Глава 9. Модульное программирование

Модульность – фундаментальный аспект всех успешно работающих крупных систем.

Бьёрн Страуструп

Самым важным отличием хорошо спроектированного модуля от плохо спроектированного является степень, в которой он скрывает свои внутренние данные и другие детали реализации от других модулей.

Джошуа Блох

Архитектура ЭВМ тесно связана со способами выполнения машинных программ, поэтому сейчас необходимо перейти к изучению большого раздела нашего курса под названием «Системы программирования», строгое определение этого понятия будет дано в следующей главе. Нами будет рассмотрен весь путь, который проходит новая программа – от написания текста на некотором языке программирования, до этапа счёта этой программы. Первая тема в этом разделе называется «Модульное программирование».

Модульное программирование является таким способом разработки программы, при котором вся программа строится из нескольких относительно независимых друг от друга частей – модулей. Понятие модуля является одним из центральных при разработке программного обеспечения, и сейчас будет начато изучение этого понятия.

Известно, что при разработке программ в основном используется метод, который называется «программирование сверху вниз» или «пошаговая детализация». Легко понять суть этого метода: исходная задача сначала разбивается на достаточно большие относительно независимые друг от друга подзадачи. В том случае, когда полученные подзадачи всё ещё сложны, чтобы написать для них соответствующий алгоритм, то каждая такая все ещё сложная подзадача снова разбивается на более простые подзадачи и т.д. Далее приступают к реализации каждой из полученных таким образом относительно простых подзадач на некотором языке программирования. Набор таких подзадач, ориентированных на решение определённого класса действий и называется чаще всего (программным) модулем.

> Модульность больших программ является следствием свойства *структурности* алгоритма. Напомним, что по этому свойству любой достаточно сложный алгоритм состоит из частей, которые, в свою очередь, тоже являются алгоритмами.

Например, можно создать модуль, реализующий работу с графикой, модуль, содержащий операции для работы с матрицами и т.д. Таким образом, использование модулей является естественным способом разработки и построения сложных программных комплексов. Заметим, что и сам программный модуль допускает дальнейшую пошаговую детализацию, однако полученные подзадачи реализуются уже, как правило, в виде процедур и функций. Таким образом, процедуры и функции являются инструментом пошаговой детализации при разработке программ на нижнем, более детальном уровне, а модули – на верхнем.

Модули задачи могут писаться как на одном языке программирования, так и на разных языках, в этом случае говорят, что используется многоязыковая система программирования. Вам уже известно одно из полезных свойств программы, отдельные части (модули) которой написаны на разных языках программирования – это позволяет нам из программ на языках высокого уровня вызывать процедуры на Ассемблере (и, вообще говоря, наоборот). Заметим, что модуль на Ассемблере при этом может быть и совсем маленьким.

Иногда использование многоязыковой системы программирования является единственнным способом достичь нужной производительности. Например, программы на интерпретируемых языках программирования (скажем, на языке Python) выполняются во много тысяч раз медленнее, чем аналогичные программы на компилируемых языках (C, Фортран и т.д.). Когда это неприемлемо, то для повышения производительности приходится вызывать подпрограммы из библиотек, написанных на «быстрых» языках (программные «ускорители»).

В основном принципы модульного программирования будут разбираться на примере языка Ассемблер, однако будут изучаться также и связи модулей, написанных на Ассемблере и языке высокого уровня Free Pascal.

Перечислим сначала те преимущества, которые предоставляет модульное программирование.

- Во-первых, как уже отмечалось, это возможность писать модули на разных языках программирования.
- Во-вторых, модуль является естественной единицей локализации имен: как уже говорилось, внутри модуля на Ассемблере почти все имена должны быть различны (уникальны). Учтите, что на Ассемблере большинство имён видно во всём модуле, а не только ниже по тексту программы, как в языках программирования высокого уровня. Из этого правила видимости имён в Ассемблерном модуле совсем немного исключений.

  Видимость имён во всём модуле не очень удобна, особенно когда модуль большой по объёму или совместно пишется разными программистами. Как Вы уже должны знать, в этом случае используется локализация имён в процедурах и функциях. Таким образом, в разных модулях Ассемблера имена могут совпадать, они не видны из другого модуля, если только это не указано явно с помощью специальных директив.
- Следующим преимуществом модульного программирования является локализация места ошибки: обычно исправление ошибки внутри одного модуля не влечёт за собой появление ошибок в других модулях. Разумеется, это свойство будет выполняться только при *хорошем* разбиении программы на модули, с малым числом *связей* между модулями, о чем будет говориться далее. Это преимущество особенно сильно сказывается во время отладки программы. Например, при внесении изменений только в один из нескольких десятков или сотен модулей программы, только он и должен быть заново проверен компилятором и заново переведён на язык машины.[1] Обычно говорят о независимой компиляции (independent compilation) модулей и *малом времени перекомпиляции* всей программы, что сильно ускоряет процесс отладки программы. Действительно, полная перекомпиляция больших программных проектов, содержащих сотни тысяч и даже миллионы строк текста, может потребовать нескольких часов машинного времени.
- Следует отметить и такое хорошее свойство модульного программирования, как возможность повторного использования (reuse) разработанных модулей в других программах.[2] Очевидно, что для повторного использования программных модулей их лучше оформлять в виде наборов процедур и функций со стандартными соглашениями о связях. И, наконец, ещё об одном преимуществе модульного программирования будет говориться позже при рассказе о динамических библиотеках.

Разумеется, за все надо платить, у модульного программирования есть и свои слабые стороны, перечислим основные из них.

- Во-первых, модули не являются совсем уж независимыми друг от друга: между ними существуют *связи*, то есть один модуль иногда может использовать переменные, константы и программный код другого модуля. Необходимость связей между модулями естественно вытекает из того факта, что модули *совместно* решают одну общую задачу, при этом каждый модуль выполняет свою часть задачи, получая от других модулей входные данные и как-то передавая им результаты своей работы. Эти связи надо безошибочно описать.
- Во-вторых, теперь перед счётом программы необходим особый этап *сборки* программы из составляющих её модулей. Этот процесс достаточно сложен, так как кроме собственно объединения всех модулей в одну программу, необходимо проконтролировать и установить все связи между этими модулями.[3] Сборка программы из модулей производится специальной

---

[1] Точнее на *объектный* язык, о чем будет говориться далее.

[2] Например, при работе на языке Free Pascal программистам хорошо известны модули `Crt` (для работы с клавиатурой и экраном), `Graph` (для графики) и другие, которые можно использовать во многих программах.

[3] Возможно, выполнение программы без её сборки из модулей, при этом установление связей между модулями будет отложено на этап счёта программы, о чем будет говориться позже при изучении схемы работы с так называемой динамической загрузкой модулей.

системной программой, которая называется редактором внешних связей между модулями (linkage editor, русское жаргонное название – линкер или линковщик).

- В-третьих, так как теперь компилятор не видит *всей* исходной программы одновременно, то, следовательно, и не может получить полностью г готовую к счёту программу на машинном языке. Более того, так как в каждый момент времени он видит только *один модуль*, он не может проконтролировать, правильно ли установлены связи между модулями. Ошибка в связях теперь выявляется на этапе сборки программы из модулей, а иногда только на этапе счёта, особенно если используется так называемое *динамическое* связывание модулей, обо всем этом будем говорить далее. Позднее обнаружение ошибок связи между модулями может существенно усложнить и замедлить процесс отладки программы [i] [см. сноску в конце главы].

Несмотря на отмеченные недостатки, преимущества модульного программирования так велики, что сейчас это основной способ разработки сложного программного обеспечения. Теперь начнём знакомиться с особенностями написания модульной программы на языке Ассемблера.

9.1. Модульное программирование на Ассемблере

После изменения в любом программном модуле, в любом другом программном модуле возникнет хотя бы одна ошибка.
Компьютерные законы Мерфи

Как уже говорилось, программа на Ассемблере может состоять из нескольких модулей. Исходным (или входным) программным модулем на Ассемблере называется текстовый файл, состоящий из предложений языка Ассемблер и заканчивающийся директивой **`end`** – признаком конца модуля.

Среди всех модулей, составляющих программу, должен быть один и только один модуль, который называется головным модулем программы. Например, в языке Free Pascal головной модуль имеет (необязательный) заголовок **`program`**, а остальные модули *обязательный* заголовок **`unit`**, в языке C в головном модуле содержится функция с именем `main`.

В головном модуле находится точка входа (entry point) программы, с которой начинается её выполнение. Так, для программ на языке Free Pascal это первый оператор в головном модуле, для языка C это первый оператор в функции `main`. Признаком головного модуля на Ассемблере является параметр-метка у директивы **`end`** конца модуля, такую метку часто называют именем `Start`, хотя можно выбрать любое подходящее имя. Эта метка должна быть меткой *команды* (т.е. быть с двоеточием) или именем процедуры и находится в секции кода головного модуля. Именно эта метка задаёт первую выполняемую команду.

Как уже отмечалось, модули не могут быть абсолютно независимыми друг от друга, так как решают разные части одной общей задачи, и, следовательно, хотя бы время от времени должны обмениваться между собой информацией. Таким образом, между модулями существуют *связи*. Говорят, что между модулями существуют связи по управлению, если один модуль может передавать управление (с возвратом или без возврата) на программный код в другом модуле. В архитектуре нашего компьютера для такой передачи управления можно использовать команду безусловного перехода (включая команды вызова **`call`** и возврата из процедуры **`ret`**).

Кроме связей по управлению, между модулями могут существовать и связи по данным. Они предполагают, что один модуль может иметь доступ по чтению и/или записи к областям памяти (переменным) в другом модуле. Частным случаем связи по данным является и использование одним модулем именованной целочисленной *константы*, определённой в другом модуле (в Ассемблере такая константа объявляется директивой эквивалентности **`equ`** или оператором макроприсваивания **`=`** (об этом будет говориться в главе, посвящённой макросредствам Ассемблера).

Связи между модулями реализуются на языке машины в виде адресов. Это верно и для случая, когда один модуль использует целую константу, определённую в другом модуле, так как такая константа – это непосредственное значение формата `i32`. «Непрямые» связи между модулями с помощью внешних файлов, прерываний и других способов взаимодействия здесь рассматриваться не будут.

Действительно, чтобы выполнить команду из другого модуля, а также считать или записать значение в переменную, нужно знать месторасположение (адрес) этой команды или переменной. Заме-

тим, что численные значения связей между модулями (значения адресов) невозможно установить на этапе *компиляции* модуля. Так происходит потому, что, во-первых, компилятор «видит» только один этот модуль. Во-вторых, будущее расположение модулей (их секций) в памяти во время счёта на этапе компиляции, как правило, неизвестно.

Связи между модулями называются статическими, если численные значения этих связей (т.е. адреса) известны сразу после размещения программы в памяти компьютера, *до начала* счёта программы (до выполнения её первой команды). В отличие от статических, значения динамических связей между модулями становятся известны только *во время* счёта программы. Как правило, статические связи являются именованными, а динамические – безымянными. Вскоре будут приведены примеры статических и динамических связей, как по данным, так и по управлению.

На языке Ассемблера именованные *статические* связи между модулями задаются с помощью специальных директив. Директива

public <список имён данного модуля>

объявляет перечисленные в этой директиве имена *общедоступными* (**public**), т.е. разрешает использование этих имен в других модулях. Общедоступными можно сделать только имена переменных, меток (в том числе имён процедур) и целочисленных констант (в частности, так называемых переменных периода генерации, о них будет говориться в главе 11). Нельзя объявлять общедоступными имена макроопределений, *имена типов пользователя* и *имена полей* упакованных битовых векторов (**record**). Это понятно, так как объекты с этими именами не имеют адресов (Вам обязательно следует это понять). Можно делать общедоступными *имена переменных* типов **record** и **struc**, однако при этом сами эти типы придётся (одинаковым образом) описывать в *каждом* модуле. Лучше всего это делать, включая одинаковые файлы (например, с расширением `.inc`) директивой **include**. Отметим, что в языке C такие включаемые файлы называются заголовочными и имеют расширение `.h`.

Имя процедуры в Ассемблере можно сделать общедоступным и по-другому, указывая слово-модификатор **public** в заголовке процедуры, например:

```
MyPublicProc proc public
```

В некоторых модульных системах программирования про такие имена говорится, что эти имена экспортируются в другие модули.

В модулях (**unit**) на языке Free Pascal для описания общедоступных имен предназначен специальный раздел **interface**, кроме того, как и в Ассемблере, некоторое имя можно объявить **public** и в головной программе. Есть языки, в которых имена одного модуля по умолчанию считаются доступными из других модулей, если они описаны или объявлены в определенной части первого модуля. Например, в модуле на языке C общедоступны все имена, описанные *вне* функций, если про них явно не сказано, что это локальные имена модуля. На Ассемблере по умолчанию общедоступны все имена процедур, впрочем, это можно запретить, указав директиву **option proc:private**. Можно убрать общедоступность и отдельной процедуры, описав её как `MyProc proc private`.

В принципе, вместе с каждым именем может экспортироваться и его *тип*, о чём будет говориться ниже. Тип имени позволяет проводить контроль использования этого имени в другом модуле. Все остальные имена модуля, не имеющие модификатора **public**, являются *локальными* и не видны извне (из других модулей). Как будет описано позже, в нашем Ассемблере имена *сегментов* тоже можно сделать видимыми из других модулей, но другим, более сложным, способом.

Экспортируемые имена одного модуля, не становятся *автоматически* доступными в других модулях на Ассемблере. Для получения доступа к таким именам, этот другой модуль должен явно объявить о своем *желании* использовать общедоступные имена других модулей. Это делается с помощью директивы

extrn <имя:тип>, ... ,<имя:тип>

В этой директиве объявляются *внешние* имена, которые используются в этом модуле, но *не описаны* в нем (у этой директивы есть синонимы **extern** и **externdef** (они используется и в языке C). Внешние имена должны быть описаны и объявлены общедоступными в каких-то других модулях. Вместе с каждым внешним именем объявляется и тип, который должно иметь это имя в другом модуле. Именно на этот тип ориентируется Ассемблер, при действиях с соответствующим внешним

именем. Проверка того, что это имя в другом модуле *на самом деле* имеет такой тип, может проводиться только на этапе сборки из модулей готовой программы, о чем будет говориться далее.

Имена **extern** и **externdef** имеет дополнительную семантику. Когда имя указано в **extern** или **externdef**, но *описано* в данном модуле, то оно автоматически считается **public**. Это позволяет задавать все внешние имена и входные точки *только* в директивах **externdef** во включаемом текстовом файле (например, с расширением `.inc`), который будет вставляться во *все* модули, причём там, где имя описано, оно становится **public**, а там, где не описано считается **extrn**. Так обеспечивается автоматическое соответствие между типом внешнего имени и входной точки (так как они описаны в одной и той же директиве).

Как уже упоминалось, этот механизм является стандартным в языке C, где для этого используются *заголовочные файлы* с расширением `.h`. К сожалению, компилятор MASM версий 6.xx объявляет все имена процедур по умолчанию **public**, для отмены этого правила можно использовать директиву `option proc:private`.

Значением типа имени является 16-битовая *знаковая* константа формата `i16`. Обычно тип задаётся служебным именем Ассемблера, например (указаны значения этого имени в Ассемблере):

abs=0	**byte**=1 **sbyte**=1	**word**=2 **sword**=2	**dword**=4 **sdword**=4	**fword**=6
qword=8 **sqword**=8	**tbyte**=10	**near**=-252= 65284	**proc**=-252= 65284	**far**=-250= 65286
real4=4	**real8**=8	**real10**=10	**oword**=16	**xmmword**=16
ymmword=32	**zmmword**=64 [1]			

Отрицательные значения типов **near**, **proc** и **far** определяют внешние метки и процедуры, **near** (и его синоним **proc**) в текущем сегменте кода и **far** в другом сегменте кода (тип **far** мы использовать не будем). Тип **abs** определяет целочисленную константу, формат константы (`i8`, `i16`, `i32`, или `i64`) не конкретизируется. Типы с положительными значениями задают *размер* внешней переменной в байтах. В качестве типа можно указывать также имя типа структуры (**struc**), упакованных битовых полей (**record**) или объединений (**union**), тогда значение типа определяется размером переменной этого типа, например:

```
extrn St1: Stud; type Stud=25 (см. разд. 8.5)
```

Можно заметить, что «внешний» тип имени соответствует «внутреннему» типу, который выдаёт оператор **type**. Например, переменная `X db ?` имеет тип `type X=1` и в другом модуле должна быть описана как `extrn X: byte`.

Таким образом, для установления связи между двумя модулями на Ассемблере первый модуль должен *разрешить* использовать некоторые из своих имен в других модулях, а второй модуль – явно *объявить*, что он хочет использовать внутри себя такие имена. В языке Ассемблера общедоступные имена называются входными точками модуля, что хорошо отражает суть дела, так как только в эти точки возможен доступ к модулю извне (из других модулей). Внешние имена модуля называются внешними адресами, так как это адреса областей памяти и команд, а также целочисленные значения констант в других модулях.

Специфическим случаем общедоступной из всех модулей Ассемблера является так называемая *общая переменная* (Common Variable). Считается, что общая переменная описана где-то вне всех модулей, и становится в каждом модуле доступной после её объявления, например:

```
comm ComMas: dword:100
```

объявляет общую для всех модулей *неинициализированную* переменную, где-то описанную как

```
extrn ComMas:dword
ComMas  dd 100 dup (?)
```

---

[1] Типы **xmmword**, **ymmword** и **zmmword** имеют векторные регистры и соответствующие им переменные (только в версиях MASM 10.xx и выше).

Преимуществом этого метода перед связью модулей по данным посредством директив **public** и **extrn** является то, что ни один модуль не должен *описывать* у себя такую переменную, т.е. резервировать для неё память, это сделает загрузчик (Вам должно быть понятно, почему это *не инициализированная* переменная).

Изучим теперь вопрос о том, какие имена имеют общедоступные переменные «во внешнем мире», т.е. как их видят другие модули. Имена общедоступных переменных для «внешнего мира» изменяются, часто говорят, декорируются (name mangling или name decoration), и не совпадают с внутренними именами этих переменных. [ii] [см. сноску в конце главы].

Компилятор с Ассемблера делает внешние имена констант, переменных и меток, просто приписывая впереди символ подчёркивания, например:

```
public  Abc,N,L
N     equ 100
Abc   db  ?
_Abc  db  ?; Внутреннее имя!
L:    inc eax
```

Имена Abc, N и L во внешнем мире приобретают вид _Abc, _N, _L (регистр букв сохраняется), именно так они записываются в паспорт (заголовок) получаемого компилятором Ассемблера объектного модуля. Программист на Ассемблере может не обращать на это внимания, потому что компилятор, увидев в другом модуле объявление имён:

```
extrn Abc:byte,N:abs,L:near
```

припишет им такие же внешние имена _Abc, _N, _L, и, таким образом такое «декорирование» имён проходит для программиста на Ассемблере незамеченным. Переменная с «изначальным» именем _Abc остаётся локальной, недоступной извне. Разумеется, такое имя лучше не использовать, чтобы не путать самого себя. Отметим, что значение *внешней* константы, скажем, N в этом модуле на этапе компиляции неизвестно, оно становится известным только на этапе счёта:

```
Y dd N dup (?); НЕЛЬЗЯ, N при компиляции неизвестна
outint N;      МОЖНО, N при счёте известна
```

Сложнее обстоит дело с назначением внешних имён *процедурам*. Для описания внешней процедуры, кроме имени, желательно получить и информацию о числе её параметров. Исходя из этого, компилятор Ассемблера, увидев предложения

```
MyProc proc public
```

сгенерирует внешнее имя процедуры в виде `_MyProc@0`. Здесь после знака **@** указывается, сколько места в байтах, по мнению компилятора Ассемблера, занимают параметры этой процедуры в стеке. В другом модуле такую процедуру придётся объявлять в виде:

```
extrn _MyProc@0:near
```

Это неудобно для программиста, и вводит в заблуждение, так как на самом деле у процедуры может быть, например, три параметра, просто это не видно по её заголовку, и компилятор Ассемблера об этом не знает. При связи только ассемблерных модулей это не так страшно (за всё отвечает программист!), но что будет, если, например, процедуру на Ассемблере будет вызывать программа на языке Free Pascal? Программа на Паскале будет вызывать процедуру (для случая с тремя параметрами по 4 байта каждый) под именем `_MyProc@12`, и, конечно, не найдёт, так как в Ассемблерном объектном модуле эта процедура имеет имя `_MyProc@0`. В том случае, когда возможно изменить описание внешней процедуры в программе на языке Free Pascal, для этой внешней процедуры можно задать явное имя, например:

```
procedure MyProc(X: byte; Y: Longint; Z: char); stdcall;
external name '_MyProc@0' { явно заданное внешнее имя }
```

Более сложным является случай, когда описание процедуры на языке высокого уровня программисту на Ассемблере недоступно для редактирования (чужая программа, есть только её объектный модуль), и он не может явно указать нужное ему внешнее имя. Тогда можно «обмануть» компилятор Ассемблера, описав это внешнее имя и процедуру так:

```
public MyProc@24;  внешнее имя MyProc@24, три параметра
MyProc@24:
MyProc proc
```

Теперь у процедуры как бы два имени, внутреннее `MyProc` и внешнее `_MyProc@12`. Здесь используется то обстоятельство, что процедуру на Ассемблере можно вызывать, указывая не имя процедуры, а обычную метку. Тогда вызов этой процедуры из другого модуля можно производить так:

```
   extrn MyProc@24:proc;  внешнее имя _MyProc@24
; тип proc не различает метки и имена процедур
   call MyProc@24;  внешнее имя _MyProc@2
```

С другой стороны, можно «честно» сообщить компилятору Ассемблера число параметров процедуры, например, так

```
public MyProc;  внешнее имя _MyProc@12
MyProc proc x: byte, y: dword, z: byte; 12 байт в стеке
```

В последнем случае, однако, как описывалось ранее, компилятор Ассемблера *автоматически* вставляет в текст процедуры команды пролога и эпилога, и программист должен чётко понимать, что при этом происходит.

Вместо объявления внешнего имени процедуры

```
extrn MyProc@12: near;  внешнее имя _MyProc@12
```

можно использовать объявление процедуры в виде её прототипа:

```
MyProc proto :byte,:dword,:byte
call MyProc;  будет внешнее имя _MyProc@12
```

Видно, что при описании внешнего имени посредством прототипа, как и в директиве **extrn**, кроме задания числа параметров, указывается также и *тип* каждого параметра.

Ещё одним способом связи по данным является использование так называемых общих (common) сегментов (секций). Такие одноимённые секции из разных модулей во время счёта располагаются в памяти с одного адреса, т.е. *накладываться* друг на друга. При несовпадении длин таких секций в разных модулях, память резервируется под самую большую из этих секций. В качестве примера рассмотрим описание обшей секции в разных модулях:

```
MyData segment common
N       equ 100000
A       dd  N dup (?)
Summa   dd  ?
D       db  ?
MyData ends
```

```
MyData segment common
M       equ 100000
B       dd  M dup (?)
Sum     dd  ?
MyData ends
```

Теперь секции данных с именем `MyData` будут *накладываться* друг на друга (в первом модуле секция данных немного длиннее, так что длина итоговой секции данных будет равна максимальной длине накладываемых секций).

Как видим, все имена в секциях являются *локальными*, однако из-за наложения секций данных друг на друга получается, что имя `A` является *синонимом* имени `B` (это имена одной и той же области памяти). Аналогично имена `Summa` и `Summ` также будут обозначать одну и ту же переменную. Можно сказать, что при таком наложении друг на друга секций разных модулей получаются *неявные* (т.е. *безымянные*) статические связи по данным. Вследствие этого можно резко сократить число *явных* связей по данным (то есть имен входных точек и внешних адресов). Надо, однако, заметить, что такой стиль модульного программирования является весьма опасным: часто достаточно ошибиться в расположении хотя бы одной переменной в накладываемых секциях, чтобы программа стала работать неправильно.

Очевидно, что общие секции могут содержать только не инициализированные переменные, т.к. неясно, какие значения будут у переменных после наложения секций **`.data`** друг на друга. Таким образом, это секции, аналогичные секции **`.data?`**.

Стиль программирования с общими (накладываемыми друг на друга) секциями данных широко используется, например, в языке Фортран, там такие секции называются общими (**common**) блоками данных для нескольких модулей. При этом часто случаются ошибки, вызванные плохим семантиче-

ским согласованием в расположении переменных в таких общих блоках памяти. Например, рассмотрите, что будет, если поменять в одной из накладываемых секций местами массив и переменную для хранения суммы этого массива (при компиляции никакой диагностики об этой *семантической* ошибке при этом, естественно, не будет).

Итак, явные статические связи по данным требуют согласования имён и типов входных точек и внешних адресов в разных модулях, однако, современные способы программирования рекомендуют как можно большее число связей между модулями по данным вообще делать не статическими, а динамическими. Другими словами, модули следует оформлять в виде наборов процедур и функций со стандартными соглашениями о связях. В этом случае, как Вам уже известно, связь между такими модулями по данным реализуется посредством механизма фактических и формальных параметров, при этом вообще отпадает необходимость «договариваться» между модулями об именах или взаимном расположении параметров в секциях.

Все написанные до сих пор в этой книге полные программы состояли из головного модуля на Ассемблере, который использовал макрокоманды, вызывающие процедуры и функции из служебных библиотек Windows. Теперь настало время написать программу, которая будет содержать уже два наших собственных модуля. В качестве примера напишем программу, которая вводит массив A *знаковых* целых чисел формата **dd** и выводит сумму всех элементов этого массива.

Сделаем следующее разбиение этой задачи на подзадачи-модули. Ввод массива и вывод результатов, а также выдачу диагностики об ошибках будет выполнять головной модуль нашей программы, а подсчёт суммы элементов массива будет выполнять процедура, расположенная во втором модуле программы. Для иллюстрации использования (статических) связей между модулями не будем делать процедуру суммирования полностью со стандартными соглашениями о связях, она будет использовать *внешние* имена для получения своих параметров, выдачи результата работы и диагностики об ошибке.

Текстовый файл, содержащий первый (головной) модуль нашей программы на Ассемблере пусть называется `p1.asm`, а файл с текстом второго модуля, содержащим процедуру суммирования массива `p2.asm`. Ниже приведён текст первого модуля.

```
; p1.asm
; Ввод массива, вызов внешней процедуры
include console.inc
.data
   public A,N;          входные точки
N  equ 10000
A  dd  N dup (?)
   extrn Summa:dword;  внешняя переменная
.code
Start:
   mov   ecx,N
   sub   ebx,ebx;       индекс массива
@@:inint A[4*ebx];      ввод массива A
   inc   ebx;           i:=i+1
   loop  @B
   extrn _Sum@0:near;   внешнее имя _Sum@0
   call  _Sum@0;        суммирование
   outintln Summa
; А теперь вызов с заведомой ошибкой
   mov   A,7FFFFFFFh; MaxLongint
   mov   A+4,1;         чтобы было переполнение
   call  _Sum@0
   outintln Summa;      сюда возврата не будет!
   exit;                хороший конец программы
   public Error;        входная точка
Error:
   outstrln "Переполнение !"
```

```
    exit  1;          аварийное окончание программы
    end Start;              это головной модуль
```

В головном модуле три входные точки с именами A, N и Error и два внешних имени: Sum, которое имеет тип процедуры, и Summa, которое имеет тип двойного слова. Работа программы будет подробно рассмотрена после написания текста второго модуля с именем p2.asm.

```
Comment * модуль p2.asm
    Суммирование массива, контроль ошибок.
    В конечном end не нужна метка Start
*
include console.inc
.data
    public Summa;   входная точка
Summa dd ?
    extrn N: abs;     внешняя константа
    extrn A: dword; внешний массив
.code
    public Sum;      внешнее имя _Sum@0
Sum proc
    push eax
    push ecx
    push ebx;         сохранение регистров
    xor  eax,eax;   eax:=0
    mov  ecx,N; N=E00000000
    xor  ebx,ebx;   индекс 1-го элемента
L:  add  eax,A[ebx]; A=E00000000
    jo   Voz;         при ошибке переполнения
    add  ebx,4;      i:=i+1
    loop L
    mov  Summa,eax
Voz: ;              OF=1 - плохой возврат
    pop  ebx
    pop  ecx
    pop  eax;         восстановление регистров
    jo   Err;         была обнаружена ошибка
    ret
Err:
    add  esp,4;       удаление адреса возврата
    extrn Error:near; внешнее имя _Error
    jmp  Error;       в первый модуль
Sum endp
    end
```

Второй модуль p2.asm не является головным, поэтому в его конечной директиве **end** нет метки входа. Модуль имеет три внешних имени A, N и Error и две входные точки с именами Sum и Summa. Разберем работу этой программы. После ввода массива A головной модуль вызывает внешнюю процедуру Sum. Это *статическая связь* модулей *по управлению*, адрес процедуры Sum будет известен головному модулю до начала счёта. Этот адрес будет иметь формат i32 на месте операнда Sum команды **call** Sum = **call** i32.

Между основной программой и процедурой установлены следующие (нестандартные) соглашения о связях. Суммируемый массив *знаковых* чисел расположен в секции данных головного модуля и имеет общедоступное имя A. Длина массива является общедоступной константой с именем N, также описанной в головном модуле. Вычисленная сумма массива помещается в общедоступную переменную с именем Summa, описанную во втором модуле. Это всё примеры *статических* связей между модулями по данным. Наша программа не содержит динамических связей по данным, в качестве примера такой связи можно привести передачу параметра *по ссылке* в процедуру другого модуля.

Действительно, адрес переданной по ссылке переменной становится известным вызванной процедуре только во время счёта программы, когда он передан ей основной программой (на регистре или в стеке).

В том случае, если при суммировании массива обнаружена ошибка (переполнение), второй модуль передает управление на общедоступную метку с именем `Error`, описанную в головном модуле. Остальные имена являются локальными в модулях, например, обратите внимание, что в обоих модулях используются две локальные метки с одинаковым именем `L`.

Обратите внимание, что на языке Ассемблера наша процедура суммирования массива имеет один вход (`public Summa`), но два выхода: **`ret`** – «нормальный» выход из процедуры (это *динамическая* связь по управлению) и «аварийный» выход `jmp Error` (это *статическая* связь по управлению). Большинство языков програмирования высокого уровня так делать не позволяют. Напомним, что по принципу структурного программирования любая часть алгоритма (условный оператор, цикл, процедура и т.д.) должна иметь один вход и один выход. Современные процессоры вообще не приветствуют выход из процедуры по команде **`jmp`**, а не **`ret`**, так как это препятствует работе так называемого механизма предсказания переходов конвейера и аппартного контроля правильного использования адресов возвратов (см. разд 14.2). У нас очень «нестандартный» пример  .

Продолжим рассмотрение работы нашей модульной программы. Получив управление, процедура `Sum` сохраняет в стеке используемые регистры (эта часть соглашения о связях у нас выполняется), и накапливает сумму всех элементов массива `A` в регистре `EAX`. При ошибке переполнения процедура восстанавливает значения регистров, удаляет из стека адрес возврата (`4` байта) и выполняет команду безусловного перехода на метку `Error` в головном модуле. В данном примере второй вызов процедуры `Sum` специально сделан так, чтобы вызвать ошибку переполнения. Заметим, что переход на внешнюю метку `Error` – это тоже статическая связь по управлению, так как адрес метки известен до начала счёта.[1] В то же время возврат из внешней процедуры по команде **`ret`** является *динамической* связью по управлению, так как конкретный адрес возврата в другой модуль будет помещен в стек только во время счёта программы. Другим примером динамической связи по управлению является передача, например, функции в качестве параметра в другую процедуру или функцию, в этом случае в стеке передается адрес этой функции.

Компилятор Ассемблера не в состоянии перевести каждый исходный модуль в готовый к счёту фрагмент программы на машинном языке, так как, во-первых, не может определить внешние адреса модуля, а, во-вторых, не знает будущего расположения секций модуля в памяти. Говорят, что компилятор Ассемблера переводит исходный модуль на специальный промежуточный язык, который называется *объектным языком*. Следовательно, компилятор Ассемблера преобразует входной модуль в объектный модуль. Полученный объектный модуль оформляется в виде файла, имя этого файла обычно совпадает с именем исходного файла на языке Ассемблера, но имеет другое расширение. Так, наши исходные файлы `p1.asm` и `p2.asm` будут переводиться (или, как чаще говорят, *компилироваться* или *транслироваться*) в объектные файлы с именами `p1.obj` и `p2.obj`.[2]

Рассмотрим теперь, чего не хватает в объектном модуле, чтобы быть полностью готовой к счёту частью программы на машинном языке. Например, команда из модуля `p1.asm`

```
mov A,7FFFFFFFh; MaxLongint
```

должна переводится в машинную команду пересылки формата `mov m32,i32`, однако значение адреса `m32` неизвестно компилятору Ассемблера, так как неизвестно будущее расположение секции **`.data`** в памяти во время счёта программы, известно только *смещение* этого адреса от начала секции. Такие адреса называются *перемещаемыми* (relocatable address), в листинге они обозначаются как `XXXXXXXX R`, где `XXXXXXXX` и есть (32-битное) смещение от начала секции. Перемещаемыми являются все адреса *локальных* переменных и *локальных* процедур. Таким образом, в объектном

---

[1] Современные процессоры не приветствуют выход из процедуры по команде **`jmp`**, а не **`ret`**, так как это препятствует работе так называемого механизма предсказания переходов конвейера и аппартного контроля правильного использования адресов возвратов.

[2] В разных языках и системах программирования объектные модули могут иметь разные расширения. Кроме `.obj` встречаются `.o` (в системах семейства Unix), `.ppt` (в языке Free Pascal) и другие.

модуле некоторые адреса остаются не полностью определенными. То же самое относится к адресам *внешних* констант, переменных и процедур, они тоже остаются незаполненными и помечаются в листинге как *внешние* адреса `00000000 E`, понятно, что для внешнего адреса компилятору Ассемблера неизвестно даже смещение в секции, где описано это имя. Очевидно, однако, что до начала счёта программы все такие адреса обязательно должны получить конкретные значения, иначе программа не сможет работать. Отметим, что на этапе компиляции «окончательные» адреса есть только в командах *относительных* переходов, где, как Вы уже должны знать, они равны знаковому *расстоянию* до точки перехода.

Для трансляции исходного модуля (например, `p1.asm`) в объектный модуль надо вызвать компилятор Ассемблера, используя команду

```
ml /c /coff /Fl p1.asm
```

Вообще говоря, после трансляции компилятор может сам вызывать служебную программу – редактор внешних связей. Об этой программе рассказывается далее, её мы будем вызывать отдельной командой, поэтому использована опция (иногда говорят, *ключ*) компилятора `/c` (только компиляция), который запрещает такой вызов. Опция `/coff` определяет формат объектного модуля (они бывают разные), наш формат называется `coff` (Common Object File Format). И, наконец, опция `/Fl` предписывает создавать протокол работы компилятора – листинг, по умолчанию это файл `p1.lst`.

Объектный модуль, получаемый компилятором Ассемблера, состоит из двух частей: *тела* модуля и *паспорта* (или заголовка – Heading) модуля. Тело модуля состоит из секций, в которых находятся команды и переменные (области памяти) нашего модуля, а паспорт содержит описание структуры объектного модуля. В этом описании должны содержаться следующие данные об объектном модуле.

- Сведения обо всех секциях модуля (имя секции, её длина, его спецификации).
- Сведения обо всех общедоступных (экспортируемых) именах модуля, заданных директивами **public**, с каждым таким именем связан его тип (**abs**, **byte**, **word**, **near** и т.д.) и адрес (входная точка `Start`) внутри какой-либо секции модуля (для константы типа **abs** это не адрес, а просто целое число – значение этой константы).
- Сведения об именах и типах всех внешних адресов модуля, заданных в директивах **extrn**.
- Сведения о местоположении всех перемещаемых адресов в секциях модуля, для каждого такого поля задано его относительное месторасположение в секции.
- Другая информация, необходимая для сборки программы из модулей.

На рис. 9.1 показано схематическое изображение объектных модулей `p1.obj` и `p2.obj`, независимо полученных компилятором Ассемблера, для каждого модуля изображены его секции, входные точки и внешние адреса. Вся эта информация содержится в паспортах объектных модулей (напомним, что для простоты изложения не принимается во внимание вызов процедур и функций из служебных библиотек). Заметим, что секция стека явно не описывалась, она автоматически задаётся Ассемблером для *головного* модуля. Заметим, что в головном модуле есть дополнительная входная точка `Start`.

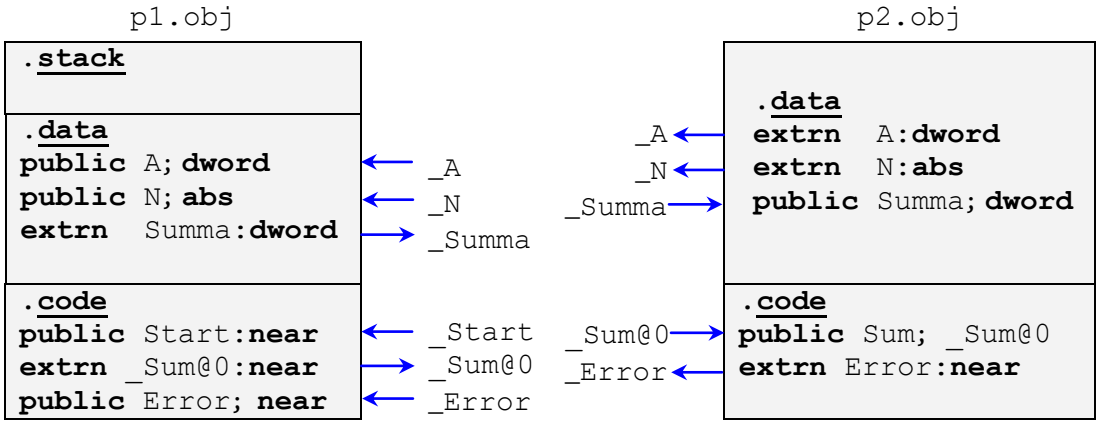

Рис. 9.1. Схематический вид двух объектных модулей с внешними адресами и входными точками.

Все имена исходного модуля, кроме внешних имен и имен входных точек, а также имен сегментов с параметром **common**, обычно заменяются в объектном модуле их численными значениями. Например, уже не важно, какое имя имела та или иная (локальная) переменная или метка программы, эта информация больше никому не понадобится (исключением являются так называемые «отладчики в терминах входного языка», но для Ассемблеров они используются редко, и эта тема не будет рассматриваться).

Заметим, что модульное построение сложных объектов широко используется и в других областях. Например, настольный компьютер состоит из устройств-модулей (системный блок, монитор, клавиатура, принтер, мышь и т.д.). Все эти модули проектируются и производятся по отдельности, а потом собираются вместе (обычно это делается с помощью проводов с разъёмами).

Рассмотрим теперь реализацию этой же задачи в многоязыковой системе модульного программирования. Головной модуль будет написан на языке Free Pascal, а вспомогательный модуль на Ассемблере. К сожалению, на Паскале при связи по управлению невозможно сделать входной точкой (**public**) обычную метку (мы программируем без меток ☺) и имя константы (есть, например, вещественные константы, которые нельзя передать в виде целочисленного адреса). Остаются только (не инициализированные) переменные, процедуры и функции.

Поэтому, чтобы сообщить об ошибке переполнения, модуль на Ассемблере будет присваивать внешней (булевской) переменной Паскаля с именем `Error` значение **true** или **false** в зависимости от того, обнаружена ли ошибка. Некоторые статические связи по данным заменены динамическими (адрес массива и его длина передаются, как обычно, через фактические параметры), остаётся статическая связь по данным через переменную `Error`. И, конечно, осталась статическая связь по управлению (имя функции `Sum`).

```
{p1.pas}
program p1(input,output);
uses Crt;
const N=10000;
type  Mas=array[1..N] of longint;
  var A: Mas; i: integer; Summa: longint;
      Error: Boolean public name '_Error';
    { внешнее имя _Error }[1]
function Sum(var X: Mas; N: longword):longint;
         stdcall; external name '_Sum@0';
{ иначе будет внешнее имя _Sum@8,
т.к. Паскаль видит два параметра }
{$Link p2.obj} { имя объектного модуля на Ассемблере }
{ итак, Sum в модуле p2.obj, внешнее имя _Sum@0:near }
begin ClrScr;
  for i:=1 to N do Read(A[i]);
  Summa:=Sum(A,N);
{ Этот оператор будет компилироваться так:
  push N
  push offset A
  call Sum
  mov  Summa,eax
}
  if Error then writeln('Переполнение !')
           else writeln('Сумма=',Summa)
end.
```

Второй модуль на Ассемблере:

---

[1] Здесь мы явно задали внешнее имя для переменной `Error`, так как сам Free Pascal, в отлисие от языка C, не "декорирует" свои внешние имена переменных (т.е. не приписывает впереди знак подчёркивания). В качестве альтернативы можно описать переменную в Паскале как `_Error: Boolean public;`

```
; p2.asm
Comment * модуль p2.asm
  Суммирование массива, контроль ошибок.
  В конечном end не нужна метка Start
  include console.inc не нужен, нет ввода/вывода
  Оттуда только режимы работы:
  .686 - модель процессора
  .model flat, stdcall; модель памяти и согл. о связях
  option casemap:none; различать мал. и боль. буквы
*
.686
.model flat,stdcall
option casemap:none
.code
   public Sum;           внешнее имя _Sum@0
   extrn  Error:byte;  ❶ внешнее имя _Error
Sum proc
   push ebp
   mov  ebp,esp
   push ecx
   push ebx
   mov  Error,-1;      -1=0FFh=true ❷
   mov  ebx,[ebp+8];   адрес A[1]
   mov  ecx,[ebp+12]; N
   xor  eax,eax;       Sum:=0
L: add  eax,[ebx];     Sum:=Sum+A[i]
   jo   Voz;           при переполнении
   add  ebx,4;         i:=i+1
   loop L
   mov  Error,0;       Error:=false
Voz:
   pop  ebx
   pop  ecx
   pop  ebp
   ret  8
Sum endp
   end
```

На Ассемблере мы описали переменную Паскаля `Error` как `byte` ❶, так как не различаем однобайтные переменные типов **`char`**, **`byte`**, **`boolean`** и **`shortint`**. Далее, как уже упоминалось, в Ассемблере в качестве константы **`true`** удобнее брать байт со значением ❷ `0FFh=-1`, так как тогда машинная команда **`not true`**=**`false`**, хотя многие языки (Free Pascal, C и др.), считают любое ненулевое значение как **`true`**.

Обратимся теперь к проблеме сборки полной программы из модулей. Как уже упоминалось, эту работу выполняет специальная системная программа, которая называется *редактором внешних связей*. Из этого названия хорошо видно одно из назначений этой программы – редактировать (в смысле устанавливать, настраивать) связи между внешними адресами и входными точками модулей. Рассмотрим схему работы редактора внешних связей на предыдущем примере.

9.2. Схема работы редактора внешних связей

Там, где есть модульность, возможно непонимание: сокрытие информации предполагает необходимость проверки связи.

Алан Перлис,
первый лауреат премии Тьюринга

Редактор – это специалист, который, плохо зная, что такое хорошо, хорошо знает, что такое плохо.

Евгений Сазонов, писатель, каких не было, нет и не надо

Целью работы редактора внешних связей (РВС) является построение из объектных модулей почти готовой к счёту программы, которая называется загрузочным модулем. При вызове РВС ему в качестве параметров передаются имена всех объектных модулей, необходимых для построения загрузочного модуля. Для нашего примера вызов редактора (его имя `link`) будет выглядеть так

```
link /subsystem:console [/out:p.exe] ^
p1.obj p2.obj
```

Первый параметр **`/subsystem:console`** предписывает создать консольную программу, использующую для ввода/вывода консоль (текстовое окно+устройства ввода). С другими параметрами можно создавать графические программы Windows, драйверы, программы для старых 16-битных ЭВМ и т.д. Параметры `p1.obj` и `p2.obj` – имена объектных модулей, а необязательный параметр `/out:p.exe` задаёт имя загрузочного модуля, который надо построить. При отсутствии этого параметра берётся имя первого объектного модуля (в нашем примере это будет `p1.exe`). Обратите внимание на символ `^` в конце первой строки команды `link`, это символ *продолжения* команды на следующей строке (просто так переносить команду Windows на следующую строку не разрешается). Первым указывается головной модуль, с которого начинается сборка, он содержит входную точку `Start`.

Метка `Start` по умолчанию описана как **`public`**, однако можно задать РВС другую входную точку программы, например, после вызова

```
link /subsystem:console ^
/entry:Sum@0 p1.obj p2.obj
```

счёт программы `p1.exe` начнётся с процедуры `Sum`, конечно, ничем хорошим это заведомо не закончится, т.к. в стеке нет адреса возврата для команды **`ret`** [iii] [см. сноску в конце главы].

В некоторых Ассемблерах (например, NASM) конечная директива **`end`** не требуется. Как следствие, невозможно *явно* задать входную точку программы, по умолчанию такой точкой считается только метка с именем `main`, и такая метка должна быть только в одном модуле программы.

К сожалению, так же сделано и в 64-битном Ассемблере MASM, можно сказать, что язык C здесь победил 🙁. Чтобы сделать входной нашу любимую метку `Start` надо поставить метку `Start`, описать её как **`public`** `Start` и задать параметр РВС **`/entry:`**`Start`.

Кроме явно заданных объектных модулей с именами `p1.obj` и `p2.obj`, РВС доступны также модули, собранные в *библиотеки* объектных модулей. Эти библиотеки бывают двух видов, со *статической* и с *динамической* загрузкой. Рассмотрим сначала библиотеки со статической загрузкой, каждая такая библиотека хранится в файле с расширением `.lib`, например `kernel32.lib`.

Большинство процедур и функции в статической библиотеке `.lib` содержат только маленькие объявления (прототипы) процедур и функций, которые «на самом деле» расположены в динамической библиотеке `.dll`, отсюда понятно, почему библиотеки `.lib` часто называют «библиотеки импорта». Таким образом можно «вытащить» нужные процедуры и функции из DLL-библиотеки и включить *их вызов* в исполняемый файл. Можно также отметить, что кроме процедур и функций DLL-библиотеки могут содержать и секции данных.

Все такие библиотеки, которые, по мнению программиста, могут понадобиться на этапе компиляции и редактирования внешних связей его программы, должны указываться в директивах `includelib` в начале программы на Ассемблере. Эти директивы для компилятора Ассемблера находятся во включаемом файле `console.inc`, который вставляется в начало модуля на Ассемблере по директиве **`include`** `console.inc`. Например, в этом файле есть директива:

```
includelib user32.lib
```

При желании, программист может создать свою библиотеку, включив туда нужные объектные модули, для этого надо вызвать служебную программу-библиотекаря, например:

```
lib /out:MyLib.lib a1.obj a2.obj
```

Теперь, если подключить эту библиотеку директивой

```
includelib MyLib.lib
```

то можно использовать в программе общедоступные имена, описанные в объектных файлах `a1.obj` и `a2.obj`.

Работа PBC включает в себя два этапа. На первом этапе происходит обработка секций, а на втором – собственно редактирование внешних связей и построение загрузочного модуля (загрузочные модули в нашей системе программирования имеют расширение `.exe`). Сначала будет рассмотрен первый этап.

В приведенном примере в двух модулях `p1.obj` и `p2.obj` имеется пять секций: три секции **`.stack`**, **`.data`** и **`.code`** в модуле `p1.obj` и две секции **`.data`** и **`.code`** в модуле `p2.obj`. Спрашивается, как эти секции будут размещаться в загрузочном модуле `p.exe` ? Здесь необходимо вспомнить, что на самом деле заголовки секций **`.data`**, **`.code`**, **`.data?`** и **`.const`** имеют полное описание в виде

```
_DATA segment para public 'DATA'
_TEXT segment para public 'CODE'
_BSS  segment para public 'BSS'
CONST segment para public 'CONST'
```

Здесь параметр **`public`** требует, чтобы одноимённые секции кода и данных из разных модулей склеивались (вторая секция размещалась вслед за предыдущей).

Вообще говоря, в нашем Ассемблере склеиваемые сегменты должны ещё принадлежать к одному и тому же *классу* сегментов, имя этого класса и задаётся строкой в апострофах. В рассмотренных нами примерах это требование выполняется, так как имена классов `'DATA'` и `'CODE'` в разных модулях совпадают. Заметим также, что одноименные секции (как с параметром **`public`**, так и без него) могут встречаться и внутри *одного* модуля. В этом случае такие секции тоже склеиваются с параметром выравнивания **`byte`**, но эту работу проводит не редактор внешних связей, а сам компилятор Ассемблера при построении объектного модуля.

Параметр **`para`** задаёт *выравнивание* начала секции в памяти по адресу, кратному 16 (так называемому *параграфу*), этот параметр предполагается по умолчанию. Если требуется склеить секции «впритык» друг другу, без возможного зазора между ними, то для второй секции задают параметр выравнивания **`byte`**. Для выравнивания на степень двойки можно применить параметр `align(n)`, где `n`=1,2,4,8,16,32,... При склеивании двух секций PBC меняет значение *перемещаемых* адресов во второй секции, увеличивая эти адреса на длину первой секции (с учётом выравнивания). Параметр выравнивания секций, заданный в программе, можно переопределить при работе PBC, задав у секций нужный параметр выравнивания, например:

```
link /subsystem:console ^
     /section:.text,,align=4 ^
     /section:.data,,align=1 ^
     p1.obj p2.obj
```

Обратите внимание на две запятые подряд, здесь у секций опущен параметр смены атрибутов доступа, он будет описан позже. Таким образом, для нашего примера в загрузочном модуле останется всего три секции. Вместо параметра **`public`**, можно задать и параметр **`common`**, что, как уже говорилось ранее, задаёт *наложение* одноимённых секций из разных модулей друг на друга, т.е. во время счёта все они будут начинаться с одного адреса оперативной памяти.

После обработки секций, начинается второй этап работы PBC – настройка всех внешних имён на соответствующие им входные точки в других модулях. Сначала нужные входные точки ищутся в объектных модулях, перечисленных в самой команде `link`, а затем в библиотеках объектных модулей `.lib`.[1] Таким образом, если программисту не понравилось, как реализована какая-либо библиотечная процедура, он может написать свою собственную процедуру «на замену». Как видим, по ана-

[1] Пути к файлам библиотек для PBC обычно задаются в командном файле, Вы можете увидеть такой путь в командном файле `makeit.bat` в директиве **`set`** `lib=..\lib;..\..\lib`.

логии в областями видимости в Паскале (где блок основной программы **program** вложен в блок модуля **unit**), здесь тоже входные точки в библиотеках располагаются как бы «во внешнем блоке» и становятся видимыми только тогда, когда их не закрывают одноимённые входные точки во «внутренних» модулях `.obj`.

Когда входная точка является процедурой, то секция кода этой процедуры включается в загрузочный модуль. Если секция процедуры имела стандартный вид **.code**, т.е. полное описание

```
_TEXT segment para public 'CODE'
```

то она склеивается с полученной ранее секцией кода, иначе включается в загрузочный модуль как отдельная секция.

Если поиск внешнего имени в объектных модулях и библиотеках оказывается безуспешным, то PBC фиксирует ошибку:

```
error LNK2001: unresolved external symbol
```

т.е. «неразрешённое внешнее имя» (термин «неразрешённое» имеет здесь тот же смысл, что и в выражении «неразрешённая, т.е. *нерешённая* проблема»). Другая ошибка PBC возникает, когда в одной области видимости находятся две и более одинаковых входных точки.

Эти две ошибки можно проигнорировать (сделать предупреждениями), если задать PBC опции **/force:unresolved** и **/force:multiple** соответственно. В первом случае программистом предполагается, что к отсутствующей внешней ссылке во время счёта обращений не будет, а во втором случае считается, что ему всё равно, к какой внешней ссылке будет выполняться обращение 🙂. В наших примерах из директории `_Examples` нашего пакета `masm 6.14.zip` вставлен параметр **/force:multiple**, что позводяет в два модуля на Ассемблере вставлять директиву **include** `console`, каждая из которых требует одни и те же внешние подпрограммы из системных библиотек.

К сожалению, большинство PBC (в том числе и для MASM не могут проверить для модулей на Ассемблере *соответствие типов* у внешнего имени и входной точки. Таким образом, например, внешнее имя-переменная размером в слово может быть связано с общедоступной переменной размером в байт или вообще с меткой. При невнимательном программировании это может привести к серьёзным ошибкам, которые будет трудно найти при отладке программы. Заметим, что в тех системах, которые уделяют большое внимание вопросам надежности программирования, такие проверки всегда делаются. Например, PBC Free Pascal считает несоответствие типов внешнего имени в одном модуле и соответствующей входной точки другого модуля фатальной ошибкой (если, конечно, оба модуля написаны на Паскале).

Скажем также, что в системе программирования с языком C, который Вами, тоже будет изучен, *автоматического* контроля соответствия типов внешнего имени и входной точки, как и модулях на Ассемблере, тоже нет. В языке C, однако, существуют некоторые приёмы, позволяющие уменьшить возможность ошибки несоответствия типов. Например, «во внешнем мире», как уже говорилось, имена изменяются (декорируются), чтобы включить в себя некоторую информацию о типе. Например, имя процедуры `Sum` заменялось на `_Sum@8`, чтобы подчеркнуть, что у процедуры два параметра. Можно пойти по этому пути и дальше и *потребовать* от программистов, чтобы каждое имя включало в себя *префиксы*, несущие информацию о типе этого имени. Это называ*ется венгерской нотацией*, предложенной программистом венгерского происхождения Чарльзом Симони (Simonyi Károly) очень давно, при написании одной из первых операционных систем MS-DOS для персональных ЭВМ. С тех пор эта нотация считается внутренним стандартом фирмы Microsoft, при этом имена имеют префиксы, записанные прописными буквами и отражающие тип имени, например:

```
uX  dw ?;           X: word; (u - unsigned int)
lY  dd ?;           Y: longword;
auA dw N dup (?);   A: array[0..N-1] of word;
szS db "Строка",0; S: Строка S оканчивающаяся нулём
```

Префиксы, естественно, сами не задают тип имени, а просто являются напоминанием программисту, какой тип должен быть у этого имени. Ясно, что это позволяет снизить вероятность несоответствия типов внешнего и входного имён, но полного контроля, конечно, не даёт. Как уже упоминалось ранее, контроль типов на Ассемблере можно обеспечить с помощью общих директив **extrn**,

вставляемых во все модули из одного включаемого по директиве **include** файла **.inc**. Аналогичные включаемые файлы с расширением .h используются и в языке C.

Продолжим описание работы PBC. Когда для некоторого внешнего имени найдена соответствующая входная точка, то устанавливается связь: адрес входной точки записывается в соответствующее поле внешнего имени. Например, для команды

```
call Sum; Формат i32
```

на место поля Sum запишется смещение начала процедуры суммирования в *склеенной* секции кода. Таким образом, поле из формата `00000000 E` превратилось в формат `i32 R` Аналогично, в поле с именем N второго операнда команды

```
mov ecx,N
```

запишется (окончательное) значение 10000:

```
mov ecx,10000
```

Итак, если для каждого внешнего имени найдена входная точка в другом объектном модуле или в библиотеке, то PBC нормально заканчивает свою работу, выдавая в качестве результата загрузочный модуль.

Теперь рассмотрим случай, когда некоторые из внешних адресов ведут в динамическую библиотеку объектных модулей с *динамической* загрузкой, такие библиотеки имеют расширение .dll (Dynamic(al) Link(ing) Libraries). Как и в случае библиотек со статической загрузкой .lib, это наборы процедур и функций.[1] Секции кода процедур и функций из таких динамических библиотек *не включаются* PBC в загрузочный модуль. Считается, что эти процедуры и функции как-то появятся в памяти уже на этапе счёта программы, этот вопрос будет рассмотрен немного позже.[2]

Таким образом, для динамически загружаемых процедур и функций PBC не знает даже смещений их адресов от начала кода программы. Для обеспечения доступа к таким процедурам и функциям PBC строит так называемую таблицу внешних (импортируемых) адресов (TBA) (IAT – Import Address Table).

Эта таблица, хотя и состоит из адресных *констант*, но обычно присоединяется PBC в конец секции кода, например, пусть в программе есть вызов процедур из DLL-библиотеки:

```
call DLLproc1; прямая адресация формат i32
. . .
call DLLproc2
. . .
call DLLproc1
```

PBC преобразует такие вызовы в команды

```
call ds:AddrDLLproc1; косвенная адресация формат m32
. . .
call ds:AddrDLLproc2
. . .
call ds:AddrDLLproc1
```

Эти команды делают косвенный переход по адресам, хранящимся в переменных с этими метками в таблице внешних адресов:

---

[1] Большинство процедур и функции в статической библиотеке `.lib` содержат только объявления (прототипы) процедур и функций, которые «на самом деле» расположены в динамической библиотеке `.dll`, отсюда понятно, почему библиотеки `.dll` часто называют «библиотеки импорта». Таким образом можно «вытащить» нужные процедуры и функции из DLL-библиотеки и включить *их вызов* в исполняемый файл. Можно также отметить, что кроме процедур и функций DLL-библиотеки могут содержать и секции данных.

[2] Впервые динамические библиотеки появились в операционной системе MULTICS в 1968 году, хотя их прототипы, были разработаны много раньше. Например, на серийной отечественной ЭВМ М-20 выпуска 1958 года работала ИС-2 (Интерпретирующая Система-2), которая осуществляла динамическое связывание и загрузку в оперативную память нужных (и выгрузку ненужных!) библиотечных процедур, написанных ещё прямо на машинном языке. Автором этой системы был профессор М.Р. Шура́ -Бура́, который работал на факультете ВМК МГУ заведующим кафедрой Системного программирования со дня его основания.

```
AddrDLLproc1 dd <адрес DLLproc1>
AddrDLLproc2 dd <адрес DLLproc2>
```

Итак, это *косвенный* переход к внешней процедуре, адрес которой во время счёта будет находиться в ТВА, как эти адреса появляются в этой таблице, рассматривается далее при описании работы загрузчика.

Загрузочный модуль, как и объектный, состоит из тела модуля и паспорта. Тело загрузочного модуля содержит все его секции, а в паспорте собраны необходимые для дальнейшей работы данные:

- информация обо всех секциях (длина и класс секции);
- информация обо всех ещё неопределённых полях в секциях модуля;
- информация о расположении входной точки программы (в нашем примере – метки `Start`);
- другая необходимая информация.

Заметьте, что параметры сегментов **`public`** и **`common`** уже не нужны. Вместе (друг за другом с учётом выравнивания) собираются все секции кода (с классом `'CODE'`) и все секции данных (с классами `'DATA'` и `'CONST'`). «Пустые» секции (с классами `'DATA?'` и `'STACK'`), как уже говорилось, в загрузочном модуле не хранятся, но полная информация о них в паспорте имеется. Метку `Start` можно рассматривать как единственную *входную точку* загрузочного модуля.

Обратите Ваше внимание на то, что теперь во всех секциях незаполненными остались только поля с перемещаемыми адресами, которые линейно зависят от местоположения секции в памяти во время счёта, и поля в таблице внешних адресов для DLL-библиотек. По умолчанию для Windows используется стандартное размещение секций в логической памяти. Секция кода загрузочного модуля размещается в памяти с адреса `400000h`. Этот адрес, называемый базой (Base), его можно сменить, задав PBC `link` параметр **`/base`**. Для программ на Ассемблере первые `1000h` байт занимают служебные данные, затем по адресу `401000h` стоит входная метка `Start`. DLL-библиотеки стандартно размещаются с адреса `1000000h`. Секция данных располагается сразу (со следующей страницы) за секцией кода. Секция стека по умолчанию занимает первый мегабайт логической памяти задачи. Можно задать другой размер стека, указав для PBC параметр **`/stack:`**<размер в байтах>.

Стандартное размещение секций позволяет убрать из паспорта загрузочного модуля всю информацию о перемещаемых адресах, так как все они уже настроены на конкретное месторасположение в памяти, это делается заданием опции PBC **`/fixed`**, загрузочный модуль `.exe` становится меньше.

Для каждого класса секций в паспорте загрузочного модуля прописаны атрибуты доступа. По умолчанию для класса `'CONST'` разрешено только чтение данных, для классов `'DATA'` и `'BSS'` чтение и запись, а класс `'TEXT'`[1] открыт на чтение данных и выполнение команд. Класс `'STACK'`, к сожалению, имеет полный доступ (по чтению, записи и выполнению команд).[2] Такие атрибуты должны обеспечивать большую надёжность программирования, но нарушают принципы фон Неймана однородности памяти и неразличимости команд и данных.

В отличие от многих языков высокого уровня, программист на Ассемблере может использовать команды, меняющие секцию кода, или организовать выполнение констант как команд. Предварительно, однако, необходимо указать PBC выставить у соответствующих секций необходимые атрибуты доступа (**`R`**`ead`**`W`**`rite`**`E`**`xecute`), это делается путём задания нужных параметров в команде `link`, например:

---

[1] В Ассемблере Windows класс `'TEXT'` соответствует секции (сегменту) команд **`.code`**, а класс `'BSS'` – секции **`.data?`**, однако в PBC, по аналогии с системой UNIX, секция **`.code`** называется **`.text`** (имеется в виду секция, содержащая *текст программы*), а «пустая» (неинициализированная) секция **`.data?`** называется `.bss` (Blank Static Storage).

[2] Открытие стека на выполнение обусловлено необходимостью эффективной работы эмуляторов, исполнителей jawa-кода, некоторых механизмов защиты и сложно вложенных друг в друга процедур и функций. Этим могут пользоваться вирусы, размещая в стеке вредоносный код. Заметим, что архитектура современных ЭВМ позволяет аппаратно закрывать стек на выполнение команд.

```
link  /subsystem:console ^
      /section:.text,W!E ^
      /section:.data,!WE ^
       p1.obj p2.obj
```

В этом примере секция кода `'TEXT' (.code)` дополнительно открывается на запись (**W**), но закрывается на выполнение (**!E**), а секция данных `'DATA' (.data)` закрывается на запись (**!W**), но открывается на выполнение (**E**) (поменять ролями команды и переменные, чтобы запутать взломщиков-хакеров ☺). Разумеется, при запуске такой программы на счёт ошибку вызовет самая первая (обычно помеченная меткой `Start:`) команда. Заметим, что программист может сменить атрибуты страниц *своей* программы и прямо во время счёта, вызвав системную функция с именем `VirtualProtect`.

Вот теперь все готово для запуска программы на счёт. Осталось только поместить нашу программу в оперативную память, задать значения ещё незаполненным полям и передать управление на начало программы (в нашем примере – на метку с именем `Start`). Почти всю эту работу делает служебная программа, которая называется загрузчиком.

9.3. Схема работы загрузчика

Опасно понимать новые вещи слишком быстро.

Джосая Уоррен.
«Истинная цивилизация»

При своем вызове загрузчик получает в качестве параметра имя файла (точнее, путь к файлу), в котором хранится загрузочный модуль. Работа начинается с чтения паспорта загрузочного модуля, затем порождается дескриптор новой задачи с полем сохранения (контекстом) `TSS` (см. разд. 6.11.1), дескриптор задачи помещается в глобальную таблицу дескрипторов `GDT`. Далее определяются дескрипторы для сегментов кода, данных и стека, они записываются в глобальную или локальную таблицу дескрипторов сегментов. Затем новой задаче выделяется своё адресное пространство, куда из загрузочного модуля читаются и размещаются в памяти все секции этого модуля (для «пустых» секций им просто выделяется память).

Для каждой секции, таким образом, определяется адрес её начала в оперативной памяти. Все секции одного класса (`'CODE'`, `'DATA'`, `'CONST'` и т.д.) идут подряд, а для разных классов по умолчанию производится выравнивания по адресам, кратным 4096 (размер *страницы* памяти).

Затем просматривается паспорт загрузочного модуля, и задаются значения всем полям с перемещаемыми адресами, так как теперь они известны. Как уже говорилось, этот шаг работы загрузчика можно пропустить, если для секций программы назначены стандартные начала в памяти (они же прописаны и в загрузочном модуле). В случае, когда у модуля есть таблица внешних адресов (ТВА) процедур и функций из DLL-библиотек, эти библиотеки ищутся в файловой системе. Поиск DLL-библиотеки производится сначала в каталоге, где находится выполнимый файл, затем в текущем каталоге, и, наконец, в каталоге операционной системы. Если DLL-библиотека найдена, то она тоже (к сожалению, целиком) загружается в память с помощью системной функции `LoadLibrary`, после чего ТВА заполняется конкретными адресами системных процедур.

Слова о том, что секции программы и библиотеки «загружается» или «читаются» в память ЭВМ из загрузочного модуля не надо понимать буквально. Секции программы и процедуры библиотеки не считываются из файловой системы (например, с диска) в оперативную память, а производится только *отображение* (проецирование) адресного пространства секций из загрузочного модуля и библиотек на логическое адресное пространство программы. «Настоящее» чтение страниц в память будет производится только по первому обращению к команде или данному в секции или в процедуре из библиотеки (это называется *загрузка по требованию*), подробно этот механизм изучается в курсе по операционным системам.

Таким образом, «целиком загруженная» DLL-библиотека занимает не *физическую* память, а только *логическое* адресное пространство задачи, куда, естественно, теперь нельзя отобразить другую библиотеку, программу или данные. Этот механизм называется отображением (содержимого) файла (file mapping) на виртуальную память задачи. Учтите, что несколько (независимых) задач могут делить один участок памяти с отображёнными на него из DLL-библиотеки процедурами и функциями, что позволяет экономить физическую память. Однако, если DLL-библиотека содержит секции

данных, которые изменяются одной из задач, то изменяемые секции *копируются* (дублируются) в память каждой задачи, позволяя этим задачам свободно изменять эти данные. Исключение составляют выполняемые программы и библиотеки, расположенные на сменных носителях (например, на флэшке) или в сети (например, в облаке), для них file mapping не производится и они на самом деле сразу загружаются в память.

Впервые механизм такого отображения файлов в память был реализован ещё в древней операционной системе Multics в 1968 году. В дальнейшем на основе Multics в 1972 году была разработана знаменитая операционная система Unix.

Такая загрузка DLL-библиотеки называется *статической* (т.е. перед началом счёта программы). Возможна и более сложная *динамическая* загрузка, когда программист сам определяет, когда и из какой DLL-библиотеки загрузить в память конкретную процедуру и получить её адрес. Этот вопрос мы рассматривать не будем. Когда статическая DLL-библиотека станет больше не нужна, её можно выгрузить из памяти, используя служебную функцию `FreeLibrary`.

На этом настройка программы на конкретное месторасположение в оперативной памяти заканчивается. Далее производится настройка контекста задачи в её `TSS`. В частности, анализируя паспорт, определяется адрес начала секции стека (ADDRSTACK) и начальный размер этой секции (например, 4096 байт), после чего в то место контекста задачи, где хранится регистр `ESP`, записывается значение `ADDRSTACK+4096`, так задаётся пустой стек программы. Аналогично в место хранения регистра `EIP` заносится входная точка программы (`Start`).[1] В те места контекста, где хранятся значения сегментных регистров (`CS`, `DS`, `ES` и `SS`) записываются селекторы сегментов кода, данных и стека (как уже говорилось, значения `DS`, `ES` и `SS` в плоской модели памяти совпадают).

Далее, для программы задаются подключения стандартного потока ввода (`stdin=input`), вывода (`stdout=output`) и ошибки (`stderr`), для настольных компьютеров и ноутбуков по умолчанию это, конечно, клавиатура и дисплей (для планшетов и смартфонов всё сложнее). После определения всех необходимых полей контекста задача ставится в очередь на выполнения к диспетчеру задач. Вызвав этот диспетчер (обычно одновременно нажав клавиши `Ctrl+Shift+Esc` или `Ctrl+Alt+Del`), можно увидеть в списке новую задачу (приложение), например `p.exe`. Далее начинается собственно выполнение загруженной программы, алгоритм работы диспетчера задач изучается в курсе по операционным системам.

Итак, на концептуальном уровне изучена *схема* разработки и выполнения модульной программы, эта схема включает в себя следующие этапы:

- Разбиение задачи на подзадачи.
- Реализация каждой такой подзадачи в виде модуля на некотором языке программирования, такой модуль принято называть исходным модулем.
- Компиляции каждого своего модуля в объектный модуль, при необходимости объединение нескольких объектных модулей в библиотеку.
- Сборка из объектных модулей загрузочного модуля с помощью РВС.
- Запуск загрузочного модуля на счёт с помощью загрузчика.

Обычно большая модульная программа состоит из одного головного модуля и нескольких DLL-библиотек с динамической загрузкой. Таким образом, в памяти находятся только те библиотеки, процедуры из которых вызваны в данном конкретном запуске программы. Заметим, что когда в загруженной библиотеке отпадёт необходимость, то, как уже упоминалось, её можно выгрузить (удалить) из памяти ЭВМ.

Сделаем теперь замечание к использованию одних и тех же программных модулей в *разных* программах при мультипрограммной работе ЭВМ. Из рассмотренной схемы счёта ясно, что загруженные в память секции кода DLL-библиотеки достаточно сделать *общими* (так как они закрыты на запись)

---

[1] В паспорте загрузочного модуля могут содержаться указания о том, что перед передачей управления на метку `Start` (а также после выхода из программы по макрокоманде `exit`), необходимо вызвать определённые системные функции обратного вызова (так называемые TLS-callback функции). Это своеобразные пролог и эпилог всей программы, обычно они автоматически включаются в загрузочный модуль компиляторами с языков высокого уровня. РВС с головным модулем на Ассемблере эти функции не задаёт. Таким образом, счёт программы, например, на языке C начинается вовсе не с первого оператора функции `main`  .

для всех находящихся в памяти программ. При этом, правда, все такие модули, использующиеся в разных программах, должны обладать особым свойством: быть так называемыми реентерабельными (или параллельно используемыми). Что это такое, будет изучаться в следующей главе.

На этом завершается наше по необходимости краткое знакомство со схемами выполнения модульных программ.

Вопросы и упражнения

Путь в тысячу ли начинается с первого шага.

Лао-Цзы, VI-V век до н.э.

1. Какую роль играют модули при разработке программного обеспечения ?
2. Что такое многоязыковая система программирования ?
3. Каковы преимущества и недостатки модульного программирования ?
4. Какой модуль называется головным и как он оформляется на Ассемблере?
5. Что такое связи между модулями на Ассемблере ?
6. Что такое связи по управлению и связи по данным ?
7. Что такое статические и динамические связи между модулями на Ассемблере ?
8. Что такое входные точки и внешние адреса модуля ?
9. Что такое объектный модуль и кто его создает ?
10. Какая информация хранится в паспорте объектного модуля ?
11. Из каких этапов состоит процесс сборки программы из объектных модулей ? Какие ошибки при этом могут быть выявлены ?
12. Как редактором внешних связей обрабатываются секции загрузочных модулей ?
13. Что такое общие области памяти модулей и как они могут использоваться при программировании ?
14. Какие поля в секциях загрузочного модуля остаются незаполненными ?
15. Опишите схему работы загрузчика.
16. Что такое таблица внешних адресов программы ?
17. Как к программе подключаются функции из DLL-библиотек ?

---

[i] Для продвинутых читателей. Отменим, что важным является и процесс *оптимизации* полученной программы. Ясно, что некоторые важные оптимизации могут быть применимы не к отдельному модулю, а только к программе целиком. Для этой цели современные оптимизирующие компиляторы могут производить *глобальную* оптимизацию программы (Whole Program Optimisations) уже на этапе редактирования внешних связей (LTCG – Link-Time Code Generation). Для этого, например, некоторые компиляторы языка C/C++ выдают объектные модули не на языке машины, а на специальном промежуточном языке CIL (C Inter-mediate Language). Внутри этих модулей находятся вызовы оптимизирующих функций компилятора, эти вызовы делает редактор внешних связей перед построением загрузочного модуля, когда уже установлены все связи между модулями и проведена настройка перемещаемых адресов. Глобальная оптимизация может значительно улучшить характеристики всей программы. Самый простой пример: теперь можно удалить "мёртвый" код процедур и функций, которые не вызываются ни из одного модуля (Dead Code Elimination).

Отметим также, что некоторые компиляторы могут получать для трансляции сразу *несколько* модулей, что тоже позволяет им делать глобальную оптимизацию. При этом, однако, надо понимать, что в этом случае при ошибке в одном из модулей при его перекомпиляции приходится заново анализировать и все ранее полученные объектные модули, что сводит на нет главное преимущество модульного программирования. Понятно, что такую глобальную оптимизацию надо производить только для *финальной* версии (релиза) программного продукта.

Современные оптимизирующие компиляторы, вероятно, являются самыми сложными программами, написанными людьми. Например, компилятор Clang для языка C++ содержит примерно 2 млн. строк исходного кода, а компилятор GCC уже более 5 млн. строк. Конечно, ядра операционных систем больше по размеру, например, для Linux это около 32 млн. строк, но это не одна программа, а целый программный комплекс.

[ii] Первоначально идея такого изменения имён возникла в языке C и мотивировалась повышением надёжности связей между модулями. Главная цель заключается в том, чтобы внести в имя больше информации об

объекте, который описывает это имя. Иногда декорирование приобретает «полный» характер, например, имя `MyProc` процедуры в языке Free Pascal

```
procedure MyProc(X: byte; Y: Longint; Z: char);
```

декорируется как `P$MODULAR_$$_MYPROC$BYTE$LONGINT$CHAR` 🙁. В теории программирования это называется сигнатурой (signature – подпись) имени. Показанная выше сигнатура имени полностью определяет его: это процедура (`P`) с именем `MyProc`, описанная в главной программа, имеет три параметра указанных типов. Так мы можем различать так называемые полиморфные подпрограммы, например, вот две сигнатуры функции `abs(x)` для целого и вещественного аргумента:

`F$MODULAR_$$_ABS$INTEGER$$INTEGER` и `F$MODULAR_$$_ABS$REAL$$REAL`

[iii] Для продвинутых читателей. При запуске программы на счёт переход на входную точку обычно производится не командой перехода **`jmp`**, а командой вызова функции **`call`**. Это легко понять, так как, например, в языке C входная точка программы описана как функция с именем **`main`** (так же сделано и в 64-битном Ассемблере MASM). Таким образом, в нашем Ассемблере можно выходить из программы командами

```
cld; При DF=1 AVOST
mov eax,<код возврата>
ret
```

При этом адрес возврата ведёт на системную программу (системный вызов) с именем `ExitProcess` (для ОС Windows) или `crt__exit` (с двумя подчёркиваниями, для ОС семейства Unix). Этот вызов завершает программу путём передачи управления в Диспетчер задач. При вызове для окончания программы макрокоманды **`exit`** она сама разворачивается в две команды

```
cld; Иначе AVOST
invoke ExitProcess,<код возврата>; или crt__exit для Unix
```

При этом, конечно, в стеке остаётся «грязь» в виде адреса возврата, но это уже не имеет значения, так как программа завершается.