAX. В частности, если после обработки сообщения WM_CREATE возвратить

значение –1, то окно не будет создано, а создающая его функция (CreateWindowEx) возвратит 0.

Отметим, что есть еще одно сообщение, которое приходит перед сообщением WM_CREATE — это сообщение WM_NCCREATE. После его обработки следует возвращать 1. Если возвратить 0, то окно не будет создано, а функция CreateWindowEx возвратит 0.

5. Использование компиляторов TASM и MASM

Рассмотрим особенности программирования на Ассемблере с учетом особенности различных компиляторов, в частности наиболее распространенных - TASM и MASM. Как было описано ранее общая структура программы следующая:

```
.386p           ;тип процессора
.model flat, stdcall  ; модель памяти и вызова подпрограмм
;объявление включаемых (заголовочных) файлов, макросов, макроопределений,
; также внешних определений
.data; Инициализированные данные
.data?; неинициализированные данные
.const; константы
.code; исполняемый код
End "метка точки входа"
```

Процессоры могут быть 386, 486, 586, обычно всегда стоит 386p, но ничто вам не мешает поставить 486p или 586. Модель памяти всегда flat и никакой другой не может быть. Вызов подпрограмм обычно всегда stdcall, стандарт вызова почти всех API функций. Секция .data - секция с инициализированными данными она включается в исполняемый файл. Секция .data? - секция с неинициализированными данными, она не включается в исполняемый файл и появляется только тогда, когда программа загружается в память. Секция.const -секция констант. Секция code содержит исполняемый код программы. В конце программы всегда должно стоять слово end, которая задаёт точку входа программы, т.е. место с которого начнётся выполняться программа. Секции .data, .data? имеют полный доступ. Разумеется, секции .const и .code имеют атрибут доступа - только чтение. Секция .const наиболее редко встречается в программах, так как константы можно задавать с помощью макроопределений.

Каждая программа в Win32 в конце своего выполнения должна вызвать функцию ExitProcess, по причинам описанным в предыдущей главе. Определение ExitProcess из MS SDK:

```
VOID ExitProcess(
   UINT uExitCode        // exit code for all threads
  );
```

uExitCode - это значение никогда не используется и всегда равно 0.

Рассмотрим программу, которая выводит легендарное сообщение "Hello, World!".

Листинг 13.

```
.386
.model  flat
extrn ExitProcess:PROC
extrn MessageBoxA:PROC
data
Ttl     db "First ASSEMBLER program",0h
Msg   db 'Hello, World!!!!',0h
.code
start:
        push 0h
        push offset Msg
        push offset Ttl
        push 0h
        call MessageBoxA
        push 0h
        call ExitProcess
end     start
```

Программа просто вызывает функцию MessageBoxA, для вывода окна сообщения потом осуществляется выход из процесса. Повторим пределение функции MessageBox:

```
int MessageBox(
   HWND hWnd,   // handle of owner window
   LPCTSTR lpText,       // address of text in message box
   LPCTSTR lpCaption,  // address of title of message box
   UINT uType      // style of message box
  );
```

Мы указали только модель памяти и модель вызов подпрограмм не указали, так как это просто не к чему. В качестве первого параметра мы передаём ноль, поэтому у окна сообщения не будет родительского окна. В качестве параметров lpText и lpCaption мы передаём адреса соответствующих строк, заметьте, что мы передаём адрес строки (т.е. её первого символа) у которой в конце стоит символ #0, это нужно, для того чтобы функция смогла найти конец строки. Последний параметр определяет стиль окна сообщения, в данном случае мы передали 0, что обозначает, что там будет только одна кнопка OK, иначе этот стиль обозначается MB_OK. Функцию ExitProcess вы уже знаете. После директивы end мы указали метку, с которой начнётся выполнение ваша программа.

В Win32 есть 2 типа функций заканчивающиеся на A и на W. Всё отличие в строках, которые принимают функции. A-функции принимают ANSI строки. W-функции принимают Unicode строки. У каждой функции, которая принимает

строки, есть обе версии. Обычно все пользуются ANSI версиями функций. Но в любом случае при вызове ANSI функции все строки сначала преобразуются в Unicode строки, и будет вызвана Unicode функция.

5.1. Компиляция с использованием TASM

Для компиляции надо написать текст программы на ассемблере и сохранить в файл с расширением *.asm. Потом надо ассемблировать. Рассмотрим процесс ассемблирования с программой tasm32.exe, которая находится в папке %tasmdir%\bin\tasm32.exe, формат её вызова такой: TASM [options] source [,object] [,listing] [,xref] (доп сведения смотрите в tasm32 /?) для большинства программ ассемблирование проходит так:

%tasmdir%\BIN\tasm32 /m3 /ml asmfile,

Где asmfile - имя файла без расширения.

Потом будет создан объектный файл, файл листинга и т.д. Теперь надо всё это слинковать. Для Линковки понадобится программа tlink32.exe. Вот формат её вызова:

TLINK objfiles, exefile, mapfile, libfiles, deffile, resfiles

%tasmdir%\BIN\tlink32 /Tpe /aa asmfile,asmfile,,%tasmdir% \Lib\import32.lib

Где asmfile - имя вашего файла БЕЗ расширения.

Статическая библиотека %tasmdir% \Lib\import32.lib содержит в себе все функции библиотек kernel32.dll, user32.dll, gdi32.dll и может быть других стандартных библиотек. Желательно чтобы asmfile находился в папке %tasmdir%.

Для быстрой компиляции можно создать *.bat файл, который автоматизирует процесс. Текст файла Compile.bat представлен в приложении 3.

5.2. Компиляция с использованием MASM

Популярный компилятор MASM создан специально для написания программ на ассемблере для Win32. В нём есть макросы и специальные директивы для упрощения программирования.

Функции.

Основное преимущество MASM это макрос invoke, позволяющий вызывать API функции по-обычному с проверкой количества и типа параметров. Это почти тот же call, как в TASM, но этот макрос проверяет количество параметров и их типы. Вот так вызывается функция:

Invoke <функция>, <параметр1>, <параметр2>, <\параметр3>

Чтобы использовать invoke для вызова процедуры, необходимо определить ее прототип:

```
PROTO STDCALL testproc:DWORD, :DWORD, :DWORD
```

Эта директива объявляет процедуру, названную testproc, которая берет 3 параметра размером DWORD. Masm также имеет контроль соответствия типов, т.е. проверяет, имеют ли параметры правильный тип (размер). В invoke можно использовать ADDR вместо OFFSET. Это сделает адрес в правильной форме, когда код будет собран.

```
testproc PROTO STDCALL :DWORD, :DWORD, :DWORD
.code
testproc proc param1:DWORD, param2:DWORD, param3:DWORD
....
ret
testproc endp
```

Данный фрагмент создает процедуру, названную testproc, с тремя параметрами. Прототип используется invoke. Все параметры можно использовать в коде процедуры, они автоматически извлекутся из стека. Также в процедурах можно использовать локальные переменные.

```
testproc proc param1:DWORD, param2:DWORD, param3:DWORD
LOCAL var1:DWORD
LOCAL var2:BYTE
mov ecx, param1
mov var2, cl
mov edx, param2
mov eax, param3
mov var1, eax
add edx, eax
mul eax, ecx
mov ebx, var1
.IF bl==var2
xor eax, eax
.ENDIF
ret
testproc endp
```

Однако, нельзя использовать эти переменные вне процедуры. Они сохранены в стеке и удаляются при возврате из процедуры.

Конструкции сравнения и повтора.

If - Она имеет тот же самый синтаксис, что и в TASM.

Repeat - Эта конструкция выполняет блок, пока условие не истинно:

```
.REPEAT
; код здесь
```

```
.UNTIL eax==1
```

Эта конструкция повторяет код между repeat и until, пока eax не станет равным 1.

While - Конструкция while это инверсия конструкции repeat. Она выполняет блок, пока условие истинно:

```
.WHILE eax==1
; код здесь
.ENDW
```

Можно использовать директиву .BREAK, чтобы прервать цикл и выйти.

```
.WHILE edx==1
inc eax
.IF eax==7
.BREAK
.ENDIF
.ENDW
```

Если eax=7, цикл while будет прерван.

Директива continue осуществляет переход на код проверяющий условие цикла в конструкциях repeat и while.

Теперь программа листинга видоизменяется следующим образом:

Листинг 14

```
   .486
   .model flat, stdcall
   option casemap :none
includelib .\masm32\lib\kernel32.lib
includelib .\masm32\lib\user32.lib
MessageBoxA PROTO STDCALL :DWORD, :DWORD, :DWORD, :DWORD
ExitProcess PROTO STDCALL :DWORD

.data
        ttl db '11111',0
.code
start:
        invoke MessageBoxA,0,offset ttl,offset ttl,0
        invoke ExitProcess,0
end start
```

При компиляции в TASM пути к статическим библиотекам указывались при компиляции, в MASM пути к статическим библиотекам указываются в тексте программы точно так же как и пути к включаемым файлам с помощью директивы includelib.

Прототипы каждой из библиотек есть в одноимённых включаемых файлах в папке include.

```
        includelib .\masm32\lib\kernel32.lib
        includelib .\masm32\lib\user32.lib
```

include .\masm32\include\kernel32.inc
include .\masm32\include\user32.inc

В этих включаемых файлах определены функции без букв A или W в конце. Директива option нужна для задания некоторых настроек компиляции. Опция casemap задаёт чувствительность к регистру символов. В коде указано none, тем самым установлена чувствительность к регистру символов. Это надо, для того чтобы избежать конфликтов включаемых файлов от разных авторов.);

В представленных ниже таблицах описаны опции компиляции и линкования

Компиляция:

ML [/опции] filelist [/link linkoptions]

Таблица 4

Опции компиляции

/c	ассемблирование без линковки В основном используется эта опция, так как используется внешний линкер (например link.exe), для компоновки файлов.
/coff	генерировать объектный файл в COFF формате генерируется формат файла для компоновщика microsoft.
/Fo<file>	имя объектного файла может использоваться, если выходной файл был с другим именем, не таким как исходный файл.
/G<c|d|z>	Использует вызовы Pascal, C, или Stdcall выберается тип вызовов для процедур.
/Zi	Добавить символьную отладочную информацию Установается эту опция, если используется отладчик.
/I<name>	Установить include путь Определяет include путь

Линкование:

LINK [опции] [файлы] [@commandfile]

Таблица 5

Опции линкования

/DEBUG	Отладка Это создаст информацию для отладки. Используется эта опция, когда используется отладчик.
/DEBUGTYPE:CV|COFF	Тип отладки: codeview / coff Выбирает выходной формат отладочной информации. Это зависит от отладчика. Softice и visual c++ отладчики оба могут обрабатывать CV(codeview)
/DEF:имя_файла	DEF файл Указывает файл определения (.def). Используется с dll, для экспортируемых функций.
/DLL DLL	Выходной файл DLL, а не EXE.
/LIBPATH:path	Указывает путь к файлам библиотек (*.lib).
/I<имя>	Устанавливает путь для inc-файлов Указывает путь для inc-файлов, по умолчанию.
/OUT:имя_файла	Out:имя_файла Может изменить имя выходного файла.
/SUBSYSTEM:{...}	Подсистема Выбирает ОС на которой должна выполнятся программа: NATIVE|WINDOWS|CONSOLE|WINDOWSCE|POSIX

5.3. О пакете MASM32

Последние версии данного пакета можно свободно скачать с сайта http://www.movsd.com. Это сайт Стива Хатчессона, создателя пакета MASM32. Данный пакет специально предназначен для создания исполняемого кода для операционной системы Windows и базируется на созданных и поддерживаемых фирмой Microsoft продуктах, таких как транслятор языка ассемблера ML.EXE или редактор связи LINK.EXE.

Пакет MASM32 является свободно распространяемым продуктом, и можно законно использовать его для создания своих программ. Особенностью пакета MASM32 является то, что он ориентирован на создание программ, состоящих из макроопределений и библиотечных процедур. В таком виде программа будет напоминать программу на языке высокого уровня. Такая

программа неизбежно также будет содержать и недостатки программ на языках высокого уровня — наличие избыточного кода.

Это объясняется очень просто: при создании библиотечных процедур разработчик неизбежно должен добавлять избыточный код, для проведения дополнительных проверок, чтобы создаваемый модуль можно было применять для широкого спектра задач. Избегая использования библиотечных процедур и макросов, можно получить наиболее компактный и производительный код.

Обзор пакета MASM32

Сам пакет распространяется в виде одного исполняемого модуля, в котором в сжатом виде содержатся все компоненты пакета. При инсталляции можно выбрать один из локальных дисков, где будет располагаться пакет. В корневом каталоге будет создана папка MASM32, куда и будут помещены все компоненты пакета.

Весь пакет MASM32 состоит из следующих частей (блоков).

Инструментальный блок. Основной инструментарий можно найти в подкаталоге \bin. Здесь располагаются программы, которые используются для трансляции программ. Это в первую очередь ассемблер ML.EXE, компоновщик LINK.EXE, а также транслятор ресурсов RC.EXE.

Из других инструментальных программ следует выделить программу QEDITOR.EXE. Это редактор, который можно с успехом использовать при написании ассемблерных программ. Хочется заметить, что для многих утилит пакета здесь имеются ассемблерные тексты, с помощью которых можно менять функциональность программ.

Информационный блок. Следует иметь в виду, что пакет MASM32 не содержит сколько-нибудь полного руководства, которое можно было бы порекомендовать программистам. В подкаталоег \help располагаются основные файлы этого блока. Они имеют устаревший в настоящее время для файлов помощи формат Winhelp9 и содержат справочную информацию. Отдельные файлы, содержащие много полезной информации, можно найти в самых разных каталогах папки

MASM32. Следует также обратить внимание на подкаталоги \icztutes и \tutorial, где можно обнаружить отдельные руководства по многим вопросам программирования на MASM32.

Блок примеров. Пакет содержит огромное количество примеров по самым разным вопросам программирования на MASM под Windows. Они располагаются в основном в подкаталогах: \examples, \com, \oop. Многие каталоги с примерами содержат пакетный файл, с помощью которого можно перетранслировать исправленные тексты.

Блок описаний API-функций. Блок представляет собой большое количество файлов с расширением inc, расположенных в подкаталоге \include. В файлах содержатся прототипы большинства функций API операционной системы Windows, а также различных полезных констант.

Блок библиотек. Располагаются библиотеки в каталоге \lib и имеют расширение lib. Большинство этих библиотек не содержат в себе кода, а являются лишь шлюзами для вызова функций API, код которых располагается в динамических библиотеках, а те, в свою очередь, размещены в подкаталоге \SYSTEM32 каталога Windows. Исключение им составляет, в частности, библиотека MASM32.LIB, которая содержит большое количество процедур, написанных авторами пакета, которые могут значительно облегчить программирование на ассемблере. Тексты этих процедур, с которыми весьма полезно познакомиться, находятся в подкаталоге \M32LIB. Описание библиотечных процедур MASM32.LIB можно найти в подкаталоге \help — файл MASMLIB.HELP. Обратим также внимание на подкаталог \fpulib, где можно найти тексты библиотеки fpu.lib, в которой располагаются процедуры для манипулирования числами с плавающей точкой.

Для использования данного формата помощи в системе должна присутствовать системная утилита winhelp32.exe, которая уже поставляется не со всеми с операционными системами.

Библиотека макросов. Библиотека макросов находится в подкаталоге \macros. Тексты макросов располагаются в файле macros.asm. При необходимости использовать макрос этот файл следует подключить к программе при помощи директивы INCLUDE.

Трансляторы

С программами ML.EXE и LINK.EXE мы уже познакомились.

Программы располагаются в подкаталоге \bin. Здесь же в подкаталоге располагается программа RC.EXE — транслятор ресурсов. Эта программа будет необходима, если в проекте используются ресурсы. Следует обратить внимание, что для транслятора ресурсов имеется файл помощи — RC.HLP, где описываются не только возможности программы RC.EXE, но и довольно подробно говорится о структуре самого файла ресурсов. К слову сказать, в каталоге \bin имеется программа IMAGEDIT.EXE, с помощью которой можно создавать графические компоненты ресурсов. Файл же ресурсов, который затем должен быть откомпилирован программой RC.EXE, можно создать в обычном текстовом редакторе или при помощи специализированного редактора ресурсов, который можно найти, например, в пакете Visual Studio .NET.

В пакет MASM32 были включены также еще три программы, которые также можно использовать при трансляции: POLINK.EXE — редактор связей, POLIB.EXE — программа-библиотекарь, PORC.EXE — транслятор ресурсов.

Программы располагаются все в том же подкаталоге \bin, их предоставил создателям пакета MASM автор Пэл Ориниус (префикс PO). Можно использовать эти программы вместо традиционных программ LINK.EXE и RC.EXE от Microsoft.

Редактор QEDITOR

Редактор QEDITOR.EXE, который поставляется вместе с пакетом MASM32, располагается непосредственно в корневом каталоге пакета. Сам

редактор и все сопутствующие ему утилиты написаны на ассемблере. Анализ их размера и возможностей действительно впечатляет. Например, сам редактор имеет длину всего 37 Кбайт. Редактор вполне годится для работы с небольшими одномодульными приложениями. Для работы с несколькими модулями он не очень удобен. Работа редактора основана на взаимодействии с различными утилитами посредством пакетных файлов. Например, трансляцию программ осуществляет пакетный файл ASSMBL.BAT, который использует ассемблер ML.EXE, а результат ассемблирования направляется в текстовый файлASMBL.TXT. Далее для просмотра этого файла используется простая утилита THEGUN.EXE (кстати, размером всего 6 Кбайт). Аналогично осуществляется редактирование связей.

Для дизассемблирования исполняемого модуля служит утилита DUMPPE.EXE, результат работы этой утилиты помещается в текстовый файл DISASM.TXT. Аналогично осуществляются и другие операции.

Можно настроить эти операции, отредактировав соответствующий пакетный файл с модификацией (при необходимости) используемых утилит (заменив, например, ML.EXE на TASM32.EXE и т. п.).

Редактор обладает одной важной и полезной особенностью: при помощи пункта Edit | Edit Menus можно удалять или добавлять пункты меню. Синтаксис описания меню довольно прост.

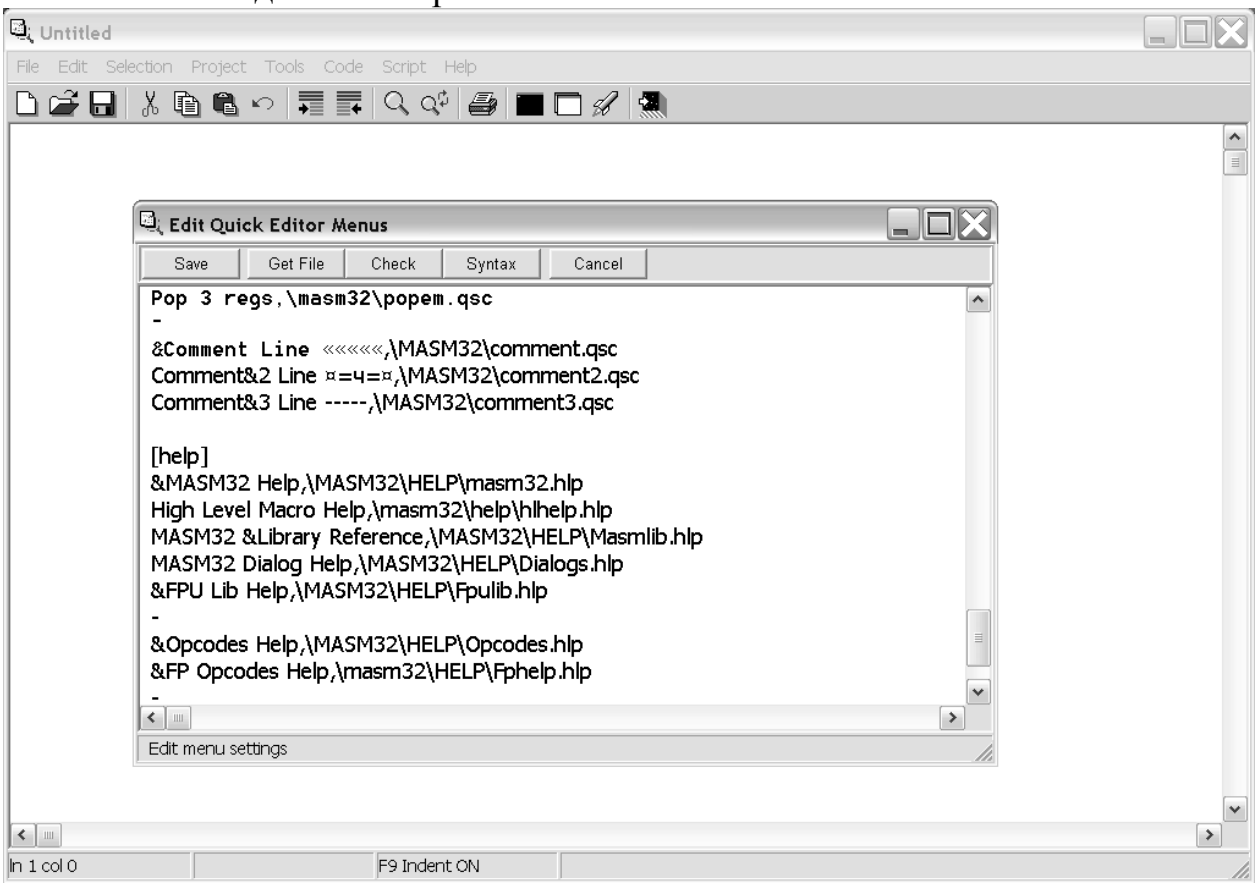

Рис.6 . Окно редактора QEDITOR.EXE вместе с окном редактирования меню

Следует обратить также внимание на пункты меню Code и Script. Первый пункт меню содержит, в частности, команды генерации шаблонов простых приложений. Можно сгенерировать такой шаблон, а затем по своему усмотрению наращивать его функциональность. В пункте Script находятся команды генерации некоторых полезных программных блоков, например блок SWITCH. Сценарии генерации содержатся здесь же в каталоге и имеют простой текстовый формат и расширение qsc.

Приложение 1

Компилятор - это транслятор, который переводит программу из нотации одного языка в нотацию машинного языка. Машинным языком может быть код конкретной машины, любой объектный код.
Существуют различные компиляторы ассемблера: MASM, FASM, TASM, NASM, YASM, GAS.

Macro Assembler (MASM) — ассемблер для процессоров семейства x86. Был произведён компанией Microsoft для написания программ в операционной системе MS-DOS . Это поддерживало широкое разнообразие макросредств и структурированность программных идиом, включая конструкции высокого уровня для повторов, вызовов процедур и чередований (поэтому MASM — ассемблер высокого уровня). Позднее была добавлена возможность написания рограмм для Windows.

Turbo Assembler (TASM) — программный пакет компании Borland, предназначенный для разработки программ на языке ассемблера для архитектуры x86. TASM может работать совместно с трансляторами с языков высокого уровня фирмы Borland, такими как Turbo C и Turbo Pascal.

Flat Assembler (FASM) — свободно распространяемый многопроходной ассемблер. Обладает небольшими размерами, имеет богатый и ёмкий макро-синтаксис, позволяющий автоматизировать множество рутинных задач. Поддерживаются как объектные форматы, так и форматы исполняемых файлов. Это позволяет в большинстве случаев обойтись без компоновщика. Компиляция программы в fasm состоит из 2 стадий: препроцессирование(раскрываются все макросы, символические константы) и ассемблирование(определяются адреса меток, обрабатываются условные ирективы). Используется Intel-синтаксис записи инструкций.

Netwide Assembler (NASM) — свободный (LGPL и лицензия BSD) ассемблер для архитектуры Intel x86. Используется для написания 16-, 32- и 64-разрядных программ. В NASM используется Intel-синтаксис записи инструкций. Особенности синтаксиса: приемник находится слева от источника, название регистров зарезервировано т.е. нельзя использовать метки с именами

eax, ebx и т.д.(пример mov ax,bx). Для ассемблирования файла нужно ввести следующую команду:

nasm –f format filename –o output

YASM (Your fAvorite aSseMbler) — ассемблер, являющийся попыткой полностью переписать ассемблер NASM. Лицензируется под лицензией BSD.

GNU Assembler или GAS (вызывается командой as) — ассемблер проекта GNU; используется компилятором GCC. Входит в пакет GNU Binutils. Кроссплатформенная программа, запускается и компилирует код для многочисленных процессорных архитектур. Распространяется на условиях свободной лицензии GPL 3. По умолчанию, GAS использует AT&T-синтаксис для x86 и x86-64, то есть регистры обозначаются префиксом % и регистр-приёмник указывается после источника; Директивы начинаются с точки. Многострочные комментарии обозначаются как в Си (/* … */). Однострочные комментарии на разных платформах обозначаются по-разному; например, в GAS для x86 и x86-64 они обозначаются как в C++ (// …) или как в sh (# …), в то время как в GAS для ARM они обозначаются символом @.

Плюсы и минусы компиляторов:

+/- MASM

Достоинства:

- Поддерживает архитектуру AMD x86-64.
- Поддерживает все расширения SSE/SSEII/SEEIII.
- Множество книг, упрощающих процесс обучения.
- Является выходным “языком” для многих дизассемблеров.
- Включен в состав DDK.

Недостатки:

- Много ошибок (см. примеры).
- Отсутствие поддержки некоторых инструкций и режимов адресации процессора (невозможно сделать jmp far seg:offset).
- Большая сложность создания смешанного 16/32 битного кода.

MASM генерирует отладочную информацию в формате CodeView, которую Microsoft Linker может преобразовывать в PDB-формат, хоть и не документированный, но поддерживаемый библиотекой dbghelp.dll, поэтому файлы, оттранслированные MASM'ом, можно отлаживать в Soft-Ice, дизассемблировать в IDA Pro и прочих продуктах подобного типа.

Некоторые примеры ошибок: dwtoa и atodw_ex не понимают знака и по скорости достаточно медленные, хотя в документации написано: "A high speed ascii decimal string to DWORD conversion for applications that require high speed streaming of conversion data"; ucFind не находит в строке подстроку, если длина подстроки равна 1 символу; функции BMHBinsearch и SBMBinSearch (поиск алгоритмом Бойера-Мура) реализованы с ошибками; некоторые функции обрушивают программу (если передать ustr2dw строку длиннее пяти байт -

программа падает); отсутствие поддержки некоторых инструкций и режимов адресации процессора, например, невозможно сделать jmp far seg:offset..
Отлично подходит под Windows9x.
+/- TASM

Достоинства:

- Имеет свою среду разработки – Tasm visual.
- Совместим с MASM, то есть TASM умеет транслировать исходники, разработанные под MASM.
- Поддерживает свой собственный режим IDEAL с большим количеством улучшений и расширений.
- Высокая производительность на DOS.

Недостатки:

- Не совместим с SDK/DDK.
- Формат отладочной информации несовместим с CodeView (поддерживается TurboDebugge).
- Давно не поддерживается.

Самый подходящий компилятор, если вы работаете с DOS

+/- FASM

Достоинства:

- Высокая скорость компиляции.
- Упрощенный синтаксис (отсутствие offset'ов и директив).
- Полная поддержка всех процессорных команд (в том числе и jmp 0007:00000000).
- Гибкая система управления форматов выходных файлов.
- Позволяет обходиться без линкера (генерирует не только Microsoft coff, но и готовые bin, mz, pe и elf файлы).
- Кроссплатформенный.

Недостатки:

- Плохо структурированная документация.
- Несовместимость с MASM, затрудняет разработку Windows-драйверов.
- Не поддерживает генерацию отладочной информации.
- Написан сам на себе, т.е. не стабильный.

FASM позволяет обходиться без линкера, однако, при этом раскладку секций в PE-файле и таблицу импорта приходится создавать "вручную" с помощью специальных директив ассемблера, так что на практике все же намного удобнее сгенерировать coff и скомпоновать его с модулями, написанными на языках высокого уровня.

+/- NASM

Достоинства:

- Поддержка всей линейки x86 процессоров.
- Богатство выходных файлов (bin, aout, aoutb, coff, elf, as86, obj, win32, rdf, ieee).
- Поддерживает порты под MS-DOS, Windows, Linux и BSD.

• Генерация отладочной информации в форматах Borland, STABS и DWARF2.
• Количество ошибок в трансляторе довольно невелико, в отличии от продуктов MASM/TASM.
• Open source.

Недостатки:

• Не помнит типов объявляемых переменных.
• Непосредственная трансляция примеров из SDK/DDK под NASM'ом невозможна.
• Не имеет нормальной поддержки структур.
• Отсутствие контроля за соответствием транслируемых инструкций типу указанного процессора (команда "cpuid" под ".486" ассемблируется вполне нормально, а ведь не должна).

Некоторые примеры: команда "mov eax, 1" не оптимизируется и транслятор умышленно оставляет место для 32-разрядного операнда. Если мы хотим получить "короткий" вариант, размер операнда необходимо специфицировать явно: "mov eax, byte 1" или использовать опцию "-On" для автоматической оптимизации.

Из небольших недочетов можно называть невозможность автоматического генерации короткого варианта инструкции "push imm8".

Отлично подходит под LINUX/BSD.

Когда развитие NASM'а затормозилось, его исходные тексты легли в основу нового транслятора – YASM

+/- YASM

Достоинства:

• Поддержка синтаксиса как NASM так и GAS(AT&T-синтаксис).
• Yasm построен «модульно», что позволяет легко добавлять новые формы синтаксиса, препроцессоры и т. п.
• Поддерживает двоичный формат выходных файлов, особенно удобных для создания shell-кода.

Недостатки:

• Неполная документация.

Исходя из выше представленного, можно сделать вывод - выбор компилятора зависит от среды разработки.

Справка:

• SoftICE — отладчик режима ядра для Microsoft Windows. Программа разработана для управления процессами на низком уровне Windows, причём таким образом, чтобы операционная система не распознавала работу отладчика. В отличие от прикладного отладчика, SoftICE способен приостановить все операции в Windows, что очень важно для отладки драйверов.
• Режим Turbo Assembler Ideal вводит новый синтаксис операндов выражений и команд. Этот не является радикально отличным от существующего синтаксиса MASM, однако в режиме Ideal операции и

ключевые слова MASM реализованы более просто и ясно, а используемые им формы более информативны, как для вас, так и для Turbo Assembler.

- CodeView - автономный отладчик. Это был один из первых отладчиков для MS-DOS, являвшийся полноэкранным, а не линейным (как его предшественники DEBUG.COM и symdeb). Во время работы с CodeView пользователи могут использовать окна, которыми можно манипулировать. Некоторые из окон: окно кода, окно данных, командное окно. Сегодня, отладчик считается комплексной и неотъемлемой частью Microsoft Visual Studio.
- GNU Compiler Collection (обычно используется сокращение GCC) — набор компиляторов для различных языков программирования, разработанный в рамках проекта GNU. GCC является свободным программным обеспечением, распространяется фондом свободного программного обеспечения (FSF) на условиях GNU GPL и GNU LGPL и является ключевым компонентом GNU toolchain. Он используется как стандартный компилятор для свободных UNIX-подобных операционных систем.
- Установить источник проблем помогает отладочная информация, которую компилятор может добавить в каждый объектный файл. Она включает в себя указание на типы данных, использованные в программе, и соответствие объектного кода строкам текста исходного файла.
- Отличия AT&T-ассемблера (gas) от Intel-ассемблера (MASM, TASM, FASM, NASM):

1.Комментарий начинается с символа "#", а не ";"; в свою очередь символ ";" разделяет команды и позволяет записывать несколько команд в одной строке.
2.Отсутствие префикса операнда указывает на адрес в памяти; поэтому movl $foo,%eax помещает адрес переменной foo в регистр %eax, а movl foo,%eax помещает в %eax содержимое переменной foo.
3.Имена регистров начинаются с символа %, то есть %eax, %dl, вместо eax, dl, и т. д. Это позволяет включать в код внешние переменные C, не опасаясь ошибок и не используя префиксов с подчёркиванием (_). Например: %eax, %ebx, %ecx, %edx

- SDK (Software Development Kit) — комплект средств разработки, который позволяет специалистам по программному обеспечению создавать приложения для определённого пакета программ, программного обеспечения базовых средств разработки, аппаратной платформы, компьютерной системы, игровых консолей, операционных систем и прочих платформ.
- DDK (Driver Development Kit) — набор из средств разработки, заголовочных файлов, библиотек, утилит, программного кода примеров и документации, который позволяет программистам создавать драйверы для устройств по определённой технологии или для определённой платформы (программной или программно-аппаратной).

Создание драйвера возможно и без использования DDK, однако DDK содержит средства, упрощающие разработку драйвера (например, готовые

примеры и шаблоны кода), обеспечивающие совместимость драйвера с операционной системой (символические определения констант, определения интерфейсных функций ОС, определения, зависящие от типа и версии ОС), а также установку и тестирование драйвера.

• Многопроходные компиляторы - получают объектный модуль не за один раз, а за несколько проходов. Результатом промежуточных проходов является некоторое внутреннее представление программы. Результатом последнего прохода является объектный модуль.

Количество проходов является важной технической характеристикой компилятора. Разработчики пытаются понизить количество проходов компиляторов для увеличения скорости работы и уменьшения объема необходимой памяти. Однако не всегда количество проходов можно уменьшить, т.к. количество оно определяется грамматикой языка и семантическими правилами.

Компилятор с языка Си является двухпроходным:

1 проход - предпроцессорный, т.е. когда выполняются подстановки;

2 проход - лексический, синтаксический и семантический анализ. Результатом является объектный модуль.

Первые компиляторы были 7-ми и 8-ми - проходные по причине нехватки памяти.

Приложение 2

Директива описания типа микропроцессора

Назначение директив описания типа микропроцессора:

1. .8086 - Разрешены инструкции базового процессора i8086 (и идентичные им инструкции процессора i8088). Запрещены инструкции более поздних процессоров.
2. .186 .286 .386 .486 .586 .686 - Разрешены инструкции соответствующего процессора x86 (x=1,…,6). Запрещены инструкции более поздних процессоров.
3. .187 .287 .387 .487 .587 -Разрешены инструкции соответствующего сопроцессора x87 наряду с инструкциями процессора x86. Запрещены инструкции более поздних процессоров и сопроцессоров.
4. .286c .386c .486c .586c .686c - Разрешены непривилегированные инструкции соответствующего процессора x86 и сопроцессора x87. Запрещены инструкции более поздних процессоров и сопроцессоров.
5. .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.
6. .mmx - Разрешены инструкции MMX-расширения.
7. .xmm - Разрешены инструкции XMM-расширения.
8. .K3D- Разрешены инструкции AMD 3D.

Модель памяти

Модель памяти задается директивой .model.

Строка .model flat указывает на создание exe-файл для 32-разрядной операционной системы Windows. Плоская (flat) модель памяти 32-разрядной Windows располагает тремя сегмента (сегмент кода, стека и данных) в едином четырехгигабайтном адресном пространстве, позволяя забыть о существовании сегментов. Но для 16-разрядных приложений MS-DOS и Windows 3.x максимально допустимый размер сегментов составляет всего лишь 64 килобайта, что явно не достаточно для большинства приложений. В крошечной (tiny) модели памяти сегмент кода, стека и данных также расположены в едином 64-килобайтном адресном пространстве, но в отличие от плоской модели это адресное пространство чрезвычайно ограничено в размерах, поэтому и код, и стек, и данные более серьезных приложений приходиться размещать в нескольких сегментах (модели памяти small, medium, compact, large, huge, tchuge). В этих моделях памяти, например, для вызова функции недостаточно знать ее смещение, а требуется указать еще и сегмент, в котором функция была расположена. Команды передачи управления переходы (jmp) и вызовы (call) в этих моделях памяти могут быть близкими (near) и дальними (far). Отметим, что при написании программы для 32/64-разрядной операционной системы Windows, другие модели памяти, кроме flat, не применяются.

Для адресации четырех гигабайтов виртуальной памяти, выделенной в распоряжение процесса, ОС Windows использует два селектора, один из которых загружается в сегментный регистр CS, а другой в регистры DS, ES и SS. Оба селектора ссылаются на один и тот же базовый адрес памяти, равный нулю, и имеют идентичные лимиты, равные четырем гигабайтам. ОС Windows использует еще и регистр FS, в который загружается селектор сегмента, содержащего информационный блок потока TIB.

Фактически существует всего один сегмент, вмещающий в себя и код, и данные, и стек процесса. Благодаря этому передача управления коду, расположенному в стеке, осуществляется близким (near) вызовом или переходом. Отличия между регионами кода, стека и данных заключаются в атрибутах принадлежащих им страниц - страницы кода допускают чтение и исполнение, страницы данных чтение и запись, а страницы стека чтение, запись и исполнение одновременно.

Допустим у нас есть логический адрес 0137:00456789h. Чтобы этот адрес перевести в линейный – в селекторе 137 находится соответствующий ему дескриптор в таблице дескрипторов: база = 0, граница = 0FFFFFFFh, следовательно, линейный адрес равен 0 (база) + 00456789h. Однако линейный адрес не является физическим адресом. Для его получения используется третья ступень – страничная адресация. То есть 20 старших бит линейного адреса используются для выбора 4 Кбайт памяти из каталога страниц, оставшиеся 12

бит представляют смещение внутри полученной страницы (в качестве упражнения рекомендую написать небольшую программу под 32-разрядной Windows, которая будет показывать сегментные регистры, значения дескрипторов для каждого селектора, базу, границу, RPL и т.п). В 32-разрядной Windows и сегмент кода, и сегмент данных и стека приложения имеют одинаковые базу и границу (0 и 0FFFFFFFFh). Это называется плоской (FLAT) моделью памяти. Хотя cs и ds имеют разные значения и дескрипторы, они указывают на одно и то же линейное адресное пространство 0..0FFFFFFFFh. Следовательно, логические адреса cs:12345678 и ds:12345678 совпадают. Есть возможность модифицировать код при помощи mov byte ptr $+8,21h (секция кода должна быть помечена как writeable). В данном случае в инструкции mov неявно подразумевается ds:, в который можно писать. Однако, при попытке сделать mov cs:xxxxxxxx, получим исключение (сегментная защита). В сегмент кода писать нельзя, но зато можно писать в сегмент данных, который «совпадает» с сегментом кода, и тем самым модифицировать код. А теперь вспомним про страничную защиту. Именно она используется в Windows, когда Вы задаете атрибуты секций PE-файла (.data, .code и т.д). Собственно, к сегментам памяти они не имеют отношения, посему когда речь идет о Win32, не путайте понятия секций PE-файлов и сегментов памяти! Когда Windows грузит PE-файл, она смотрит атрибуты секций и соответственно им устанавливает «защиту» страниц памяти, в которые будет загружена секция. Это и есть типа страничная защита.

Помимо этого каждая страница имеет специальный флаг, определяющий уровень привилегий, необходимых для доступа к этой странице. Некоторые страницы, например, те, что принадлежат операционной системе, требуют наличия прав супервизора, которыми обладает только код нулевого кольца. Прикладные программы, исполняющиеся в кольце 3, таких прав не имеют и при попытке обращения к защищенной странице порождают исключение.

Директива .model

Таблица 6

Модель памяти

Модель памяти	Количество и размер сегментов		Тип указателя		Описание
	кода	Данных	для кода	для данных	кода 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, <=64K	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, <=64K	несколько, <=64K	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, >64K	несколько, >64K	huge	huge	Код может размещаться в нескольких сегментах, на каждый модуль новый сегмент. Данные и код имеют тип huge

Tchuge	несколько, <=64K	несколько, <=64K	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

Для создания приложения на ассемблере необходимо сохранить

Файл с кодом под любым именем и дать ему расширение .asm. Чтобы запустить эту программу, надо будет этот файл ассемблировать и скомпоновать. Для автоматизации процесса можно создать bat-файл следующего содержания:

Листинг asm1.bat.

```
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
```

Необходимо разместить его в папке Windows/System32 и связать файлы с расширением .asm с файлом asm1.bat.

Теперь для компиляции и линковки достаточно будет просто щелкнуть по файлу с расширением .asm. Если все набрано правильно, то в текущем каталоге появится файл с именем исходника, но с расширением .exe. Некоторые особенности компилятора MASM представлены в приложении 1

Также представим файл для компиляции в TASM

```
Compile.bat
cls
echo off
d:\tasm_5\BIN\tasm32 /m3 /ml %1,,;
d:\tasm_5\BIN\tlink32 /Tpe /aa %1,%1,,d:\TASM_5\Lib\import32.lib
echo "Succesful"
```

Приложение 3

Листинг программы printf-1.com

```
use16                   ;Генерировать 16-битный код
org 100h                ;Программа начинается с адреса 100h
   jmp start            ;Переход на метку start
s_sf   db 'FLAGS: SF=$'
s_zf   db ' ZF=$'
s_af   db ' AF=$'
s_pf   db ' PF=$'
s_cf   db ' CF=$'
s_endl      db 13,10,'$'
s_pak db 'Press any key...$'
start:
   mov al,120
   add al,56            ;Пример арифметической операции
   call print_flags     ;Вызов процедуры вывода состояния флагов
   cmc                  ;Инверсия флага CF
   call print_flags     ;Вызов процедуры вывода состояния флагов
   xor ax,ax            ;Пример логической операции
   call print_flags     ;Вызов процедуры вывода состояния флагов
   mov ah,9
   mov dx,s_pak
   int 21h              ;Вывод строки 'Press any key...'
   mov ah,8             ;Функция DOS 08h - ввод символа без эха
   int 21h
   mov ax,4C00h         ;\
   int 21h              ;/ Завершение программы
;Процедура вывода состояния флагов на консоль
```

```
print_flags:
    push ax
    push cx
    push dx
    pushf               ;Сохранение регистра флагов
    lahf            ;Загрузка младшего байта FLAGS в AH
    mov cl,ah           ;CL = AH
    mov ah,9            ;Функция DOS 09h - вывод строки
    mov dx,s_sf         ;DX = адрес строки 'FLAGS: SF='
    int 21h             ;Обращение к функции DOS
    shl cl,1            ;Сдвиг CL влево на 1 бит
    call print_cf       ;Печать выдвинутого бита
    mov dx,s_zf
    int 21h             ;Вывод строки ' ZF='
    shl cl,1            ;Сдвиг CL влево на 1 бит
    call print_cf       ;Печать выдвинутого бита
    mov dx,s_af
    int 21h             ;Вывод строки ' AF='
    shl cl,2            ;Сдвиг CL влево на 2 бита
    call print_cf       ;Печать выдвинутого бита
    mov dx,s_pf
    int 21h             ;Вывод строки ' PF='
    shl cl,2            ;Сдвиг CL влево на 2 бита
    call print_cf       ;Печать выдвинутого бита
    mov dx,s_cf
    int 21h             ;Вывод строки ' CF='
    shl cl,2            ;Сдвиг CL влево на 2 бита
    call print_cf       ;Печать выдвинутого бита
    mov dx,s_endl
    int 21h             ;Вывод конца строки
    popf                ;Восстановление регистра флагов
    pop dx
    pop cx
    pop ax
    ret
;Процедура вывода значения флага CF в виде символа
print_cf:
    push ax
    push dx
    mov ah,2            ;Функция DOS 02h - вывод символа
    mov dl,'0'          ;DL = '0'
    adc dl,0            ;Если CF = 1, то в DL будет символ '1'
    int 21h             ;Обращение к функции DOS
```

```
pop dx
pop ax
ret
```

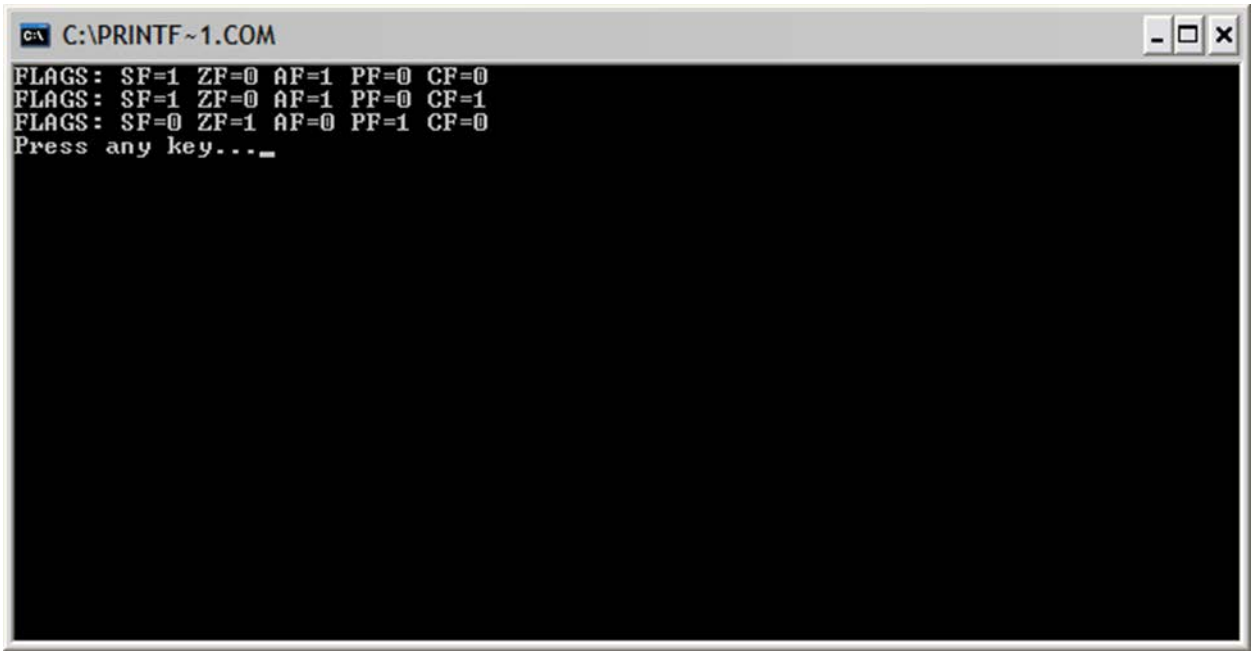

Рис.7 Результат работы программы (printf-1.com)

В таблице приведены воздействия команд на флаги в регистре флагов EFLAGS, при этом используются следующие обозначения:

T : Флаг проверяется.
M : Флаг изменяет своё значение
0 : Флаг сбрасывается.
1 : Флаг устанавливается.
- : Значение флага не определено.
R : Значение флага восстанавливается в предыдущее.
Пустое место : Флаг не меняется.

Таблица 7

Взаимодействие команд с флагами.

Команда	OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
AAA	-	-	-	TM	-	M					
AAD	-	M	M	-	M	-					
AAM	-	M	M	-	M	-					
AAS	-	-	-	TM	-	M					
ADC	M	M	M	M	M	TM					
ADD	M	M	M	M	M	M					

AND	0	M	M	-	M	0					
ARPL			M								
BSF	-	-	M	-	-	-					
BSR	-	-	M	-	-	-					
BT	-	-	-	-	-	M					
BTS	-	-	-	-	-	M					
BTR	-	-	-	-	-	M					
BTC	-	-	-	-	-	M					
CLC						0					
CLD									0		
CLI								0			
CMC						M					
CMOVcc	T	T	T		T	T					
CMP	M	M	M	M	M	M					
CMPS	M	M	M	M	M	M			T		
CMPXCHG	M	M	M	M	M	M					
CMPXCHG8B			M								
COMSID	0	0	M	0	M	M					
COMISS	0	0	M	0	M	M					
DAA	-	M	M	TM	M	TM					
DAS	-	M	M	TM	M	TM					
DEC	M	M	M	M	M						
DIV	-	-	-	-	-	-					
FCMOVcc			T		T	T					
FCOMI			M		M	M					
FCOMIP			M		M	M					
FUCOMI			M		M	M					
FUCOMIP			M		M	M					
IDIV	-	-	-	-	-	-					
IMUL	M	-	-	-	-	M					
IN											
INC	M	M	M	M	M						
INS									T		
INT							0			0	
INTO	T						0			0	
UCOMSID	0	0	M	0	M	M					
UCOMISS	0	0	M	0	M	M					
IRET	R	R	R	R	R	R	R	R	R	T	
Jcc	T	T	T		T	T					
LAR			M								
LODS									T		
LOOPE			T								
LOOPNE			T								
LSL			M								
MOV control	-	-	-	-	-	-					
MOV debug	-	-	-	-	-	-					
MOV test	-	-	-	-	-	-					

MOVS									T		
MUL	M	-	-	-	-	M					
NEG	M	M	M	M	M	M					
OR	0	M	M	-	M	0					
OUTS									T		
POPF	R	R	R	R	R	R	R	R	R	R	
RCL на 1 бит	M					TM					
RCR на 1 бит	M					TM					
RCL на n бит	-					TM					
RCR на n бит	-					TM					
ROL на 1 бит	M					M					
ROR на 1 бит	M					M					
ROL на n бит	-					M					
ROR на n бит	-					M					
RSM	M	M	M	M	M	M	M	M	M	M	M
SAHF		R	R	R	R	R					
SAL на 1 бит	M	M	M	-	M	M					
SAR на 1 бит	M	M	M	-	M	M					
SHL на 1 бит	M	M	M	-	M	M					
SHR на 1 бит	M	M	M	-	M	M					
SAL на n бит	-	M	M	-	M	M					
SAR на n бит	-	M	M	-	M	M					
SHL на n бит	-	M	M	-	M	M					
SHR на n бит	-	M	M	-	M	M					
SBB	M	M	M	M	M	TM					
SCAS	M	M	M	M	M	M			T		
SETcc	T	T	T		T	T					
SHLD	-	M	M	-	M	M					
SHRD	-	M	M	-	M	M					
STC						1					
STD									1		
STI								1			
STOS									T		
SUB	M	M	M	M	M	M					
TEST	0	M	M	-	M	0					
VERR			M								
VERRW			M								
XADD	M	M	M	M	M	M					
XOR	0	M	M	-	M	0					

Приложение 4

Листинг 15. Простой пример программы для Windows (MASM32)

```
.586P
;плоская модель памяти
.MODEL FLAT, stdcall
;константы
;сообщение приходит при закрытии окна
WM_CLOSE equ 10h
;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
;сообщение при щелчке правой кнопкой мыши в области окна
WM_RBUTTONDOWN equ 204h
;свойства окна
CS_VREDRAW equ 1h
CS_HREDRAW equ 2h
CS_GLOBALCLASS equ 4000h
WS_OVERLAPPEDWINDOW equ 000CF0000H
style equ CS_HREDRAW+CS_VREDRAW+CS_GLOBALCLASS
;идентификатор стандартной пиктограммы
IDI_APPLICATION equ 32512
;идентификатор курсора
IDC_CROSS equ 32515
;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
;директивы компоновщику для подключения библиотек
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib;структуры
;структура сообщения
MSGSTRUCT STRUC
MSHWND DD ? ;идентификатор окна, получающего сообщение
MSMESSAGE DD ? ;идентификатор сообщения
MSWPARAM DD ? ;дополнительная информация о сообщении
MSLPARAM DD ? ;дополнительная информация о сообщении
MSTIME DD ? ;время посылки сообщения
MSPT DD ? ;положение курсора, во время посылки сообщения
MSGSTRUCT ENDS
;---------
WNDCLASS STRUC
CLSSTYLE DD ? ;стиль окна
CLWNDPROC DD ? ;указатель на процедуру окна
CLSCEXTRA DD ? ;информация о дополнительных байтах
;для данной структуры
CLWNDEXTRA DD ? ;информация о дополнительных байтах для окна
CLSHINSTANCE DD ? ;дескриптор приложения
CLSHICON DD ? ;идентификатор пиктограммы окна
CLSHCURSOR DD ? ;идентификатор курсора окна
CLBKGROUND DD ? ;идентификатор кисти окна
CLMENUNAME DD ? ;имя-идентификатор меню
CLNAME DD ? ;специфицирует имя класса окон
WNDCLASS ENDS
;сегмент данных
_DATA SEGMENT
NEWHWND DD 0
MSG MSGSTRUCT <?>
WC WNDCLASS <?>
HINST DD 0 ;здесь хранится дескриптор приложения
TITLENAME DB 'Простой пример 32-битного приложения',0
CLASSNAME DB 'CLASS32',0
CAP DB 'Сообщение',0
MES1 DB 'Вы нажали левую кнопку мыши',0
MES2 DB 'Выход из программы. Пока!',0
MES3 DB 'Закрытие окна',0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
```

```
;получить дескриптор приложения
PUSH 0
CALL GetModuleHandleA@4
MOV HINST, EAX
REG_CLASS:
;заполнить структуру окна
; стиль
MOV WC.CLSSTYLE,style
;процедура обработки сообщений
MOV WC.CLWNDPROC, OFFSET WNDPROC
MOV WC.CLSCEXTRA, 0
MOV WC.CLWNDEXTRA, 0
MOV EAX, HINST
MOV WC.CLSHINSTANCE, EAX
;----------пиктограмма окна
PUSH IDI_APPLICATION
PUSH 0
CALL LoadIconA@8
MOV WC.CLSHICON, EAX
;----------курсор окна
PUSH IDC_CROSS
PUSH 0
CALL LoadCursorA@8
MOV WC.CLSHCURSOR, EAX
;----------
MOV WC.CLBKGROUND, 17 ;цвет окна
MOV DWORD PTR WC.CLMENUNAME,0
MOV DWORD PTR WC.CLNAME, OFFSET CLASSNAME
PUSH OFFSET WC
CALL RegisterClassA@4
;создать окно зарегистрированного класса
PUSH 0
PUSH HINST
PUSH 0
PUSH 0
PUSH 400 ; DY - высота окна
PUSH 400 ; 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 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_CLOSE ;закрытие окна
JE WMCLOSE
CMP DWORD PTR [EBP+0CH], WM_CREATE
JE WMCREATE
CMP DWORD PTR [EBP+0CH],WM_LBUTTONDOWN ;левая кнопка
JE LBUTTON CMP DWORD PTR [EBP+0CH],WM_RBUTTONDOWN ;правая
кнопка
JE RBUTTON
JMP DEFWNDPROC
;нажатие правой кнопки мыши приводит к закрытию окна
RBUTTON:
JMP WMDESTROY
;нажатие левой кнопки мыши
LBUTTON:
;выводим сообщение
PUSH 0 ;MB_OK
PUSH OFFSET CAP
PUSH OFFSET MES1
PUSH DWORD PTR [EBP+08H]
CALL MessageBoxA@16
MOV EAX, 0
JMP FINISH
WMCREATE:
MOV EAX, 0
JMP FINISH
WMCLOSE:
PUSH 0 ;MB_OK
PUSH OFFSET CAP
PUSH OFFSET MES3
PUSH DWORD PTR [EBP+08H] ;дескриптор окна
CALL MessageBoxA@16
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 ;MB_OK
PUSH OFFSET CAP
PUSH OFFSET MES2
PUSH DWORD PTR [EBP+08H] ;дескриптор окна
```

Приложение 5

Дизассемблер

Дизассемблирование (От англ. disassemble - разбирать, демонтировать) – это процесс или способ получения исходного текста программы на ассемблере из программы в машинных кодах. Полезен при определении степени оптимальности транслятора и при генерации кодов собственной программы. Позволяет понять алгоритм или метод построения программ, у которых отсутствуют исходные тексты.

Существуют специальные программы дизассемблеры, которые выполняют этот процесс Одна из фундаментальных проблем дизассемблирования заключается в синтаксической неотличимости констант от адресов памяти (сегментов и смещений). Потребность распознавания смещений объясняется необходимостью замены конкретных адресов на метки, действительное смещение которых определяется на этапе ассемблирования программы.

Цель дизассемблирования заключается в содействии исследованию функционирования программ, когда их исходные коды не доступны.
Наиболее распространенные цели дизассемблирования:

1. - анализ вредоносного программного обеспечения;
2. - анализ уязвимостей программного обеспечения с закрытым кодом;
3. - анализ совместимости программного обеспечения с закрытым кодом;
4. - валидация компилятора;
5. - отображение команд программы в процессе отладки.

По типу реализации интерфейса взаимодействия с пользователем, существующие дизассемблеры можно разделить на две категории – автономные и интерактивные. Автономные дизассемблеры требуют от пользования задания всех указаний до начала дизассемблирования и не позволяют вмешиваться непосредственно в сам процесс. Интерактивные дизассемблеры обладают развитым пользовательским интерфейсом, благодаря которому приобретают значительную гибкость, позволяя человеку «вручную» управлять разбором программы, помогая автоматическому анализатору– отличать адреса от констант, определять границы инструкций и т.д.

Примером автономного дизассемблера является SOURCER, а интерактивного – IDA. Дизассемблеры бывают однопроходные и многопроходные. Основная трудность при работе дизассемблера — отличить данные от машинного кода, поэтому на первых проходах автоматически или интерактивно собирается информация о границах процедур и функций, а на последнем проходе формируется итоговый листинг. Интерактивность позволяет улучшить этот процесс, так как просматривая дамп дизассемблируемой области памяти, программист может сразу выделить

строковые константы, дать содержательные имена известным точкам входа, прокомментировать разобранные им фрагменты программы.

В процесс дизассемблирования входит:

1) идентификация математических операций таких как: сложение, вычитание, деление, операция вычисления остатка, умножение и определения комплексных операций;

2) Идентификация операторов SWITCH - CASE – BREAK.

Базовый алгоритм дизассемблирования:

1. Первым шагом в процессе дизассемблирования является идентификация кодового сегмента. Так как команды обычно смешаны сданными, то дизассемблеру необходимо их разграничить.
2. Получив адрес первой команды, необходимо прочитать значение, содержащееся по этому адресу (или смещению в файле) и выполнить табличное преобразование двоичного кода операции в соответствующую ему мнемонику языка ассемблера.
3. Как только команда была обнаружена и декодирована, ее ассемблерный эквивалент может быть добавлен к результирующему листингу. После этого необходимо выбрать одну из разновидностей синтаксиса языка ассемблера.
4. Далее необходимо перейти к следующей команде и повторить предыдущие шаги до тех пор, пока каждая команда файла не будет дизассемблирована.

Одним из передовых продуктов для дизассемблирования программ является пакет программ от CSO Computer Services - IDA (Interactive Disassembler). IDA не является автоматическим дизассемблером. Это означает, что IDA выполняет дизассемблирование лишь тех участков программного кода, на которые имеются явные ссылки. При этом IDA старается извлечь из кода максимум информации, не делая никаких излишних предположений. После завершения предварительного анализа программы, когда все обнаруженные явные ссылки исчерпаны, IDA останавливается и ждет вмешательства; просмотрев готовые участки текста, можно как бы подсказать ей, что нужно делать дальше. После каждого вмешательства снова запускается автоматический анализатор IDA, который на основе полученных сведений пытается продолжить дизассемблирование.

IDA является не только дизассемблером, но и одним из самых мощных средств исследования программ. Это возможно благодаря наличию развитой навигационной системы, позволяющей быстро перемещаться между различными точками программы, объектами и ссылками на них, отыскивать неявные ссылки и т.д. Исследование даже больших и сложных программ в IDA занимает в десятки и сотни раз меньше времени, чем путем просмотра текста, полученного обычным дизассемблером.

Примеры программ-дизассемблеров:

1. IDA (отличается исключительной гибкостью, наличием встроенного командного языка, поддерживает множество форматов исполняемых файлов для большого числа процессоров и операционных

систем.Позволяет строить блок-схемы, изменять названия меток, просматривать локальные процедуры в стеке и многое другое.В последних версиях имеет встроенный отладчик

2. Sourcer(интерактивный дизассемблер. Т.е. в нем, как и в IDA, тоже есть возможность взаимодействия с пользователем. Правда, весьма кривая и сложная - через редактирование.
3. Hiew (редактор двоичных файлов, ориентированный на работу с кодом. Имеет встроенный дизассемблер для x86, x86-64 и ARM, ассемблер для x86, x86-64)
4. Beye (мультиплатформенный редактор файлов с поддержкой бинарного, шестнадцатеричного и дизассемблерных режимов. Для дизассемблирования используется Intel-синтаксис. Цветные AVR/Java/x86-i386-AMD64/ARM-XScale/PPC64 дизассемблеры, перекодировщик кодовых страниц, полный просмотр форматов — MZ, NE, PE, NLM, coff32, ELF частичный — a.out, LE, LX, Phar Lap; навигатор по коду.)
5. HT editor (ассемблер/дизассемблер)
6. Hacker Disassembler Engine (Небольшой дизассемблерный движок, предназначенный для анализа x86-32 кода. HDE определяет длину команды, префиксы, байты ModR/M и SIB, опкод, значения immediate, и др. HDE может пригодиться, например, при написании распаковщиков, расшифровщиков, вирусов исполняемых файлов, когда простого дизассемблера длин недостаточно, а большой дизассемблер тащить тяжело)
7. CADt
8. Vb-decompiler (это декомпилятор исполняемых модулей (EXE, DLL, OCX), созданных при помощи Visual Basic 6.0)
9. Radare2 (свободный кроссплатформенный фреймворк для реверс-инжиниринга написанный на Си, который включает дизассемблер, шестнадцатеричный редактор, анализатор кода и т. д. Используется при реверсе и отладке вредоносного ПО и прошивок)

Кроссплатформенность — способность программного обеспечения работать более чем на одной аппаратной платформе и (или) операционной системе.

Фре́ймворк (иногда фреймво́рк ; англицизм, неологизм от framework — каркас, структура) — программная платформа, определяющая структуру программной системы; программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта.

Обра́тная разрабо́тка (обратный инжиниринг, реверс-инжиниринг; англ. reverse engineering) — исследование некоторого готового устройства или программы, а также документации на него с целью понять принцип его работы; например, чтобы обнаружить недокументированные возможности (в том числе программные закладки), сделать изменение или воспроизвести устройство,

программу или иной объект с аналогичными функциями, но без прямого копирования.

Декомпиляция – получение кода языка высокого уровня из программы намашинном языке или ассемблере. Декомпиляция – достаточно сложный процесс. Это обусловлено следующими причинами:

- Процесс компиляции происходит с потерями. В машинном языке нет имен переменных и функций, и тип данных может быть определен только по производимым над ними операциям. Наблюдая пересылку 32-х бит данных, требуется значительная работа, чтобы определить, являются ли эти данные целым числом, дробью или указателем.
- Компиляция это операция типа множество-множество. Компиляция и декомпиляция могут быть выполнены множеством способов. Поэтому результат декомпиляции может значительно отличаться от исходного кода.
- Декомпиляторы в значительной степени зависимы от конкретного языка и библиотек. Обрабатывая исполняемый файл, созданный компилятором Delphi, декомпилятором, разработанным для C, можно получить фантастический результат.
- Необходимо точное дизассемблирование исполняемого файла. Любая ошибка или упущение на фазе дизассемблирования практически наверняка размножатся в результирующем коде.

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.Сван Т. Программирование для Windows в Borland C++. - М: Бином, 1996.
4. Финогенов К.Г. C/C++:Win32. Основы программирование. - М: Диалог. - МИФИ, 2006.
5. Таненбаум Э. Современные операционные системы. - СПб: Питер, 2010.
6. Джонсон М. Харт. Системное программирование в среде WIN 32. - 2-е изд. - М. Вильямс, 2014.
7. Солдатов В.П. Программирование драйверов под Windows. - М.:Бином, 2011.
8. Олифер В.Г., Олифер Н.А. Сетевые операционные системы. - СПб: Питер, 2015.
9. www.microsoft.ru
10.http://www.movsd.com